Explorer's Guide to the ZX Spectrum and ZX 81

Kasper Boon

SMALL COMPUTER SERIES

Explorer's Guide to the ZX Spectrum and ZX81

SMALL COMPUTER SERIES

Consulting editors

Jan Wilmink
University of Twente

Max Bramer
Open University

Explorer's Guide to the ZX Spectrum and ZX81

KASPER BOON

ADDISON-WESLEY PUBLISHING COMPANY
London · Reading, Massachusetts · Menlo Park, California
Amsterdam · Don Mills, Ontario · Manila · Singapore · Sydney · Tokyo

Phototypesetting by Parkway Group, London and Abingdon

Printed in Finland by Otava, member of Finnprint.

British Library Cataloguing in Publication Data
Boon, Kasper L.
Explorer's Guide to the ZX Spectrum and ZX81.—(Small computer series)
1. Sinclair ZX81 (Computer)—Programming
2. Sinclair ZX Spectrum (Computer)—Programming
I. Title II. Series
001.64'2 QA76.8.S62/

ISBN 0-201-14638-X

ABCDEF 89876543

Contents

Introduction

The purpose of this book is to offer you a straightforward and complete introduction to the range of microcomputers from Sinclair – the new and exciting ZX Spectrum, the very popular ZX81 and their Timex counterparts the TS1000 and TS2000.

First a short survey is given of the main components and structure of the hardware and software. Naturally an important part of the book deals with the high-level language BASIC. We hope to convince you that it is not necessary to be a mathematician in order to use a computer. This strange idea is still held by many. However, many microcomputers including the Sinclair range are designed to be used by non-specialists as personal machines for games, education, 'housekeeping' and business management. In fact everybody can work with a computer! Our computers are the marvellous result of new technologies and offer you many possibilities.

We have written simply and clearly so that first-time users of a computer will understand and use their machines to more benefit. At the end of the book some interesting mathematical applications are given. If you are not interested in mathematics . . . just skip them.

Finally some appendices provide technical details of these computers. The final collection of programs is again intended both for mathematical applications and . . . let us say fun!

Kasper Boon

PART ONE

Learning about your computer and how to program in BASIC

CHAPTER 1

About information and information processing

Information

Our world is full of information. There are books with written words, drawings with lines and colours, spoken words and so on . . .

All these things deal with information. A computer also deals with information and therefore it seems sensible to take a closer look at the concept *information*.

It is in fact rather difficult to give an exact definition of what information is.

Information has something to do with messages, it tells us something and furthermore, in order to transmit information we use symbols like written characters or with respect to the spoken word we use certain sounds.

At the start of this century many investigators were searching for a proper definition of information.

One of the answers was given by the famous American Claude Shannon. He argued that with respect to *information* at least two basic aspects could be indicated:

the semantic aspect
the technical aspect

In order to illustrate the meaning of these terms we present an example.

Let us consider the next sentence:

FOLLOWING THE WORLD-BEATING SUCCESS OF THE SINCLAIR ZX80/ZX81 – OVER 400,000 SOLD SO FAR – COMES THE SINCLAIR ZX SPECTRUM

This sentence (from a well-known advertisement) consists of letters, numbers, spaces and some punctuation marks; in short it consists of characters.

The semantic aspect deals with the meaning of this sentence. Unfortunately the meaning of most sentences depends on the person who reads, or more generally receives the information. If an Eskimo

without any knowledge of the English language reads our sentence, he will probably shrug his shoulders to express, 'It's all Greek to me!'

If you are a computer fanatic and you already own a ZX81/TS1000 you probably interpreted the sentence as 'I already have a ZX81 and as 400,000 have already been sold I have probably made a good choice, but . . . now the ZX Spectrum is coming and . . .'. Shannon argued that the semantic aspect of information should be left to the philosophers, it does not provide any starting point for the technician.

The second aspect, the technical aspect, deals with the symbols that are used in a message. In our example symbols are used that belong to a particular alphabet. Shannon showed that this aspect, in contrast with the semantic aspect gives some very interesting starting points. For example, if we investigate messages we usually find that some symbols occur more often than others, e.g. the letter *o* occurs more often in our sentence than the letter *u*.

Shannon developed a theory by which it was possible to give a definition of 'the amount of information'. This amount can be quantified just as we can quantify length or weight, and in order to do so Shannon introduced the measure *bit*. We will encounter this measure later on.

In his theory the 'chance of occurrence' of each symbol plays an important role. The details of Shannon's theory are beyond the scope of this book, but the idea of a 'bit' of information can help us to understand the way our computer works. We shall explore the idea further in this chapter. At this point the most important thing for us to realize is that the symbols of an *alphabet* are always used to transmit information. An alphabet can generally be defined as a *collection* of symbols. Usually we are dealing with a fixed number of symbols. There are all kinds of alphabets. First there is the well-known Latin alphabet:

A B C . . . X Y Z

But if you look at the keyboard of a common typewriter it is obvious that the character set of a typewriter contains more symbols like

a b c . . . x y z
1 2 3 . . . 8 9 0

and the punctuation marks

, : ? ; ! etc.

If we look at the keyboard of the ZX81/TS1000 or ZX Spectrum we see even more symbols:

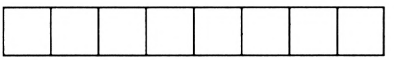
< >= + * etc.

Besides these examples we can point out other alphabets—Morse, Chinese or the alphabet of music symbols:

 etc.

A computer like the ZX81/TS1000 or ZX Spectrum works only with a character set of zero and one. If we deal with a particular set of characters, e.g. the alphabet of letters or the alphabet of music symbols and if such an alphabet consists of a *fixed* number of symbols, we can easily convert (or code) such an alphabet into a series of zeros and ones.

In order to store these series a computer contains a lot of so-called *registers*. Such a register can be compared with a drawer consisting of a row of cells. The ZX Spectrum and ZX81/TS1000 contain registers of 8 cells. For example the memory is a large collection of such 8-cell registers. A register is represented by the diagram:



Each cell can contain only a zero or a one, for example:

1	0	0	1	0	1	1	0

Now we arrive at a very crucial point. We wish to convert our particular alphabet or set of characters into a series of zeros and ones.

As an example we shall illustrate how some letters, numbers and musical tones are represented in the registers of the ZX Spectrum.

Some letters:

contents of register								meaning
0	1	0	0	0	0	0	1	A
0	1	0	0	0	0	1	0	B
0	1	0	0	0	0	1	1	C

Some numbers:

0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	2
0	0	0	0	0	0	1	1	3

Some tones:

0	0	0	0	0	1	0	1	F
0	0	0	0	0	1	1	0	F#
0	0	0	0	0	1	1	1	G

Of course the question arises 'How does the computer know if a certain series of zeros and ones applies to a character or to a number?'

The answer is that we can develop programs (we shall see later on what, exactly, a program is) that remember which registers store characters and which registers store numbers.

In the same way we can develop programs that remember which registers store series of zeros and ones that apply to musical tones and which registers store information that represents symbols to be displayed on the television screen.

Maybe this knowledge frightens you a little. It suggests that programming deals with the manipulation of zeros and ones. Basically this is true but we will see how programming languages like BASIC will actually mask those awkward zeros and ones. If you want to indicate in BASIC that a sentence like `"HOW ARE YOU?"` has to be displayed on the screen you can write down those letters in a `PRINT` statement:

```
PRINT "HOW ARE YOU?"
```

The message to be printed is placed between quotes . . . there are no zeros and ones in it! Your instruction will automatically be converted into this awkward zeros and ones language, but you will not be aware of it.

It is important to remember however that only a *fixed* collection of symbols can be handled. For this reason we can never display a completely smooth curve on the screen. Take a closer look at such a curve and you will see that it is composed of a fixed number of steps.

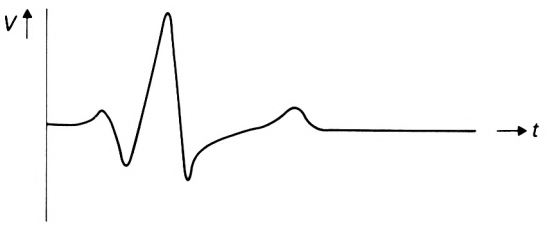
The need for a fixed number of steps or symbols to represent a curve forces us to convert our smooth input into a stepped, or digital, input. The device that performs this task is called an *analog to digital converter*.

For example, consider an electronic thermometer where 0°F corresponds to 0V, 1°F to 1V etc. Each temperature is represented by an analog voltage. In fact all the temperatures between 0°F and 1°F are possible and this means that an infinite number of values is possible. But we can only represent such analog values on our computer if we divide the interval between 0°F and 1°F into a fixed number of discrete

values. This is exactly the job of an analog to digital converter: it converts a smooth analog curve consisting of an infinite number of values into a series of discrete digital numbers. For example we can divide the interval between 0°F and 1°F into

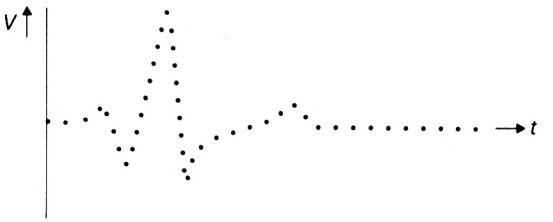
0.0 0.1 0.2 0.3 0.4 0.5
0.6 0.7 0.8 0.9 1.0

Let us consider an example.



The figure shows how an analog voltage changes as a function of time. This graph is the output of an ECG (electrocardiogram) recorder, and it represents the electrical activity of a cardiac (heart) muscle as it is measured on the surface of a body.

In order to represent and display this graph on a computer the signal must be converted into a series of discrete values. Each value is actually represented by a number. The result of this analog to digital conversion is shown in the following figure.



Information processing

We have discussed the meaning of information and how it is represented by symbols. You may have encountered the expression 'information processing'. What do we mean by this? In order to answer this question we will first look at an ordinary pocket calculator. Suppose you want to calculate

$56 \times 23 = ?$

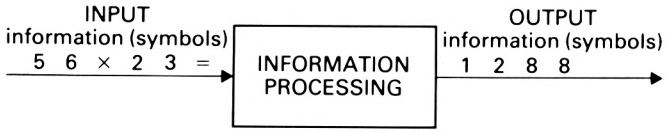
then you have to key in

5	6	×	2	3	=

and your calculator will display

1288

This means that your calculator has processed the *symbols* 5, 6, ×, 2, 3 and = in order to 'convert' these symbols to the *symbols* 1, 2, 8 and 8. Notice that the calculator processes the information (symbols) put into it, resulting in new information (symbols). We can represent this in a diagram:



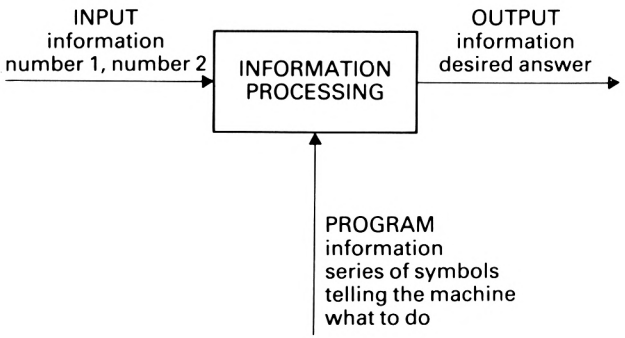
If we look at the figure it is obvious that 'information processing' is merely the conversion between input and output.

With a programmable calculator you also have the possibility of entering different programs. A program is a series of instructions telling the machine what to do.

A program to add two numbers on your calculator will look like the following:

1 Enter the first number and store it in the memory.
2 Enter the second number and add it to the contents of the memory.
3 Show the contents of the memory.

In this case our diagram becomes



Notice that the program itself consists of a list of symbols, hence information. This means that our programmable calculator is an information processing device and also that the way in which the

information has to be processed (or in other words how the symbols have to be converted) can be indicated by the user.

Both input data and program are temporarily stored in a memory. This may sound quite obvious but it took a genius, namely John von Neumann, to discover this 'stored program concept'. John von Neumann was continuously looking at the possible similarities between our brains and the computer. If you are interested in this matter, we strongly advise you to read his book *The computer and the brain*, Yale University Press.

It will be clear that the programmable calculator is only one step away from our computer. In fact the differences are few from a theoretical point of view. The main differences between a programmable calculator and a computer are as follows:

- The memory of a computer is usually much larger. This means that more data can be manipulated and longer programs can be executed.
- Modern computers like the ZX81/TS1000 and ZX Spectrum offer the possibility of manipulating all kinds of symbols. Besides numbers they can handle graphical symbols and characters. As we saw earlier, they simply represent the symbols as a series of zeros and ones and practically all kinds of manipulations can be performed with zeros and ones. It was the famous English mathematician A. M. Turing (1912–1954) who showed that with only a small number of amazingly simple manipulations, all kinds of complex calculations can be performed with ones and zeros.
- Modern computers offer a wide variety of devices that can be coupled to the computer. You can connect an ordinary cassette recorder to record programs and data on tape. Furthermore you can use a printer in order to obtain hard copies of programs and data. Other devices include floppy disc drives, colour graphics screens, interfaces to networks, teletext etc.

CHAPTER 2

Hardware and Software

Introduction

In the previous chapter we saw that a computer is an information processing machine. In order to build a computer two things are important:

(a) We need *components* to

feed data
store data
manipulate data
display data

These components exist physically; you can touch them. For this reason the collection of components is called the *hardware*.

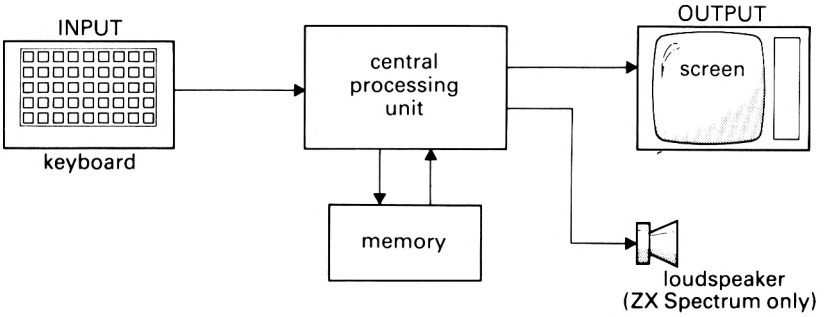
(b) We need a structure that enables us to put thoughts/instructions into the machine. A series of instructions that tells the machine what to do is called a program. When we use the word *software*, we refer to the collection of programs, i.e. those written to control the operations of the computer and those specially written by the user for a particular job.

In simple words: the computer is a machine that is constructed using many components (the hardware) and the thinking we can put into the machine is called the software. The next example will once more illustrate the difference between hardware and software. If you have paper and pencil you have components, hence hardware. The things you can write down are immaterial things – ideas and thoughts that exist in your brain. This immaterial stuff is the *soft*ware, meaning that you cannot touch it.

The rest of the chapter gives more details about the hardware and software. If no special reference is made, all data concerning the ZX 81 also applies to the TS1000.

Hardware

The following diagram illustrates the most important units of the computer.



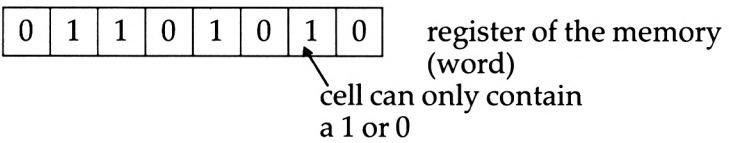
The keyboard is depicted on the left. Every key that is pressed is converted into a series of zeros and ones.

In the centre is a block called the 'central processing unit' (CPU). This unit can be considered as the brain of the computer. The CPU consists of the well-known Z80A microprocessor and it is this unit that actually processes the information. It only obeys instructions that are stored in the memory in a series of zeros and ones.

If we use high-level programming languages like BASIC, we need special programs (again consisting of elementary zeros and ones) to convert our BASIC instructions into these series of elementary instructions. More information on these elementary instructions (called machine code) is given in chapter 14.

For the time being all we need to know is that data is continuously transported to and from the memory. The 'memory' is also indicated in the diagram.

It can be compared with a chest of drawers consisting of a large number of separate drawers. Each drawer is a register of 8 cells. In each cell only a one or a zero can be stored.



An individual cell is usually called a *bit*. A more correct definition is:

> *A bit is information that can be stored in a cell that can only* contain a 1 or a 0

Since the cells of the register only contain 1 bit of information, it seems sensible to use the term bit for each storage cell. A row of 8 bits is usually called a *byte*. From the above figure it will be clear that one

register contains exactly 1 byte. Instead of memory register the term *word* may be used.

The storage capacity of a memory is usually indicated by 'number of bytes'. For instance the standard ZX Spectrum uses about 16000 bytes or in short 16Kbyte. We see that the letter K is used to represent the number 1000. For technical reasons however it is more practical to construct memories consisting of multiples of 1024 bytes. This is the reason why we used the expression 'the ZX Spectrum uses *about* 16000 bytes'; the real number is 16 × 1024 which is 384 bytes more than 16000 bytes!

A part of the memory is already filled with data – programs that are written by the manufacturer. For instance, this part contains the program that will convert your BASIC program into the elementary instructions that can be understood by the Z80A. Furthermore it contains programs necessary to transport data to devices like the cassette recorder. This part of the memory can never be used by the user as a 'free memory' because the information in it cannot be erased. This means that the computer can only *read* this part of the memory and therefore it is called *R*ead *O*nly *M*emory (ROM).

In addition to this ROM, the computer needs a memory in which data can be stored and retrieved whenever required. Each word of the memory has a label called the address. The CPU is directly able to store or write data *at any* address in this memory. This characteristic is emphasized by the name *R*andom *A*ccess *M*emory (RAM). If we speak of the storage capacity of a computer we are obviously referring to RAM. In the following section more details of the hardware of the ZX Spectrum and ZX81 (and therefore also of the TS1000) are given.

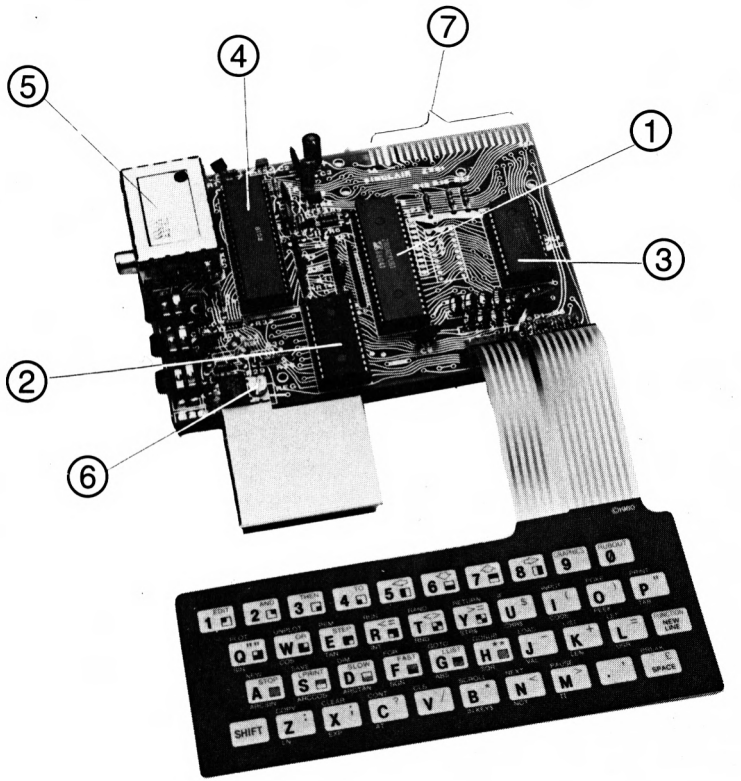
A glance inside

If we look inside the computer the *p*rinted *c*ircuit *b*oard (PCB) can be seen.

1 CPU – The central processing unit

The central processing unit consists of a Z80A microprocessor. This is the chip we actually see on the PCB. In the CPU the elementary instructions of series of zeros and ones are executed. The speed of processing is mainly determined by a timing device called the clock.

Clock/Frequency	*Type*
3.25 MHz	ZX81/TS1000
3.5 MHz	ZX Spectrum



Printed Circuit Board of ZX81

2 ROM – Read Only Memory

This memory contains programs already written for:

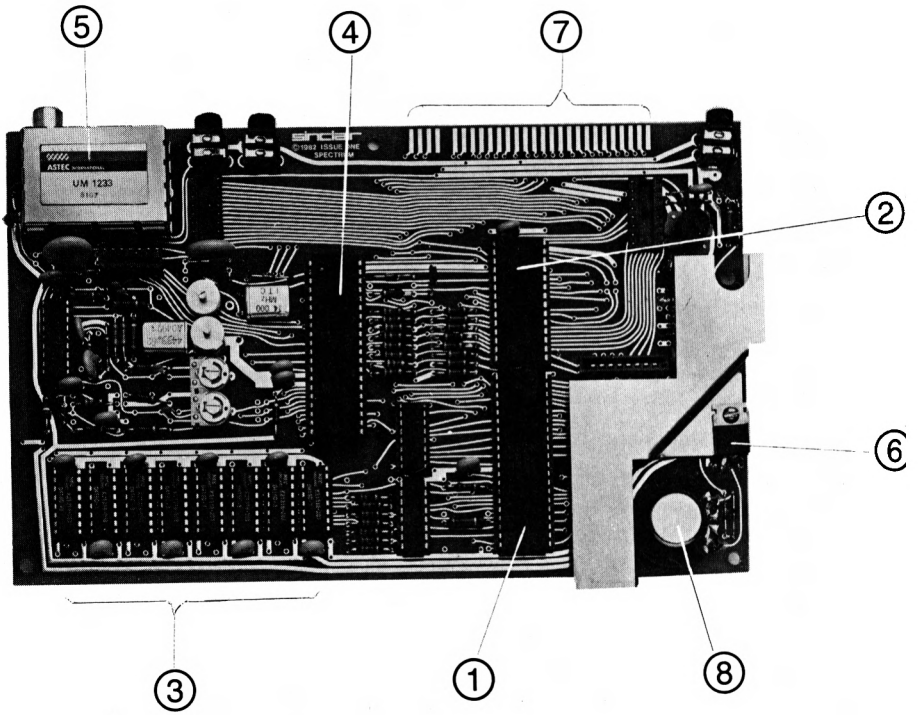
- Entering and executing BASIC programs.
  Each BASIC instruction is converted into elementary instructions that can be understood by the Z80A. This part of the ROM is called 'the BASIC interpreter'.

- Enabling communication with peripheral devices.
  Both computers can be coupled to an ordinary cassette recorder and a Sinclair printer. The ZX Spectrum can also be connected to a microfloppy and an RS232/network interface. RS232 is a 'label' to indicate a standardized way of coupling devices to the computer. For instance many printers can be coupled by means of an RS232 'connection'.

The ROM memory can be compared to an internal instruction manual for the computer itself. It tells the computer what to do under all circumstances. The collection of programs besides the BASIC interpre-

ter is usually referred to as an 'operating system'. Notice that the operating system is in fact a collection of programs.

ROM (*in* Kbyte)	*Type*
8	ZX81/TS1000
16	ZX Spectrum



Printed Circuit Board of ZX Spectrum

3 RAM – Random Access Memory

This is the memory in which data and the user's programs can be stored. The amount of RAM determines to a large extent the number of possible applications. The standard 1 Kbyte of the ZX81 is only interesting if we want to use this computer for an initial introduction. If the ZX Spectrum is used with microdrives (microfloppies) for professional applications, 48 Kbyte is recommended.

RAM (*in* Kbyte)	*Type*
1	ZX81
16	ZX81 with external RAM pack
16 or 48	ZX Spectrum

These figures apply to the possibilities offered by Sinclair. Many companies are offering add-ons (devices that can be coupled to the computer) and among these are expansion RAM packs.

4 ULA – Uncommitted Logic Array

This chip contains a large circuit to interface (connect) the CPU with the memory and, furthermore, to interface the memory with the TV.

5 TV Encoder

This unit converts the information, generated by the computer, to an appropriate signal for a television. The ZX Spectrum uses a PAL encoder in order to obtain appropriate signals for a colour TV. Both the ZX81 and ZX Spectrum are designed to work with UHF channel 36 in Europe. In the USA the ZX81/TS1000 and ZX Spectrum are designed to work on VHF channel 2 or 3.

6 Voltage regulator

This unit contains the voltage regulator. The ZX81/TS1000 requires a power supply of 700 mA at 9 V DC (unregulated). The voltage regulator will control this unregulated voltage (a ripple of 1–2 V, 100 Hz can occur) providing a constant voltage of 5 V. The expansion port (see **7**) provides a 5 V supply from which a maximum of 200 mA can be drawn.

The ZX Spectrum requires a power supply of 1400 mA at 9 V DC (unregulated). Several voltages are derived from it. First 5 V is required for most of the logic circuits. The expansion port also provides 5 V and a maximum of 100 mA can be drawn from it. Furthermore there is a 12 V output which can supply 30 mA and a −5 V output which only supplies a few mA. Finally there is a −12 V output which is in fact a high frequency square wave that can be used to obtain −12 V, e.g. by a simple diode-network.

7 Expansion port

This unit consists of a connector that can be used to attach all kinds of peripheral devices to the computer. It has the full data, address and control 'busses' from the Z80A (an explanation of these 'strange' words is given in chapter 15). Furthermore it has several voltage supplies (see **6**). With the ZX Spectrum, BASIC commands (**IN** and **OUT**) are provided to directly control the expansion port. The expansion port of the ZX Spectrum is different from that of the ZX81/TS1000. This means that most of the add-ons cannot be interchanged. Only the Sinclair printer is interchangeable.

8 Loudspeaker

The ZX Spectrum has an internal loudspeaker that can be operated over a range of more than 10 octaves. An external amplifier can be used by means of the jack sockets at the rear of the computer (sockets MIC and EAR).

A bird's eye view

If we look at the computer from the outside, the following hardware aspects can be considered.

Dimensions

The dimensions are governed by Clive Sinclair's proved philosophy that 'small is beautiful'. It shows that Sinclair has scored a hit.

Dimensions	*ZX81/TS1000*	*ZX Spectrum*
width (mm)	174	233
depth (mm)	218	144
height (mm)	38	30
weight (g)	300	600

Keyboard

Both computers use 40 keys. The ZX81/TS1000 uses a touch-sensitive membrane with which only capitals, 20 graphics symbols and 54 inverse video characters can be entered. Inverse video means that the colours of 'paper and ink' are exchanged with respect to the normal representation. The ZX Spectrum has a keyboard with 40 moveable keys with upper and lower case (and a capital lock feature). We can enter 16 graphics characters, 22 colour control modes and 21 user definable graphics (symbols that can be indicated by the user). The keyboard of the ZX81/TS1000 has received much criticism. This is the reason why many companies offer professional keyboards for this machine.

Screen

The screen is divided into 22 lines plus 2 lines for special user information (special messages, lines that are currently being entered etc.). Each line is divided into 32 parts and each part can display exactly one character. In order to display 'high resolution pictures' each part is once more divided into *pixels*. As a matter of fact a pixel is the smallest part of the screen that can be indicated. The screen of the ZX81/TS1000 consisting of 22 lines (each containing 32 parts) is divided into 44×64 pixels. The screen of the ZX Spectrum consists of 24 lines each containing 32 parts and this screen is divided into 192×256 pixels.

If we compare the resolution of the ZX Spectrum with that of the ZX81/TS1000 we find that it is 4 times better. The ZX81/TS1000 can operate in two modes – FAST and NORMAL. During the FAST mode no pictures will be displayed during computations. NORMAL mode (25% of the FAST speed) allows flicker-free animated pictures to be

continuously displayed. The ZX Spectrum uses a different hardware organization which enables it to display pictures during computations. The ZX Spectrum only works in the FAST mode.

The ZX microdrive

Sinclair is offering a microfloppy for the ZX Spectrum. A microfloppy contains a small disc on which programs and data can be stored. The big difference between a cassette recorder and a floppy disc is the time needed to search for the data indicated. This time is usually referred to as access time. With our microfloppy an average access time of 3.5 seconds is all that is necessary whereas the cassette recorder can search for minutes.

Besides the devices just mentioned a large number of others are offered by advertisers in most of the personal/hobby computer magazines.

Software

Software deals with the instructions we can put into the computer. First, let us consider the kinds of program that a user may wish to develop. Suppose you want to add two numbers, for example 7638 and 15379. It would be nice to instruct our computer to add these numbers by typing

```
add 7638 and 15379 and
print the result
```

Unfortunately this will not work. The computer will only listen to you provided you speak its language. The collection of possible sentences and the way these sentences can be used form a *programming language*.

The first computers could only be programmed in machine code instructions. In chapter 14 we shall present an introduction to machine code programming. The need for a high-level language, closer to English and easier for the user to read and understand, was soon recognized. Machine code programs are rather difficult to read and they are also relatively difficult to develop. Moreover, errors are easily made and it is often very difficult to find the bugs. Our computer uses BASIC as a high-level programming language. As a matter of fact most microcomputers use BASIC and there are good reasons for this. BASIC is a language that can be easily mastered. Even children can learn this language! A large part of this book deals with BASIC so you can find out for yourself how easy (or difficult) it is. Most jobs can be programmed perfectly well in BASIC and this should certainly set your mind at rest.

Now we will pay some attention to the programs stored in ROM. Besides the machine code program that translates our BASIC programs into machine codes, the ROM also contains programs to control

communication between the computer and devices that are coupled to it (like the cassette recorder). This may seem odd: we know that communication between the computer and its 'peripheral' devices is controlled by electronic circuits. However, these circuits are themselves controlled by machine code programs.

Nearly every action of the computer, like the entering of keystrokes, and the displaying of pictures, is controlled by machine code programs. All these programs together form a sort of system that controls the operations of your computer system. This is the reason why this collection of programs is usually called the *Operating System*.

Finally we will take a look at the enormous amount of software that is offered by advertisers in all sorts of periodicals. It is really unnecessary to become a programmer at present since so many programs can be bought. Indeed the number of 'companies' that are challenged by what is called ZX computing is overwhelming. Even ICL is in the market.

It is astonishing to observe that programs are developed for nearly every application, for example programs for text processing on a ZX81. But don't let yourself be fooled, these computers are not intended for 'professional' applications in text processing and business administration. The ZX Spectrum however forms an exception since the microdrives make this computer an exceptionally suitable tool for complicated business administrations. The most appropriate and important use of our computers is that they give a very good opportunity to get acquainted with computing. Personal computing is certainly becoming one of the most popular hobbies in the world!

CHAPTER 3

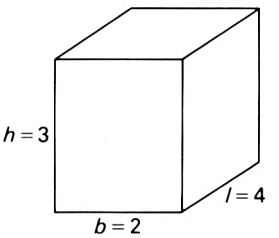
And now we can really start

In this chapter we shall describe a short BASIC program. It is a very simple program because we just want to illustrate some elementary facts about the structure of BASIC.

BASIC is a high-level programming language; the term 'high-level programming language' indicates that the programmer will not be hampered by those characteristics typical of a digital computer – that it only processes zeros and ones. BASIC allows a programmer to focus his total attention on the problem to be solved.

The LET statement

Our first problem will be to compute the volume of a box, such as the one drawn in the figure below.



In this figure the dimensions are also indicated. The volume of a box is given by the well known formula

$$v = h \times b \times l$$

In order to convert this expression into a BASIC sentence we merely have to copy the formula together with a specific instruction. The correct BASIC sentence is

```
LET v = h * b * l
```

The resemblance to the formula is evident, only the word LET is placed in front of the formula. The meaning of LET is to indicate that a value has to be assigned to v – the value that is found on computing the

expression h * b * l. This is important because it underlines the fact that some computations have to be done before the value of v is found.

Notice that an asterisk * is used instead of the usual multiplication sign ×. The reason for this is simply to avoid confusion with the letter x. For the same reason a line is drawn through the numeral Ø (look at your keyboard). This ensures that the difference between zero and the letter O is always evident.

Let us return to the expression given above. We first have to *assign* values to h, b and l because otherwise the computer cannot compute the value of v. This is easily indicated in BASIC by three 'sentences':

```
LET h = 3
LET b = 2
LET l = 4
```

The sentence LET h = 3 means 'let h be 3' or 'assign 3 to h'.

At this point it is interesting to take a closer look at what is going on inside the computer. If a computer obeys an instruction like

```
LET h = 3
```

it will first reserve a part of the memory and label it with the symbolic *name* h. Then the computer will store the value 3 (of course represented by a series of zeros and ones) in this part of the memory labeled h. In fact it can store any number we choose in this part, which means that the contents may vary. For this reason, we speak about 'the *variable* with the label h' or in short 'the variable h'.

Variables

In order to label a part of the memory or, in other words, to assign a name to a variable, certain rules have to be considered.

> The name of a variable must always begin with a letter and furthermore only letters and digits are accepted. Spaces are ignored.

A few examples follow:

Variable name	
CONTENTS	correct
X,1	wrong: no , is allowed
A4	correct
4A	wrong: name starts with a digit

With the ZX Spectrum you can use both capital and lower-case letters but remember the computer does not distinguish between them. For example

```
CONTENTS
```
and `CONtents`

refer to the same name.

The ZX81/TS1000 only uses capitals. With this computer our first sentence is

```
LET V = H * B * L
```

So remember:

> With the ZX81/TS1000 you can only use capitals. In this book lower case letters are often used. If you are a ZX81/TS1000 owner don't bother about it just take them as capitals!

The `PRINT` statement

Up to now we have given our program 4 sentences but no expression to indicate that a value has to be displayed. In fact we only want to see the value of `V` and this is simply indicated in BASIC by

```
PRINT V
```

The computer understands each of the expressions given above, but in order to obtain a sensible answer we must indicate *in which sequence* these sentences have to be executed. First it is clear that values have to be assigned to the variables `H`, `B` and `L`. After this `V` can be computed and finally the result has to be displayed.

Line numbers

In order to indicate in which sequence the instructions have to be obeyed we place *line numbers* at the beginning of each sentence.

We may only use positive whole numbers (from 1 to 9999). The sentences now will be executed in the order indicated by the line numbers, that is from low to high numbers. For example, we could use the following line numbers:

```
1 LET h = 3
2 LET b = 2
3 LET l = 4
4 LET v = h * b * l
5 PRINT v
```

This is a correct BASIC program and can be directly executed on our computer. Usually, however, the line numbers `10`, `20`, `30`, etc. are used. This numbering introduces the possibility of inserting other

lines as we shall see later on. Therefore our final program is

```
10  LET h = 3
20  LET b = 2
30  LET l = 4
40  LET v = h * b * l
50  PRINT v
```

(remember with the ZX81/TS1000 only capitals are used). From now on we shall speak of lines rather than of sentences and furthermore we will indicate each line by its line number in our text. Every line expresses a task the computer has to carry out. Such tasks are called *instructions* or *statements*.

Switching on

Now the time has arrived to see how such a program can be executed on our computer. To switch on the ZX Spectrum or the ZX81/TS1000 the following connections have to be made:

(a) Connect your television (black and white or colour) by means of the cable provided – use the UHF socket. Tune the television to approximately channel 36, or in the USA VHF channel 2 or 3.

(b) Connect the power supply to the socket marked '9VDC'.

After the power supply is connected, you can adjust the tuning of the television in order to obtain a clear picture. Notice that our computer has no power (on/off) switch. If the power supply is connected, the computer is 'on', otherwise it is 'off'. The television shows the letter K in a black square:

(ZX81/TS1000)

or © 1982 Sinclair Research Ltd (ZX Spectrum)

If you own a ZX Spectrum you should press

and the screen now also shows

(flashing)

We may now enter our program. Look at the keyboard of your computer – undoubtedly the large number of symbols will frighten you a little. However you will find that things are in fact not so difficult as they seem.

Entering the program

In order to enter the first line number press keys

[1] [0]

Each digit is immediately displayed at the bottom of the screen and the black square with the letter K in it automatically moves to the right. This black square is usually called the *cursor* and the letter in it indicates the *mode*. The letter K indicates *K*eyword-mode and we shall immediately see why.

At this point we must enter the word LET. Just press the letter L and the word LET is displayed on the screen. Notice that the word LET is also written on the key with the letter L. We should realize that the word LET is a *reserved word* (a Keyword) in BASIC, just like the word PRINT. Reserved words have a special meaning.

Our computer knows that each word after a line number is always a reserved word, and therefore no confusion can occur. Notice that the letter in the cursor is now changed to

L

This indicates that the computer is expecting a *L*etter or more generally a character. Now we enter

h = 3

by pressing

[H] = [3]

The symbol = is entered by pressing

[SYMBOL SHIFT] and [L] (ZX Spectrum)

or [SHIFT] and [L] (ZX81/TS1000)

If this line does not contain any errors you can now press

[ENTER] (ZX Spectrum and TS1000)

or [NEWLINE] (ZX81)*

The line moves immediately to the top of the screen and the computer is ready to accept another line. Notice also that a cursor with an arrow appears at the top of the screen directly after the line number.

* American versions of the ZX81 also use ENTER.

It is possible that you inadvertently made a mistake – do not worry. There are a number of ways to make corrections. A very rough way is to disconnect the power supply and by connecting it again to 'reset' our computer to its initial state. This is not to be recommended if you make an error in the very last line! Another way of correcting is offered by the erase key. This key is activated by

CAPS SHIFT + DELETE 0 (ZX Spectrum and TS1000)

or

SHIFT + RUBOUT 0 (ZX81)*

From now on we shall refer to this key as the

ERASE key

Pressing the **ERASE** key results in erasing the symbol just to the left of the cursor. The ZX Spectrum has a special feature: if we hold the **ERASE** key down further symbols will be erased. It is as if we are repeatedly pressing the **ERASE** key. This effect is described by the term 'auto repeat'. In fact all the keys of the Spectrum include this feature.

If a serious mistake is made the computer will not accept the line and an *error-cursor* appears:

? (ZX Spectrum)

or S (ZX81/TS1000)

We must then correct this line first, e.g. by the **ERASE** key (more information is given on page 28).

After entering the first line, we can easily enter lines **20** and **30**. Line **40** shows the expression

```
40 LET v = h * b * l
```

The multiplication sign * is entered by

SYMBOL SHIFT + B* (ZX Spectrum)

or SHIFT + B* (ZX81/TS1000)

Note that the **SHIFT** key has to be used each time if the symbol to be entered is indicated on the *upper part* of the key.

Line **50** contains the keyword **PRINT**. First the line number **50** is entered and then when **P** has been pressed **PRINT** is automatically displayed.

* American versions use SHIFT + DELETE

If the last line is moved upwards, our program is completely entered. This means that the complete program has been stored in the memory. From now on we can give *commands* to the computer, in order to do something with that stored program. For example:

LIST command – show the program once more
RUN command – execute the program
SAVE command – store the program on a cassette, provided that the recorder is connected
NEW command – erase the program in order to enter a new program

Of course we are anxious to see if the program really 'works' and so we give the **RUN** command simply by pressing

| R |

followed by

| ENTER | (ZX Spectrum and TS1000)

or | NEWLINE | (ZX81)

Notice that each command is followed by **ENTER** (or **NEWLINE**). From now on we will not mention this again, so please do not forget it! Immediately the screen displays the correct answer to our problem:

```
24
```

In addition the screen also displays a message to inform you at which line the computer stopped and that everything is OK! (all right). This is indicated by

```
0 OK,50:1
```
(ZX Spectrum)

or `0/50` (ZX81/TS1000)

The first number, **0**, indicates the message

OK . . . I am ready to obey a new command

The second number, **50**, indicates the line number at which the computer stopped. The ZX Spectrum allows more statements to be written on a single line and this is the reason why we also see **:1**. The number **1** indicates the statement number on the line where the computer stopped.

Notice that the program remains unchanged in the memory. As proof we may use the **LIST** command by pressing

| K | (and **ENTER** or **NEWLINE**)

(Notice that LIST is indicated on this key.) We may also run it again just by giving the RUN command once more:

Finally if you are tired of this program you can erase it completely by giving the NEW command

CHAPTER 4

Changing a program

In practice we often want to change a program. The computer offers many ways of doing this. There are basically three types of changes we may wish to make.

- to change a particular line
- to insert a new line
- to delete a particular line

In order to illustrate these possibilities we again look at our first program:

```
10  LET h = 3
20  LET b = 2
30  LET l = 4
40  LET v = h * b * l
50  PRINT v
```

To change a line: `LIST`, `EDIT` and cursor control

If we want to change any line of a program, there are two possible ways of doing this. The first way is the 'standard' method:

> type the line number of the line to be changed and enter the correct expression

For example if we want to change line **20** into

```
20  LET b = 4
```

just type in

```
20  LET b = 4
```

followed by **ENTER** (or **NEWLINE**)

Our screen then shows

```
10 LET h = 3
20 LET b = 4
30 LET l = 4
40 LET v = h * b * l
50 PRINT v
```

A second way of changing a line is based on the EDIT command. If you look at your program on the screen, a cursor with an arrow, >, can also be seen. This cursor can be moved:

↓ (above key 6 – moves cursor downwards)
↑ (above key 7 – moves cursor upwards)

Do not forget to use the SHIFT keys as you did before (the symbols ↓ and ↑ are indicated on the upper part of the keys). Now move the cursor to the line to be changed and give the EDIT command by pressing on 1 together with SHIFT. The line to be changed will appear at the bottom of the screen. As an example we will change line 20 back to the original by using the EDIT command. First move the cursor to line 20 and the display shows

```
10 LET h = 3
20>LET b = 4
30 LET l = 4
40 LET v = h * b * l
50 PRINT v
```

We now give the EDIT command by pressing 1 (with SHIFT key). After pressing ENTER (or NEWLINE) line 20 appears also at the bottom of the screen:

```
20 [K] LET b = 4
```

Notice again a cursor appears; this can also be moved.

→ (above key 8 – moves cursor to the right)
← (above key 5 – moves cursor to the left)

Experiment with these keys; 'hands-on' experience is the most important practical experience you can have. For instance if you press key 8 (with SHIFT) you will see

```
20 LET [L] b = 4
```

Finally move the cursor again just right of the character to be changed

```
20 LET b = 4 [L]
```

Here the character can be erased (key 0 with SHIFT) to give

```
20 LET b = [L]
```

and the correct digit can be entered. Now the display shows

```
20 LET b = 2 [L]
```

Finally the procedure is terminated by pressing **ENTER** (or **NEWLINE**).

The procedure just described has to be used if we make an error during the entering of a line. If an error is made the line will not move upwards to the part of the program already displayed on the upper part of the screen. In this case as we said before an error cursor will appear

```
S                                    (ZX81/TS1000)
?                                    (ZX Spectrum)
```

at the place where the error is found. First the error must be corrected and only then can we proceed with the programming.

The cursors ↑ and ↓ can also be used if a program is too long to be listed entirely on the screen. We can move the program upwards or downwards just by pressing the appropriate cursor control keys.

The ZX Spectrum shows the message

```
scroll?
```

if the program cannot be completely displayed. If you press **Y** or any other key except **N** the computer will show you the next part. The part to be displayed can also be requested by entering a line number with a **LIST** command. For example

```
LIST 45
```

will show you the program starting at line **45**.

To insert a new line

When we wish to insert a new line the procedure is quite simple:

> Select a line number that lies between the numbers of the two lines between which the new line has to be inserted, then key in the new line.

For example, if we want to insert a line between lines **10** and **20** in order to examine the value of **h**, we type

```
15 PRINT h
```

followed by **ENTER** (or **NEWLINE**).

The screen will show

```
10 LET h = 3
15 PRINT h
```

```
20  LET b = 4
30  LET l = 4
40  LET v = h * b * l
```

(Note: if line numbers are used that are greater than the last line number we are obviously extending the program.)

To delete a line

To delete a line the procedure is again simple:

Just enter the line number of the line to be erased.

For example if we want to delete line 15 we type

```
15
```

followed by **ENTER** (or **NEWLINE**). Notice that the cursor seems to have disappeared. Press one of the cursor keys ↑ or ↓ and the cursor will reappear.

More about errors

There are several types of errors. We can point to at least three categories:

(1) syntax errors
(2) wrong keystrokes not resulting in a syntax error
(3) runtime errors

Syntax errors are errors with respect to the rules of BASIC. For example the line

```
10  LET 1A = 2
```

contains a wrongly spelled name (remember – each name has to start with a letter!). Syntax errors are always detected by the computer, so they are never catastrophic. However you must change them, otherwise the computer will not run (execute) your program.

The second type of error is a very tricky one. Suppose you want to type

```
100  LET a = 14
```

but instead of this the following was entered:

```
100  LET a = 15
```

This is really a troublesome error. The computer will never detect it, because nothing is wrong with the syntax (the rules of BASIC). Even on running the program no mysterious or alarming things will

happen. There is only one way to avoid such errors:

CHECK, CHECK AND CHECK YOUR PROGRAM AGAIN!

and do not blame the computer for your mistakes.

The third category of errors is again not very harmful. These are the errors which occur on running your program. As all programmers do, you first accuse the machine, 'It must be that so and so computer!' Of course it is possible . . . but 999 times out of 1000 it is one of those 'stupid errors'.

As an example of a runtime error we examine the following program:

```
10 LET A = 5
20 LET B = 4 + 1
30 LET C = 5 / (A - B)
40 PRINT C
```

Line 10 assigns 5 to A and line 20 assigns 4 + 1 which equals 5 to B. Line 30 indicates that 5 has to be divided by the difference of A and B. Notice that a division is always denoted by

```
/
```

not by the symbol ÷, and also that brackets are used in the way we are familiar with.

In our example the difference between A and B will be zero and therefore 5 has to be divided by 0. This will certainly lead to an error because dividing by zero is not possible – even with computers!

The error will only be detected after the RUN command is given and that is the reason why we call this kind of error a runtime error.

Besides the errors just mentioned there are others more difficult to trace:

- Your program may contain errors with respect to the logic. This means that the computer is not computing what you expect to be computed.
- The program is a correct translation of the formulae but unfortunately your formulae are not correct.
- The program is correct but unfortunately your machine is not working well.

From a philosophical point of view you can never be sure of any result. In practice however things are never as bad as that. Do not be discouraged by errors – keep trying and with a little luck and a lot of effort you will learn a great deal from them and enjoy your computing.

CHAPTER 5

Fifteen easy programs or more about PRINT, INPUT *and arithmetic*

In this chapter we shall focus our attention on the programming tools that deal with the input and output of data. Furthermore information is given about arithmetic functions. All the tools are illustrated by means of 15 very simple programs.

PRINT, the comma and semicolon

The most important instruction used to show results and texts on the screen is the PRINT statement. With this we can add text to explain or describe the results, and by using commas and semicolons we can force the computer to print the results and texts in columns or any format we choose.

Program 1

```
10  LET a = 4
20  LET b = 7
30  LET c = 5
40  LET d = 8
50  LET sum = a + b + c + d
60  LET mean = sum / 4
70  PRINT a,b
80  PRINT c,d
90  PRINT
100 PRINT "mean value = "; mean
```

The result will appear on the screen as follows:

```
4                7
5                8

mean value = 6
```

Lines 10–40 assign values to the variables a–d. Line 50 assigns the sum of these values to the variable with the leading name sum. Line 60 assigns the value 'sum divided by 4' to mean. Therefore mean will represent the mean value of the variables a–d. Line 70 causes the

values a and b to be printed. In this case these values are separated by a comma. This means that the computer will automatically use 2 columns to print these values.

> If a comma is used as a separator in a PRINT statement, the items will be printed in columns.

The next PRINT statement is indicated in line 80. This PRINT statement has a similar construction and therefore the values c and d will also appear in columns. Since only 2 columns are used, c and d will appear directly below a and b. An identical output could be obtained on combining lines 70 and 80:

```
PRINT a,b,c,d
```

Line 90 shows a PRINT statement apparently without any item and indeed 'nothing' will be printed. The effect however is clearly visible – a line is skipped. Line 100 shows a PRINT statement with two items. First the text mean value = has to be printed and secondly the value of the variable mean has to be displayed. These items are separated by a semicolon. This separator forces the computer to display these items immediately after each other.

> If a semicolon is used as a separator the separated items will be printed immediately after each other.

Notice that the text indicated in the PRINT statement is placed between quotes and furthermore that this text is displayed as a result of the PRINT statement *without* these quotes.

PRINT, direct calculations

We can perform a calculation and display the result *directly* using a PRINT statement.

Program 2

```
10 LET a = 10
20 PRINT 4 * a - 2
```

The result is displayed:

```
38
```

This possibility is of special interest. A PRINT statement may be used without any line number so that our computer acts as an ordinary pocket calculator.

Try for instance

```
PRINT 2 * 3                          (result: 6)
PRINT 2 * (3-4)/(6-4)                (result: -1)
```

and

```
PRINT  "HELLO"                                (result: HELLO)
```

This feature is extremely useful if we want to trace errors. Immediately a program is executed we can see the values of the different variables by means of these PRINT commands. In order to illustrate this you can run our first program 'Volume of a cube' and immediately after the result is displayed you can enter these direct PRINT commands.

LET

The LET statement has already been demonstrated. Nevertheless the next example illustrates a 'classic' construction.

Program 3

```
10  LET x = 0
20  LET x = x + 1
30  PRINT x
```

The result is

```
1
```

Line 10 indicates that the value 0 has to be assigned to x. Line 20 indicates that a new value is assigned to x. This value is obtained by adding 1 to the *old* value of x. The old value was 0 hence 0 + 1 = 1 is now assigned to x. Finally the value is printed.

Notice that an expression like x = x + 1 is a correct expression in BASIC whereas in mathematics it is a contradictory expression. Our advice would be to read the symbol = in such a LET statement as 'will be'. In this way line 20 reads 'x will be the old value + 1' instead of 'x equals x + 1' which in fact is never true.

PRINT AT and PRINT TAB

Using the term TAB we can indicate at which position, or column, an item will be printed. For instance,

```
PRINT TAB (3);  "HELLO"
```

or

```
PRINT TAB 3;  "HELLO"
```

moves the PRINT position to column 3.

It is also possible to indicate both the line and the column using a PRINT statement in conjunction with the term AT.

```
AT line number, column position;
```

For instance,

```
PRINT AT 7,5;  "HELLO"
```

indicates that HELLO has to be printed at line 7 and starting at column 5.

Note that the top line starts with line number 0 and the first column with column number 0. The top left corner corresponds to AT 0,0 and the bottom right corresponds to AT 21,31. By means of TAB and AT we can generate many kinds of figures.

The next example shows how a straight line of stars is printed using TAB.

Program 4

```
10 PRINT TAB(1);"*"
20 PRINT TAB(2);"*"
30 PRINT TAB(3);"*"
40 PRINT TAB(4);"*"
```

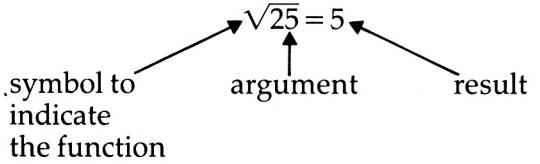
Functions

A scientific pocket calculator provides many functions. A function is merely a 'subprogram' that converts an entered value into another value.

Let us consider the well known function square root:

$\sqrt{\ }$

The value entered is called the argument and the value calculated is called the value of the function or in short the result. For instance



In BASIC such a function is indicated by a term (called a token) and the argument is indicated immediately after the token. For instance the square root function is indicated by the token SQR and $\sqrt{25}$ is indicated by

```
SQR (25)
```

or

```
SQR 25
```

In most versions of BASIC the brackets around the argument must be used. In Sinclair BASIC these brackets can be omitted. With the ZX81/TS1000 this may be important with respect to the limited memory. If memory plays a less dominant role we advise you to use these brackets always.

Program 5

```
10 LET a = 9
20 LET b = SQR(a)
30 LET c = SQR(5 - SQR(1))
40 LET d = SIN(a)
50 LET f = COS(a)
60 LET g = d * d + f * f
70 PRINT "b = ";b,"c = ";c
80 PRINT "d = ";d,"f = ";f
90 PRINT "g = ";g
```

Result

```
b = 3                c = 2
d = 0.412118485      f = - 0.91113026
g = 1
```

Line 10 assigns 9 to a and this value is used as argument in the SQR function (line 20). Line 30 shows that complex expressions containing functions may also be used as arguments. Line 40 shows the sine function: note that the argument is always given in radians. The value sin(a), that is sin(9), is assigned to d. The cosine of a is assigned to f. Line 60 represents the well known mathematical equation

$$\sin^2(a) + \cos^2(a)$$

This ought to be 1 and therefore this expression can be used to test the accuracy of our computer.

On the ZX81/TS1000 the functions are indicated below the keys. In this case we have to press SHIFT and NEWLINE (or ENTER) first. The cursor will change to an F and if for example the function SQR (indicated below the H key) has to be entered we simply press H. On the ZX Spectrum, functions are indicated above the keys (in green) as well as below (in red). Here we must first press CAPS/SHIFT and SYMBOL/SHIFT simultaneously. As a result the cursor will change to an E. Now a function written in green letters can be entered by pressing the key where the function is indicated. The functions written in red are evoked by pressing simultaneously on the appropriate key and the SYMBOL/SHIFT key. Chapter 6 gives an extensive description of the use of the keyboard.

Both ZX computers provide a number of mathematical functions. These are listed in the following table.

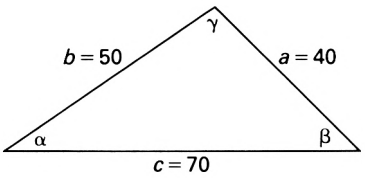
Table of standard mathematical functions

Function	*Indicated on the ZX81 and TS1000*	*Indicated on the ZX Spectrum*	*Meaning*
sin (x)	`SIN`	`SIN`	computes the sine; argument in radians
cos (x)	`COS`	`COS`	computes the cosine; argument in radians
tan (x)	`TAN`	`TAN`	computes the tangent; argument in radians
arcsin (x)	`ARCSIN` (keyboard) `ASN` (screen)	`ASN`	computes the arcsine; result in radians
arccos (x)	`ARCCOS` (keyboard) `ACS` (screen)	`ACS`	computes the arccosine; result in radians
arctan (x)	`ARCTAN` (keyboard) `ATN` (screen)	`ATN`	computes the arctangent; result in radians
e^x	`EXP`	`EXP`	computes e^x
ln (x)	`LN`	`LN`	computes the natural logarithm (base e)
$\sqrt{x}$	`SQR`	`SQR`	computes the square root of x
$\|x\|$	`ABS`	`ABS`	computes the absolute magnitude of x
no indication	`INT`	`INT`	rounds a number down to a whole number (an *integer*); for example, `INT(4.7)` results in `4` and `INT(-3.1)` results in `-4`
no indication	`SGN`	`SGN`	results in a `-1` if the argument is negative, `0` if the argument is zero and `1` if the argument is positive; for example `SGN(-6)` results in `-1` `SGN(7-7)` results in `0` `SGN (6)` results in `1`

Example

In order to give a more realistic example we will calculate the angles of a triangle with given sides a, b and c.

Consider the following figure.



The angles α, β and γ can be found by means of the following mathematical formulae.

$$\alpha = 2 \arccos\left(\sqrt{\frac{S(S-a)}{bc}}\right) \tag{1}$$

$$\beta = \arctan\left(\frac{b \sin \alpha}{c - b \cos \alpha}\right) \tag{2}$$

$$\gamma = \arccos\left(-\cos\left(\alpha + \beta\right)\right) \tag{3}$$

where $S = (a+b+c)/2$ (4)

Note that $\arccos = \cos^{-1}$, $\arctan = \tan^{-1}$ and α, β and γ are given in radians.

These formulae lead directly to the program. In many handbooks the word *algorithm* is used to describe the framework (recipe) that leads directly to the program. This means that formulae (1)–(4) describe our algorithm. The following program can now be written.

Program 6

```
10  LET a = 40
20  LET b = 50
30  LET c = 70
40  LET s = (a + b + c) / 2
50  LET alf = 2 * ACS(SQR(s * (s - a) / (b * c)))
60  LET bet = ATN(b * SIN(alf) / (c - b * COS(alf)))
70  LET gam = ACS( - COS(alf + bet))
80  PRINT "alfa = "; alf
90  PRINT "beta = "; bet
100 PRINT "gamma = "; gam
```

Result

```
alfa = 0.59424503
beta = 0.77519337
gamma = 1.7721543
```

(Note that lines **50**, **60** and **70** actually use two lines on the screen.)

Lines **10**–**30** assign the given sides to a, b and c. Lines **40**–**70** are exact copies of formulae (4), (1), (2) and (3). Lines **80**–**100** result in the output. Note that each argument is enclosed by brackets. As we mentioned before this is not strictly necessary in Sinclair BASIC but since it is obligatory in almost all other versions of BASIC we usually stick to this notation.

Note that expressions like

$$\frac{b \sin \alpha}{c - b \cos \alpha}$$

are 'translated' into

```
b * SIN(alf) / (c - b * COS(alf))
```

The formula is written on a single line by using the symbol / and brackets. If a formula is too long we can divide it, for example

$$z = \frac{(a+b)(c+\sin\alpha)(a+c)}{(b+c)(c+\cos\alpha)(a+\sin\alpha)}$$

can be written in a program as

```
LET x = (a + b) * (c + SIN(alfa)) * (a + c)
LET y = (b + c) * (c + COS(alfa)) * (a + SIN(alfa))
LET z = x / y
```

Note how the nominator and denominator are separately calculated and finally z is determined by dividing x by y.

Note also that we must always indicate the multiplication sign * whereas in the original formula this sign is often omitted.

INPUT

Program 6 obviously gives a very 'rigid' solution. If other values of *a*, *b* and *c* have to be entered the first lines must be changed and errors may occur. A more elegant solution is provided by the **INPUT** statement. The basic form of the **INPUT** statement is

```
line number  INPUT name of a variable
```

This is the only construction that is allowed with the ZX81/TS1000. When a program is run and the computer arrives at such an **INPUT** statement it will halt. A number must now be entered on the keyboard. This number will be assigned to the variable indicated in the **INPUT** statement and immediately after this, the computer proceeds with the execution of the program. When a value has to be entered the display will show a cursor with the letter **L**.

Program 7

```
10 PRINT "illustration of INPUT statement"
20 INPUT a
30 LET b = 2 * a
40 PRINT "* 2 = ";b
```

Example of a result

```
illustration of INPUT statement
```

(now the computer apparently stops and we see a cursor with the letter L indicating that we have to enter a number)

```
6                                   (followed by ENTER or NEWLINE)
* 2 = 12
```

As a result of line **10** the message `illustration of INPUT statement` is printed. The next line shows an **INPUT** statement. This will cause the computer to stop temporarily the execution of the program. Now a number must be entered*. This number will be assigned to **a** and immediately the computer proceeds with the execution of the program. Line **30** assigns $2 \times$ **a** hence 2×6 to **b** and finally this value is printed (line **40**).

Notice that exactly the same program can be used for any value of **a**. This means that the program does not have to be changed in order to work with other values. **INPUT** statements indeed provide us with a very powerful tool to write programs that have a general application.

In order to make programs *user-friendly* we must inform the user of the purpose of the program and furthermore we must provide information about the values that have to be entered. These messages can be displayed by means of **PRINT** statements. In this way we can write a program that can be used by anyone without any previous knowledge.

Sometimes it is important that the programmer writes notes or comments in his program. BASIC provides an easy way of inserting comments that will not be printed during the execution of the program. The only way to read these comments is by giving the **LIST** command. For this purpose we use the **REM** statement which has the following construction:

```
line number  REM text
```

Program 8 shows a user-friendly program with **REM** statements. We have used the very first program of our book (see page 22) in order to see how it can be converted into a user-friendly program.

* If **STOP** (on key **A**) is entered the program is terminated.

Program 8

```
10  REM illustration of a user friendly program
20  REM this program is based on the first
30  REM program of this book
40  PRINT "calculation of volume of a block"
50  PRINT "enter width:";
60  INPUT b
70  PRINT b
80  PRINT "enter length:";
90  INPUT l
100 PRINT l
110 PRINT "enter height:";
120 INPUT h
130 PRINT h
140 REM now comes the calculation
150 LET v = h * b * l
160 PRINT
170 PRINT "the volume is ";v
```

Example of a result

```
calculation of volume of a block
enter width: 3
enter length: 4
enter height: 2

the volume is 24
```

Lines 10–30 contain only REM statements. We can see that the remarks written after REM are not printed during the execution of the program. They will only appear if we give a LIST command.

Lines 50, 80, and 110 provide information necessary to the user. Notice that each PRINT statement ends with a semicolon and furthermore that after each INPUT statement a PRINT instruction is given so that the entered value is displayed on the screen. In this way all the values will be printed immediately after the appropriate texts.

Line 140 gives another illustration of the REM statement.

Finally the answer is printed with the text "the volume is ". Notice the space between is and the last quote. This puts a space between is and the value of v as they are printed.

The ZX Spectrum offers a number of options:

(a) Text can be entered in an INPUT statement, for example:

```
10 INPUT "enter width"; b
```

The text

```
enter width
```

will only appear during the execution of the INPUT statement.

(b) INPUT statements may contain a part that is executed like a PRINT statement. This option is really quite unique. Let us consider the next example.

```
10 LET a = 4
20 INPUT ("previous value = ";a);" enter new value ";a
30 PRINT a
```

results in

```
previous value = 4 enter new value
```

Notice how the part between brackets is printed; even the value of a is displayed!

READ, DATA and RESTORE

These statements again can only be used on a ZX Spectrum. READ and DATA provide a third method of assigning data to variables. This method is of special interest if we are dealing with a relatively large number of variables. The next program gives an example.

Program 9

```
10 READ a,b,c,d
20 LET mean = (a + b + c + d) / 4
30 PRINT "mean = "; mean
40 DATA 4,5,6,5
```

Result mean = 5

Line 10 starts with the keyword READ followed by a list of variables. The result is that the first value indicated after DATA is assigned to the first variable after READ. This means that 4 is assigned to a, 5 to b, 6 to c and finally 5 to d.

Line 20 computes the mean value of a, b, c and d and assigns it to mean. This value is printed (line 30). In fact this is the last executable statement of our program. The line that starts with DATA can be put anywhere in the program. You may split the READ and DATA lists as is illustrated by the following programs.

Program 10

Version 1

```
10 READ a,b
15 READ c,d
20 LET mean = (a + b + c + d) / 4
30 PRINT "mean = ";mean
40 DATA 4,5,6,5
```

Version 2

```
10 READ a,b,c,d
20 LET mean = (a + b + c + d) / 4
30 PRINT "mean = ";mean
40 DATA 4,5
45 DATA 6,5
```

Both programs are equivalent to program 9. Notice that in version 1 the list of variables is divided into two **READ** statements. Version 2 shows a program in which the list of values is divided into two **DATA** lines.

If a program with a **READ–DATA** combination is executed, a pointer points (so to speak) at the first value of the first **DATA** line. Every time a value is assigned to a variable the pointer shifts to the next value to be read. We can control this pointer by means of the **RESTORE** statement, illustrated in the next program.

Program 11

```
10 READ a,b
20 PRINT a,b
30 RESTORE
40 READ c,d
50 PRINT c,d
60 DATA 1,2
70 DATA 3,4
```

Result

```
1             2
1             2
```

Line **10** assigns **1** to **a** and **2** to **b**. These values are printed as a result of the next line. Then comes the **RESTORE** statement. As a result of this statement the pointer is reset to the first value of the first **DATA** statement. The proof is given by the next **READ** statement. Instead of **3**, **1** is again assigned to **c** and in the same way **2** is assigned to **d**.

Now if we enter a new line in our program

```
5 RESTORE 70
```

then the result becomes

```
3                4
1                2
```

Note that a line number may also be indicated in a **RESTORE** statement. It is a unique option of Sinclair BASIC. As a result the pointer will point to the **DATA** line indicated. We may even indicate this line number by an expression as is illustrated by our next example.

Program 12

```
10 PRINT "Which table do you want?"
20 INPUT "Enter number of table:";n
30 RESTORE 50 + n
40 READ a,b,c,d,e,f,g,h,i,j
50 PRINT a,b,c,d,e,f,g,h,i,j
51 DATA 1,2,3,4,5,6,7,8,9,10
52 DATA 2,4,6,8,10,12,14,16,18,20
53 DATA 3,6,9,12,15,18,21,24,27,30
54 DATA 4,8,12,16,20,24,28,32,36,40
55 DATA 5,10,15,20,25,30,35,40,45,50
56 DATA 6,12,18,24,30,36,42,48,54,60
57 DATA 7,14,21,28,35,42,49,56,63,70
58 DATA 8,16,24,32,40,48,56,64,72,80
59 DATA 9,18,27,36,45,54,63,72,81,90
```

Example of a result

```
Which table do you want?
Enter number of table:8
8                16
24               32
40               48
56               64
72               80
```

By means of the statement **RESTORE 50 + n** the appropriate table is automatically selected.

Multiple statements in a single line

The ZX Spectrum allows more than one statement to be written in a

single line. The symbol : is used as a separator. Consider the next example:

```
10 LET a = 4: LET b = 5: PRINT a * b
```

This line actually describes a complete program consisting of three statements.

With the ZX81/TS1000 we cannot write more than one statement in a single line and the previous program becomes:

```
10 LET A = 4
20 LET B = 5
30 PRINT A * B
```

In chapter 7 we shall discuss the IF...THEN statement. It will be shown that between IF and THEN a particular test can be indicated and further that after THEN a statement is indicated that will only be executed if the test proves true. With the ZX Spectrum more statements can be indicated after THEN simply by using : as a separator.

Different number systems

Consider the next very simple program.

Program 13

```
10 INPUT a
20 INPUT b
30 PRINT a * b
```

Two values must be entered and the result of the multiplication of these two numbers will be displayed. Not very interesting . . .? Just wait . . . suppose that after the RUN command we enter

```
   15 000 000
and 45 000 000
```

This will actually result in

```
6.75E + 14
```

If you already use a scientific pocket calculator you are probably familiar with this notation.

A number like

6.75E + 14

indicates

6.75×10^{14}

or in other 'words'

$6.75 \times 100000000000000$

or

675000000000000

We see that a number like 6.75E + 14 is understood as a multiplication of a number (6.75) and a power of 10 (E + 14 = 10^{14}). The number after the letter E indicates the power of 10. Both negative and positive numbers can be used.

The advantage of this notation is of course that it provides an easy way to indicate very large (or very small) numbers. For example, if we enter

0.00000015
and 0.00000045

the computer shows

6.75E−14

which indicates

6.75×10^{-14}

or

0.000000000000675

The number before **E** is called the mantissa; it gives us the meaningful digits of the answer.

Up till now we have met three kinds of numbers:

whole numbers (integers) like:
−2, −1, 0, 1, 2
numbers with a decimal fraction (real numbers) like:
5.67, 8.14, −3.1415
numbers with the letter E (also real numbers) like:
6.75E + 14, −6.75E − 14

The ZX Spectrum also allows the possibility of entering numbers that are indicated by a series of zeros and ones. We refer to the binary numbers that are explained in more detail in chapter 14. In this case the word **BIN** is indicated before the series of zeros and ones.

Program 14

```
10 LET a = BIN 101
20 LET b = BIN 10011
30 LET c = a * b
40 PRINT "a = ";a, "b = ";b, "c = ";c
```

Result

```
a = 5          b = 19
c = 95
```

Notice that the token **BIN** is indicated in green letters above the **B** key.

Besides binary numbers our computers allow us to use *random numbers*. A random number is generated by the **RND** function and its most remarkable characteristic is the fact that you will never know in advance the exact value.

The next example illustrates the generation of random numbers.

Program 15

```
10 LET a = RND
20 LET b = RND
30 LET c = RND
40 LET d = RND
50 PRINT a,b,c,d
```

The aim of this program is to display four random numbers between **0** and **1**.

Each time the program is entered and executed the same sequence appears. However if we extend the program with

```
5 RAND
```

different, hence truly, random numbers appear. After **RAND** an integer can be used, for instance,

```
5 RAND 6
```

Now every time the program is run, the same sequence of random numbers will appear. Note that in the *User's Manual* of the ZX Spectrum **RAND** is indicated as **RANDOMIZE** whereas on the keyboard only **RAND** is indicated.

Random numbers play a dominant role in all kinds of games. The next expression gives an example by showing you how a dice is simulated.

```
PRINT INT (RND * 6 + 1)
```

So if you want to gamble with your computer . . . now is the time!

CHAPTER 6

A survey of the keyboard

Easy familiarity with the keyboard is essential to your enjoyment of computing so we will give a complete survey of the keyboards of the ZX81 and TS1000 as well as that of the ZX Spectrum. We use the same method of entering data with both keyboards. All reserved words, function names etc. that are indicated on the keyboard are entered by a single key, only the cursor has to be changed sometimes.

Colours usually indicate what to do, e.g. to enter the red symbols on the keys we have to press the red `SHIFT` key first. If we look at this survey undoubtedly the question arises, 'Shall I ever get it right in practice?' Don't worry, everything will fall into line!

The keyboard of the ZX81 and TS1000

1 Words written in white above the keys

If the mains adapter is connected the screen will display a cursor with the letter

```
K
```

This means that the computer accepts a line number or a reserved word (Keyword) like `PRINT` that is indicated *above* each key. Note also that the commands, for example `RUN`, `LIST` and `NEW`, are indicated above the keys. This means that we can give a command directly by pressing the appropriate key — at least if the cursor shows a `K`!

2 Main (largest) symbols on the keys

The cursor can change from a `K` to an `L`

- after a keyword is entered during programming
- due to an `INPUT` statement

The `L` indicates the *L*etter mode. If a key is pressed now we shall evoke the largest (main) symbol on the key. For instance if we press `P` the letter `P` will be displayed.

3 Red symbols and words (tokens) on a key

Every key also has a symbol or a word (token) indicated in red. If the cursor shows an L we can evoke this symbol by pressing SHIFT (also indicated in red) and while holding it down press the key with the symbol to be entered.

4 Graphics symbols

Many keys show graphics symbols. A graphics symbol is a square with a certain pattern. You can evoke such a graphics symbol by first pressing

SHIFT and 9 together

(Notice that the word GRAPHICS is also indicated on this key.) Now the cursor will change to

G

indicating that we are now in the graphics mode. Now if we press for example H with SHIFT we can observe a 'dotted square'. We can quit this mode by pressing SHIFT and 9 once more or by using the NEWLINE key (or ENTER—American versions of ZX81).

5 Words or symbols below the keys (functions)

Below most of the keys a word or symbol is indicated. Besides the symbol π which represents the well known number 3.141592653 these words indicate functions which can be evoked by first pressing

SHIFT and NEWLINE (or ENTER)

(Notice that the key NEWLINE also has FUNCTION on it.) Now the cursor will change to

F

indicating that we are in *F*unction mode. The key with the function we want to enter is pressed and immediately afterwards the cursor will change to

L

6 Special keys

BREAK
BREAK is indicated above the key with SPACE. This key can be pressed (together with SHIFT) if a program is running and we want to stop (break) the execution.

NEWLINE (or ENTER—American versions of ZX81)
NEWLINE has to be pressed immediately after the following:

- a command is given
- a line has to be entered into a program
- data is entered in order to obey an INPUT statement

7 Syntax error cursor S

If a line to be entered contains a syntax error (see page 30) a cursor showing S will be displayed at the scene of the crime!

The keyboard of the ZX Spectrum

The keyboard of the Spectrum shows even more symbols. We shall see that in addition more cursors may appear. In many cases however we shall find that things proceed in exactly the same way.

1 Words written in white on the keys

If the power is turned on we see

```
© 1982 Sinclair Research Ltd
```

Pressing ENTER will result in a cursor with the letter

```
K
```

This indicates that the computer will accept a line number or a reserved word written in white *on* the key (like PRINT on the key with P). Note also that commands like RUN and NEW are indicated on the keys. Only EDIT is written above a key (see paragraph 7 of this section).

2 Main (largest) symbols on the keys

The cursor can change from a K to an L:

- after a keyword is entered during programming
- due to an INPUT statement

As a matter of fact during programming the cursor will usually indicate the *L*etter mode:

```
L
```

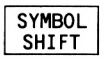
If L is shown we can obtain lowercase letters (and numbers) just by pressing on the keys. If on the other hand we first press

and hold it down we can enter capitals.

The L cursor changes to a C cursor on pressing CAPS LOCK. In order to do this we have to press 2 while pressing CAPS/SHIFT. In the C mode all letters appear as capitals. By entering CAPS LOCK again the computer will return to the L mode.

3 Red symbols and words on the keys

The red symbols and words that are indicated on the keys can be entered by using the red key



While holding this key down we can enter the symbol or word required by pressing the appropriate key.

4 Green words above the keys

To enter the green words we have to press first on *both* SHIFT keys. Now the cursor will indicate the *E*xtended mode:

E

This mode lasts for one key depression—when a key is pressed the green word above it will be entered. If a digit key is pressed it gives the control function indicated (in white) above this key.

5 Red words and symbols below the keys

In order to enter the red words and symbols that are indicated below the keys we first have to enter the E mode (see above). Now the red word or symbol is entered if the key is pressed *simultaneously* with the red SYMBOL/SHIFT key.

6 Graphics symbols

Graphics symbols are the squares with specific patterns. First we have to enter the *G*raphic mode by pressing on 9 while holding down CAPS/SHIFT. The screen shows

G

Now we can enter graphics symbols. For instance pressing 6 results in a pattern with black squares. This mode can be left (exited) by pressing CAPS/SHIFT and 9 again. On using the CAPS/SHIFT key we can obtain the inverse pattern of the patterns indicated on the keys 1–8. For instance a black square is obtained on pressing 8 while holding the CAPS/SHIFT key down.

7 White words and symbols above the digit keys

The white words are entered in both K, L, and C modes by pressing on the key while holding down the CAPS/SHIFT key.

8 Other words above the digit keys

Other words apply to colour control functions. In order to invoke them we first have to enter the extended E mode (see paragraph 4) and then we must press the CAPS/SHIFT key simultaneously.

9 Special keys

BREAK
BREAK is indicated above SPACE. This control function can be entered by pressing CAPS/SHIFT and SPACE. It will stop the computer if a program is running.

ENTER
The ENTER key has exactly the same function as NEWLINE has with the ZX81. We have to press ENTER for the following:

- for entering a command
- for entering a line
- for entering data in order to obey an INPUT statement

STOP
STOP is indicated in red on the A key. This key is of special interest if we want to stop a program while it is waiting for data to be entered as a result of an INPUT statement.

CHAPTER 7

Putting logic into your programs

Let us consider the following quiz-game. A program will display a certain question and you have to enter an answer. If the answer is correct the message 'right answer' must appear and if the answer is wrong the message 'wrong answer . . . try again!' must be displayed. With the programming tools we have met so far this is quite impossible. With the programs we have dealt with, all instructions were obeyed successively one by one.

In order to solve problems like this in which decisions have to be made BASIC offers the IF statement. By using this statement we can, in a manner of speaking, put logic into our programs.

IF...THEN and GOTO

The IF statement always has the following form

```
IF condition THEN statement
```

After the term IF a particular condition is always indicated and *if* this condition is true, the statement indicated after THEN will be executed.

In many cases we use a GOTO statement after the term THEN. A GOTO statement has the following form:

```
GOTO line number
```

As a result of such a GOTO statement the computer will jump to the line indicated.

The following program illustrates the IF statement as well as the GOTO statement.

```
10 PRINT "WHAT'S THE RESULT OF 2 * 3?"
20 INPUT A
30 IF A = 6 THEN PRINT "RIGHT ANSWER"
40 IF A = 6 THEN GOTO 70
50 PRINT "WRONG ANSWER ... TRY AGAIN"
60 GOTO 20
70 PRINT "END OF GAME"
```

An example of a result would be as follows:

```
WHAT'S THE RESULT OF 2 * 3?
4
WRONG ANSWER ... TRY AGAIN
6
RIGHT ANSWER
END OF THE GAME
```

In order to please the ZX81/TS1000 owners we have only used capitals this time, but it makes no difference if lower-case letters are used.

Lines 10 and 20 will not cause any difficulties. Line 30 indicates an IF statement. The condition to be tested is

```
A = 6?
```

This means that *if* A equals 6 the statement after THEN is obeyed otherwise the IF statement is just skipped over.

In our example the first answer that is entered is 4. Obviously this is wrong so both lines 30 and 40 are ignored. Now the PRINT statement of line 50 is executed and the computer reaches line 60. This line describes a GOTO statement and as a result the computer jumps back to line 20. Now the answer 6 is entered. As a result of the IF statement at line 30 the message RIGHT ANSWER will appear. The computer continues with the next line which again indicates an IF statement. Also with this IF statement the condition is true and the GOTO statement after THEN will be obeyed so that the computer jumps to line 70.

One of the funny things with logic is the fact that an equivalent program, having a different logical structure, can nearly always be developed. As an example we will present two programs that behave in exactly the same way as the previous program.

Example 1

```
10 PRINT "WHAT'S THE RESULT OF 2 * 3?"
20 INPUT A
30 IF A = 6 THEN GOTO 60
40 PRINT "WRONG ANSWER ... TRY AGAIN"
50 GOTO 20
60 PRINT "RIGHT ANSWER"
70 PRINT "END OF THE GAME"
```

Example 2

```
10 PRINT "WHAT'S THE RESULT OF 2 * 3?"
20 INPUT A
30 IF A <> 6 THEN 60
40 PRINT "RIGHT ANSWER"
50 GOTO 80
60 PRINT "WRONG ANSWER ... TRY AGAIN"
70 GOTO 20
80 PRINT "END OF THE GAME"
```

The first example contains only one `IF` statement. The second example contains an `IF` statement with a different condition:

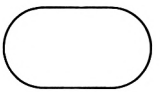
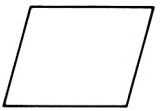
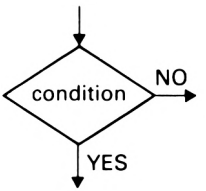
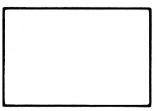
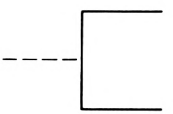
```
A <> 6
```

By means of the expression `A <> 6` the condition '`A` not equal to `6`?' is indicated.

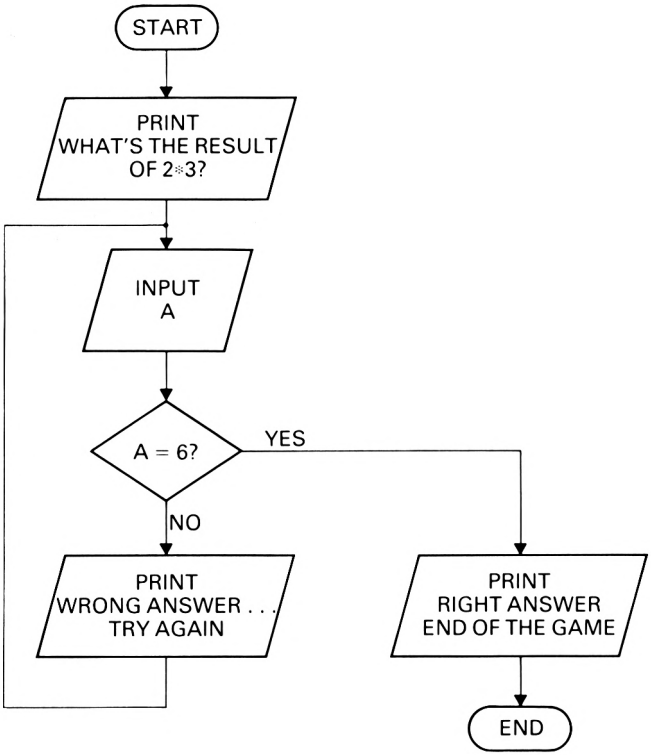
It is a challenge to many programmers to write programs that are as short as possible. Personally however I would question this style. Of course it seems nice to write programs that are as short as possible, but 'clever logic' can make a program difficult to understand. Clear programming usually means that a program reads like a novel.

Flow charts

It is a good tradition to make schematic diagrams of the structure of the job to be performed. Such diagrams are called flowcharts because they show the flow of the instructions or, more precisely, the sequence in which the instructions are obeyed. Special symbols are used as follows:

Symbol	*Use*
	to indicate the start or end of a program
	to indicate an `INPUT` or `PRINT` statement (input/output)
	to indicate a decision, the condition (test) is indicated within the symbol and the possible results are indicated by YES and NO
	to indicate any operation on data
	to indicate the direction of the flow
	to indicate comments on the flowchart

As an example the following flowchart describes the program of example 1 (page 53).



If we examine this diagram it is clear that there is a path by which it is possible to return to a previous point in the program. This path forms a *loop*. We shall see that loops play an important role in many programs. There are even special BASIC statements to indicate loops (for example the `FOR ... NEXT` statement, see chapter 9).

Finally we should emphasize that flowcharts are usually constructed before writing the program. They can be used as a tool for developing programs.

Relational operators

In the expression `IF A = 6 THEN GOTO 70` the variable `A` is *related* to `6` by means of the symbol `=`. Symbols that are used to express a possible relation between two items are called *relational operators*. We may use a number of relational operators and in the next table a survey is given.

Relational operator	*Meaning*
=	equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
<>	not equal to

If you look at the keyboard of the computer you will find all these symbols. Note that compound symbols like <= are entered by pressing only one key!

Combining logical conditions

It is often useful to combine logical conditions. In true life many combinations of logical conditions are met. For instance if we want to drink some milk we obey the instruction:

IF there's still milk AND the milk is not bad THEN drink the milk.

In this case we used the word AND to combine the two conditions. There are three basic logical operators (terms):

AND **OR** **NOT**

Let us first explain the meaning of these terms by using *truth tables*. In a truth table all possible situations are depicted together with the results of each combination.

Truth table of **AND**

Condition 1	*Condition* 2	*Condition* 1 **AND** *Condition* 2
true	true	true
true	false	false
false	true	false
false	false	false

Note that the combination 'condition 1 **AND** condition 2' only comes out true if both conditions are true.

Truth table of **OR**

Condition 1	*Condition* 2	*Condition* 1 **OR** *Condition* 2
true	true	true
true	false	true
false	true	true
false	false	false

Note that the combination 'condition 1 **OR** condition 2' comes out as true if *at least* one of the conditions is true. In other words, the combination is only false if both conditions are false.

Truth table of **NOT**

Condition	*Result of* **NOT** *condition*
true	false
false	true

Obviously the term **NOT** just inverts the result.

The next example gives a simple illustration of the use of a logical operator.

```
10  PRINT "ENTER AN EVEN NUMBER < 10"
20  INPUT A
30  LET B = 2 * INT (A/2)
40  IF B = A AND A < 10 THEN PRINT "OK"
50  GOTO 10
```

An example of a result is as follows:

```
ENTER AN EVEN NUMBER < 10
7
ENTER AN EVEN NUMBER < 10
8
OK
ENTER AN EVEN NUMBER < 10
.
.
.
```

Line **10** describes a **PRINT** statement that tells the user to enter a particular even number and line **20** indicates the corresponding **INPUT** statement. Line **30** shows an interesting construction. If **7** is entered **2 * INT (A/2)** leads to

2 * INT (7/2) $\rightarrow$ **2 * INT (3.5)** $\rightarrow$ $2 \times 3 \doteq 6$

If however 8 is entered it leads to

```
2 * INT (8/2) → 2 * INT (4)    → 2×4=8
```

Obviously `2 * INT (A/2)` leads exactly to `A` if `A` is an even number.

Line `40` shows an `IF` statement in which two conditions are combined by means of an `AND` operator. Only if both conditions are true will the message OK be printed.

As a result of the `GOTO` instruction at line `50` this program will always start again. In order to prevent it from becoming a never ending story, we have to enter the `STOP` function:

```
STOP
```

Note that the `BREAK` key does not work since the computer is usually not calculating with this program but it is waiting for a number to be entered. Only by means of entering `STOP` can we terminate the execution.

`AND`, `OR` and `NOT` can also be used in expressions that lead to a numerical result. The next program shows an illustration.

```
10  INPUT A
20  INPUT B
30  LET MAX = (A AND A >= B) + (B AND A < B)
40  PRINT "MAX=";MAX
```

An example of a result is as follows:

```
5
6
MAX = 6
```

In order to explain this result we have to look at the rules for AND, OR and NOT in numerical expressions.

A AND B	*becomes*	A if B is true (non zero) 0 (false) if B is false (zero)
A OR B	*becomes*	1 if B is true (non zero) A if B is false (zero)
NOT A	*becomes*	0 (false) if A is true (non zero) 1 (true) if A is false (zero)

Notice that 'true' corresponds to 'non zero' and 'false' to 'zero'.

CHAPTER 8

String variables

BASIC offers the possibility of assigning characters to variables. This is a very important feature because many tasks deal with the manipulation of characters: for example think about the broad field of administrative applications. The mathematics is usually very simple with these kinds of problems.

If we deal with stock administration, we work with names that represent items and numbers that indicate the amount of items. The mathematics required is usually restricted to multiplication, addition and subtraction.

Another field of interest is text processing where a draft text may be entered and before it is finally printed, you may erase the errors and rewrite parts of the text. In addition you may give a command to insert extra spaces so that each line ends at exactly the same place. This has the effect of setting the text like the pages of a book. For all these purposes we can write BASIC programs; alternatively commercial programs are available and may be bought.

The next simple program illustrates how a certain series of characters or, as we shall call it, *a string* is assigned to a variable.

```
10  LET A$ = "TO BE OR NOT TO BE"
20  PRINT A$;" ";"THAT IS THE QUESTION"
```

Result

```
TO BE OR NOT TO BE THAT IS THE QUESTION
```

In line **10** the string `TO BE OR NOT TO BE` is assigned to the variable **A$**. As always strings are placed between quotes. Because the variable **A$** apparently stores a string we call it a *string variable*. Notice that the name of a string variable always ends with a dollar sign.

> Each string variable ends with a dollar sign. Only single letters may serve as the name of a string variable.

Line **20** shows a `PRINT` statement. It illustrates that there are two ways to print texts: first the variable **A$** is printed, then a space is printed and finally the text `THAT IS THE QUESTION`.

Types of string

There are two types of strings that need our special attention. The first type is the 'tricky' string that only contains digits (and eventually a decimal point). This type of string suggests that we are dealing with a number, while of course we are not!

An example:

```
10 LET B$ = "12"
20 LET C = 3
30 LET D = C * B$
40 PRINT D
```

Result

```
error message
```

Indeed this program contains a rather gross error. Line **30** requires **C** to be multiplied by **B$**, but since **B$** is a string we are looking for trouble.

It is worth mentioning that BASIC provides many string functions. One of these functions offers the possibility of converting such a 'number string' into a real number (**VAL** function).

The second type of string we have to pay attention to is the

empty or *null string*

You may compare this string with the value zero in mathematics.

The null string contains no characters at all. It is indicated by two successive quotes *without* a space between them:

""

To enter these quotes we have to use the key **P** twice (of course with the **SHIFT** key). ZX81/TS1000 owners also have the symbol "" on the **Q** key but this symbol has to be used if we also want to place quotes inside a string.

At first glance a null string seems rather curious; however in many programs they play an important role. Let us look for example at the next program:

```
10 IF INKEY$ = "" THEN GOTO 10
20 PRINT INKEY$
```

N.B. In order to start this program, press once on the **NEWLINE** or **ENTER** key.

Line **10** shows an **IF** instruction. **INKEY$** is apparently a wrong name for a string variable, since we have been taught that only single letters may be used for the name of a string variable. This is certainly true, but **INKEY$** is a special variable with a reserved name. You will find this name above the **N** key (ZX Spectrum) or below the **B** key (ZX81/TS1000). The key that is momentarily pressed is assigned to **INKEY$**.

This means that at the moment the computer arrives at the expression `IF INKEY$...`, it looks at the keyboard and assigns to `INKEY$` the key that is pressed at that very moment. Usually no key is pressed and in this case the null string is assigned to `INKEY$`.

The result of the `IF` statement is simply that as long as no key is pressed, the computer will always return to line `10`, so in fact it stays at line `10`. If on the other hand a key is pressed `INKEY$` will no longer be `""` and the computer skips to the next line. Now the `PRINT` statement will be obeyed. This little program is often encountered as a part of a larger program. An illustration of the use of this instruction follows:

```
10 REM REACTION TIMER
20 PAUSE 300
30 LET time = 0
40 LET d = 3
50 PRINT "NOW!"
60 LET time = time + d
70 IF INKEY$ = "" THEN GOTO 60
80 PRINT "reaction time = ";time/100
```

Line `70` shows our well known construction. As long as no key is pressed the computer returns to line `60` and as a result `d` is added to `time`. If we start our program first a pause of 300/50 = 6 seconds is caused by the `PAUSE` statement. This statement always has the following form:

```
PAUSE number or expression
```

The computer counts the number of TV scans, so 50 (USA 60) corresponds to one second. Line `30` assigns the initial value 0 to time. Notice that in contrast to many BASIC versions, we have to indicate this initial value. Line `40` assigns the step value to `d`. Now a message is displayed (line `50`):

```
NOW!
```

At this moment you must press a key as quickly as possible.

ZX81/TS1000 owners have the disadvantage that messages are only displayed during computing if the computer is running in SLOW mode. This means that `d` has to be changed and the accuracy is lowered because larger steps are obviously needed.

Manipulations with strings

There is only one operator that can be used with strings:

```
+
```

By means of + you can 'add' strings or, as it is called, you can *concatenate* strings.

Example

```
10 LET A$ = ""
20 LET Z$ = INKEY$
30 IF Z$ = "" THEN GOTO 20
40 LET A$ = A$ + Z$
50 PRINT A$
60 PAUSE 25
70 IF Z$ <>"S" THEN GOTO 20
80 PRINT "THE END"
```

Example of a result

```
d
do
dol
doll
dolls
THE END
```

Line **10** assigns the null string to **A$**. This defines the initial value of **A$**. Line **20** assigns **INKEY$** to **Z$** and line **30** tests if **Z$** is still a null string which means that no key is pressed. Once a key is pressed it is added (concatenated) to **A$** and printed. Line **60** indicates a **PAUSE** statement. The computer will introduce a pause of approximately 0.5 seconds. Without this **PAUSE** statement the computer immediately returns to line **20** (at least if no **S** is pressed). If the key is still pressed it is again displayed. This 'bouncing effect' is eliminated by the **PAUSE** statement. Finally a test is performed to see if an 'S' is pressed (line **70**). If so the message **THE END** will appear and the program is terminated.

Although we may only use the symbol + in order to indicate a specific manipulation with strings, there are many functions that apply to strings. In the next section we shall briefly discuss each function.

LEN

LEN determines the number of characters in a string. Spaces in a string are also counted as characters.

Example

```
10 LET A$ = "not so much"
20 LET N = LEN A$
30 PRINT N
```

Result

```
11
```

With most BASIC versions parentheses have to be used in order to indicate the argument of the function. In this case line 20 has to be written as

```
20 LET N = LEN (A$)
```

In fact this method of notation is also allowed in Sinclair BASIC. If you want to develop programs that can also run on other computers, we strongly advise you to use parentheses.

A short equivalent 'program' to our example is

```
10 PRINT LEN "not so much"
```

CODE

Each character has its own code. This code consists of a decimal number. The ZX81/TS1000 uses rather unique codes whereas the ZX Spectrum uses the well-known ASCII codes. In Appendix III all the codes are listed and we can observe the differences between those of the ZX81/TS1000 and those of the ZX Spectrum.

The function CODE determines the code of a character.

Example

```
PRINT CODE "A"
```

Result (ZX Sprectrum)

```
65
```

You can check this result in Appendix III.

If the argument of the function is a string of more characters, for example

```
PRINT "AMSTERDAM"
```

the function CODE will only look at the first character of this string.

Example

```
10 PRINT "Are you bored?"
20 INPUT A$
30 IF CODE A$ <> CODE "y" THEN GOTO 20
40 PRINT "Well do something"
```

Result

```
Are you bored?
yes
Well do something
```

Some users of the ZX81/TS1000 will certainly want to convert their programs so that they can be executed on a ZX Spectrum.

The codes of characters may cause difficulties, but the next program shows how this problem can be solved. It determines the ASCII code for a given character and can be used with the ZX Spectrum and the ZX81/TS1000.

Program

```
10 LET S = 0
20 IF CODE "A" = 38 THEN LET S = 27
30 INPUT Q$
40 LET C = CODE Q$ + S
50 PRINT "ASCII CODE = ";C
```

Result

```
A
ASCII CODE = 65
```

In this case the program first tests whether it is dealing with a ZX Spectrum or a ZX81/TS1000. If it is executed on a ZX81 `CODE "A"` results in `38` hence `S` is changed to `27`. Now the value of `CODE "A" + S` = 38 + 27 = 65 is assigned to `C` (line `40`). In this way the ASCII code is always found.

CHR$

The function `CHR$` is the inverse function of `CODE`. Now the argument of the function is a number (between 0 and 255) and the function returns the corresponding character.

Example

```
PRINT CHR$ 65
```

Result

```
A
```

on the ZX Spectrum. In order to obtain an `A` on the ZX81/TS1000 you have to enter

```
PRINT CHR$ 38
```

STR$

The function `STR$` is used to convert a number into a corresponding string.

Example

```
10 LET k = 81
20 LET a$ = "ZX"
30 LET b$ = STR$ k
40 LET c$ = a$ + b$
50 PRINT c$
```

Result

```
ZX81
```

In line 10 the number 81 is assigned to k. In line 30 STR$ k is assigned to b$. This means that the string 81 is assigned to b$. In order to illustrate that string manipulations are possible line 40 describes a concatenation.

VAL

The function VAL can be considered as the inverse function of STR$. Here the argument is a string and the result is a number written with the same characters.

Example

```
10 LET a$ = "81"
20 LET k = VAL a$
30 PRINT k/9
```

Result

```
9
```

In line 10 the string consisting of the characters 8 and 1 is assigned to a$. This string is converted to the number 81 by means of the VAL function at line 20. The result illustrates once more that we are now dealing with numbers.

The ZX Spectrum also provides the function VAL$. This function converts a string to a string and strips the bounding quotes.

Example

```
VAL$ """ NAME""" becomes "NAME"
```

Slicing

Slicing is a very powerful tool of our computers. It is essentially designed for creating or generating *substrings*. A substring is a string

that is entirely contained within another string. For example the string

```
" ABC "
```

contains the following substrings:

```
" A "    " B "    " C "    " AB "    " BC "    and    " "
```

" AC " is not a substring because it cannot be sliced in 'one piece' out of the original string " ABC " .

The usual notation to indicate a substring is

string (start-position TO end-position)

A few examples

Consider the string A$ = " MONKEY " . In this case we can obtain:

```
A$(2 TO 4)      results in    " ONK "
A$(2 TO)        results in    " ONKEY "
A$(2)           results in    " O "
A$(TO 4)        results in    " MONK "
```

Notice that if no start- or end-position is mentioned the first and last positions in the string are automatically taken. Furthermore it is apparently possible to obtain a single character just by indicating its position.

Instead of numbers we can also use expressions or variables to indicate the positions.

Example

```
10  INPUT q$
20  LET k = 0
30  LET k = k + 1
40  PRINT q$(1 TO k)
50  IF k < LEN q$ THEN GOTO 30
60  STOP
```

For instance MONKEY results in

```
M
MO
MON
MONK
MONKE
MONKEY
```

Time for a game!

The next program deals with a word game. Player 1 enters a word and player 2 will try to guess it.

Program

```
 10  PRINT  "word game"
 20  LET c = 0
 30  LET z$ = ""
 40  PRINT  "enter a word"
 50  INPUT w$
 60  LET k = 0
 70  LET k = k + 1
 80  LET z$ = z$ + "."
 90  IF k < LEN w$ THEN GOTO 70
100  PRINT AT 5,0;  "enter a letter"
110  INPUT l$
120  LET k = 0
130  LET k = k + 1
140  IF w$(k) = l$ THEN LET z$(k) = l$
150  PRINT z$(k);
160  IF k < LEN w$ THEN GOTO 130
170  LET c = c + 1
180  PRINT
190  IF w$ = z$ THEN GOTO 210
200  GOTO 100
210  PRINT  "you needed ";c;"  goes"
```

Example

```
word game
enter a word
moon  (this word disappears immediately)
enter a letter  (now the second player comes into action)
a
enter a letter
....
o
enter a letter
.oo.
n
enter a letter
.oon
m
moon
you needed 4 goes
```

Line 20 assigns 0 as an initial value to c. The variable c is used to count the number of turns. The variable z$ (line 30) represents a word to which the null string is initially assigned. Line 50 defines that a word has to be entered and will be assigned to w$. Lines 70–90 result in a string of points and the number of points corresponds to the number of characters of the word entered. Lines 100–110 ask for a letter to be entered. Lines 120–160 check whether this letter corresponds to a letter of the word, if so it will be printed, otherwise a point is displayed. Line 170 counts the number of turns. The IF statement at line 160 checks whether the complete word has been guessed—if not the program is repeated from line 100.

Using STOP to break a program

Consider the following program:

```
10  PRINT  "LITTLE ECHO PROGRAM"
20  INPUT A$
30  PRINT A$
40  PRINT A$
50  GOTO 10
```

Example of a result

```
LITTLE ECHO PROGRAM
ECHO
ECHO
LITTLE ECHO PROGRAM
PARROT
PARROT
LITTLE ECHO PROGRAM
etc.
```

Obviously this program goes on and on and if we try to stop it by means of the BREAK key and the STOP function we will be unsuccessful. The only way out is to use the ERASE key (DELETE or RUBOUT) and then enter the STOP function followed by ENTER (or NEWLINE).

CHAPTER 9

FOR...NEXT *statement*

In chapter 7 we described a program in which it was possible to display a question and see if the user knew the answer. In fact only one question was shown, which is not very many. The next example shows a program that may serve as an educational exercise.

```
 10  REM program to learn multiplications tables
 20  REM k counts the number of questions
 30  LET k = 1
 40  RAND
 50  LET n1 = INT(RND * 9 + 1)
 60  LET n2 = INT(RND * 9 + 1)
 70  LET n3 = n1 * n2
 80  CLS
 90  PRINT n1;"*";n2;"=";
100  INPUT v
110  PRINT v
120  IF v <> n3 THEN GOTO 80
130  LET k = k + 1
140  IF k > 100 THEN STOP
150  PRINT  "OK ... next question"
160  PAUSE 200
170  GOTO 50
```

Example of a result

```
4 × 3 =
12                                          (entered by user)
OK...next question
6 × 8 =
etc.
```

Lines **10** and **20** will not cause any problems. Line **30** assigns the initial value 1 to the 'counting variable' k. Every time a correct answer is given this value will be incremented by 1 (line **130**). Line **40** introduces **RAND** in order to obtain different random numbers. Lines **50** and **60**

assign two whole numbers between 1 and 9 to n1 and n2. These numbers are multiplied and assigned to n3 (line 70). At this point the initial work is done and now the question must be displayed. First the screen is cleared using the Clear Screen statement:

```
CLS
```

Then the text is displayed

```
number 1 * number 2 =
```

for instance

```
6 * 7 =
```

Obviously an answer must be entered (line 100) and this must be compared with the correct answer n3 (line 120). If the answer is wrong, the question is displayed again (GOTO 80). If on the other hand the answer is correct the computer proceeds with line 130. The counting variable k is incremented (line 130) and if k exceeds 100 our lesson comes to an end (STOP). This means that 100 different questions are displayed. As long as k <= 100 the computer will proceed with lines 150, 160 and 170. First the text OK...next question is printed, followed by a pause of about 4 seconds. This pause is indicated in order to simulate (pretend) that the computer is thinking about the next question. Such little tricks seem to make a computer behave like a human being!

Incidentally, our program contains a flaw. A new question could turn out to be the same as the previous question — you never know what those crazy random generators will produce. If this error irritates you, simply insert an IF instruction to check this unpleasantry.

The main point illustrated in this program is the construction

```
 30  LET k = 1
 ..  ...
 ..  ...
 ..  ...
 ..  ...
 80  ...
 ..  ...
130  LET k = k + 1
140  IF k > 100 THEN STOP
 ..  ...
```

This construction is used to indicate that exactly 100 'original' questions must be generated. Such constructions are often met in programs and for this reason we will introduce a new programming tool, the FOR...NEXT statement.

The FOR...NEXT statement

The form of a FOR...NEXT statement is

```
FOR name of variable = initial value TO end value
```

The variable indicated after FOR is called the counting variable (or control variable). Sinclair BASIC *allows us to use only one letter to indicate a counting variable.* A number of lines are placed after the line that starts with FOR, ending with

```
NEXT name of counting variable
```

For instance:

```
20  FOR K = 1 TO 100
    ...
    ...   number of lines
    ...
50  NEXT K
```

The line that starts with FOR indicates for which successive values of K the lines in between (until NEXT K) will be repeatedly executed.

A very simple program illustrates the FOR...NEXT statement:

```
10  PRINT  "BEGIN"
20  FOR K = 1 TO 5
30  PRINT K,
40  PRINT K * K
50  NEXT K
60  PRINT  "END"
```

Result

```
BEGIN
1              1
2              4
3              9
4             16
5             25
END
```

Note how lines 30 and 40 are repeatedly executed since

```
initial value:        K = 1
initial value + 1:    K = 2
    .                   .
    .                   .
    .                   .
up to:                K = 5
```

The next program shows another example:

Program

```
10  LET sum = 0
20  FOR k = 1 TO 1000
30  LET square = k * k
40  LET sum = sum + square
50  NEXT k
60  PRINT "sum = ";sum
```

*Result**

```
    sum = 3.338335E + 8
or  SUM = 333833500                                (ZX81/TS1000)
```

Line 10 assigns an initial value 0 to sum. With Sinclair BASIC this is a must! Line 20 introduces our FOR...NEXT statement. Lines 30 and 40 have to be executed for k = 1,2...1000. Line 30 assigns the square of k to the variable square and this value is successively added to sum. Finally, this results in the sum of the squares 1...1000 being calculated! Line 60 ensures the result is printed.

The STEP instruction

We may also indicate the size of steps by which the counting variable has to be changed. A step may even be negative. Consider the next example:

Example

```
10  FOR k = 6 TO 1 STEP -2
20  PRINT k
30  NEXT k
40  PRINT "end"
```

Result

```
6
4
2
end
```

The expression STEP −2 specifies that k has to be decremented (reduced) each time by 2. The last value printed is 2 since the next value 2 − 2 = 0 is out of the range (less than 1). Notice that the 'end value' 1 is in fact never assigned to k and therefore it seems better to speak of the 'boundary value' instead of 'end value'.

* ZX81/TS1000 owners now use the FAST mode.

The initial, boundary and step values may all be specified by means of an expression.

Example

```
10 LET A = 10
20 LET B = 5
30 FOR K = B TO A + B STEP A/B
40 PRINT K
50 NEXT K
```

Result

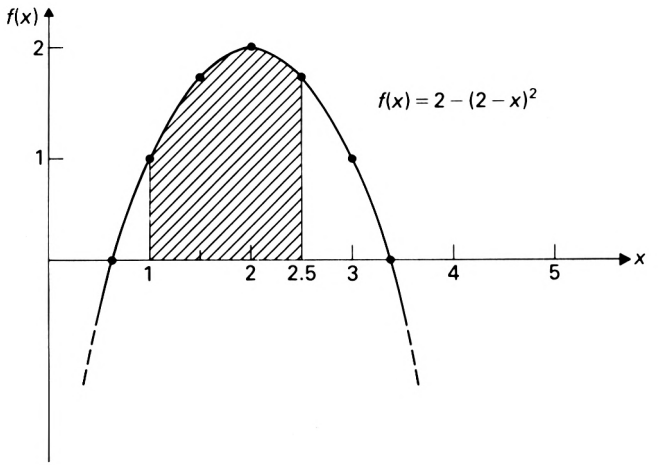
```
 5
 7
 9
11
13
15
```

The program can be explained easily—just fill in the values of A and B and line 30 becomes

```
30 FOR K = 5 TO 15 STEP 2
```

A mathematical example

The next example shows how the FOR...NEXT statement can be used in order to find the integral of a certain function. Consider the following figure.

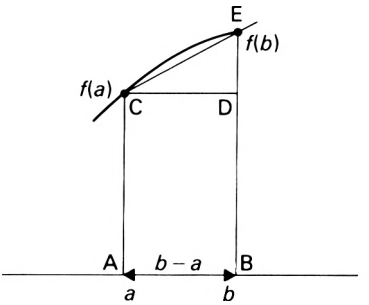


We want to write a program to find the shaded area. This area is mathematically expressed by

$$\text{area} = \int_1^{2.5} [2 - (2 - x)^2] dx$$

The solution is straightforward—split the area into a number of slices and add the areas of each of these slices.

Let us consider one particular slice:



A fair approximation of this area can be obtained by adding the area of the square ABCD

$$(b - a).\mathrm{f}(a)$$

to the area of the upper triangle CDE

$$\tfrac{1}{2}(b - a)[\mathrm{f}(b) - \mathrm{f}(a)]$$

Hence the total area is

$$(b - a).\mathrm{f}(a) + \tfrac{1}{2}(b - a).\mathrm{f}(b) - \tfrac{1}{2}(b - a).\mathrm{f}(a) = \tfrac{1}{2}(b - a).[\mathrm{f}(a) + \mathrm{f}(b)]$$

or in short

$$\tfrac{1}{2}\Delta[\mathrm{f}(a) + \mathrm{f}(b)] \qquad \text{where } \Delta = b - a$$

If the x-axis is divided into $(n-1)$ equal parts and the value of the function at each point, $1,2\ldots(n-1),n$, is given by

$$\mathrm{f}_1,\mathrm{f}_2,\ldots\mathrm{f}_{n-1},\mathrm{f}_n$$

The total area becomes the sum of all slices:

$$\tfrac{1}{2}\Delta(\mathrm{f}_1 + \mathrm{f}_2) + \tfrac{1}{2}\Delta(\mathrm{f}_2 + \mathrm{f}_3) + \tfrac{1}{2}\Delta(\mathrm{f}_3 + \mathrm{f}_4)\ldots\tfrac{1}{2}\Delta(\mathrm{f}_{n-1} + \mathrm{f}_n)$$

or

$$\tfrac{1}{2}\Delta(\mathrm{f}_1 + \mathrm{f}_2 + \mathrm{f}_2 + \mathrm{f}_3 + \mathrm{f}_3 + \mathrm{f}_4 \ \ldots\ \mathrm{f}_{n-1} + \mathrm{f}_n)$$

or

$$\tfrac{1}{2}\Delta(\mathrm{f}_1 + 2\mathrm{f}_2 + 2\mathrm{f}_3 + \ \ldots\ 2\mathrm{f}_{n-1} + \mathrm{f}_n)$$

This formula leads directly to the program to be written. In other words the formula represents our algorithm. Note that all the values $f_2 \ldots f_{n-1}$ must be multiplied by 2. The initial and final values f_1 and f_n are not multiplied by 2.

Program

```
 10  INPUT "left boundary =";l
 20  INPUT "right boundary =";r
 30  LET par = l
 40  LET func = 2 - (2 - par) * (2 - par)
 50  LET f1 = func
 60  LET par = r
 70  LET func = 2 - (2 - par) * (2 - par)
 80  LET fn = func
 90  INPUT "number of points =";n
100  LET del = (r - l)/(n - 1)
110  LET sum = 0
120  FOR k = 2 TO n - 1
130  LET x = l + (k - 1) * del
140  LET par = x
150  LET func = 2 - (2 - par) * (2 - par)
160  LET fi = func
170  LET sum = sum + 2 * fi
180  NEXT k
190  LET sum = sum + f1 + fn
200  PRINT "integral ="; sum * 0.5 * del
```

Result

```
left boundary = 1
right boundary = 2.5
number of points = 50
integral = 2.6247657
```

The two boundaries are first assigned to `l` and `r`. We have used the `INPUT` construction of the ZX Spectrum. Owners of a ZX81/TS1000 have to write

```
 5  PRINT "LEFT BOUNDARY = "
10  INPUT L
```

Lines **30**–**50** result in the value of `f1` and lines **60**–**80** result in the value of `fn`. Note how the function is calculated. In this way the *same instruction* can always be used to calculate the value of the function. This instruction is an exact copy of the formula

$$2-(2-x)^2 = 2-(2-x)(2-x)$$

If you want to use the program for another function you only have to change lines **40**, **70** and **150**. Later we will see how subroutines can be of help in such constructions. Line **90** enters the number of points n and line **100** defines the width of each slice. The initial value of sum is set to zero in line **110**. Lines **120**–**180** describe a **FOR...NEXT** statement. The values f_2 to f_{n-1} (multiplied by 2) are calculated and added to sum. Finally f_1 and f_n are also added to the sum (line **190**) and the result is printed.

At the end of this exercise we should point out that the program has a serious disadvantage — you are not informed how accurate your solution is. One way of solving this problem is to take more and more steps and look at how the answer changes.

Final remarks

(a) Nested **FOR...NEXT** statements

FOR...NEXT statements can be *nested* inside each other. Consider the next program:

```
10  FOR K = 1 TO 2
20  FOR N = 1 TO 3
30  PRINT K,N
40  NEXT N
50  NEXT K
```

Result

```
1             1
1             2
1             3
2             1
2             2
2             3
```

For each value of K the inner loop **FOR N = 1 TO 3...NEXT N** is executed. Note that the inner loop does not cross into the outer loop. Constructions like

```
10  FOR K = 1 TO 2 ┐
20  FOR N = 1 TO 3 │┐
30  PRINT K,N      ┘│
40  NEXT K          ┘
50  NEXT N
```

in which two loops cross into each other are not allowed.

(b) **GOTO** statements and **FOR...NEXT** statements

FOR loops can be jumped out of, for example by a GOTO statement, before the counting variable has reached the boundary value. Consider the next program:

```
10  FOR K = 1 TO 10
20  IF K = 3 THEN GOTO 40
30  NEXT K
40  PRINT K
```

Result

```
3
```

It is obvious that the FOR...NEXT statement is terminated when K becomes 3. Note that jumping into a FOR...NEXT statement is not allowed.

CHAPTER 10

Arrays

In all our examples so far, we have used simple variables such as X, A, B and SUM. However there are many problems where we have to deal with a list of numbers, for example 100 numbers. Of course we could use 100 different names, as we are used to. However, BASIC provides an interesting new way of introducing variables and this way is of especial interest if we are dealing with lists such as these.

The DIM statement

Consider the next program:

```
10  DIM A(100)
20  FOR K = 1 TO 100
30  INPUT A(K)
40  NEXT K
```

Line 10 shows a DIM statement. The word DIM means '*dim*ension'. The effect of this statement is that the computer reserves space for 100 variables. These variables are

```
A(1),A(2),.....A(100)
```

Note that each variable consists of a general name (A) and a number between brackets. Once the DIM statement is executed we can use these variables in the usual way. For example we can write

```
    LET A(3) = 5
```

and `LET A(1) = A(3) + 5`

Line 20 introduces a FOR...NEXT loop starting by setting K to 1. The third line defines an INPUT statement:

```
30  INPUT A(K)
```

We must enter a number and during the first loop (K = 1) this number will be assigned to A(1). Now the loop starts with K = 2 and again a number has to be entered and this number is assigned to the variable A(2). In the same way the third number is assigned to A(3), the fourth to A(4), etc. Hence finally 100 different numbers are stored. Note how

the expression A(K) is used to indicate successively A(1), A(2), A(3), etc.

The variables A(1), A(2), A(3), . . . are sometimes called *subscripted* variables. The number between brackets is the subscript and it may be indicated by

- a constant e.g. A(2)
- a variable e.g. A(K)
- an expression e.g. A(K + 2)

The list of variables is called the *array*. Arrays are often used together with FOR...NEXT statements.

Using an array to plot a function

The next program shows a realistic use of an array in order to plot a function. First 20 values of the function

$$f(x) = x^2 + \sin(x) \qquad (-1 < x \leq 0)$$

are calculated and assigned to the array F. Then the maximum and minimum values are computed and every value is normalized according to

$$\text{normalized value} = \frac{\text{old value} - \text{minimum}}{\text{maximum} - \text{minimum}}$$

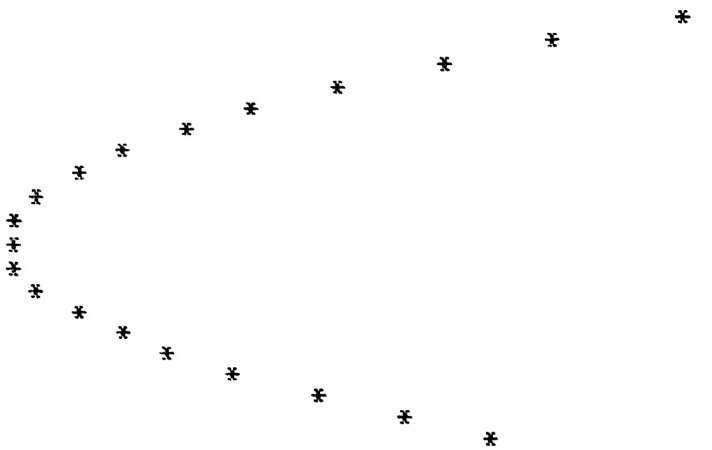
In this way values between 0 and 1 are obtained. Finally these values are multiplied by a constant and the results are used in a TAB function.

Program

```
 10  DIM F(20)
 20  FOR K = 1 TO 20
 30  LET X = - 1 + K/20
 40  LET F(K) = X * X + SIN(X)
 50  NEXT K
 60  REM DETERMINE MAX AND MIN
 70  LET MIN = F(1)
 80  LET MAX = F(1)
 90  FOR K = 1 TO 19
100  IF F(K + 1) > MAX THEN LET MAX = F(K + 1)
110  IF F(K + 1) < MIN THEN LET MIN = F(K + 1)
120  NEXT K
130  REM NORMALIZE
140  FOR K = 1 TO 20
150  LET F(K) = (F(K) - MIN)/(MAX - MIN)
```

```
160  NEXT K
170  REM DETERMINE TAB AND PLOT
180  FOR K = 1 TO 20
190  LET P = 31 * F(K)
200  PRINT TAB P;"*"
210  NEXT K
```

Result



Line **10** introduces the array **F** (the name of an array must be a *single* letter). Lines **20**–**50** show that the values of f(x) according to

$$f(x) = x^2 + \sin(x)$$

have to be calculated for $x = -1$, $x = -1 + \frac{1}{20}$, $x = -1 + \frac{2}{20}$, . . . $x = -1 + \frac{20}{20} = 0$.

The **REM** statement introduces the next part of the program where the maximum and minimum values are computed. In order to avoid memory problems ZX81/TS1000 owners should miss out the **REM** statements.

A **FOR...NEXT** loop is used again. Note how every value of **F(K + 1)** is compared to **MAX** and **MIN**. If **F(K + 1) > MAX** the variable **MAX** is updated to **F(K + 1)** and, in the same way, if **F(K + 1) < MIN** the variable **MIN** is updated. **MAX** and **MIN** are initially **F(1)**.

Now we arrive at the third **FOR...NEXT** statement. The values **F(K)** are normalized according to the formula given on page 80. The last **FOR...NEXT** statement (lines **180**–**210**) actually generates the plotted figure. All values are adapted to the maximum number of columns (line **190**). Finally the function **TAB** controls the plotting of the figure. In chapter 13 we will show how we can use the function **PLOT** in order to obtain smoother curves.

Ordering

The next program deals with ordering, in this case with the ordering of numbers. Ordering plays a very important role in computing.

Program

```
10  PRINT "NUMBER OF VALUES = ";
20  INPUT n
30  PRINT n
40  DIM a(n)
50  FOR k = 1 TO n
60  INPUT a(k)
70  NEXT k
80  FOR k = 1 TO n - 1
90  FOR j = 1 TO n - 1
100 LET x = a(j)
110 LET y = a(j + 1)
120 IF x < = y THEN GOTO 150
130 LET a(j) = y
140 LET a(j + 1) = x
150 NEXT j
160 NEXT k
170 PRINT "RESULT"
180 FOR k = 1 TO n
190 PRINT a(k)
200 NEXT k
```

Example of a result

```
NUMBER OF VALUES = 4
3
2
5
4
RESULT
2
3
4
5
```

values that are entered

same values but now in right order

Lines 10–30 will not cause any problems. Line 40 introduces the subscripted variables

```
a(1),a(2),. . . a(n)
```

In our example n = 4 so line 40 introduces

```
a(1),a(2),a(3),a(4)
```

Lines 50–70 indicate that n numbers have to be entered. These numbers are assigned to the variables a(1),a(2), . . . a(n). Lines 80–160 describe the ordering.

In order to understand the key to the solution we shall first discuss the situation when k = 1. All successive values are compared (lines 90–150). If a(j) is less than or equal to a(j + 1) these two values are not interchanged, but if a(j + 1) is less than a(j) both values are interchanged (lines 130–140).

Consider the situation where the last value is the smallest number (in fact the 'worst' situation). After the FOR j = 1 TO n − 1 loop is executed, this value is interchanged so the next-to-last value will now be the smallest number. If we repeat this procedure n − 1 times this number will be the first number. This is controlled by the outer loop FOR k = 1 TO n − 1.

Finally the results are printed (lines 170–200). In fact we can speed up the ordering if we realize that after the first pass (k = 1) the last value will be the largest, because it is successively shifted to the right. This means that next time we need not check the last pair of numbers again. After the third time we need not check the last three numbers, and so on. Hence line 90 can be changed to

```
90  FOR j = 1 TO n - k
```

This method of ordering can also be used if we are dealing with strings because expressions such as

```
IF a$ < b$ THEN...
```

are allowed. The expression

```
a$ < b$
```

becomes true if a$ is first in alphabetical order.

String arrays

Sinclair BASIC also allows string arrays. For instance the expression

```
DIM A$(10,5)
```

will reserve memory for 10 strings with a fixed length of 5 characters. After the DIM statement, strings can be assigned, for example

```
LET A$(1) = "HOUSE"
```

Note that a single string is indicated by a single number between brackets. A string like A$(1) can be represented as

```
A$(1,1)A$(1,2)A$(1,3)A$(1,4)A$(1,5)
```

If a string that is too small is assigned to A$(1) it is padded with spaces and if it is too large it is chopped off.

We can also use the slicing technique. For instance if A$(1) corresponds to HOUSE then

```
A$(1,1)   corresponds to "H"
A$(1)(2 TO 4)   corresponds to "OUS"
A$(1,2 TO 4)   corresponds to "OUS"
```

An expression like

```
DIM A$(10,5)
```

introduces a two dimensional array. It is also possible to introduce arrays with more than one dimension that deal with numbers. For instance

```
DIM a(2,3)
```

introduces the variables

```
a(1,1) a(1,2) a(1,3)
a(2,1) a(2,2) a(2,3)
```

(all these variables are initialed to zero).

Coupled lists

The next program illustrates the use of two lists that are coupled by means of their subscripts. It shows how we can use the computer as a personal telephone directory. The names are stored in array b$ and the telephone numbers in a in such a way that each name b$(k) corresponds to the number a(k).

Program

```
 10  DIM a(4)
 20  DIM b$(4,10)
 30  FOR k = 1 TO 4
 40  READ a(k)
 50  READ b$(k)
 60  NEXT k
 70  PRINT "Name = ";
 80  INPUT v$
 90  PRINT v$
100  FOR k = 1 TO 4
110  LET z$ = ""
120  FOR j = 1 TO 10
130  LET w$ = b$(k,j)
140  IF w$ = " " THEN LET w$ = ""
150  LET z$ = z$ + w$
```

```
160  NEXT j
170  IF v$ = z$ THEN GOTO 190
180  NEXT k
190  PRINT "number = ";a(k)
200  DATA 4312,"Jerry"
210  DATA 5216,"Tom"
220  DATA 2112,"Donald"
230  DATA 2157,"Mickey"
```

Example of a result

```
Name = Tom
number = 5216
```

ZX81/TS1000 owners should replace lines **40** and **50** by

```
40  INPUT a(k)
50  INPUT b$(k)
```

and may skip the **DATA** statements. They may also use a number of **LET** statements:

```
30  LET a(1) = 4312
31  LET b$(1) = "Jerry"
```

We leave it to the reader to work through this program. Only one hint is given; it is not possible to use an expression like

```
IF v$ = b$(k) THEN...
```

since **b$(k)** is always 10 characters long and **v$** usually not. For this reason the lines **110**–**160** are used.

CHAPTER 11

Subroutines

Sometimes a specific series of instructions, called a subroutine, is needed at several places in a program. Of course it would be very useful if we could give this series of instructions a label in order to refer to it simply by indicating that label.

The GOSUB and RETURN statements

BASIC offers this possibility by means of the GOSUB and RETURN statements. A specific series of instructions (the subroutine) is referred to in our program by

```
GOSUB line number
```

The line number indicates the line number of the first instruction of the subroutine. The last statement of the subroutine is always

```
RETURN
```

The expression 'GOSUB line number' is very similar to 'GOTO line number' and in fact, the effect is also quite similar. As a result of the GOSUB statement a jump will be executed to the line number indicated. Compared to the GOTO statement the major difference is that now the computer automatically jumps back to the instruction following the GOSUB statement, once the RETURN statement is encountered.

Example

```
 10  PRINT  " START "
 20  GOSUB  100
 30  PRINT  " 1 "
 40  GOSUB  100
 50  PRINT  " 2 "
 60  GOSUB  100
 70  PRINT  " END "
 80  STOP
100  PRINT  " SUBROUTINE "
110  RETURN
```

Result

```
START
SUBROUTINE
1
SUBROUTINE
2
SUBROUTINE
END
```

Line 20 indicates the GOSUB statement. A jump to line 100 is made. As a result the word SUBROUTINE is printed. The next statement to be obeyed is the RETURN statement. As a result the computer now jumps back to the line number that follows the GOSUB statement, i.e. that 'called the subroutine'. Hence the computer proceeds with line 30. Hereafter a GOSUB statement is again encountered, etc.

After the PRINT instruction at line 70 is obeyed we have to prevent the computer 'jumping' into our subroutine. In this case a STOP statement is used, but we could also use a GOTO statement.

The next example is more realistic. It not only illustrates the subroutine construction but also demonstrates the circumstances in which subroutines are useful.

Our example deals with a well-known game in which we must pit our wits against the computer. The game proceeds as follows.

After the RUN command the screen shows

```
HERE ARE 13 BOXES
YOU MUST TAKE 1, 2 OR 3 BOXES
IF YOU HAVE TO TAKE THE LAST ONE
YOU LOSE
```

After a few seconds this text disappears and the screen shows the boxes and challenges us to play (we shall explain later how this 'miracle' is achieved).

■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■

```
HOW MANY WILL YOU TAKE?
```

Of course we are careful . . . we take 1. The computer answers with

```
OK...I TAKE
```

apparently thinks for a few seconds and selects a number. The screen displays the boxes that are left and again asks us to enter a number. After a while the screen only shows a single box . . . and of course it is our turn. Honesty is the best policy but . . . you never know. We try to enter 0 which of course is not allowed. The computer answers

```
DON'T FOOL ME
YOU LOSE
```

Let us consider how the algorithm (the method) knocks us out. For the computer to win, the last number of boxes has to be 1 and the previous number had to be 5. If 5 boxes are left and you are only allowed to take 1, 2 or 3 boxes, you never can win. Maximally $5-1=4$ boxes are left and the computer takes 3 and wins. Minimially $5-3=2$ boxes are left and in this case the computer takes 1 and also wins. With the same reasoning we can find that the 'magic number' prior to 5 has to be $5+4=9$ and the number prior to 9 has to be 13.

The computer begins by showing you 13 black boxes—this means that the computer will always win since it is your turn! If you take 3 boxes, the computer takes 1 to reach the magic number 9. On the second move the computer will always reach 5 and on the last move the computer will always reach 1 . . . and then it is your turn.

A very simple program to produce this game could be

```
10  PRINT "text that explains what to do"
20  INPUT V
30  LET M = 13 - V
40  PRINT "computer takes"; M - 9
50  INPUT V
60  LET M = 9 - V
70  PRINT "computer takes"; M - 5
80  INPUT V
90  LET M = 5 - V
100 PRINT "computer takes"; M - 1
110 PRINT "oh oh...there's only 1 left"
120 PRINT "you lose"
```

Of course this is a perfectly correct program but is it such a nice program . . .? Let us not argue about the exact meaning of a 'nice program'. From a player's point of view it just means that the computer must resemble a human being; it has to give normal answers, it has to decide whose turn it is, etc. (Perhaps in the year 2000 computers will even look like people!) Let us consider the program once more. We could insert statements between lines **40** and **50**, **70** and **80**, and **90** and **100** in order to instruct the computer to show how many boxes are left. Since these instructions are apparently similar, a subroutine construction seems reasonable.

The next program shows us once more an example of the game with 13 boxes. But this time subroutines are used in order to:

- show the boxes that are left
- simulate the 'thinking of the computer'

The result we have previously discussed is produced by this program:

```
 10  PRINT  "HERE ARE 13 BOXES"
 20  PRINT  "YOU MUST TAKE 1, 2 OR 3 BOXES"
 30  PRINT  "IF YOU HAVE TO TAKE THE LAST ONE"
 40  PRINT  "YOU LOSE"
 50  PAUSE 300
 60  LET M = 13
 70  GOSUB 300
 80  GOSUB 400
 90  PRINT M - 9;"  BOXES"
100  LET M = 9
110  GOSUB 300
120  GOSUB 400
130  PRINT M - 5;"  BOXES"
140  LET M = 5
150  GOSUB 300
160  GOSUB 400
170  PRINT M - 1;"  BOXES"
180  LET M = 1
190  GOSUB 300
200  IF V <> 1 THEN PRINT  "DON'T FOOL ME"
210  PRINT  "YOU LOSE"
220  STOP
300  PAUSE 150
310  CLS
320  FOR K = 1 TO M
330  PRINT  "■ ";
340  NEXT K
350  PRINT AT 10,0;  "HOW MANY WILL YOU TAKE?"
360  INPUT V
370  PRINT V
380  LET M = M - V
390  RETURN
400  PRINT  "OK...I TAKE  ";
410  PAUSE 150
420  RETURN
```

Line numbers indicated by expressions

A rather curious feature of Sinclair BASIC is that we can indicate line numbers in GOTO and GOSUB statements by means of expressions.

For instance we can extend our previous program with

```
45  LET DISPLAY = 300
```

and change all GOSUB 300 statements in

```
GOSUB DISPLAY
```

The computer reads this statement as

```
GOSUB 300
```

Obviously the readability of the program is improved. This method also provides advantages if we want to change the line numbers of a subroutine. All GOSUB statements are updated if the value of the variable after GOSUB is updated! The next program illustrates that expressions can also be used with subroutines.

```
 10  PRINT  "ENTER 1, 2 OR 3"
 20  INPUT N
 30  GOSUB 100 * N
 40  STOP
100  PRINT  "ONE"
110  RETURN
200  PRINT  "TWO"
210  RETURN
300  PRINT  "THREE"
310  RETURN
```

Example of a result

```
ENTER 1, 2 OR 3
2
TWO
```

The key to this program is line 30:

```
GOSUB 100 * N
```

If for instance 2 is entered, the computer obeys

```
GOSUB 200
```

and the screen shows the word TWO.

DEF FN and FN

The ZX Spectrum provides a special way of defining a function. We begin directly with a program:

```
10  DEF FN a(x,y) = 100 * x + y
20  LET b = FN a(2,3)
30  PRINT b
```

Result

```
203
```

Line 10 describes the function. It starts with DEF FN and immediately after FN a single letter is indicated. This letter indentifies the name of the function. After the name we see two names of variables separated by a comma and enclosed in brackets. These variables are only used to define the function. In order to understand the construction we first look at line 20. In this line a value is assigned to b and this value is obtained by calling the function FNa. A substituting mechanism now comes into action. The computer reads line 20 as

```
LET b = 100 * 2 + 3
```

Note how line 10 is used to replace FNa(2,3) by 100 * 2 + 3 and note also that x is replaced by 2 and y by 3.

We may also define string functions as is illustrated by the next program:

```
10 DEF FN a$(x$) = x$(2 TO 4)
20 PRINT "enter a word"
30 INPUT v$
40 LET w$ = FN a$(v$)
50 PRINT w$
```

Example

```
enter a word
COMPUTER
OMP
```

We leave it to the reader to 'solve' this little word destroyer.

CHAPTER 12

Colours and sounds

Colours

Real life is full of colours. We have a blue sky, yellow sand, red bricks, green leaves, etc.

The ZX Spectrum offers a wide variety of applications in the use of colours. If black and white are also considered the ZX Spectrum can display 8 colours, represented by numbers on the keyboard as follows.

Number	*Colour*
0	black
1	blue
2	red
3	purple (magenta)
4	green
5	pale blue (cyan)
6	yellow
7	white

If you take a look at the keyboard you will find these colours above the corresponding numeric keys.

In this chapter we will give an introduction to the use of colours by means of short programs. At the end of the chapter a simple program is given in which several 'tricks' are combined. This program offers a game in which we have to attack different flashing stars.

Here is our first program:

```
10  INPUT  "colour = ";n
20  PRINT INK n; BRIGHT 0;  "■ ■ ■"
30  PRINT INK n; BRIGHT 1;  "■ ■ ■"
40  PRINT INK n;  "■ ■ ■"
```

Line **10** shows an **INPUT** statement. We have to enter a number (**0**–**7**) and this number will be used in the lines following. For instance if we enter **6** we see:

yellow
bright yellow
yellow

Line **20** indicates that a bar (3 successive squares) has to be 'written' with special ink. If `n = 6` then apparently yellow ink is used. Notice how the type of ink is indicated by

```
INK n;
```

This colour statement is part of the **PRINT** statement in this case. The next colour statement is also part of the **PRINT** statement

```
BRIGHT 0;
```

By means of **BRIGHT** a particular brightness is defined. There are two possibilities:

`BRIGHT 0`	corresponds to normal
`BRIGHT 1`	corresponds to 'extra bright'

Line **30** illustrates the extra brightness and line **40** shows that if no brightness is defined then **BRIGHT 0** is taken.

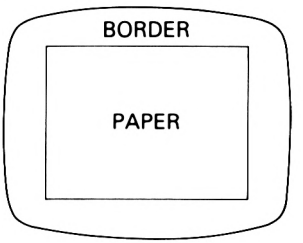
Notice that the rest of the screen is not affected by these instructions. For this reason the effects of our colour statments are called 'local':

> If colour statements are given as part of a **PRINT** or **PLOT** statement, these colour statements only apply to the items to be displayed.

Now we can extend our program with two new statements:

```
5 PAPER 2: CLS
6 BORDER 4
```

After the **RUN** command a rather dramatic effect can be seen. The screen changes into a coloured border with a contrasting central part:



The inner part is indicated by means of the term **PAPER** . . . in fact it is the paper on which we are, so to speak, writing. In this example we see a red paper and a green border. Remember that 2 corresponds to red hence

```
PAPER 2
```

leads to red paper, and in the same way

```
BORDER 4
```

leads to a green border.

Even if the program has stopped the border stays green and the paper stays red. Obviously these colour statements are more powerful than the colour statements that were indicated in the PRINT statements. To return your ZX Spectrum to its original state just press

```
NEW
```

The next program illustrates the difference between *local* and as we shall call it *global* colour statements.

```
10 PRINT "normal"
20 PRINT INK 2; "red letters"
30 PRINT "again normal"
40 INK 4
50 PRINT "green letters"
60 PRINT PAPER 6; "yellow paper"
```

After the RUN command the screen shows:

```
normal                    (black letters on white paper)
red letters                 (red letters on white paper)
again normal              (black letters on white paper)
green letters            (green letters on white paper)
yellow paper          (green letters on yellow paper)
```

If a LIST command is now given we shall see a green program! In order to return to the original situation we give the command

```
INK 0
```

and the LIST command again. Now the listing will appear in black characters. Line 10 results in normal black characters on a white screen (paper). Line 20 results in red letters on a white paper and line 30 shows that the ink colour is black again. Hence if INK is placed *after* PRINT, its effect is only with respect to that particular PRINT statement. In this case INK obviously has a local effect. Line 40 shows an INK statement as a single statement. From now on our computer will use green ink for all the items to be printed (and plotted). So now the effect is not local, but as usually indicated it is global. Line 60 shows PAPER after PRINT and again the effect is local. Only the paper on which the text "yellow paper" is printed, is yellow. The ink colour continues to stay green and we saw that even after the LIST command the ink colour was green.

The next program shows you a board with randomly coloured fields:

```
10 FOR k = 1 TO 32 * 22
20 LET n = INT (RND * 7 + 1)
30 PRINT INK n; "■";
40 NEXT k
```

Line 20 forms the key to this program. Random values (1–7) will be assigned to n and these values are used to indicate the colour of the ink (line 30).

The items INK, PAPER, FLASH, etc. can be placed in any order, for instance the lines

```
10 PRINT INK 2; FLASH 1; "HELLO"
20 PRINT FLASH 1; INK 2; "HELLO"
```

will result in the same flashing, red HELLO.

OVER

If you have ever examined a Swedish text, you may have been puzzled by all those diagonal dashes through the letter o. By means of the item OVER 1 we can compose such an ø.

```
10 PRINT AT 0,0; "O"
20 PRINT OVER 1; AT 0,0; "/"
```

Result

ø

As a result of line 20 the dash will be printed over the letter O. Note that 1 is placed after the term OVER. If we had used the digit 0 after OVER only / will appear. Hence with 0 the OVER effect will not occur, it is as if we switched "OVER" off. In a previous example we also used the expression FLASH 1. In addition to FLASH we can indicate a 1 and a 0, where 1 indicates 'on' (FLASH becomes effective) and 0 indicates 'off'.

Interesting effects occur if we use OVER in combination with colour. In order to illustrate this we can enter the next program:

```
10 FOR k = 1 TO 20
20 PRINT INK 2; AT k,10; "■"
30 NEXT k
40 FOR k = 1 TO 20
50 PRINT INK 6; AT 10,k; "■"
60 NEXT k
```

Now we will observe a cross of two coloured bars where the yellow bar clearly crosses the red bar. But if we change line 50 to

```
50 PRINT INK 6; OVER 1; AT 10,k; "■"
```

the field where the yellow bar crosses the red will show the colour of the paper (white).

What colours are used?

In many games we want to know if a certain field is flashing with a certain colour. This is the point where the ATTR function may help you. The function

```
ATTR (x,y)
```

results in presenting the colour information of field x,y according to the following formula:

```
(128 × FLASH) + (64 × BRIGHT) + (8 × PAPER) + INK
```

where FLASH and BRIGHT can have the values 1 and 0, and PAPER and INK indicate the colours.

For instance, if the field indicated by the line number 3 and the column number 4, is flashing

```
10 PRINT FLASH 1; AT 3,4; "■"
```

then the line

```
20 PRINT ATTR (3,4)
```

will result in (128 × 1) + (64 × 0) + (8 × 7) + 0 = 184.

The following program shows how all the attributes of a certain field can be determined. The original listing as it appears on a ZX printer is used. The advantage of using original listings is to minimize the risk of printer's errors occurring. In the rest of the book we shall use original listings of all relatively long and complicated programs.

Program

```
  10 PRINT "ink=";: INPUT i: PRI
NT i
  20 PRINT "flash=";: INPUT f: P
RINT f
  30 PRINT "paper=";: INPUT p: P
RINT p
  40 PRINT "bright=";: INPUT b:
PRINT b
  50 PRINT INK i; PAPER p; FLASH
 f; BRIGHT b;AT 5,5;"■"
  60 PRINT AT 7,0;"k  at";TAB 6;
"flash";
```

```
  70 PRINT TAB 13;"bright";
  80 PRINT TAB 21;"paper  ink";
  90 FOR k=3 TO 7
 100 LET at=ATTR (5,k)
 110 PRINT AT 5+k,0;k;TAB 3;at;
 120 LET fl=INT (at/128): LET at
=at-128*fl
 130 PRINT TAB 8;fl;
 140 LET br=INT (at/64): LET at=
at-64*br
 150 PRINT TAB 15;br;
 160 LET pa=INT (at/8): LET at=a
t-8*pa
 170 PRINT TAB 23;pa;
 180 LET in=at
 190 PRINT TAB 28;in
 200 NEXT k
```

Example of a result (also using the original listing)

```
ink=3
flash=1
paper=6
bright=1
```

```
k   at  flash   bright   paper   ink
3   56    0       0        7     0
4   56    0       0        7     0
5   243   1       1        6     3
6   56    0       0        7     0
7   56    0       0        7     0
```

Lines 10–40 are used to enter the values for INK, FLASH, PAPER and BRIGHT. Line 50 causes a square to be printed according to the attributes given in the previous lines. Lines 60–80 cause the printing of of a heading for our table. Line 90 starts a FOR...NEXT statement. In this way the attributes of the fields

5,3 5,4 5,5 5,6 and 5,7

will be determined.

Note that something is only printed on field 5,5 (line 50); all other fields are white. Lines 100, 120, 140, 160 and 180 show the recipe to determine the different values of the attributes. Lines 110, 130, 150, 170 and 190 are used to print the calculated values. The table shows that field 5,5 (k = 5) gives the values previously entered, whereas all the other fields yield constant values. A white (paper = 7) field results in a value 56 as FLASH and BRIGHT are 0 and the colour of the ink although not used in that particular field is 0 (black).

INVERSE

It is also possible to print text in what is called *inverse* video mode. Try for instance:

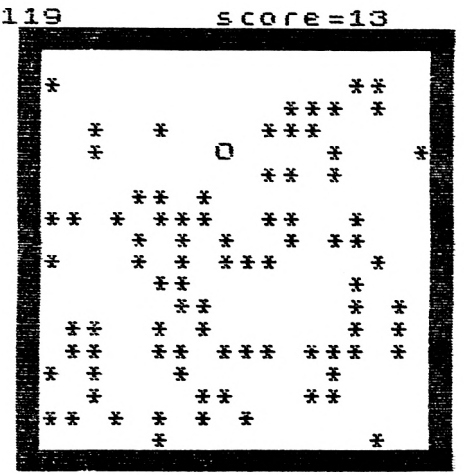
```
10  PRINT  "NORMAL"
20  PRINT  INVERSE 1;  "INVERSE"
30  PRINT  INVERSE 0;  "NORMAL"
```

Result

Line 20 generates inverse video characters resulting in the colours of paper and ink being changed.

Time for a game: FLASHING STAR!

The next game, FLASHING STAR, uses almost all the tricks we have discussed so far. The program generates a red square and more and more stars appear. Some stars flash and some shine with a constant colour. There is also a circle which represents your space module.



You can control your space module by means of the keys **5, 6, 7** and **8** and your task is to destroy as many flashing stars as possible. But beware of the constant shining stars . . . they can kill you. The illustration given in this book is rather poor since it does not show the marvellous colours that are produced. Each time you destroy a flashing star your score is increased. The number on the left side indicates the number of steps you have made. You are allowed only 199 steps after which the game automatically stops.

Program: FLASHING STAR

```
  10 CLS
  20 LET c=0: LET sc=0
  30 INK 2
  40 FOR k=1 TO 20
  50 PRINT AT 1,k;"■";AT 20,k;"■
";AT k,1;"■";AT k,20;"■"
  60 NEXT k
  70 INK 0
  80 LET x=3: LET y=3
  90 PRINT AT x,y;" "
 100 LET x=x+(INKEY$="6")-(INKEY
$="7")
 110 LET y=y+(INKEY$="8")-(INKEY
$="5")
 120 IF x>19 THEN LET x=19
 130 IF y>19 THEN LET y=19
 140 IF x<2 THEN LET x=2
 150 IF y<2 THEN LET y=2
 160 PRINT FLASH (RND+.2); INK R
ND*6;AT RND*17+2,RND*17+2;"*"
 170 IF INT (ATTR (x,y)/128)=1 T
HEN LET sc=sc+1: BEEP .2,1: BEEP
 .5,7
 180 IF INT (ATTR (x,y)/128)=0 A
ND ATTR (x,y)<>56 THEN GO TO 270
 190 PRINT AT x,y;"0": BEEP .02,
-20
 200 LET c=c+1
 210 PRINT AT 0,0;c;TAB 10;"scor
e=";sc
 220 PAUSE 30
 230 IF c<199 THEN GO TO 90
 240 BEEP 2,2
 250 PRINT AT 21,0;"game over"
 260 STOP
 270 PRINT AT x,y;"KILLED"
 280 FOR k=20 TO -40 STEP -1
 290 BEEP .05,k
 300 NEXT k
```

A short description of the program follows:

Line **10**	clears the screen.
Line **20**	determines the initial values of **c** (number of steps) and **sc** (score).
Lines **30–60**	causes a red border to be printed.
Line **70**	changes the colour of the ink to black.
Line **80**	determines the initial position of the space module.

Line 90 shows a PRINT statement that causes a space module to be erased at (x,y). Each time a space module is printed (line 190) the GOTO statement at line 230 will cause a jump to this PRINT statement. Therefore it will erase the space module at its last position and if the space module is displayed at a new position the effect is as if it actually moves!

Lines 100–110 determine the new coordinates of the space module. Note how INKEY$ is used to scan the keys that are momentarily pressed.

Lines 120–150 check that before the space module is displayed at the new coordinates the module does not fly off the screen.

Line 160 generates new stars. RND is used to obtain different attributes.

Lines 170–180 determine if the new position of the space module corresponds to a star. If it corresponds to a flashing star the score is increased (line 170). If it is a constant shining star a 'suicidal jump' to line 270 is made.

Line 190 prints the space module.

Line 200 adds 1 to the number of steps made.

Line 210 prints the number of steps made and the score.

Line 220 determines the speed of the game.

Line 230 determines when the game is over.

Line 240 generates a sound by means of the BEEP statement. This statement will be discussed in the next section.

Lines 250–260 denote the end of the game.

Line 270 denotes an unhappy end.

Lines 280–290 generate a 'chilling' sound effect.

Sound

The ZX Spectrum provides the facility of generating sounds. This is especially entertaining in games programs, for now we can hear balls bouncing, invaders bombing and space modules flying, etc. In order to produce sound the ZX Spectrum has a very small loudspeaker built into it. If you want to hear louder tones you can connect an external audio amplifier to the EAR socket.

The basic statement to generate a sound is BEEP. BEEP is always followed by two values, the first indicates the duration of a tone and the second indicates the pitch:

```
BEEP duration, pitch
```

For instance

```
BEEP 1,0
```

generates a middle C as the value 0 corresponds to the pitch of a middle C. The duration will be one second. If we entered BEEP 1,1 we will hear C-sharp for one second. BEEP 1,2 produces D for one second and BEEP 2,3 produces D-sharp for two seconds. In other words the integers 0,1,2,3...69 correspond to the semitones as found on a piano. Negative numbers correspond in a similar manner to the semitones lower than middle C. For instance −1 corresponds to B, −2 to A-sharp and so on to −60.

The best way to explore the possibilities offered is of course by experimenting. We can use the following program to try out combinations of pitch and duration.

```
10 INPUT "duration = ";d
20 INPUT "pitch = ";p
30 BEEP d,p
40 GOTO 10
```

If you are tired of this program, you can interrupt it, by means of the BREAK command or by entering STOP.

The next program is based on the previous one but now we can directly indicate the notes by capitals (first use the CAPS LOCK key)

C D E F G A B and H (= high C)

in order to play all kinds of simple tunes.

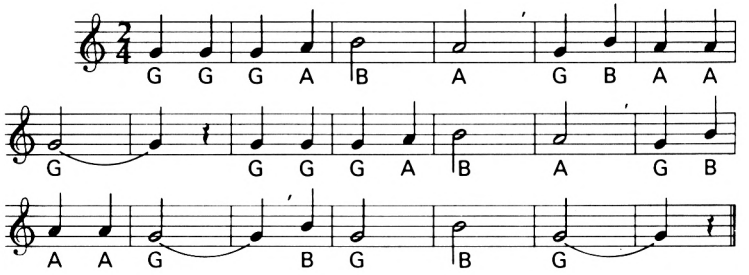
```
 10 PRINT "the SPECTRUM organ"
 20 DIM t(8)
 30 FOR k = 1 TO 8
 40 READ t(k)
 50 NEXT k
 60 INPUT "duration = ";d
 70 LET p$ = INKEY$
 80 LET c = CODE p$
 90 IF (c < 65) OR (c > 72) THEN GOTO 70
100 PRINT p$;
110 LET pitch = t(c - 64)
120 BEEP d,pitch
130 GOTO 70
140 DATA 9,11,0,2,4,5,7,12
```

Line 20 reserves space for an array in order to indicate eight tones. As a result of lines 30–40, t(1) becomes 9, t(2) becomes 11, . . . t(8) becomes 12. Up to now the meaning of these numbers is probably mysterious, but just watch . . . and listen! As a result of line 70 a character C, D, E, etc. is assigned to p$. Each character is converted to its ASCII code (line 80) and finally this code is used to determine the subscript of t (line 110). For instance if you entered a D, the ASCII code of D which is 68 is assigned to the variable c and line 110 becomes

```
110 LET pitch = t(68-64)
```

In this way `t(4)` is assigned to the variable `pitch`. The variable `t(4)` corresponds to 2 and therefore we shall hear a D (line **120**). The rest of the program speaks for itself; only line **90** may raise some questions. In fact this line tests whether you have entered an appropriate note.

Try for instance the following well-known tune.



Ideas to try

Idea 1 *Moving object*

```
10 FOR k = 0 TO 31
20 PRINT INK RND * 7; AT 10,k; " "; "■"
30 BEEP 0.05,k
40 NEXT k
```

Idea 2 *Laser beam*

```
10 FOR k = 1 TO 31
20 BEEP .01,k
30 NEXT k
```

Idea 3 *Bumble-bee*

```
 10 LET u = 15: LET w = 10
 20 LET a = u: LET b = w
 30 FOR k = 1 TO 100
 40 LET x = - 2 + INT (RND * 3 + 1)
 50 LET y = - 2 + INT (RND * 3 + 1)
 60 LET u = u + x
 70 LET w = w + y
 80 FOR j = 1 TO 10
 90 BEEP .015, RND * 1 + 5
100 NEXT j
110 IF u > 21 THEN LET u = 21
120 IF u < 0 THEN LET u = 0
130 IF w > 31 THEN LET w = 31
```

```
140  IF w < 0 THEN LET w = 0
150  PRINT AT a,b; " "
160  PRINT INK 2; AT u,w; "V"
170  LET a = u: LET b = w
180  NEXT k
```

Idea 4 Stochastic Symphony
(Composed by A. Stolmeÿer).

```
 10 RANDOMIZE
 20 PAPER 0
 30 BORDER 0
 40 CLS
 50 LET T=0
 60 FOR R=0 TO 21
 70 FOR K=0 TO 31
 80 GO SUB 160
 90 NEXT K
100 NEXT R
110 LET T=RND/10
120 LET K=INT (RND*32)
130 LET R=INT (RND*22)
140 GO SUB 160
150 GO TO 110
160 INK RND*8
170 PRINT AT R,K;CHR$ (128+INT
(RND*16));
180 BEEP T,-38+(R+K)+INT (RND*6
)
190 RETURN
```

PART TWO

More advanced programming and machine code; practical interfacing and analog simulation

CHAPTER 13

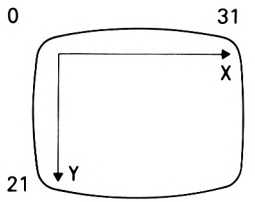
Plotting and drawing

We can draw all kinds of figures on the screen, made up of symbols, lines and curves . . . we can even draw objects like invaders that move. The overwhelming popularity of computers can certainly be attributed to their plotting capability. A famous and ancient Chinese saying is that 'one *picture* says more than a thousand *words*'. Although this statement apparently deals with saving *words* of memory, it primarily indicates the importance of pictures. In the previous chapter we used a powerful statement for plotting all kinds of figures; we refer to the `PRINT AT` statement, for example

```
PRINT AT y,x ; "*"
```

By means of the coordinates `x` and `y` we indicate directly the position where the asterisk will be plotted.

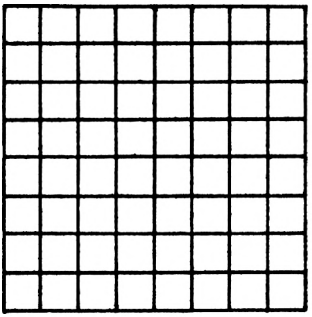
The next figure shows how the screen is divided up.

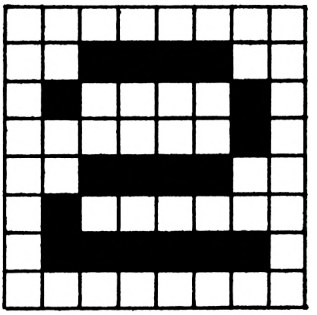


The `x` coordinate can indicate 32 positions (0–31) and the `y` coordinate 22 positions (0–21). Hence we can specify $22 \times 32 = 704$ different positions on the screen.

User-defined graphics

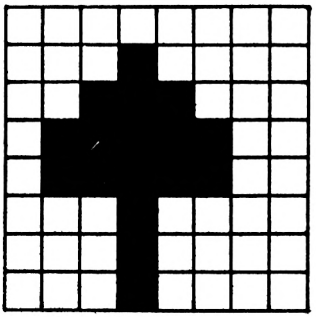
All kinds of figures can be composed if we make use of the graphic symbols. In chapter 6 we discussed how graphic symbols can be entered. The ZX Spectrum provides a special feature which is called *user-defined graphics*. In order to understand this feature, we first have to realize that each character (letter or digit) is composed by filling in squares drawn on an eight by eight grid.





The figure on the left shows an eight by eight grid. By filling in squares we can create all kinds of characters, digits or letters, and also stylistic figures. The figure on the right shows an example of how the digit 2 is composed.

The ZX Spectrum provides the user with the option of defining and displaying such a grid just by indicating one of the keys A to U in a PRINT statement. Such a key represents a *user-defined* graphics character. Let us consider an example. The grid below shows a simple tree that may be used in a game.



Grid with a stylistic tree and a field of corresponding zeros and ones.

Note how the tree is obtained by filling in some of the square elements of the grid. The whole figure is *defined* by eight series of zeros and ones. A zero corresponds to an open square and a one corresponds to a dark square. Now we can couple this pattern, for instance, to the A key.

This program shows the recipe:

```
10  REM TREE
20  FOR x = 0 TO 7
30  READ v
40  POKE USR "A" + x,v
50  NEXT x
60  DATA BIN 00000000, BIN 00010000,
    BIN 00111000, BIN 01111100, BIN 01111100,
    BIN 00010000, BIN 00010000, BIN 00010000
70  PRINT "A"
```

Note that the DATA statement describes exactly the 8 series of zeros and ones indicated in the figure. Every series is preceeded by the term BIN (key B). The character A (lines 40 and 70) is entered in a special way. First we enter the graphics mode (SHIFT 9) and then we enter the capital A.

Below we see the listing as it appears on the screen, immediately after the program is entered.

```
  10 REM TREE
  20 FOR x=0 TO 7
  30 READ v
  40 POKE USR "A"+x,v
  50 NEXT x
  60 DATA BIN 00000000,BIN 00010
000,BIN 00111000,BIN 01111100,BI
N 01111100,BIN 00010000,BIN 0001
0000,BIN 00010000
  70 PRINT "A"
```

After the RUN command we see our 'own' tree

♠

and after the LIST command the letter A in lines 40 and 70 is obviously changed into this tree.

```
  10 REM TREE
  20 FOR x=0 TO 7
  30 READ v
  40 POKE USR "♠"+x,v
  50 NEXT x
  60 DATA BIN 00000000,BIN 00010
000,BIN 00111000,BIN 01111100,BI
N 01111100,BIN 00010000,BIN 0001
0000,BIN 00010000
  70 PRINT "♠"
```

For the time being we shall consider the program simply as a *recipe* since the functions USR and POKE will be discussed later.

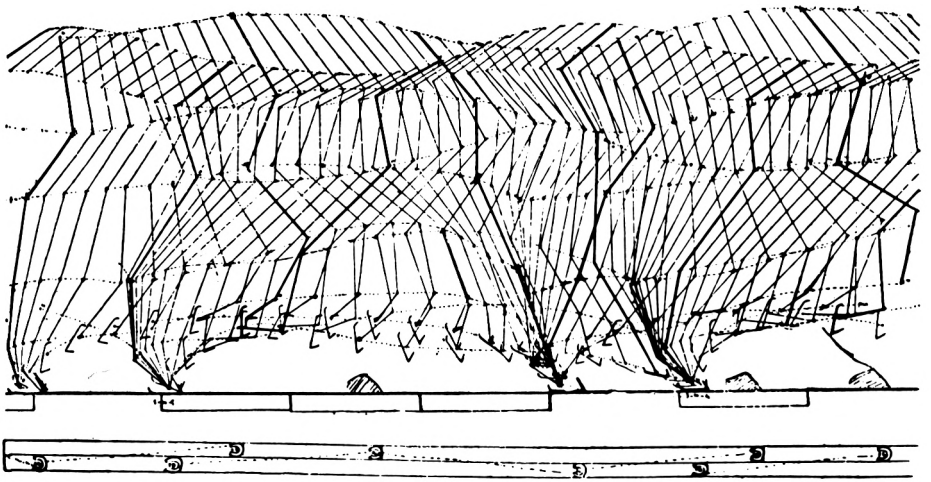
Animation project

In order to show what interesting projects can be carried out, we will create a human figure that *moves* in a natural way.

Our starting point is the biomechanical study of human gait. This study is the only sensible starting point for creating realistic human movements. If you are interested in the history of the study of movements we strongly advise you to read the classic book *Movement*, by E. J. Marey, first published in English in 1895 (reprinted in 1972 by the Arno Press and *The New York Times*).

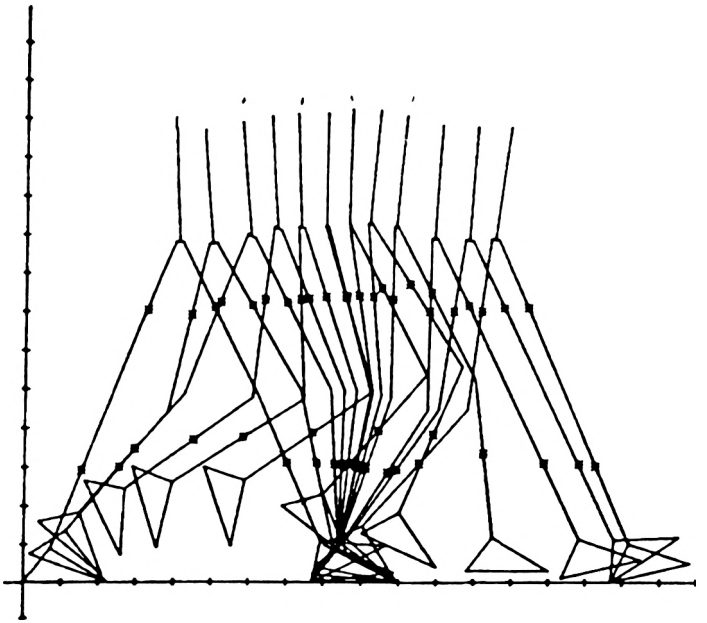
Marey used a photographic gun as illustrated here to study the movement of animals and man. The results of his work were recorded in stick diagrams like the one shown here of the movement of the limbs of a horse at walking pace. This particular diagram is too complicated for us to adapt to write our own 'walking' program. But the idea of using scientific studies of movement will be taken as our starting point.

Marey's photographic gun



Stick diagram of horse at walking pace

The next figure shows a stick diagram compiled by K. Öberg and published by him as 'Mathematical modelling of the human gait: an application of the SELSPOT-system', in *Biomechanics IV*, edited by Nelson and Morehouse, University Park Press, Baltimore.

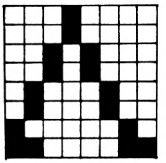


This stick diagram represents the different positions of the legs during normal gait. The triangles represent the feet. Note how one foot remains on the ground while the other is moving. This whole series represents a single step.

Program: Walkman

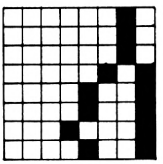
We can use these 'stick diagrams' to create our own walking man program. The stick diagrams (not all) are projected on to an eight by eight grid and the square elements are filled in accordingly.

This is illustrated in the figures on pages 112–13. We see how every position of the legs is indicated by a collection of filled squares. Note how one foot stays in the same position while the other foot and the trunk are shifted forwards. Sometimes two grids are necessary to indicate the legs—to be precise one grid represents most of both legs and one grid (U) represents the toes. In addition we see how the position of the trunk is also indicated by a series of user-defined characters. Each position of the legs corresponds to a position of the trunk. Furthermore on the figure the part of the program that defines each character is listed next to the character. Finally our man is really



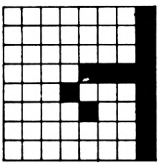
P

```
500 FOR x=0 TO 7
510 READ v
520 POKE USR "P"+x,v
530 NEXT x
540 DATA BIN 00010000, BIN 00010000,
         BIN 00101000, BIN 00101000,
         BIN 01000100, BIN 01000100,
         BIN 10000010, BIN 11000011
```



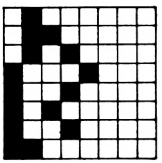
Q

```
600 FOR x=0 TO 7
610 READ v
620 POKE USR "Q"+x,v
630 NEXT x
640 DATA BIN 00000010, BIN 00000010,
         BIN 00000010, BIN 00000101,
         BIN 00001001, BIN 00001001,
         BIN 00010001, BIN 00001001
```



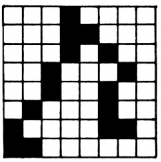
R

```
700 FOR x=0 TO 7
710 READ v
720 POKE USR "R"+x,v
730 NEXT x
740 DATA BIN 00000001, BIN 00000001,
         BIN 00000001, BIN 00001111,
         BIN 00010001, BIN 00001001,
         BIN 00000001, BIN 00000001
```



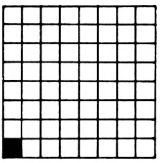
S

```
800 FOR x=0 TO 7
810 READ v
820 POKE USR "S"+x,v
830 NEXT x
840 DATA BIN 01000000, BIN 01100000,
         BIN 01010000, BIN 10001000,
         BIN 10010000, BIN 10100000,
         BIN 10010000, BIN 11000000
```



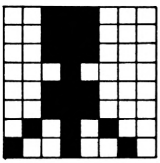
T

```
900 FOR x=0 TO 7
910 READ v
920 POKE USR "T"+x,v
930 NEXT x
940 DATA BIN 00010000, BIN 00011000,
         BIN 00010100, BIN 00100010,
         BIN 00100100, BIN 01000100,
         BIN 10000110, BIN 11000000
```



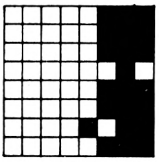
U

```
1000 FOR x=0 TO 7
1010 READ v
1020 POKE USR "U"+x,v
1030 NEXT x
1040 DATA BIN 00000000, BIN 00000000,
          BIN 00000000, BIN 00000000,
          BIN 00000000, BIN 00000000,
          BIN 00000000, BIN 10000000
```



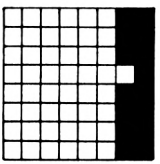
K

```
1100 FOR x=0 TO 7
1110 READ v
1120 POKE USR "K"+x,v
1130 NEXT x
1140 DATA BIN 00111000, BIN 00111000,
          BIN 00111000, BIN 00010000,
          BIN 00111000, BIN 00111000,
          BIN 01010100, BIN 10010010
```



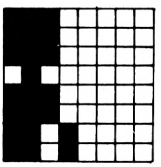
```
1200  FOR x=0 TO 7
1210  READ v
1220  POKE USR "L"+x,v
1230  NEXT x
1240  DATA BIN 00000111, BIN 00000111,
            BIN 00000111, BIN 00000010,
            BIN 00000111, BIN 00000111,
            BIN 00001011, BIN 00000111
```

L



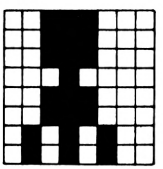
```
1300  FOR x=0 TO 7
1310  READ v
1320  POKE USR "M"+x,v
1330  NEXT x
1340  DATA BIN 00000011, BIN 00000011,
            BIN 00000011, BIN 00000001,
            BIN 00000011, BIN 00000011,
            BIN 00000011, BIN 00000011
```

M



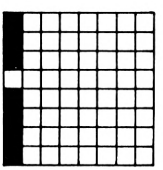
```
1400  FOR x=0 TO 7
1410  READ v
1420  POKE USR "N"+x,v
1430  NEXT x
1440  DATA BIN 11100000, BIN 11100000,
            BIN 11100000, BIN 01000000,
            BIN 11100000, BIN 11100000,
            BIN 11010000, BIN 11010000
```

N



```
1500  FOR x=0 TO 7
1510  READ v
1520  POKE USR "O"+x,v
1530  NEXT x
1540  DATA BIN 00111000, BIN 00111000,
            BIN 00111000, BIN 00010000,
            BIN 00111000, BIN 00111000,
            BIN 01010100, BIN 01010100
```

O



```
1600  FOR x=0 TO 7
1610  READ v
1620  POKE USR "I"+x,v
1630  NEXT x
1640  DATA BIN 10000000, BIN 10000000,
            BIN 10000000, BIN 00000000,
            BIN 10000000, BIN 10000000,
            BIN 10000000, BIN 10000000
```

I

All the user-defined graphics that are used to create our walking man. Beside each symbol the part of the program is also written that defines the character. All the sub-programs are part of the program **WALKMAN**. Characters **P, Q, R, S,** and **T** deal with the legs and the characters **K, L, M, N** and **O** deal with the trunk. Character **U** defines a tip of a foot and character **I** defines a part of the trunk. Character **U** is used with **P** and **Q** and character **I** is used together with **M**. The original line **5000** of the program is

```
5000  DATA "P", "Q", "R", "S", "T", "U",
           "K", "L", "M", "N", "I"
```

All these capitals are changed into the graphics that are given in this illustration.

walking by successively displaying the following user defined characters:

K PU → L QU → MI RU → N S → O T → K PU → L QU → etc.

← one step → ← new step →

Each arrow indicates the next collection of characters that is displayed. After the last collection of characters (O/T) the sequence is repeated.

Here is our total program and you can be assured that it really does produce a realistic walking man, although the individual grids seems rather rough. Try it for yourself.

```
  10 INPUT "speed=";sp
  20 DIM a$(12,1)
  30 RESTORE 5000
  40 FOR x=1 TO 12
  50 READ a$(x)
  60 NEXT x
  70 RESTORE
  80 FOR z=1 TO 12
  90 GO SUB 500+(z-1)*100
 100 NEXT z
 110 CLS
 120 LET y=0
 130 LET y=y+1
 140 FOR b=1 TO 3
 150 PRINT AT 9,y;a$(b+6)
 160 IF b=3 THEN PRINT AT 9,y;a$
(b+6);a$(12)
 170 PRINT AT 10,y;a$(b);a$(6)
 180 PAUSE sp
 190 PRINT AT 9,y;"  "
 200 PRINT AT 10,y;" "
 210 NEXT b
 220 LET y=y+1
 230 FOR b=4 TO 5
 240 PRINT AT 9,y;a$(b+6)
 250 PRINT AT 10,y;a$(b)
 260 PAUSE sp
 270 PRINT AT 9,y;" "
 280 PRINT AT 10,y;" "
 290 NEXT b
 300 IF y<30 THEN GO TO 130
 310 STOP
 500 FOR x=0 TO 7
 510 READ v
 520 POKE USR "p"+x,v
 530 NEXT x
 540 DATA BIN 00010000,BIN 00010
000,BIN 00101000,BIN 00101000,BI
N 01000100,BIN 01000100,BIN 1000
0010,BIN 11000011
 550 RETURN
 600 FOR x=0 TO 7
 610 READ v
 620 POKE USR "A"+x,v
 630 NEXT x
 640 DATA BIN 00000010,BIN 00000
010,BIN 00000010,BIN 00000101,BI
N 00001001,BIN 000001001,BIN 000
```

```
10001,BIN 00001001
 650 RETURN
 700 FOR x=0 TO 7
 710 READ v
 720 POKE USR "d"+x,v
 730 NEXT x
 740 DATA BIN 00000001,BIN 00000
001,BIN 00000001,BIN 00001111,BI
N 00010001,BIN 00001001,BIN 0000
0001,BIN 00000001
 750 RETURN
 800 FOR x=0 TO 7
 810 READ v
 820 POKE USR "e"+x,v
 830 NEXT x
 840 DATA BIN 01000000,BIN 01100
000,BIN 01010000,BIN 10001000,BI
N 10010000,BIN 10100000,BIN 1001
0000,BIN 11000000
 850 RETURN
 900 FOR x=0 TO 7
 910 READ v
 920 POKE USR "f"+x,v
 930 NEXT x
 940 DATA BIN 00010000,BIN 00011
000,BIN 00010100,BIN 00100010,BI
N 00100100,BIN 01000100,BIN 1000
0110,BIN 11000000
 950 RETURN
1000 FOR x=0 TO 7
1010 READ v
1020 POKE USR "g"+x,v
1030 NEXT x
1040 DATA BIN 00000000,BIN 00000
000,BIN 00000000,BIN 00000000,BI
N 00000000,BIN 00000000,BIN 0000
0000,BIN 10000000
1050 RETURN
1100 FOR x=0 TO 7
1110 READ v
1120 POKE USR "h"+x,v
1130 NEXT x
1140 DATA BIN 00111000,BIN 00111
000,BIN 00111000,BIN 00010000,BI
N 00111000,BIN 00111000,BIN 0101
0100,BIN 10010010
1150 RETURN
1200 FOR x=0 TO 7
1210 READ v
1220 POKE USR "i"+x,v
1230 NEXT x
1240 DATA BIN 0000111,BIN 000001
11,BIN 00000111,BIN 00000010,BIN
 00000111,BIN 00000111,BIN 00001
011,BIN 0000101
1250 RETURN
1300 FOR x=0 TO 7
1310 READ v
1320 POKE USR "j"+x,v
1330 NEXT x
1340 DATA BIN 00000011,BIN 00000
011,BIN 00000011,BIN 00000001,BI
N 00000011,BIN 00000011,BIN 0000
0011,BIN 00000011
1350 RETURN
1400 FOR x=0 TO 7
```

```
1410 READ v
1420 POKE USR "[graphic]"+x,v
1430 NEXT x
1440 DATA BIN 11100000,BIN 11100
000,BIN 11100000,BIN 01000000,BI
N 11100000,BIN 11100000,BIN 1101
0000,BIN 11010000
1450 RETURN
1500 FOR x=0 TO 7
1510 READ v
1520 POKE USR "[graphic]"+x,v
1530 NEXT x
1540 DATA BIN 00111000,BIN 00111
000,BIN 00111000,BIN 00010000,BI
N 00111000,BIN 00111000,BIN 0101
0100,BIN 01010100
1550 RETURN
1600 FOR x=0 TO 7
1610 READ v
1620 POKE USR "[graphic]"+x,v
1630 NEXT x
1640 DATA BIN 10000000,BIN 10000
000,BIN 10000000,BIN 00000000,BI
N 10000000,BIN 10000000,BIN 1000
0000,BIN 10000000
1650 RETURN
5000 DATA "[graphic]","[graphic]","[graphic]","[graphic]","[graphic]","
","[graphic]","[graphic]","[graphic]","[graphic]","[graphic]","[graphic]"
```


Example of one situation during the walk

N.B. The original line 5000 of the program is

```
5000 DATA "P", "Q", "R", "S", "T", "U", "K
          "L", "M", "N", "I"
```

All capitals are entered in graphics mode.

The largest part of the program deals with the definition of the characters. Line 10 requests the speed. A realistic walking speed is obtained by entering 3. If we enter 0 we can observe all the different positions of our walking man by repeatedly pressing ENTER. The user-defined characters are indicated by means of an array (line 20). Lines 40–100 deal with the definition of the characters. Line 120 indicates the starting position of our pedestrian. By means of two FOR...NEXT statements (lines 140–210 and 230–290) the successive collections of characters are displayed. Note how each collection of characters is erased after a certain pause (lines 180 and 260). In this way a realistic moving pedestrian is created. If our man begins to walk off the screen the program is stopped (lines 300–310).

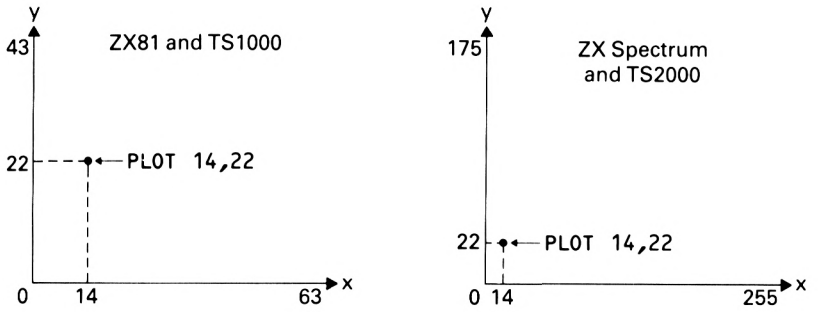
The PLOT statement

In order to obtain drawings with a higher resolution we can use the PLOT statement:

```
PLOT x,y
```

As a result of this statement, the 'part' of the screen indicated by the coordinates x and y is filled with a black square (dot).

The next figure illustrates how the screen is divided up.



As can be seen the resolution is improved with respect to the PRINT AT statement by a factor of 2 (ZX81/TS1000) and a factor of 8 (ZX Spectrum).

By means of the PLOT statement we can draw all kinds of curves as a collection of points. Try for instance the following program:

```
10 FOR x = 0 TO 20
20 LET y = x * x/20
30 PLOT x,y
40 NEXT x
```

As a result a part of a parabola is drawn. For each value of x the function y is calculated according to

$$y = \frac{x^2}{20}$$ (line 20)

Each pair x,y is finally used in the PLOT statement. The values will automatically be rounded off to whole numbers, since each position on the screen has to be indicated by whole numbers. The ZX81/TS1000 uses the statement UNPLOT, for example

```
UNPLOT x,y
```

to erase a point that has been previously plotted. With the ZX Spectrum instead of the UNPLOT statement we use the following construction:

```
PLOT OVER 1; x,y
```

In fact we can also use the items PAPER, INK, FLASH and BRIGHT in a PLOT statement in the same way as was discussed with the PRINT statement.

A very interesting result can be found if we use the indication FLASH. As an illustration change line 30 of the previous program to

```
30 PLOT INK 2; FLASH 1;x,y
```

As a result we observe not only a flashing red parabola but also the 'surrounding squares' will flash. The reason is that colour information is not restricted only to the points of the parabola but also to the square in which this point lies (the screen is divided into a grid of 22×32 squares, see page 35). In order to find out the exact effect of FLASH try changing line 30 to

```
30 PLOT INK 2;x,y
```

Now we shall observe a fine red line!

Program: pseudo 3 dimensional figures

In the previous section we plotted a function $f = f(x)$ or, in other words, we plotted a function where each value of f depends only on the value of x.

Now we ask ourselves if it is also possible to plot functions that depend on two variables

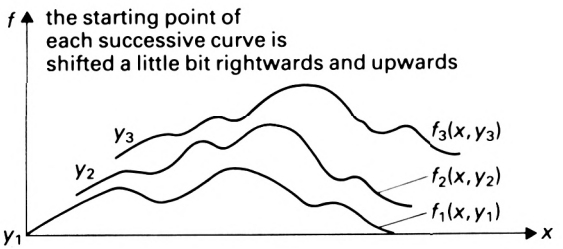
$$f = f(x,y)$$

or, in other words, each value of f depends both on the value of x and the value of y. In fact this is possible by using a pseudo 3 dimensional plotting technique. Results are often amazing!

The 'wrinkle' or trick is to calculate curves $f_1, f_2, \ldots f_n$ for serveral values of y:

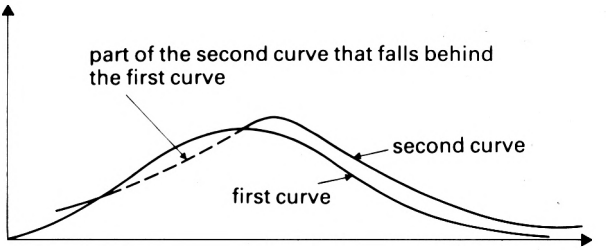
$$f_1 = f(x,y_1) \quad f_2 = f(x,y_2) \ldots f_n = f(x,y_n)$$

and to plot these curves one after the other in the way indicated below.

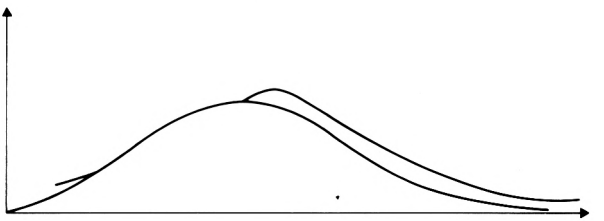


Note how each successive curve is shifted upwards and to the right in order to obtain the impression of a 3 dimensional figure—it is as if we are looking at a landscape with mountains.

Very attractive results can be obtained if we draw the lines very close together, but obviously a problem now arises.



A part of the second curve falls behind the first one (shown in the figure by a dotted line). This part should be ommitted during the plotting in order to obtain a realistic shape.

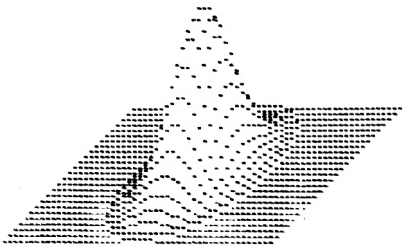


The solution of this 'hidden line' problem is easily achieved. We compare each new vertical or y-value with the previous y-value plotted at the same position on the horizontal x-axis. If the new value is higher it may be plotted, otherwise it is discarded. In order to make this comparison possible we use two arrays. One array stores the computed values and the other contains the last values that have been plotted. Each time a value is plotted the latter array is updated.

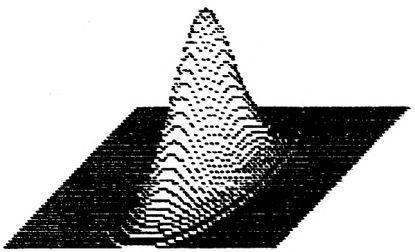
The next program gives an example. The array that stores the newly computed values is `l` and the one that stores the last plotted values is `f`. The program follows exactly the method just described. Line **60** determines that each new curve will be shifted to the right and line **90** determines that each new curve is also shifted upwards. Lines **70–80** determine the function and lines **100** and **110** describe how the two arrays are compared. `l(k)` is eventually plotted, while `f(k)` is eventually updated.

```
  10 INPUT "resolution=";r
  20 DIM f(150)
  30 DIM l(150)
  40 FOR y=1 TO 50 STEP r
  50 FOR x=1 TO 100 STEP r
  60 LET k=x+y-1
  70 LET arg=(x-50)*(x-50)+(y-25
)*(y-25)
  80 LET fxy=60*EXP (-arg/130)
  90 LET l(k)=fxy+y
 100 IF l(k)>=f(k) THEN PLOT k+2
0,30+l(k)
 110 IF l(k)>=f(k) THEN LET f(k)
=l(k)
 120 NEXT x
 130 NEXT y
```

The result in the case where the resolution r of line **10** is taken as 2 is



With a resolution r = 1 the result becomes



This program actually calculates the function

$$f(x,y) = 60\ e^{-\frac{(x-50)^2+(y-25)^2}{130}}$$

which we give you for the sake of completeness . . . it looks aesthetic even if it is involved!

Even on a ZX81 or TS1000 with 1K memory!

The next program shows that this plotting procedure can also be used with a ZX81 or TS1000 with 1K memory. Obviously the resolution is much lower but the result is certainly interesting.

Program

```
  10 DIM F(24)
  20 DIM L(24)
  30 FOR Y=1 TO 10
  40 FOR X=1 TO 12
  50 LET K=X+Y-1
  60 LET A=(X-6)*(X-6)+(Y-5)*(Y-
5)
  65 LET FXY=15*EXP (-A/5)
  70 LET L(K)=FXY+Y
  80 IF L(K)>=F(K) THEN PLOT K+4
10+L(K)
  90 IF L(K)>=F(K) THEN LET F(K)
=L(K)
 100 NEXT X
 110 NEXT Y
```

Result

DRAW and CIRCLE

The ZX Spectrum can plot straight lines, parts of circles and whole circles. We can start with straight lines. The procedure is as follows:

- Indicate a start position with a PLOT statement.
- Use the DRAW statement

  ```
  DRAW x,y
  ```

  where x indicates how many steps in the x direction and y how many steps in the y direction have to be taken to find the end point.

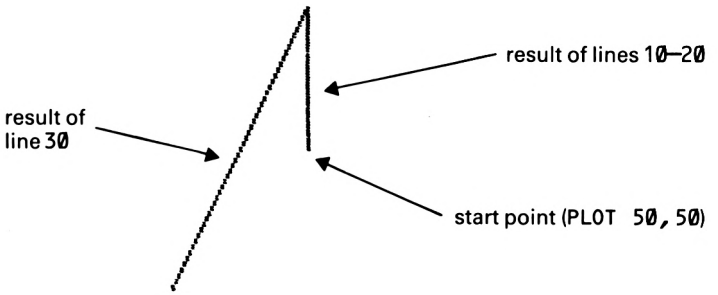
For instance

```
10 PLOT 50,50
20 DRAW 0,50
```

will result in a vertical line from position 50,50 to position 50 + 0 = 50, 50 + 50 = 100. Note that the values x and y actually indicate the increments to find the end point and not the coordinates of the end point itself. This is clearly demonstrated by extending the program with

```
30 DRAW - 50, - 100
```

The start point of the straight line that will be plotted as a result of line 30 is the last point plotted as a result of the previous line. Now we shall observe a line that runs to the 'origin' 0,0.



In the same way as **PLOT** we can also use the items **PAPER, INK, FLASH** and **BRIGHT** in a **DRAW** statement. However to begin with it is advisable to use only the item **INK**.

Program: Brownian motion

The **DRAW** statement opens the door to many interesting applications. Our first example shows you how to simulate Brownian motion, first discovered by the Scots botanist Robert Brown.

Program

```
 10 LET XN=120: LET YN=80: PLOT
XN,YN
 20 LET X=INT ((RND-.5)*40*RND)
 30 LET Y=INT ((RND-.5)*40*RND)
 40 LET XN=XN+X: LET YN=YN+Y
 50 IF XN>255 OR XN<0 THEN LET
XN=XN-X: LET YN=YN-Y: GO TO 20
 60 IF YN>175 OR YN<0 THEN LET
XN=XN-X: LET YN=YN-Y: GO TO 20
 70 DRAW X,Y
 80 GO TO 20
```

Example of a result

Brownian motion

Line **10** determines the start position. By means of lines **20** and **30** random increments to be used in the **DRAW** statement of line **70** are determined. Line **40** determines the end point of each line but before a line is actually drawn, a check is made to ensure it does not run off the screen (lines **50** and **60**).

The above pattern is obtained by drawing random lines connected to each other.

Program: Moire patterns

By plotting many straight lines in a *regular* way, we can produce many attractive patterns. For instance if you place one comb over another and rotate it you will see very interesting changing patterns. These patterns are called Moire patterns after the French term 'la moiré ondé' meaning a watered mohair fabric with a lustrous wavy pattern.

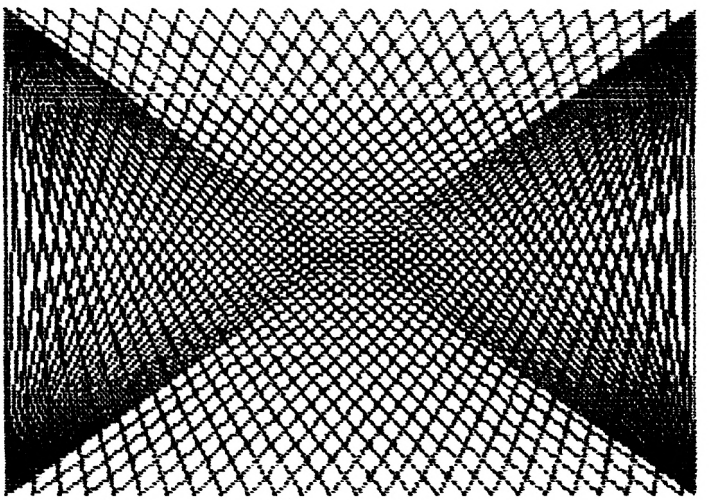
In the next program, lines are drawn from **0,0** and **255,0** to equidistant points on the upper horizontal line and in the same way from **0,175** and **255,175** to the lower horizontal line. The resolution can be chosen by the user.

Two variants of the program are shown. The only difference is the use of the indication **OVER 1** in the **DRAW** statements.

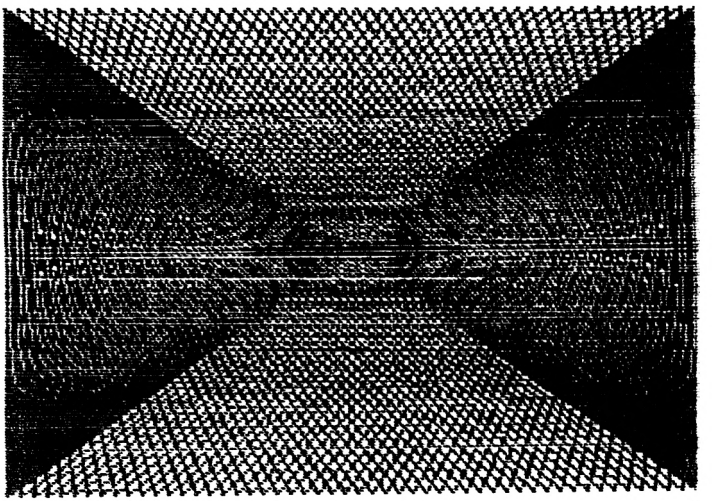
Variant 1

```
  5 INPUT "step=";s
 10 FOR k=0 TO 255 STEP s
 20 PLOT 0,0
 30 DRAW (255-k),175
 40 PLOT 255,0
 50 DRAW -k,175
 60 PLOT 0,175
 70 DRAW k,-175
 80 PLOT 255,175
 90 DRAW -(255-k),-175
100 NEXT k
```

Two results



Resolution (step) = **10**

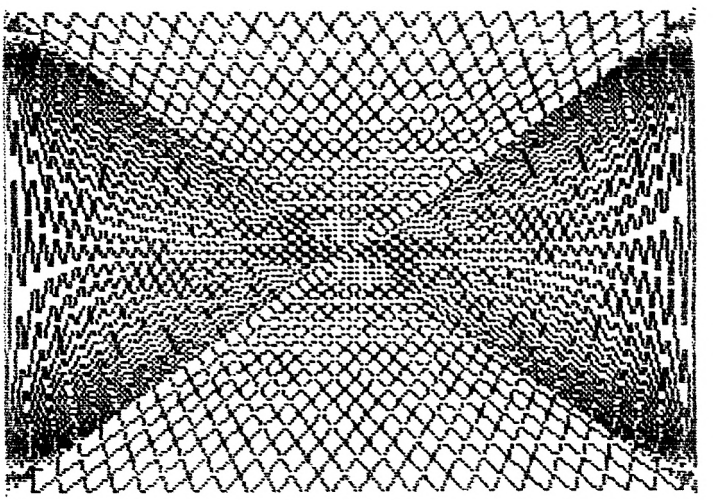


Resolution (step) = 5

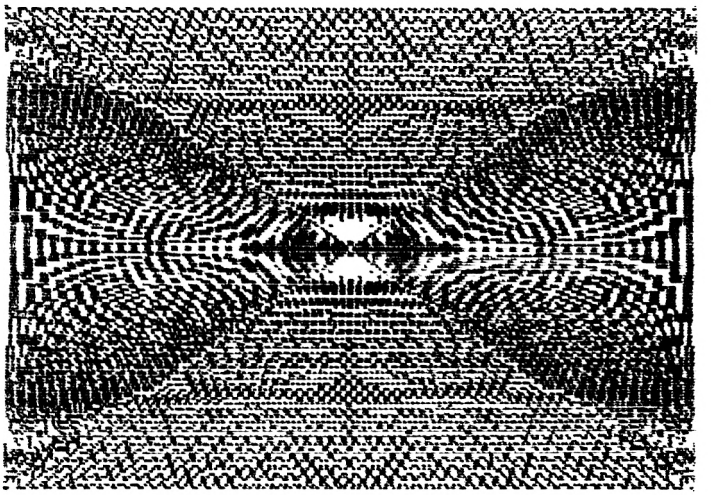
Variant 2

```
  5 INPUT "step=";s
 10 FOR k=0 TO 255 STEP s
 20 PLOT 0,0
 30 DRAW OVER 1;(255-k),175
 40 PLOT 255,0
 50 DRAW OVER 1;-k,175
 60 PLOT 0,175
 70 DRAW OVER 1;k,-175
 80 PLOT 255,175
 90 DRAW OVER 1;-(255-k),-175
100 NEXT k
```

Two results



Resolution (step) = 5



Resolution (step) = 10

Program: Astronomy

My father has an intriguing collection of strange astronomical instruments and simply by looking at them, it is easy to get bitten by the 'astronomy bug'.

The astronomers among us can have many hours of fun with the computer—you can construct your own planetarium or predict eclipses. All the algorithms (rules) necessary for these and other projects can be found in the excellent book *Practical astronomy with your calculator* by Peter Duffett-Smith, Cambridge University Press.

The next program illustrates the power of the `DRAW` statement with respect to the drawing of a celestial map as a function of siderial time (measured by the apparent motion of the stars and constellations). Most constellations are easily drawn by the following procedure:

- Begin with a certain star in a `PLOT` statement.
- Use the coordinates of the other stars to draw the lines that finally form the constellation.

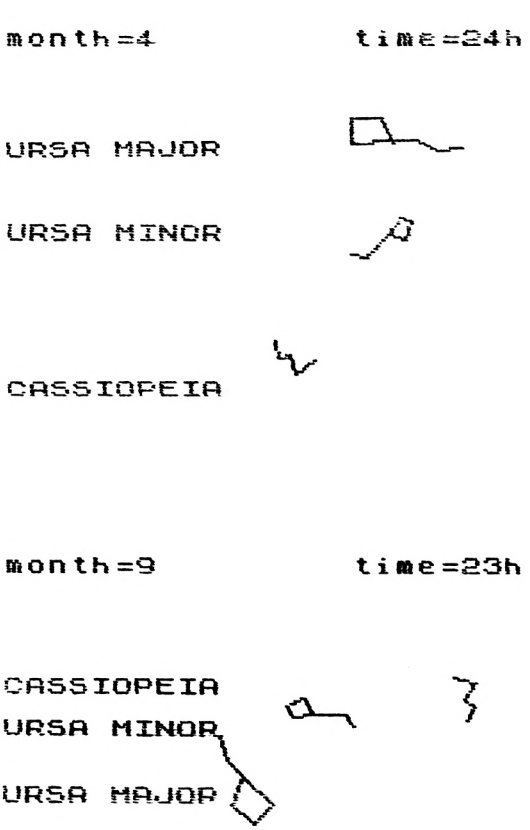
In this program three constellations are drawn, Ursa Major (Great Bear), Ursa Minor (Little Bear) and Cassiopeia. Remember that the Pole-star is one 'end' of Ursa Minor and the constellations turn around this star. The program determines the sky at a particular time which you enter as the month (1–12) and the hour (0–24).

Program

```
10 DIM a(19,2): DIM b(19,2)
20 INPUT "time=";t
25 INPUT "number of month=";nm
30 LET alfm=-nm/12*2*PI
35 LET alft=-t/24*2*PI
40 PRINT "month=";nm,"time=";t
;"h"
50 FOR k=1 TO 19
60 READ a(k,1): READ a(k,2)
70 NEXT k
```

```
  80 REM  new coordinates
  90 FOR k=1 TO 19
 100 REM determine radius and an
gle
 110 GO SUB 400
 120 LET b(k,1)=r*SIN al+128
 130 LET b(k,2)=r*COS al+85
 140 NEXT k
 150 REM plot ursa minor
 160 PLOT b(1,1),b(1,2)
 170 FOR k=2 TO 7
 175 GO SUB 500
 180 NEXT k
 190 DRAW b(4,1)-b(7,1),b(4,2)-b
(7,2)
 200 REM plot ursa major
 210 PLOT b(8,1),b(8,2)
 220 FOR k=9 TO 14
 230 GO SUB 500
 240 NEXT k
 250 DRAW b(11,1)-b(14,1),b(11,2
)-b(14,2)
 260 REM plot cassiopeia
 270 PLOT b(15,1),b(15,2)
 280 FOR k=16 TO 19
 290 GO SUB 500
 300 NEXT k
 310 PRINT AT (11+(11-b(11,2)/8)
),0;"URSA MAJOR"
 320 PRINT AT (11+(11-b(5,2)/8))
,0;"URSA MINOR"
 330 PRINT AT (11+(11-b(16,2)/8)
),0;"CASSIOPEIA"
 340 STOP
 400 LET r2=a(k,1)*a(k,1)+a(k,2)
*a(k,2)
 410 LET r=SQR r2
 420 LET alf0=ATN (a(k,1)/a(k,2)
)
 430 LET al=2*PI+alfm+alft+alf0-
.70
 440 IF a(k,1)<0 AND a(k,2)<0 TH
EN LET al=al+PI
 450 IF a(k,1)>0 AND a(k,2)<0 TH
EN LET al=al+PI
 460 RETURN
 500 LET xp=b(k,1)-b(k-1,1)
 510 LET yp=b(k,2)-b(k-1,2)
 520 DRAW xp,yp
 530 RETURN
 600 DATA 0,1, -5,0.1, -7,-4, -1
0,-10, -12,-17, -16,-17, -16,-10
 610 DATA -25,-49, -17,-45, -8,-
46, -.05,-43, 14,-37, 17,-46, 7,
-50
 620 DATA 0,42, 5,49, 9,43, 13,4
5, 16,40
```

The next figure shows two examples of results. Note how Cassiopeia and Ursa Major turn around the Pole-star. Ursa Major and Ursa Minor are easily recognized by their saucepan shaped patterns. Cassiopeia resembles the letter W.



The program uses two arrays. Array `a` is used to store all the coordinates that are given in the `DATA` lists. These coordinates apply to a particular initial position that corresponds to a map originally used to find the coordinates. Array `b` is used to store the coordinates that correspond to a particular sidereal time. After entering a time a corresponding angle is calculated (lines **20–40**). Lines **50–70** are used to read the coordinates of the different constellations. All values `a(k,1)` apply to the x-coordinates, and `a(k,2)` apply to the y-coordinates. Lines **90–140** determine the coordinates that are used for plotting. By means of a subroutine (starting at line **400**) the polar coordinates are calculated for each star. This facilitates the determination of the new coordinates since only the angle is changed. Lines **400–410** determine the radius. Line **420** determines the original angle and line **430** indicates the new angle.* Lines **440** and **450** add π to the angle if a star is lying in a certain quadrant. The computer proceeds with lines **120** and **130** that determine the new `x` and `y` coordinates. The three constellations are then drawn in a straightforward manner. Note how each constellation starts with `PLOT`. The `DRAW` statement always uses the x-increments (line **500**) and y-increments (line **510**).

* We can adjust the program to a specific geographical position by entering a constant in line **430**.

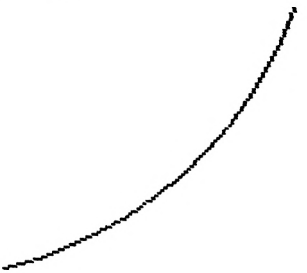
Once you have mastered the program, I am sure you will write extensions in order to display as many constellations as possible and later you may wish to show the movement of planets through the sky—Mars, Venus etc. In short, hours and hours can be spent on the intriguing subject of astronomy.

About circles and parts of circles

One of the remarkable things about DRAW is that it can be used to draw parts of a circle. For example the mini-program

```
10  PLOT 20,20
20  DRAW 100,100,1
```

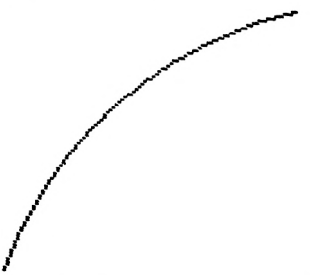
results in



while

```
10  PLOT 20,20
20  DRAW 100,100, - 1
```

results in

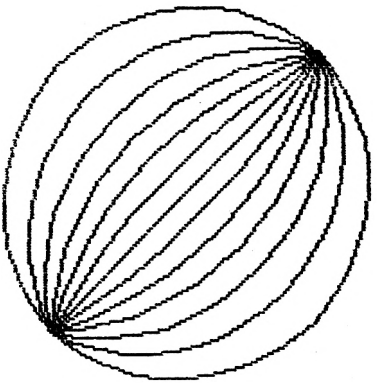


The third value corresponds to an angle. If this value corresponds to π the line describes a semicircle, if it is $\pi/2$ it describes a quarter of a circle and so on.

Would you like to see a beachball?

```
10  FOR k = 0 TO 3 STEP .5
20  PLOT 20,20
30  DRAW 100,100,k
40  PLOT 20,20
50  DRAW 100,100, - k
60  NEXT k
```

Result



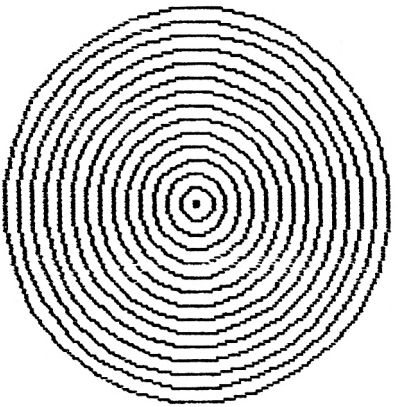
If we simply want to draw pure circles, the **CIRCLE** statement is our choice. It has three parameters, the x and y coordinates of the centre and the radius:

```
CIRCLE x,y,radius
```

The next program gives a simple illustration that may be interesting if you are a darts fanatic.

```
10 FOR k = 1 TO 75 STEP 5
20 CIRCLE 128,87,k
30 NEXT k
```

Result

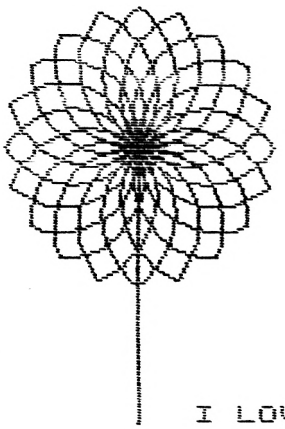


The next program illustrates once more the power of the **DRAW** statement.

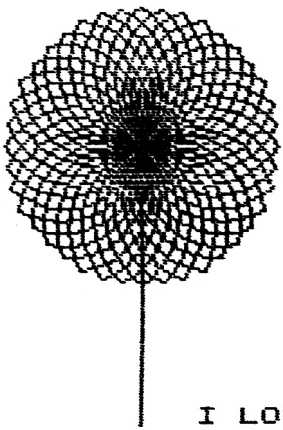
Program: Flowerpower

```
10 INPUT "number of leaves=";l
20 INPUT "radius=";r
30 LET arg=2*PI/l
35 INK 2
40 FOR k=0 TO l-1
50 LET x=r*SIN (k*arg)
60 LET y=r*COS (k*arg)
```

```
 70 PLOT  100,100
 80 DRAW  x,y,2
 90 PLOT  100,100
100 DRAW  x,y,-2
110 NEXT  k
120 PLOT  100,100
130 DRAW  0,-100
140 PRINT AT 21,15;"I LOVE YOU"
```



Two results



Number of leaves = 20
Radius = 50

Number of leaves = 40
Radius = 50

Lines **10–20** speak for themselves. Line **30** determines the angle between the leaves. The **FOR...NEXT** statement of lines **50–120** determines the plotting of the leaves. Finally a straight line is drawn (lines **130–140**) and the text **I LOVE YOU** is printed (line **150**).

Program: The Chinese fan

In a previous section we saw how **ATTR** could be used to find out which plotting directives are used. **ATTR** does not say however *what* is actually plotted or printed on the screen. The ZX Spectrum provides two statements to inform us about what is actually displayed at a given position:

SCREEN$ (x,y) results in the character that is printed on line **x** at column **y**.

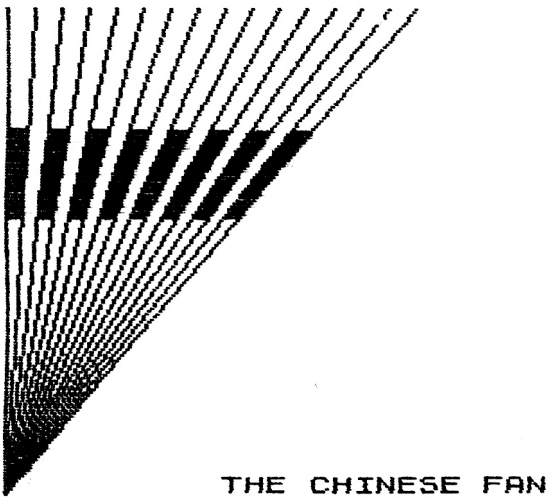
POINT (x,y) results in the value 1 if a point is plotted at pixel **x,y**.

We shall give one demonstration. The following program shows how **POINT** is used to find out whether a line is crossed. If such a point is found a line is drawn until another line is crossed etc. This procedure is repeated several times and in this way we can apparently fill in certain areas.

Program

```
 10 FOR k=150 TO 0 STEP -10
 20 PLOT 0,0
 30 DRAW k,175
 40 NEXT k
 50 FOR y=100 TO 130
 60 LET flag=-1
 70 FOR x=0 TO 255
 80 IF POINT (x,y) THEN LET fla
g=flag*-1
 90 IF flag=1 THEN PLOT x,y
100 NEXT x
110 NEXT y
120 PRINT AT 21,10;"THE CHINESE
FAN"
```

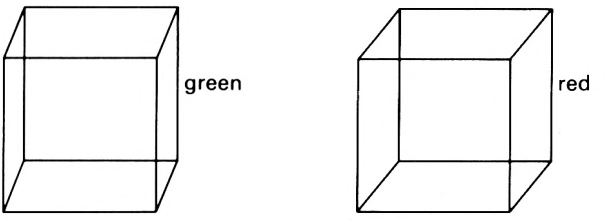
Result



This time we will leave it to the reader to puzzle out the program in detail as an exercise!

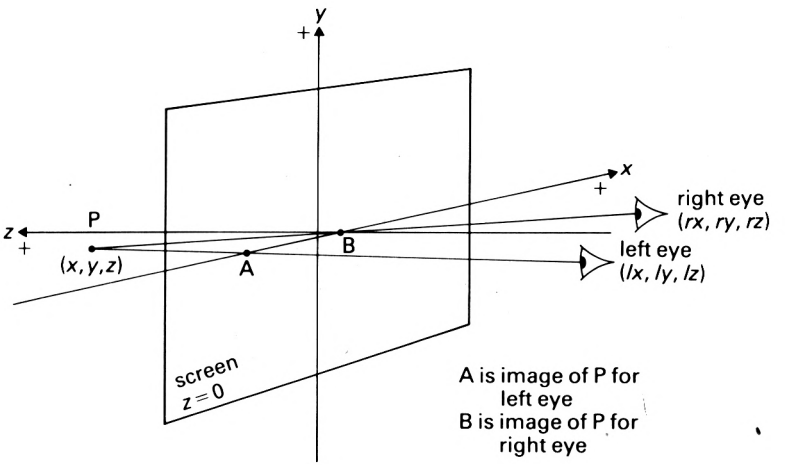
Real 3D

A very interesting application of the plotting statements is to create the illusion of real three dimensional figures. The key to the solution is to use special spectacles with one green and one red glass and two figures (one red, one green) on the screen at the same time. If the red picture shows the image of a three dimensional figure that will be seen by the right eye and the green picture shows the image seen by the left eye, both images will form a three dimensional figure to the viewer wearing the spectacles. As an example consider the two figures that are related to the images of the left and the right eye observing a cube.



Of course the problem arises, 'How can we obtain the images for the left and the right eye?'

In order to answer this question it is easier to draw the following diagram. This diagram shows a screen, two eyes and a point P. Just imagine that the screen is actually the screen of your TV. We ask ourselves how can point P be displayed as a red and a green point on the screen in order to observe the illusion of P in space.

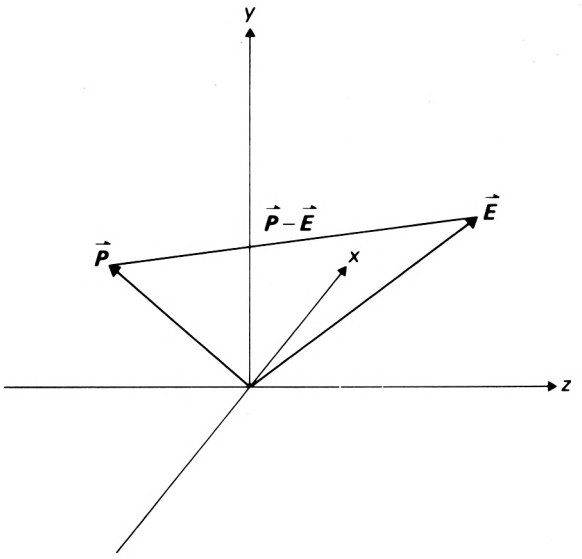


In this diagram we see cartesian coordinates x, y and z, the left and right eye, the point P that we wish to see stereoscopically and the screen that will be observed. The screen is defined by $z = 0$ (the screen lies perpendicular to the z-axis and crosses the origin). Point P is defined by (x,y,z) and finally the positions of the eyes are

left eye: lx,ly,lz
right eye: rx,ry,rz

Point A is the point where the line P → left eye pierces the screen and B is the point where the line P → right eye pierces the screen. This means that A is the green point and B the red point and both points lead to a stereoscopic view of P.

If we can calculate the (x,y) coordinates of A and B (note A and B lie on the screen hence $z = 0$) our problem is almost solved. Consider the next diagram in which two vectors are drawn.



One vector points at P and the second points at one of the eyes, for instance the left eye. The left vector is defined by

$$\begin{pmatrix} px \\ py \\ pz \end{pmatrix}$$

and is denoted by $\vec{P}$. The 'eye-vector' is defined by

$$\begin{pmatrix} lx \\ ly \\ lz \end{pmatrix}$$

and is denoted by $\vec{E}$.

A third vector can be seen, which runs from $\vec{E}$ to $\vec{P}$. This vector can be indicated by the difference of vector $\vec{P}$ and vector $\vec{E}$:

$$\vec{P}-\vec{E}$$

A straight line through $\vec{P}$ and $\vec{E}$ can now be defined as

$$\vec{E}+\lambda(\vec{P}-\vec{E})$$

This formula describes for each of the coordinates of a point on the line the relation between the coordinates of $\vec{E}$ and $\vec{P}$:

x coordinate: $lx+\lambda(px-lx)$
y coordinate: $ly+\lambda(py-ly)$
z coordinate: $lz+\lambda(pz-lz)$

This line pierces the screen at $z = 0$ and in this way λ can be found

$$lz + \lambda(pz - lz) = 0$$

hence $$\lambda = -\frac{lz}{pz - lz}$$

If we now substitute λ the x and y coordinates of point A are easily found:

$$x_A = lx + \left(\frac{-lz}{pz - lz}\right)(px - lx)$$

$$y_A = ly + \left(\frac{-lz}{pz - lz}\right)(py - ly)$$

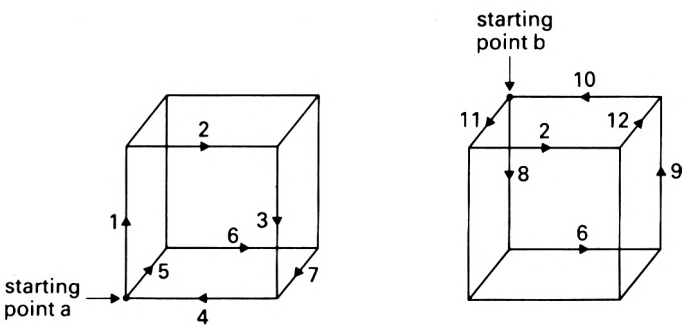
Similarly we can derive the x and y coordinates of point B:

$$x_B = rx + \left(\frac{-rz}{pz - rz}\right)(px - rx)$$

$$y_B = ry + \left(\frac{-rz}{pz - rz}\right)(py - ry)$$

These formulae completely described the solution to our problem.

The full program 'Real 3D' is listed on page 200. In this program a cube is displayed by means of two almost similar frames. The formulae as they are given above are exactly copied in the program.



First the coordinates of starting point a are calculated and then plotted by

```
PLOT xa,ya
```

Now the sides 1, 2, 3, 4, 5, 6 and 7 are drawn by 7 successive **DRAW** statements (see also figure above)

```
DRAW Δx,Δy
```

In each **DRAW** statement the differences **Δx** and **Δy** between the ending and starting point of that particular side are indicated. Next a new

starting point (starting point b, see figure above) is used. This point is again plotted:

```
PLOT xb,yb
```

and then sides 8, 6, 9, 10, 11, 2 and 12 are drawn.

The **DATA** statements describe, for each side of the cube, the x, y and z coordinates of both the starting and the end position. Since there are 12 sides there are also 12 such **DATA** statements. These **DATA** statements are used in lines **600–710** for the calculations of the images on the screen. As can be seen two arrays are used, one array for the storage of x coordinates and one for the storage of y coordinates. The first number of these arrays indicates the side of the cube, the second number (can only be 1 or 2) indicates if the coordinate applies to a starting point (indicated by 1) or an end point of a side (indicated by 2). Finally the last number (can only be 1 or 2) indicates if we are dealing with an image for the left or for the right eye.

The creation of stereoscopic images is really a very interesting field as you can now find out for yourself.

CHAPTER 14

Machine code

Why machine code?

Machine code instructions are the most elementary instructions and they are in fact the only codes that can be understood by the 'brain' of the computer, the central processing unit (CPU). Although the BASIC programs we have previously met suggest that our computer understands BASIC, it is however these BASIC statements converted into machine code that are actually executed. In order to make the conversion, our computer has a memory (ROM) that contains a program entirely written in machine code to do this job. This machine code program is called 'the BASIC interpreter'.

Of course some questions arise at this point like: 'Why do we need BASIC, considering the fact that BASIC programs are always translated into machine code programs? . . . Why do we not use machine codes directly?' The reason is quite simple. Machine code instructions are indeed very elementary (as will be seen in a moment) and in order to perform a simple job like multiplying

$$453 \times 567$$

many instructions in machine code are needed. In BASIC such a multiplication is indicated by a single statement:

```
PRINT 453 * 567
```

Besides the fact that usually many more instructions in machine code have to be used, we must also realize that machine code instructions are dominated by the specific characteristics of the computer. For instance they are dominated by the fact that our computer works with registers with a fixed number of cells.

Let us be honest, to write a machine code program is often a tedious job. Hence the next question arises: 'Why do we need to discuss machine code instructions if, after all, we have BASIC?'

In general this is a good question. It apparently emphasizes the philosophy, 'Just use BASIC if you can do the job in BASIC!' There are

however three exceptions that force use to use machine code programs. These exceptions are related to
(1) speed,
(2) memory usage, and
(3) jobs that can never be programmed in BASIC.

1: Speed

Machine code programs are the fastest there are. They can be more than 300 times faster than BASIC programs. For some programs speed is obviously required. As an example consider the well-known video games with fast animated graphics. Machine code instructions provide the only tools for programming such fast actions.

2: Memory

The argument for using less memory sounds odd since we previously stated that usually many more instructions in machine code have to be used compared to the number of BASIC statements. We have to realize however that simple BASIC statements, like `PRINT 453 * 567`, are also converted into machine code. Most of the time we can find optimal solutions with respect to memory, if we use machine code directly. A good example to illustrate this point is the ZX81/TS1000 1K chess program. It is really impossible to write a BASIC chess program if we only have 1 Kbyte of memory at our disposal.

3: Jobs that can never be programmed in BASIC

There are a number of jobs that can never be programmed in BASIC, because they deal with some characteristics of the computer that cannot be controlled by any of the BASIC statements. Examples of these are some kinds of interfacing problems—problems that deal with controlling specific electronic devices coupled to our computer (see chapter 15).

Initial introduction

In this chapter we shall give you an initial introduction to the use of machine code. We cannot give a complete introduction since this would certainly require another complete book.

We will first focus our attention on the basic structure of a machine code program.

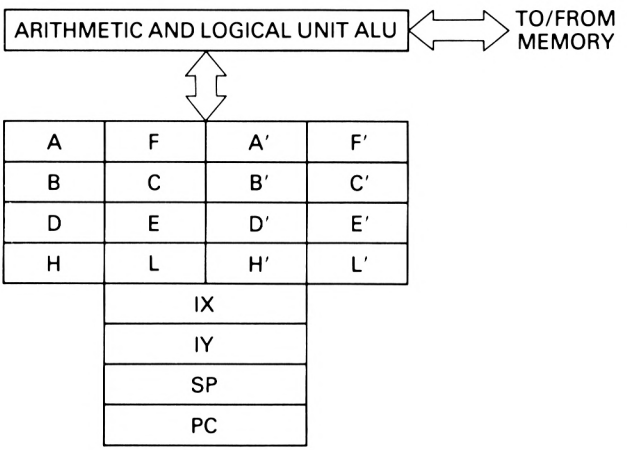
Each machine code is always indicated by a series of zeros and ones. For instance the series

0	0	1	1	1	1	1	0

is interpreted by the CPU as the machine code instruction meaning

load the contents of the next byte into the special register A.

What is 'next byte' and what is 'special register A'? We start with the last question. The CPU contains a number of special registers. These registers are labelled A, B, etc. The next figure shows the Z80 CPU with the special registers.



The Z80 has 14 general-purpose registers, the set A, B, C, D, E, H, L and the alternative set A', B', C', D', E', H', L'. At any time only one set can be active. Therefore we will only consider A, B, C, D, E, H and L. All these registers are 8 bit registers, but some of them can be coupled in order to form a 16 bit register. These registers are B/C, D/E and H/L. Many machine code instructions use the A register and therefore this register has a special name—*accumulator*.

The IX and IY registers are called index registers. They are used for special techniques to point out specific memory locations. The register SP, or Stack Pointer, is a 16 bit register that is used to control a special part of the memory. Furthermore the 16 bit PC register, or Program Counter, is a register that is used to point out the memory location containing the next machine code to be executed. Finally the registers F and F' store *F*lags (see page 154) which are used to indicate certain modes of the processor.

Now to answer the second question with respect to our machine code example, 'What is meant by *the next byte*?' We have to realize that both machine code and data are stored in memory. All memory locations are labelled with a number called the memory address. The first location has number 0, the second 1 etc. If we use a 64 Kbyte memory the highest number is 65535.

The expression 'next byte' indicates that the memory location that follows the location in which our machine code is stored contains the data that has to be loaded into register A. For instance:

Memory address	*Contents*	*Meaning*
16513	. . . .	
16514	0011 1110	load next byte in A
16515	0000 0001	
16516	. . .	

In this example the machine code is stored in location 16514. As a result the contents of 16515 which is 0000 0001, will be stored in register A. Hereafter the computer automatically proceeds with the machine code at address 16516.

A complete program

The next figure illustrates a part of the memory with a complete machine code program. This program is used to add 1 and 2 (don't laugh!). We shall discuss the structure of such a program then explain every instruction and finally show how such a program can be loaded and executed from a BASIC program.

Example of a complete machine code program:

Memory address	*Contents*
16514	0011 1110
16515	0000 0001
16516	1100 0110
16517	0000 0010
16518	0100 1111
16519	0000 0110
16520	0000 0000
16521	1100 1001
16522	. . .
. . .	. . .

If we look at this jungle of zeros and ones, the question arises: 'Which is a machine code instruction and which is data?' The solution to this is to realize that we can indicate which address contains the first machine code instruction (start address). In our case this start address is 16514. This first machine code instruction indicates how the next memory location has to be interpreted—as a new instruction or as data. The first machine code instruction is in fact the one we have just discussed. We saw that memory location 16515 has to be considered as data. It is the 'next byte' to be loaded into the special register A. The computer

automatically proceeds with address 16516 whose content is interpreted as a new machine instruction, etc. The 'orchestral' conductor that leads this process is the program counter, PC. At the start of the program the PC is loaded with the start address 16514. Once the CPU has decoded this first instruction (load next byte into A) the PC is augmented by 2 and therefore it will point to 16516, the location with the next machine code instruction. Note that each instruction is decoded by the CPU and the PC is augmented accordingly.

The instruction at address 16516 is

1100 0110

which means

add next byte to register A

We already know what is meant by 'next byte'. The term 'add' indicates a mathematical operation. The 'next byte' (address 16517)

0000 0010

is interpretated as a number. We shall see later that this byte represents the decimal number 2.

The action 'add next byte to A' can be visualized in the following way:

0000 0001		old contents of A
0000 0010	+	'next byte'
0000 0011		new contents of A

This result represents the decimal number 3.

The last machine codes deal with some specific rules that have to be taken into account if we call a machine code program from a BASIC program. One of these rules is that the final result has to be stored in the coupled B/C registers. Hence the final result is always a 16 bit number that, as we will see, will automatically be converted to a decimal number. For this reason the result, which is now stored in A, is transfered to the C register, and the B register is loaded with zeros. In this way B/C will contain

0000 0000 0000 0011

which actually means '3'.

The machine code that commands the computer to copy the contents of A into C (see address 16518) is indicated by

0100 1111

which means

copy the contents of A into C

Sometimes this instruction will also be indicated by 'store A into C' or 'move A to C'. The trouble with these descriptions however is that they do not indicate what is left in A. Since the original contents remain the same it is better to use the word 'copy'.

Schematically this can be represented as follows:

Before

After

The next machine code instruction is found in location 16519,

0000 0110

and indicates

load next byte into B

Since the next byte is 0000 0000 the register B will then contain only zeros.

The last machine code instruction is stored in location 16521,

1100 1001

which means

return to the program from which this machine code program is called

More about notation

Obviously you have to be a total computer fanatic to read such a jungle of zeros and ones in terms of 'the first instruction means add the next byte to A, etc.'. Do not worry—nobody else does it either.

In order to indicate a machine code program in a more comprehensible way, we need to understand two important points:

(1) All series of zeros and ones can be interpreted as *binary numbers*. This means that we can also convert these into other numbers like decimal numbers that are more familiar to us. We shall see that decimal as well as *hexadecimal* numbers are used.

(2) All series of zeros and ones have a meaning. Each meaning can be indicated by short codes that are easy to remember (mnemonic codes). We shall also see how each machine code is actually represented by such a short letter code. Programs written in such codes are called *assembly language* programs.

Usually if we want to write a machine code program we first use these short letter codes because they directly indicate the meaning.

Once the program is written, it is first converted into hexadecimal numbers. We then use special tables that describe which hexadecimal number corresponds to a special machine code instruction. As a matter of fact there are special programs that can also do this job. The programs are called *assemblers*. In this text we will translate all codes by hand. Finally the hexadecimal numbers are again converted this time into decimal numbers. These decimal numbers are used in POKE-instructions, so called in order to 'poke' the machine code program into the memory. First a short introduction to hexadecimal and binary numbers is given because these numbers are of basic importance.

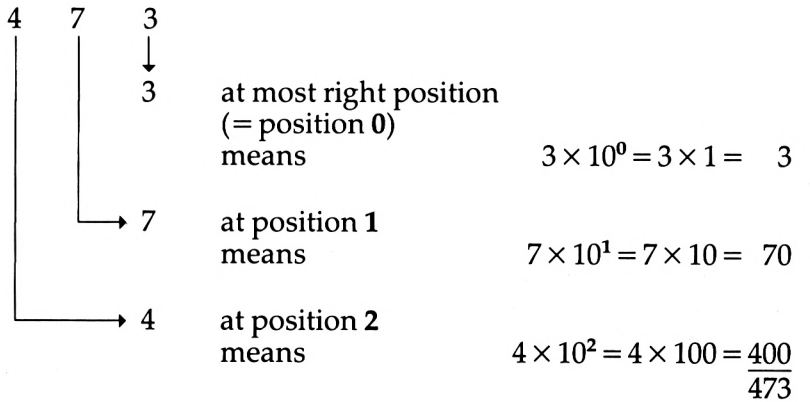
Binary and hexadecimal numbers

The basic principle of decimal, hexadecimal and binary numbers is that each digit represents a multiplication by a certain power of the base number.

In order to illustrate this, consider the following 'normal' (decimal) number

473

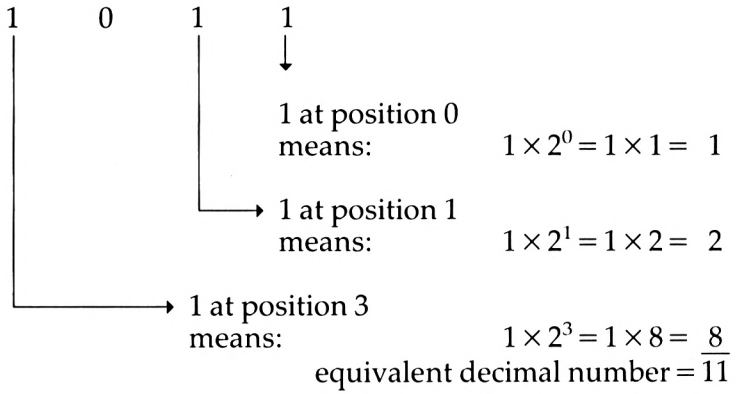
This number consists of three digits and each of them represents a multiplication by a power of base number 10:



Note how the position of each digit (numbered from right to left, beginning at position 0) indicates a power of 10: 10^0, 10^1, 10^2.

A binary number only contains zeros and ones and the position of each digit again represents a power, but in the case of binary numbers we deal with powers of 2.

Consider the following example:



You can see how easily a binary number can be converted to a decimal number. Of course the question may arise if we look at a number like 1011 . . . is it a binary number or the decimal number one thousand and eleven?

In order to avoid such problems, we often give an extra indication e.g. by placing BIN in front of the number (see also page 46), or by using an index. For example

BIN 1011 means binary number 1011

1011_2 means binary number 1011

1011_{10} means decimal number 1011

In order to code a program hexadecimal numbers are used. In this case the digits represent a multiplication with a power of 16.

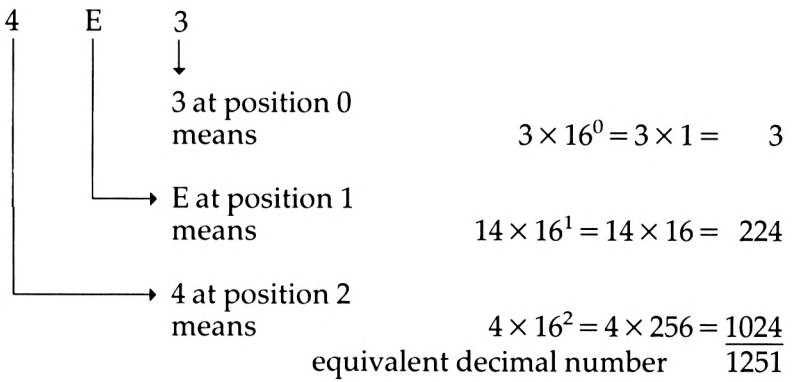
With binary numbers we have only 2 possible digits (0 and 1) and with decimal numbers there are 10 possible numbers (0 . . . 9). With hexadecimal numbers we have 16 possible digits. They are listed in the following table together with their decimal and binary equivalents.

On pages 198 and 199 two programs are given. The first program can be used to convert hexadecimal numbers into decimal numbers, and the second program can be used to convert decimal numbers into hexadecimal numbers.

Table of hexadecimal numbers

Hexadecimal digit	*Decimal equivalent*	*Binary equivalent*
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

An example of a hexadecimal number follows:



The conversion between binary numbers and hexadecimal numbers (and *vice versa*) is easily done by a 'trick':

> Split the binary number into groups of 4 digits and convert these groups of 4 digits into corresponding hexadecimal digits by means of the table given above.

Example

Convert the binary number 0011 1110 into the equivalent hexadecimal number.

Step 1 Split the number into two groups of 4 digits:

0011 1110

Step 2 Convert these groups into hexadecimal digits by means of the table:

0011 corresponds to 3
1110 corresponds to E

Hence our number is 3E.

In order to avoid confusion, an H is sometimes placed after a Hexadecimal number.

Notation of machine code programs

In the next section we shall discuss how our machine code program is written in both hexadecimal and assembly codes.

The original program, written in common language, is

store 1 in register A
add 2 to contents of A
copy contents of A into C
store 0 in register B
return to program from which this machine code program is called

The corresponding assembly language program is

```
LD  A,1
ADD A,2
LD  C,A
LD  B,Ø
RET
```

With the knowledge we already have, the assembly language program is easily understood. `LD` stands for LOAD and `RET` for RETURN. Furthermore the value to be moved is always indicated on the right side of the comma. The destination is defined on the left side of the comma—that is the place where the data has to be loaded.

Assembly language programming means that we need to know the possible machine codes in the way that we know what elementary instructions are provided. In fact, the large number of possible machine codes can easily be divided into a small number of special categories as follows:

I Machine code instructions to indicate that data is moved from one place to another place;
II Machine code instructions to indicate arithmetic operations (add and substract only), logical operations (like AND and OR, see page 57) and comparison operations;
III Machine code instructions to indicate tests (like the `IF` instruction of BASIC) and jumps;
IV Machine code instructions to indicate *stack* operations. A stack is a part of the memory that serves as a scribbling pad;
V Machine code instructions to indicate shifts (shift of the bit pattern) and operations on single bits;
VI Machine code instructions to indicate I/O operations.

Our program contains three codes of category I, one code of category II (ADD) and finally one instruction of category III.

Usually an assembly language program is written first. Then the mnemonic codes are converted into hexadecimal numbers (by means of special tables). For instance the first instruction `LD A,1` is an example of the general instruction

```
LD A,n
```

and by means of special tables (for example look in Appendix IV) we see that this instruction corresponds to 3E. The hexadecimal codes are finally converted into decimal numbers because these numbers are actually used if we want to enter our machine code program by means of a BASIC program.

The next table shows our program once more. Now the binary codes, hexadecimal numbers and decimal numbers are indicated.

Address	*Binary code*	*Hexadecimal code*	*Decimal number*	*Assembly code*
16514	0011 1110	3E	62	`LD A,1`
16515	0000 0001	01	1	
16516	1100 0110	C6	198	`ADD A,2`
16517	0000 0010	02	2	
16518	0100 1111	4F	79	`LD C,A`
16519	0000 0110	06	6	`LD B,Ø`
16520	0000 0000	00	0	
16521	1100 1001	C9	201	`RET`

Entering the program

BASIC provides two powerful functions in order to store data directly in specific memory locations and to read data from specific memory locations. These functions are `PEEK` and `POKE`.

PEEK assigns the contents of a specified memory location to a variable. For example

```
LET I = PEEK 16514
```

assigns the number stored in memory location **16514** to the variable **I**.

POKE stores a value in the memory location indicated. For example

```
POKE 16514,62
```

writes the value **62** (hence the bit pattern 0011 1110, see table above) to the memory location **16514**.

One of the big problems is, 'Where can I store the machine code program?' With the ZX81/TS1000 we usually start at address **16514** and furthermore we use a special **REM** statement as will be shown.

```
10 REM a certain number of characters
20 ⎫
.. ⎬ POKE decimal numbers into addresses
.. ⎭ starting from 16514

.. evoke (call) machine code program
```

The number of characters after **REM** has to correspond to the number of codes that will be entered. The key to this strange construction is the fact that comments of a **REM** statement are always stored in particular locations (from 16514 in the ZX81/TS1000). These locations, as we saw before, are always neglected in a BASIC program (comments are neglected during the *execution* of a program) hence they provide a safe place for storing machine codes. With the ZX Spectrum we can use the command

```
CLEAR 32499
```

and load our machine code program from address **32500**.

The next example shows how to load our program using the ZX81/TS1000.

```
10 REM AAAAAAAAA
20 PRINT "ENTER MACHINE CODES"
30 FOR K = 1 TO 8
40 INPUT N
50 POKE 16514 + (K - 1),N
60 NEXT K
70 PRINT "BEGIN"
80 LET BC = USR 16514
90 PRINT BC
```

After the RUN command the screen shows

```
ENTER MACHINE CODES
```

now we enter

```
 62
  1
198
  2
 79
  6
  0
201
```

and the computer shows

```
BEGIN
3
```

Line 10 starts with the REM statement that provides the room for our machine code program. As a result of the FOR...NEXT statement (lines 30–60) the values 62,1,198, etc. are stored in 16514, 16515, 16516, etc. Once the FOR...NEXT statement is terminated the computer prints BEGIN and obeys the statement

```
LET BC = USR 16514
```

The function USR indicates that the computer has to execute a machine code program starting at address 16514. As a rule, the contents of the BC register is finally assigned to the variable indicated (variable with the leading name BC). We already know that this machine code program adds 1 and 2 and the result printed can be taken as a proof.

ZX Spectrum users can try the following program (it has exactly the same structure but begins with a CLEAR statement rather than REM).

```
10  CLEAR 32499
20  PRINT "enter machine codes"
30  FOR k = 1 TO 8
40  INPUT n
50  POKE 32500 + (k - 1),n
60  NEXT k
70  PRINT "begin"
80  LET bc = USR 32500
90  PRINT bc
```

Once the program has been executed we can look to the different memory locations in order to see what bit patterns are stored. For instance

```
PRINT PEEK 32500
```

results in

```
62
```

which actually corresponds to (see table on page 147):

```
0011 1110
```

Some final remarks about machine code instructions

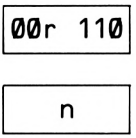
As we explained before, it is impossible to give a complete course in machine code programming, since this would really require another book.* However we will briefly discuss some instructions and some addressing techniques.

Immediate addressing mode

This mode is of interest if we want to load constants directly into a register. For example

```
LD r,n
```

means load constant n into register r (A, B, etc.). This instruction takes two bytes, one for the code 'loads into r immediately' and one for the constant n.



where the register r is given by the following table.

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

For instance 00111 110 means 'load into A . . .'

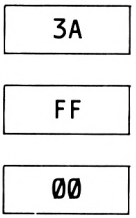
Extended or direct addressing

This addressing mode is of interest if we want to indicate absolute addresses of the memory. For instance

```
LD A,(00FF)
```

* *Programming the Z80* by Rodnay Zaks published by Sybex.

indicates 'load contents of memory location 00FF into register A'. Note that (00FF) indicates 'contents of memory location 00FF'. This instruction takes three memory locations:



The code 3A represents 'load into A with direct addressing mode'. FF represents the low-order part of the memory address and 00 represents the high-order part of the address. Another example

```
LD (00FF),A
```

means 'load contents of A into memory location 00FF'.

If we want to move data from location 1 to location 2 we can apparently use

```
LD A,(location 1)
LD (location 2),A
```

Note how register A is used as an intermediate container. Besides the LD instructions many other instructions for moving data are offered. Those which move blocks of data are especially interesting.

Decrement and increment

Simple instructions to decrement (by 1) or increment (by 1) a register are as follows:

```
DEC B
INC C
```

Add and subtract

Instructions that describe a summation and a subtraction are described below.

```
ADD A,n
```

means 'add constant n to contents of A', and

```
ADD A,(HL)
```

means 'add contents of memory location indicated by register pair HL to contents of register A'.

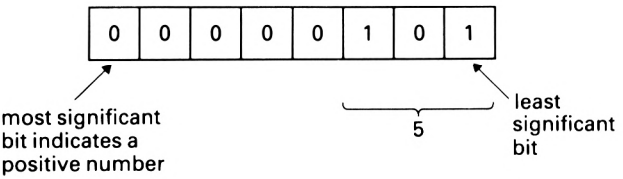
```
SUB n
```

means 'subtract constant n from contents of register A' (register A is always implicitly assumed).

With these arithmetic instructions binary numbers are automatically represented in a special way. In this way both negative and positive numbers can be stored in a single byte. The rule is that the first bit (the one furthest left) is the most significant bit and indicates the sign of a number:

0 indicates a positive number
1 indicates a negative number

Positive numbers are represented in the last seven bits according to the rules of binary numbers as discussed on page 143. For example, 5 is represented by



Negative numbers are represented by means of the *twos complement code*. According to this code a negative number is obtained in the following way

- take the positive number as a starting point
- flip the bits
- add 1

For instance −5 is obtained by

- take positive number (5) 00000101
- flip the bits 11111010
- add 1 11111011

Hence 11111011 represents −5

Note that the highest positive number in a single byte is

0111 1111 (= 127)

Furthermore problems arise if 1 is added to this number

1000 0000

apparently the result is negative!

Such problems always occur if a fixed number of digits is used to represent a number. As an illustration you may think of a mileometer (odometer) in a car with 5 digits. If the car has travelled 99999 miles, the next mile will apparently result in a brand new car!

In order to warn us of such situations the Z80 uses several flags (specific bits of the F register, see page 152). For instance, the bit furthest to the right of this F register represents the C flag or C bit (Carry bit). This bit is set (becomes 1) if a shift from the most significant bit occurs. Besides the carry bit an overflow bit can also become 1. This takes place if a shift from the second most significant bit occurs. On combining these flags we can construct programs that deal with large numbers. We shall not discuss this here because it is rather a complicated subject and it is only necessary for 'complex' arithmetic.

AND and OR

The well know logical operations AND and OR (see also page 57) can be performed.

```
AND B
```

performs logical AND with contents of register A and the result is stored in A (again register A is implicitly assumed).

For example

```
        A = 10101100
        B = 11011010
A AND B = 10001000
```

AND instructions are very useful to check if a certain series of bits are set.

Jumps

There are also instructions to indicate GOTO and GOSUB instructions. For instance

```
JP 020E
```

means 'jump to memory location 020E' (GOTO 020E) and

```
JP (HL)
```

indicates 'jump to the memory location that is indicated by the register pair HL'.

The GOSUB statement is indicated by CALL. For example

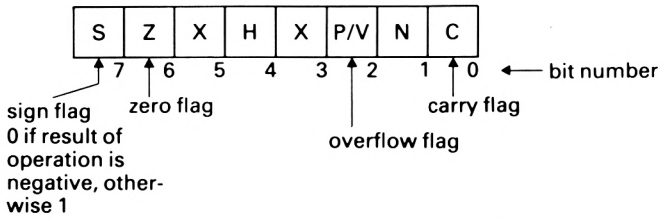
```
CALL 020E
```

means 'proceed with the program at location 020E and return to main program if the next instruction encountered is RET'.

Finally we want to show that statements such as

```
IF A = B THEN GOTO
```

also have an equivalent in machine code. Again use is made of the F register of the CPU which contains a number of flags. For example, bit number 6 corresponds to the so-called zero flag indicated by means of the letter Z.



The F register with a number of flags

These flags can be changed, for example, as a result of an arithmetic instruction or a compare instruction. For instance, if a SUB instruction leads to a zero, the Z flag is set (i.e. the Z bit of F register becomes 1).

These flags are used by instructions like the conditional jump. For example

```
JZ 020E
```

indicates 'jump to memory location 020E *if* the Z flag is set'.

```
CALL Z,020E
```

indicates 'jump to subroutine starting at memory location 020E *if* the Z flag is set'.

```
RET Z
```

indicates 'return to main program *if* Z flag is set'. It is obvious that these conditional jump instructions do not influence the flags of the F register since these flags are used by the conditional jump instructions themselves.

A last example follows:

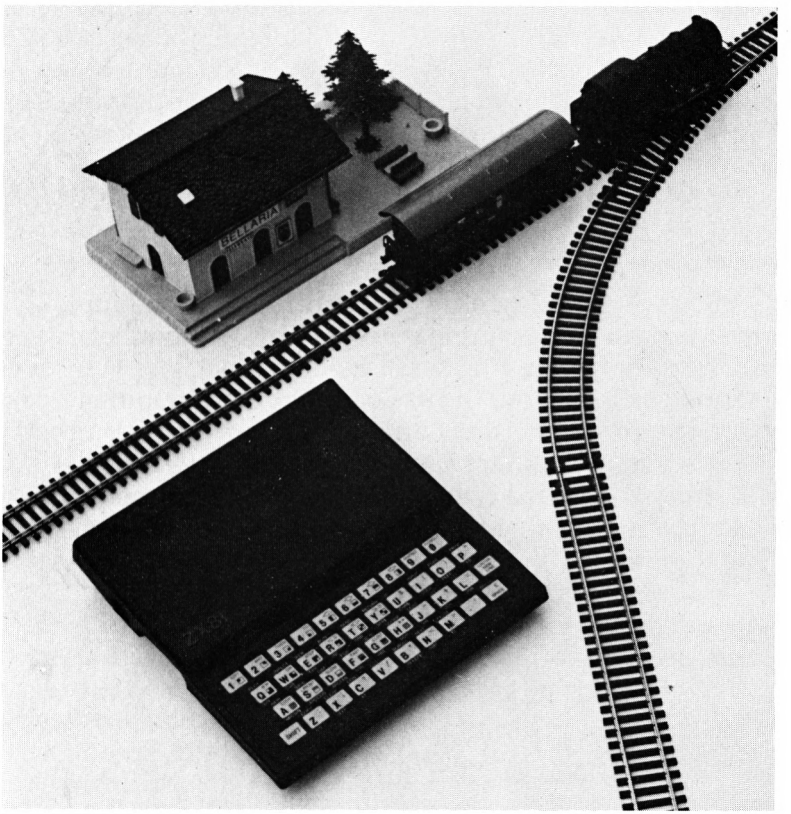
```
CP 100
JP Z,020E
```

The first instruction indicates that the contents of A must be compared with 100. If A equals 100 the Z flag is set. Hence the instruction JP Z,020E indicates that a jump to 020E has to be made *if* A = 100.

The next chapter introduces the machine codes that control the input and output functions of our computer.

CHAPTER 15

Interfacing or connecting your computer to the outside world



It is of course possible to connect your computer to all kinds of devices. Some hardware will be needed and I am well aware of the fact that this really frightens many enthusiastic computer owners. Many interesting projects however can be undertaken, like controlling a model train or a robot arm. One of the well-known hobbies of some computer 'professionals' is to control a device called a 'mouse', that has to find its way through a maze.

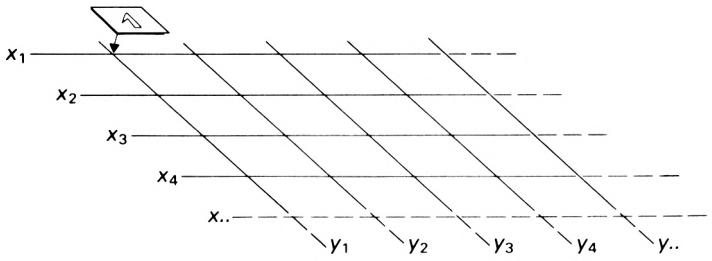
All these examples have at least one thing in common—a connection by means of wires is made to the hardware of the computer. We will show you two methods of making these connections or *interfacing*. The first method is more of a 'trick', but it is a nice example of an easy way to connect one or more switches to your computer. Many hardware specialists will certainly find it a bad way of interfacing, because we do not use the connector at the rear of the computer. In order to compensate for this bad method of interfacing we shall show a really suitable application. We will show how the computer can be used as a powerful tool for handicapped people.

The second method of interfacing is based on the connector at the rear of the computer. This connector provides all the necessary inputs and outputs to connect any device to your computer. Many readers probably have no previous background knowledge of interfacing and therefore some practical information on digital electronics is given.

A simple way of connecting one or more switches to your computer

In this section we shall describe how we can connect a simple switch to our computer. Later an example is given of a program that can be used to provide a special typewriter for disabled people.

First let us consider how the keyboard of our computer works. We shall then see how the keyboard mechanism can be used in our interface. As a matter of fact each key can be considered as a simple switch. If a key is pressed two wires are brought in contact with each other. The next figure illustrates schematically a part of the keyboard.



Schematic representation of a part of the keyboard. If a key is pressed two wires (x and y) are brought in contact with each other.

In this figure one key (the key with '1') is depicted and in addition we see a part of an array or *matrix* of wires. There are 8 x-wires: x_1, x_2, ... x_8 and 5 y-wires: y_1, y_2, ... y_5. Each (x,y) pair corresponds to a key, for instance x_1 and y_1 correspond to key '1'. If this key is pressed x_1 and y_1

are connected and as a result the computer knows that '1' has been pressed.

Surveys of the (x, y)-combinations together with the corresponding keys are given in the following tables.

Table of (x, y) pairs and corresponding keys of the ZX81/TS1000)

	y_1	y_2	y_3	y_4	y_5
x_1	1	2	3	4	5
x_2	Q	W	E	R	T
x_3	Ø	9	8	7	6
x_4	A	S	D	F	G
x_5	P	O	I	U	Y
x_6	SHIFT	Z	X	C	V
x_7	NEWL/ENTER	L	K	J	H
x_8	SPACE		M	N	B

Table of (x, y) pairs and corresponding keys of the ZX Spectrum

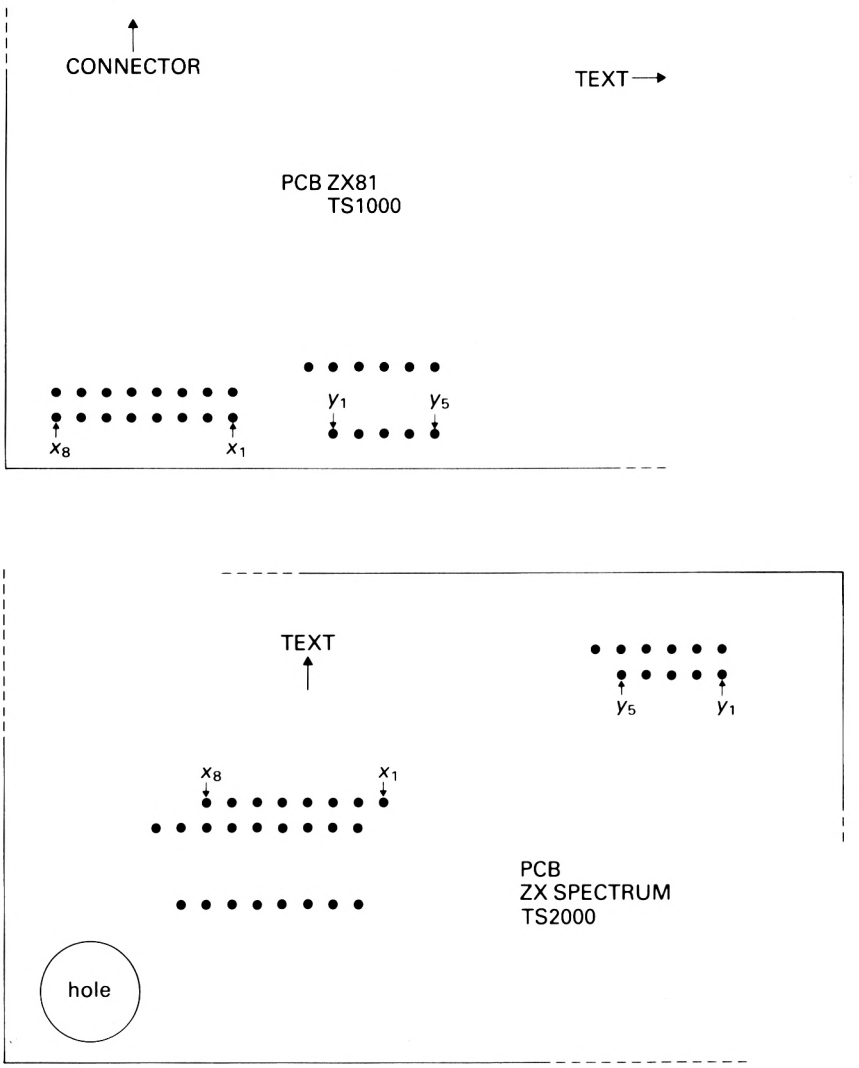
	y_1	y_2	y_3	y_4	y_5
x_1	1	2	3	4	5
x_2	Q	W	E	R	T
x_3	A	S	D	F	G
x_4	Ø	9	8	7	6
x_5	P	O	I	U	Y
x_6	caps/shift	Z	X	C	V
x_7	enter	L	K	J	H
x_8	space	symbol/shift	M	N	B

The wires x_1 to x_8 and y_1 to y_5 are finally arranged in two flat cables that are connected by means of a special flat-cable socket to the computer printed circuit board (PCB). On the rear side of this PCB we can solder two wires—one wire to x_1 and another to y_1.

The figure on the next page illustrates the two PCBs and the positions of the flat-cable connectors.

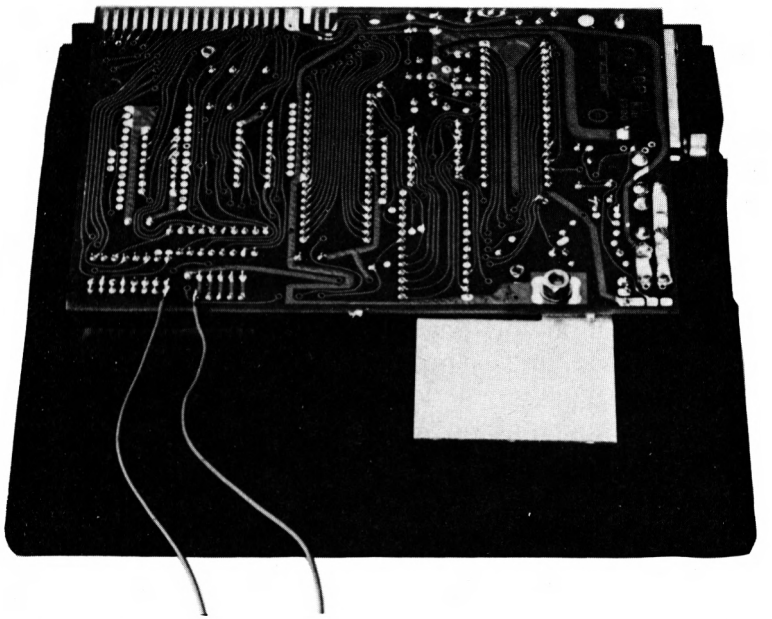
Details of PCBs of the ZX81/TS1000 and ZX Spectrum are shown with the soldered contacts that correspond to the signals $x_1 - x_8$ and $y_1 - y_5$. One corner of the ZX81/TS1000 PCB and two opposite corners of the ZX Spectrum are shown. The details are enlarged so that the total figure is not true to scale.

The photograph (page 159) illustrates a ZX81 that has been opened and we can clearly see the two wires that are soldered to the points x_1 and y_1. By the way, in order to open the ZX81/TS1000 you must use a fine screwdriver and remember that some screws are hidden under the feet that support the computer (the ZX Spectrum has no hidden screws).

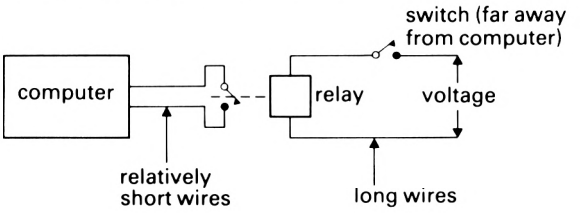


At this point we must warn you—only use a soldering iron that is intended for this kind of work because good soldering is vital! Do not use a 'big' soldering iron because you may seriously damage your computer due to the excess heat.

The two wires are finally led through the openings and connected to a switch outside the computer. Use only short wires (at the most 30 cm = 12 inches) otherwise problems may arise. If you need a switch at a distance from the computer you can for example use a relay as in the following diagram.



Inside the ZX81. Two wires are soldered: one to point x_1 and the other to point y_1. In this way key '1' can be pressed externally.



Set up of switch at a distance from the computer.

First we shall give a simple illustration of our switch in use. Type in the following program:

```
10  PRINT  "BEGIN"
20  LET Z$ = INKEY$
30  IF Z$ = ""THEN GOTO 20
40  PRINT Z$
```

If we start this program the display shows

```
BEGIN
```

Now if the *switch* is pressed we see

```
1
```

It is as if we had pressed key '1'. In this case only two wires have been used in order to simulate one key but of course more wires could be used.

This leads to some interesting projects like:

- if all (x,y) pairs of the digit keys are used we can develop a numeric pad (extra keyboard for entry of digits);
- if more (x,y) pairs are provided (e.g. 4) we can develop simple joysticks that can be used in many games.

The one switch typewriter

This interfacing technique can hardly be considered as a 'good' technique because of obvious reasons. In order to compensate for this we shall give an interesting example: we shall show how our computer can be used as a typewriter by disabled people who cannot use their fingers.

The key point of the design is that *only one switch* needs to be used to enter information into the computer. Such a switch can be controlled for example by the head or by a leg.

Program

```
 10 CLS
 20 LET c$=""
 25 LET b$="
                "
 30 DIM a$(128)
 40 LET a$="A    B    C    D    E
F    G    H    I    J    K    L    M
N    O    P    Q    R    S    T    U
V    W    X    Y    Z    .    ,    ?
     CR   STOP"
 50 PRINT AT 0,10; PAPER 1; INK
7;"ONE KEY WRITER"
 200 FOR i=1 TO 4
 210 FOR j=1 TO 8
 215 LET k=(i-1)*32+(j-1)*4+1
 220 PRINT AT (2*i),(4*j)-4; PAP
ER 2; INK 7;a$(k)
 230 PAUSE 15
 240 PRINT AT (2*i),(4*j)-4;a$(k
TO (k+3))
 250 IF INKEY$<>"" THEN GO TO 30
0
 260 NEXT j
 270 NEXT i
 280 GO TO 200
 300 IF a$(k TO (k+3))="STOP" TH
EN GO TO 500
 310 IF a$(k TO (k+1))<>"CR" THE
N GO TO 360
 320 PRINT AT 15,0;b$;AT 16,0;b$
;AT 15,0;c$
 330 LET c$=""
 340 LPRINT CHR$ (13)
 350 GO TO 400
 360 LPRINT a$(k);
 370 LET c$=c$+a$(k)
 400 PRINT AT 16,0;c$
 410 GO TO 200
 500 LPRINT
 510 INPUT "START AGAIN WITH ENT
ER "; LINE b$
 520 GO TO 10
```

If the RUN command is given, a matrix of letters and some special symbols are displayed on the screen. Furthermore we see a cursor that jumps from symbol to symbol.

```
                ONE KEY WRITER
A   B   C   D   E   F   G   H
I   J   K   L   M   N   O   P
Q   R   S   T   U   V   W   X
Y   Z   .   ,   ?       CR  STOP
```

These symbols are displayed with the 'one key typewriter'. The blank position indicates a space and CR indicates 'carriage return'.

We instruct the user to press the switch when the cursor points at the symbol that he wants to type.

Immediately the switch is pressed the required character is displayed and in the same way the next required character can be entered. If CR is pressed a new line can be entered. The text will also be printed provided that a printer is connected to the computer. We can adapt the speed by means of the PAUSE statement at line 230.

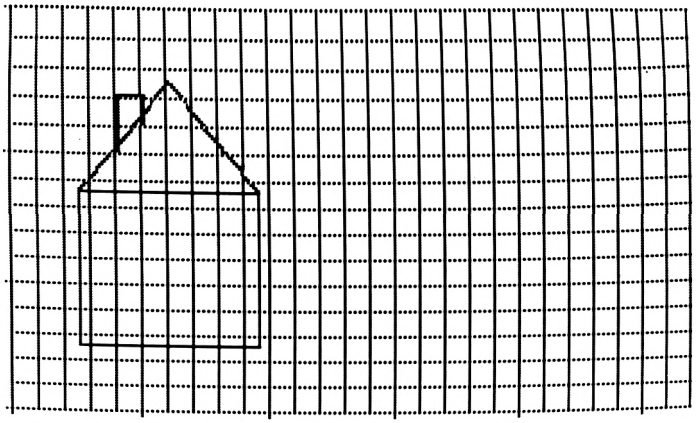
This program shows you a very simple 'one key writer'. Of course you can adapt it to a more sophisticated one, for example by using two cursor directions. First a cursor moves vertically and a row can be chosen, and then the cursor moves horizontally to select the chosen character. Furthermore you can introduce a special symbol, for example /, and if the user selects this symbol the character last entered will be erased.

Finally we would like to point out that many programs can be developed for the disabled using the principle of 'one switch entry' as we have done. Here are a few examples:

- By means of a (slowly moving) cursor we can select a point on the screen. To achieve this a grid is displayed and then the cursor moves horizontally so that the x-coordinate of the point can be chosen. After this the cursor moves vertically in order to select the y-coordinate. Hence one particular point is selected and displayed. In the same way a second point can be chosen. The cursor now can offer two possibilities, for example by jumping between two squares with numbers 1 and 2:

  (1) means draw a line between the last entered point and the preceding point;
  (2) means use this point as the start of a new line.

The figure on page 162 illustrates an example of a result.



- By means of a (slowly moving) cursor and our switch we can ask the user to choose from a list of possible answers to questions that are displayed. In this way we can develop many educational programs for disabled people.

Using the rear connector

One of the obvious disadvantages of the previous method of interfacing is the fact that the computer cannot send messages to devices. The computer can only receive messages but can never control an electronic device.

In order to achieve this we must use the connector at the rear of the computer. This connector provides a number of inputs and outputs by which it is possible to enter and to send messages.

Before we actually describe what signals are provided we will first give an elementary introduction to digital electronics. We shall discuss all the components that will be used in our special interface.

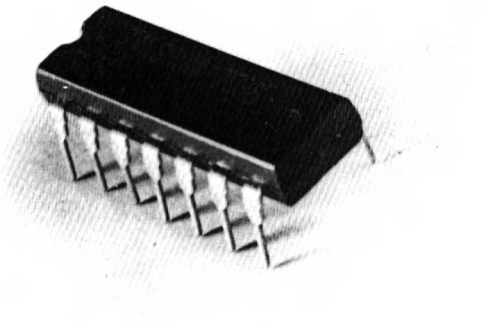
Elementary introduction to digital electronics

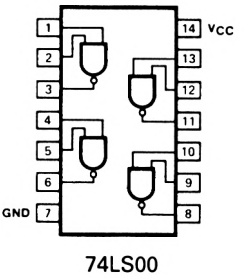
A digital circuit is an electronic circuit that only works with voltages representing ones and zeros. The ones and zeros are represented by voltage levels:

0: a voltage between 0 and 0.6 V
1: a voltage with a minimum level of 2.4 V

We can buy integrated circuits (ICs) with standard digital electronic circuits. A well-known series is the so-called TTL series. In fact we can buy several types of TTL ICs; the standard type, the LS type and the high speed type. The LS type consumes less power (LS stands for

'Low Schottky'). As a rule we shall use the LS series. The next figure shows a typical TTL IC together with the circuit diagram provided by the manufacturer.

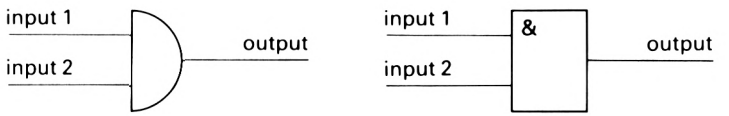




Example of a typical TTL integrated circuit (SN 74LS00). This circuit contains 4 so called NAND gates. These are indicated in the diagram.

All TTL integrated circuits use only 5 V supply and this voltage is supplied by one of the pins of the connector at the rear of the computer.

The supply voltage, Vcc, is connected to pin 14 and the ground lead, GND, to pin 7. Note that the SN 74LS00 contains 4 identical logic circuits called NAND gates. In order to illustrate the function of such a NAND gate we start with the AND gate.



Two schematic representations of the AND gate.

This AND gate is characterized by the fact that the output is only 1 if both inputs are 1.

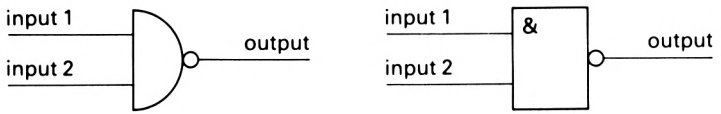
This behaviour can be depicted by means of a *truth table* that indicates all the possibilities with respect to the input together with the output.

Truth table of an AND gate

Input 1	*Input 2*	*Output*
1	0	0
0	1	0
0	0	0
1	1	1

We see that the output of an AND gate is only 1 if both inputs are 1. A NAND gate is a 'NOT-AND gate' which means that the final result is inverted once more. Hence the output of a NAND gate is only 0 if all inputs are 'high' (1).

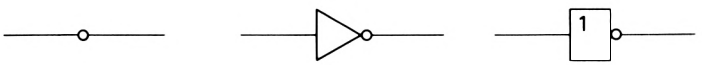
The difference between the schematic representation of a NAND and an AND gate is just a little circle at the output:



Schematic representations of a NAND gate

Note that the NAND gates of the SN 74LS00 have only 2 inputs. We can construct all kinds of digital circuits by combining these NAND gates. In practice many special circuits are also offered as a TTL IC. For instance the SN 74LS30 offers a NAND gate with 8 inputs. The output of this 'large' NAND gate will only be low (0) if all inputs are high (1).

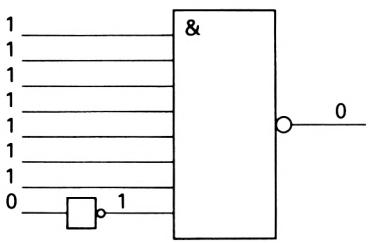
A very simple circuit is the one called an inverter.



Schematic representations of an inverter

This circuit just inverts the input: 1 becomes 0 and 0 becomes 1. By means of these inverters and an 8 input NAND we can easily decode a specific 8 bit pattern.

Consider the example in the following diagram:



The output of this circuit will only be 0 if the bit pattern of the input is

11111110

In other words this circuit 'decodes' the bit pattern: 11111110. We shall see how this technique can be used to decode addresses that are also indicated by specific bit patterns. Remember that the Z80 uses 16 bits to indicate an address. Therefore we require two 8-input NANDs and a number of inverters to decode a specific address.

Besides the ANDs, NANDs and inverters we often use OR and NOR gates. An OR gate is a circuit with an output that is only low (0) if both inputs are also low (0). Conversely the output is high if at least one input is high.

Schematic representations of an OR gate

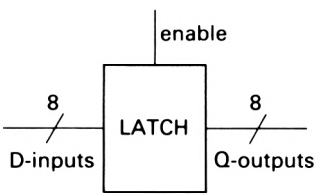
Again we can use a truth table to define the exact functioning of an OR gate.

Truth table of an OR gate

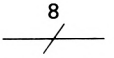
Input 1	*Input 2*	*Output*
1	0	1
0	1	1
0	0	0
1	1	1

With a NOR gate the output is converted once more meaning that the output is only high (1) if all inputs are low (0).

All the circuits we have met so far manipulate ones and zeros in a very particular way; if the inputs are known the output is also known. We also need circuits that can *store* a one or a zero. Such a circuit is called a latch.



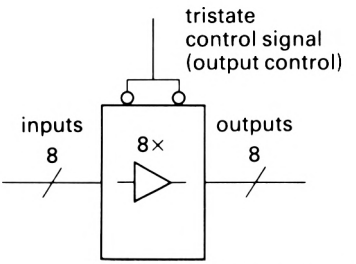
Schematic representation of a latch

By means of the symbol 

we indicate that there are in fact 8 wires. Hence this latch has 8 inputs and 8 outputs. In the diagram provided by the manufacturer the inputs are indicated by D_0, D_1, ... D_7 and the outputs by Q_0, Q_1, ... Q_7. In fact the inputs and outputs form pairs as will be clear in a moment.

If the enable line becomes high, Q_0 will equal D_0, Q_1 will equal D_1, etc. The outputs remain the same, even if the enable signal becomes low again. This clearly indicates that such a latch serves as a memory. Only if the enable signal becomes high again will the outputs change (because they copy the inputs), hence new values will be stored.

Finally we shall use circuits that act like a switch. By means of these circuits we can connect outside signals to the computer. Such circuits are provided by so-called tristate buffers. The next diagram shows a schematic representation of an 8-tristate buffer with 8 inputs and 8 outputs.



Schematic representation of an octal (8) tristate buffer

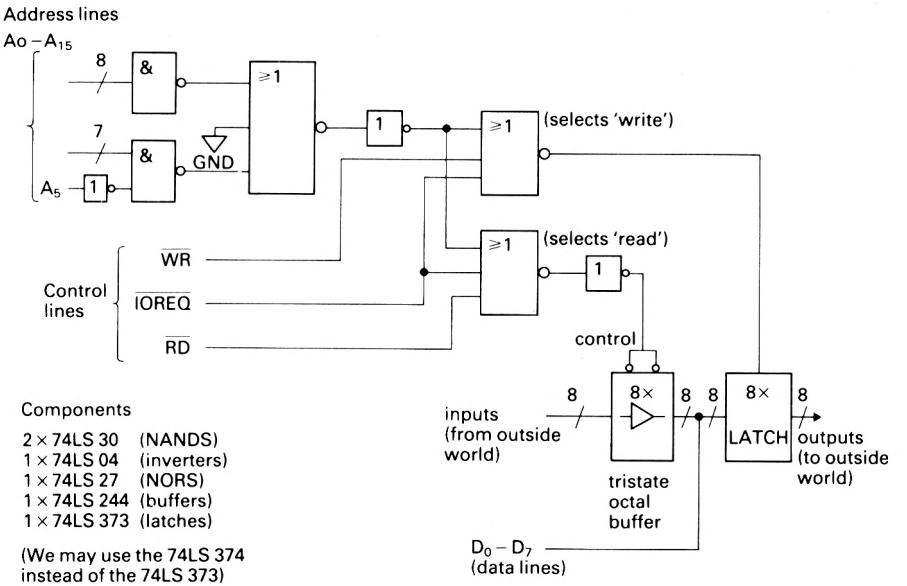
The inputs are usually indicated by the character I and a number and the outputs by Y. If the tristate control signal becomes low (in fact this signal is inverted, note the circles) all inputs will be connected to the outputs, hence I_1 is connected to Y_1, I_2 is connected to Y_2 etc. If the tristate control signal becomes high all switches are open.

After this brief introduction it is time to look at a practical example and build our own interface.

A practical general interface

This scheme shows the general interface that can be used by our computer. The signals A_0—A_{15}, D_0—D_7, $\overline{WR}$, $\overline{IOREQ}$ and $\overline{RD}$ are obtained from the rear connector. The lines A_0—A_{15} are used to indicate an address, lines D_0—D_7 are used to represent data which in fact is always a series of eight bits. The lines $\overline{WR}$, $\overline{IOREQ}$ and $\overline{RD}$, called control lines, will be discussed later on.

The voltage supply and ground (GND) are also provided by the computer (see page 15). Note that line A_5 is inverted. In this way this interface is only active if the next address is selected.

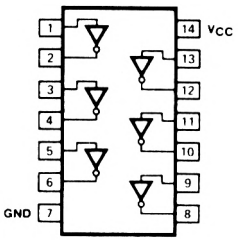


Components

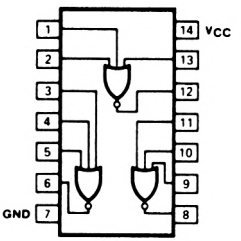
2 × 74LS 30 (NANDS)
1 × 74LS 04 (inverters)
1 × 74LS 27 (NORS)
1 × 74LS 244 (buffers)
1 × 74LS 373 (latches)

(We may use the 74LS 374 instead of the 74LS 373)

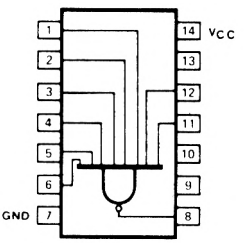
Scheme of general interface



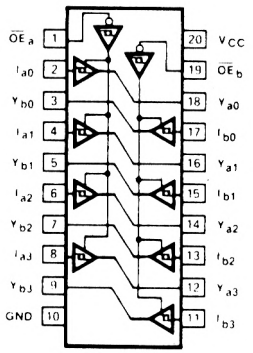
74LS04



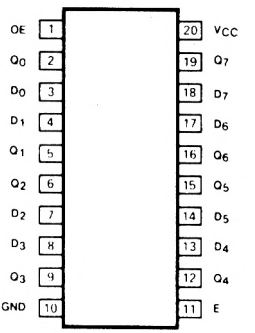
74LS27



74LS30

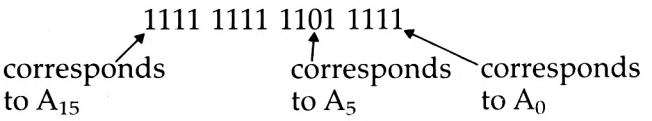


74LS244



74LS373

Pin connections of ICs (integrated circuits)



In hexadecimal notation this address corresponds to FFDF.

We will now pay attention to the functioning of this interface. In the previous chapter we saw that the computer uses bit patterns in order to indicate addresses of memory locations in order to fetch or store data.

For example, LD A,(FFDF) indicates load register A with the contents of memory location FFDF. In this case FFDF indicates a specific memory location. The Z80 has a special feature since it can also indicate external devices by means of an address.

Obviously a problem arises . . . how does the computer know if an address deals with a memory location or with an external device? In order to solve this problem the computer has a special control line with the name

IOREQ (I/O request)

This signal becomes high (1) if an address deals with that of an external device like our interface. If this control line is low, the address is automatically taken as a memory address. For example with an instruction like LD A, (FFDF) this line will automatically be low so the computer takes FFDF as a memory location. However with the IN and OUT instructions that will be discussed later on, the line IOREQ will automatically become high.

Note that the rear connector actually provides $\overline{\text{IOREQ}}$. The horizontal line above IOREQ indicates 'inverted signal'. So as a matter of fact $\overline{\text{IOREQ}}$ is usually high and only becomes low (active) if an external device is to be selected.

If we take a closer look at our interface we see that all signals A_0—A_{15} except A_5 are fed to two NAND gates, and these two NAND gates are connected to one NOR gate. The output of this NOR gate will only be high if the appropriate address 1111 1111 1101 1111 (or hex: FFDF) is generated by the lines A_0—A_{15} of the computer. If now the $\overline{\text{IOREQ}}$ is low ('active' IOREQ) one of the signals $\overline{\text{WR}}$ and $\overline{\text{RD}}$ will activate our interface as follows.

$\overline{\text{WR}}$ (active = low)
The output of the NOR (selects 'write') will become high and the latch is activated: information from the computer (data: D_0—D_7) will be copied into the latches. The 8 output pins now provide the information that is indicated by the computer. This means that information (data) is now transmitted from the computer to the interface.

$\overline{RD}$ (active = low)
The output of the NOR (selects 'read') will become high and this signal controls the tristate octal buffer. The functioning of this buffer can be compared with a switch. Now the 8 inputs are connected to the data lines D_0—D_7 and as a result the computer is fed with data from the outside world. This means that data is now transmitted from the outside world to the computer (D_0—D_7).

The next 'assembly language' program shows how our interface is controlled.

```
 1  REM  0123456789012345678   (reserves space)
 2  POKE 16514,1          }
 3  POKE 16515,233        } ld,bc,FFDFH
 4  POKE 16516,255        }
 5  POKE 16517,237        }
 6  POKE 16518,120        } in a,(c)
 7  POKE 16519,6          }
 8  POKE 16520,0          } ld b,oH
 9  POKE 16521,79         } ld c,a
10  POKE 16522,201        } ret
11  POKE 16523,1          }
12  POKE 16524,223        } ld bc,FFDFH
13  POKE 16525,255        }
14  POKE 16526,62         }
15  POKE 16527,0          } ld a,oH
16  POKE 16528,237        }
17  POKE 16529,121        } out (c),a
18  POKE 16530,201        } ret
```

The instruction `ld bc,FFDFH` loads the (device) address `FFDF` into `bc`. The instruction `in a,(c)` indicates 'reads input'. As a result $\overline{RD}$ and $\overline{IOREQ}$ will become low (active) and the input (D_0—D_7) is fed to register `a`. After `ld b,oH` and `ld c,a` the result is loaded in the `bc` register pair (see also the example on page 147). The instruction `ret` terminates this part of the program. The second part shows how information is sent to the interface. First the address is loaded in `bc` using `ld bc,FFDFH`. Then the data (in this case `oH`) is loaded into `a` using `ld a,oH`. We can of course enter any data in `a` by means of a `POKE` instruction:

```
POKE 16527,data
```

The data is then sent to the device indicated by `out (c),a`. As a result the $\overline{IOREQ}$ and $\overline{WR}$ signals become low (active) and the data (D_0—D_7) is copied into the latches. The instruction `ret` again terminates this part. Incidentally the assembly instructions `in` and `out` are directly available in BASIC with the ZX Spectrum.

We can now read data from the inputs from the 'outside' world (see diagram page 167) by means of

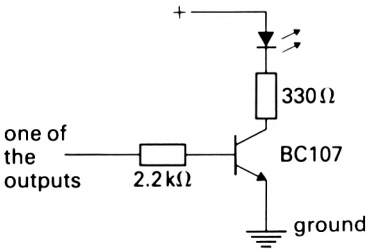
```
X = USR 16514
```

and send data to the outputs of the interface by means of

```
POKE 16527,X
DUMMY = USR 16523
```

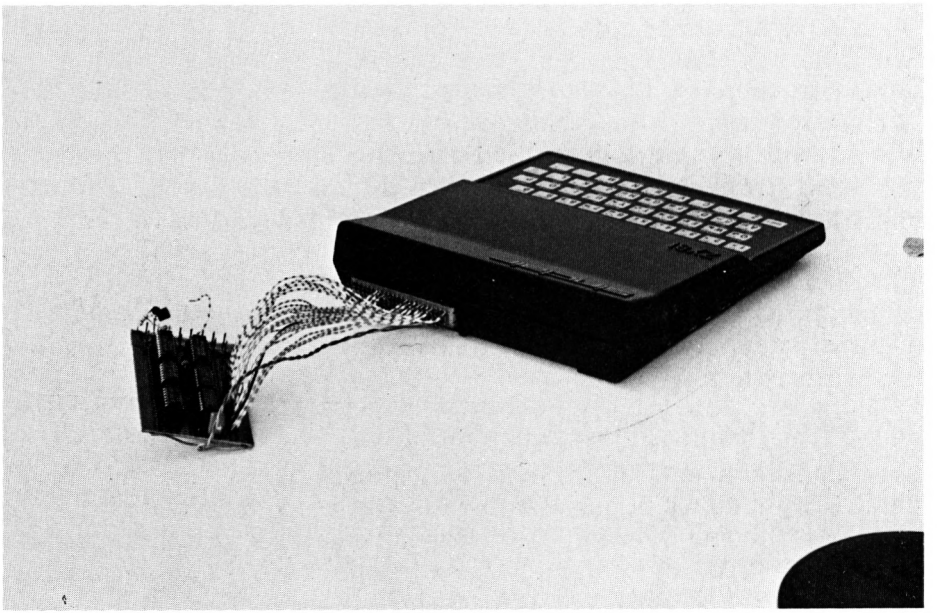
In order to illustrate this we will write a BASIC program. This program controls our interface in such a way that an LED (red lamp) flickers until one of the inputs becomes 0.

The LED is connected in the following way:

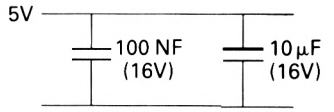


A wire is connected to the ground line and by means of this wire we can feed '0' to one of the inputs.

The photo shows our complete interface.



To filter the power supply two extra capacitors are used in the interface:



The BASIC program is

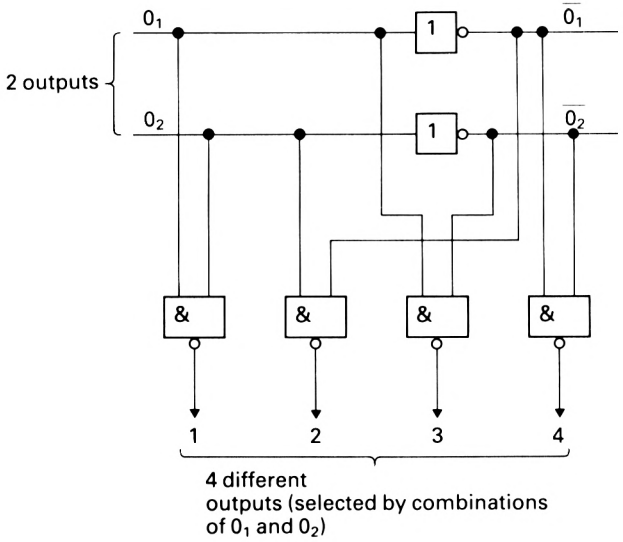
```
20  LET I = 1
25  LET I = - 1 * I
30  POKE 16527,I
35  LET X = USR 16523
40  FOR J = 1 TO 30
45  NEXT J
50  IF USR 16514 = 255 THEN GOTO 25
55  STOP
```

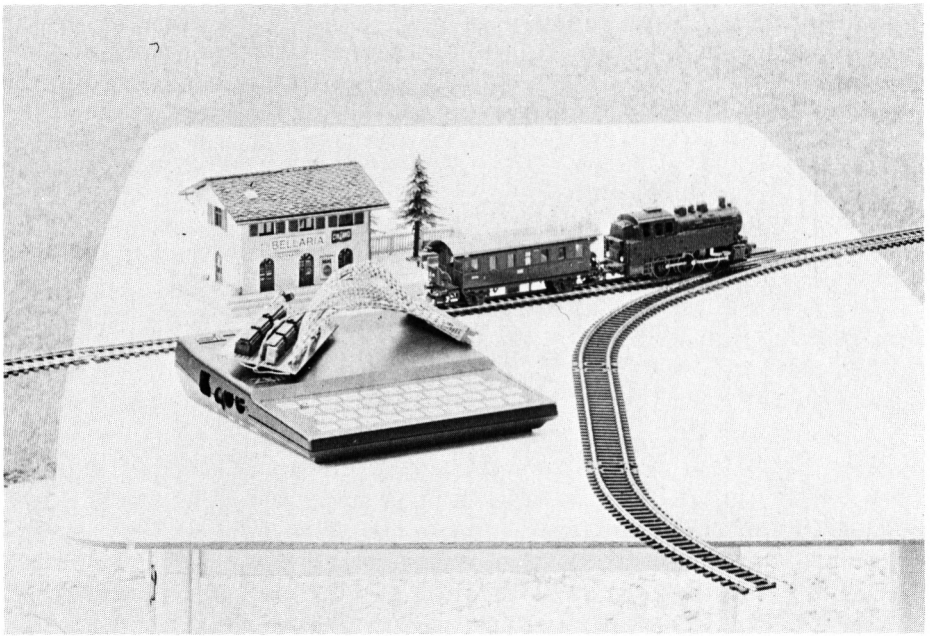
Line **30** determines the output that has to be sent to the interface. Line **35** actually activates the interface and **I** is sent to the output pins of the interface. Lines **40** and **45** serve as a waiting loop. Line **50** activates the interface again. If one of the input pins is low, the computer reads a number that differs from **255** and the program is terminated. If not, the computer jumps back to line **25**. **I** changes from 1, −1, 1, etc. and as a result most output pins will be 0, 1, 0, etc. If one of these output pins (for example pin 3) is connected to the LED, the LED will flicker.

Controlling a model railway and other applications

This interface provides us with 8 inputs and 8 outputs. Each signal may be used individually but in this case we can only control 8 points of our model railway. However with the use of NAND gates and inverters we can decode $2^8 = 256$ different situations. As an illustration of how NANDs and inverters may be used the following diagram shows how with 2 outputs from the computer we can decode $2^2 = 4$ outputs to our model railway.

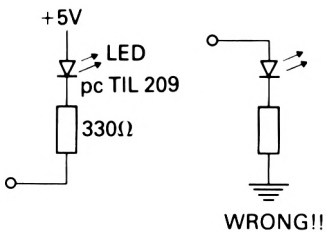
Hence by building up the circuits of NAND gates and inverters we can control $2^8 = 256$ points of our railway . . . have a good trip!


0_1
$\overline{0_1}$
2 outputs
0_2
$\overline{0_2}$
1
1
&
&
&
&
1
2
3
4
4 different
outputs (selected by combinations
of 0_1 and 0_2)

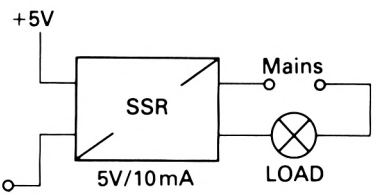


The next illustration shows you some common input and output circuits that can be directly coupled to the input and output pins of the interface. The components are also given for each circuit so that you can easily build it.

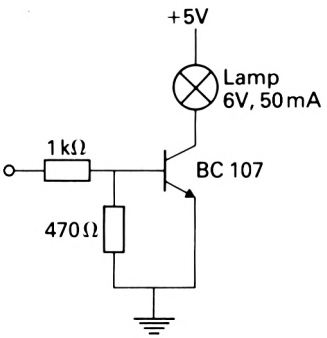
Some output devices



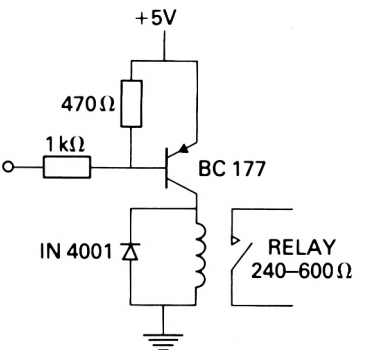
SIMPLE LED INTERFACE
Ø = on
1 = off



SOLID STATE RELAY
Ø = on 1 = off
(e.g. to drive a
motor or another
LOAD)

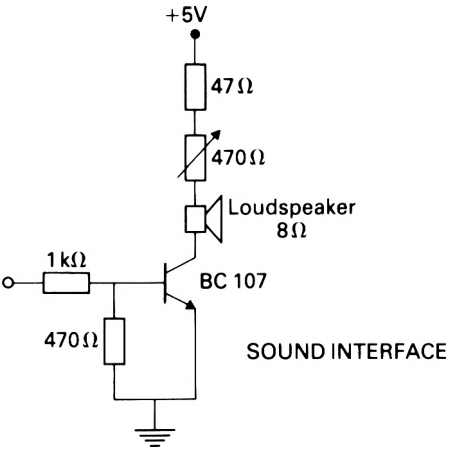


LAMP DRIVER
Ø = off
1 = on

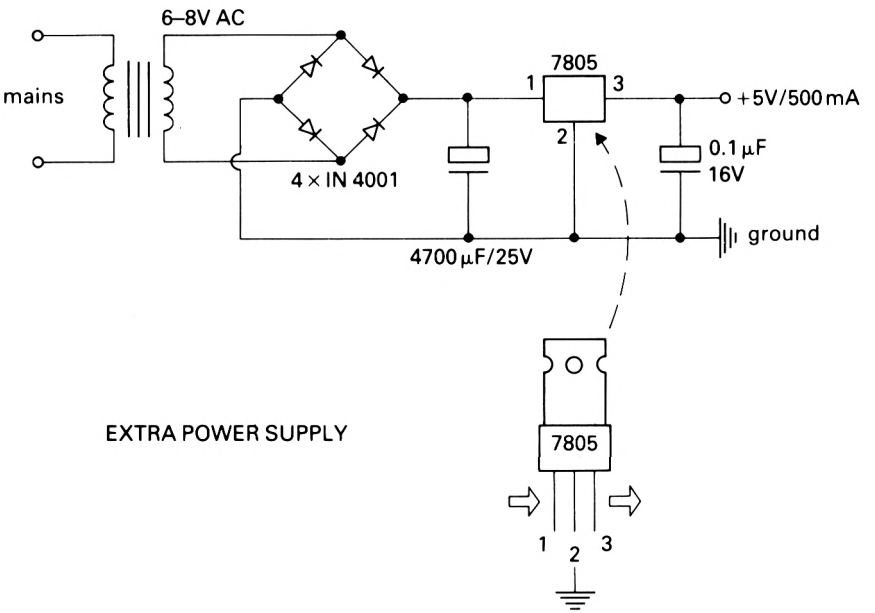


RELAY DRIVER
e.g. to control
a point of a
model train
1 = off
Ø = on

Some output devices

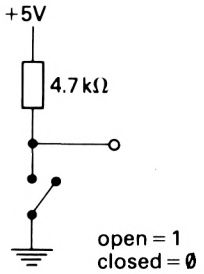


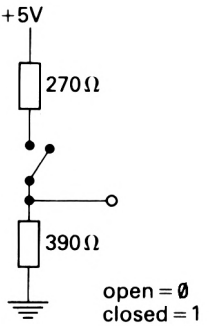
SOUND INTERFACE

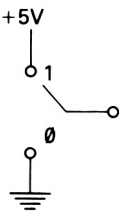


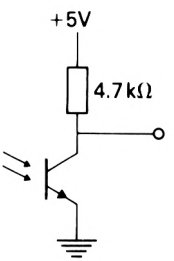
EXTRA POWER SUPPLY

Some input devices

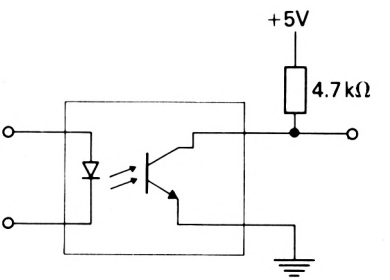








LIGHT SWITCH 2N5778



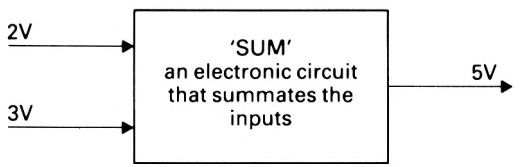
OPTO COUPLER MCT2E

CHAPTER 16

Converting your ZX computer into a very powerful analog computer

About 15 years ago every book on computers started with a section in which it was explained that there were two types of computers. The first type was the *digital* computer. With this computer all data is represented by series of zeros and ones. The ZX81/TS1000 and ZX Spectrum are obviously digital computers.

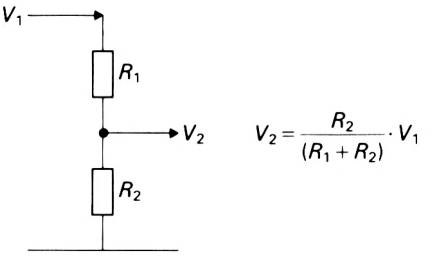
The second type of computer was the *analog* computer. The data values are now internally represented by physical quantities, for example electrical potentials. For instance the number 1 is represented by a voltage of 1V, the number 2 by a voltage of 2V, 3 by a voltage of 3V, etc. If two values have to be added, the corresponding potentials are summated by a special electronic circuit. The output of this circuit is again a voltage. The next figure illustrates this.



Analog computation of 2 + 3

These analog computers are still used today mainly because of two reasons. The first is that many mathematical solutions can be solved more easily with such analog computers. We shall come to this point later on. The second reason is that in some cases they beat the digital computer with respect to speed. This can be explained quite easily. If for instance the quotient of two values has to be found, the digital computer will always need a fixed number of elementary instructions. Each instruction needs a certain execution time T and roughly speaking a time nT is needed to execute n instructions. The analog computer does not use elementary instructions; it is the electronic circuitry that determines the speed.

Consider for example the next very simple circuit.



R_1 and R_2 represent resistors. It is obvious that this circuit represents a division. If for instance R_1 is taken equal to R_2 the value V_1 will be divided by 2. If we are dealing with ideal resistors (no capacitance) the operation 'dividing by 2' will be infinitely short!

In many cases we are interested in the use of analog computers because of their ability to simulate (resemble or mimic) complex physical systems and not because of their speed. Strangely enough in some applications computer scientists are now using digital computers running a program that *simulates* an analog computer! This will be the main subject of this chapter and we shall see how powerful our ZX computer can be as an analog computer. First an introduction is given to 'analog simulation'—a technique to examine the behaviour of complex physical (and other) systems. Then the program to convert our computer into an analog computer will be discussed. This program is partly based on TUTSIM, a program developed at the Twente University of Technology in the Netherlands. Analog simulation is a rather involved mathematical field, so if you are not very skilled in mathematics, perhaps you should skip this chapter . . . although there's no harm in trying it.

Analog computers and analog simulation

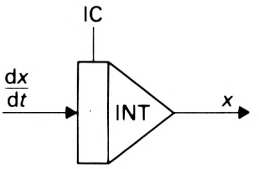
The 'keyboard' of an analog computer consists of a board with a large number of standard electronic circuits, which can be connected as we choose by means of wires. In this way a complex electronic circuit can be built that has a similar behaviour to the system that we want to examine.

For many systems we can easily find an electronic circuit that describes the behaviour of the physical system. For instance we can simulate not only all kinds of mechanical systems, but also biological systems and many others.

The basic circuits in an analog computer are as follows.

(1) Integrator (INT)

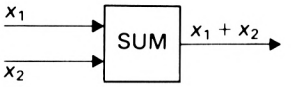
This circuit integrates the input signal. So if the input signal is represented by a time derivative dx/dt, the output signal is represented by x. This circuit is indicated by the symbol



IC defines the output x at time $t = 0$.

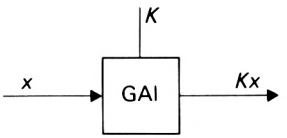
(2) Summer (SUM)

This circuit adds two signals. If the inputs are given as x_1 and x_2 the output is $x_1 + x_2$. This circuit is indicated by

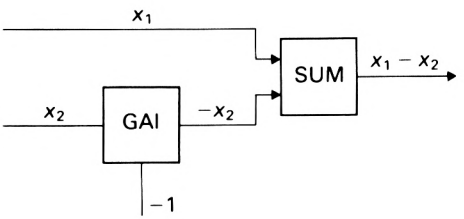


(3) Gain factor (amplifier or attenuator: GAI)

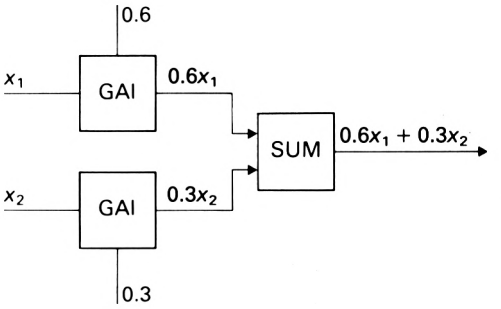
This circuit multiplies the input by a factor K. If K is greater than 1 ($K>1$) the signal is amplified. If K is less than 1 ($K<1$) the signal is attenuated (reduced). Of special interest is $K=-1$; in this case the input is inverted. This circuit is indicated by



The letters GAI stand for GAIN. By means of a summer and a gain factor we can easily construct a subtracting circuit:

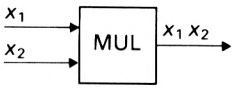


The next scheme shows how $0.6x_1 + 0.3x_2$ is obtained.

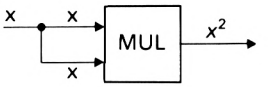


(4) Multiplier (MUL)

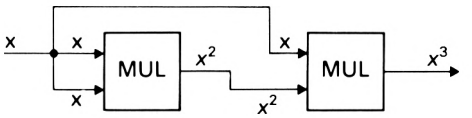
This circuit multiplies two input signals. It is indicated by



By means of this circuit we can easily obtain the square of a signal:

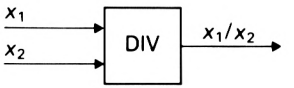


Or x to the power of 3:



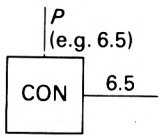
(5) Divider (DIV)

This circuit divides x_1 by x_2:



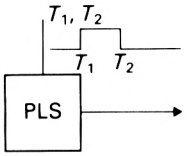
(6) Constant (CON)

This circuit generates a constant value which is depicted by P. Note that this circuit has no input.

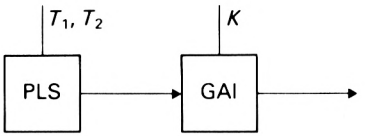


(7) Pulse (PLS)

Again this is a circuit without an input. This circuit generates a pulse between T_1 and T_2.

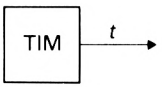


The amplitude of the pulse is always 1 but by means of a GAI block we can easily select any amplitude.



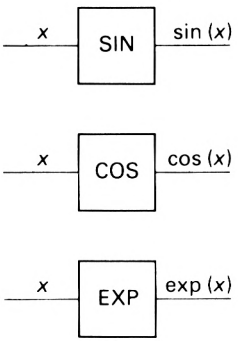
(8) Time (TIM)

This circuit generates a linearly increasing signal. Since time can also be considered as a linearly increasing 'signal' we call it the time circuit.



(9) Function circuits: SIN,COS,EXP

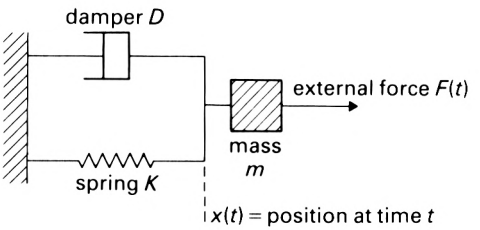
These circuits can be compared with the standard functions we met in chapter 5. The input is the argument of the function and the output is the result of the function:



The circuits (1) to (9) will be used as our basic circuits. Instead of 'circuit' we will use the word 'block' from now on.

Building an analog block diagram

The following section illustrates how we can construct a circuit of blocks in order to study a particular system. Let us consider the following mechanical system:



If $F(t)$ is a tractive force on the system then the following reaction forces are found:

$$\text{inertia force} = m\frac{d^2x}{dt^2}$$

$$\text{force of damper} = D\frac{dx}{dt}$$

$$\text{force of spring} = Kx$$

Since action equals reaction we can write

$$m\frac{d^2x}{dt^2} + D\frac{dx}{dt} + Kx = F(t) \qquad (1)$$

This formula represents a differential equation that can be solved (at least we hope so) if $F(t)$ and the initial acceleration d^2x/dt^2 and velocity dx/dt are known.

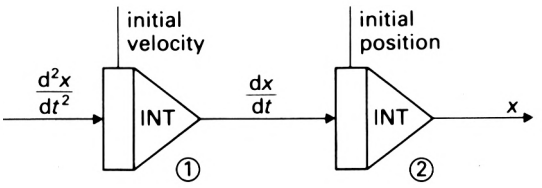
In a rather straight-forward way, analog simulation yields any solution of the equation that we are interested in. The interesting thing about it is that in fact we do not need any mathematics to understand the circuit. We can construct the circuit as if we are playing with LEGO bricks!

First we start by rewriting equation (1) in such a way that the highest derivative is on the left of the equal sign:

$$\frac{d^2x}{dt^2} = \frac{F(t) - D(dx/dt) - Kx}{m} \qquad (2)$$

The 'wrinkle' or dodge now is that we will construct the right hand side of this equation by means of our basic blocks and since the right hand side equals d^2x/dt^2 we can use it as the input of the block that will represent d^2x/dt^2. A little confusing . . .? Not at all. Just see how easy it really is.

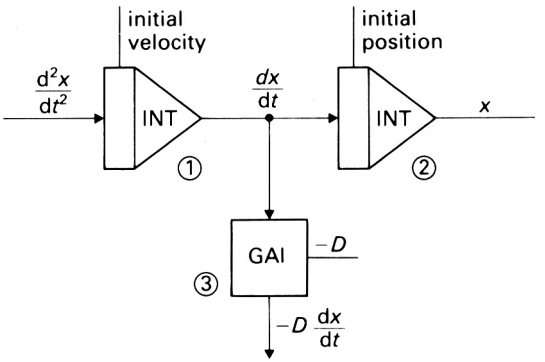
First we connect two integrators as follows.



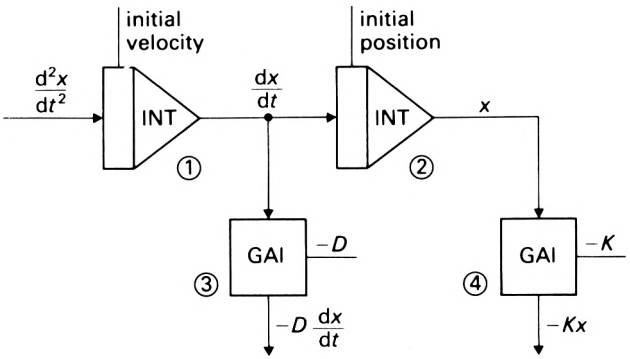
If the input of integrator 1 is d^2x/dt^2 then the output is dx/dt and if this signal is used as the input of the second integrator, the output of the latter will be x. Notice that we have numbered our blocks 1 and 2. By means of a GAI block we can construct

$$-D\frac{dx}{dt}$$

by using the output of integrator 1.

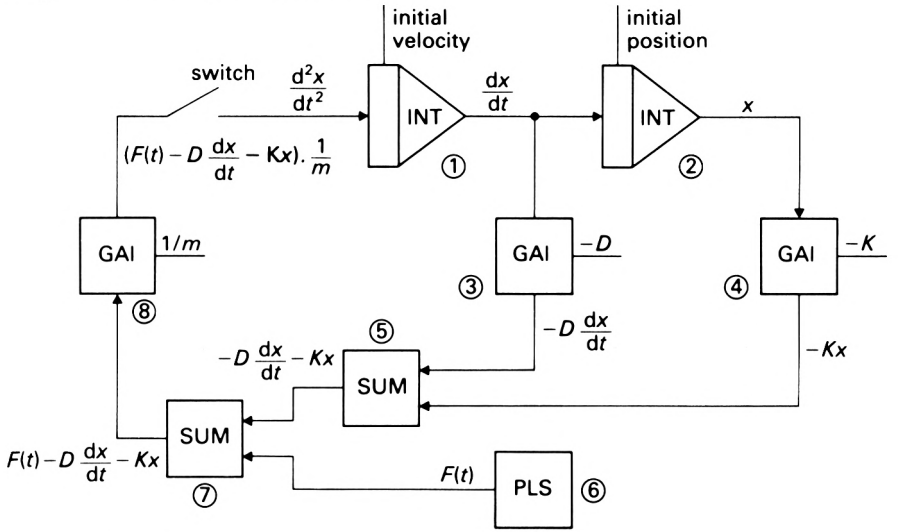


In a rather similar way $-Kx$ is obtained.



By means of a summer we obtain $-D\mathrm{d}x/\mathrm{d}t - Kx$. Suppose we want to study our mechanical system with a particular pulse as input. In this case the block PLS (pulse) has to be used and the output of PLS

represents $F(t)$. A second summer must be used to add $F(t)$ to $-D\mathrm{d}x/\mathrm{d}t - Kx$. Finally this 'signal' has to be multiplied by $1/m$ in order to obtain the complete expression of equation (2). Our complete diagram becomes as follows.

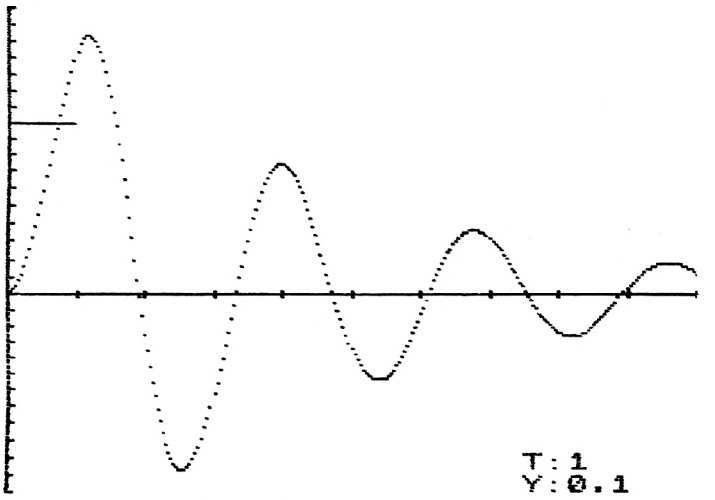


Complete scheme to simulate a mass damper spring system

Notice that if the switch is closed the input of integrator 1 is in fact

$$\frac{F(t) - D(\mathrm{d}x/\mathrm{d}t) - Kx}{m}$$

This means that our circuit is equivalent to the original differential equation which in fact describes the mechanical system to be studied (see equation (2)).



Result of simulation.

If the switch is closed the output voltages of integrators 1 and 2 will change as a function of time. The output of 2 exactly simulates the 'output' of the mechanical system (x-coordinate) if a pulse-like force (similar to the parameters of the PLS) acts on the system.

The figure (page 183) shows a result. In our example, after a short 'push' in one direction (pulse) the mass will oscillate about its original position. The amplitude of oscillation will die away (be damped) over a period of time.

Description of the simulation diagram

In this section we shall describe how the program SIMUL can be used in order to simulate all kinds of systems. A complete listing of this program is given at the end of the chapter. After the **RUN** command is given the screen shows

```
STRUCTURE
```

and this message indicates that the structure of the simulation diagram has to be entered. We shall see that this is done in a very 'elegant' way.

First notice that each block in the diagram is given a number. As a matter of fact this block number identifies the output of each block. Hence if we speak of signal '6' we refer to the output signal of block number 6. The complete structure of the diagram is indicated by a listing of all blocks according to the following rule:

```
block number; type of block; inputs
```

Note that first the block number is given, then the type of the block (**SUM, INT, GAI**, etc.) and finally the inputs are indicated (provided the block has inputs). The inputs are indicated by numbers and these numbers correspond to the blocks that generate the inputs. As an example we again consider the diagram on page 183. Block 5 is described by

```
5; SUM; 3; 4
```

The first number **5** indicates the block number, **SUM** indicates the type and **3** and **4** define the inputs. As you can see the inputs are the outputs of blocks **3** and **4**. Block **6** of this diagram has no inputs; it is simply indicated by

```
6; PLS
```

In this way we can describe our complete diagram of page 183 just by enumurating each block. We enter

```
1; INT; 8
2; INT; 1
3; GAI; 1
4; GAI; 2
5; SUM; 3; 4
6; PLS
7; SUM; 5; 6
8; GAI; 7
```

Each line is terminated by **ENTER** (or **NEWLINE**) and after the last line is entered we type **ENTER** once more to indicate that the complete structure is entered.

You can in fact give any number to a block provided that a particular number is not used twice. Another point of interest is that the sequence in which the blocks are listed is not important. You may for instance enter

```
3; GAI; 1
1; INT; 8
7; SUM; 5; 6
2; INT; 1
4; GAI; 2
8; GAI; 7
5; SUM; 3; 4
6; PLS
```

This listing defines a structure that is identical to the structure of the previous listing.

After the structure is entered our program automatically prompts:

```
PARAMETERS
```

We now have to enter the parameters by entering the block numbers together with the parameters. Not all the parameters are indicated on the diagram of page 183. The initial velocity and the initial position are taken as zero. This means that the parameters for both integrators are zero. Furthermore we choose (more of less arbitrarily): $D = 0.1$, $K = 1$ and finally $m = 0.2$ (hence $1/m = 5$). The pulse block has two parameters T_1 and T_2. T_1 is taken as 0.0 and T_2 1.0 seconds. These parameters are entered according to the following rule:

```
block number; parameters
```

If more parameters have to be entered (e.g. **PLS**) these parameters are separated by means of

```
;
```

We enter

```
1; 0
2; 0
3; - 0.1
4; - 1
6; 0.0; 1.0
8; 5
```

After the last line is entered we once more type **ENTER** (or **NEWLINE**) in order to indicate that the complete list of parameters is entered.

Again the sequence in which the block numbers are listed is not of importance. Now the complete diagram is entered; the computer knows the structure as well as the parameters. The computer shows

```
PLOTBLOCKS
```

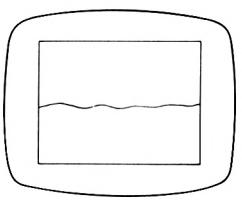
We now have to enter the block numbers whose outputs we want to examine. The output signals of these blocks will be displayed on the screen. It is interesting to see how the position (x = output of block 2) varies as a function of time together with the pulse (block number 6). We enter

```
2; 6
```

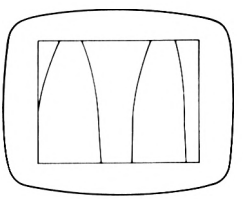
Now the screen shows

```
PLOTRANGE
```

We will use two values to indicate a minimum and a maximum value. All values to be plotted are related to these values. Usually we select the maximum and minimum by means of trial and error. If the difference between the values is too large the computed curve shows hardly any deflection. On the other hand if the difference is too small, a part of the result is out of the picture.



Difference between values of **PLOTRANGE** is too large



Difference between values of **PLOTRANGE** is too small

We try −1.2 and 1.7 (to be honest we tried a few other values before arriving at these):

```
- 1.2; 1.7
```

Now we have to indicate the time increments and the total duration for which the outputs have to be calculated. The screen shows

```
TIMING:
```

and we enter, for instance, 0.04 (Δt) and 10 (total duration)

```
0.04; 10
```

This means that 10.0/0.04 = 250 points will be calculated.
The screen now shows

```
GIVE COMMAND
```

We can enter any of the following commands.

Command	*Meaning*
E	terminate program (exit)
CP	change parameters
CB	change blocks
CR	change range
CT	change timing
SD	show drawing (execute!)

Note that we can make corrections just by entering CP, CB, CR or CT. If for instance a parameter has to be changed, the command CP is given and the new parameter is entered. By the way, if a line contains an error we immediately type the line again.

Of course we are anxious to see if the program works:

```
SD                                                    (and ENTER)
```

After a while the y and t axes are drawn. Then we see the generation of the curve . . . (beware . . . it takes some time!).

We can interrupt the computation by the BREAK key and in this case we can restart the program by

```
GOTO 1080
```

If the total result is plotted the computer shows

```
SAVE?
```

If we type ENTER the result will not be saved on cassette; if we type another key the screen will be copied.

Some special remarks

The program is far too long to explain in detail. The key to the solution is a look up table (array A(40,4)) and an array that specifies in which

sequence the blocks have to be computed (array B(40)). Array A(40,4) is defined according to

A(K,1)	A(K,2)	A(K,3)	A(K,4)
block number	type of block	inputs or	parameters

For example:

1	4	3	0.1

indicates: block 1 is an integrator (type number 4), the input is 3 and the parameter is 0.1. The types are indicated by numbers according to the following list.

Type	*Number*
TIM	1
CON	2
PLS	3
INT	4
GAI	5
SIN	6
COS	7
EXP	8
SUM	9
MUL	10
DIV	11

Lines 30–300 describe how the structure is entered and stored in array A. A$ is used as a buffer and each line that is entered is assigned to A$. The main part of lines 30–300 deals with the decoding of a line that is entered and with the storing of the codes in array A. Lines 400–660 describe the decoding of the list of parameters in a similar way.

In the next part the PLOTBLOCKS (lines 670–780), the PLOTRANGE (lines 790–940) and the TIMING (lines 950–1070) are entered. Lines 1080–1170 describe the part that decodes the commands. Lines 1190–1530 describe the part in which the blocks are ordered. The following principle has been used. First those blocks are selected that have a defined output at time t = 0 (TIM, CON, PLS and INT). The blocks are then ordered in such a way that a block is placed in the row if the inputs correspond to a block that is already there.

Array B finally describes the order in which the outputs of the successive blocks will be calculated. Lines 1555–1595 determine the plotting of the axes.

ZX81 and TS1000 owners should omit all the specific Spectrum plotting statements like INK (e.g. line 1555) and DRAW (e.g. line 1570). You will only need to use the PLOT statements, but remember that the plot range of the ZX81/TS1000 differs from that of the ZX Spectrum (see page 117).

The real computation now follows. First the initial (t = 0) output values of all the blocks are determined. Line 1830 controls the routine that computes the outputs of the blocks for each time. These values are stored in array S.

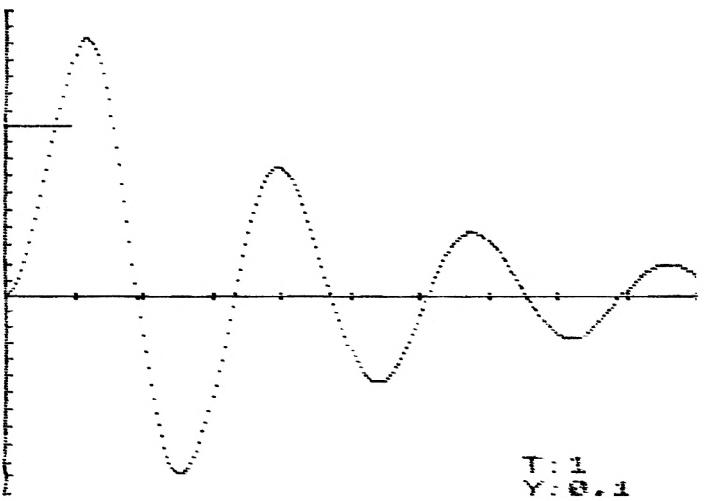
```
    5 DIM H$(2)
   10 DIM A(40,4)
   15 DIM P(5)
   20 DIM A$(40)
   25 DIM C$(33)
   26 LET C$="TIMCONPLSINTGAISINC
OSEXPSUMMULDIV"
   27 DIM D$(11)
   28 LET D$="22233333444"
   29 LET Z=0
   30 PRINT "STRUCTURE"
   31 PRINT
   38 LET M=0
   39 LET E=0
   40 INPUT A$
   50 IF A$(1)=" " THEN GO TO 400
   70 IF E>=40 THEN GO TO 0
   80 LET I=1
   90 GO SUB 2000
  100 LET A(M+1,1)=VAL A$(I TO J-
1)
  110 LET I=J+1
  120 GO SUB 2000
  130 IF (J-I)<>3 THEN GO TO 40
  140 LET G$=A$(I TO I+2)
  150 LET K=1
  160 LET L=1
  170 IF G$=C$(L TO L+2) THEN GO
TO 220
  180 LET K=K+1
  190 IF K>11 THEN GO TO 40
  200 LET L=L+3
  210 GO TO 170
  220 LET L=VAL D$(K)
  230 LET A(M+1,2)=K
  240 FOR N=3 TO L
  250 LET I=J+1
  260 GO SUB 2000
  265 IF A$(I)=" " THEN GO TO 40
  270 LET A(M+1,N)=VAL A$(I TO J-
1)
  280 NEXT N
  285 LET M=M+1
  287 LET E=E+1
  290 PRINT A$( TO J-1)
  300 GO TO 40
  400 PRINT
  410 PRINT "PARAMETERS"
  420 PRINT
  430 INPUT A$
  440 IF A$(1)=" " THEN GO TO 650
  450 LET I=1
  460 GO SUB 2000
  470 IF A$(I)=" " THEN GO TO 430
  480 LET K=VAL A$(I TO J-1)
  490 FOR N=1 TO E
  500 IF A(N,1)=K THEN GO TO 530
  510 NEXT N
  520 GO TO 430
```

```
 530 LET K=A(N,2)
 532 IF K=3 THEN GO TO 600
 534 IF (K>5) OR (K=1) THEN GO T
O 430
 540 LET I=J+1
 550 GO SUB 2000
 560 IF A$(I)=" " THEN GO TO 430
 570 LET A(N,4)=VAL A$(I TO J-1)
 580 PRINT A$( TO J-1)
 590 GO TO 430
 600 LET I=J+1
 610 GO SUB 2000
 620 IF A$(I)=" " THEN GO TO 430
 630 LET A(N,3)=VAL A$(I TO J-1)
 640 GO TO 540
 650 IF Z=1 THEN GO TO 1080
 660 PRINT
 670 PRINT "PLOTBLOCKS"
 680 PRINT
 690 INPUT A$
 700 LET I=1
 710 LET Q=0
 720 GO SUB 2000
 730 IF (Q>5) OR (A$(I)=" ") THE
N GO TO 765
 740 LET Q=Q+1
 750 LET P(Q)=VAL A$(I TO J-1)
 755 LET I=J+1
 760 GO TO 720
 765 IF Q=0 THEN GO TO 690
 767 PRINT A$( TO I)
 770 IF Z=1 THEN GO TO 1080
 780 PRINT
 790 PRINT "PLOTRANGE"
 800 PRINT
 810 INPUT A$
 820 LET I=1
 830 GO SUB 2000
 840 IF A$(I)=" " THEN GO TO 810
 850 LET R=VAL A$(I TO J-1)
 860 LET I=J+1
 870 GO SUB 2000
 880 IF A$(I)=" " THEN GO TO 810
 890 LET U=VAL A$(I TO J-1)
 900 PRINT A$( TO J-1)
 910 LET V=175/(U-R)
 920 LET W=-V*R
 930 IF Z=1 THEN GO TO 1080
 940 PRINT
 950 PRINT "TIMING"
 960 PRINT
 970 INPUT A$
 980 LET I=1
 990 GO SUB 2000
1000 IF A$(I)=" " THEN GO TO 970
1010 LET D1=VAL A$(I TO J-1)
1020 LET I=J+1
1030 GO SUB 2000
1040 IF A$(I)=" " THEN GO TO 970
1050 LET T1=VAL A$(I TO J-1)
1060 PRINT A$( TO J-1)
1070 LET Z=1
1080 PRINT
1100 INPUT "GIVE COMMAND",H$
1115 LET ST=1/D1
```

```
1120 IF H$="E " THEN GO TO 9999
1130 IF H$="CP" THEN GO TO 400
1140 IF H$="CB" THEN GO TO 660
1150 IF H$="CR" THEN GO TO 780
1160 IF H$="CT" THEN GO TO 940
1170 IF (H$<>"SD") AND (H$<>"ST"
) THEN GO TO 1090
1180 LET Y1=ST*T1/255
1190 DIM B(40)
1200 FOR I=1 TO E
1210 LET B(I)=I
1220 NEXT I
1230 LET J=1
1240 LET I=E
1250 IF A(B(J),2)<5 THEN GO TO 1
330
1260 LET K=B(J)
1270 FOR L=J TO I-1
1280 LET B(L)=B(L+1)
1290 NEXT L
1300 LET B(I)=K
1310 LET I=I-1
1320 IF J<=I THEN GO TO 1250
1330 LET J=J+1
1340 IF J<=I THEN GO TO 1250
1350 FOR J=I+1 TO E
1360 LET L=0
1370 LET K=3
1380 GO SUB 3000
1390 IF L=0 THEN GO TO 1480
1400 IF A(B(J),2)<9 THEN GO TO 1
450
1410 IF L=0 THEN GO TO 1480
1420 LET K=4
1430 LET L=0
1440 GO SUB 3000
1450 IF L=0 THEN GO TO 1480
1460 NEXT J
1470 GO TO 1540
1480 LET G=B(J)
1490 FOR N=J+1 TO E
1500 LET B(N-1)=B(N)
1510 NEXT N
1520 LET B(E)=G
1530 GO TO 1360
1540 DIM S(40)
1550 CLS
1555 INK 0
1556 IF H$="ST" THEN GO TO 1600
1560 PLOT 0,0
1570 DRAW 0,175
1580 PLOT 0,0
1590 DRAW 255,0
1595 GO SUB 5000
1600 FOR J=1 TO I
1610 LET BJ=B(J)
1620 IF A(BJ,2)=3 THEN GO TO 165
0
1630 LET S(A(BJ,1))=A(BJ,4)
1640 GO TO 1690
1650 IF A(BJ,3)=0 THEN GO TO 168
0
1660 LET S(A(BJ,1))=0
1670 GO TO 1690
1680 LET S(A(BJ,1))=1
```

```
1690 NEXT J
1700 LET J=I+1
1710 LET M=0
1720 LET T=0
1740 IF J>E THEN GO TO 1830
1750 GO TO 1770
1760 LET J=1
1770 LET BJ=B(J)
1780 LET ABJ1=A(BJ,1)
1790 LET ABJ2=A(BJ,2)
1800 LET ABJ3=A(BJ,3)
1810 LET ABJ4=A(BJ,4)
1820 GO TO (3950+50*ABJ2)
1830 IF J>=E THEN GO TO 1860
1840 LET J=J+1
1850 GO TO 1770
1870 IF H$="ST" THEN GO TO 1910
1880 FOR I=1 TO 0
1881 LET SPI=S(P(I))
1882 IF SPI>U THEN GO TO 1886
1883 IF SPI>R THEN GO TO 1890
1884 LET SPI=R
1885 GO TO 1890
1886 LET SPI=U
1890 PLOT INK I;M/Y1,W+SPI*V
1895 NEXT I
1900 GO TO 1950
1910 PRINT T;
1920 FOR I=1 TO 0
1930 PRINT " ";S(P(I));
1940 NEXT I
1945 PRINT
1950 LET T=T+D1
1960 LET M=M+1
1970 IF M<=T1*ST THEN GO TO 1760
1971 INPUT "SAVE?",A$
1972 IF A$(1)=" " THEN GO TO 108
0
1973 SAVE A$( TO 10) SCREEN$
1980 GO TO 1080
2000 LET J=I
2010 LET J=J+1
2020 IF (J<40) AND (A$(J)<>";")
AND (A$(J)<>" ") THEN GO TO 2010
2030 RETURN
3000 LET J2=1
3010 IF L=1 THEN GO TO 3070
3020 IF J2=J THEN GO TO 3070
3030 IF A(B(J),K)<>A(B(J2),1) TH
EN GO TO 3050
3040 LET L=1
3050 LET J2=J2+1
3060 GO TO 3010
3070 RETURN
4000 LET S(ABJ1)=T
4010 GO TO 1830
4050 GO TO 1830
4100 IF (T>ABJ4) OR (T<ABJ3) THE
N GO TO 4130
4110 LET S(ABJ1)=1
4120 GO TO 1830
4130 LET S(ABJ1)=0
4140 GO TO 1830
4150 LET S(ABJ1)=S(ABJ1)+S(ABJ3)
*D1
4160 GO TO 1830
```

```
4200 LET S(ABJ1)=S(ABJ3)*ABJ4
4210 GO TO 1830
4250 LET S(ABJ1)=SIN S(ABJ3)
4260 GO TO 1830
4300 LET S(ABJ1)=COS S(ABJ3)
4310 GO TO 1830
4350 LET S(ABJ1)=EXP S(ABJ3)
4360 GO TO 1830
4400 LET S(ABJ1)=S(ABJ3)+S(ABJ4)
4410 GO TO 1830
4450 LET S(ABJ1)=S(ABJ3)*S(ABJ4)
4460 GO TO 1830
4500 IF S(ABJ4)=0 THEN GO TO 453
0
4510 LET S(ABJ1)=S(ABJ3)/S(ABJ4)
4520 GO TO 1830
4530 PRINT "OVERFLOW IN BLOCK ";
BJ
4540 GO TO 1080
5000 LET LNT=LN 10
5005 LET A1=10↑(INT ((LN T1)/LNT
-0.3))
5007 PRINT AT 20,24;"T:";A1
5010 FOR N=0 TO INT (T1/A1)
5020 PLOT N*A1*255/T1,W-1
5025 PLOT N*A1*255/T1,W+1
5030 NEXT N
5040 LET A1=10↑(INT ((LN (U-R))/
LNT-0.5))
5045 PRINT AT 21,24;"Y:";A1
5047 PRINT AT 0,0;
5050 FOR N=0 TO INT ((U-R)/A1)
5060 PLOT 1,N*A1*175/(U-R)
5065 PLOT 2,N*A1*175/(U-R)
5070 NEXT N
5080 RETURN
```



5-65

PART THREE

Programs

If not stated otherwise, programs can be run either on a ZX81/TS1000 (1K) or a ZX Spectrum.

HISTOGRAM
(ZX81 and TS1000 only)

This program generates an histogram of 10 cells. First the program asks the user to enter a minimum and a maximum and then the number of values has to be entered.

Example

```
MIN = 0
MAX = 10
NUMBER = 5
```

Now the computer asks for the successive values to be entered by displaying 1 = etc. If we list the following values

```
1 = 4
2 = 5
3 = 4
4 = 4
5 = 4
```

the computer will display the histogram:

```
0
0
0
4 ■■■■
0
1 ■
0
0
0
0
```

Program

```
 10 DIM C(10)
 20 PRINT "MIN=";
 30 INPUT A
 40 PRINT A
 50 PRINT "MAX=";
 60 INPUT B
 70 PRINT B
 80 PRINT "NUMBER=";
 90 INPUT N
100 PRINT N
110 FOR K=1 TO N
120 SCROLL
130 PRINT K;"=";
140 INPUT X
150 PRINT X
160 LET J=INT ((X-A)/(B-A)*10)+
1
170 LET C(J)=C(J)+1
180 NEXT K
190 CLS
200 FOR K=1 TO 10
210 PRINT C(K);TAB 4;
220 FOR J=1 TO C(K)
230 PRINT "■";
240 NEXT J
250 PRINT
260 NEXT K
```

DECIMAL TO HEX

This program converts decimal numbers into hexadecimal numbers. Hexadecimal numbers are used in assembly language programs. After the number is entered (line **10**) the number of digits is determined (line **20**). The decimal number is now divided by powers of 16 and at each step the difference between the original value and the power of 16 is taken as the starting point for a new division (lines **40–100**). If an intermediate result is >9 the letters **A...F** are used (line **80**) in such a way that **10** becomes **A**, 11 becomes **B**, 12 becomes **C**, 13 becomes **D**, 14 becomes **E** and 15 becomes **F**.

Program

```
   5 PRINT "ENTER DECIMAL NUMBER
=";
  10 INPUT G
  15 PRINT G
  20 LET N=LEN STR$ G
  30 DIM H(N)
  35 PRINT "HEX NUMBER=";
  40 FOR K=1 TO N
  50 LET H(K)=INT (G/(16**(N-K))
)
  60 LET G=G-H(K)*16**(N-K)
  70 LET Z$=STR$ H(K)
  80 IF H(K)>9 THEN LET Z$=CHR$
(CODE ("A")+H(K)-10)
  90 PRINT Z$;
 100 NEXT K
```

Example

```
ENTER DECIMAL NUMBER=1251
HEX NUMBER=04E3
```

Notice that ** (line **50,60**) is indicated by ↑ on the ZX Spectrum.

HEX TO DECIMAL

This program converts hexadecimal numbers into decimal numbers. It is of special interest if we need to calculate values that have to be POKEd in order to enter machine code programs. The key to the program is line 150 in which the hexadecimal digits are used to indicate powers of 16. The rest of the program deals with the problem of how the letters A–F can be converted to 10–15.

Program

```
 10 PRINT "SPECTRUM Y OR N"
 20 INPUT Q$
 30 LET C=37.5
 40 IF Q$="Y" THEN LET C=57.5
 50 LET B=28
 60 IF Q$="Y" THEN LET B=87
 70 PRINT "HEX NUMBER=";
 80 INPUT A$
 90 PRINT A$
100 LET S=0
110 LET L=LEN A$
120 FOR K=1 TO L
130 IF CODE A$(K)>C THEN LET V=
CODE A$(K)-B
140 IF CODE A$(K)<C THEN LET V=
VAL A$(K)
150 LET S=S+V*16**(L-K)
160 NEXT K
170 PRINT "DECIMAL NUMBER=";S
```

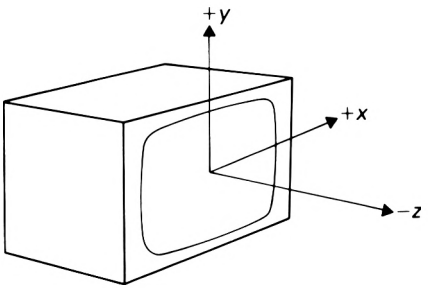
Notice the symbol ** is indicated on the ZX Spectrum by ↑.

Example

```
SPECTRUM Y OR N
HEX NUMBER=4E3
DECIMAL NUMBER=1251
```

REAL 3D
(ZX Spectrum only)

This program draws a cube in a realistic three dimensional way. Two colours, red and green, are used: one image in one colour represents the view of the cube seen by one eye. So if you use spectacles with one green and one red glass you will observe a three dimensional cube. The next figure shows the screen of the television and the coordinates x, y and z.



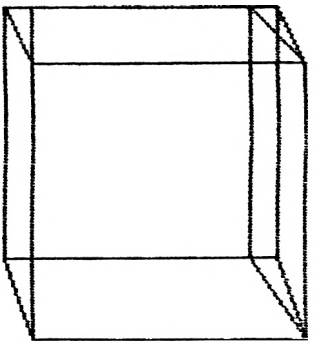
The program first asks for the coordinates of the left and the right eye. A number of STOP statements are inserted in order to demonstrate how the figures are constructed. Use the CONTINUE command to proceed after a STOP. In chapter 13 an extensive description is given.

Program

```
   10 CLEAR
   15 RESTORE
   20 DIM x(12,2,2)
   30 DIM y(12,2,2)
   40 LET deltax=127
   41 LET deltay=87
   42 INPUT AT 0,0;"LO-x= ";lox;A
T 1,0;"LO-y= ";loy;AT 2,0;"LO-z=
 ";loz
   43 INPUT AT 0,0;"RO-x=: ";rox;
AT 1,0;"RO-y= ";roy;AT 2,0;"RO-z
= ";roz
   50 FOR i=1 TO 12
   60 READ x1: READ y1: READ z1
   70 READ x2: READ y2: READ z2
   80 LET labl=-loz/(z1-loz)
   85 LET labr=-roz/(z1-roz)
   90 LET y(i,1,1)=loy+labl*(y1-l
oy)
   95 LET x(i,1,1)=lox+labl*(x1-l
ox)
   97 LET labl=-loz/(z2-loz)
  100 LET y(i,2,1)=loy+labl*(y2-l
oy)
  102 LET y(i,2,1)=y(i,2,1)-y(i,1
,1)
  103 LET y(i,1,1)=y(i,1,1)+delta
y
```

```
 105 LET x(i,2,1)=lox+labl*(x2-l
ox)
 107 LET x(i,2,1)=x(i,2,1)-x(i,1
,1)
 108 LET x(i,1,1)=x(i,1,1)+delta
x
 110 LET y(i,1,2)=roy+labr*(y1-r
oy)
 115 LET x(i,1,2)=rox+labr*(x1-r
ox)
 117 LET labr=-roz/(z2-roz)
 120 LET y(i,2,2)=roy+labr*(y2-r
oy)
 122 LET y(i,2,2)=y(i,2,2)-y(i,1
,2)
 123 LET y(i,1,2)=y(i,1,2)+delta
y
 125 LET x(i,2,2)=rox+labr*(x2-r
ox)
 127 LET x(i,2,2)=x(i,2,2)-x(i,1
,2)
 128 LET x(i,1,2)=x(i,1,2)+delta
x
 160 NEXT i
 200 PAPER 0: BORDER 0: CLS
 205 INK 4: LET k=1
 220 GO SUB 500
 225 STOP
 230 INK 2: LET k=2
 240 GO SUB 500
 245 STOP
 250 GO TO 10
 500 PLOT x(1,1,k),y(1,1,k)
 510 DRAW x(1,2,k),y(1,2,k)
 520 DRAW x(2,2,k),y(2,2,k)
 530 DRAW x(3,2,k),y(3,2,k)
 540 DRAW x(4,2,k),y(4,2,k)
 545 STOP
 550 DRAW x(5,2,k),y(5,2,k)
 560 DRAW x(6,2,k),y(6,2,k)
 570 DRAW x(7,2,k),y(7,2,k)
 575 STOP
 580 PLOT x(8,1,k),y(8,1,k)
 590 DRAW x(8,2,k),y(8,2,k)
 591 DRAW x(6,2,k),y(6,2,k)
 592 DRAW x(9,2,k),y(9,2,k)
 593 DRAW x(10,2,k),y(10,2,k)
 594 DRAW x(11,2,k),y(11,2,k)
 595 DRAW x(2,2,k),y(2,2,k)
 596 DRAW x(12,2,k),y(12,2,k)
 597 RETURN
 600 DATA -100,-50,0,-100,50,0
 610 DATA -100,50,0,0,50,0
 620 DATA 0,50,0,0,-50,0
 630 DATA 0,-50,0,-100,-50,0
 640 DATA -100,-50,0,-100,-50,10
0
 650 DATA -100,-50,100,0,-50,100
 660 DATA 0,-50,100,0,-50,0
 670 DATA -100,50,100,-100,-50,1
00
 680 DATA 0,-50,100,0,50,100
 690 DATA 0,50,100,-100,50,100
 700 DATA -100,50,100,-100,50,0
 710 DATA 0,50,0,0,50,100
```

Example

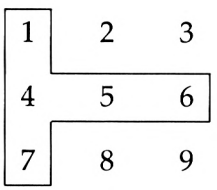


FLAT CUBE

This is certainly an exasperating game. I saw it for the first time on a CASIO calculator. Consider the following matrix:

1	2	3
4	5	6
7	8	9

Through each number we can draw a horizontal and a vertical line. For example, in the case where the lines are determined by the number 4, we have:



In this game we have to enter numbers (1–9) and each time two lines are indicated as illustrated above.

When the game starts we see a random array, for instance:

2	2	4
1	1	2
1	5	4

Now we enter one of the numbers 1–9 and each number that lies on the vertical and horizontal lines of our original matrix is incremented by 1. If a 5 is incremented by 1 it becomes 0. For instance if we enter 7 the following numbers are incremented:

2	2	4		3	2	4
1	1	2	result →	2	1	2
1	*5*	*4*		2	0	5

The aim of the game is to get:

0	0	0
0	0	0
0	0	0

Program

```
    1 DIM B(9)
    3 LET A$="123471235812369456172584636945147897892578936"
    5 RAND
    7 FOR K=1 TO 9
    9 LET N=INT (RND*9+1)
   11 GOSUB 23
   13 NEXT K
   14 GOSUB 39
   15 PRINT "MOVE="
   16 INPUT N
   17 GOSUB 23
   19 GOSUB 39
   21 GOTO 15
   23 LET M=(N-1)*5+1
   25 LET H$=A$(M TO M+4)
   27 FOR J=1 TO 5
   29 LET W=VAL H$(J)
   31 LET B(W)=B(W)+1
   32 IF B(W)>5 THEN LET B(W)=B(W)-5
   35 NEXT J
   37 RETURN
   39 CLS
   40 PRINT B(1);B(2);B(3)
   41 PRINT B(4);B(5);B(6)
   42 PRINT B(7);B(8);B(9)
   43 RETURN
```

Some remarks

The array `B` contains the values of the matrix. `A$` contains all the fields that have to be incremented for each key, e.g. key `1` corresponds to the fields `1,2,3,4` and `7`. This string is decoded in lines `23` and `25`. First the computer starts with some random moves that are not displayed in order to obtain a matrix with apparently random numbers (lines `7`–`13`). Lines `27`–`35` indicate that each field has to be incremented by `1` and if a field is `5` the result has to be `0`. Lines `39`–`43` describe the printing of the fields.

TEST YOUR MEMORY

This game is a memory test. After the RUN command 4 numbers are displayed. Just try to remember how many ones, twos, etc. are displayed. After 6 seconds the numbers disappear and the screen shows

```
HOW MANY 0?
```

and you have to enter how many zeros were displayed, etc. In the program we use two arrays A and B. A contains 4 random numbers and B contains for each digit the appropriate number. The 4 random numbers are generated by lines 40–90. An extra test is made to check if a number starts with a zero (lines 60 and 70). Such numbers are not allowed for practical reasons; we only want numbers that consist of 5 digits and that start with 1–9. Line 100 determines how long the random numbers are displayed, so if your memory is not as good as it used to be just change this line! Lines 110–180 describe how the values of B are obtained. Lines 200–270 ask the user to enter his answers and finally the result is displayed.

Program

```
 10 DIM A(4)
 20 DIM B(10)
 30 RAND
 40 FOR K=1 TO 4
 50 LET A(K)=INT (RND*1E5)
 60 LET Q=INT (A(K)/1E4)
 70 IF Q=0 THEN GOTO 40
 80 PRINT A(K)
 90 NEXT K
100 PAUSE 1000
110 FOR K=1 TO 4
120 LET V$=STR$ A(K)
130 FOR J=1 TO 5
140 LET H$=V$(J)
150 LET U=VAL H$
160 LET B(U+1)=B(U+1)+1
170 NEXT J
180 NEXT K
190 CLS
200 LET R=0
210 FOR K=0 TO 9
220 PRINT "HOW MANY ";K;"?";
230 INPUT N
240 PRINT N
250 IF N=B(K+1) THEN PRINT "OK"
260 IF N=B(K+1) THEN LET R=R+1
270 NEXT K
280 PRINT "RESULT=";R
```

Example

```
40411
30920
19108
33235
```

After 6 seconds these numbers disappear.

```
HOW MANY 0?2
OK
HOW MANY 1?1
OK
HOW MANY 2?1
HOW MANY 3?2
HOW MANY 4?2
OK
HOW MANY 5?1
HOW MANY 6?2
HOW MANY 7?3
OK
HOW MANY 8?2
HOW MANY 9?1
OK
RESULT=5
```

HOFSTADTER
(For mathematicians!)

Ever since Douglas Hofstadter wrote *Gödel, Escher, Bach: The eternal golden braid*, described as a 'metaphorical fugue on minds and machines in the spirit of Lewis Carroll', many people have developed an interest in *recursive functions*.

Hofstadter speaks of a chaotic sequence and introduces the series which follows – it is an example of recursion in number theory which leads to a small mystery.

Maybe the word 'recursive' is new to you. It means that every new term in a series is obtained by combining old terms in a defined way. In this case the series is defined by the formula

$$Q(n) = Q[n - Q(n-1)] + Q[n - Q(n-2)] \qquad \text{for } n > 2$$

$$Q(1) = Q(2) = 1$$

Indeed such a series leads to rather astonishing results as you can find out by means of the following program.

Program

```
 10 PRINT "NUMBER OF TERMS=";
 20 INPUT K
 30 PRINT K
 40 DIM Q(K)
 50 LET Q(1)=1
 60 LET Q(2)=2
 70 PRINT 1,1
 80 PRINT 2,1
 90 FOR N=3 TO K
100 LET T1=Q(N-Q(N-1))
110 LET T2=Q(N-Q(N-2))
120 LET Q(N)=T1+T2
130 PRINT N,Q(N)
140 NEXT N
```

Example

```
NUMBER OF TERMS=16
1                 1
2                 1
3                 3
4                 3
5                 4
6                 5
7                 5
8                 6
9                 6
10                6
11                8
12                8
13                8
14                10
15                9
16                10
```

Another strange series that leads to a very unexpected result is generated by the following program. (Beware . . . maybe it's the machine!)

```
 10 PRINT "NUMBER OF TERMS=";
 20 INPUT K
 30 PRINT K
 40 DIM Q(K)
 50 LET Q(1)=1
 60 LET Q(2)=2
 70 PRINT 1,1
 80 PRINT 2,1
 90 FOR N=3 TO K
100 LET T1=Q(Q(N-1))
110 LET T2=Q(Q(N-2))
120 LET Q(N)=T1+T2
130 PRINT N,Q(N)
140 NEXT N
```

```
NUMBER OF TERMS=20
1                    1
2                    1
3                    3
4                    5
5                    3
6                    6
7                    9
8                    6
9                    6
10                   12
11                   6
12                   6
13                   12
14                   12
15                   12
16                   12
17                   12
18                   12
19                   12
20                   12
```

LIVES

The well known program LIFE is usually considered to be a game but in fact it is a simulation that demonstrates the successive generations of cells. Each cell is represented on the screen by a black square. The game was originally introduced by John Horton Conway in *Scientific American* (October, 1970). This version is written by A. Stolmeÿer and it is based on a version by W. Englander published in *Byte* (December, 1978). Instead of a local pattern of living cells (black squares), our starting point will be a collection of patterns. This is the reason why we called this version LIVES. The rules of birth, survival and death are as follows. A cell is born if in the preceding generation exactly three of the eight neighbours are living cells. If two or three neighbours are living cells the cell will survive, otherwise it becomes a dead cell.

Watch and see how the population alters from generation to generation.

Program

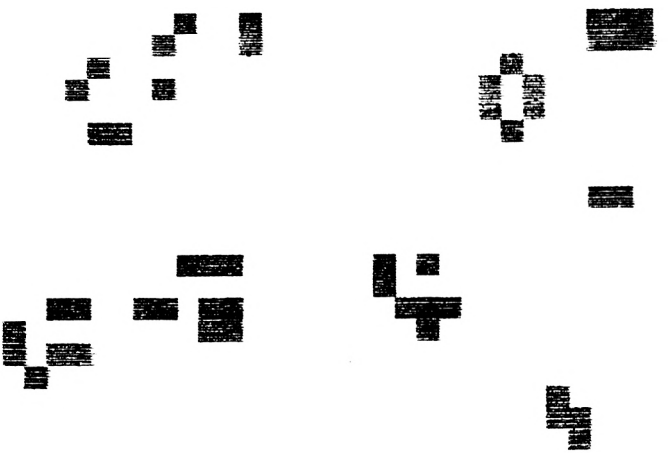
LIFE

```
    1 DIM A(32,22): DIM B(32,22)
    2 INK 0: PAPER 7: BORDER 1: C
LS : PRINT "One moment please"
    3 RANDOMIZE : FOR X=1 TO 32:
FOR Y=1 TO 22
    5 LET A(X,Y)=INT (RND*14)
    6 NEXT Y: NEXT X
   19 PRINT AT 0,0;
   20 FOR Y=1 TO 22: FOR X=1 TO 3
2
   21 IF A(X,Y)<10 THEN PRINT " "
;: GO TO 23
   22 PRINT "■";
   23 LET B(X,Y)=0
   24 NEXT X: PRINT : NEXT Y
   27 FOR X=2 TO 31: FOR Y=2 TO 2
1
   28 LET AW=A(X,Y)
   29 IF (AW<>3) AND (AW<>13) AND
  (AW<>14) THEN GO TO 34
   30 LET B(X,Y)=B(X,Y)+10
   31 FOR W=X-1 TO X+1: FOR V=Y-1
  TO Y+1
   32 LET B(W,V)=B(W,V)+1
   33 NEXT V: NEXT W
   34 NEXT Y: NEXT X
   36 PRINT AT 0,0;
   37 FOR Y=1 TO 22: FOR X=1 TO 3
2
   38 IF B(X,Y)<10 THEN PRINT " "
;: GO TO 40
   39 PRINT "■";
   40 LET A(X,Y)=0
   41 NEXT X: PRINT : NEXT Y
   44 FOR X=2 TO 31: FOR Y=2 TO 2
1
```

```
 45 LET BW=B(X,Y)
 46 IF (BW<>3) AND (BW<>13) AND
(BW<>14) THEN GO TO 51
 47 LET A(X,Y)=A(X,Y)+10
 48 FOR W=X-1 TO X+1: FOR V=Y-1
TO Y+1
 49 LET A(W,V)=A(W,V)+1
 50 NEXT V: NEXT W
 51 NEXT Y: NEXT X
 55 GO TO 19
 56 STOP
```

An example of two successive generations:

Generation n



Generation n + 1

BEWARE OF THE SNAKE
(ZX81/TS1000 only)

Your task is simply to pass the snake! After the RUN command you will see a snake and a zero that represents your position. You can control this position by using the keys 6 and 8. The snake moves towards a wall and you must pass the snake before it reaches the wall. In addition to the snake black holes will appear . . . do not fall in these holes!

Program

```
  1 LET C=0
  2 CLS
  3 FOR K=1 TO 15
  5 PRINT AT K,10;"■"
  7 NEXT K
 10 LET A=16398
 20 LET N=0
 30 LET M=7
 40 LET P=N
 50 LET Q=N
 60 PRINT AT P,Q;" "
 65 PRINT AT RND*5,RND*9;"■"
 70 LET P=P+(INKEY$="6")
 80 LET Q=Q+(INKEY$="8")
 90 PRINT AT 3,0;C
100 PRINT AT P,Q;
110 IF PEEK (PEEK A+256*PEEK (A
+1))=128 THEN GOTO 500
120 PRINT "Q";AT M,N;"■"
125 IF INT N=10 OR Q=10 THEN GO
TO 180
130 LET N=N+RND
140 LET M=M+RND*2-1
145 LET C=C+1
170 GOTO 60
180 IF INT M-P<=0 THEN GOTO 2
500 PRINT "KILLED"
```

Some remarks

Line 10 together with line 110 are used in order to check if the field that is to be occupied by a zero is already a black hole. PEEK A + 256 * PEEK(A + 1) with A = 16398 leads to an address that contains the code of the next field that will be used in a PRINT statement. A black square corresponds to 128. Lines 3–7 cause a wall to be printed. Line 60 erases the previous position of the zero. Line 65 causes black holes to be randomly printed. P and Q determine the next position of the zero. Line 120 contains PRINT statements—both your position and the snake's position (coordinates M,N) are printed. Line 145 shows the counting variable C, which counts the score. If you reach the wall the program checks whether you have passed the snake.

BINOMIAL

Let us consider an experiment in which the result or outcome depends on chance. For example when throwing a dice the chance of 6 being thrown is 1/6. Mathematically we can determine exactly how many times a certain outcome will occur during a certain number of experiments by a formula known as the binomial distribution. For example when throwing a dice 5 times, the chance of 6 occurring twice is given by the formula

$$f(x) = \binom{n}{x} p^x (1-p)^x$$

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}$$

where $x = 2$ (6 occurs *twice*)
$p = 1/6$ (chance of 6 in each throw)
$n = 5$ (number of throws)

Program

```
  10 PRINT AT 10,7; PAPER 1; INK
 7;"BINOMIAL DISTRIBUTION"
  20 INPUT AT 0,0;"n= ";n;AT 1,0
;"p= ";p
  23 LET m=n
  25 GO SUB 200
  26 LET nf=facul
  27 LET d$="
                "
  30 PRINT AT 13,0;"n= ";n;AT 13
,8;"p= ";p
  40 INPUT "X= ";x
  50 LET m=x
  60 GO SUB 200
  70 LET xf=facul
  80 LET m=n-x
  90 GO SUB 200
 100 LET nx=facul
 110 LET y=nf/(xf*nx)*p↑x*(1-p)↑
(n-x)
 120 PRINT AT 15,0;d$;AT 15,0;"F
(";x;") = ";y
 130 GO TO 40
 200 LET facul=m
 210 IF facul=1 OR facul=0 THEN
LET facul=1: GO TO 250
 220 LET m=m-1
 230 LET facul=facul*m
 240 IF m>1 THEN GO TO 220
 250 RETURN
```

```
BINOMIAL DISTRIBUTION

n= 5      p= 0.61
F(2)  = 0.220726
```

The program asks for n, p and x. Using the example above $n = 5$, $p = 1/6$ and $x = 2$. This provides the result that the chance that 6 will be thrown twice in 5 throws is slightly less than the chance of 6 being thrown in 1 throw.

N.B. The symbol ↑ is indicated on the ZX81/TS1000 as **.

LETTER INVADERS
(ZX81 and TS1000 only)

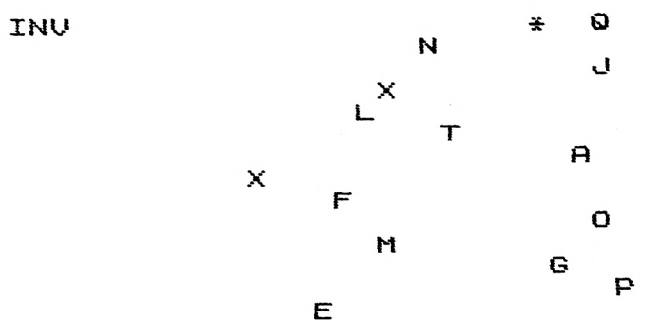
Personally I think that this game shows how powerful a 1 K ZX81 or TS1000 can be. Furthermore it illustrates the power of the **SCROLL** statement with respect to achieving animated graphics. Naturally you are a captain of a spaceship (it's amazing what stories can be imagined when we only see simple characters on a screen, but it's all in the game). Your spaceship is indicated by a circle. The space invaders are disguised by letters. Each time you touch an invader, the letter is displayed on the screen. In this way you try to spell

INVADERS

You will certainly find out how difficult it is. Use keys **5** and **8** to control you ship. Good luck!

Program

```
  10 LET S=0
  20 LET Z$=""
  30 LET K=15
  40 PRINT AT 13,9+RND*21;CHR$ (
38+RND*25)
  50 SCROLL
  60 IF INKEY$="5" THEN LET K=K-
1
  70 IF INKEY$="8" THEN LET K=K+
1
  80 PRINT AT 0,K;
  90 LET N1=PEEK 16398+256*PEEK
16399
 100 IF PEEK N1>37 AND PEEK N1<6
4 THEN GOTO 120
 110 GOTO 160
 120 LET Z$=Z$+CHR$ PEEK N1
 130 PRINT "BANG"
 140 IF Z$="INVADERS" THEN STOP
 150 GOTO 40
 160 PRINT "*"
 170 PRINT AT 0,0;Z$
 180 GOTO 40
```



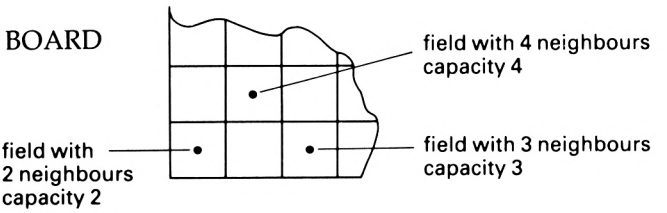
Some remarks

This program illustrates how powerful the **SCROLL** statement is with respect to the generation of moving objects. By means of line **40** letters are randomly generated. Lines **60–70** control the position of the spaceship. This spaceship is always plotted at line **0** (top) by means of the statement **PRINT AT 0,K**. The construction of line **90** is explained in the program 'BEWARE OF THE SNAKE'. If you are tired of this game you can use the **BREAK** key. Personally it took me hours to . . . Well, you can't win 'em all!

EXPLOSION
(ZX Spectrum only)

This program provides a very interesting board-game. A maximum of 7 players can play on a board of 20 × 20 fields maximum. Each field has a 'capacity', which is defined as the number of neighbours, as shown in the next figure.

PART OF THE BOARD



The program begins with the question

```
number of players
```

The number of players is entered (for example 2) and we now see

```
size of the board
```

As an example let us play with a very small board by entering 2. The computer now shows

```
   1   2
1
2
```

A coloured cursor indicates that the first player has to enter his field of choice, for example

```
1   2
```

The computer shows

```
   1   2
1      [1]◄—— colour of player 1
2
```

Now the second player enters a field, for example

```
2   1
```

and the computer shows

```
   1      2
1         [1]◄—— colour of player 1
2 [1]◄———— colour of player 2
```

Suppose the first player enters 1,2 again (you may only indicate an empty field or an own field). As a result field (1,2) reaches the value 2 which is in fact its capacity. Once the capacity is reached the field explodes; it becomes empty and the contents are spread to the neighbours. If a neighbour is an own field the value will increase by 1 and if it is a field of an enemy it is conquered and shows an initial value 1. You will notice that explosions become bigger and bigger—the game ends with one continuous explosion in a specific colour. This colour indicates the winner.

Program

```
  10 PRINT "Number of players: "
;
  20 INPUT P
  25 IF P>7 THEN GO TO 20
  30 PRINT P
  40 PRINT "Size of the board: "
;
  50 INPUT S
  55 IF S>20 THEN GO TO 50
  60 PRINT S
  70 DIM B(S,S)
  80 FOR I=1 TO S
  90 FOR J=1 TO S
 100 LET B(I,J)=0
 110 NEXT J
 120 NEXT I
 125 CLS
 130 FOR I=1 TO S
 140 LET J=I-10*INT (I/10)
 150 PRINT AT 0,I;J
 160 PRINT AT I,0;J
 170 NEXT I
 180 DIM C(7)
 185 DIM D(7)
 190 FOR I=1 TO 7
 200 READ C(I): READ D(I)
 210 NEXT I
 220 LET K=1
 300 INPUT PAPER C(K);" ";X,Y
 305 IF X>S OR Y>S THEN GO TO 30
0
 310 LET BXY=B(X,Y)
 315 IF BXY=0 THEN GO TO 335
 320 LET COL=INT (BXY/100)
 330 IF COL<>C(K) THEN GO TO 300
 332 GO TO 340
 335 LET COL=C(K)
 336 LET BXY=100*COL
 340 LET B(X,Y)=BXY+1
 350 GO SUB 2000
 355 LET VAL=BXY+1-100*COL
 360 IF VAL>=CAP THEN GO TO 900
 370 PRINT AT X,Y; PAPER COL; IN
K D(COL);VAL
 375 LET K=K+1
 380 IF K>P THEN LET K=1
 390 GO TO 300
 900 LET E=0
 910 GO SUB 2300
```

```
 920 IF E=0 THEN GO TO 375
1000 LET E=0
1010 FOR X=1 TO 5
1020 FOR Y=1 TO 5
1030 GO SUB 2300
1040 NEXT Y
1050 NEXT X
1060 IF E=1 THEN GO TO 1000
1080 GO TO 375
2000 LET CAP=4
2010 IF X=1 OR X=5 THEN LET CAP=
CAP-1
2020 IF Y=1 OR Y=5 THEN LET CAP=
CAP-1
2030 RETURN
2100 LET BIJ=B(I,J)
2120 LET CL=INT (BIJ/100)
2130 LET VL=BIJ-100*CL
2140 IF VL=0 THEN GO TO 2170
2150 PRINT AT I,J; PAPER CL; INK
 D(CL);VL
2160 GO TO 2180
2170 PRINT AT I,J;" "
2200 RETURN
2300 GO SUB 2000
2305 LET BXY=B(X,Y)
2307 LET COL=INT (BXY/100)
2308 IF (BXY-100*COL)<CAP THEN G
O TO 2390
2310 LET BXY=BXY-CAP
2315 IF BXY=100*COL THEN LET BXY
=0
2317 LET B(X,Y)=BXY
2318 LET I=X: LET J=Y
2319 GO SUB 2100
2320 LET J=Y-1
2330 IF J>0 THEN GO SUB 2500
2340 LET J=Y+1
2350 IF J<=5 THEN GO SUB 2500
2360 LET I=X-1: LET J=Y
2365 IF I>0 THEN GO SUB 2500
2370 LET I=X+1
2380 IF I<=5 THEN GO SUB 2500
2390 RETURN
2500 LET BIJ=B(I,J)
2510 LET BIJ=BIJ+100*(COL-INT (B
IJ/100))+1
2515 IF E=1 THEN GO TO 2560
2520 LET CAP=4
2530 IF I=1 OR I=5 THEN LET CAP=
CAP-1
2540 IF J=1 OR J=5 THEN LET CAP=
CAP-1
2550 IF (BIJ-100*COL)>=CAP THEN
LET E=1
2560 LET B(I,J)=BIJ
2570 GO SUB 2100
2580 RETURN
9000 DATA 1,7
9010 DATA 6,7
9020 DATA 2,7
9030 DATA 3,7
9040 DATA 4,0
9050 DATA 5,0
9060 DATA 7,0
```

ERROR FUNCTION

This program determines erf x according to

$$\text{erf } x = \frac{2}{\sqrt{2\pi}} \int_0^x e^{-u^2}\, du$$

This formula is approximated by

$$1 - (a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5) e^{-x^2}$$

where $t = \dfrac{1}{1 + px}$

$p = 0.3275911$
$a_1 = 0.254829592$
$a_2 = -0.284496736$
$a_3 = 1.421413741$
$a_4 = -1.4531152027$
$a_5 = 1.06140529$

Program

```
  10 LET p=.3275911
  20 LET a1=.254829592
  30 LET a2=-0.284496736
  40 LET a3=1.421413741
  50 LET a4=-1.4531152027
  60 LET a5=1.06140529
  65 LET d$="
                "
  70 PRINT AT 10,10; PAPER 1; IN
K 7;"ERROR FUNCION"
  80 INPUT "X= ";x
  90 LET t=1/(1+p*x)
 100 LET y=1-(a1*t+a2*t↑2+a3*t↑3
+a4*t↑4+a5*t↑5)*EXP (-x↑2)
 110 PRINT AT 13,0;d$;AT 13,0;"e
rf(";x;") = ";y
 120 GO TO 80
```

Example

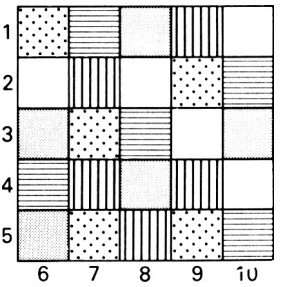
```
                ERROR FUNCION

erf(1.21)  = 0.91295334
```

The program can be terminated by entering the **STOP** function.

N.B. the symbol ↑ is indicated on the ZX81/TS1000 as **.

THE TERRIBLE COLOUR PROBLEM
(ZX Spectrum only)

This game shows you a 5×5 board apparently with all randomly coloured blocks.



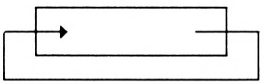
We have 5 horizontal rows (1–5) and 5 vertical rows (6–10). (Note that the columns are called 'rows' in this program. The computer makes a request:

```
Enter number of row
```

We enter one of the row numbers and the computer requests:

```
Enter number of steps
```

Now the blocks in the row indicated will rotate cyclically, for instance if we enter 2 all blocks will shift two steps and each block that 'falls' from the board is shifted to the other side:



Positive steps will move the blocks in the horizontal rows rightward and the blocks in the vertical rows upward. Negative steps are allowed and this reverses the direction of movement of the blocks. The aim of the game is simply to end with 5 horizontal bars – each bar must have only one colour.

Program

```
 10 DIM a(6,6)
 20 FOR x=1 TO 5
 30 FOR y=2 TO 6
 40 LET a(x,y)=y-1
 50 NEXT y
 60 NEXT x
 70 FOR i=1 TO 20
 80 LET k=1+INT (10*RND)
 90 GO SUB 400
120 NEXT i
125 CLS
```

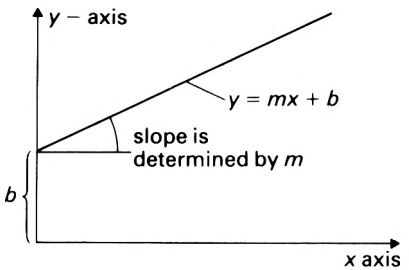
```
  130 REM start game
  140 FOR y=2 TO 6
  150 LET p=(y-2)*4-1
  160 PRINT AT (p+2),5;y-1
  170 FOR i=1 TO 4
  180 PRINT AT (p+i),7; INK a(1,y
);"■■■■■■"; INK a(2,y);"■■■■■■"; INK
 a(3,y);"■■■■■■"; INK a(4,y);"■■■
■"; INK a(5,y);"■■■■■■"
  190 NEXT i
  200 NEXT y
  210 PRINT AT 21,7;" 6    7    8
  9    10"
  220 INPUT AT 0,0;"Enter number
of row ";k;AT 1,0;"Enter number
of steps ";s
  240 IF s>=1 THEN GO TO 260
  250 LET s=s+5
  260 FOR i=1 TO s
  270 GO SUB 400
  280 NEXT i
  290 GO TO 130
  400 IF k>5 THEN GO TO 470
  410 IF k>10 THEN GO TO 220
  420 FOR x=6 TO 2 STEP -1
  430 LET a(x,k+1)=a(x-1,k+1)
  440 NEXT x
  450 LET a(1,k+1)=a(6,k+1)
  460 RETURN
  470 FOR y=2 TO 6
  480 LET a(k-5,y-1)=a(k-5,y)
  490 NEXT y
  500 LET a(k-5,6)=a(k-5,1)
  510 RETURN
```

HI RES CURVE FITTING
(ZX Spectrum only)

This program determines how we can draw a straight line through a collection of points in the best way possible. The points are given by their x and y coordinates. Before we actually enter the points by typing the x and y coordinates, we have to enter the maximum values of x and y and also the minimum values of these coordinates. This means that we must scan the data in order to find these values. The program needs these values in order to plot the points as well as the straight line on the screen. We know that each straight line is given by a fomula that relates the x coordinate and y coordinate of each point according to

$$y = mx + b$$

Hence each value of y of a point on the line is found by multiplying x by m and adding b to it. If we draw this line we will see that m determines the slope and b determines the intercept of the line on the y axis:



The program also determines the correlation coefficient r. This coefficient determines how well the line fits the points given. If all points lie exactly on the line, the absolute value of r is 1, otherwise it is less than 1.

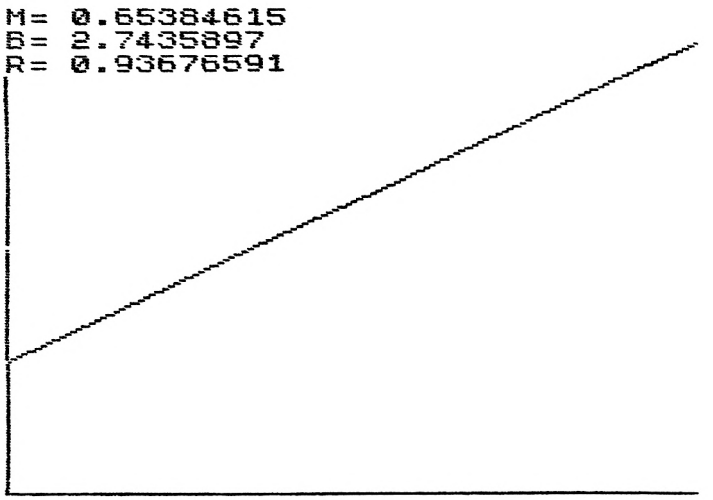
Program

```
   10 CLEAR
   20 PRINT AT 10,10; INK 7; PAPE
R 1;"CURVE FITTING"
   30 INPUT AT 0,0;"X MAXIMUM= ";
xmax;AT 1,0;"X MINIMUM= ";xmin;A
T 2,0;"Y MAXIMUM= ";ymax;AT 3,0;
"Y MINIMUM= ";ymin;AT 5,0;"X= ";
x;AT 6,0;"Y= ";y
   40 LET lenx=255
   50 LET leny=175
   60 LET y0=((ABS ymin)/(ymax+AB
S ymin))*leny
   70 LET x0=((ABS xmin)/(xmax+AB
S xmin))*lenx
```

```
  80 LET schaalx=lenx/(xmax+ABS
xmin)
  85 LET schaaly=leny/(ymax+ABS
ymin)
  90 CLS : INK 1
 100 PLOT 0,y0
 110 DRAW lenx,0
 120 PLOT x0,0
 130 DRAW 0,leny
 135 PLOT x0+x*schaalx,y0+y*scha
aly
 140 LET xs=x: LET ys=y
 150 LET x2=x*x: LET y2=y*y
 160 LET xy=x*y: LET n=1
 165 LET xbegin=0: LET ybegin=le
ny/2
 166 LET xeind=lenx: LET yeind=y
begin
 167 PLOT xbegin,ybegin
 168 DRAW OVER 1;xeind-xbegin,ye
ind-ybegin
 200 INPUT AT 0,0;"X= ";x;AT 1,0
;"Y= ";y
 205 LET n=n+1
 210 LET xs=xs+x
 220 LET ys=ys+y
 230 LET x2=x2+x*x
 240 LET y2=y2+y*y
 250 LET xy=xy+x*y
 260 LET m=(xs*ys/n-xy)/(xs*xs/n
-x2)
 270 LET b=ys/n-m*xs/n
 280 LET nm=SQR ((n*x2-xs*xs)*(n
*y2-ys*ys))
 290 LET t=n*xy-xs*ys
 300 PRINT AT 0,0; INK 2;"M= ";m
 310 PRINT AT 1,0; INK 2;"B= ";b
 320 PRINT AT 2,0; INK 2;"R= ";t
/nm
 330 PLOT OVER 1;xbegin,ybegin
 340 DRAW OVER 1;xeind-xbegin,ye
ind-ybegin
 345 PLOT x0+x*schaalx,y0+y*scha
aly
 350 LET xa=xmin
 360 GO SUB 500
 370 LET xbegin=xa: LET ybegin=y
a
 380 PLOT OVER 1;xbegin,ybegin
 390 LET xa=xmax
 400 GO SUB 500
 410 LET xeind=xa: LET yeind=ya
 420 DRAW OVER 1;xeind-xbegin,ye
ind-ybegin
 430 GO TO 200
 500 LET ya=m*xa+b
 510 IF ya>ymax THEN LET xa=(yma
x-b)/m: LET ya=ymax
 520 IF ya<ymin THEN LET xa=(ymi
n-b)/m: LET ya=ymin
 530 LET xa=x0+xa*schaalx
 540 LET ya=y0+ya*schaaly
 550 RETURN
```

Control example

The pairs (3,4), (5,7) and (10,9) should lead to



The program can be terminated by entering the **STOP** function.

APPENDIX 1

Survey of commands

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
BEEP x,y		generates a tone during x seconds with pitch y
BORDER m		indicates colour of border
BRIGHT n		indicates brightness: 1 = 'extra bright' and 0 = 'normal'
CAT		produces catalog (microdrives)
CIRCLE x,y,z		draws a circle with radius z and centre x,y
CLEAR	CLEAR	deletes all variables; executes CLS and RESTORE and resets PLOT position
CLEAR n		changes RAMTOP to n
CLOSE #		closes indicated file (microdrives)
CLS	CLS	clears the screen
CONTINUE	CONT	if last report was 9 computer continues with following statement; otherwise it continues with statement where the error occurred
COPY	COPY	copies upper 22 lines of screen on the printer
DATA		describes a list of values that can be assigned to variables by means of the READ statement, example: 10 READ a,b,c . . . 70 DATA 3,4,7

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
DEF FN		defines a function, example: 10 DEF FNa(b,c)=b+c . . . 70 LET X=FNa(6,4)
DELETE		deletes file (microdrive)
DIM	DIM	introduces an array, a list of variables; example: 10 DIM A(10) introduces variables A(1),A(2)...A(10) other examples: 10 DIM A(2,3,6) 20 DIM B$(4,10) only one letter may be used to indicate name of array
DRAW x,y		draws line from last point to x,y
DRAW x,y,z		draws line while turning through an angle z, moving x horizontally and y vertically
ERASE		erases action (microdrives)
	FAST	starts calculating in fast mode; during calculations display is dark
FLASH		indicates if an item has to be displayed in a flashing mode: 0 = not flashing 1 = flashing
FOR α=x TO y STEP z	FOR α=x TO y STEP z	construction to indicate that part of a program has to be executed repeatedly for α=x, α=x+z, α=x+2z etc. if α>y execution is terminated

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
NEXT α	NEXT α	if STEP z is ommitted STEP 1 is taken; example: 10 FOR K=1 TO 10 20 LET A(K)=K 30 NEXT K NEXT K defines the end of the specific part; name of counting variable is only 1 letter
FORMAT f		format floppy (microdrives)
GOSUB n RETURN	GOSUB n RETURN	proceeds with line number n and if RETURN is encountered goes back to statement that follows GOSUB n statement
GOTO n	GOTO n	proceeds with line n
IF x THEN s	IF x THEN s	if the condition x comes out as true the statement after THEN is executed: with the Spectrum more statements can be indicated on a single line
INK n		specifies the colour of the ink
INPUT	INPUT	statement to assign value during execution of program; example: 10 INPUT A 20 PRINT A with the ZX Spectrum prompting is allowed; examples: 10 INPUT "colour=";n 20 LET c$="what is the colour?" 30 INPUT (c$);n if we want to stop a program during an INPUT statement we eventually erase the prompting character and enter the STOP function

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
INVERSE		prints item with colour ink = colour previous paper and colour paper = previous ink: `INVERSE 0` gives normal printing `INVERSE 1` gives inverse printing

LET	LET	statement to assign a value to a variable; examples:
		`10 LET A=10`
		`20 LET B$="COMPUTER"`
		`30 LET A=A+1`

LIST n	LIST n	lists program starting at line n; list without any number generates a listing that starts with the first line of the program
LLIST n	LLIST n	like LIST but applies to printer
LOAD f	LOAD f	loads file f; examples: `LOAD""` `LOAD "INVADERS"`
LOAD f DATA()		loads a numeric array: `LOAD "test" DATA A()`
LOAD f DATA		in case of a string array: `LOAD "testz" DATA b$()`
LOAD f CODE m,n		loads m bytes starting at address m; example: `LOAD "test" CODE 32500,10` it is also allowed to indicate only the starting address
LOAD f SCREEN$		loads the screen: `LOAD "picture" SCREEN$`
LPRINT	LPRINT	see PRINT; instead of the screen the printer is now used

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
MERGE		has the same effect as LOAD; however LOAD deletes the old program and variables in the computer, MERGE does not
NEW	NEW	erases old program
NEXT	NEXT	part of FOR...NEXT, see FOR
OUT m,n		outputs byte n at port m: loads bc register with m, and a register with n and executes assembler instruction out (c),a, see page 169
OVER n		if n = 0 obliterates previous characters on screen by printing statement; if n = 1 new characters are mixed in with old characters; see page 95
PAPER n		determines colour of background
PAUSE n	PAUSE n	introduces a pause: 50 ≈ 1 second
PLOT c,m,n	PLOT m,n	plots pixel at m,n; c is optional and indicates colour
POKE m,n	POKE m,n	writes value n in memory location m
PRINT	PRINT	statement that controls output; after PRINT we can indicate a list of items: variables constants strings (between quotes) expressions (numerical) separators can be used: ; following item is printed directly after previous item , two columns are used by means of the functions AT and TAB we can control the position of printing
RANDOMIZE	RAND	controls random generator: different random numbers will be generated every time

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
RANDOMIZE n	**RAND** n	if n = 0 this statement corresponds to previous statement: if n = integer the *same* random numbers will be generated
READ		see **DATA**
REM	**REM**	statement to insert remarks
RESTORE		resets **DATA** pointer, see page 43
RESTORE n		see page 44
RETURN	**RETURN**	see **GOSUB**
RUN	**RUN**	starts execution of program: eventually a line number can be indicated
SAVE f	**SAVE** f	saves program f on cassette
SAVE f **LINE** n		see **SAVE**: now the program will automatically start at line n if program is loaded
SAVE f **DATA ()**		saves numeric array
SAVE f **DATA $()**		saves character array
SAVE f **CODE** m,n		saves memory image, n bytes starting at address m
SAVE f **SCREEN$**		saves screen
	SCROLL	scrolls screen one line up
	SLOW	puts computer into compute and display mode, speed approximately $\frac{1}{4}$ of **FAST** mode
STOP	**STOP**	stops execution of program; **CONTINUE** will resume with the following statement
	UNPLOT m,n	blanks pixel m,n
VERIFY f		verifies if loaded program " f " is correctly loaded; use " **play** " mode of cassette

APPENDIX 2

Survey of functions

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
ABS	ABS	computes absolute magnitude
ACS	ACS	computes arccosine (radians)
AND	AND	performs logical and numerical operation, see page 57
ASN	ASN	computes arcsine (radians)
ATN	ATN	computes arctangent (radians)
ATTR		determines the attributes of a pixel, see page 96
BIN		converts series of zeros and ones as a binary number, see page 46
CHR$	CHR$	converts code to character
CODE	CODE	converts first character of string to code
COS	COS	computes cosine (radians)
EXP	EXP	computes e^x
FN		indicates a predefined function; FN is followed by a letter arguments are enclosed in brackets, see page 90
IN		IN X loads bc register with x and does the assembly instruction in a(c), see page 169
INKEY$	INKEY$	scans keyboard; if a key is pressed it is assigned to INKEY$, see page 61
INT	INT	rounds number down to the integer part

ZX Spectrum	*ZX81 and TS1000*	*Meaning*
LEN	LEN	determines number of characters of a string
LN	LN	computes natural logarithm
NOT	NOT	logical and numerical operator, see page 57
OR	OR	logical and numerical operator, see page 57
PEEK	PEEK	determines integer value of indicated memory address, see page 148
PI	PI	3.14159265
POINT		POINT (x,y) indicates if pixel is ink colour (1) or paper colour (Ø)
RND	RND	generates random value, see page 47
SCREEN$		SCREEN$ (x,y) determines character on the screen at line x and column y
SGN	SGN	computes sign of number: −1 if negative, 0 if zero, 1 if positive
SIN	SIN	computes sine (radians)
SQR	SQR	computes square root
STR$	STR$	converts number to an equivalent string, see page 65
TAN	TAN	computes tangent (radians)
USR	USR	calls machine code at address indicated, see page 149
VAL	VAL	converts string to equivalent number, see page 66
VAL$		converts string to new string (without boundary quotes)

APPENDIX 3

Character set

ZX81/TS1000		*ZX Spectrum*	
Code	*Character*	*Code*	*Character*
0	space	0	not used
1		1	
2		2	
3		3	
4		4	
5		5	
6		6	**PRINT** comma
7		7	**EDIT**
8		8	cursor left
9		9	cursor right
10		10	cursor down
11	"	11	cursor up
12	£	12	**DELETE**
13	$	13	**ENTER**
14	:	14	number
15	?	15	not used
16	(	16	**INK** control
17	)	17	**PAPER** control
18	>	18	**FLASH** control
19	<	19	**BRIGHT** control
20	=	20	**INVERSE** control
21	+	21	**OVER** control
22	−	22	**AT** control
23	*	23	**TAB** control
24	/	24	not used
25	;	25	
26	,	26	
27	.	27	
28	0	28	
29	1	29	
30	2	30	
31	3	31	
32	4	32	space
33	5	33	!
34	6	34	"

ZX81/TS1000		ZX Spectrum	
Code	Character	Code	Character
35	7	35	#
36	8	36	$
37	9	37	%
38	A	38	&
39	B	39	'
40	C	40	(
41	D	41	)
42	E	42	*
43	F	43	+
44	G	44	,
45	H	45	−
46	I	46	.
47	J	47	/
48	K	48	0
49	L	49	1
50	M	50	2
51	N	51	3
52	O	52	4
53	P	53	5
54	Q	54	6
55	R	55	7
56	S	56	8
57	T	57	9
58	U	58	:
59	V	59	;
60	W	60	<
61	X	61	=
62	Y	62	>
63	Z	63	?
64	**RND**	64	@
65	**INKEY$**	65	A
66	**PI**	66	B
67		67	C
68		68	D
69		69	E
70		70	F
71		71	G
72	not used	72	H
73		73	I
74		74	J
75		75	K
76		76	L
77		77	M
78		78	N
79		79	O
80		80	P
81		81	Q

ZX81/TS1000		*ZX Spectrum*	
Code	*Character*	*Code*	*Character*
82		82	R
83		83	S
84		84	T
85		85	U
86		86	V
87		87	W
88		88	X
89		89	Y
90		90	Z
91		91	[
92		92	/
93		93	]
94	not used	94	↑
95		95	_
96		96	£
97		97	a
98		98	b
99		99	c
100		100	d
101		101	e
102		102	f
103		103	g
104		104	h
105		105	i
106		106	j
107		107	k
108		108	l
109		109	m
110		110	n
111		111	o
112	cursor up ◇	112	p
113	cursor down ◇	113	q
114	cursor left ◇	114	r
115	cursor right ◇	115	s
116	**GRAPHICS**	116	t
117	**EDIT**	117	u
118	**NEWLINE**	118	v
119	**RUBOUT**	119	w
120	K / L mode	120	x
121	**FUNCTION**	121	y
122	not used	122	z
123	not used	123	{
124	not used	124	|
125	not used	125	}
126	number	126	~
127	cursor	127	©
128	■	128	□

ZX81/TS1000		ZX Spectrum		
Code	Character	Code	Character	
129		129		
130		130		
131		131		
132		132		
133		133		
134		134		
135		135		
136		136		
137		137		
138		138		
139	inverse "	139		
140	inverse £	140		
141	inverse $	141		
142	inverse :	142		
143	inverse ?	143		
144	inverse (	144	(a)	user graphics
145	inverse)	145	(b)	
146	inverse >	146	(c)	
147	inverse <	147	(d)	
148	inverse =	148	(e)	
149	inverse +	149	(f)	
150	inverse –	150	(g)	
151	inverse *	151	(h)	
152	inverse /	152	(i)	
153	inverse ;	153	(j)	
154	inverse ,	154	(k)	
155	inverse .	155	(l)	
156	inverse 0	156	(m)	
157	inverse 1	157	(n)	
158	inverse 2	158	(o)	
159	inverse 3	159	(p)	
160	inverse 4	160	(q)	
161	inverse 5	161	(r)	
162	inverse 6	162	(s)	
163	inverse 7	163	(t)	
164	inverse 8	164	(u)	
165	inverse 9	165	**RND**	
166	inverse A	166	**INKEY$**	
167	inverse B	167	**PI**	
168	inverse C	168	**FN**	
169	inverse D	169	**POINT**	
170	inverse E	170	**SCREEN$**	
171	inverse F	171	**ATTR**	
172	inverse G	172	**AT**	
173	inverse H	173	**TAB**	
174	inverse I	174	**VAL$**	
175	inverse J	175	**CODE**	

ZX81/TS1000		*ZX Spectrum*	
Code	*Character*	*Code*	*Character*
176	inverse K	176	**VAL**
177	inverse L	177	**LEN**
178	inverse M	178	**SIN**
179	inverse N	179	**COS**
180	inverse O	180	**TAN**
181	inverse P	181	**ASN**
182	inverse Q	182	**ACS**
183	inverse R	183	**ATN**
184	inverse S	184	**LN**
185	inverse T	185	**EXP**
186	inverse U	186	**INT**
187	inverse V	187	**SQR**
188	inverse W	188	**SGN**
189	inverse X	189	**ABS**
190	inverse Y	190	**PEEK**
191	inverse Z	191	**IN**
192	""	192	**USR**
193	**AT**	193	**STR$**
194	**TAB**	194	**CHR$**
195	not used	195	**NOT**
196	**CODE**	196	**BIN**
197	**VAL**	197	**OR**
198	**LEN**	198	**AND**
199	**SIN**	199	<=
200	**COS**	200	>=
201	**TAN**	201	<>
202	**ASN**	202	**LINE**
203	**ACS**	203	**THEN**
204	**ATN**	204	**TO**
205	**LN**	205	**STEP**
206	**EXP**	206	**DEF FN**
207	**INT**	207	**CAT**
208	**SQR**	208	**FORMAT**
209	**SGN**	209	**MOVE**
210	**ABS**	210	**ERASE**
211	**PEEK**	211	**OPEN #**
212	**USR**	212	**CLOSE #**
213	**STR$**	213	**MERGE**
214	**CHR$**	214	**VERIFY**
215	**NOT**	215	**BEEP**
216	**	216	**CIRCLE**
217	**OR**	217	**INK**
218	**AND**	218	**PAPER**
219	<=	219	**FLASH**
220	>=	220	**BRIGHT**
221	<>	221	**INVERSE**

ZX81/TS1000		*ZX Spectrum*	
Code	*Character*	*Code*	*Character*
222	**THEN**	222	**OVER**
223	**TO**	223	**OUT**
224	**STEP**	224	**LPRINT**
225	**LPRINT**	225	**LLIST**
226	**LLIST**	226	**STOP**
227	**STOP**	227	**READ**
228	**SLOW**	228	**DATA**
229	**FAST**	229	**RESTORE**
230	**NEW**	230	**NEW**
231	**SCROLL**	231	**BORDER**
232	**CONT**	232	**CONTINUE**
233	**DIM**	233	**DIM**
234	**REM**	234	**REM**
235	**FOR**	235	**FOR**
236	**GOTO**	236	**GO TO**
237	**GOSUB**	237	**GO SUB**
238	**INPUT**	238	**INPUT**
239	**LOAD**	239	**LOAD**
240	**LIST**	240	**LIST**
241	**LET**	241	**LET**
242	**PAUSE**	242	**PAUSE**
243	**NEXT**	243	**NEXT**
244	**POKE**	244	**POKE**
245	**PRINT**	245	**PRINT**
246	**PLOT**	246	**PLOT**
247	**RUN**	247	**RUN**
248	**SAVE**	248	**SAVE**
249	**RAND**	249	**RANDOMIZE**
250	**IF**	250	**IF**
251	**CLS**	251	**CLS**
252	**UNPLOT**	252	**DRAW**
253	**CLEAR**	253	**CLEAR**
254	**RETURN**	254	**RETURN**
255	**COPY**	255	**COPY**

APPENDIX 4

Z80 CPU Instruction set

Reproduced by permission of Zilog.

OBJ CODE	SOURCE STATEMENT		OPERATION
8E	ADC	A,(HL)	Add with Carry Oper-
DD8E05	ADC	A,(IX+d)	and to Acc.
FD8E05	ADC	A,(IY+d)	
8F	ADC	A,A	
88	ADC	A,B	
89	ADC	A,C	
8A	ADC	A,D	
8B	ADC	A,E	
8C	ADC	A,H	
8D	ADC	A,L	
CE20	ADC	A,n	
ED4A	ADC	HL,BC	Add with Carry Reg.
ED5A	ADC	HL,DE	Pair to HL
ED6A	ADC	HL,HL	
ED7A	ADC	HL,SP	
86	ADD	A,(HL)	Add Operand to Acc.
DD8605	ADD	A,(IX+d)	
FD8605	ADD	A,(IY+d)	
87	ADD	A,A	
80	ADD	A,B	
81	ADD	A,C	
82	ADD	A,D	
83	ADD	A,E	
84	ADD	A,H	
85	ADD	A,L	
C620	ADD	A,n	
09	ADD	HL,BC	Add Reg. Pair to HL
19	ADD	HL,DE	
29	ADD	HL,HL	
39	ADD	HL,SP	
DD09	ADD	IX,BC	Add Reg. Pair to IX
DD19	ADD	IX,DE	
DD29	ADD	IX,IX	
DD39	ADD	IX,SP	
FD09	ADD	IY,BC	Add Reg. Pair to IY
FD19	ADD	IY,DE	
FD29	ADD	IY,IY	
FD39	ADD	IY,SP	
A6	AND	(HL)	Logical 'AND' of
DDA605	AND	(IX+d)	Operand and Acc.
FDA605	AND	(IY+d)	
A7	AND	A	
A0	AND	B	
A1	AND	C	
A2	AND	D	
A3	AND	E	
A4	AND	H	
A5	AND	L	
E620	AND	n	
CB46	BIT	0,(HL)	Test Bit b of Location
DDCB0546	BIT	0,(IX+d)	or Reg.
FDCB0546	BIT	0,(IY+d)	
CB47	BIT	0,A	
CB40	BIT	0,B	
CB41	BIT	0,C	
CB42	BIT	0,D	
CB43	BIT	0,E	
CB44	BIT	0,H	

OBJ CODE	SOURCE STATEMENT		OPERATION
CB45	BIT	0,L	Test Bit b of Location
CB4E	BIT	1 (HL)	or Reg.
DDCB054E	BIT	1,(IX+d)	
FDCB054E	BIT	1,(IY+d)	
CB4F	BIT	1,A	
CB48	BIT	1,B	
CB49	BIT	1,C	
CB4A	BIT	1,D	
CB4B	BIT	1,E	
CB4C	BIT	1,H	
CB4D	BIT	1,L	
CB56	BIT	2,(HL)	
DDCB0556	BIT	2,(IX+d)	
FDCB0556	BIT	2,(IY+d)	
CB57	BIT	2,A	
CB50	BIT	2,B	
CB51	BIT	2,C	
CB52	BIT	2,D	
CB53	BIT	2,E	
CB54	BIT	2,H	
CB55	BIT	2,L	
CB5E	BIT	3,(HL)	
DDCB055E	BIT	3,(IX+d)	
FDCB055E	BIT	3,(IY+d)	
CB5F	BIT	3,A	
CB58	BIT	3,B	
CB59	BIT	3,C	
CB5A	BIT	3,D	
CB5B	BIT	3,E	
CB5C	BIT	3,H	
CB5D	BIT	3,L	
CB66	BIT	4,(HL)	
DDCB0566	BIT	4,(IX+d)	
FDCB0566	BIT	4,(IY+d)	
CB67	BIT	4,A	
CB60	BIT	4,B	
CB61	BIT	4,C	
CB62	BIT	4,D	
CB63	BIT	4,E	
CB64	BIT	4,H	
CB65	BIT	4,L	
CB6E	BIT	5,(HL)	
DDCB056E	BIT	5,(IX+d)	
FDCB056E	BIT	5,(IY+d)	
CB6F	BIT	5,A	
CB68	BIT	5,B	
CB69	BIT	5,C	
C86A	BIT	5,D	
CB6B	BIT	5,E	
CB6C	BIT	5,H	
CB6D	BIT	5,L	
CB76	BIT	6,(HL)	
DDCB0576	BIT	6,(IX+d)	
FDCB0576	BIT	6,(IY+d)	
CB77	BIT	6,A	
CB70	BIT	6,B	
CB71	BIT	6,C	
CB72	BIT	6,D	
CB73	BIT	6,E	
CB74	BIT	6,H	
CB75	BIT	6,L	
CB7E	BIT	7,(HL)	
DDCB057E	BIT	7,(IX+d)	

OBJ CODE	SOURCE STATEMENT		OPERATION
FDCB057E	BIT	7,(IY+d)	Test Bit b Location or Reg.
CB7F	BIT	7,A	
CB78	BIT	7,B	
CB79	BIT	7,C	
CB7A	BIT	7,D	
CB7B	BIT	7,E	
CB7C	BIT	7,H	
CB7D	BIT	7,L	
DC8405	CALL	C,nn	Call Subroutine at Location nn if Condition True
FC8405	CALL	M,nn	
D48405	CALL	NC,nn	
C48405	CALL	NZ,nn	
F48405	CALL	P,nn	
EC8405	CALL	PE,nn	
E48405	CALL	PO,nn	
CC8405	CALL	Z,nn	
CD8405	CALL	nn	Unconditional Call to Subroutine at nn
3F	CCF		Complement Carry Flag
BE	CP	(HL)	Compare Operand with Acc.
DDBE05	CP	(IX+d)	
FDBE05	CP	(IY+d)	
BF	CP	A	
B8	CP	B	
B9	CP	C	
BA	CP	D	
BB	CP	E	
BC	CP	H	
BD	CP	L	
FE20	CP	n	
EDA9	CPD		Compare Location (HL) and Acc. Decrement HL and BC
EDB9	CPDR		Compare Location (HL) and Acc. Decrement HL and BC, Repeat until BC = 0
EDA1	CPI		Compare Location (HL) and Acc., Increment HL and Decrement BC
EDB1	CPIR		Compare Location (HL) and Acc. Increment HL, Decrement BC, Repeat until BC = 0
2F	CPL		Complement Acc. (1's Comp)
27	DAA		Decimal Adjust Acc.
35	DEC	(HL)	Decrement Operand
DD3505	DEC	(IX+d)	
FD3505	DEC	(IY+d)	
3D	DEC	A	
05	DEC	B	
0B	DEC	BC	
0D	DEC	C	
15	DEC	D	
1B	DEC	DE	

OBJ CODE	SOURCE STATEMENT		OPERATION
1D	DEC	E	Decrement Operand
25	DEC	H	
2B	DEC	HL	
DD2B	DEC	IX	
FD2B	DEC	IY	
2D	DEC	L	
3B	DEC	SP	
F3	DI		Disable Interrupts
102E	DJNZ	e	Decrement B and Jump Relative if B = 0
FB	EI		Enable Interrupts
E3	EX	(SP),HL	Exchange Location and (SP)
DDE3	EX	(SP),IX	
FDE3	EX	(SP),IY	
08	EX	AF,AF'	Exchange the Contents of AF and AF'
EB	EX	DE,HL	Exchange the Contents of DE and HL
D9	EXX		Exchange the Contents of BC,DE,HL with Contents of BC',DE',HL' Respectively
76	HALT		HALT (Wait for Interrupt or Reset)
ED46	IM	0	Set Interrupt Mode
ED56	IM	1	
ED5E	IM	2	
ED78	IN	A,(C)	Load Reg. with Input from Device (C)
ED40	IN	B,(C)	
ED48	IN	C,(C)	
ED50	IN	D,(C)	
ED58	IN	E,(C)	
ED60	IN	H,(C)	
ED68	IN	L,(C)	
34	INC	(HL)	Increment Operand
DD3405	INC	(IX+d)	
FD3405	INC	(IY+d)	
3C	INC	A	
04	INC	B	
03	INC	BC	
0C	INC	C	
14	INC	D	
13	INC	DE	
1C	INC	E	
24	INC	H	
23	INC	HL	
DD23	INC	IX	
FD23	INC	IY	
2C	INC	L	
33	INC	SP	
DB20	IN	A,(n)	Load Acc. with Input from Device n
EDAA	IND		Load Location (HL) with Input from Port (C), Decrement HL and B

OBJ CODE	SOURCE STATEMENT		OPERATION
EDBA	INDR		Load Location (HL) with Input from Port (C), Decrement HL and Decrement B, Repeat until B = 0
EDA2	INI		Load Location (HL) with Input from Port (C); Increment HL and Decrement B
EDB2	INIR		Load Location (HL) with Input from Port (C), Increment HL and Decrement B, Repeat until B = 0
C38405	JP	nn	Unconditional Jump to Location
E9	JP	(HL)	
DDE9	JP	(IX)	
FDE9	JP	(IY)	
DA8405	JP	C,nn	Jump to Location if Condition True
FA8405	JP	M,nn	
D28405	JP	NC,nn	
C28405	JP	NZ,nn	
F28405	JP	P,nn	
EA8405	JP	PE,nn	
E28405	JP	PO,nn	
CA8405	JP	Z,nn	
382E	JR	C,e	Jump Relative to PC+e if Condition True
302E	JR	NC,e	
202E	JR	NZ,e	
282E	JR	Z,e	
182E	JR	e	Unconditional Jump Relative to PC+e
02	LD	(BC),A	Load Source to Destination
12	LD	(DE),A	
77	LD	(HL),A	
70	LD	(HL),B	
71	LD	(HL),C	
72	LD	(HL),D	
73	LD	(HL),E	
74	LD	(HL),H	
75	LD	(HL),L	
3620	LD	(HL),n	
DD7705	LD	(IX+d),A	
DD7005	LD	(IX+d),B	
DD7105	LD	(IX+d),C	
DD7205	LD	(IX+d),D	
DD7305	LD	(IX+d),E	
DD7405	LD	(IX+d),H	
DD7505	LD	(IX+d),L	
DD360520	LD	(IX+d),n	
FD7705	LD	(IY+d),A	
FD7005	LD	(IY+d),B	
FD7105	LD	(IY+d),C	
FD7205	LD	(IY+d),D	
FD7305	LD	(IY+d),E	
FD7405	LD	(IY+d),H	
FD7505	LD	(IY+d),L	
FD360520	LD	(IY+d),n	
328405	LD	(nn),A	
ED438405	LD	(nn),BC	

OBJ CODE	SOURCE STATEMENT		OPERATION
ED538405	LD	(nn),DE	Load Source to Des-
228405	LD	(nn),HL	tination
DD228405	LD	(nn),IX	
FD228405	LD	(nn),IY	
ED738405	LD	(nn),SP	
0A	LD	A,(BC)	
1A	LD	A,(DE)	
7E	LD	A,(HL)	
DD7E05	LD	A,(IX+d)	
FD7E05	LD	A,(IY+d)	
3A8405	LD	A,(nn)	
7F	LD	A,A	
78	LD	A,B	
79	LD	A,C	
7A	LD	A,D	
7B	LD	A,E	
7C	LD	A,H	
ED57	LD	A,I	
7D	LD	A,L	
3E20	LD	A,n	
ED5F	LD	A,R	
46	LD	B,(HL)	
DD4605	LD	B,(IX+d)	
FD4605	LD	B,(IY+d)	
47	LD	B,A	
40	LD	B,B	
41	LD	B,C	
42	LD	B,D	
43	LD	B,E	
44	LD	B,H	
45	LD	B,L	
0620	LD	B,n	
ED4B8405	LD	BC,(nn)	
018405	LD	BC,nn	
4E	LD	C,(HL)	
DD4E05	LD	C,(IX+d)	
FD4E05	LD	C,(IY+d)	
4F	LD	C,A	
48	LD	C,B	
49	LD	C,C	
4A	LD	C,D	
4B	LD	C,E	
4C	LD	C,H	
4D	LD	C,L	
0E20	LD	C,n	
56	LD	D,(HL)	
DD5605	LD	D,(IX+d)	
FD5605	LD	D,(IY+d)	
57	LD	D,A	
50	LD	D,B	
51	LD	D,C	
52	LD	D,D	
53	LD	D,E	
54	LD	D,H	
55	LD	D,L	
1620	LD	D,n	
ED5B8405	LD	DE,(nn)	
118405	LD	DE,nn	
5E	LD	E,(HL)	
DD5E05	LD	E,(IX+d)	
FD5E05	LD	E,(IY+d)	
5F	LD	E,A	
58	LD	E,B	
59	LD	E,C	

OBJ CODE	SOURCE STATEMENT		OPERATION
5A	LD	E,D	Load Source to Destination
5B	LD	E,E	
5C	LD	E,H	
5D	LD	E,L	
1E20	LD	E,n	
66	LD	H,(HL)	
DD6605	LD	H,(IX+d)	
FD6605	LD	H,(IY+d)	
67	LD	H,A	
60	LD	H,B	
61	LD	H,C	
62	LD	H,D	
63	LD	H,E	
64	LD	H,H	
65	LD	H,L	
2620	LD	H,n	
2A8405	LD	HL,(nn)	
218405	LD	HL,nn	
ED47	LD	I,A	
DD2A8405	LD	IX,(nn)	
DD218405	LD	IX,nn	
FD2A8405	LD	IY,(nn)	
FD218405	LD	IY,nn	
6E	LD	L,(HL)	
DD6E05	LD	L,(IX+d)	
FD6E05	LD	L,(IY+d)	
6F	LD	L,A	
68	LD	L,B	
69	LD	L,C	
6A	LD	L,D	
6B	LD	L,E	
6C	LD	L,H	
6D	LD	L,L	
2E20	LD	L,n	
ED4F	LD	R,A	
ED7B8405	LD	SP,(nn)	
F9	LD	SP,HL	
DDF9	LD	SP,IX	
FDF9	LD	SP,IY	
318405	LD	SP,nn	
EDA8	LDD		Load Location (DE) with Location (HL), Decrement DE,HL and BC
EDB8	LDDR		Load Location (DE) with Location (HL), Repeat until BC = 0
EDA0	LDI		Load Location (DE) with Location (HL), Increment DE,HL, Decrement BC
EDB0	LDIR		Load Location (DE) with Location (HL), Increment DE,HL, Decrement BC and Repeat until BC = 0
ED44	NEG		Negate Acc. (2's Complement)
00	NOP		No Operation
B6	OR	(HL)	Logical "OR" of Operand and Acc.
DDB605	OR	(IX+d)	

OBJ CODE	SOURCE STATEMENT		OPERATION
FDB605	OR	(IY+d)	Logical "OR" of Operand and Acc.
B7	OR	A	
B0	OR	B	
B1	OR	C	
B2	OR	D	
B3	OR	E	
B4	OR	H	
B5	OR	L	
F620	OR	n	
ED8B	OTDR		Load Output Port (C) with Location (HL) Decrement HL and B, Repeat until B = 0
EDB3	OTIR		Load Output Port (C) with Location (HL), Increment HL, Decrement B, Repeat until B = 0
ED79	OUT	(C),A	Load Output Port (C) with Reg.
ED41	OUT	(C),B	
ED49	OUT	(C),C	
ED51	OUT	(C),D	
ED59	OUT	(C),E	
ED61	OUT	(C),H	
ED69	OUT	(C),L	
D320	OUT	(n),A	Load Output Port (n) with Acc.
EDAB	OUTD		Load Output Port (C) with Location (HL), Decrement HL and B
EDA3	OUTI		Load Output Port (C) with Location (HL), Increment HL and Decrement B
F1	POP	AF	Load Destination with Top of Stack
C1	POP	BC	
D1	POP	DE	
E1	POP	HL	
DDE1	POP	IX	
FDE1	POP	IY	
F5	PUSH	AF	Load Source to Stack
C5	PUSH	BC	
D5	PUSH	DE	
E5	PUSH	HL	
DDE5	PUSH	IX	
FDE5	PUSH	IY	
CB86	RES	0,(HL)	Reset Bit b of Operand
DDCB0586	RES	0,(IX+d)	
FDCB0586	RES	0,(IY+d)	
CB87	RES	0,A	
CB80	RES	0,B	
CB81	RES	0,C	
CB82	RES	0,D	
CB83	RES	0,E	
CB84	RES	0,H	
CB85	RES	0,L	
CB8E	RES	1,(HL)	
DDCB058E	RES	1,(IX+d)	
FDCB058E	RES	1,(IY+d)	
CB8F	RES	1,A	

OBJ CODE	SOURCE STATEMENT		OPERATION
CB88	RES	1,B	Reset Bit b of
CB89	RES	1,C	Operand
CB8A	RES	1,D	
CB8B	RES	1,E	
CB8C	RES	1,H	
CB8D	RES	1,L	
CB96	RES	2,(HL)	
DDCB0596	RES	2,(IX+d)	
FDCB0596	RES	2,(IY+d)	
CB97	RES	2,A	
CB90	RES	2,B	
CB91	RES	2,C	
CB92	RES	2,D	
CB93	RES	2,E	
CB94	RES	2,H	
CB95	RES	2,L	
CB9E	RES	3,(HL)	
DDCB059E	RES	3,(IX+d)	
FDCB059E	RES	3,(IY+d)	
CB9F	RES	3,A	
CB98	RES	3,B	
CB99	RES	3,C	
CB9A	RES	3,D	
CB9B	RES	3,E	
CB9C	RES	3,H	
CB9D	RES	3,L	
CBA6	RES	4,(HL)	
DDCB05A6	RES	4,(IX+d)	
FDCB05A6	RES	4,(IY+d)	
CBA7	RES	4,A	
CBA0	RES	4,B	
CBA1	RES	4,C	
CBA2	RES	4,D	
DBA3	RES	4,E	
CBA4	RES	4,H	
CBA5	RES	4,L	
CBAE	RES	5,(HL)	
DDCB05AE	RES	5,(IX+d)	
FDCB05AE	RES	5,(IY+d)	
CBAF	RES	5,A	
CBA8	RES	5,B	
CBA9	RES	5,C	
CBAA	RES	5,D	
CBAB	RES	5,E	
CBAC	RES	5,H	
CBAD	RES	5,L	
CBB6	RES	6,(HL)	
DDCB05B6	RES	6,(IX+d)	
FDCB05B6	RES	6,(IY+d)	
CBB7	RES	6,A	
CBB0	RES	6,B	
CBB1	RES	6,C	
CBB2	RES	6,D	
CBB3	RES	6,E	
CBB4	RES	6,H	
CBB5	RES	6,L	
CBBE	RES	7,(HL)	
DDCB05BE	RES	7,(IX+d)	
FDCB05BE	RES	7,(IY+d)	
CBBF	RES	7,A	
CBB8	RES	7,B	
CBB9	RES	7,C	
CBBA	RES	7,D	

OBJ CODE	SOURCE STATEMENT		OPERATION
CBBB	RES	7,E	Reset Bit b of Operand
CBBC	RES	7,H	
CBBD	RES	7,L	
C9	RET		Return from Subroutine
D8	RET	C	Return from Subroutine if Condition True
F8	RET	M	
D0	RET	NC	
C0	RET	NZ	
F0	RET	P	
E8	RET	PE	
E0	RET	P0	
C8	RET	Z	
ED4D	RETI		Return from Interrupt
ED45	RETN		Return from Non-Maskable Interrupt
CB16	RL	(HL)	Rotate Left Through Carry
DDCB0516	RL	(IX+d)	
FDCB0516	RL	(IY+d)	
CB17	RL	A	
CB10	RL	B	
CB11	RL	C	
CB12	RL	D	
CB13	RL	E	
CB14	RL	H	
CB15	RL	L	
17	RLA		Rotate Left Acc. Through Carry
CB06	RLC	(HL)	Rotate Left Circular
DDCB0506	RLC	(IX+d)	
FDCB0506	RLC	(IY+d)	
CB07	RLC	A	
CB00	RLC	B	
CB01	RLC	C	
CB02	RLC	D	
CB03	RLC	E	
CB04	RLC	H	
CB05	RLC	L	
07	RLCA		Rotate Left Circular Acc.
ED6F	RLD		Rotate Digit Left and Right between Acc. and and Location (HL)
CB1E	RR	(HL)	Rotate Right Through Carry
DDCB051E	RR	(IX+d)	
FDCB051E	RR	(IY+d)	
CB1F	RR	A	
CB18	RR	B	
CB19	RR	C	
CB1A	RR	D	
CB1B	RR	E	
CB1C	RR	H	
CB1D	RR	L	
1F	RRA		Rotate Right Acc. Through Carry
CB0E	RRC	(HL)	Rotate Right Circular
DDCB050E	RRC	(IX+d)	
FDCB050E	RRC	(IY+d)	
CB0F	RRC	A	

OBJ CODE	SOURCE STATEMENT		OPERATION
CB08	RRC	B	Rotate Right Circular
CB09	RRC	C	
CB0A	RRC	D	
CB0B	RRC	E	
CB0C	RRC	H	
CB0D	RRC	L	
0F	RRCA		Rotate Right Circular Acc.
ED67	RRD		Rotate Digit Right and Left Between Acc. and Location (HL)
C7	RST	00H	Restart to Location
CF	RST	08H	
D7	RST	10H	
DF	RST	18H	
E7	RST	20H	
EF	RST	28H	
F7	RST	30H	
FF	RST	38H	
DE20	SBC	A,n	Subtract Operand from Acc. with Carry
9E	SBC	A,(HL)	
DD9E05	SBC	A,(IX+d)	
FD9E05	SBC	A,(IY+d)	
9F	SBC	A,A	
98	SBC	A,B	
99	SBC	A,C	
9A	SBC	A,D	
9B	SBC	A,E	
9C	SBC	A,H	
9D	SBC	A,L	
ED42	SBC	HL,BC	
ED52	SBC	HL,DE	
ED62	SBC	HL,HL	
ED72	SBC	HL,SP	
37	SCF		Set Carry Flag (C = 1)
CBC6	SET	0,(HL)	Set Bit b of Location
DDCB05C6	SET	0,(IX+d)	
FDCB05C6	SET	0,(IY+d)	
CBC7	SET	0,A	
CBC0	SET	0,B	
CBC1	SET	0,C	
CBC2	SET	0,D	
CBC3	SET	0,E	
CBC4	SET	0,H	
CBC5	SET	0,L	
CBCE	SET	1,(HL)	
DDCB05CE	SET	1,(IX+d)	
FDCB05CE	SET	1,(IY+d)	
CBCF	SET	1,A	
CBC8	SET	1,B	
CBC9	SET	1,C	
CBCA	SET	1,D	
CBCB	SET	1,E	
CBCC	SET	1,H	
CBCD	SET	1,L	
CBD6	SET	2,(HL)	
DDCB05D6	SET	2,(IX+d)	
FDCB05D6	SET	2,(IY+d)	
CBD7	SET	2,A	
CBD0	SET	2,B	
CBD1	SET	2,C	
CBD2	SET	2,D	

OBJ CODE	SOURCE STATEMENT		OPERATION
CBD3	SET	2,E	Set Bit b of Location
CBD4	SET	2,H	
CBD5	SET	2,L	
CBD8	SET	3,B	
CBDE	SET	3,(HL)	
DDCB05DE	SET	3,(IX+d)	
FDCB05DE	SET	3,(IY+d)	
CBDF	SET	3,A	
CBD9	SET	3,C	
CBDA	SET	3,D	
CBDB	SET	3,E	
CBDC	SET	3,H	
CBDD	SET	3,L	
CBE6	SET	4,(HL)	
DDCB05E6	SET	4,(IX+d)	
FDCB05E6	SET	4,(IY+d)	
CBE7	SET	4,A	
CBE0	SET	4,B	
CBE1	SET	4,C	
CBE2	SET	4,D	
CBE3	SET	4,E	
CBE4	SET	4,H	
CBE5	SET	4,L	
CBEE	SET	5,(HL)	
DDCB05EE	SET	5,(IX+d)	
FDCB05EE	SET	5,(IY+d)	
CBEF	SET	5,A	
CBE8	SET	5,B	
CBE9	SET	5,C	
CBEA	SET	5,D	
CBEB	SET	5,E	
CBEC	SET	5,H	
CBED	SET	5,L	
CBF6	SET	6,(HL)	
DDCB05F6	SET	6,(IX+d)	
FDCB05F6	SET	6,(IY+d)	
CBF7	SET	6,A	
CBF0	SET	6,B	
CBF1	SET	6,C	
CBF2	SET	6,D	
CBF3	SET	6,E	
CBF4	SET	6,H	
CBF5	SET	6,L	
CBFE	SET	7,(HL)	
DDCB05FE	SET	7,(IX+d)	
FDCB05FE	SET	7,(IY+d)	
CBFF	SET	7,A	
CBF8	SET	7,B	
CBF9	SET	7,C	
CBFA	SET	7,D	
CBFB	SET	7,E	
CBFC	SET	7,H	
CBFD	SET	7,L	
CB26	SLA	(HL)	Shift Operand Left
DDCB0526	SLA	(IX+d)	Arithmetic
FDCB0526	SLA	(IY+d)	
CB27	SLA	A	
CB20	SLA	B	
CB21	SLA	C	
CB22	SLA	D	
CB23	SLA	E	
CB24	SLA	H	
CB25	SLA	L	

OBJ CODE	SOURCE STATEMENT		OPERATION
CB2E	SRA	(HL)	Shift Operand Right
DDCB052E	SRA	(IX+d)	Arithmetic
FDCB052E	SRA	(IY+d)	
CB2F	SRA	A	
CB28	SRA	B	
CB29	SRA	C	
CB2A	SRA	D	
CB2B	SRA	E	
CB2C	SRA	H	
CB2D	SRA	L	
CB3E	SRL	(HL)	Shift Operand Right
DDCB053E	SRL	(IX+d)	Logical
FDCB053E	SRL	(IY+d)	
CB3F	SRL	A	
CB38	SRL	B	
CB39	SRL	C	
CB3A	SRL	D	
CB3B	SRL	E	
CB3C	SRL	H	
CB3D	SRL	L	
96	SUB	(HL)	Subtract Operand
DD9605	SUB	(IX+d)	from Acc.
FD9605	SUB	(IY+d)	
97	SUB	A	
90	SUB	B	
91	SUB	C	
92	SUB	D	
93	SUB	E	
94	SUB	H	
95	SUB	L	
D620	SUB	n	
AE	XOR	(HL)	Exclusive "OR"
DDAE05	XOR	(IX+d)	Operand and Acc.
FDAE05	XOR	(IY+d)	
AF	XOR	A	
A8	XOR	B	
A9	XOR	C	
AA	XOR	D	
AB	XOR	E	
AC	XOR	H	
AD	XOR	L	
EE20	XOR	n	

Index

Explorer's Guide to the ZX Spectrum and ZX 81

Are you making the most of your ZX micro?

New owner or confirmed Sinclair user? This book will provide you with a complete and practical guide to the world's most popular micro - the ZX 81 and its stablemate the ZX Spectrum.

It will enable you to take full advantage of the power and capabilities of your micro. The author gives an easy and gentle introduction to BASIC programming before 'changing gear' and delving into more advanced techniques and practical projects. The style is easy to follow with numerous illustrations, so that even machine code and analog simulation become tools you can use. Practical projects interfacing your micro to the outside world are described including an elementary introduction to digital electronics.

Over 40 programs are fully listed with clear, detailed explanations and hints on how to extend and develop them for your own purposes. These original and exciting programs and projects include the following:

- ★ Colour graphics, plotting and drawing
- ★ Animation program 'Walkman'
- ★ Special simulation programs
- ★ Pseudo 3D and real 3D programs
- ★ The one switch typewriter for handicapped people
- ★ A model railway controller
- ★ Games and puzzles
- ★ Mathematical programs

 Addison-Wesley Publishing Company 0 201 14638 X

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/