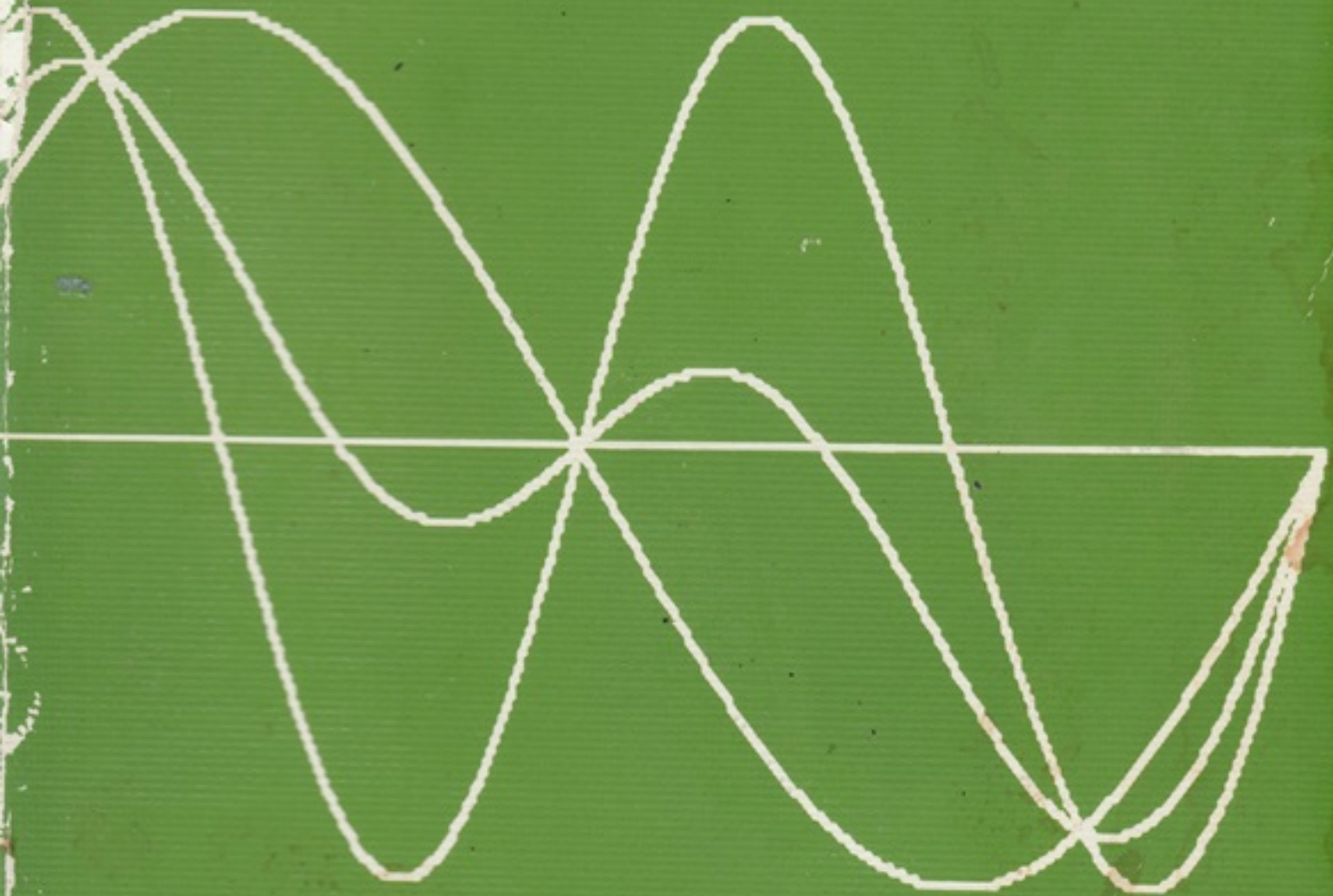
The ZX Spectrum in science teaching



R.A. Sparkes

The ZX Spectrum in science teaching

The ZX Spectrum in science teaching

R. A. Sparkes

Hutchinson
London Melbourne Sydney Auckland Johannesburg

Hutchinson & Co. (Publishers) Ltd

An imprint of the Hutchinson Publishing Group

17–21 Conway Street, London W1P 6JD

Hutchinson Publishing Group (Australia) Pty Ltd
PO Box 496, 16–22 Church Street, Hawthorn,
Melbourne, Victoria 3122

PO Box 151, Broadway, New South Wales 2007

Hutchinson Group (NZ) Ltd
32–34 View Road, PO Box 40–086, Glenfield, Auckland 10

Hutchinson Group (SA) (Pty) Ltd
PO Box 337, Bergvlei 2012, South Africa

First published 1984

Typeset in Times and Univers by Folio Photosetting, Bristol
Printed and bound in Great Britain by
Anchor Brendon Ltd, Tiptree, Essex

British Library Cataloguing in Publication Data
Sparkes, R. A.
The ZX Spectrum in science teaching.
1. Science — Study and teaching (Secondary)
— Great Britain — Data processing
2. Science — Computer assisted instruction
3. Sinclair ZX Spectrum (Computer)
I. Title
507'.1241 Q183.4.G7

ISBN 0 09 158 201 6

For Mum and Dad

Acknowledgements

The ZX Spectrum microcomputer, on which the programs in this book were written, was given to me by Griffin and George and I am most grateful for the use of it. Dave Palmer and Griffin and George between them helped me by providing the laboratory interfaces as well as help and encouragement. Once again the artwork is the product of Annie Hynes and this time I have had help in preparing one of the programs from Dr Leon Firth. I am most grateful to both of them. I acknowledge the support given by the publishers, especially Bob Osborne and Sue Walton. I am especially grateful to teachers who attended in-service courses at St Andrew's College and were willing to try out my ideas and offer further suggestions.

However, none of these can share any blame for the errors and omissions that occur in this book, and I take full responsibility for them. I look forward to receiving comments from readers on how this book, and the use of the Spectrum in the areas I have discussed, might be improved.

Once again most thanks are due to my wife, Margaret, for her encouragement and criticisms and for her patience and understanding. The development of this book and the ideas in it has been at the expense of both Margaret and the children. I can only hope that their sacrifice is found to be worthwhile.

The University of Stirling
1984

Contents

Listed programs

The programs listed in the Appendix are described below. Each has also been given a shortened name as a file name for tape or Microdrive storage.

Program 1	DIGITAL ELECTRONICS (LOGIC)	these programs teach and test the principles of Boolean logic and show the use of a microcomputer in solving logic problems. They require a logic board connected to an interface, details of which are given in the text.
Program 2	LOGIC TEST (LOGICTEST)	
Program 3	LEAST SQUARES PLOT (BESTFIT)	a utility program.
Program 4	MASTERMIND (MASTERMIND)	a science-based game.
Program 5	Z80 SIMULATION (Z80SIM)	teaches the instruction set of the Z80 microprocessor (written by Leon Firth).
Program 6	REACTION TIMER (REACT)	measures reaction times.

The next sixteen programs require the Interspec interface.

Program 7	STOPCLOCK (STOPCLOCK)	measures time intervals with a visual display of the elapsed time in large digits.
Program 8	FAST TIMER (FASTTIMER)	measures time intervals in ten microsecond units.
Program 9	TSA METER (TSA)	measures time, speed and acceleration.
Program 10	ACCELERATION TUTOR (ACCTUT)	a more transparent version of TSA, to allow pupils to see how the computer carries out its calculations.

Program 11	CONSERVATION OF MOMENTUM (CONSMOM)	measures the speeds of two colliding trolleys, simultaneously if necessary.
Program 12	SPEED-TIME PLOTTER (SPEEDPLOT)	plots a speed-time or distance-time graph.
Program 13	FREQUENCY METER (FREQMTR)	measures pulse frequency.
Program 14	PENDULUM PERIOD (PENDULUM)	measures the period of a pendulum.
Program 15	PULSE GENERATOR (PULSER)	produces square pulses of variable frequency.
Program 16	PROGRAMMABLE OSCILLATOR (PROGOSC)	provides alternating voltages with changeable waveforms and frequencies. This program needs a digital to analogue converter.
Program 17	X-Y PLOTTER (XYPLOTTER)	displays two voltages at the same time like an X-Y CRO.
Program 18	STORAGE OSCILLOSCOPE (STRGOSC)	captures analogue data from up to four channels for subsequent display.
Program 19	FAST ADC (FASTADC)	for rapid voltage readings (up to 50 000 per second).
Program 20	DIGITAL MULTIMETER (DIGMULT)	displays voltage, current, power and resistance.
Program 21	CURRENT-VOLTAGE PLOTTER (IVPLOT)	automatically plots I-V characteristics.
Program 22	FOUR-CHANNEL CHART RECORDER (CHRTREC)	displays four channels of voltage input and scrolls horizontally.

The remaining programs do not need interfaces. Their use is described in Chapter 1 and they are referred to throughout the text as examples.

Programs for drill and testing

Program 23	MECHANICS DRILL	(MECHDRILL)
Program 24	INTEGRATED SCIENCE TEST	(INTSCITEST)

Introduction

This book is a ZX Spectrum microcomputer version of my previous book *Microcomputers in Science Teaching*, which was written mainly for PET and Apple users. The differences between these machines and the Spectrum are such that a major rewrite has been necessary. To some extent, this book is also a sequel to *Microelectronics* (Hutchinson, 1984). That book concluded that the most sensible way to introduce students to microelectronics is through programming a computer to control its environment. Accordingly, a large part of this book considers the use of the Spectrum in analogue and digital measurement and control.

To do this, some way of interfacing the Spectrum to other laboratory equipment is necessary. In this book I rely heavily on one particular interface – the Interspec. This is so good that teachers are unlikely to be able to improve on it themselves. It is inexpensive and it would be impossible for the average teacher to produce a home-made version any cheaper. The Interspec has an expansion bus and TTL input and output ports, to which other devices may be connected. I have, therefore, given some guidance on adding extra facilities to the Interspec and this part of the book assumes a knowledge of basic electronics.

To reduce the overall amount of material, I have tried to exclude things that are described in the ZX Spectrum manual (also called the user guide) and I assume that readers are well acquainted with that book. I have attempted to fill in the gaps in the user guide to allow Spectrum owners to get even more out of their machines. I have concentrated mostly on those applications of the Spectrum that are particularly relevant to science teachers. The term 'science' has been interpreted pretty widely and there is a great deal to interest teachers of engineering science, CDT and mathematics too. Most examples are taken from physics, but the principles they demonstrate apply to all subjects. This area is one of very rapid development and new ways of doing things are constantly being found. For this reason I have emphasized the principles involved as well as providing specific examples. Forty programs are listed in the Appendix and these are referred to in the text as examples of the points being made. In addition, many other listings are included in the text to illustrate particular ideas. Note that these examples (which will eventually be available on Microdrive for readers who wish to save time) are not 'idiot proof', that is, they have not been tested and protected against pressing the wrong keys or entering the wrong information etc.

My programs are mainly intended to help Spectrum users to write their own programs. The listings are utilities that can be developed by teachers for their own purposes. There are those who decry this attitude, saying that we can't expect teachers to become program writers. Unfortunately, there is never enough money

in education to pay for the programs that teachers want, which results in teachers having to write their own (or steal them from someone else). In any case program writing is well within the capabilities of the average science teacher (like learning to drive a car).

I often use the analogy of the motor car in this context. If you occasionally need to travel from one part of the country to another in reasonable comfort, you may take a taxi. This will be very expensive. Alternatively, you may learn to drive the car yourself. This will take time initially and is only worthwhile if you expect to do a lot of travelling. Likewise, if you only expect to use the microcomputer on a few rare occasions, or if you want pupils to use it without supervision, then by all means pay the extra and get crash proof programs. But if you intend to make considerable use of the microcomputer, it is better to learn programming for yourself. Then you will be able to take control, you will not be afraid if a program crashes, because you will know how to recover it. You will be able to adapt an unsatisfactory program to your own specification and you will pay very much less for programs.

The effort in writing programs is less in getting them to work than in making them absolutely idiot proof. I appreciate that programs designed for use by novices must have this protection built into them. If this is an important criterion for you, then you will be quite happy to pay for someone to create the program for you. But if you have the ability to write your own programs and therefore the ability to recover from a crash, you will not be so happy at having to pay extra for someone else's lack of competence. Also you will want the ability to stop programs, list them and alter them to your own requirements – commercial programs generally prevent this. One way of overcoming this 'protection racket' is by writing your own programs and making them available to others.

In support of this precept, my programs are presented so that you will be able to modify them for your own applications. If they were locked up on a no-copy disk, the benefits they can give would be more limited. I hope that anyone else making use of these programs will have the same attitude and will acknowledge authorship in the traditional way.

1 The new resource

'Where shall I begin, please your Majesty?'
(Lewis Carroll, *Alice's Adventures in Wonderland*)

One of the unfortunate results of the history of computing is that most people still regard it as a branch of mathematics. A common response to the call to learn programming is, 'I'm no good at maths.' This is a mistake, since there is no longer much relationship between mathematics and computing. For science teachers, the microcomputer is much more a new piece of educational technology than a super calculating machine. Its use is not confined to the mathematics department nor to a computing department. This chapter explores the possible applications of the microcomputer in science teaching.

To emphasize the difference between the traditional computer and its modern counterpart the new phrase 'information technology' has been invented. The modern microcomputer is mainly concerned with collecting, processing and presenting information. The machine should therefore appeal instantly to the teacher, whose task it is to disseminate information in its widest sense.

There are several aspects of such 'presentation'. First of all the microcomputer can be used to display a page of text on its television screen (or VDU). The information could also include a set of figures or a list of names in columns. Alternatively, the information could be presented graphically (i.e. as a diagram or picture or graph) or by an animation or moving picture. This is where the video screen has an immediate advantage over the blackboard or OHP, since animation is not available on the latter. The microcomputer is thus a textbook, blackboard, slide projector and film loop all together in one instrument. It is not restricted to use by individuals and there are several ways in which it can be used with quite large groups. In this case the display is unlikely to be just text, because this cannot be read from a distance (although there are ways of displaying a few words at a time in large letters). More likely it is a picture or an animation that is being presented for all to see, but with the added advantage of interaction. At any stage during a demonstration the students can be asked to suggest how the parameters should be changed. A discussion can then take place as to the likely effects of this change upon the phenomenon being investigated. The changes may then be made to check on the predictions. The general name for this application is **electronic blackboard**, where the microcomputer is used by the teacher in front of the whole class.

The microcomputer is also a powerful tool for helping small groups of pupils. Until class sets of microcomputers become available, it is envisaged that this

application will be confined to use by students in a stations laboratory (where there are a number of work-stations and the students move from one to the next). The microcomputer can thus be used by small groups for short periods of time within a lesson. Alternatively students might use the computer in a library or resource centre. I use the generic term **computer assisted learning** or **CAL** for this application.

At the other end of this spectrum, the microcomputer can be used by one individual pupil working alone. The program being used might be simple drill and practice or a tutorial or the microcomputer might be managing a complete programme of work, adjusting the level of presentation to the particular abilities of each individual pupil. One reason why microcomputers have suddenly become important is because they make the dream of individualized learning a reality. The difficulties of managing the workcards and the tests etc. that are needed in the self-paced learning situation are overcome if they are presented by the microcomputer. New material can be written on the screen for the student to read and answer questions about. If the student is correct, then some other material can be presented, but a wrong answer causes the microcomputer to behave differently, either by presenting the question again or by branching to a remedial teaching loop. It is this ability to react differently to different situations that makes the microcomputer more powerful than any other resource we have had before.

The interaction between the user and the microcomputer creates possibilities for monitoring the teaching process much more efficiently than hitherto. The process of instruction can be halted frequently to check that the student is still following. This is something that every teacher tries to do but cannot achieve in the conventional way for each individual student. Given these facilities, the microcomputer's role in programmed learning is obvious.

Scientists have an application of microcomputers that is peculiar to their discipline – its use as a powerful laboratory instrument. We have already reached the stage where no physics laboratory is complete without a microcomputer and I think that this situation will soon apply in other areas. With suitable transducers and interfaces the computer is fast becoming the only equipment in some industrial laboratories. I do not think that this will happen in schools, but they do need to mirror the real world to some extent. The Spectrum may be used to measure almost any physical quantity desired. At a rough estimate its use in this way can save up to a thousand pounds worth of alternative apparatus, as well as enabling some hitherto unmeasurable quantities (like acceleration) to be displayed. This is my own favourite use of the microcomputer and much of this book is devoted to it.

Inside every microcomputer is an incredibly powerful device, called a **microprocessor**. By talking to this device, new horizons can be opened up, especially for animated diagrams and for using the microcomputer as a laboratory instrument. Because this is a new idea for most teachers it is presented in Chapter 6 as a microcomputer simulation and tutorial, providing a step by step

approach to the principles of assembly language programming. This is intended not only to explain microprocessor instructions, but also to demonstrate the advantages of a computer simulation. Readers who follow this through might care to reflect on how this way of visually presenting a new topic could be transferred to teaching in other areas, for example, the operation of a nuclear power station or the electrics of a motor car.

Outside the classroom, the microcomputer could take over the role of keeping records, in the same way that bigger computers have been doing in commerce for some time. As might be expected, a great deal of research and development has already been done in this area, and there is little point in any individual teacher doing it all again. There are several projects under way on the development of administration packages for schools and, before very long, these will become generally available. Not only will these include student records, timetabling, equipment records, library loans, etc., but also there will be complete packages for marks processing and assessment. Even if no other part of the school is affected by microcomputers, the school office certainly will be.

Under this heading too, I consider the use of a microcomputer as a wordprocessor or texteditor to be very exciting. Readers of this book will note the way that parts of a previous book *The BBC microcomputer in science teaching* have been used here too. It was a simple matter to call up the text of that book onto the screen, to select the parts required, alter them and save them once more on disk. There are already several such wordprocessor programs available for the Spectrum and their use more than repays the cost. Teachers who prepare their own worksheets will find that their productivity increases by a factor of three or four at least. There is an even bigger saving of time for one-fingered typists like me.

Let us now explore some of these ideas in more detail with particular examples to illustrate the principles discussed. Note that these examples (which are listed in the Appendix and will also be available in machine-ready form for readers who wish to save time) are not thoroughly tested programs, guaranteed to work with even the most stupid of users. They are examples only of the sort of things that can be done with microcomputers. Nevertheless, they have been tried and they do work. Provided the user has a moderate understanding of programming, they will present no problems.

Specific examples

Testing

A common use of microcomputers in schools is testing. This means not so much the end-of-term examination as the routine question and answer sessions, with which teachers attempt to reinforce learning. Because time does not permit the conventional method to be used on an individual basis, not all children benefit from it. Indeed, the public nature of the responses often causes pupils to adopt

strategies for avoiding an answer. If a child remains dumb for long enough, most teachers will direct the question elsewhere. The microcomputer can be viewed as a resource for handling question-and-answer sessions.

At the simplest level are numerical tests: the microcomputer is perfectly capable of setting its own arithmetic questions and working out the answers for itself. MECHANICS DRILL (23) illustrates this application. It would be relatively easy to adjust the number range and the difficulty of the programs like this to suit the user. For practical purposes this program needs to be improved in several ways. Where is the power of the microcomputer being used? There are no diagrams or pictures or animations. INTEGRATED SCIENCE TEST (24) shows what can be done in this area. In this program the number of correct and wrong answers may be counted, so that a final score can be given. It is also useful to note which questions the student gets wrong, in case this reveals the source of the ignorance. A properly structured test would be written for this purpose anyway. A way of doing this can be seen in the score routines of this program.

A particularly powerful use of the microcomputer is to allow the student to ask for help, if the offered problem proves too difficult. This could be given automatically after, say, three attempts, or it could be available upon pressing key H. After the first few questions, it is a little wearisome to a student to be given exactly the same 'well done' response each time. No teacher would do this, so why should we accept a lower standard from the microcomputer? It is not difficult to create a whole range of responses in an array, and to pick one out at random. Also, thought should be given to more dramatic ways of responding. Arcade invaders leap about with delight, when they score a hit on the defenders; why can't the same graphics be used in education? As a suggestion, the fast screen transfer routine discussed in Chapter 7 illustrates how this might be done, by flashing pictures onto the screen for a short time. This could be incorporated into a test program to indicate whether the student has the right or wrong answer.

This area is also known as drill and practice. The microcomputer is programmed to ask the questions and to monitor the responses. To do this there has to be some way for the user and the microcomputer to interact with the user, an aspect which is covered in the next chapter. INTEGRATED SCIENCE TEST also illustrates several of the basic principles of using multiple choice items. This program can be used as the framework for any other multiple choice test. The items are kept separate from the main program, which handles all keyboard inputs and scores etc. The question numbers, clues and correct responses are passed to procedures as parameters. Scoring is a separate routine and the final presentation of the results is also self-contained. Note the way that graphics have been included with each item. These are not essential in all cases, but they do increase motivation.

The most exciting thing about test examples presented via the microcomputer is that children tend to treat them more as a game. They aim to 'beat the computer' or to 'do better than last time'. The longer test, from which INTEGRATED SCIENCE TEST was derived, was the one that made me realize the power of the

microcomputer. Some children ran the test again and again to see if they could get full marks. I have never noticed this in a traditional school test.

Simulations

Almost any phenomenon, model or experiment can be imitated or simulated by the computer. Some programs of this type give tables of numbers as results, while others give graphs or animations. GRAVITY (35) is an example of the former and the remaining simulations show the use of graphics.

Computer simulations are most useful where the real experiment is impossible (negative gravity) or very difficult to perform satisfactorily (Millikan's experiment) or not accessible (the behaviour of an atomic pile). I do not think that students should carry out computer simulations of experiments, where the practical experiment itself could be performed. A microcomputer could be used to demonstrate, for example, how to titrate an acid against an alkali. One could press keys to allow the acid to drip in and, with high resolution colour graphics, could produce a superb effect of the indicator changing colour. A meter could be displayed also to indicate the current pH as the acid is added. As an introduction only, this could be very useful for showing the student what steps were involved. The only objection to this would be if it replaced the actual experiment.

There is also another danger in simulation experiments, of implying that one is actually observing nature. Students may come to think that the characters moving around the screen are behaving just like molecules in a real gas. This cannot be true, because we have no notion of what the molecules of a real gas are actually doing. We can make observations and draw conclusions about their behaviour and then produce simulations that appear to produce the same behaviour. But that does not mean that the gas molecules are like the particles on the screen. The students are really being encouraged to 'discover' our model of the behaviour of the molecules, which is the reason why the simulation experiments must be integrated with experiments on the real world, so that our theories about its behaviour can be tested.

Programs 27 to 40 are straightforward simulations of physical events, some of which make use of machine-code graphics to achieve the necessary speed. The calculations needed to keep 256 particles continuously moving at once are quite beyond the capabilities of BASIC. RADIOACTIVE DECAY (29) is a simulation of the decay of radioactive particles using the RND function of the Spectrum. A graph of the number of nuclei remaining after each time interval is displayed. The aim of the simulation might be for students to discover about half-life from a series of runs, but a teacher might wish to use it for a different purpose instead. For example it could be used in comparison with CAPACITOR DISCHARGE (37) and students asked why the results are so similar from such different physical starting points. Alternatively, it could be incorporated into a CAL package and the student instructed to make certain observations.

SUM OF TWO DICE (30) is another example of the use of the random number generator to simulate the shaking of two dice. The program adds the dice together

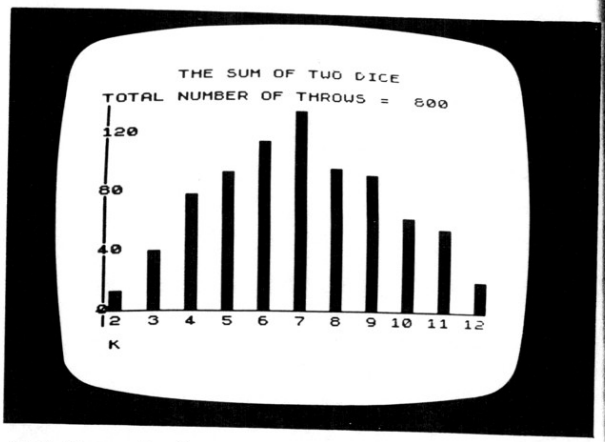


Plate 1 The sum of two dice

and displays the number produced each time. This program illustrates the graphics capabilities of the Spectrum in displaying a bar chart, while at the same time continuously updating it (Plate 1).

Some simulations are designed to get across ideas of the behaviour of waves – STANDING WAVES (31). This shows what happens when two waves, travelling in opposite directions interfere to produce standing waves. WAVE SUPERPOSITION described in Chapter 7 is designed to explain the relationships betweeen speed, frequency and wavelength and also to demonstrate the nature of a transverse wave. The amplitude, frequency and relative phase between two waves may be altered and the production of beats between two waves of different frequency demonstrated. Classical interference between two waves that only differ in phase may also be shown. The way that the microcomputer is used to obtain these effects is discussed in detail in Chapter 7 – basically they use machine-code plotting or scrolling routines.

LONGITUDINAL WAVES allows the user to investigate the propagation of a longitudinal pulse. By holding down key Z, a continuous stream of waves can be generated. Then, by setting the reflection coefficient to zero, the motion of the individual particles is clearly seen (simple harmonic motion). A reflection

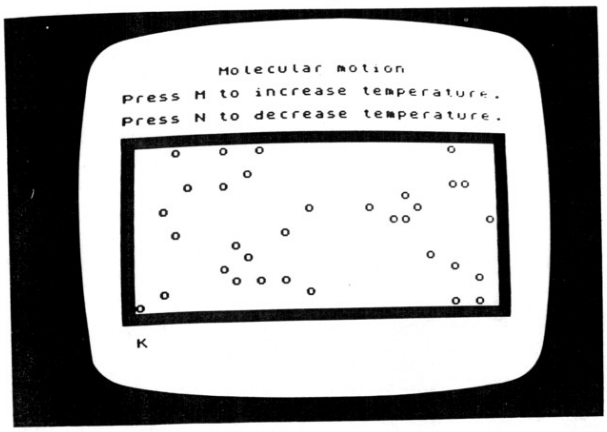


Plate 2 Molecular motion

coefficient of 1 causes the wave to be reflected in phase, thus producing standing waves with an antinode at the reflected end. Conversely, a reflection coefficient of –1 creates standing waves with a node at the reflecting end. The positions of nodes and antinodes are clearly visible and can initiate a detailed discussion on the whole nature of longitudinal wave motion.

Another application of machine-code techniques is to keep a record of the positions of characters on the screen and so to move them around under the control of certain laws. Graphics characters can be directed across the screen in straight lines to bounce off the walls simulating the behaviour of molecules. MOLECULAR MOTION (33) demonstrates what happens to gas molecules at different temperatures (Plate 2).

Three demonstration programs for computer science are included. ANALOGUE-DIGITAL SIMULATION (27) shows how the computer interprets a decimal number as a set of memory switches or as a 'continuously' varying analogue quantity. BYTE SIMULATION (28) explains the principles of adding, subtracting and shifting binary data in a byte of memory (Plate 3). Z80 MICROPROCESSOR SIMULATION (5) teaches some of the properties of the Z80 instruction set and, in conjunction with the tutorial in Chapter 6, provides a good introduction to machine-code programming.

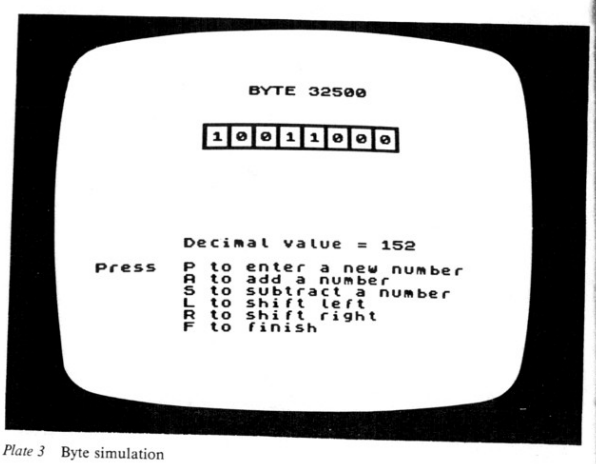


Plate 3 Byte simulation

Computer assisted learning

This area has many names depending upon whether it is emphasizing what the program is doing (instruction) or the student is supposed to be doing (learning). I shall ignore the fine distinctions involved, while still using the general term or CAL, for short. The above discussion of drill and practice inevitably leads on to the use of the microcomputer for CAL. INTEGRATED SCIENCE TEST moves some way towards it, since that program replies to each response with a statement about why the chosen answer is correct or wrong. It is clearly possible to integrate such testing with the teaching of new material in the same way. The idea is to present the topic and then ask questions to establish whether the student understands. Then, if it becomes clear that the student does not understand, remedial action can be undertaken.

A program that does this is termed a **tutorial** and there are many in circulation. The most common are self-instructional tutorials in BASIC programming. Most students, particularly of those subjects which lend themselves to linear progression, such as mathematics and computing, find such tutorials useful. They may even prefer them to traditional classroom methods, because of the immediacy of the feedback and the fact that they can learn at their own pace. Programs like this are not difficult to write, but they should use the full range of

interaction, reinforcement and, of course, graphics that is available. Several author languages, like PILOT, exist to aid writers of CAL programs, but these can be restrictive. They were not developed with microcomputers in mind and may need special adaptation to allow an author to incorporate sound, graphics or other special techniques.

There is, though, a great deal more to CAL than is implied above. To begin with, there is a clear distinction between teaching and telling. Too many of the self-teaching packages, that have been published so far, fall into the latter category. What is involved in producing a good CAL package?

There are two broad categories of CAL programs, one of which favours a structured approach to learning and the other a more open-ended approach. The former is based on programmed learning theory, which may be summarized as follows:

1 The main objectives of the topic to be learned are specified, in terms of observable outcomes, as precisely as possible. Not 'the student should understand something about molecular weight', but specific, like 'given a list of ten chemical compounds and a table of the atomic masses of the elements, the student should be able to calculate the corresponding molecular masses for at least seven of them'.
2 The objectives should then be listed in hierarchical order, in the sense that each objective earlier in the list should not be dependent upon objectives that come later. For example, the following objective should be attained before the one stated above, 'given a list of ten chemical compounds, the student should be able to write out the corresponding chemical formulae for at least eight of them'.
3 The next step is to arrange the objectives into a learning sequence. Teachers tend to do this automatically, so they usually find no difficulty here. The difference with programmed learning is the attempt to ensure mastery of the earlier objectives before the later ones are tackled. One of the difficulties of traditional classroom teaching has been the insistence that all pupils should progress at the same rate. Thus pupils who had a particular learning difficulty, might never acquire later objectives, not because they were unable to, but because they had never quite mastered the earlier ones. This is why the objectives above are criterion referenced. Students do not just have to get higher marks than average, they actually have to attain the external standard set by the objectives.
4 The learning sequence is then turned into a series of lessons, using appropriate teaching strategies for each objective. At certain stages throughout the sequence, tests have to be devised to see whether a student is ready to proceed to the next objective. These diagnostic tests are not stored up for the students' end-of-term grades, their purpose is to inform the student of his or her mastery of each particular objective.
5 Finally the package needs to be tested on a sample of students similar to those who will ultimately use it. Any or all of the preceding stages may have to be modified in the light of this experience.

A CAL package is thus not just something that any knowledgeable person can write down in an evening. Estimates vary as to the length of time needed, but a good

average figure is that 100 hours of development time must be devoted to produce material to keep a student occupied for one hour. So an expert programmer could put a year's full-time work into a CAL package to keep a class occupied for one week! The failure of programmed learning in the past has not necessarily been that it doesn't work, but that there were not enough people around to write the packages. This position has not changed with the introduction of microcomputers. It requires a massive effort to produce good software.

Even then there are hardware problems to be overcome. With graphics and animations a complete teaching package, which could adapt its teaching to the individual needs of its students, could not be run with a cassette system for program loading. A Microdrive or some other form of readily accessible mass storage is essential.

Should teachers, therefore, give up the whole idea of CAL? I do not think so, because it can never come, unless there is a substantial number of teachers who have experience of it. But I think that this is a task for a properly funded team of writers, not individuals. Unfortunately the ease with which software can be copied is likely to deter commercial organizations from being interested.

Teachers, or better still a group of teachers, could begin by taking some topic that is particularly suited to a programmed learning approach: one that is linear in structure, will fit into the video text method of presentation and where it is easy to write the objectives. The commonest fault is to attempt too much, so that insufficient time is spent in ensuring the mastery of each component part. When the program is written, several trial runs with students (and not just the school's computer addicts) should be made with the teacher in attendance. They should be challenged to crash the program, if they can. All problems discovered by them should be noted and rectified. Only then should it be placed on the market; it should not be the end users who have to debug the programs!

There is one powerful reason for not spending a great deal of effort at the moment on CAL (apart from the fact that few schools possess a classful of microcomputers) – the technology is changing fast. Within a few years the video disk will be used to present the graphics, text, tests and other items that currently have to be put into a CAL program. In future the microcomputer will become much more of a manager, calling up from the disk the current lesson and also having previous lessons available for remedial review. With a single video disk holding the equivalent of several hundred floppy disks' worth of information, CAL will no longer be a dream.

Discovery learning

The other way of using the microcomputer for CAL is, in my opinion, much more exciting anyway. It is also less likely to be superseded when the video disk arrives. This is its use in open-ended investigation. Instead of the computer asking the student, the student interrogates the computer. Already several database programs exist (e.g. MICROQUERY and QUEST) to allow students to obtain information by typing certain keywords into the computer. In biology this

promises to be very useful since a student can then carry out a search without being forced into a particular direction by the program. At a simpler level many programs can be developed that allow the student to determine what he or she would like to know.

Imagine that you wanted to teach a student about standing waves. This could be done by direct instruction in the laboratory using a helical spring or Melde's experiment with the teacher pointing out the essential details. Or it could be left to the student to discover the principles for himself or herself. My experience is, however, that students do not know what to look for when doing an experiment and help is needed. STANDING WAVES (31) strips away the inessentials and allows the pupil to concentrate on the features that are important. A set of programs based upon STANDING WAVES and covering most aspects of wave motion is available from Griffin and George. With these, the student may alter the frequency and speed of the waves and then observe the results. This approach does not teach directly, but it does point the student along a particular path. There is no guarantee that learning will take place. But all our experience indicates that if it does, then the student will not just have learned the facts, he or she will have gained an insight, which could transfer to other properties of waves too.

Most of the simulation programs listed in this book were originally devised for this purpose. Although they illustrate the principles of discovery learning, their use is not restricted to it. The versatility of the microcomputer ensures that a program can be used for many different purposes, only a few minutes of adaptation being required.

Number-crunching

A glance at a list of available software reveals programs on Fourier transforms, least squares fit, linear circuit analysis, linear programming, numerical methods, integration by Simpson's rule and so on. The microcomputer is being used as a programmable calculator, with all the advantages of screen display and editing, error detection and program storage.

There are occasions in teaching when an equation needs to be solved many times and where the result is more important than the solution itself. One example is typing experimental data into a microcomputer to obtain an automatic straight-line plot – LEAST SQUARES PLOT (3) (Plate 4). In this case the important aspect is the interpretation of the data, not the long process of plotting it out by hand. GRAVITY (35) gives another instance, calculating the height of a ball thrown vertically against gravity. It is the nature of the motion that is being investigated, not the solution of algebraic equations. Even here though a graph of the results would be even more meaningful. The techniques of graph plotting with the Spectrum are fully discussed in Chapter 3.

Modelling

The equation of motion used in GRAVITY is a mathematical model of the behaviour of a real stone falling. It is inaccurate because it ignores certain features

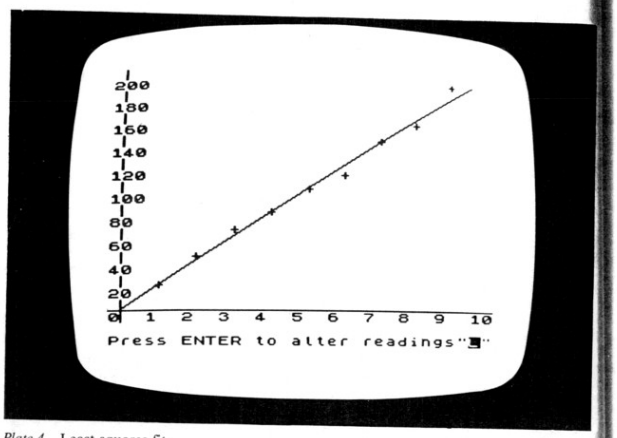


Plate 4 Least squares fit

such as friction, but it does give some insight into the nature of the motion. In Chapter 3 we shall discuss ways of making the model more real by using iterative methods. Physics and chemistry abound with such models and most students can understand an equation much better if they can see what happens to it when different parameters are changed. For example RESONANCE (36) uses a simple technique to plot the resonance curve for an LCR circuit. The student may observe the effect of altering the capacitance or the resistance of the circuit.

Usually in science we eliminate some of the variables in order to make the mathematical analysis of the phenomenon easier. The microcomputer allows some of these other variables to be considered. GRAVITY ignores the effects of friction, but this is not too difficult to incorporate provided the traditional technique of analysis is abandoned in favour of the iterative method. PROJECTILES (38) uses this technique to provide a more accurate picture of the motion of real stones being thrown through the air. The iterative method, which is discussed in detail in Chapter 3, is particularly powerful when dealing with central forces since the motion of satellites is obtained without recourse to integral calculus (a solution that Newton would himself have liked). In addition, the motion is not confined to the circular case, elliptical motion is no more difficult for the microcomputer than

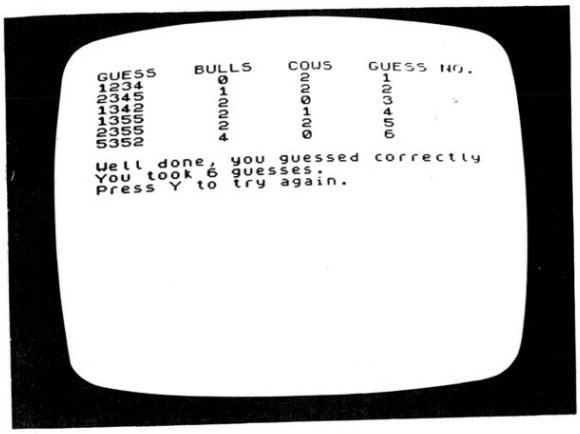


Plate 5 Mastermind

is the imaginary circular case. As well as doing this, NEWTON (39) is also a mathematical simulation of Newton's thought experiment on why the moon falls towards the earth and yet never gets any closer. RUTHERFORD (40) is a variation of this program, that replaces the attractive force with a repulsive one to simulate the scattering of alpha particles by gold nuclei.

Games

If the recent fury that has developed over video games does not obscure the issue, there may be very little distinction between this section and discovery learning. It may be possible to distinguish between educational and recreational games, but I doubt if even that could be maintained. There are reports of slow learners who have been very greatly helped by space invaders, which, it is claimed, has increased their span of attention at other, more academic activities. Nevertheless, I do think that some games exercise the intellect more than others and it is in these that I am interested.

A standard favourite amongst beginners to computing is learning to program MASTERMIND (4) or one of its traditional forerunners like Bulls and Cows, which is easier because it uses numbers (Plate 5). I like it because it illustrates the

essence of the scientific method. Each guess is an hypothesis to be tested by experiment. Each experiment leads to a refining of the model until perfection is achieved. Most important, each guess should be wholly consistent with all the evidence so far gathered and it should also be framed so as to achieve the maximum amount of new information. Hence this game requires a strategy for getting the answer. I should like to improve on the program given by encouraging the user to develop the correct strategy. I have seen even older children adopting a trial and error method rather than using the information in previous guesses as a basis for the next. If strategy training could be done here, would a similar system be possible to teach students a strategy for, say, solving equations or devising laboratory experiments? It is clearly an important potential development.

Guessing games are among the most popular and I have included my own – ELEMENTS (25). I am not sure that I agree with the traditional version of this game (HANGMAN) on educational grounds. Doesn't learning theory require us to reward success rather than punish failure? I have included my version to illustrate the technical ways of handling guessed inputs. The game is easily adaptable to other topics by changing the nature of the words: this one is based upon the elements. This is easily done because they are all contained in data statements at the end. This program chooses the next word at random and, to avoid repetition, contains a routine to pick each word once only. Therefore, if you intend to adapt it to your own use, you will need to alter the maximum number of words available (103 in this case) wherever it appears in the program. A similar program, with different objectives, is listed as CHEMICAL NAMES (26).

My favourite guessing game is called ANIMALS and several versions are available for the Spectrum. The computer 'learns' the names of different animals and guesses the one that you are thinking about, by asking a series of yes/no questions.

```
Does your animal live in the sea?
Does your animal fly?
Does your animal have horns?
```

When the computer gets to the end of its branching search without success, it gives up and asks the user to say what the animal is and to suggest a suitable question for distinguishing it from the previously named animal. Thus the computer 'learns' a new animal. The form of the game usually given needs alteration, since it asks whether the animal in question has long ears before even discovering whether it is insect, bird, fish or mammal. As a strategy for guessing, it is therefore very poor. In the hands of a competent biologist the program could be invaluable for teaching about classification. In chemistry too it could be used to develop an understanding of the periodic table.

The new curriculum

I suppose it is inevitable that teachers first use microcomputers to enhance the current curriculum. At the drill and practice level it is even reinforcing current

syllabuses. The discussion under discovery learning above, though, does imply that the microcomputer will eventually alter both *what* as well as *how* we teach. The way forward has been shown by Papert and the LOGO language. With this pupils can explore the world of space, shape, size and angle and discover the properties of language at the same time. Can we use a microcomputer as a context-free method of developing process skills in science in the same way?

It might be possible to invent different worlds with particular properties to be investigated. Gamov's *Mr Tompkins in Wonderland* describes worlds where the speed of light is reduced to ten m.p.h. and where Planck's constant is unity. The purpose of this is not just entertaining science fiction, it is rather to explain the real world by exploring the properties of an imaginary one. I should like to see this done with a microcomputer. At a simple level GRAVITY and some of the simulation programs in Chapter 3 allow the acceleration due to gravity to be altered from its normal value. Could this be extended to exploring situations where an inverse cube law of force existed? What would be the properties of visible light if our eyes could see into the X-ray or microwave regions? Why are all current adventure games based on myths and legends? Why can't some imaginative scientist devise an adventure on, say, exploring the atom, sailing through the blood stream in a miniature submarine or managing an atomic power station? This exploratory use of microcomputers cuts across traditional subject boundaries, so that science, mathematics and art become united.

At the moment few schools possess teletext facilities allowing them access to vast data-banks of information. When these do arrive they will raise important questions regarding the content of school syllabuses. In particular we shall have to question the current emphasis upon knowledge. Brain of Britain 1984 is the one who can remember the most information. What will be the value of this skill when we each have access to any desired information via a home computer terminal? A good memory will be as out-moded as the ability to extract square roots by pencil and paper (which I was taught). The skills we shall come to prize will be the processes of handling information. Brain of Britain 1999 will be the one who can solve problems.

Despite a generation or more of protagonists for process skills, most school science (and nearly all university physics) is still heavily content based. Students have little chance to apply their minds to new situations, they are too busy learning about old ones. Given the opportunity, the microcomputer could be used to put us back on the right track. This is why I call this section 'the new curriculum'. I believe that the introduction of microcomputers will be far more revolutionary than any of us expect.

2 Programming techniques

'I'm afraid I don't quite understand,' said Alice.
'It gets easier farther on.' Humpty Dumpty replied.
(Lewis Carrol, *Through the Looking Glass*)

This chapter is not an introduction to BASIC programming. I assume you can do that already. Instead it attempts to explain some of the things that the Spectrum user guide omits (because they are of specialist interest). It also looks at ways of improving tutorial programs with the use of graphics, proper display of text and methods of collecting and processing responses from the keyboard. Finally, this chapter explores the processes involved in the development of an educational program.

Bits and bytes

The heart (or perhaps it should be brain) of any computer is its **central processing unit** (**CPU**). A microcomputer like the Spectrum is no exception, its CPU is the Zilog Z80 microprocessor. Note that this word 'microprocessor' refers only to the CPU. People who use it in place of the word 'microcomputer' are fundamentally incorrect. The microprocessor is only one of many chips inside the microcomputer, even if it is the one which does all the work. Figure 2.1 is a simple picture of the way that a microcomputer works.

For most purposes the **input** to the microcomputer is via its keyboard. The output is via the television screen or monitor (in computer jargon this is a VDU or visual display unit). One purpose of this book is to show you how to make use of other forms of input and output.

The microprocessor is a programmable device. The programs it runs are of two kinds, the **resident** program and the **user** program. The same Z80 microprocessor is used in the RML 380Z, the Exidy Sorcerer and the ZX81 as well as in the Spectrum. These machines all behave in different ways because they have different operating systems, which tell the microprocessor how to read the keyboard, where to print characters on the screen and so forth.

A programmer can write different application or user programs for the microcomputer to execute. For example, one program can be written to draw pictures on the video screen, another can search through a list of numbers for the smallest value. This user program will not remain in the machine after it has been switched off (it is said to be **volatile**). Every time that the microcomputer is switched on, a new user program must be placed in its program memory. This can, of course, be entered from the keyboard or loaded from cassette tape. To allow the microcomputer to store different programs, the memory for user programs is

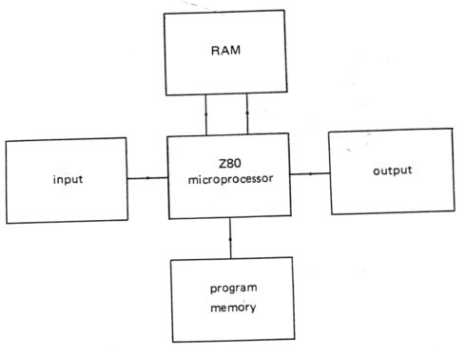


Figure 2.1 The microcomputer as a system

alterable. It is called **RAM** (which stands for **random access memory**).

A useful picture of RAM is to imagine it as being rows and rows of on/off switches. An electric light switch can be up or down. If you switch the light on, it stays on. If you switch the light off, it stays off. It is a simple memory device. The Spectrum's RAM contains thousands of simple switches rather like this light switch. Each one is a single **bit** of memory and when it has been switched on, it stays on and when it has been switched off, it stays off. These switches are in groups of eight and each group is called a **byte**.

From our point of view each set of eight bits can be considered to be a binary number. With eight bits there are 256 possible binary numbers (in the range 0000 0000 to 1111 1111) and any information received by the microprocessor must be one of these numbers. Each digit of this binary number is either a 0 or a 1. To make it easier for us, we usually convert these binary numbers into decimals using the following values for each bit position:

Binary	Decimal
0000 0000	0
0000 0001	1
0000 0010	2
0000 0100	4
0000 1000	8
0001 0000	16

```
0010 0000     32
0100 0000     64
1000 0000    128
```

The leftmost bit (worth 128) is called the **most significant bit (MSB)** and the rightmost is called the **least significant bit (LSB)**. Converting such numbers to and from decimal is easily accomplished – it is part of most pupils' mathematical upbringing. The place-value of each bit position is multiplied by the bit itself and the resulting numbers are added together. Thus 1011 0110 is:

```
  1     0     1     1     0     1     1     0
128  +  0  + 32  + 16  +  0  +  4  +  2  +  0
```

which is 182 in decimal.

Each byte can be used to store a different bit-pattern or code, which can then represent numbers or letters or even whole messages. The Spectrum can remember the letters of the alphabet, the decimal digits from zero to nine and other characters like ?, !, (,) and +. Even a blank space has to be represented by a special code. The Spectrum also has graphics characters (pictures) and user-defined characters.

The **American Standard Code for Information Interchange (ASCII)** is quite often used in computers to represent letters of the alphabet. For example:

```
0100 0001 (decimal 65) represents the letter A
0100 0010 (decimal 66) represents the letter B
0100 0011 (decimal 67) represents the letter C
```

The Spectrum also uses ASCII to represent characters. This is given in full in Appendix A of the user guide.

Types of program

To make it easier to produce programs, they are often written in the language called **BASIC**. The microprocessor does not understand BASIC, it is a digital device and only 'understands' digital signals; hence information is sent to the microprocessor as a set of HIGH and LOW voltage levels. The Z80 microprocessor has eight lines for this information and it reads all eight lines at once, that is, it reads one byte at a time. One measure of the power of a computer is the number of characters that it can store, that is, the number of bytes in its RAM.

The 16K Spectrum can store about 16 000 characters and the 48K model about 49 000.

The instructions given to the microprocessor are also in an eight-bit code. For example:

```
0011 1010 means 'fetch a number from the memory'
0011 0010 means 'send a number to the memory'
```


It might seem that having only eight bits to a byte is very limiting, if we can only give the microprocessor 256 different instructions. However there are only seventy keys on a typewriter keyboard, yet how many different books can be written? It is clearly the sequence of the instructions given to the microprocessor that is important. This sequence of instructions is called a **program**.

Machine language

One way of programming the microprocessor would be to give it sequences of binary numbers via eight switches. A separate switch could be used to tell the microprocessor when the next coded instruction was ready. This is obviously very slow and many mistakes might be made. (It was the way that the early computers were programmed.)

A better way would be to type in the required numbers from the keyboard. A still better way would be to write all the binary numbers into the memory beforehand. The microprocessor could then fetch each one in turn and execute it. This is the purpose of a **machine language monitor**. (The word 'monitor' here has no connection with the television monitor.) The Spectrum does not possess a machine language monitor, but several are available as separate programs. Older microcomputers, like the PET and the Apple have machine language monitors as part of their resident program.

Assembly language

Using a machine language monitor is still slow, laborious and very prone to mistakes. An **assembler** allows the programmer to type in instructions for the microprocessor in a special mnemonic language. For example, the instruction to the microprocessor to fetch a number from the memory is 0011 1010 in binary and LD in **assembly language**. The latter is easier to remember, it is the mnemonic for LOAD.

It is possible to buy an assembler program for the Spectrum, which takes each line of an assembly language program and turns it into the correct binary number for the microprocessor to execute. This is a very powerful tool for a programmer especially when the Spectrum is being used for measurement or control. There is, however, a way of using machine code instructions without either an assembler or a machine code monitor. This is discussed in detail in Chapter 7.

BASIC

Even assembly language is not simple, so **high-level** languages have been developed. BASIC is one of these. Instead of codes it uses English words. For example:

```
PEEK    means 'fetch a number from the memory'
POKE    means 'send a number to the memory'
PRINT   means 'write something on the television screen'
```

BASIC still uses codes to represent letters, it is just that these codes are not

normally visible (since they are not usually of interest). They can, however, be accessed through BASIC. For example, the command:

```
PRINT CODE "B"
```

will produce the ASCII code for the letter B. Another way is to use the CHR$ function, which is the opposite of CODE. It turns a coded number into its proper character. For example:

```
PRINT CHR$ 65
```

will print the letter A, which is the character for the code 65. These functions provide useful programming techniques, as the hexadecimal conversion programs below will show.

Although we use words in BASIC statements, the microprocessor has to turn them into its special codes before it can understand them. The word POKE is turned into the code 0011 0010 and then the microprocessor can carry out (or execute) this instruction. The microcomputer has a special program, called the BASIC interpreter, to turn BASIC statements into the binary numbers needed by the microprocessor. This interpreter also contains error checking, so that errors in programming give the error messages to the programmer. BASIC is so very easy (by comparison with the other methods) that only a fanatic would use assembly language unnecessarily. BASIC programs are used wherever possible throughout this book. However, it takes time for the interpreter to collect each BASIC instruction and to turn it into the correct machine code instructions, so BASIC programs run hundreds of times slower than their counterparts written directly in machine code. So for certain purposes, like rapid measurements and animated graphics, machine code programs are essential.

The resident program

The operating system and the BASIC interpreter are part of the resident program in the Spectrum. Since this must always be there when the machine is switched on, it is **non-volatile**, and is written in **ROM (read only memory)**. ROM cannot be changed, but it has the advantage of not disappearing when the machine is switched off. Because it has to do so much, there is quite a lot of it in the Spectrum. Some of this is useful to us even when we are not using BASIC.

Hexademical notation

In BASIC most users are unaware of binary, but when we start to use the microprocessor directly, it is not possible to avoid it altogether. But what are we to make of a binary number like 1110 0110 1010 0111 – even copying it down might produce errors. We use a shorthand system called hexadecimal coding. Each set of four bits (half a byte is called a nybble) is represented by a code according to the following table.

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

The sixteen-bit number 1111 1100 0000 0001 is thus written as FC01. To show that it is a hexadecimal number, it is usually followed by the letter H, but there can still be a confusion between a hexadecimal number like FEEAH and a variable name. It is therefore common practice with most assemblers to prefix with 0 every hexadecimal number that begins with a letter, so our number finally becomes 0FC01H. The addresses used in the Spectrum have sixteen bits giving a total of 65 536 different locations (from 0000H to 0FFFFH), the contents or data in any location are eight-bit bytes with 256 possible different values (from 00H to 0FFH).

Hexadecimal numbers are converted to and from decimal in a way similar to binary numbers, except that the place values are now in multiples of sixteen instead of two.

```
     F        C       0       1
   x4096    x256    x16     x1
15x4096 + 12x256 + 0x16 + 1x1 = 64121
```

The programs ANALOGUE-DIGITAL SIMULATION (27) and BYTE SIMULATION (28) may be found useful in explaining these ideas more fully. The following programs will convert numbers from hex to decimal or from decimal to hex automatically.

```
10  REM HEX TO DECIMAL
20  LET A = 0
30  INPUT A$
40  FOR I = 1 TO LEN A$
50  IF A$(I) >= "A" THEN LET A = 16 * A + (CODE A$(I) − 55)
```

```
60  IF A$(I) < "A" THEN LET A = 16 * A+(CODE A$(I) - 48)
70  NEXT I
80  PRINT A$,A
```

```
10  REM DECIMAL TO HEX
20  LET A$=" "
30  INPUT A
40  PRINT A
50  LET N = A - 16 *INT (A/16)
60  IF N > 9 THEN LET A$ = CHR$(55 + N) + A$
70  IF N < 10 THEN LET A$ = CHR$(48 + N) + A$
80  LET A = (A - N) / 16
90  IF A > 0 THEN GOTO 50
100 PRINT A$
```

This program is an excellent example of the advantages of delving beneath the surface of Spectrum BASIC.

Sometimes the computer has to remember an address. This causes problems, because the address is sixteen bits long and a single byte is only eight bits long. The solution is to split the address into two, the bottom eight bits of the address (the low byte) being held in one byte and the top eight bits (the high byte) being held in the next byte. To find the address is a matter of calculating it with the formula:

```
address = (low byte) + 256*(high byte)
```

Talking directly to the memory

BASIC allows the user to be unaware of how the microcomputer works. This is usually advantageous, but occasionally better results are obtained, if the peculiar characteristics of the machine are exploited to the full. Usually this prevents a program from being transportable to a different microcomputer, but this is not in itself a sufficient excuse for avoiding it. After all, each new microcomputer soon has its own specific version of invaders written for it, and these are usually totally machine specific. Graphics are a particular example of the advantages of machine dependent programming, so a little time will be devoted to looking at Spectrum graphics from the microcomputer's viewpoint.

The Spectrum memory contains 65 536 locations each with its own address. The contents of any address (for example 65535) can be seen with the BASIC statement

```
PRINT PEEK 1000 (which gives the value 9)
```

New data can be sent to a particular memory location with the statement

```
POKE 1000,0
```


With this particular address there will be no effect as location 1000 is in ROM and its contents cannot be changed. Only RAM can be altered in this way. However, if you start changing RAM indiscriminately, you may upset the operating system of the microcomputer. Certain parts of RAM are reserved by the machine for its own use. If you change these the Spectrum may get lost inside itself. The screen may 'freeze' or go blank and the microcomputer may refuse to respond to the keyboard. Even the BREAK key may not work or may produce the situation where everything seems normal, but unexplained error messages appear. On listing your program you find that it has been changed and someone has written rubbish in parts of it.

None of this causes any permanent damage to the microcomputer. In computer jargon you have caused a crash. The remedy is very simple. Switch off the microcomputer, wait a few seconds and then switch on again. The proper operating system will be restored and all will be well. The only casualty will be that your program has disappeared. This is your own fault for not obeying the maxim:

ALWAYS SAVE A PROGRAM BEFORE YOU RUN IT.

This is particularly sound advice when running machine code programs, when writing directly to the memory or when external devices are connected to the microcomputer.

One very useful place to write is the screen memory. Certain parts of the memory hold the information that is displayed on the screen. This RAM can be read and written to without any fear of disaster. It also has the advantage that you can see what happens to the location. Let us try this now.

First clear the screen with CLS. Each dot on the screen is now the visible representation of a particular bit in the screen memory and can be turned on or off directly. For example, type

```
POKE 18000,1
```

A single dot should appear approximately in the middle of the screen near the top, because bit 0 of memory location 18000 has been turned on. Try

```
POKE 18000,16
```

to get a different dot. A good investigation now is to discover the positions of the dots corresponding to each bit. Try this program.

```
10  FOR i = 0 TO 255
20  POKE 18000,i
30  PAUSE 25
40  NEXT i
```

Line 30 is a delay to slow everything down. You should observe that combinations of the numbers 1, 2, 4, 8, 16, 32, 64 and 128 give different combinations of dots. In particular, 255 switches on all the bits and produces a line. Now try different addresses, such as

```
POKE 19000,255 or
POKE 20000,255
```

To find out where the different memory addresses are located on the screen, run this program:

```
10  FOR i = 16384 TO 22527
20  POKE i,255
30  PAUSE 25
40  NEXT i
```

You will soon discover one fact: the screen locations are not contiguous. That is, the end of one line is not followed immediately by the start of the next. Each block of thirty-two continuous bytes produces a line, but the next bytes produce a line eight positions further down. This continues until the top third of the screen is filled and then a start is made on the next third. This makes it more difficult to address the screen directly, but a method of doing this is shown in Chapter 7.

Spectrum graphics

The graphics facility of the microcomputer makes it the most powerful resource in education since books were invented. In this respect the ZX Spectrum is exceptional and will allow you to do just about everything you want. By the term 'graphics' is means not only pictures, but also drawings, enlarged figures and letters, graphs, bar charts, histograms and animations. Their use makes computer assisted learning more interesting, increases motivation and enhances retention. So how do we write BASIC programs with Spectrum graphics?

There are three different ways of producing pictures on the video screen: with the chunky graphics characters; with user-defined characters; or with the PLOT, DRAW and CIRCLE statements. The latter use the high resolution screen, meaning that any of its 49 152 dots (called pixels) can be individually switched on or off. We saw above how this can be done directly. The method is identical to that which will be used in Chapter 4 for switching LEDs on and off. Each bit at each address controls a single pixel, so any combination of dots could be produced anywhere on the screen by turning on the appropriate bits. You could theoretically paint a complete picture by specifying each individual dot but in practice this is time consuming and impracticable. BASIC has much simpler ways of doing it.

When using BASIC we can imagine the TV screen as a matrix of pixels (256 wide by 176 high), independently addressed by its x and y coordinates. For drawing lines and circles this is very useful and there are three functions to help: PLOT (which plots a single point); DRAW (which draws a straight or curved line); and CIRCLE (which draws a circle). These functions are all described in the Spectrum user guide, so there is no need to go over them again here. It should be

noted, though, that drawing pictures with these functions is still not easy, and you need to be quite good at coordinate geometry to do it well. A discussion of this is taken up in Chapter 3. A useful point to note here is the ease with which pictures can be transferred to paper, using the ZX printer and the COPY command. How many other microcomputer systems can dump the screen onto paper with just two keystrokes?

Chunky graphics

The other methods of producing pictures use graphics characters. This is similar to the way that text is written on the screen.

```
PRINT AT 5,10;"ABC"
```

places the letters ABC on line number 5 of the screen with the letter A starting at column number 10. There are 133 possible characters that can be printed on the screen in this way, listed in Appendix A of the user guide (with code numbers from 32 to 164). One of these characters is a blank or space, which might seem rather pointless, but in fact this character is invaluable (remember that arithmetic was very complicated until zero was invented).

For pictures the most useful of these are the graphics characters, coded 128 to 143 (page 92 of the user guide). They are obtained by first entering the graphics mode and then using the numeric keys 1 to 8 (unshifted and shifted). For example, in the graphics mode key 3 gives a row of upper-half bars. If a shifted 3 is used instead, the result is the lower-half bars (one is the reverse or INVERSE of the other – this applies to all the graphics characters).

The normal rules about PAPER and INK apply to the graphics symbols too. For example, by defining the PAPER to be RED (key 2) and the INK to be BLUE (key 1), it is possible to get an edge which is half red and half blue. If the directions about PAPER and INK are to be altered frequently within a line, it becomes easier to enter them more directly. The other advantage of doing it this way is that the final result becomes visible immediately – you don't have to wait until the program is run before you see what the picture will look like. The disadvantage is that editing a printout is made more difficult since their codes do not appear in the listing.

Pictures

The best way of using these graphics characters to draw a picture is to take a large sheet of squared paper and mark it out as a grid of thirty-two horizontal squares and twenty-two vertical squares. Then sketch the required picture and fill in each square with the graphics character that best fits it, making a note of the required colours (paper and ink) at the same time. The picture can then be built up from a set of PRINT statements.

There are some aspects of such pictures that are not pleasing. Diagonal edges are stepped in Lego fashion and fine details are not included. To allow for this the Spectrum lets the user define a set of his or her own graphics characters which

may be vertical, horizontal or sloping lines of triangles, etc. The technique of defining a shape is described in the Spectrum user guide (Chapter 14), where a set of chess pieces is defined.

The shape to be defined should be drawn on an eight by eight grid. As an example let us make a diamond character. Note which dots have to be coloured in and write down a 1 for each corresponding position. Where the dot is not coloured in, a 0 is written down. Each row is then represented by a binary number, which can be written down as BIN 00001000 say. All eight rows are similarly represented. Then a particular letter (from A to U) is defined by the USR function to be the chosen character.

```
100  POKE USR "A" + 0,BIN 00011000
110  POKE USR "A" + 1,BIN 00111100
120  POKE USR "A" + 2,BIN 01111110
130  POKE USR "A" + 3,BIN 11111111
140  POKE USR "A" + 4,BIN 11111111
150  POKE USR "A" + 5,BIN 01111110
160  POKE USR "A" + 6,BIN 00111100
170  POKE USR "A" + 7,BIN 00011000
```

The defined character may now be used. After the program is run, each graphics-A character will be a diamond. The above method of defining characters is good for those who do not know about the binary code, but most users find no difficulty in converting each of these binary numbers to its decimal equivalent with the table given below

Binary number	Decimal equivalent
00011000	16 + 8 = 24
00111100	32 + 16 + 8 = 60
01111110	64 + 32 + 16 + 8 + 4 + 2 = 126
11111111	= 255

It is then easier to define the diamond like this:

```
100  POKE USR "A"+0,24
101  POKE USR "A"+1,60
102  POKE USR "A"+2,126
103  POKE USR "A"+3,255
104  POKE USR "A"+4,255
105  POKE USR "A"+5,126
106  POKE USR "A"+6,60
107  POKE USR "A"+7,24
```

The ultimate step is to put these numbers into DATA statements as follows:

```
100    FOR n = 0 to 7
```



```
110  READ row
120  POKE USR "A" + n,row
130  NEXT n
200  DATA 24,60,126,255,255,126,60,24
```

which is very much shorter than the original method.

User-defined character sets

A useful task is to define a whole set of graphics characters. In general there are two such sets required: line characters and filled characters. The following programs produce each of these character sets. Although each character in the PRINT statements that make up a picture looks like a letter, this is before the program has been run. After this has happened, each letter is redefined, so that subsequent listing produces the graphics character instead.

The filled set

```
3000  REM filled graphics
3010  FOR i=0 TO 20
3020  FOR j=0 TO 7
3030  READ row
3040  PRINT USR CHR$ (i+144)+j,row
3050  NEXT j
3060  NEXT i
3080  RETURN
3100  DATA 0,0,0,0,0,0,0,0
3110  DATA 0,0,0,0,0,0,0,255
3120  DATA 0,0,0,0,0,0,255,255
3130  DATA 0,0,0,0,0,255,255,255
3140  DATA 0,0,0,0,255,255,255,255
3150  DATA 0,0,0,255,255,255,255,255
3160  DATA 0,0,255,255,255,255,255,255
3170  DATA 0,255,255,255,255,255,255,255
3180  DATA 255,255,255,255,255,255,255,255
3190  DATA 128,128,128,128,128,128,128,128
3200  DATA 192,192,192,192,192,192,192,192
3210  DATA 224,224,224,224,224,224,224,224
3220  DATA 240,240,240,240,240,240,240,240
3230  DATA 248,248,248,248,248,248,248,248
3240  DATA 252,252,252,252,252,252,252,252
3250  DATA 254,254,254,254,254,254,254,254
3260  DATA 255,254,252,248,240,224,192,128
3270  DATA 128,192,224,240,248,252,254,255
3280  DATA 1,3,7,15,31,63,127,255
3290  DATA 255,127,63,31,15,7,3,1
3300  DATA 24,60,126,255,255,126,60,24
```

The filled set consists of corner to corner 'sandwiches' and characters produced by having one row, two rows, three rows, etc. (or one column, two columns etc.). This set is most useful for horizontal and vertical bar charts, which are then more easily constructed than with high resolution graphics. A wavy surface to water is easily obtained by printing these graphics characters in a line thus:

```
200  PRINT at 21,0;"ABCDEFGHIIHGFEDCBA"
```

The line set

```
3000  REM line graphics
3010  FOR i=0 TO 20
3020  FOR j=0 TO 7
3030  READ row
3040  PRINT USR CHR$ (i+144)+j,row
3050  NEXT j
3060  NEXT i
3080  RETURN
3100  DATA 0,0,0,255,255,0,0,0
3110  DATA 16,16,16,255,255,16,16,16
3120  DATA 16,16,16,16,16,16;16,16
3130  DATA 16,16,16,31,31,0,0,0
3140  DATA 16,16,16,240,240,0,0,0
3150  DATA 0,0,0,31,31,16,16,16
3160  DATA 0,0,0,240,240,16,16,16
3170  DATA 16,16,16,255,255,0,0,0
3180  DATA 0,0,0,255,255,16,16,16
3190  DATA 16,16,16,31,31,16,16,16
3200  DATA 16,16,16,240,240,16,16,16
3210  DATA 128,128,128,255,255,128,128,128
3220  DATA 1,1,1,255,255,1,1,1
3230  DATA 255,0,0,0,0,0,0,0
3240  DATA 0,0,0,0,0,0,0,255
3250  DATA 128,128,128,128,128,128,128,128
3260  DATA 1,1,1,1,1,1,1,1
3270  DATA 255,1,1,1,1,1,1,1
3280  DATA 1,1,1,1,1,1,1,255
3290  DATA 255,128,128,128,128,128,128,128
3300  DATA 128,128,128,128,128,128,128,255
```

Since it is most likely to be used for line drawings, the use of colour is much less important in this set, black ink being the most favoured (Plate 6). The line set is used extensively in this book, for example in Z80 MICROPROCESSOR SIMULATION and INTEGRATED SCIENCE TEST.

By defining the characters in other ways it is possible to create almost any type of picture, the worst problem being that there is a maximum of twenty-one user-definable characters. Therefore use may also be made of some of the letters,

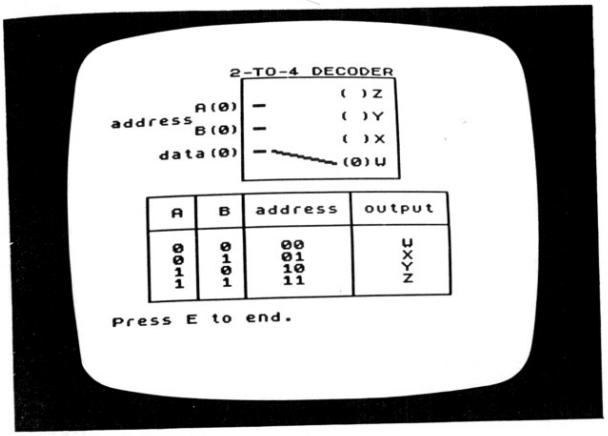


Plate 6 Using the line set

particularly o, 0, X, x and other symmetrical shapes, like * and +. The part of the program containing the character definitions should normally be placed at the end of the program, so that it can be merged with any other program that uses it.

Movement

One useful requirement is to make pictures that move around. The movement of individual characters around the screen is done by erasing the character from its old position by overprinting with a space. This is fairly easy with the 'PRINT AT line, column' statement, since the values of 'line' and 'column' can be changed each time.

```
200  for i = 0 TO 25
210  PRINT AT 20,i;" "
220  PRINT AT 20,i+1;A$
230  NEXT i
```

or

```
200  LET l = 8:LET c = 10
```

```
210 PRINT AT I, c ;" "
220 LET I = I +1:LET c = c + 1
230 PRINT AT I,c;A$
240 GOTO 210
```

In both cases line 210 erases the character from the old position before the position is changed. There ought also to be checks that the character is not moving off the edge of the screen, otherwise the program crashes.

For moving whole pictures it is not always necessary to rub out the whole picture from one place before reprinting it in the adjacent place. Horizontal motion is easiest since a space at the beginning of each line of the picture can be used to perform the rubbing out. Movement in other directions is generally a little slower, since the picture may have to be surrounded by spaces, to stop bits of it remaining behind when it is moved to the next place. In general, for moving more than one object or for moving large objects, BASIC is unacceptable. Smooth motion can only be achieved with machine-code programming.

Machine-code programs cannot, though, make use of the BASIC PRINT AT statement (at least not without slowing them down too much). We must therefore move characters around the screen by writing directly to the screen memory. I used to be unhappy that it was not possible to POKE characters onto the Spectrum screen directly. This would allow programs on molecular motion, the behaviour of molecules being simulated by the movement of certain characters around the screen. Such characters cannot be written directly onto the Spectrum screen (at least I haven't yet found out how), but there is a way round the problem. All the molecules can be printed on the screen beforehand in invisible ink (white-on-white). Each molecule can then be highlighted in turn, by turning the ink to black momentarily, which gives the appearance of motion. This is done by changing the ATTRIBUTES of each screen location with the appropriate data. The ATTRIBUTES for each screen position are held contiguously in memory from 22528 to 23295. For example:

```
POKE 22528,56
```

turns the ink of the top-left position to black, whereas

```
POKE 22528,63
```

turns it back to white.

There is no point in doing this in BASIC – the PRINT AT and PRINT OVER commands are fast enough. But the POKE command can be implemented in machine code, thus giving at least a hundredfold increase in the speed of the graphics, which allows a hundred molecules to be moved around at the same time and this is not possible in BASIC.

The ATTRIBUTES control the screen positions on a thirty-two (wide) by twenty-four (high) matrix. Their chief feature is that they are contiguous in the memory as follows:

	column no.	0	31
row 0	address	22528	22559
row 1		22560	22591
row 2		•	•
•		•	•
•		•	•
row 23		23264	23295

PRINT AT cannot access the lower two lines which BASIC uses for error and INPUT messages, but these are accessible with this direct method. If the address of the top-left corner is subtracted from each address, the pattern becomes clearer.

	column no.	0	31
row 0		0	31
row 1		32	63
row 2		64	95
•		•	•
•		•	•
row 23		736	767

It can be seen that the addition of thirty-two moves down one row. This makes for very easy programming of motion.

To begin with we shall just 'move' a black square around the screen to illustrate the principles. If an ATTRIBUTE is POKED with 0, it goes black, if POKED with 56 it returns to its original state (which is white since the screen is cleared when the program is RUN.) To make the black square move across the top of the screen 0 must be written into each successive ATTRIBUTE in turn, and then replaced by 56 again after a short delay to give it time to be observed.

```
10 FOR X = 22528 TO 22559
20 POKE X,0:REM PLACE BLACK SQUARE ON SCREEN
30 PAUSE 2:REM DELAY
40 POKE X,56:REM ERASE BLACK SQUARE
50 NEXT X
```

To move the character vertically 32 must be added to or subtracted from the current position.

```
10 FOR X = 22528 TO 23168 STEP 32
20 POKE X,0:REM PLACE BLACK SQUARE ON SCREEN
30 PAUSE 2:REM DELAY
40 POKE X,56:REM ERASE BLACK SQUARE
50 NEXT X
60 FOR X = 23168 TO 22528 STEP −32
70 POKE X,0:REM PLACE BLACK SQUARE ON SCREEN
```

```
 80  PAUSE 2:REM DELAY
 90  POKE X,56:REM ERASE BLACK SQUARE
100  NEXT X
110  GOTO 10
```

Diagonal motion is achieved by a combination of these two methods.

Value	Direction
1	east
33	south-east
32	south
31	south-west
−1	west
−33	north-west
−32	north
−31	north-east

```
 10  FOR X = 22528 TO 23189 STEP 33
 20  POKE X,0:REM PLACE BLACK SQUARE ON SCREEN
 30  PAUSE 2:REM DELAY
 40  POKE X,56:REM ERASE BLACK SQUARE
 50  NEXT X
 60  FOR X = 23189 TO 22528 STEP −33
 70  POKE X,0:REM PLACE BLACK SQUARE ON SCREEN
 80  PAUSE 2:REM DELAY
 90  POKE X,56:REM ERASE BLACK SQUARE
100  NEXT X
110  GOTO 10
```

More usually it is small pictures that are moved around the screen in this way (for example, the piston in the cylinder of a motor car engine). Very low resolution pictures, made from whole blocks of colour, can be moved about in the same way. The required coloured block is obtained by POKING nine times its numerical value into the required ATTRIBUTE position. This is because the INK colour is held by the lower three bits (decimal 9 to 7) and the PAPER colour by the next three bits (decimal 8 to 56). So by making PAPER and INK the same, the whole square becomes the required colour, irrespective of what is already on the screen at that position. The coloured block can be erased by restoring the original number into the ATTRIBUTE position. For example, 56 restores PAPER 7, INK 0. In this way coloured blocks may be moved around, passing in front of or behind any existing characters originally printed on the screen.

In the same way individual characters may be moved around by alternately making PAPER and INK different and then the same. The character must already be in position, by filling the required area of motion with the character. The following program fills the top third of the screen with *−characters.


```
 10  CLS
 20  FOR x=0 TO 31
 30  FOR y=0 TO 7
 40  PRINT AT y,x;"*"
 50  NEXT y
 60  NEXT x
```

By altering x and y this 'area' may be placed anywhere on the screen. In the BASIC molecular motion program ONEMOL (33) the molecules are constrained inside a box by this technique. The ATTRIBUTES of each location can then be changed to 63 (INK 7, PAPER 7) which effectively erases each *−character. By moving the number 56 (INK 0, PAPER 7) from one position to another, exactly as above, the molecule apparently moves around the screen, bouncing off the walls of the container.

Techniques of interaction

The most usual means of communication from the microcomputer to the user is the display. In this there are numerous pitfalls for those writing their own programs, which will now be described.

The display of text

The statement PRINT "PARIS IS THE CAPITAL OF FRANCE" is probably the most easily understood of all BASIC statements. The sentence is just written out on the video screen of the microcomputer. It is so easy to use, that some programmers fail to give any attention to the result.

The use of capitals (upper case) makes for difficult reading at the best of times, and if the programmer does not use double spacing either, it is doubly difficult to read. With lower case letters and the use of double spacing the result is more pleasant. The amount of text presented also needs to be adjusted to the level of the user: secondary pupils in particular merely scan the text without reading it properly. Later they complain that they 'don't know what to do'.

An automatic line feed occurs when there are thirty-two characters in a line. The thirty-third character appears on the next line and the crime of wrap-around is committed. There is no excuse for this, it simply requires the programmer to read what the program prints with a critical eye and not accept inferior presentation. If the same things were done on paper, they would be glaringly obvious. BASIC has the ability to display figures in neat columns, so there is no excuse for not doing so. This is described in the user guide.

In the days of tele-typewriter output there was no way to prevent text from scrolling up from the bottom. Part-sentences remained at the top of the screen,

and these were most distracting. There is no need to continue with this practice today. The programmer should clear the screen before each new page of text. Also less text should be displayed at one time, in which case the student will need to indicate when a new page of text is to be displayed. This is described later.

Input from the keyboard

Some published programs limit interaction to 'press SPACE' at the foot of each page of video text. This is a misuse of a powerful machine, especially if the opportunity to return to a previous page is denied. The microcomputer is more than an electronic page turner and its facility for interaction should be fully utilized. At the highest level, an interactive program could determine the level of understanding attained by its users and adjust the presentation to suit. At the lower levels, the interaction will probably be confined to responding to questions.

The INPUT statement

The simplest way of managing the response situation is via the INPUT statement. This needs careful handling, since the user can easily wait in vain for the microcomputer to reply, while it waits for the ENTER key to be pressed. Full instructions need to be given, especially to first time users. The first INPUT in a program might be to get the student to enter his or her own name, so that the microcomputer can appear more personal. Some instructions such as the following need to be displayed, not only on the screen itself, but also on any accompanying documentation.

```
Hello.

Please type in your first name.
If you make a mistake, you can
rub it out with the DELETE key
(top-right of the keyboard).
Press DELETE at the same time
as the CAPS-SHIFT key.

When you have typed your name
correctly, press the ENTER
key to say you have finished.
```

Note the double-spacing between paragraphs, the use of lower case text and the use of capitals for emphasis. As mentioned above, the text should be preceded by CLS to give a clean presentation.

The PRINT statements that produce this text are followed by the INPUT statement. In this case, a string response is required, that is INPUT a$, but this can

cause problems. The student, who presses ENTER before typing anything, causes a$ to be set to the null string and subsequent PRINT a$ statements produce no name. To avoid this, the INPUT statement should be followed by a check thus:

```
120  INPUT a$:IF a$=" " THEN GO TO 120
```

Because an alphabetic INPUT requires quotes around it, these are automatically supplied by the Spectrum. Their presence may, however be disconcerting to the novice user and it may be better to use the alternative form

```
INPUT LINE a$
```

which assumes the presence of the quotes without printing them.

The Spectrum is unlike most other microcomputers in that text input from the keyboard using the INPUT statement appears at the bottom of the screen. It is often necessary to include text in the INPUT statement to reassure the student that he or she is doing the right thing. This is described on page 104 of the user guide. Several examples of its use will be seen later.

The INKEY$ statement

This difficulty, of having to write at the bottom of the screen, and the problems of users having to remember to press ENTER, results in many programs studiously avoiding the INPUT statement, preferring the INKEY$ statement. This retrieves a single key entry, which may be any character on the keyboard and places it in the temporary variable INKEY$. The most obvious use of INKEY$ is to tell the microcomputer to turn to the next page. INKEY$ takes a quick look at the keyboard to collect the current key being pressed. If no key is being pressed, the program simply carries on.

```
100  LET A$=INKEY$:IF A$=" " THEN GO TO 100
```

causes the program to halt until one of the keys is pressed. An alternative is to use the PAUSE statement as follows:

```
100  PAUSE 0
110  LET A$=INKEY$
```

PAUSE is always interrupted by a keypress. This is useful if it is desired to give the student a limited time in which to respond. If the student has failed to answer within say twenty seconds, the program could jump to a remedial loop. The time delay is adjusted by altering the number in the PAUSE statement (the maximum is twenty minutes).

```
100  PAUSE 1000
110  LET A$=INKEY$
120  IF A$=" " THEN GO TO. . .(remedial loop)
```

An alternative wait routine, which is not interrupted by a keypress is

```
100  FOR T = 1 TO 8000:NEXT T
```

which produces a delay of several seconds and may be used to give a user time to read the text before presenting the next page. While this is adequate for single words or sentences, readers differ so markedly in their speed that no common time can be fixed for them all. The alternative technique, using PAUSE 0, is more satisfactory. The student is requested to 'hit a key', or better, to 'press SPACE to continue'. The SPACE can be detected with the BASIC statements

```
100  IF INKEY$<>" " THEN TO TO 100
```

This has the advantage that pressing a different key has no effect.

A common use of INKEY$ is to 'select from the menu'. The user is offered several alternatives and invited to choose one. A typical menu in a drill tutorial might look like this:

```
You are correct, the shutter
speed must be as fast
as possible, i.e. 1/1000th
of a second.

What would you like to do now?

1  Try another problem
   on shutter speeds?
2  Try a problem on apertures?
3  Go on to study film speeds?
4  Finish the lesson for now?

Press one of these numbers
to make your choice
```

In response to INKEY$ the Spectrum accepts a null string from the keyboard when no key is pressed and this is programmed out as follows:

```
110  LET a$=INKEY$:IF a$=" " THEN GO TO 110
120  IF VAL a$ <1 OR VAL a$ >4 THEN GO TO 110
```

A numerical key is required, but a student who presses an alphabetic key by mistake produces an error. For this reason VAL should only be used once it has been determined that the entry is correct.

```
110  LET a$=INKEY$
120  IF a$<>"1" AND a$ <>"2" AND a$<>"3" AND a$<>"4"THEN GO TO 110
130  LET number = VAL a$
140  GO TO (5000 + number*100)
```

The last line carries out the function of


```
140  ON number GOTO 5100, 5200, 5300, 5400
```

which is available on some other microcomputers.

In some situations an alphabetic key entry is preferred to a numeric one, even though this can cause more problems. For example, the user may be asked to select one response from a set of three or more possible correct answers to a question. These possible responses have to be labelled A, B, C etc. since numbers are probably being used for the questions themselves. The student's guess may be obtained with

```
100  LET a$=INKEY$
110  IF a$<>"a" AND a$<>"b" AND a$<>"c" THEN GO TO 100
```

The problems occur when the choice is given in upper case letters. The statement above is not obeyed if the student presses CAPS A to get the required entry accepted. This problem is overcome on the Spectrum with the statement

```
LET z = CODE INKEY$
```

which places the ASCII value of the pressed key in z (or 0 if no key is being pressed). The lower case ASCII value of any key is its upper case ASCII value plus 32. The following routine can thus be used to ignore the differences between upper and lower case:

```
100  PRINT "Press A or B or C"
110  LET z = CODE INKEY$
120  IF z = 0 THEN GO TO 110
130  IF z < 97 THEN LET z = z + 32
140  LET a$ = CHR$ z
150  IF a$ <> "a" AND a$ <> "b" AND a$ <> "c" THEN GO TO 110
```

Alternatively the ASCII key values may be handled directly thus:

```
140  IF z<97 AND z>99 THEN GO TO 110
```

These ASCII values are given on pages 183 to 185 of the Spectrum user guide. Their use to prevent novice users from crashing programs is recommended.

Whole words can be entered with INKEY$, one letter at a time, and the word can be assembled from these letters. This avoids the problems of having to use the ENTER key, but the possibility of erasing an error is then also removed (and has to be programmed back in). The advantage of this clumsy method of entry is that text can then be entered from the keyboard and printed at any part of the screen. An example of this is MASTERMIND (4).

INKEY$ has the particular advantage that it is continuous (unlike the GET of most other microcomputers). This enables something to occur for as long as a particular key is pressed, useful for games and for organ keys.

```
100  IF INKEY$ = "z" THEN BEEP .1,0
110  IF INKEY$ = "x" THEN BEEP .1,2
```

```
120  IF INKEY$ = "C" THEN BEEP.1,4
130  GO TO 100
```

If you think this will make a good electronic organ, you will be disappointed. BASIC is too slow and machine code routines are really needed.

Other techniques

Novices take ages to find a particular key on the keyboard. One way to overcome this is to use alternative methods of input. These also remove the need for disabling keys and all the other problems encountered above. The best of these devices is a light pen which can be pointed at a particular part of the screen. These are available commercially and plug directly into the back of the microcomputer.

For some responses, switches can be connected to the Spectrum through an interface (see Chapter 4) and detected with fairly simple routines. An alternative for the future is the soft or concept keyboard, which plugs into the microcomputer, and where the number and function of the keys can be changed by the program itself. The keys can thus become letters, numbers, pictures or special symbolic characters as in BLISS. This is a far better way of communication with younger pupils, avoiding all the above pitfalls and giving more freedom to the programmer.

Crash protection

Ideally it should be impossible for a novice user to crash a program by indiscriminately pressing the wrong keys. This can be such an effort (as the above discussion shows) that it may take too much time. The best way is to put key entry checks into a separate subroutine, which already contains the protection. This can then be called whenever it is needed. Even then a determined pupil can crash by pressing the BREAK key. My solution is to teach pupils to be careful and not to press all the keys in sight. The display should tell them exactly which keys to press and if they press others, then they can jolly well find out how to recover from the crash themselves. (Actually it is quite amazing how quickly even young children can learn to use the machines properly; there is such a thing as over-protection.)

Processing the response

Once the response of the student has been collected, the microcomputer has to process it. If the entry is the student's name, presumably this is so that a personal touch can be added to requests:

```
Now, Bob,
can you tell me . . . . .
```

This is achieved by printing out the string variable that was used for the original input. That variable name must not be used again, or the microcomputer will later change the student's name to PHOTOSYNTHESIS or whatever. Note also the

need to leave a long space after the student's name. If this is not done, you will find the computer responding to a long name with:

```
Now, Stephanovitchi, can you tel
l me . . . . . .
```

Wrap-around is unforgiveable in video text.

The response PHOTOSYNTHESIS might be the answer to a question set by the microcomputer. Once this response has been collected, the program has to decide if PHOTOSYNTHESIS is the correct answer. A sequential list of questions can retain the correct response in a DATA statement, which is then collected by READ. If responses have to be accessed at random, then a better way is to keep the correct responses in a string array, thus:

```
100  LET R$(1)="PHOTOSYNTHESIS"
110  LET R$(2)="RESPIRATION"
120  LET R$(3)= . . . . .etc.
500  PRINT "What name is given to . . . .
510  INPUT A$
520  IF A$ = R$(1) THEN PRINT "CORRECT"
530  etc.
```

The unfortunate thing about checking responses by the method shown in line 520 above, is that misspelled inputs or even things like PHOTO-SYNTHESIS are considered incorrect. The program could contain a selection of possible responses and check each one separately, but the range of possible correct responses could be enormous.

One solution is to use the string slicing functions to check that the majority of a word is correct, but every word tends to behave differently and about the best that can be achieved is to disregard leading spaces and hyphens. The problem mentioned above, about the use of upper and lower case letters, can be overcome by the use of the CODE and CHR$ functions.

One desirable feature of tutorials is to give clues if the student has no idea. In the case above, after the first wrong response, the microcomputer could prompt with

```
CLUE: PHOTO————
```

w$(1 to 5) is used to extract the initial letters, and this can be printed out on top of

```
FOR I=1 TO LEN(w$(1)):PRINT"—";:NEXT I
```

ELEMENTS (25) demonstrates the way that this is achieved in practice.

Techniques like these are learned by studying the user guide, the programs of others and books specifically about BASIC and the Spectrum microcomputer (a list of such books is given in the Appendix). The most essential requirement is self-criticism.

Writing a program

This topic is a subject in its own right and at least one book has been entirely devoted to it. Thus it is not possible to do more than indicate the overall principles. The whole process of writing a program can be subdivided into three parts:

Design
Coding
Debugging

Of these the most important, and the one most often neglected, is the design stage. There is always a great urge to begin coding, that is, to write BASIC statements into the microcomputer. This should be resisted as long as possible, because the faster one begins coding, the poorer the program will be.

Examples of this abound. Often a program is started with one plan in mind and then extra features are added as they occur to the programmer. Then it is found that certain things have been overlooked and the program needs extra routines fitted in. Often there will not be enough room for these, so the programmer resorts to using subroutines, where their use is unjustified. It soon becomes more and more difficult to deal with new problems and the project is abandoned (or, worse still, full of bugs and impossible to interpret, sold to unsuspecting teachers).

This is what can happen if the planning stage is neglected. What I have just described is called **bottom up programming** – starting from a simple idea and adding refinements to it. Programs should be designed as a whole from the start and the problems that might arise should be anticipated. This is called **top down programming** and is what the rest of this chapter is about. I do, however, want to give a note of caution.

It often happens that programs are developed by chance. For example my first (PET) programs on wave motion were the result of an accident. I had spent some time trying to make waves that moved across the screen, but BASIC was much too slow. Then, while working on a routine to paint a picture on the screen in machine code, I assumed that the end of the screen was in position 40 (forgetting that it runs from 0 to 39). The routine painted the picture quite happily, which then scrolled across the screen. I realized that a sine curve would become a travelling wave and the solution to my problem had been overcome. I was able to use this accidental discovery to write several wave programs for the PET.

The point of this story is that planning by itself does not always produce a program. There nearly always has to be interaction between experimentation and program development. In the commercial world, program designers must specify accurately what they want to do. Poorly constructed programs cost money, so top down programming is an economic necessity. The educational world is not quite the same as this. Teachers are almost certainly writing programs in their own time, which is never costed. More importantly, they do not have all the necessary programming skills at their fingertips beforehand. For them, strict top down

programming is not possible until they become more expert.

I shall therefore describe a technique that can be used by non-experts. To aid the discussion we shall look closely at one particular program BYTE (28), which is listed in the Appendix. This is not a program merely developed to illustrate the principles, but a genuine one. Thus it gives a better insight into the whole process of program developments than any artificial example can provide. It also utilizes graphics and illustrates most of the interaction techniques discussed in this chapter.

I wanted a program to simulate what happens to a byte of memory, when certain operations are carried out on it. These operations are left and right shifting, addition and subtraction. I also wanted to demonstrate the connection between binary and decimal numbers. I had already seen the principle demonstrated in the BBC television programme Making the Most of the Micro, so I had a good idea of what I wanted to achieve.

Design

The program would display a diagram of a byte of memory, with a statement of what decimal number it contained. Inside the byte would be the binary number, displayed as a series of 1s and 0s. the user would be asked to choose from the options described above. If a number was required, this would be entered and the display changed accordingly.

This specification immediately threw up problems. What measure of protection was required? Considering the probable ability range of the users (secondary), I did not consider that too much protection was necessary. I decided to trap the obvious mistakes, such as numbers out of range and numbers instead of letters. Further protection could be added after the evaluation stage, if it became clear that it was needed.

When we have decided what we want to achieve, it is time to start top down programming. We do not go straight to the computer and start programming, that state is still some way off. We begin by writing the program on paper in **pseudo code**; meaningful statements that can later be turned into BASIC statements (or indeed any other language). For this code we recognize three distinct processes:

Sequence
Repetition
Choice

A sequence is a set of instructions that follow one another in strict order. TRAFFIC LIGHTS in Chapter 4 is a good example of this.

```
Turn on red traffic light
Long delay
Turn on red and amber traffic lights
Short delay
Turn on green traffic light
Long delay
```

Turn on amber traffic light
Short delay

Choice is achieved by IF...THEN...ELSE. The sequence branches into two or more separate routes depending upon the conditions encountered initially.

Repetition is similarly obvious, but here there are different kinds. The traffic lights sequence may need to be repeated forever. This can be achieved by a GO TO back to the beginning. A pelican crossing has the green traffic light on until a pedestrian requests the traffic to stop. This can be achieved by a WHILE...DO structure:

WHILE the pedestrian is not requesting traffic to stop,
DO keep the green traffic light on.

A pedestrian crossing at crossroads may be incorporated into the traffic lights sequence itself, but this is wasteful since it makes traffic wait when there are no pedestrians. It is better if the pedestrian request switch interrupts the normal sequence to make it behave differently. The normal sequence is repeated until an event occurs to change it – the REPEAT...UNTIL structure. Finally, it may be necessary to repeat some sequences a given number of times. This uses the well-known FOR...NEXT structure.

In none of these processes are we concerned with BASIC – exactly how we implement this pseudo code is irrelevant. Spectrum BASIC does not, in fact, recognize all of them anyway, so some have to be specially constructed. 'WHILE condition DO loop' is carried out by 'IF condition THEN GO TO start of loop'. REPEAT...UNTIL is implemented by a similar construction. For our purposes at the moment, it is the process that is important, not how it is later turned into BASIC.

One way of designing a program (long taught in schools) is flowcharting. This has sequences (rectangular boxes), choices (diamond boxes) and repetitions (returning lines and junction boxes). To introduce the ideas of design, flowcharting is a good method, but it is not popular with serious programmers. Programs of any size spill over onto several sheets of paper and are difficult to follow. Also it is not easy to plan a flowchart until all its limbs are known. This results in the same chart being endlessly redrawn to accommodate extra requirements. Most programmers draw the flowchart after the program has been written!

Top down programming allows the program to be developed from the general plan right down to the level of coding in BASIC by a process known as **stepwise refinement**. This cuts out a great deal of the redrawing (or rewriting in this case) of those elements that are already known. It also allows each step to be checked for error before it is turned into code. In this way any bugs in the final program will only require simple patches, not wholesale rewriting. Now that we have an overall strategy for our program, let us begin this process.

BYTE
1 Initialize variables etc.
2 Draw boxes for byte
3A Display options
3B REPEAT Collect option
3C Execute option
3D UNTIL option is 'finish'
4 Finish with program.

The structure of the program is becoming obvious. 1 and 2 are sequential and are executed once each time the program is run. 3 is executed repetitively until the program is halted by some means. Within this REPEAT...UNTIL loop, there may well be other nested loops, each of which is terminated by a different condition. The question raised now is, where to go next. As a rule one should stick to the order of execution unless there are some processes that are not yet clearly defined. These should be tackled first, because they may throw up problems that cause the original design to be modified. The earlier such modification takes place, the better. In our case we have to ask about 3C.

3C carries out the operation on the number in the byte. While it does so the options are removed. Our program has made no allowance for recalling them, so immediately there is a fault in the design. But it is far better to discover this now than later. Our first revision is thus:

BYTE
1 Initialize variables etc.
2 Draw boxes for byte
3A REPEAT display options
3B Collect option
3C Execute option
3D UNTIL option is 'finish'
4 Finish with program.

Now let us look more closely at what 3C is required to do. There will be six possibilities, depending on the option chosen:

P POKE a number into the byte
A ADD a number to the byte
S SUBTRACT a number from the byte
L LEFT SHIFT
R RIGHT SHIFT
F Finish

Each of these will be a subroutine, returning control to the main program once executed. Our refinement of this step will be as follows:

```
3C   Execute option
3CP      IF option = P THEN subroutine P
3CA      IF option = A THEN subroutine A
3CS      IF option = S THEN subroutine S
3CL      IF option = L THEN subroutine L
3CR      IF option = R THEN subroutine R
3CF      IF option = F THEN subroutine F
```

Step 3B can also be modified further. It consists of collecting the option from the user, probably with a single key entry. It becomes clear that collecting an option will follow immediately after displaying the options, so the two should go together as a single subroutine. This could also contain a routine to remove the choice of options immediately before executing them. So 3A and 3B become:

```
(S1)    Subroutine to handle options
(S1)A   Display options
(S1)    REPEAT
(S1)B       collect option
(S1)C   UNTIL option is acceptable
(S1)D   Point to chosen option
(S1)E   Clear choice of options from screen
(S1)F   RETURN to calling routine
```

Refining this further gives:

```
(S1)B   Collect single key entry
(S1)C   IF key is not one of P, A, S, L, R or F
        THEN collect another key
        ELSE continue
```

Each of these can be instantly turned into code, since they are all straightforward. We can therefore leave further development of this until later. We ought first to check through, that nothing has been overlooked. To do this we carry out a dry run with dummy data. Imagine we can see the options displayed, something like:

```
Choose one from P, A, S, L, R or F.
```

If the user presses Q, the routine will not accept it, but will instead wait until an acceptable response is made. At this point I ask myself whether I ought to build in some protection. Should I indicate that the response was not acceptable? On the other hand, the instructions are explicit. If a user enters Q and gets no response, then he or she has not obeyed the instructions. Should that user expect to get any more information? I decide that this is unreasonable, so no action is to be taken.

After this simple test, we need to see if there is anything else that is not yet obvious. It is clear that we have still to do a great deal with the options.

Considering each option in turn shows that options P, A and S require a number to be entered from the keyboard. In each case this number must be an integer between 0 and 255. These options ought, therefore, to share a common subroutine.

In this subroutine, it will not be possible to collect the number with a single keystroke, it will have to be an INPUT statement. So this gives us:

```
(S2)    Subroutine to collect a number
(S2)A       Display range of acceptable inputs
(S2)B       REPEAT
              collect input number
              convert to integer
            UNTIL number is positive and less than 256
(S2)C       RETURN to calling routine
```

The whole structure can be searched and refined further in just the same way until it all ends up as simple statements, each of which can be converted into code without problems. Before coding, though, it is necessary to check that all the likely problems have been foreseen and allowed for. The programmer should make a dummy run through the program with imaginary data to see what will happen, as we did above. Such a dry run through the program might reveal several more problems to be overcome. For example, how will (S2)B cope with an alphabetic input? Having discovered such problems, their solutions must be built into the program at this planning stage.

This is the theory! In practice the strict pattern of top down programming breaks down whenever a problem is encountered, for which the programmer can see no solution. For example, I needed to know how to split the decimal number up into a binary number. This is not something I had ever done before, so I could not see how to do it. This is where the advice of computer scientists has to be ignored; no amount of stepwise refinement will tell me how to do this, only experimentation, that is bottom up programming. I used to feel guilty at ignoring the advice of expert computer scientists, until I realized that they are dealing with different problems. They already know how to handle their machines, so they do not need to break off to find out. I have not yet reached this stage and I am sure that few other science teachers have either. The problem with bottom up programming is the restrictions it might impose on later top down refinements. It is advisable to discard any code created during the experiment – its retention might force the programmer into a predetermined mould and lead to later problems.

It is difficult to follow this advice because of lack of time. In my particular case, I could not display the binary number without the boxes to put the results into. Checking this meant drawing the boxes and this, in turn, meant developing the graphics characters. I decided to put the latter well out of the way, at line 9000 onwards. I then set up a routine to draw the boxes (line 1000 onwards) and finally the routine to turn the decimal number into binary and display it (starting at line

1500). I had several attempts at this (which explains the missing line numbers 1630 to 1690), before hitting on one that was acceptably fast.

All this meant that a good deal of the programming had already been done, before the final structure was determined. Having developed some code that works, I tend to want to keep it. Any major drawbacks, likely to be created by beginning the coding early like this, are best overcome by writing it as separate subroutines. It is then much easier to merge this with the main program later, or to throw it away if it does not, in the end, work satisfactorily. Either way, it does not become a millstone that forces subsequent planning into a predetermined mould. In fact, such problems did occur. The subroutine to split up the number is called repeatedly. The subroutine to draw the boxes is called once. By not rewriting this part of the program already coded, I upset the natural order of the subroutines. This feature remains. Of course I could have renumbered to get rid of it, but that would have been cheating. That is not the way that it happened.

Coding

Having refined each process until we are sure how to do it (or experimenting with the bits we are not sure about), we are now in a position to begin turning it into a BASIC program. I did this linearly from the beginning. With the fundamental structure developed, this was quite an easy job. Some problems were encountered and needed ad hoc solutions (see later), but the structure remained intact throughout. Even so, a structure alone does not necessarily lead to a readable program. There are some ground rules for structured programming that should be borne in mind.

One oft-quoted rule is 'avoid GOTO and GOSUB'. I agree with this up to a point. Some programs are such a mass of convoluted GOSUBs and GOTOs that it is impossible to see what different conditions are doing. Nevertheless, the intention is to make the structure of the program as obvious as possible. Contrived techniques for avoiding GOTOs may not necessarily achieve this end. Structured BASICs are certainly easier to use, but even Spectrum BASIC is not hopeless in this respect. The ability to have several statements following an IF . . .THEN means that several GO TOs can be avoided (unlike the ZX81). Because Spectrum BASIC allows the number in a GO TO statement to be a variable, it becomes fairly easy to perform something akin to a procedure. Study lines 2290 to 2350 in conjunction with line 280 and you will see what I mean.

Many novice programmers fail to use REM statements. It is true that they double the length of a program and take up memory space, which is important if you only have 1K of it! The Spectrum user has no such excuse. Although REMs take time initially, this is amply repaid later. If you can easily find each routine, debugging is much faster. This is even truer if the debugging is separated from the coding by more than a few days. In our case, speed was only important when printing the binary number in the boxes, so REM statements could be pared to a minimum there. Elsewhere I was able to be very liberal with REM statements, using them to mark off the different sections and to explain what each was doing.

Another help in this respect is the Spectrum facility for using long variable names. These do make life so much easier. My one regret about the Spectrum is, that this facility does not extend to arrays and the counting variable in a FOR. . .NEXT loop.

Debugging

As mentioned above, correcting any errors in the program is not something that can be left until last. Each step should be checked with dummy data to ensure that nothing has been overlooked. Even so there will be errors in the program once it has been coded. Simplest to eliminate are syntax errors (or mistakes) since BASIC contains error detection routines and obligingly tells the programmer where the error has occurred. More difficult to determine are errors in the logic. Hopefully these should not exist, but that is a counsel of perfection. In my case several such problems arose, which were detected as soon as the program was first run.

For example, I wanted to show users what was happening to their option after they had made their choice. In all cases except POKE, the delays incurred in calculating meant that enough time was given to this, without any more being needed. The POKE routine turned out to be much faster, so line 3095 was added later to overcome this fault.

I intended to display each option choice on one line, allowing the program to ask for the number immediately following. The total length of line needed to do this was too great in the case of the POKE statement, so some extra coding was added (line 3093), to allow this to be written on two lines. I should have noticed this at the planning stage, but it was, unfortunately, one that I failed to observe.

After a program has been debugged by the programmer, it should be let loose on users. This stage is about to be reached. The full listing of the program is in the Appendix. Doubtless it contains further bugs, but in the time-honoured method of all lecturers. I leave them as an exercise for the student.

3 Computation and mathematical modelling

'She can't do sums a bit!' the Queens said
together, with great emphasis.
(Lewis Carroll, *Through the Looking Glass)*

This chapter explores the uses of the Spectrum as a mathematical tool, including calculations, graphical display of functions, plotting experimental data, simulations using the random number generator and problem solving by iterative methods.

The super calculator

Calculation is the traditional domain of the computer (as its name implies). There are many books that deal exhaustively with this aspect of computing, with many illustrative examples. In fact, there may even be too many! Why do so many books of programs include one on the solution of quadratic equations? It is not because there are many problems that require its solution, in fact, hardly anyone uses it after leaving school. I suspect the real reason is that it has become a standard example upon which mathematical programmers cut their teeth (while physicists do radioactive decay and the rest write programs on sorting). The real value of writing such programs is the insight they give the programmer into the nature of the problem. Try writing your own quadratic equations program and you will see what I mean. How do you interpret 'too big' or 'syntax error'? Perhaps you forgot about equal or imaginary roots. If this is true, then one way to teach students about LCR circuits might be to get them to write their own LCR circuit analysis program.

There is no point in using a computer just to substitute numbers into equations. Nor do we want students to enter a set of data into some previously prepared program on, say, Newton's rings, that then automatically calculates the wavelength of sodium light. In both cases the process is more important than the product – we are trying to get students to appreciate the properties of the equations being used.

The microcomputer can aid this understanding of equations and concepts in two ways. One of these, the iterative method, is left until last. The other is the sledgehammer technique of getting the computer to solve an equation many times over while varying one of the parameters. As an example, consider the motion of a stone being thrown vertically against gravity (GRAVITY, program 35). By entering different starting speeds a pupil should be able to discover the relation between the vertical height reached and the initial speed. This technique may be

used with almost any other standard equation in science. It would be much better though if the graphics capabilities of the microcomputer were used as well.

Producing a neat table of results is not easy with the Spectrum, since it has no formatting statements. GRAVITY shows what can be achieved with the PRINT AT statement. Small numbers cause problems since they are converted to scientific notation, which overflows the space available. Such numbers may be produced in lieu of an expected zero result, since microcomputers are calculating in binary and decimal numbers do not always convert to an exact binary equivalent. A calculation that should produce zero, such as (3↑2–9) might well produce an answer like 3.7252903E–9 rather than 0. The subroutine at line 1000 of GRAVITY shows how such numbers can be ignored, while proper decimals are converted to four digits (or five if the number is negative) before being printed.

Graph plotting

The high resolution graphics of the Spectrum are particularly useful for sketching functions. PLOT and DRAW are easily used and some very sophisticated graphs can be drawn. The process is a little slow for complex functions, but this is not necessarily a disadvantage. One can ask the students to predict 'what will happen next?'. For those whose coordinate geometry is a little rusty, the following discussion may be of assistance.

PLOT

The statement to plot a single dot is

```
PLOT 0,88
```

You may just be able to see the small dot on the left of the screen and half-way up, which is the point you have just plotted. Now type

```
PLOT 10,88
```

which gives a point nearer to the right, but at the same height as the other point. The first number in the PLOT command tells how far the point is from the left edge. Type

```
PLOT 10,40
```

to get a point below the ones plotted before. This shows that the second number in the PLOT command gives the vertical position of the point. The smaller the number, the nearer it is to the bottom. The largest value for the horizontal position is 255 (extreme right) and the smallest is 0 (extreme left). The largest value for the vertical position is 175 (top) and the smallest is 0 (bottom). Any attempt to plot points outside these limits cause a fatal error (the program crashes).

Clear the screen with CLS and prove for yourself the positions of the extreme corners of the screen as follows:

```
TOP-LEFT      : PLOT 0,175
TOP-RIGHT     : PLOT 255,175
BOTTOM-LEFT   : PLOT 0,0
BOTTOM-RIGHT  : PLOT 255,0
```

Lines

We get lines by drawing a set of dots close together using the DRAW statement. This draws a line from the previous point visited (PLOT or a previous DRAW) with the new dimensions specified in the DRAW statement. For example:

```
MOVE 0,0
DRAW 100,100    (diagonal line)
DRAW 50,0       (horizontal line)
DRAW 0,50       (vertical line)
```

This shows that the new line is drawn starting from the last point visited. DRAW a,b does not draw to the point a,b, but a distance a in the x direction and a distance b in the y direction. It is called a relative draw. To get an absolute draw, the coordinates of the previous point visited have to be subtracted from a and b beforehand. This can be most easily achieved by

```
DRAW a - PEEK 23677,b - PEEK 23678
```

This technique works because locations 23677 and 23678 are where the Spectrum keeps a record of the x and y coordinates of the last point visited. Using this method, the points on the screen have the coordinates x,y (as in coordinate geometry) and lines can thus be drawn to an absolute position. To plot graphs there must be some relationship between x and y, which must be included in the program. Here is a simple example:

```
100  PAPER 7
110  INK 0
120  FOR x = 0 TO 255
130  LET y = x/2
140  DRAW x - PEEK 23677,y - PEEK 23678
150  NEXT x
```

Note how the program plots the equation given in line 130. Any equation connecting x and y can be used, provided the equation is of the form y = function of x only. If, however, the first point plotted is not 0,0, the first DRAW statement produces an unwanted line from the bottom left corner of the screen. This can be overcome with a flag, which is set after the first point has been plotted. If the flag is set, a DRAW is executed and if the flag is cleared, then a PLOT is executed. The subroutine to do this (and also to avoid out-of-range errors) is shown below.

```
100  PAPER 7
110  INK 0
```



```
115  LET flag = 0
120  FOR x = 0 TO 255
130  LET y = 200 - x/2
140  GOSUB 1000
150  NEXT x
160  STOP
1000 REM Draw a line to x,y
1010 IF x<0 OR x>255 THEN LET flag=0:RETURN
1020 IF y<0 OR y>175 THEN LET flag=0:RETURN
1030 IF flag=0 THEN PLOT x,y:LET flag=1
1040 DRAW x - PEEK 23677,y - PEEK 23678
1050 RETURN
```

Try this for yourself, with different equations in line 130. For example:

```
130  LET y = x*x/200
130  LET y = 250 - 3*x + x*x/100
```

Different origins

Axes can also be drawn as follows:

```
10  REM Draw axes
20  PLOT 0,0:DRAW 255,0
30  PLOT 0,0:DRAW 0,175
```

This only allows us to plot graphs in one quadrant, for positive values of x and y. Some graphs, particularly sines and cosines, produce negative values too. To plot these requires us to draw the axes in a different place. To keep the origin of the x axis at the left of the screen (x = o) and put the y axis in the middle (y = 88) we write

```
10  REM Draw axes
20  PLOT 0,88:DRAW 255,0
30  PLOT 0,0:DRAW 0,175
```

The graph will now show points in the range 0 to 255 (x coordinate) as before, but −88 to +87 (y coordinate). Now to plot lines with this new origin requires the addition of a displacement to the y value with

```
PLOT x,88+y
```

Another problem with sine and cosine graphs is that they are functions of angles in radians. To get at least two cycles on the screen, the range for the angle must be from 0 to 4*PI radians. The range for x is 0 to 255, so a conversion factor has to be included to make 256 equivalent to 4*PI. It is better to define a conversion factor (confac) to carry out this operation at the start of the program and to do this in such a way that it is obvious what is happening.

```
LET cycles = 2
LET confac = 2 * PI * cycles / 256
```

The value of any sine function goes from −1 to +1, so it must be multiplied by an amplitude (maximum of 87 to get the full range on the vertical axis). Here is the program for the sine function. This shows the way of handling the new origin.

```
 10  REM Draw axes
 20  PLOT 0,88:DRAW 255,0
 30  PLOT 0,0:DRAW 0,175
100  PAPER 7
110   INK 0
115  LET flag = 0
200  LET cycles = 2
210  LET confac = 2*PI*cycles/256
220  LET amplitude = 60
230  FOR x = 0 TO 255
240  LET y = amplitude*SIN (x*confac)
250  GOSUB 1000
260  NEXT x
300  STOP
1000 REM Draw a line to x,y
1010 LET a = x
1020 LET b = 88 + y
1030 IF a<0 OR a>255 THEN LET flag=0:RETURN
1040 IF b<0 OR b>175 THEN LET flag=0:RETURN
1050 IF flag=0 THEN PLOT a,b:LET flag=1
1060 DRAW a - PEEK 23677,b - PEEK 23678
1070 RETURN
```

A program to plot the cosine function involves changing line 240 to

```
240 y = amplitude * COS (x*confac)
```

A program to plot two functions at the same time requires two FOR-NEXT loops. Let us plot three cycles of the sine function and two of the cosine function. The use of DRAW within a single loop now becomes awkward and it is better to use two separate loops. This also allows the two curves to be drawn in different colours.

```
 10  REM Draw axes
 20  PLOT 0,88:DRAW 255,0
 30  PLOT 0,0:DRAW 0,175
100  PAPER 7
110  INK 1
115  LET flag=0
```



```
200  LET sincycles = 3
210  LET sinconfac = 2*PI*sincycles/256
220  LET sinamplitude = 60
230  FOR x = 0 TO 255
240  LET y = sinamplitude*SIN (x*sinconfac)
250  GOSUB 1000
260  NEXT x
300  INK 3
310  LET flag=0
320  LET coscycles = 2
330  LET cosconfac = 2*PI*coscycles/256
340  LET cosamplitude = 80
350  FOR x = 0 TO 255
360  LET y = cosamplitude*COS (x*cosconfac)
370  GOSUB 1000
380  NEXT x
500  STOP
1000 REM Draw a line to x,y
1010 LET a=x
1020 LET b=88+y
1030 IF a<0 OR a>255 THEN LET flag=0:RETURN
1040 IF b<0 OR b>175 THEN LET flag=0:RETURN
1050 IF flag=0 THEN PLOT a,b:LET flag=1
1060 DRAW a - PEEK 23677,b - PEEK 23678
1070 RETURN
```

With other trigonometrical functions an error message is produced if the plotted point is not within the range of the screen. The function plotted should therefore be checked for its maximum and minimum values and the amplitude adjusted. An example is the function 60sin(3A) + 80cos(2A), which can have a value of 140, so the amplitudes in lines 220 and 340 should be reduced accordingly, in this case by half. To plot this function as well as the functions that go to produce it, add these lines to the previous program:

```
400  INK 6
410  LET flag=0
420  FOR x = 0 TO 255
430  LET y = sinamplitude * SIN(x * sinconfac) +
     cosamplitude * COS(x * cosconfac)
440  GOSUB 1000
450  NEXT x
```

The function tan(A) goes to infinity when A is ninety degrees producing an error. A sufficient knowledge of the properties of the function avoids crashing the program. The following program plots tan(A) for two cycles:

```
 10  REM Draw axes
 20  PLOT 0,88:DRAW 255,0
 30  PLOT 0,0:DRAW 0,175
100  PAPER 7
110  INK 0
115  LET flag=0
200  LET cycles = 1
210  LET confac = 2*PI*cycles/256
220  LET amplitude = 5
230  FOR x = 0 TO 255
235  IF x=64 OR x=192 THEN GO TO 260
240  LET y = amplitude*TAN(x*confac)
250  GOSUB 1000
260  NEXT x
300  STOP
1000 REM Draw a line to x,y
1010 LET a=x
1020 LET b=88+y
1030 IF a<0 OR a>255 THEN LET flag=0:RETURN
1040 IF b<0 OR b>175 THEN LET flag=0:RETURN
1050 IF flag=0 THEN PLOT a,b:LET flag=1
1060 DRAW a - PEEK 23677,b - PEEK 23678
1070 RETURN
```

Some functions still cause problems. Consider the equation of the circle

$$x^2 + y^2 = \text{radius}^2$$

where the maximum value for the radius is 511. BASIC cannot handle the equation as it is, it must be transformed to get a single value of y (or x) on the left of the equation.

```
y = SQR(radius*radius - x*x)
```

Care must now be taken to prevent the absolute value of x from exceeding the radius, otherwise y becomes imaginary. Also the square root is automatically positive, so we shall only get the whole circle by separately including the negative value.

```
LET radius=400
FOR x = -radius TO radius
LET y = SQR(radius*radius - x*x)
PLOT x,88+y
PLOT x,88-y
NEXT x
```

This gives uneven spacing between the plotted points and a more satisfactory way, which makes use of a separate parameter is preferred. For circular functions

angle is the most useful parameter. There is little point in doing this for circles, since Spectrum BASIC already contains a CIRCLE statement, but for other conic sections it is essential. The program for an ellipse is as follows:

```
 10  REM Draw axes
 20  PLOT 0,88:DRAW 255,0
 30  PLOT 128,0:DRAW 0,175
100  PAPER 7
110  INK 0
120  LET xradius=80
130  LET yradius=40
140  PLOT xradius+128,88:REM Move to first point
200  FOR a = 0 TO 360 STEP 10
210  LET angle=a*PI/180
220  LET x = xradius*COS(angle)
230  LET y = yradius*SIN(angle)
240  DRAW 128 + x - PEEK 23677,88 + y - PEEK 23678
250  NEXT a
```

The parabola is given by

```
x=2*a*t
y=a*t*t
```

For example:

```
 10  REM Draw axes
 20  PLOT 0,88:DRAW 255,0
 30  PLOT 128,0:DRAW 0,175
100  PAPER 7
110  INK 0
120  PLOT 58,171:REM Move to first point
200  FOR t = -70 TO 70
210  LET x = t
220  LET y = t*t/30 - 80
230  DRAW 128 + x - PEEK 23677,88 + y - PEEK 23678
240  NEXT t
```

Particularly pleasing to the physics teacher is the production of Lissajous figures using sine equations with different frequencies and phase angles.

```
100  CLS
110  PAPER 7
120  INK 0
130  INPUT "Phase angle = ";ph
140  INPUT "Frequency Ratio = ";freq
150  LET amplitude = 80
160  LET phase=ph*PI/180
```

```
170  PLOT 128+amplitude*SIN (phase),88
200  FOR a = 0 TO 100000
210  LET angle = a*PI/180
220  LET x = amplitude * SIN(freq*angle + phase)
230  LET y = amplitude * SIN(angle)
240  DRAW 128 + x - PEEK 23677,88 + y - PEEK 23678
250  NEXT a
```

If non-integral values of the frequency ratio are desired, it can be many cycles before the pattern repeats itself, hence the need for the large number of cycles in line 200.

Applications

These ideas can be turned to practical classroom use in a number of ways. Once the principles are appreciated a few hours at the keyboard will tell students more about the behaviour of functions than a whole series of lectures.

Simple functions

If a phenomenon can be described by a simple equation then it can be plotted in the ways just described. For example the distance-time graph of a body that falls from rest can be plotted with the equation

```
s = g * t * t / 2
```

This translates into a program as follows:

```
100  CLS
110  PAPER 7
120  INK 0
130  INPUT "Acceleration due to gravity = ";g
140  LET acc = -g
150  PLOT 0,160
200  FOR t = 0 TO 255
210  LET x = acc * t * t / 2
220  LET y = 160 + s/1000
230  IF y<0 THEN GO TO 250
240  DRAW t - PEEK 23677,y - PEEK 23678
250  NEXT t
260  GO TO 130
```

Different values for gravity may be entered and their effects noted. In this program values between 0 and 10 give the best results.

Wherever there are more than two variables, the others can be held constant during each scan of the screen and altered later by entering new values in precisely the same way as this. This process fits most equations experienced in O

level physics and chemistry. Typical examples are as follows:

```
V = I * R
W = I * I * R
P * V = const
1/v + 1/u = 1/f
F = k * m * M / (r * r)
```

Trigonometrical functions allow some of the properties of vibrations and waves to be investigated. The superposition of two waves to give interference, beats and modulated waves was demonstrated above. Here is another example: a program for an object executing damped oscillations. This includes a plot of the wave envelope too, so that the student can appreciate which part of the equation causes the different shapes of the graph. This program is actually an oversimplification, since no account has been taken of the effect of damping on the frequency of the oscillations. A much better way of doing the whole thing is discussed later in this chapter.

```
  1  REM DAMPED OSCILLATIONS
 10  CLS
 20  PLOT 0,0
 30  DRAW 0,160
 40  PLOT 0,80
 50  DRAW 255,0
130  INPUT "Amount of friction (0 to 0.1) = ";friction
150  LET cycles = 4
160  LET confac = 2 * PI * cycles/256
190  LET amplitude = 75
200  PLOT 0,80 + amplitude
210  FOR t = 0 TO 255
220  LET angle = t * confac
230  LET displacement = amplitude * EXP(-t * friction) * COS(angle)
240  DRAW t - PEEK 23677,80 + displacement - PEEK 23678
250  NEXT t
300  GO TO 100
```

A particularly satisfactory demonstration of the Fourier synthesis of a square wave is obtained with the following program:

```
 10  REM FOURIER SYNTHESIS
100  CLS
110  PAPER 7
120  INK 0
130  PLOT 0,88:DRAW 255,0
140  PLOT 0,0:DRAW 0,175
150  PLOT 0,88
```

```
200  LET cycles = 2
210  LET confac = 2 * PI * cycles / 256
220  LET amplitude = 80
230  FOR x = 0 TO 255
240  LET angle = x * confac
250  LET y1 = amplitude * SIN(angle)
260  LET y2 = amplitude / 3 * SIN(3 * angle)
270  LET y3 = amplitude / 5 * SIN(5 * angle)
280  LET y4 = amplitude / 7 * SIN(7 * angle)
290  LET y5 = amplitude / 9 * SIN(9 * angle)
300  LET y = y1 + y2 + y3 + y4 + y5
310  DRAW x - PEEK 23677,88 + y - PEEK 23678
320  NEXT x
```

Provided you are prepared to wait, this process may be continued for as many harmonics as you wish.

Complicated functions

Many functions cannot easily be rearranged to make one variable into the subject of the equation. There is usually no necessity for this in any case as the microcomputer is quite capable of carrying out the calculation in parts. A good example of this is the voltage across a capacitor in an LCR circuit (Figure 3.1). If this is plotted against frequency a resonance curve is produced. The input voltage is assumed to be constant (E) and this produces a current in the circuit (I).

I is given by E/Z, where Z is the impedance of the circuit at the given frequency (f). The voltage across the capacitor (C) is thus $I/2\pi fC$. the value for Z is obtained from the formula

$$Z^2 = R^2 + (2\pi fL - 1/2\pi fC)^2$$

RESONANCE (36) plots the desired curve. The values of L and C should be chosen to make the resonant frequency come near the middle of the screen (Plate 6). Assuming inductances in millihenries and capacitors in microfarads, this gives

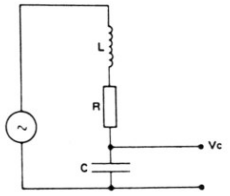


Figure 3.1 LCR resonance

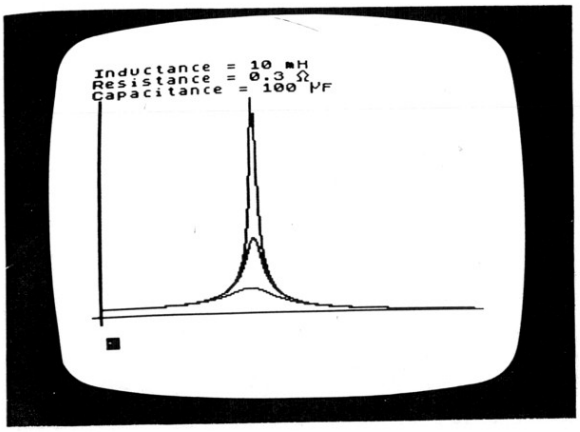


Plate 6 Resonance curve

L = 20 mH and C = 100 μ F. (Strictly, this frequency is the angular frequency, but this is not apparent in the final plot, so it is ignored here. If required it is simple enough to allow for it.) Here is the essential structure of the program:

```
INPUT "Inductance = " L
INPUT "Capacitance = " C
INPUT "Resistance = " R
LET E = 50 :REM APPLIED VOLTAGE
FOR frequency = 1 TO 2500 STEP 10
LET XL = frequency * L
LET XC = 1/(frequency * C)
LET X = XL - XC
LET Z = SQR(R*R + X*X)
LET I = E/Z
LET VC = I * XC
DRAW frequency, VC
NEXT frequency
```

It can be seen how the final capacitor voltage is obtained after several separate

calculations, each of which should be familiar to the student. By showing each step of the calculation like this, it is easier to keep sight of the physics. The value of this kind of program is that students can investigate the effects of carying one parameter at a time.

Graph plotting with experimental data

Probably the most useful application of graphs in science is the plotting of experimental data. This is usually carried out to obtain the slope or intercept of a straight line graph, where the best line is obtained from the data by guesswork. The computer can be a great help in teaching students to do this, since the 'best' line can then be obtained by the method of least squares. The use of PLOT and the relative DRAW easily allows a cross to be produced at the point x,y.

A complete program to accept students' data and to process it is not easy if the data can have all possible values. LEAST SQUARES PLOT (3) works to some extent and may be adapted to suit any particular application.

For statistical data a bar chart is preferred. In this case the x coordinate is probably discontinuous, but whether it increases in steps of one, two or five, etc. is a matter of choice in each case. So once again a single program will not suffice for all occasions and the example given – SUM OF TWO DICE (30), will need to be adapted for each particular case. Horizontal bar charts are just as easy to achieve.

The use of RND

The random number function of BASIC is not provided just for computer games! It is invaluable for carrying out statistical experiments, particularly where the results can be displayed graphically. RADIOACTIVE DECAY (29) illustrates the use of this function to decide which nucleus should decay next. Since the position of this next nucleus is decided at random, the chance of choosing a position with an undecayed nucleus depends upon the number of such nuclei remaining. This therefore simulates radioactive decay quite well (Plate 7).

If one of the variables is discontinuous, then the bar chart is an obvious means of display as SUM OF TWO DICE (30) illustrates. This is a standard experiment, but few students could do it more than a few times as a practical exercise, so the microcomputer can help to make the pattern more obvious. In the space of a few minutes the experiment is performed hundreds of times.

The use of RND is particularly valuable in biology for simulating genetic linkage and there are very many programs available for this. It is also used in the simulation of Geiger and Marsden's experiment discussed later (RUTHERFORD, 40).

Iterative methods

The *Nuffield Advanced Physics* originators were far-sighted in noting probable

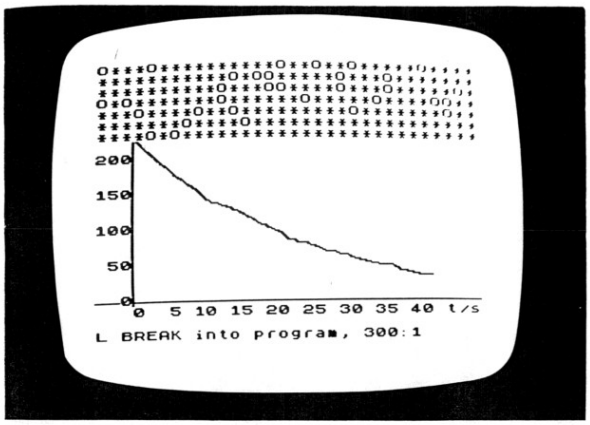


Plate 7 Radioactive decay

trends towards more and cheaper calculators. They describe several experiments which run very nicely on a microcomputer. Basically they suggest that as well as the traditional algebraic (usually integral calculus) analysis of physical phenomena, teachers should explore numerical solutions. A good example is the discharge of a capacitor through a resistor. This can be solved algebraically by noting that the current flowing through the resistor is the differential of the charge on and hence the voltage across the capacitor. Since this current is directly proportional to voltage, all that has to be done is integrate a reciprocal and end up with an exponential logarithm. The mathematics so obscures the physics that it is better to seek a step-by-step solution to the problem.

The voltage (V) across the capacitor is related to the charge (Q) in the capacitor by

```
Q =V * C      (Eq. 1)
```

This voltage causes a current (I) to flow through the resistor according to the well-known formula

```
V = I * R     (Eq. 2)
```

If a current of one ampere flows for one second, the capacitor will lose one coulomb of charge, so in one millisecond, say, it will lose one millicoulomb of charge. Thus the remaining voltage on the capacitor after one millisecond is a bit less than it was before, and we can use Eq. 1 to calculate exactly how much less. This gives us a new value for V, with which to begin the next millisecond. By hand it could take some time to see how the capacitor voltage is falling but the microcomputer makes very short work of the calculations. The exponential curve is obtained with only the three fundamental equations. The actual program is listed as CAPACITOR DISCHARGE (37), and any student, particularly one able to comprehend the calculus approach, could write such a program.

The main difficulty is ensuring that the chosen values give results that fit the screen. The time axis (x axis) goes from 0 to 255 units. If these are seconds, then a time constant of about 50 seconds is needed for the RC circuit. This is somewhat unrealistic, so we pretend that our time scale is in microseconds instead. The value for R can thus be a few ohms and the value for C between 1 and 10 microfarads. Since different values for R and C can be entered, students can be asked to discover how the rate of decay depends upon R and C. In so doing, they learn a great deal about the decay curve, which should aid their understanding of, say, radioactive decay too.

This approach to the analysis of phenomena is called the **iterative method**. It is applicable in very many areas (and not just physics). Programs 38 to 40 show how it may also be applied to motion. Plate 8 shows the sort of results obtained with PROJECTILES (38). The basic algorithm is as follows:

1 Assume initial position, velocity and acceleration.
2 Assume a small increment of time.
3 Determine the new velocity after this time interval.
4 Determine the distance travelled at this velocity during this time interval.
5 Calculate the new position.
6 Return to step 1, with new values of velocity and acceleration.

This gives a delightful way of tackling simple (and damped) harmonic motion, without recourse to differential equations.

```
  1  REM DAMPED OSCILLATIONS
  2  REM BY THE ITERATIVE METHOD
 10  CLS
 20  PLOT 0,0
 30  DRAW 0,160
 40  PLOT 0,80
 50  DRAW 255,0
100  PRINT AT 0,0;"                "
110  PRINT AT 1,0;"                "
120  PRINT AT 2,0;"                "
130  INPUT "Amount of friction (0 to 0.1) = ";friction
```



```
140  PRINT AT 0,0;"Amount of friction = ";friction
150  INPUT "Spring stiffness (0 to 10) = ";spring
160  PRINT AT 1,0;"Spring stiffness = " spring
170  INPUT "Mass of object (0 to 10) = ";mass
180  PRINT AT 2,0;"Mass of object = ";mass
190  LET amplitude = 75
200  LET displacement = amplitude
210  LET speed = 0:REM INITIAL SPEED
220  MOVE 0,amplitude
230  LET time = 0
240  LET timeinc = 1
250  PLOT 0,155
260  Iteration begins
270  LET restoringforce = -spring * displacement/100
280  LET frictionalforce = -friction * speed
290  LET totalforce = restoringforce + frictionalforce
300  LET acceleration = totalforce/mass
310  LET speed = speed + acceleration * timeinc
320  LET displacement = displacement + speed * timeinc
330  LET time = time + timeinc
340  DRAW time - PEEK 23677,80 + displacement - PEEK 23678
350  IF time<255 THEN GO TO 260
360  GO TO 100
```

On each run different values can be entered to discover the role that each variable plays in the overall motion. If this is coupled with actual experimental work with masses on the end of a spring, I believe the approach to be much more truly physics than the traditional mathematical approach.

For projectiles there are two directions - x and y - to consider. However, these can be considered entirely independently, so the only complication is that there are twice as many calculations in each cycle. PROJECTILES (38) illustrates this: the motion in the x direction is constant velocity, while that in the y direction is constant acceleration. This program also shows how easy it now is to include more difficult ideas. The usual treatment of projectiles ignores friction and leads to the ideal case of 45 degrees as the angle for maximum range. PROJECTILES incorporates a frictional drag, proportional to the speed, which reduces the speed and leads to the idea of terminal velocity. The resulting motion is not unlike that predicted by Bacon's impetus theory (Plate 8). The acceleration due to gravity and the friction (dragcoeff) can be altered for different effects (projectiles in treacle?).

Motion under a central force is rarely understood. NEWTON (39) is a game that any student should be able to solve, but it often fools physics graduates. The objective is to put a rocket into moon orbit from outside. Try it and see if you understand Newton's laws yourself. The program first calculates the distance

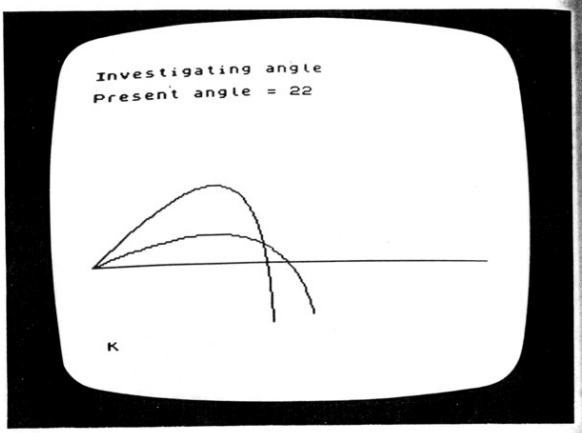


Plate 8 Projectile motion

between the rocket and the centre of the moon. This is converted into two forces, one which affects the acceleration in the x direction, the other the y direction. This in turn leads to predictions of where the rocket will be after the next unit of time (timeinc) and the process reiterates until the rocket crashes on the moon's surface or disappears off the screen. The value of 'timeinc' can be altered as before to achieve smoother if slower motion.

Alpha-particle scattering by a gold nucleus provides a classic derivation for university undergraduates. I understand that the mathematics of this was too difficult for Rutherford and was handed over to a mathematician. I imagine that Rutherford would have loved the iterative method. The essential part of RUTHERFORD (40) is very similar to its equivalent in NEWTON, except that the force acting is reversed to produce repulsion instead of attraction. The motion is also speeded up (with a loss in resolution) to allow a large number of particles to be observed. These are fired at random at the gold nucleus and only a few pass close enough to be deflected (Plate 9).

So the mathematics is reduced to the level where any sixth former can understand it. I am not sure that many teachers, particularly of physics, have yet realized the implications of this. If, as I suspect it will, computer programming

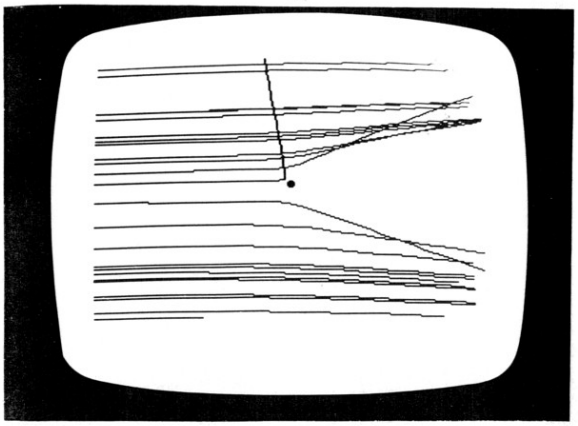


Plate 9 Alpha-particle scattering

becomes the fourth R, then the traditional dependence of advanced science subjects upon mathematics could be allowed to decline, thus opening them up to more students than hitherto.

Modelling the environment

The iterative process has wider applications than those above and it was used by the Huntingdon Project, which produced the well-known simulations in biology and chemistry. One of these, POLLUT, analyses the effect of certain types of pollutant upon water life and another, HABER, looks at the effects of changing the temperature and pressure etc. of the reactants in an industrial process. Practically anything that can be quantified, can be mathematically modelled, although the accuracy of the predicted outcomes is not necessarily reliable. It depends upon whether all the important factors have been taken into account.

The principles are well illustrated by the FOX AND RABBITS program on the Horizon introductory cassette that comes with the Spectrum. Fox and rabbit populations are modelled to predict how they change with time. It is assumed that the rabbits' food is infinite so that they can reproduce without restriction. The

growth in the fox population is dependent upon the supply of rabbits. If foxes only eat rabbits, then they will begin to die if their population exceeds some factor of the rabbit population. Foxes with abundant food reproduce at a constant rate, which is also chosen before the start of the iteration. It is assumed that the starvation rate of foxes depends upon the ratio of foxes to rabbits, which seems reasonable. It is further assumed that the death rate of rabbits is proportional to the product of rabbits and foxes. This assumes that one fox with, say, 1000 rabbits will still eat twice as much as the same fox with 500 rabbits. (I greatly suspect the model at this point.) The number of rabbits that are eaten depends upon the number of foxes and the number of foxes depends upon the number of rabbits. This classic problem can only be solved by an iterative process, since the equations generated have no analytical solution. The essential structure of the program is shown below.

```
REM FOX AND RABBIT SIMULATION
LET weeks = 0
LET rabbits = 3000
LET foxes = 20
REPEAT
LET babyrabbits = rabbits * rabbitgrowthrate
LET deadrabbits = foxes * rabbits / 1000
LET rabbits = rabbits + babyraboits - deadrabbits
LET babyfoxes = foxes * foxgrowthrate
LET deadfoxes = 5 * foxes / rabbits
LET foxes = foxes + babyfoxes - deadfoxes
LET weeks = weeks + 1
PLOT weeks,foxes
PLOT weeks,rabbits
UNTIL finished
```

As a physicist I find this much less satisfying than the same approach applied to physics, because I can justify some of the values entered into the equations of motion. I am not at all sure about the constants entered into the fox and rabbits program.

4 Microcomputer timing and control

'The question is,' said Humpty Dumpty, 'which is to be Master - that's all.'
(Lewis Carroll, *Through the Looking Glass)*

Interfacing a microcomputer

Most control applications use **two-state** devices. An electric light switch can be up or down. An electromagnetic relay can be on or off. A valve can be open or closed. Digital electronic systems are used to switch such devices on or off. Although quite complex, a microcomputer is still only another digital system, so it is possible to use a microcomputer to control the above devices. It can switch lamps, relays, motors and valves on or off.

This is not a normal function of a microcomputer and it has not been designed specifically to do this. Consequently the current needed to switch on these devices may be larger than that provided by the microcomputer output. There has to be some **interface** between the microcomputer and the device being switched, to boost the switching current to the correct levels.

A microcomputer can also be used to detect whether any particular two-state device is in its on or its off state. Here, the switching voltages involved may be different for each device, so some interface must be used to change the voltage levels of the device to the levels acceptable to the microcomputer.

In digital electronics we are only concerned with two-state devices, ones that can be switched on or off. Generally, to switch a device on, we send a HIGH voltage to its input. To turn it off, we send a LOW voltage. HIGH and LOW are obviously not the same for different devices. Here are a few examples:

Device	On	Off
Light-emitting diode	1.2 V	0.5 V
Torch bulb	3.0 V	1.5 V
Electromagnetic relay	5.0 V	2.0 V
Silicon transistor	0.7 V	0.5 V
TTL integrated circuit	2.4 V	0.4 V

To remove this uncertainty about what is 'HIGH' and what is 'LOW' engineers use **TTL** logic levels. TTL stands for **transistor transistor logic**; it is a particular standard used in the electronics industry. A TTL HIGH voltage is between 2.4 and 5.5 V, which, as you can see, will switch on all the above devices. A TTL LOW voltage is between 0.4 and 0 V, which will switch all these devices off. A HIGH voltage is also called a **logic level 1** and a LOW voltage is called a **logic level 0**.

Connections to the ZX Spectrum microcomputer are made through its extension port. When this book was planned, it was intended to give details of how to construct an interface to fit onto this port, to allow a whole range of control experiments and measurements to be carried out. I was then introduced to the Interspec, designed by Dave Palmer and decided that I was wasting my time. This interface did all that I required and cost much less than the average teacher would pay for the components alone. I thus decided to abandon the original plan and just discuss the uses of the Interspec. These are formidable and take up most of the next two chapters. I make no apology for being so specific about one particular interface, the general principles described are true for most other Spectrum interfaces too.

Introduction

The outward appearance of the DCP interface (henceforth called the Interspec) is shown in Plate 10. The initial applications of this, now described, assume no understanding of electronics; they aim to show beginners how to use the Interspec for experiments in science and control technology. Later applications are for those who wish to write their own programs. They explain in more detail how the interface works and give more examples of its uses. To understand these ideas you will need some knowledge of digital electronics, such as is found in *Microelectronics* (Hutchinson, 1984).

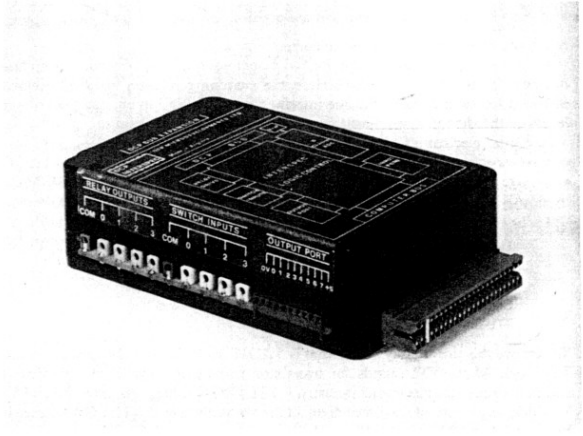


Plate 10 Interspec interface

Eighteen of the programs listed in the Appendix, illustrate the use of the Interspec. Some of these will be immediately useful in teaching, for example, the time, speed and acceleration meter TSA (9). Others may be readily adapted for this purpose. It is possible to load and run these progrms without being a microelectronics expert. Many other short programs are listed in these notes as examples to be entered from the keyboard. Their purpose is to demonstrate ideas rather than provide working programs. Most simple programs listed require only a 16K RAM Spectrum and they will also run on the larger 48K Spectrum. The timing programs require a 48K Spectrum for the reasons given later.

Connecting the Interspec to the ZX Spectrum

The power to the Spectrum must be switched off before the Interspec is connected. Failure to do this may damage both the Interspec and the computer.

Unlike most other microcomputers, the ZX Spectrum does not have a user port. Instead it has an extension connector at the back, which connects to the address and data lines of the microprocessor. The Interspec must be plugged onto this connector. The slot in the connector makes sure that the interface can only fit one way, so there is no possibility of plugging it in wrongly. If the ZX printer is being used too, the printer should be plugged in first and then the Interspec.

Some Spectrum power supply units do not seem able to cope with the printer, Interspec and 48K Spectrum all at once. A dark band appears across the screen, indicating a power overload. If you are in this situation, you will have to omit the printer when the Interspec is connected. You should not connect a separate 5 V supply to the Interspec unless you are very familiar with the problems that this could produce.

In all cases the sockets and connectors should be eased together without twisting them. The best way is to place them flat on the table and then push them together gently but firmly. Careless connection may damage both the interface and the microcomputer itself. If you are unsure of what to do, please seek expert help. It is better to lose a few days than to damage your microcomputer through ignorance.

Outputs

As mentioned above, different devices need different voltages and currents to drive them. The Interspec has TTL outputs, but for most purposes the relay outputs (Figure 4.1) are easier to use. These are the four yellow sockets on the left side of the interface. The common terminal for these relay contacts is the green socket next to these.

The relays are controlled through an output port of the interface. Each pair of contacts can be opened or closed separately by writing a 1 or a 0 into the correct bit positions of this output port. This 'writing' is achieved with the OUT statement of Spectrum BASIC. The address of the output port is 63, so the syntax is

```
OUT 63,X
```

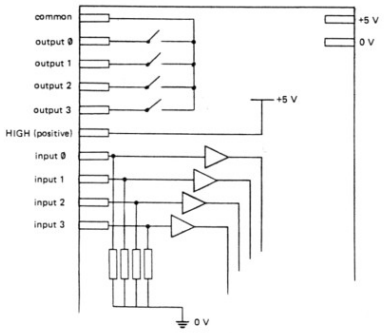


Figure 4.1 Relay outputs and switch inputs

where X is a number between 0 and 15. Each relay is closed when its value is written to the output port address according to the following table:

Relay number	Bit pattern	X value
3	1000	8
2	0100	4
1	0010	2
0	0001	1

OUT 63,0 (in binary 0000) opens all contacts.
OUT 63,8 (binary 1000) opens contacts 0, 1 and 2 and closes 3.

Combinations of different relays are made by adding these values together.

OUT 63,3 (binary 0011) opens contacts 2 and 3 and closes 0 and 1.
OUT 63,15 (binary 1111) closes all contacts.

This is called **positive logic.**

In order to 'see' the relays operate, they need to be connected to lamps or light emitting diodes (LEDs) or the relays can be connected to motors or other devices. The diagram in Figure 4.2 shows how to connect such devices to the relay sockets, with the common terminal connected to an external power source.

Some users may wish to use the lamp indicator units of the *Nuffield Advanced Physics* course in the above arrangement instead. These work perfectly well with an external 6 V power supply. For any external power supply, it must not be connected to the 5 V terminal of the Interspec. One end of the external power supply may be connected to the common terminal of the relays and in most cases the other end may be connected to the 0 V terminal of the Interspec. Extra care

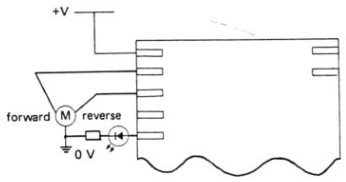


Figure 4.2 Connecting to the relay outputs

should be taken when the external power supply is mains-derived.

Others may wish to use the indicator units of the various digital electronics boards now available commercially. Rather than sourcing the LED as in Figure 4.2, such boards may sink them instead, with one of the methods shown in Figures 4.3 and 4.4. This too is quite satisfactory, except that the relay contacts may then need to be connected differently. With the Unilab indicator unit, each input is held LOW by a resistor, so the above discussion is valid. The Griffin indicator unit does not have this resistor, so each LED is normally on when its input is left unconnected (as is normal with TTL devices). So when the relay is open, the LED comes on (the opposite of the above). The method of using the relays in this case is to connect the common line to 0 V, so that it switches the LED off when the relay is closed. The number to be sent to the output port is then the opposite or inverse of the above, as follows:

Relay number	Bit pattern	X value
3	0111	7
2	1011	11
1	1101	13
0	1110	14

OUT 63,0 (in binary 0000) opens all contacts and switches all LEDs on.
OUT 63,8 (1000) switches LEDs 0, 1 and 2 on and switches 3 off.

The easiest way to remember this is to use the previous table, but to subtract the bit value from 15 each time (see Example 1a below). This technique is called **negative logic**. Whether any board uses positive or negative logic must be determined by inspection or experiment.

Controlling the environment

The examples that follow show how to use the OUT instruction. The listings provided are copies of printouts of real working programs, and we hope, contain no typing errors. It should be possible for even a novice to enter and run these programs with no further theory. For those wishing to write their own programs

later chapters contain further help. Example 1 is written in two forms to illustrate both positive and negative logic. All other programs illustrate positive logic only and use a logic board like that shown in Figure 4.4. Those using the simpler form shown in Figure 4.3 will need to alter the programs in the way discussed above. These sink methods have been used because some readers may wish to utilize the TTL output port of the Interspec instead of the relay sockets. If this is done, then the output address needs to be altered to 95 instead of 63 in the programs. Those using the relay sockets only can simply drive the LEDs directly as shown in Figure 4.2. The Griffin programmable logic board uses this method and so is directly usable in the following applications.

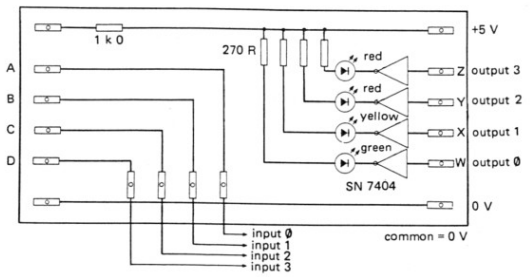


Figure 4.3 Negative logic

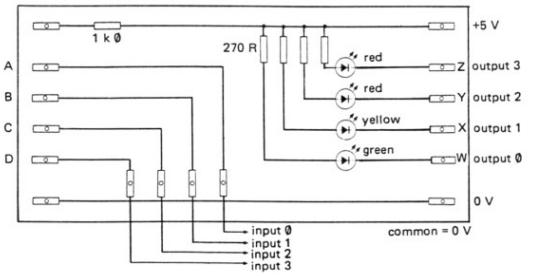


Figure 4.4 Positive logic

Increasingly in industry, the solutions to problems in electronics are becoming ones of adapting a general purpose circuit to a specific application, rather than designing a special circuit each time. Traditional control technology in schools has laid emphasis upon the second of these approaches: the hardware solution. The Spectrum and its interface can be used to demonstrate the more modern software approach. The programs described below demonstrate how the unit can be used to switch the LEDs on and off. Note that in each case, the electronic circuit remains the same, it is only the program that is changed.

Example 1 Traffic lights (positive logic)

```
  1 REM TRAFFIC LIGHTS
 10 REM Define outputs
 20 LET red = 4
 30 LET amber = 2
 40 LET green = 1
 50 LET outputs = 63
100 REM traffic lights sequence
110 OUT outputs,red
120 PAUSE 250
130 OUT outputs,red + amber
140 PAUSE 100
150 OUT outputs,green
160 PAUSE 250
170 OUT outputs,amber
180 PAUSE 100
500 GO TO 100
```

Example 1a Traffic lights (negative logic)

```
  1 REM TRAFFIC LIGHTS
 10 REM Define outputs
 20 LET red = 4
 30 LET amber = 2
 40 LET green = 1
 50 LET outputs = 63
100 REM traffic lights sequence
110 OUT outputs,15 - red
120 PAUSE 250
130 OUT outputs,15 - (red + amber)
140 PAUSE 100
150 OUT outputs,15 - green
160 PAUSE 250
170 OUT outputs,15 - amber
180 PAUSE 100
500 GO TO 100
```

For this program it is assumed that three LEDs represent the red, amber and

green traffic lights. The program shows how these lights can be controlled by sending the numbers 1, 2 and 4 (and combinations of them) to the output port. As an exercise try switching on the LEDs in a different sequence. In particular make them follow the continental system, where the lights change directly from red to green.

Example 2 Random lights

```
 10  REM RANDOM LIGHTS
 20  REM Define outputs
 50  LET outputs = 63
100  REM switch lamps
110  LET randnum = INT (RND*16)
120  OUT outputs,randnum
130  PAUSE 25
500  GO TO 100
```

To satisfy those critics of Example 1, that they can do traffic lights just as well without a microcomputer, Example 2 is almost impossible with traditional hardware - switching the LEDs on and off in random sequence. For this purpose a random number between 0 and 15 is sent to the output port.

Example 3 Psychedelic lights

```
 10  REM PSYCHEDELIC LIGHTS
 20  REM Define outputs
 30  LET odd = 1 + 4
 40  LET even = 2 + 8
 50  LET outputs = 63
100  REM switch lamps
110  OUT outputs,odd
120  PAUSE 25
130  OUT outputs,even
140  PAUSE 25
500  GO TO 100
```

Patterns of lights can easily be achieved with this sort of program. Can you discover other interesting sequences?

Example 4 A binary counter

```
 10  REM BINARY COUNTER
 20  REM Define outputs
 50  LET outputs = 63
100  REM switch lamps
110  FOR i = 0 TO 15
120  OUT outputs,i
130  PAUSE 50
140  NEXT i
500  GO TO 100
```


This example also switches on the LEDs in a more orderly way, by adding 1 to the number sent to the output port each time. The LEDs thus count up in binary. Can you discover how to make the LEDs count down in binary instead?

Example 5 A shift register

```
 10  REM SHIFT REGISTER
 20  REM Define outputs
 50  LET outputs = 63
100  REM switch lamps
110  OUT outputs,0
120  PAUSE 25
130  OUT outputs,1
140  PAUSE 25
150  OUT outputs,2
170  PAUSE 25
180  OUT outputs,4
190  PAUSE 25
200  OUT outputs,8
210  PAUSE 25
500  GO TO 100
```

One common chip used in microelectronics is the shift register, which is simulated by this program. They are particularly useful for converting serial data (where the bits are sent one after the other along a single pair of lines) into parallel data (where all eight bits are sent simultaneously along separate lines).

Example 6 A metronome

```
 10  REM METRONOME
 20  REM Define outputs
 30  LET on = 15
 40  LET off = 0
 50  LET outputs = 63
100  REM Collect frequency
110  CLS
120  PRINT AT 3,10;"METRONOME"
130  PRINT AT 6,0;"Enter the number"
140  PRINT AT 8,0;"of beats per minute."
150  INPUT beats
160  LET interval = 50*60 / beats - 2
170  PRINT AT 20,0;"Press any key to change."
200  REM toggle relays
210  OUT outputs,on
220  PAUSE 2
230  OUT outputs,off
240  PAUSE interval
```

```
250  IF INKEY$="" THEN GO TO 200
500  GO TO 100
```

This program uses the PAUSE instruction to determine the interval between pulsing the lights. This instruction waits for the time interval (in fiftieths of a second) to elapse before repeating the cycle. Line 250 tests the keyboard and, if any key is being pressed, restarts the program.

Further ideas

The outputs from the Interspec are so easy to control, that I now use them for most work in elementary electronics. The programs are quite simple – in most cases simple sequential programs are sufficient (Example 5). Even those who have never programmed a computer before can get control programs working in a very short time. Couple this to the possibility of getting the programs to monitor and check the student's progress and understanding and you have the embryo of a new way of teaching electronics in schools. This is currently being investigated further. Some of this is described in more detail later.

By connecting a reversible electric motor to the relays (Figure 4.2), it is possible to drive the motor forwards or backwards. If the motor is joined to a model car (Meccano, Lego or Fishchertechnik) the motion of this car can then be controlled. The car could move forwards, stop for a given time and then reverse to its starting point. Unfortunately, this system is unreliable: the car rarely returns to its exact starting point, so after a few cycles, it has tended to progress down the table. Clearly there has to be some feedback to the microcomputer, so that it 'knows' exactly where the car is. We need to use the inputs of the interface too.

Inputs

The interface can be used to detect whether any particular two-state device is in its on or its off state. Here again the voltages involved will be dependent upon the devices. So the interface must change the voltage levels of the device to the levels acceptable to the microcomputer. This is achieved by the four switch-inputs (the white terminals on the left of the interface – Figure 4.1). When these inputs are left unconnected, they are normally LOW. They can be switched HIGH by connecting directly or indirectly to the red socket nearby. (It is possible to change this way of working; details are given later.)

In this context HIGH and LOW refer to voltage levels. If a switch input is connected to a voltage greater than 1.2 V, the interface interprets this as a HIGH (or on) level. If the voltage applied to the switch input is less than 0.8 V, the interface interprets this as a LOW (or OFF) level. These voltages are with respect to the 0 V line of the interface (the black socket). Voltages between 0.8 V and 1.2 V may be interpreted as HIGH or LOW, so this region should be avoided. Voltages outside the range 0 to 5.5 V may damage the interface, so care must be taken not to use such voltages. This means that alternating voltages should not be connected directly to the switch inputs.

Alternatively, a switch input can be sent HIGH by connecting it to the red socket through a resistor of less than 5 kilohms resistance. If the resistance is more than about 15 kilohms, the switch input will be LOW. This is very useful since a photocell or a **light dependent resistor (LDR)** can be connected directly between these terminals to provide a light sensitive switch. Or the resistor could be a thermistor thus giving a temperature sensitive switch.

Any real switch such as a float switch, mercury tilt switch or pushbutton switch could also be used to provide inputs. The microcomputer can thus be used to detect its environment and to respond to it in different ways. We now have a way of communicating information to the microcomputer other than via the keyboard.

The information about whether a switch input is HIGH or LOW, is sent to the microcomputer by the interface. The microcomputer interprets it as a **binary digit** (or **bit**). A LOW level is called a logic 0 and a HIGH level is called a logic 1. The microcomputer reads the information from the input port, which shares the same address as the output port above. This too is a four-bit port, meaning that the logic levels of all four switch inputs are read at the same time. Each of the inputs is connected to a different bit as before. If any input is HIGH, its corresponding bit will be a 1. If the input is LOW, then its corresponding bit will be a 0. In reality the input port has eight lines of which we are using only the bottom four. The top four lines are always HIGH. The whole set of eight bits is read at once, (thus giving a binary number from 1111 0000 to 1111 1111, depending on the logic state of the inputs). When the input port is read, this binary number is converted to its decimal equivalent by the Spectrum, thus giving a number between 240 and 255. Subtracting 240 from this number (the top four bits) gives a value, between 0 and 15.

The address of the input port of the Interspec is 63, exactly the same as for the output port. The BASIC statement

```
LET X = IN 63
```

will set the value of X to some integer between 240 and 255, depending upon which of the switch inputs are HIGH and which are LOW. After subtracting 240 this number will be in the range 0 to 15. In binary, these extremes are the numbers 0000 (corresponding to all four switch inputs being LOW) and 1111 (corresponding to all switch inputs being HIGH). To determine which lines are HIGH and which are LOW, this decimal number must be decoded into its equivalent bits using this table:

Input number	Bit pattern	X value
3	1000	8
2	0100	4
1	0010	2
0	0001	1

If more than one line is HIGH, the X value will be a combination of the corresponding numbers above. Thus if the X value is 12, this means that lines 2

and 3 are HIGH and the others are LOW. Similarly, if LET X = (IN 63) - 240 yields the value 3, this means that the lines 0 and 1 are HIGH and the others are LOW. Routines can be written in BASIC to inspect the value in X to find out which lines are HIGH or LOW.

The Spectrum can occasionally read wrong data from the Interspec. This can be cured by preceding each IN statement by a PAUSE statement. (We have no idea why this works.) The following programs show how this is done. Recent versions of the Interspec have the top four bits held LOW, so LET X = IN 63 gives a value between 0 and 15. For this version omit the -240 from the programs.

Example 7 Switch indicator

```
 10  REM SWITCH INDICATOR
 20  REM Define inputs and outputs
 30  LET inputs = 63
 40  LET outputs = 63
100  REM Collect switch inputs
110  PAUSE 1:LET status = (IN inputs) - 240
120  REM Send to outputs
130  OUT outputs,status
500  GO TO 100
```

The first program in this section shows how the state of each line can be echoed to LEDs connected to the relay outputs.

Example 8 Selective indicator

```
 10  REM SELECTIVE INDICATOR
 20  REM Define inputs and outputs
 30  LET inputs = 63
 40  LET outputs = 63
 50  LET on = 1
 60  LET off = 0
100  REM Collect switch inputs
110  PAUSE1:LET status = (IN inputs) - 240
120  LET A = off
130  LET B = off
140  LET C = off
150  LET D = off
160  IF status > 7 THEN LET status = status - 8 : LET D = on
170  IF status > 3 THEN LET status = status - 4 : LET C = on
180  IF status > 1 THEN LET status = status - 2 : LET B = on
160  IF status > 0 THEN LET A = on
200  REM Send to outputs
210  LET W = A
220  LET X = B
230  LET Y = C
240  LET Z = D
```



```
250  LET result = 8*Z + 4*Y + 2*X + W
260  OUT outputs,result
500  GO TO 100
```

Example 7 does not allow individual switches to be inspected. Spectrum BASIC is rather clumsy in this respect, but Example 8 does do this task satisfactorily.

Example 9 Logic maker

```
 10  REM LOGIC MAKER
 20  REM Define inputs and outputs
 30  LET inputs = 63
 40  LET outputs = 63
 50  LET on = 1
 60  LET off = 0
100  REM Collect switch inputs
110  PAUSE 1:LET status = (IN inputs) - 240
120  LET A = off
130  LET B = off
140  LET C = off
150  LET D = off
160  IF status > 7 THEN LET status = status - 8 : LET D = on
170  IF status > 3 THEN LET status = status - 4 : LET C = on
180  IF status > 1 THEN LET status = status - 2 : LET B = on
190  IF status > 0 THEN LET A = on
200  REM Send to outputs
210  LET W = A AND B
220  LET X = A OR B
230  LET Y = NOT(C AND D)
240  LET Z = NOT(C OR D)
250  LET results = 8*Z + 4*Y + 2*X + W
260  OUT outputs,result
500  GO TO 100
```

This program shows how the switch inputs may be combined to affect the outputs in different ways.

Output W is the AND combination of A and B.
Output X is the OR combination of A and B.
Output Y is the NAND combination of C and D.
Output Z is the NOR combination of C and D.

This program has simulated a hard-wired board with the gates shown in Figure 4.5. Clearly these gates may be changed to give any combination required. The Spectrum is behaving like a programmable logic gate, which is the more modern way of using electronics. Try out some logical combinations of your own by changing lines 210 to 240 of Example 9.

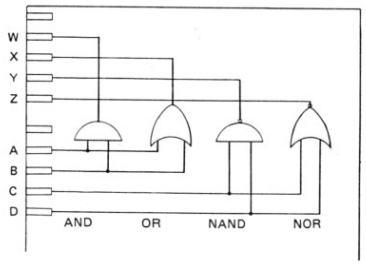


Figure 4.5 Simulated logic on the logic board

Sensing and controlling the environment

We have seen how to connect simple switches and LEDs to the Interspec to simulate the real world, but we are not limited to these. The switch inputs can be connected to different devices, such as photocells, trip switches, water-level indicators, temperature switches and the like. The outputs can be connected to motors, lamp indicators, heaters, water valves and pumps. It is thus possible to produce an automatic washing machine with a suitable control program. The examples that follow are more simulations to demonstrate how the inputs and outputs are used together in such control systems.

Example 10 Pedestrian crossing

```
  1 REM PEDESTRIAN CROSSING
 10 REM Define inputs and outputs
 20 LET red = 4
 30 LET amber = 2
 40 LET green = 1
 50 LET wait = 8
 60 LET outputs = 63
 70 LET inputs = 63
 80 LET off = 0
100 REM set lights at green and wait for pedestrian request
110 OUT outputs,green
120 PAUSE 1:LET status = IN inputs
130 PAUSE 1:IF status = IN inputs THEN GO TO 130
140 REM input status has changed
150 REM traffic lights sequence
```



```
160 OUT outputs,(green + wait)
170 PAUSE 150
180 OUT relays,(amber + wait)
190 PAUSE 100
200 OUT relays,red
210 PAUSE 250
220 REM flash amber
230 FOR c = 1 TO 10
240 OUT outputs,amber
250 PAUSE 20
260 OUT outputs,off
270 PAUSE 20
280 NEXT c
500 GO TO 100
```

This program shows how the switch inputs of the Interspec can be used to send information to the microcomputer. This program uses the same red, amber and green lights as for TRAFFIC LIGHTS, but adds another, connected to output 3, which simulates the wait lamp on a pedestrian crossing. The pedestrian request button of the crossing is simulated by momentarily closing (or opening) a switch connected to any of the switch inputs. This simple arrangement allows for the pedestrian to request the traffic to stop, upon which, the traffic lights go through their sequence to red and the wait lamp then goes out to indicate that it is safe to cross. This program could include a BEEP produced by the Spectrum's own speaker, but this is left for you to add.

Example 11 Burglar alarm

```
  1 REM BURGLAR ALARM
 10 REM Define inputs and outputs
 20 LET outputs = 63
 30 LET inputs = 63
 40 LET off = 0
 50 LET on = 1
100 CLS
110 PRINT AT 0,8;"BURGLAR ALARM"
120 PRINT AT 3,0;"Press 'c' to set the system."
130 IF INKEY$<>"c" THEN GO TO 130
140 PRINT AT 6,0;"There will now be a short delay"
150 PRINT AT 8,0;"to give you time"
160 PRINT AT 10,0;"to get out of the house."
170 PAUSE 250
180 LET status = IN inputs
190 PRINT AT 15,0;"The alarm is now set."
200 PRINT AT 17,0;"Cross the light beam"
210 PRINT AT 19,0;"to set off the alarm."
```

```
220  PAUSE 1:IF status = IN inputs THEN GO TO 220
230  CLS:PRINT AT 1,0;"The photocell has been crossed."
240  PRINT AT 3,0;"There will be a short delay"
250  PRINT AT 5,0;"to allow the owner to switch"
260  PRINT AT 7,0;"the alarm off before it sounds."
270  PAUSE 250
280  IF INKEY$<>" " THEN GOTO 100
290  REM sound the alarm
300  FOR c = 1 TO 20
310  OUT outputs,on
320  PAUSE 10
330  OUT outputs,off
340  PAUSE 20
350  NEXT c
360  GO TO 100
```

One of the traditional circuits for teaching about electronics is the burglar alarm. The simple version of this suffers from all sorts of drawbacks. How does the owner of the house get into it, or even out of it, without triggering off the alarm? Example 11 is an alarm routine, which includes these extra features. Again sound can be added if required.

Teaching electronic logic

Example 9 above indicated how the microcomputer might be used to simulate a **programmable logic array** (**PLA**). This idea can easily be extended to the teaching of electronic logic. In *Microelectronics* I proposed that one of the best ways of introducing microelectronics is with a microcomputer, this view will now be justified. For the following experiments a logic board should be connected to the Interspec. This board can be the four-input logic board described previously, although only three inputs are used. Better still, the Griffin programmable logic board can be used (Figure 4.6).

These boards connect directly to the switch-input and relay output ports. The power supply for the LEDs comes from the microcomputer itself. From the point of view of the user, the logic board consists of three input sockets and four LEDs to indicate the logic state of the outputs. It can be used by the microcomputer to simulate each of the standard logic gates and digital electronic circuits encountered in elementary electronics. Once the board has been connected to the microcomputer, LOGIC (1) should be loaded and run. It works in the following way.

LOGIC presents a menu of options, most of which are self-explanatory. As an example, choose the first option (Two-input logic gates). This option then asks which logic gate is to be simulated (the choice is AND, OR, NOT, NAND, NOR, EXCLUSIVE-OR or EQUIVALENCE). After the selection is made (by pressing one of keys 1 to 7) the screen displays a diagram of the board, indicates the current

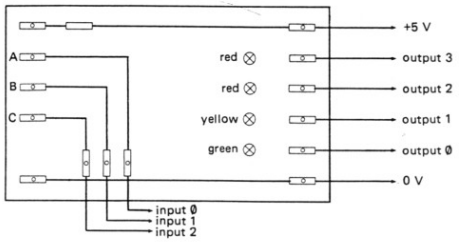


Figure 4.6 The three-input programmable logic board

logic states of the inputs and the output, displays the appropriate truth table and highlights the particular line of this truth table which is currently being implemented (Plate 11).

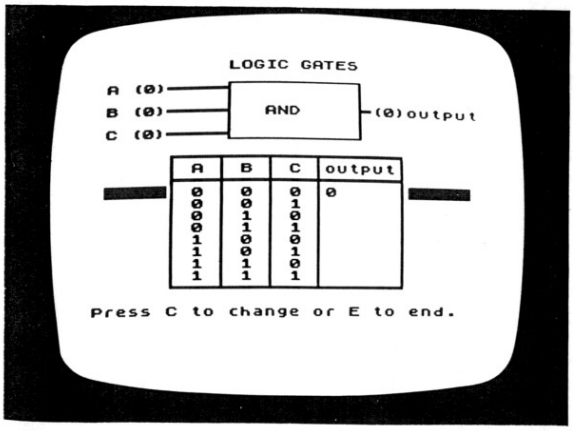


Plate 11 Logic gates

The logic board has three input terminals labelled A, B and C and four output terminals labelled W, X, Y and Z, which are connected to LED indicators to show their logic state. When a terminal is HIGH, its LED is on, when a terminal is LOW, its LED is off. The input logic levels can be changed by connecting them to the 5 V terminals (red), which makes them go HIGH, or they may be connected to the black 0 V terminals, which makes them go LOW. Unconnected inputs are LOW, which is not the normal condition for TTL devices but is easier for beginners. When the logic level of either input is changed, the display also changes accordingly.

Other options in LOGIC allow the logic board to become a shift register, binary counter, J-K bistable, two-to-four decoder or to simulate three-input logic gates. The program has been found to give a good introduction to the principles of digital electronics. It also illustrates the way that a programmable device, like a microcomputer, can be used to produce different logic functions under the control of a program. In the same way that I deplore the unnecessary use of computer simulations in science instead of doing the actual experiment itself, some teachers may be horrified at the replacement of 'real' chips by a microcomputer. I think the latter is justified, because 'chips' are not an essential part of electronics (whereas science is not science without practical experimentation). When teaching electronics in school, our aim is to introduce the concepts of logic gates and binary counters; we are not too bothered whether such devices are implemented with CMOS or TTL integrated circuits, with transistors or even with relays. In a sense the computer is behaving like an uncommitted logic array, which is probably the best approximation schools can get to current industrial practice.

After investigating the different types of logic gate, students can be asked to run LOGIC TEST (2), which tests their understanding. The computer chooses a particular logic gate from a set of ten options and sets up the two-input board to simulate this gate. The student is invited to alter the input logic levels and observe the LED output and so determine its truth table. He or she then guesses which gate is being simulated and the computer marks this response. The program demonstrates a further use of microcomputers in the laboratory, to monitor the understanding of a practical activity. I am sure that this has applications in very many more areas than the one chosen as an example.

Electronic logic is concerned with the solution of problems that require different things to happen depending upon the input conditions. The burglar alarm and pedestrian crossing programs are good examples. We used to build these systems using integrated circuits. It is useful now to see what difference the microprocessor has made. Before the invention of the microprocesor, in order to make a new electronic system, an engineer would have to design a new circuit. It was most unlikely that new components could just be added on to a previous circuit, so the whole system would have to be re-made from the beginning. This is how digital systems were built in the 1960s and 70s, from combinations of separate integrated circuits. They were all wired together in the correct way to produce the desired function. Even if the system was sold in large numbers, each

one had still to be built up separately on a printed circuit board, so that the different gates could be correctly wired together.

The microprocessor changes this, because the same hardware can be made to do different things merely by changing its program. The same microprocessor can thus be made to do many different things, from shearing sheep to controlling a power station, making a teddy bear speak or running a microcomputer or even video games. Because it is the same microprocessor in each case, a very large number of them can be produced very cheaply.

With integrated circuits different Boolean functions are made by connecting NAND gates together. Each function is made by combining the gates in a different way. The advantage of a programmable system is that the same circuit can be used to produce these different functions, under the control of the program. The 'User-defined logic' option in LOGIC behaves similarly allowing each of the four outputs to become different Boolean functions of the inputs. LOGIC MAKER (Example 9) behaves in the same way, allowing you to create your own Boolean functions. To do this, stop the program and LIST lines 200 to 500. You can create any function of your own, provided it conforms to the syntax rules of BASIC and the ways already described for writing out Boolean functions. After changing the program, re-run it to execute with your new function.

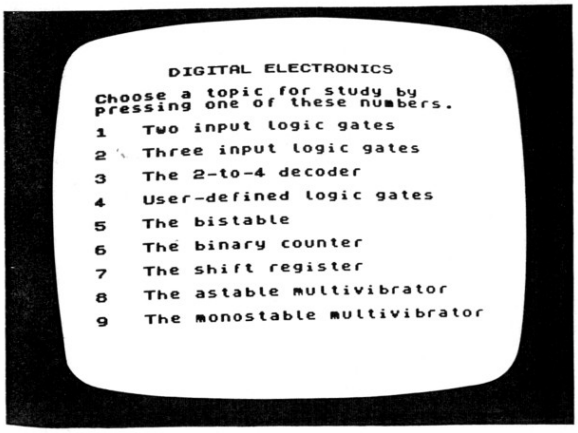


Plate 12 Digital electronics simulation

For example:

```
210  LET Z = (NOT A OR B)
210  LET Z = NOT(NOT A AND NOT B)
```

The variables should be A, B or C but you will not have to declare beforehand which you have used. The final outputs should be W, X, Y or Z. It is possible to use other variables, although you will not be able to find out what values they take. For example:

```
210  LET T = NOT A AND B
211  LET S = NOT B AND A
213  LET Z = T OR S
```

This example also shows that it is possible to put in more than one line for the function, provided it does not have to work backwards. That is, you cannot put

```
210  LET Z = NOT T
211  LET T = NOT B OR A
```

because T does not have its correct value in line 211 until after line 210 has been executed. This causes a 'no such variable' message to appear. A few more examples are given below, but the fun in this program is to create your own functions and then see what you have produced. Do this by stepping through the truth table and noting the outputs in each case.

```
210  LET Z = NOT (A OR B)
210  LET Z = NOT (NOT A AND NOT B)
210  LET Z = NOT (A AND B)
210  LET Z = (NOT A AND B) OR (A AND NOT B)
210  LET Z  (A AND B) OR (NOT A AND NOT B)
```

Timing

One most important application of the digital interface is timing, particularly for mechanics experiments. The Spectrum can be made to measure the time interval between two logic-level changes at the switch inputs. These status changes can be caused by switches or, more importantly, by photocells. The photocells can then be operated as cards mounted on trolleys cross in front of them.

A simple photocell can be made by connecting a light-dependent resistor between a switch input and the 5 V socket (red). This will, however, be too slow for serious timing and a photodiode is essential. Two methods of connecting a photodiode to a switch input are shown in Figures 4.7 and 4.8. It is important to use some form of input buffer, like this, to prevent switching oscillations from producing spurious pulses. On the other hand, such methods work by making the switch-on level different from the switch-off level (a procedure known as

hysteresis). This means that the light intensity required to switch the photocell on is different from that needed to switch it off. This, in turn, means that the apparent length of a card, passing in front of the photocell, is not its real length and this introduces errors.

The best arrangement is to get the smallest width of light beam falling on the photocell. This reduces errors due to the photocell switching on in a different place from where it switches off as the card crosses in front of it. This chapter assumes that two photocells are available, one connected to input 0 and the other to input 1 of the switch inputs. For some programs only one of these is needed.

The changes at the inputs are used to measure time intervals in the following way. Inside the microcomputer a 'clock' is first set ticking away in some unit of time. The switch inputs are read and stored in a memory location called **status**. The current state of the inputs is then monitored continuously and compared

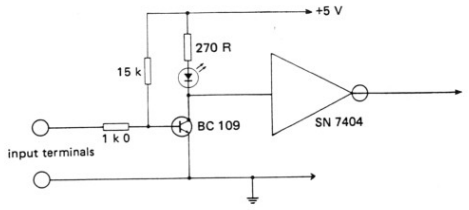


Figure 4.7 Connecting a photocell via a transistor circuit

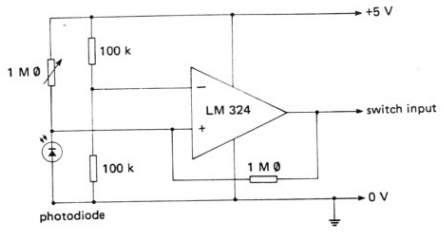


Figure 4.8 Connecting a photocell via an op. amp.

with status. Normally it will be the same, but when it is different, this is because one or other of the photocells has been activated. The contents of the clock are then noted. When the timing is finished, the time intervals involved can be calculated and displayed.

There are two ways of achieving the clock. The first is to make use of the Spectrum's own 50 Hz clock. The technique is to set this clock to zero when a photocell is first triggered and to read it when the next 'event' occurs.

Example 12 A simple timer

```
  1 REM SIMPLE TIMER
 10 REM Define inputs
 20 LET inputs = 63
 30 PAUSE 1:LET status = IN inputs
100 REM wait for input status to change
110 PAUSE 1:IF status = IN inputs THEN GOTO 110
120 PAUSE 1:LET status = IN inputs
130 REM reset clock
140 POKE 23672,0
150 POKE 23673,0
160 POKE 23674,0
170 REM wait for input status to change again
180 PAUSE 1:IF status = IN inputs THEN GOTO 180
190 REM read clock
200 LET time1=65536*PEEK(23674)+256*PEEK(23673)+PEEK(23672)
210 LET time2=65536*PEEK(23674)+256*PEEK(23673)+PEEK(23672)
220 IF time1 > time2 THEN LET time2 = time1
230 PRINT "Time taken = ";time2/50
```

This program assumes that switch inputs 2 and 3 are not used and are therefore LOW. The status of the input port is monitored continuously and when it changes for a second time, the number of elapsed fiftieths of a second is calculated. This is done twice since it is possible to get a wrong time otherwise (see page 130 of the Spectrum manual).

For the majority of timing purposes, BASIC is too slow. For example, in mechanics experiments the card mounted on the air-track trolley crosses the photocell in a few hundredths of a second. This requires machine code timing routines, which are discussed later. For the moment we shall just make use of these routines, without discussing them in detail.

REACTION TIMER (6) accepts input from the keyboard. After the screen clears, the SPACE key should be pressed and the reaction time of the operator will then be displayed on the screen in large digits. The Spectrum's own 50 Hz clock is used to measure this time interval.

STOPCLOCK (7) also uses the Spectrum's 50 Hz clock in a similar way. However, it now displays the elapsed time since the start of the timing sequence. Calculating and displaying the figures in large digits takes too much time in

Plate 13 STOPCLOCK

BASIC, so it is handled in machine code. Any change in logic level at input 0 or input 1 will start the stopclock and any other change in either logic level will stop it, leaving the elapsed time displayed on the screen. The SPACE key will halt the display temporarily without stopping the internal clock at the same time (a lap facility).

This program has many applications. It can replace a centisecond timer in most instances. The usual problems over 'make to start, break to stop' are avoided, since the routine detects any change at input 0 or 1.

FAST TIMER (8) illustrates another method of timing – counting machine-code cycles. The Z80A microprocessor in the Spectrum is itself under the control of a crystal oscillator, which produces clock pulses at a frequency of 3.5 MHz. Each machine code operation of the microprocessor requires a given number of such clock pulses. These can be counted, thus giving a measured time interval. Because the microprocessor is so fast, very short time intervals can be measured with this technique, in fact down to fractions of a millisecond. One problem with the method is that it requires a knowledge of machine-code programming to understand how it works and this is not an easy subject. However, the timing routine used in this program is of universal application and can be used in other

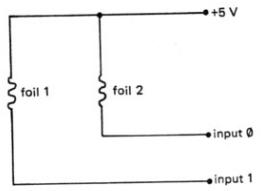


Figure 4.9 Connecting foils to the switch inputs

programs without knowing how it works. It is called from within a BASIC program with the USR instruction and after the timing is over, command is returned to BASIC. To allow for easy use by beginners, this routine is placed at the end of the program, where it can be merged with any other program requiring it. This and other programs also use a large digits routine to display numbers that can be read by the whole class. This routine is placed at the start of each program using it. Its use is described more fully in Chapter 7.

Applications

i) Speed of a rifle pellet Inputs 0 and 1 should be connected HIGH through the thin pieces of foil as in Figure 4.9. When the pellet breaks the first foil, the clock starts and when it breaks the second foil, the clock stops. The program will then note the elapsed time and display it in large digits on the video screen.

ii) Contact bounce Some idea of the speed of the timing routine can be gained by using a single push-button switch connected to one of the inputs 0 or 1. Run the program and when the display asks for an input, press the switch once. In most instances the program will display a result, indicating that at least two input changes have been detected. There were probably many more changes than this caused by the contact bounce in the switch when it is closed. FAST TIMER is fast enough to measure this contact-bounce time.

iii Switchover time Using the program with a two-way switch, as indicated in Figure 4.10, enables the changeover time of this switch to be measured. An interesting experiment is to see if the switchover time is dependent upon the speed at which the toggle is operated.

iv) Camera shutter speed Instead of switches to produce changes in the input status, this can also be done by the interruption of a beam of light focussed on a photocell, with the photocell connected to input 0 or 1. It then becomes possible to measure the effective shutter 'speed' of a camera. The photocell should be

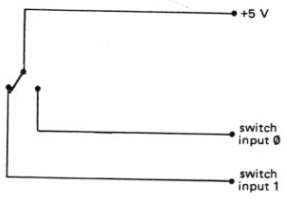


Figure 4.10 Connecting a two-way switch

mounted inside the camera at the image of an external light source. When the camera is operated, the time measured by the FAST TIMER program is a good indication of the exposure time that the film receives.

v) Trolley speed measurement If a card attached to a trolley crosses a light beam focussed on the photocell, the time taken for it to do so is measured by the routine and displayed. In this instance both changes take place at the same input. If the length of the card is measured too, the speed of the trolley can then be calculated (or the Spectrum can automatically compute the speed if it is programmed to do so).

Note that this program cannot be used with two photocells connected to separate inputs. This would be very useful, since the speed of the card could then be measured over a large distance. Unfortunately, as the card crosses the first photocell, it starts and then stops the clock at this point. A more sophisticated timing routine is needed to measure the time between two different photocells.

Advanced timing

The advanced-timing routine of the following programs needs some explanation. To enable multiple measurements of speed for studying the law of conservation of momentum, there can be two photocells. Furthermore, in the latter experiment, it is possible for a second trolley to begin a transit of its photocell before the first has finished crossing the other photocell. Thus it must be possible to detect the two inputs independently and to keep their results separate. We still only need the one clock, but at the start or finish of a transit, the time on the clock is copied into a store. In fact up to thirty-two stores are available for each input. Thus, in the conservation of momentum experiment, it is possible to have two trolleys approach from different directions, to collide in the middle and both go off in one particular direction at different speeds. This involves two events at one input and six events at the other, but the routine can easily cope with this. (An **event** is any change in logic level at either of the inputs.)

This advanced-timing routine can be called from a BASIC program in a variety

Plate 14 Large-digit display

of ways, to measure time and speed as above and also to measure period, frequency and acceleration. All measurements are displayed in large digits on the screen using a large-digit display machine-code routine in the manner described in Chapter 7. The advanced-timing routine is used as follows. The number of events to be recorded is first placed in location 64247. For example, the timing of the single pass of a card in front of a photocell requires two events, one to start the clock and one to stop it. The measurement of acceleration with a double card requires four events, two for each half of the card. The routine is called with USR 64000, which then waits for the photocell to be crossed the requisite number of times. Each time that such an event occurs, the current time, measured by an internal clock, is copied into a fresh set of three bytes. When all events have been recorded, or if the SPACE key on the keyboard is pressed, the timing routine hands back control to the BASIC program.

If this return occurs after a proper timing sequence, the recorded times are first converted into time intervals. The intervals recorded by the photocell connected to switch input 0 are stored in locations 64256 to 64383. The intervals recorded by the photocell connected to switch input 1 are stored in locations 64384 to 64511. For most purposes, only the first few such locations will contain data. The BASIC

subroutine at line 4000 of TIME, SPEED and ACCELERATION METER (program 9 – called TSA for short) collects the measured time intervals from these locations and converts them into seconds. A study of programs 9 to 12 will reveal exactly how this routine is used.

Speed and acceleration measurements with TSA are based upon the photocell technique using a card length of 40 mm. By changing lines 650 and 752 of the program, this may be converted to any other length. However, there is considerable inaccuracy introduced by the photocells, because the point at which they switch on is not necessarily the same point at which they switch off. So a 40 mm card may not necessarily look like a 40 mm card to the photocell. The error is only a few mm, and this is only important if very short cards are being used. If great accuracy is desired, then 100 mm cards or longer should be used. The advantage of short cards is that some meaning can be given to the difficult concept of instantaneous velocity.

The double card shown in Figure 4.11 enables acceleration to be determined and displayed directly. This quantity is computed from the standard equation

acceleration = (final speed - initial speed) / time taken

The double card provides for the two measurements of speed required in the calculation. Only the 40 mm lengths of the card are critical, the distance between them is not. If different lengths are used for this double card, then line 752 of the program should be changed.

An interesting experiment is simply to drop this double card vertically in front of a photocell, while TSA is being run. The screen will display the acceleration due to gravity directly (but see the educational note below).

By connecting two photocells in series, they can be placed any distance apart, and then a single card can pass in front of both photocells to provide the initial and final speeds for this calculation. This would be a good way to introduce the function of the double card.

TSA was not designed to explain how the microcomputer is making its measurement, so ACCELERATION TUTOR (10) attempts to remedy that situation. It is designed to be used with a trolley, carrying a single 40 mm card, running down an inclined place. On the way it crosses in front of two photocells connected in series to one of the switch inputs. The time taken for the card to cross each photocell can be displayed and used to determine the speed of the trolley in two different places. The time taken for the trolley to get from one photocell to

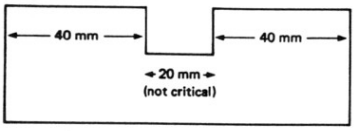


Figure 4.11 The double card for measurement of acceleration

the other can also be displayed, so that pupils can calculate the acceleration of the trolley too. These values can also be calculated automatically by the program and displayed for confirmation. Finally, the distance between the photocells can be measured, thus enabling all of the common kinematic equations to be investigated.

The advanced timing routine is also designed for measuring the speeds resulting from trolley collisions. It is incorporated into CONSERVATION OF MOMENTUM (11). The same considerations about card lengths apply as above. The speeds are displayed for each photocell separately, with the readings in chronological order for each separate channel. In this case there should be two photocells, one connected to each input. The display will first list the speeds recorded at switch input 0 (in the chronological order that they were measured). Then the speeds measured at switch input 1 will be shown (again in chronological order). Which of these represent the velocities before the collision must be determined from an observation of the experiment itself. Usually there is no confusion, so the initial and final momenta are easy to determine.

Using a 16-slot card the speed of a trolley in front of a photocell can be measured thirty-one times and distance-time and speed-time graphs can be plotted and displayed automatically (SPEED-TIME PLOTTER, 12). This program is useful for demonstrating the graphical relationships between distance, speed and acceleration.

Educational note

At this point a cautionary note must be made to discourage the over-zealous use of the microcomputer in the laboratory. The acceleration due to gravity experiment mentioned above can be carried out much more easily and accurately by the following program.

```
100  PRINT "Acceleration due to gravity = ";
110  PRINT "9.81 m/s2"
```

This is not, of course, a measurement, but to a pupil who does not know how a microcomputer works, it is no less valid than the method described in TSA! It is essential that pupils understand what the microcomputer is doing, when it is taking measurements. This does not mean that pupils understand in the sense that they should know about programming and interfacing; that is clearly impracticable. What is needed is a demonstration that the microcomputer is giving the same results that could have been obtained by other, more long winded, methods. The teaching sequence could be as follows:

i) Show the microcomputer as a measurer of time by getting pupils to press a switch for an estimated ten seconds, say, using the simple timer (Example 12). Then show the microcomputer being used as a stopwatch.
ii) Show the microcomputer as a measurer of short time intervals, using REACTION TIMER (6).

iii) Measure the time of transit of a card in front of a photocell using TSA. Use calculators to determine the speed of this card and then show that the microcomputer can carry out the same calculations automatically.
iv) Having shown how the microcomputer can calculate speed, allow it to measure the speed of a trolley at several different places as it runs down an inclined plane. ACCELERATION TUTOR (10) allows this to be done, so that the times of transit of the cards and the time intervals between these transits can be measured. Pupils can again use their calculators to determine the acceleration of the trolley.
v) The principle of the double card should now be apparent – the microcomputer is measuring three time intervals and using them to compute the acceleration of the card. The acceleration due to gravity experiment can now be understood.
vi) TSA can now be used to demonstrate Newton's second law. Because acceleration is so easily measured, it is probable that pupils will get a better understanding of this law than they usually do with ticker-timer measurements of acceleration.
vii) Conservation of momentum experiments are now more easily carried out, because it is no longer necessary to use stroboscopic techniques to measure the speeds of the colliding trolleys. Nor is it necessary to restrict the experiments to perfectly elastic or perfectly inelastic collisions.

At all times the teacher must be wary of using the microcomputer 'because it is there'. It must offer a clear advantage over the conventional ways of teaching before its use can be justified. The teaching of motion is an example of its advantage; the measurement of time in hours and minutes just to display it as an analogue clock on the video screen is a gimmick, there are better ways of doing this. A microcomputer should not be used for such purposes.

Frequency measurement

The computer could be used to measure the input changes caused by an alternating input voltage and use these to compute and display its frequency. This would be accurate up to about 1 kHz. However, a purpose-built program would allow higher frequencies to be measured.

The Z80 microprocessor has a very useful input-repeat facility, which allows even short pulse lengths to be measured, thus producing a frequency meter for square wave inputs. These can be obtained from sinusoidal waves with a suitable squaring circuit. The alternating voltage to be measured is fed to one of the inputs through this circuit. With time interval measurements of about 5 microseconds, the maximum frequency that can be measured to an accuracy of 5 per cent is thus about 10 kHz. FREQUENCY METER (13) is thus still of little use at the higher frequencies, but it is invaluable for experiments in sound, for example, in measuring the frequency of a sonometer wire.

The lower frequencies of a pendulum are more easily measured with the

advanced timer routine. A 40 mm card is fixed to the pendulum, which then swings in front of the photocell. The time for one complete swing (five events) is recorded and displayed as the measured period by the program PENDULUM (14).

Another aspect of timing is producing output pulses, for example, to take single film shots every five seconds for time-lapse photography. Example 6 showed this technique using the PAUSE instruction to measure the time interval. Faster pulses above say 10 Hz require a machine-code routine, such as that in PULSER (15). The length of each pulse and the time interval between pulses may be altered, with pulses up to frequencies of 50 kHz possible. The machine code routine for this program is discussed in Chapter 8.

Hardware details

Other digital features of the Interspec will now be discussed. The Interspec connects to the spectrum via the usual gold-plated edge connector. If required, the Interspec may be fitted to the ZX printer instead, after the latter has been plugged into the Spectrum.

The addresses of the Interspec ports have been chosen to avoid conflict with the Spectrum's own hardware. They are as follows:

Address	Command	Operation
31	OUT 31,a	Select ADC channel (a = 0 to 7)
31	IN 31	Read selected ADC channel
63	OUT 63,a	Relay outputs
63	IN 63	Switch inputs
95	OUT 95,a	Write to TTL output port
95	IN 95	Read TTL input port

Switch inputs

As supplied, the Interspec has its switch inputs connected to the 0 V line through 2.2 kilohm resistors. The adjacent red socket (common) is connected to the positive 5 V line. Thus these inputs are normally LOW and may be sent HIGH by connecting them to some external device or to the red socket. This may be undesirable for some applications (for example, if driving the switch inputs from TTL integrated circuits), so it is possible to alter this as shown in the Griffin I-pack manual.

TTL ports

In addition to the switch inputs and the relay outputs so far described, the Interspec contains an 8-bit input port and an 8-bit output port (Figure 4.12). These are TTL compatible and are an alternative to changing to the option for the switch inputs mentioned above. The TTL ports are very useful for driving other electronic equipment, such as keyboards, digital displays and more extensive

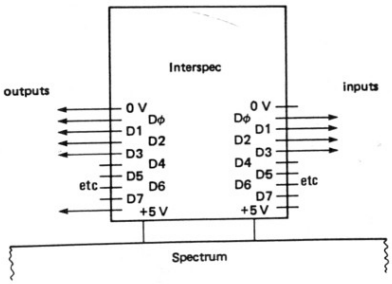


Figure 4.12 TTL ports

logic boards. This, therefore, takes the place of the user port found on other microcomputers. These ports share the same address (95), so that writing to the output port is achieved with OUT 95,n, and the input port is read with LET X = IN 95. Note that all lines of the input port are normally HIGH, unless pulled LOW by some external device. Note too, that the output port cannot provide current for driving external devices; it can 'sink' the usual 8 mA per bit (subject to an overall maximum of 50 mA), but it will not 'source' more than 100 microamps. Connections to external devices should therefore be made through suitable buffers. The most useful of these is an 8-input Darlington driver array (Figure 4.13), which can be connected directly to more relays or other high current devices or to a set of LEDs. It is now possible to get seven or more such drivers on a single chip (RS Components 307–109). This can provide currents up to 500 mA with driving voltages up to 50 V.

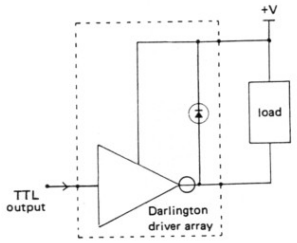


Figure 4.13 Darlington driver array

One useful display that might use this device is a seven-segment display. By switching certain bits of the output port on, different digits can be displayed in this way. An interesting project is a program to display these digits and as many letters of the alphabet as possible.

The inputs can be driven by any TTL compatible voltage levels, such as from switches or photocells as described earlier for the switch inputs. In conjunction with the output port, the inputs could also be used to monitor a simple hexadecimal keyboard (Figure 4.14). Each row of the keyboard is pulled LOW in turn and the columns are scanned to see if any of them have gone LOW. If so, then the key at the intersection of that particular row and column is being pressed. This too could form a suitable programming project for a student looking for a practical problem to solve.

Connection to these input and output ports is best made with ten-pin Molex plug connectors (RS Components 467–582), although if absolutely necessary bared solid-cored wires can be used. The 5 V and 0 V needed for small external circuits can also be obtained through these ports, although care should be taken not to overload the Spectrum's power supply.

DCP bus

The 15-way socket at the rear of the Interspec is provided for further circuits. One of these is a digital to analogue converter, described in Chapter 5. It is quite easy for the user to build his or her own circuits. The DCP bus provides all eight data lines to the Spectrum, the 0 V, 5 V (regulated) and 9 V (unregulated) power supplies and four device-select lines – NWR, NRD, NDCPAD1 and NDCPAD2.

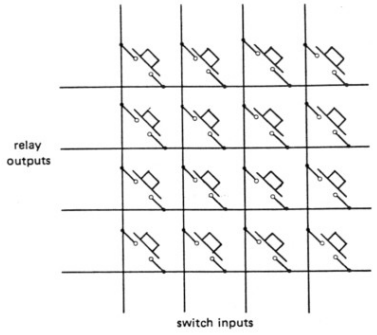


Figure 4.14 Monitoring a keyboard

NWR and NRD are the write and read signals from the computer. Whenever the latter performs an IN instruction, the NRD line is pulled LOW. If the OUT instruction is executed then NWR is pulled LOW. NDCPAD1 is pulled LOW when IN 127 or OUT 127,n is executed. NDCPAD2 is pulled LOW for IN 159 or OUT 159. These signals may thus be ORed together to provide more input and output ports, which are then addressed as follows:

Address	Command	Operation
127	OUT 127,a	Send data to port A
127	IN 127	Input data from port B
159	OUT 159,a	Send data to port C
159	IN 159	Input data from port D

For those with some knowledge of digital electronics, the DCP bus is very useful for exploring the world of microelectronics. For most purposes the Spectrum can take the place of simple microprocessor boards, like Microprofessor. Because BASIC is readily available, students can be introduced to microelectronics more gently.

One example of this is to produce an eight-digit, seven-segment display (similar to a calculator). One output port provides the lines to the segments, while the other pulls down each digit select line in turn. A full eight-digit number may thus be displayed. With a suitable program, this could even remove the need for a TV set to be connected to the Spectrum, so that it could become a dedicated laboratory instrument. Plans are afoot for a whole range of add-on equipment, that can be plugged into the Interspec, the possibilities for which are endless.

5 Analogue interfacing

'One side will make you grow taller, and the other side will make you grow shorter.'
(Lewis Carroll, *Alice's Adventures in Wonderland*)

Interfacing is the general name given to all connections between the microcomputer and other equipment. In Chapter 4 we looked at ways of enabling the Spectrum to monitor and control the outside world using digital interfacing. This chapter extends these ideas to analogue input and output too, showing how their use turns a microcomputer into a general purpose laboratory instrument. It would be more usual to begin this discussion with a digital to analogue converter, but this is not a standard part of the Interspec. Consequently I shall begin with analogue to digital conversion.

Analogue to digital conversion

As we have seen a microcomputer uses two-state electronic devices to carry out all of its different functions, which is why it is possible for a microcomputer to detect whether an external sensor is HIGH or LOW. However, if the microcomputer is required to measure a voltage, that voltage may have any value within a given range, it will not be limited to these two HIGH or LOW states. This problem is overcome by changing the voltage being measured into a binary number, which can have one of 256 discrete values. The analogue inputs of the Interspec can convert a voltage in the range 0 to 2.5 volts into a binary number from 0000 0000 to 1111 1111, with an accuracy of about 5 mV. All eight bits of this binary number are read by the microcomputer simultaneously.

Each of these eight bits is digital, because it can only be switched HIGH or LOW. The voltage being measured is an analogue, because it is able to take a whole range of values within certain limits. Thus the interface uses an **analogue to digital converter (ADC)** to change the analogue voltage into the eight digital signals required by the microcomputer.

The Interspec contains eight analogue inputs (or channels), but only one ADC, so it can only convert one of these input voltages at a time. This is done with a switching system, selected by a number sent to the Interspec. Any particular input is first selected by the instruction OUT 31,x, where x is the desired channel (number from 0 to 7). This starts the conversion process and 100 microseconds later, the binary equivalent of this voltage is available for reading. This time

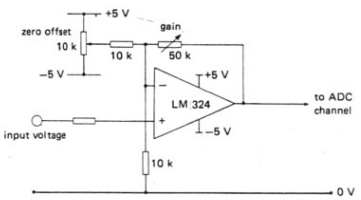


Figure 5.1 Voltage adjuster

interval is less than that taken by BASIC to get from one statement to the next, so the reading can occur immediately after selecting the channel. This is accomplished with the instruction LET V = IN 31, which sets the value of V to some integer between 0 and 255, depending upon the voltage being measured. In binary, these are the numbers 0000 0000 (corresponding to a voltage of 0 V) and 1111 1111 (corresponding to a voltage of 2.5 V).

Owing to the erratic clock of the Spectrum, wrong data can often be read if IN 31 follows immediately after OUT 31,n. This can be avoided by a short delay between selecting the channel and reading the data, such as PAUSE 1. This is not necessary if programming in machine code, as Chapter 8 will demonstrate.

The analogue inputs allow the microcomputer to measure different voltages. But just to do this is rather pointless, a cheaper voltmeter will do the job just as well and with far less trouble. The microcomputer should be used in areas where a simple voltmeter is of no use. For example the speed of the microcomputer can be used to measure voltages several thousand times per second or to measure several different voltages repeatedly in rapid succession. The microcomputer memory can be used to store these voltage readings for later output to a cathode ray oscilloscope or to a chart recorder. The readings may also be listed on the microcomputer screen or presented graphically as a bar chart or a graph.

Before measuring any voltage, a check should be made to see that it is within the range 0 to 2.5 V. If it is not, then the ADC will simply return the saturation values of 255 or 0. If the voltage to be measured is greater than 2.5 V, it can be passed to a suitable voltage divider network to reduce it to the acceptable range. If it is too small, the voltage should first be amplified with a suitable op. amp. circuit. The circuit shown in Figure 5.1 may be used to do both jobs.

ADC calibration

```
 1  REM ADC CALIBRATION
10  PRINT AT 3,0;"Enter voltage as measured"
20  PRINT AT 5,0;"by the voltmeter."
30  INPUT Z
```

```
 40  PRINT AT 8,0;"Enter channel number "
 50  INPUT c
 60  IF c <> INT(c) OR c>7 OR c<0 THEN GO TO 50
 70  OUT 31,c
 80  PAUSE 1
 90  LET V = IN 31
100  LET convfactor = Z/V
110  PRINT AT 10,0;"The conversion factor for"
120  PRINT AT 12,0;"channel ";c;" is ";convfactor
```

This program should initially be used to calibrate the ADC, to allow for differing power supplies, etc. Connect the input terminal of a channel to a voltage of about 2 V and measure it with a good voltmeter. Run the program and make a note of the conversion factor produced. Check this for different voltages and different channels. If necessary calculate some compromise value that suits most circumstances.

Graphical display of voltage

```
  1  REM ADC GRAPH PLOT
 10  PRINT AT 0,5;"ADC GRAPH PLOT"
 20  PRINT AT 3,0;"Enter channel number "
 30  INPUT c
 40  IF c <> INT(c) OR c > 7 OR c < 0 THEN GO TO 30
 50  FOR x = 0 TO 255
 60  OUT 31,c
 70  PAUSE 1
 80  LET V = IN 31
 90  LET y = 0.7*V
100  PLOT x,y
110  NEXT x
```

The Spectrum is capable of producing high resolution graphs of the voltages it measures. In this program the voltage measurements are made continuously and plotted on the screen immediately (after scaling to ensure that the plotted points are on the screen). This example needs to be improved in several ways. Firstly, axes should be drawn and labelled. Secondly, the time interval between readings needs to be made variable. This allows readings to be taken at different data acquisition rates, from about 100 per second up to several minutes per reading in BASIC. This range may be extended with a machine-code routine to take the readings. This is achieved with STORAGE OSCILLOSCOPE (18). In this program, up to four channels may be monitored, provided one of these is channel 0. The time interval between successive readings may be adjusted in the range 100 microseconds to 25 milliseconds (10 000 readings per second to 40 per second). Slower rates than this can be handled by BASIC. STORAGE OSCILLOSCOPE

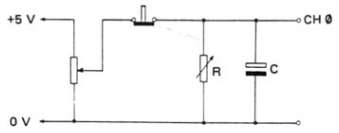


Figure 5.2 Capacitor discharge

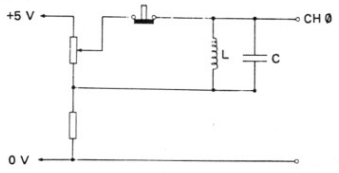


Figure 5.3 Bias voltage

uses a machine-code subroutine to collect the readings and stores them in successive memory locations, from where they are collected by the graph-plot routine. 250 voltage readings are collected from each channel and later plotted on the screen. At fast rates of measurement it is important for the microcomputer to know when to start taking readings. This is done by waiting until the voltage being monitored at channel 0 changes significantly, i.e. by about 80 mV, before beginning to take readings.

This program is especially valuable for studying transient phenomena, like the discharge of a capacitor through a resistor (Figure 5.2) or an inductor. In the latter case, though, the voltage can go negative and a bias voltage must be added to prevent this (Figure 5.3).

The same data acquisition routine can also be adapted to take 250 successive readings, which are later output to a chart recorder or a cathode-ray oscilloscope using adaptations of digital to analogue programs described later.

Current and resistance measurement

To measure current with the ADC, it should be allowed to flow through a known resistor and the voltage across that resistor measured by the ADC. If both the voltage across a component and the current flowing through it are measured at the same time, their product gives the power developed in the component. Similarly, the resistance of the component can be calculated and displayed. This

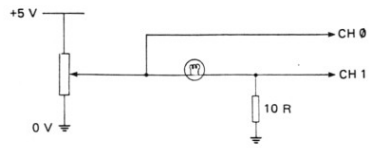


Figure 5.4 Resistance/power measurement

gives very effective demonstrations of the change of resistance of a lamp as it gets brighter. DIGITAL MULTIMETER (20) does this, displaying voltage, current, resistance or power in large digits. It requires a circuit like that given in Figure 5.4 (Plate 15).

Other measurements

Any other physical quantity that can be turned into a voltage can be measured by the ADC too, provided it is turned into a voltage within the correct range. Devices that turn other physical quantities into voltages are called **transducers** and there

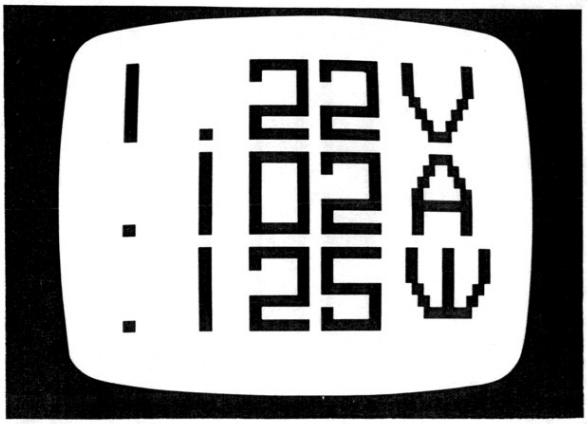


Plate 15 Digital multimeter

are a large number of these available. Here are some examples from the current RS Components catalogue:

RS stock no.	Measurement	Output range
308–809	Temperature	0 to 1 V
303–337	Pressure	0 to 75 mV
304–267	Magnetic field	0 to 400 mV
305–462	Light intensity	0 to 1 V

In addition there exist transducers to measure force, displacement, wind speed, humidity, oxygen content, acidity and sound intensity. The last of these is called a microphone! This illustrates the point that alternating voltages are easily turned into direct voltages using a.c. to d.c. converters. The latter can be a diode rectifier or the more expensive r.m.s. to d.c. converter (RS Components' AD536A). With this range of transducers, an ADC and a microcomputer most laboratories will need no other instrument.

Many useful devices convert some physical quantity into a change of resistance. Examples of this are the thermistor (which changes its resistance with temperature) and the light-dependent resistor. These devices can be turned into transducers by putting them into a voltage-divider network connected to the op. amp. circuit of Figure 5.1.

Another device in this category is the strain gauge, which converts the strain in a bar of metal into a voltage. Since strain is proportional to stress, this allows force and hence weight to be measured too. Also, by connecting a spring to the force transducer and an object to the other end of the spring, the displacement of this object may be measured too (replacing the metre rule?). Griffin and George produce a whole range of devices that can be connected to the Interspec, as well as a universal laboratory unit to convert voltages to the correct range required by the Interspec (called the Expand Pack).

Potentiometer

A potentiometer is a transducer too. It is particularly easy to connect a potentiometer to the Interspec analogue input (Figure 5.5). Four such potentiometers may be mounted on a board side by side to simulate a control panel. Pushbutton switches from the switch inputs, and LEDs from the relay outputs, may also be connected to make a control panel, enabling a range of industrial processes to be simulated (Figure 5.6). The simulation of Millikan's experiment

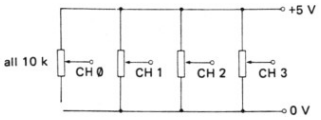


Figure 5.5 Potentiometer input

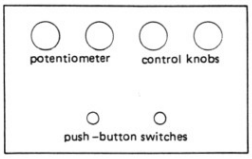


Figure 5.6 A simulated control panel

is much more satisfactory if voltages are entered via a control knob than by typing them in at the keyboard. This idea was suggested by M. Ryan and J. Stewart at the Dundee College National Course in 1982.

If two potentiometers are mounted perpendicularly the result is a joystick (RS Components 162-732). This allows the coordinates of a physical position (the knob of the joystick) to be plotted directly on the screen (which is what many video games are all about). The joystick is actually a displacement transducer, but with two-dimensional capabilities. A two-dimensional plotter based on this idea is as follows:

```
 10  REM ETCHASKETCHA
100  OUT 31,0
110  PAUSE 1
120  LET X = IN 31
130  OUT 31,1
140  PAUSE 1
150  LET Y = 0.7 * IN 31
160  DRAW X-PEEK 23677,Y-PEEK 23678
170  GOTO 100
```

XYPLOTTER (17) is a machine-code routine to carry out the same process very much faster. This program may thus be used with rapidly changing voltages, in much the same way as a cathode-ray oscilloscope when using its x plates.

Some devices do not produce values that are directly proportional to the quantity being measured. For example, a simple thermistor or LDR in a voltage-divider circuit gives an ADC reading that is related to the physical quantity but not in a linear way. If twenty degrees produces an ADC value of 100, then forty degrees will not produce an ADC value of 200. To obtain the true value (for temperature, etc.) a look-up data table needs to be created. The next example shows the general idea.

```
100  REM SET UP THE DATA TABLE
110  FOR I = 0 TO 15
120  READ T$(I)
130  NEXT I
140  DATA "OUT OF RANGE"
150  DATA "IMPOSSIBLE TO MEASURE"
160  DATA "IMPOSSIBLE TO MEASURE"
170  DATA "22 degrees C"
180  DATA "24 degrees C"
190  DATA "27 degrees C"
200  DATA "30 degrees C"
210  DATA "34 degrees C"
220  DATA "37 degrees C"
230  DATA "41 degrees C"
240  DATA "46 degrees C"
250  DATA "50 degrees C"
270  etc.
3000 REM CONVERT READING AND DISPLAY IT
3010 OUT 31,0
3020 PAUSE 1
3030 LET X = IN 31
3040 PRINT "THE TEMPERATURE IS ";T$(X)
3050 etc.
```

This program should obviously be expanded to 256 or more values to become sensible, otherwise a mercury thermometer is more accurate and easier to use. Care should always be taken not to use the microcomputer where an ordinary instrument does the job easier and more cheaply. The microcomputer is much more suited to areas where a simple instrument will not work. For example, the speed of the microcomputer can be used to measure voltages several thousand times per second or to measure several different voltages repeatedly in rapid succession. The microcomputer memory can be used to store these voltage readings for later output to a cathode ray oscilloscope or to a chart recorder. The readings may be listed on the microcomputer screen or presented graphically as a bar chart or a graph. From there they can be printed out for everyone to see using the COPY facility of the Spectrum.

Multi-channel measurements

```
  1  REM MULTI-CHANNEL DATA ACQUISITION
 10  PRINT AT 0,5;"DATA ACQUISITION"
 20  PRINT AT 2,0;"This program takes readings"
 30  PRINT AT 3,0;"every second at each channel.
 40  REM n = number of readings
 50  REM c = number of channels
 60  DIM V(255,7):REM stores for measured data
 70  FOR n = 0 TO 255
 80  FOR c = 0 TO 7
```

```
 90  OUT 31,c
100  PAUSE 1
110  LET V(n,c) = IN 31
120  PRINT AT (n*2 + 5,0);"
130  PRINT AT (n*2 + 5,0);"Channel ";n;" reading ";V(n,c)
140  NEXT c
150  PAUSE 40
160  NEXT n
```

The technique of taking a large number of readings and storing them is relatively simple. This program takes readings on all channels every second, printing them on the screen and storing them in arrays at the same time. (Users with the 16K machine will need to restrict the number of channels or readings taken to avoid an out of memory error.) Later the readings can be recalled, printed or plotted on the video screen, output to a chart recorder or otherwise processed. Machine-code routines for doing the same at faster data-acquisition rates are discussed in Chapter 8.

The Spectrum screen is arranged so that it can be easily scrolled from side to side. This is discussed in Chapter 7. If the measured voltages are plotted on the extreme right of the screen at the same time, the effect is like a four-channel chart

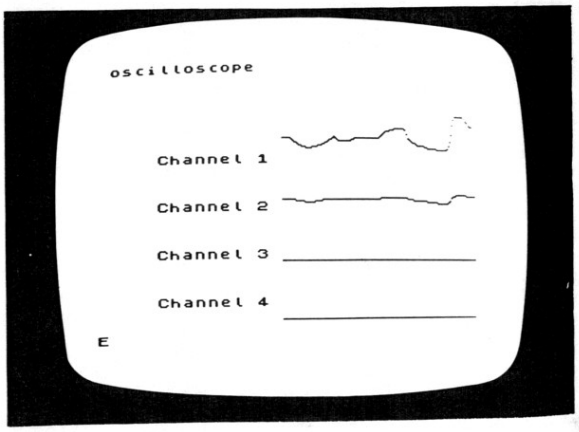


Plate 16 Four-channel chart recorder

recorder. CHART RECORDER (22) uses channels 1, 2, 3 and 4 and is very useful in biology for monitoring several variables at the same time. By introducing a delay, the scrolling process may be slowed down to any acceptable rate. The program has been arranged so that it may be stopped at any time and the display copied using the COPY facility of the Spectrum (Plate 16).

Some phenomena occur extremely rapidly, and even 10 000 readings per second are not enough. A faster ADC (Griffin ADpack) can be connected to the DCP expansion bus of the Interspec. This is free running and requires no start conversion pulse. The data is continuously available and is updated automatically every ten microseconds, so that a data-acquisition rate of 100 000 per second is theoretically possible. In practice, the need to store the readings taken and to provide a variable delay, means that the actual rate is reduced to 60 000 readings per second. This is fast enough for most purposes and a suitable program is listed as FAST ADC (19) in the Appendix. The converted data is available at the address 127, so that all that has to be done is to read this address with LET V=IN 127 (or its equivalent in machine code). Plate 17 uses FAST ADC with the arrangement shown in Figure 4.10 to show the contact bounce that occurs, when a simple push-button switch is pressed.

Data-acquisition routines can be expanded to take several thousand successive readings, storing them in the memory and later outputting them to a chart

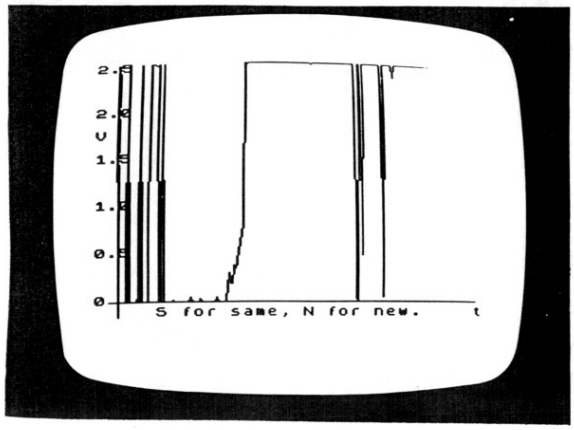


Plate 17 Storage oscilloscope

recorder of a cathode-ray oscilloscope using adaptations of the DAC programs to be described. In this case the microcomputer is being used as a data memory, later displaying the readings it has remembered. Where the data is displayed graphically on the screen instead, the program is also carrying out the function of a storage ocsilloscope. This can replace the cathode-ray oscilloscope in many instances. It is better for some purposes, since it only needs to take a single set of readings, which can then be displayed indefinitely to allow measurements to be taken (for example, the gradient of the graph). There is also the possibility of overlaying two or more successive sets of results.

Educational note

We must again beware of using the microcomputer to carry out functions that are more easily done with other instruments. One common program is to make a microcomputer thermometer. At best this can only be accurate to the nearest degree over the Celsius range 0 to 100, and a mercury in glass device is better. If, however, the reading is displayed in large letters for the whole class to see, or if multiple readings of temperature are taken simultaneously in different places, then the use of the microcomputer may be justified. The value of the microcomputer lies in its ability to take readings (quickly or slowly) and to store them in memory until they are required. Merely to use the microcomputer as an expensive voltmeter or thermometer is a misuse of a powerful resource.

Teachers should also be careful of the black-box nature of the microcomputer. It is not at all obvious what a microcomputer is doing to measure temperature, whereas the mercury in glass instrument is more transparent! The teaching sequence must be carefully checked, so that pupils are convinced that the microcomputer is measuring what the teacher says it is measuring.

Digital to analogue conversion

Digital to analogue conversion is the reverse process used by the Spectrum to produce direct or alternating voltages of its own. This is done with a **digital to analogue converter (DAC)**. This uses all eight lines of an output port (a whole binary number from 0000 0000 to 1111 1111) and converts them into a voltage directly proportional to that number. Thus all eight lines of the output port are connected to the DAC. Each of these eight lines is a digital line, because it can only be switched on HIGH or LOW. Since an eight-bit number can have 256 possible values, the DAC can produce 256 different voltages. The DAC described here produces voltages from 0 V to 2.5 V, in steps of 10 mV.

One simple way to connect a DAC to the spectrum is via the TTL output port of the Interspec (Figure 5.7). Any number between 0 and 255 can then be sent to the output port in the normal way (OUT 95,n). The required voltage appears at the ZN425 output almost immediately (the conversion time is about one microsecond). The op. amp. is necessary to boost the current that can be obtained.

Griffin and George have a DAC as an add on unit to the Interspec (called

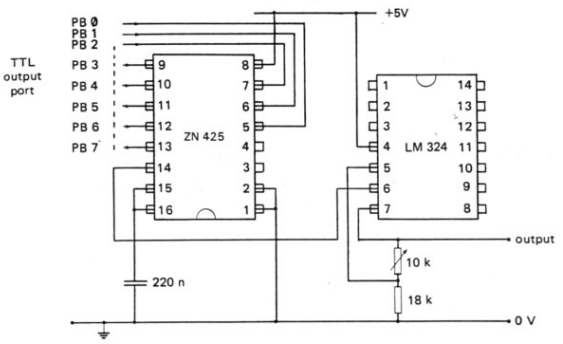


Figure 5.7 ZN425 DAC connected to the TTL output port

the DAC-pack). Thus unit uses a ZN428 device and connects directly to the DCP expansion bus using the address 127. The binary number to be converted is sent to this address using the instruction OUT 127,n (where n is the decimal number between 0 and 255, which determines the final output voltage). Either of these arrangements greatly extends the applications of the Interspec as a laboratory instrument. The first example described is a program to test a student's ability to read a meter:

```
  1  REM VOLTMETER TUTOR
  5  LET convfactor = 0.0048
 10  LET randnum = INT (RND*256)
 20  OUT 127,randnum
 30  CLS
 40  PRINT AT 3,0;"Enter the reading"
 50  PRINT AT 5,0;"on the voltmeter."
 60  INPUT W
 70  LET V = convfactor * randnum
 80  IF ABS(V - W) > 0.01 THEN GOTO 200
100  REM Correct reading
110  PRINT:PRINT"Correct. Press 'Y' for another guess."
120  IF INKEY$<>"y" THEN GOTO 120
```

```
130  GOTO 10
200  REM Wrong reading
210  PRINT:PRINT"You are not right."
220  PRINT:PRINT"Press 'Y' to try again."
230  IF INKEY$ <> "y" THEN GOTO 230
240  GO TO 30
```

Connect a 0 to 3 V (or 0 to 5 V) voltmeter betwen the output terminal of the DAC and the 0 V terminal. The conversion factor in line 5 may need to be adjusted for different DACs. The program sends a voltage to the voltmeter and the student is asked to read it and enter the value obtained. A one per cent error is allowed, but in the case of a discrepancy between the student's result and the computer's, first check that the voltmeter is sufficiently accurate.

Sine waveform

```
 1  REM SINE WAVE OUTPUT
10  FOR a = 0 TO 255
20  LET radians = a * PI / 128
30  LET v = 128 + 127 * SIN(radians)
40  OUT 127,v
50  NEXT a
60  GO TO 10
```

As well as steady voltages, the DAC can also produce alternating voltages of almost any waveform. This example gives a sine waveform, which is slow enough to be observed on the voltmeter. The reason for choosing 256 readings (rather than 360, for example) is because machine-code routines are then easier to write.

Different waveforms can be produced by altering the equation in line 30. For example:

```
30  LET v = a will produce a ramp voltage,
30  LET v = ABS(128 - a) will give a triangular waveform and
30  LET v = 255*NOT(128 > a) will give a square waveform.
```

The period of this oscillation is about twelve seconds. If a longer period is required then a delay can be included; for example:

```
25  PAUSE 5
```

This principle may be used to produce an output slow enough for a chart recorder to monitor. The DAC output should be connected directly to the chart recorder input and the speed, gain and zero of the latter adjusted to give the best results.

The production of higher frequency oscillations is more difficult owing to the

slow speed of BASIC. One way is to reduce the resolution of the waveform, by having fewer output points per cycle.

```
10  FOR a = 0 TO 255 STEP 16
```

Another solution is to do all the calculations beforehand and to store them in the memory as individual bytes. These can then be collected one by one from the memory and sent directly to the DAC in a separate FOR...NEXT loop. This raises the output frequency to a few hertz, but even with this technique, it is not possible to achieve very much.

Data output

```
  1  REM DATA OUTPUT
100  .................
110  ...... DATA .......
120  .. ACQUISITION ....
130  .... ROUTINE ......
140  .................
etc.
200  REM OUTPUT CHANNEL 1 DATA TO CHART RECORDER
210  FOR n = 0 TO 255
220  OUT 127,V(n,1)
230  PAUSE 10
240  NEXT n
```

Assuming that data has been collected with a routine like STORAGE OSCILLOSCOPE, the readings may be output to a chart recorder with this routine. The speed of the chart recorder and the length of the PAUSE may be adjusted to give the best results.

Programmable oscillator

A faster way to send the stored values to the DAC is with a machine-code routine. PROGRAMMABLE OSCILLATOR (16) contains a short delay to enable different frequencies to be produced. The length of this delay is POKED into a memory location prior to calling the routine. The whole range from a few hertz to a frequency of 800 Hz can be obtained by adjusting this delay period.

The waveform of the output is determined by the function used to load the memory with the initial data table. This function, which can be changed in line 20, can be a sine, triangular or ramp function, similar to that used above. Even a square wave can be produced by this method, but there are better ways of doing this as we saw in Chapter 4. At the higher frequencies the waveform can be inspected by connecting the output from the DAC to a cathode-ray oscilloscope.

Automatic graph-plotter

One interesting application is the use of an ADC to measure voltages that have

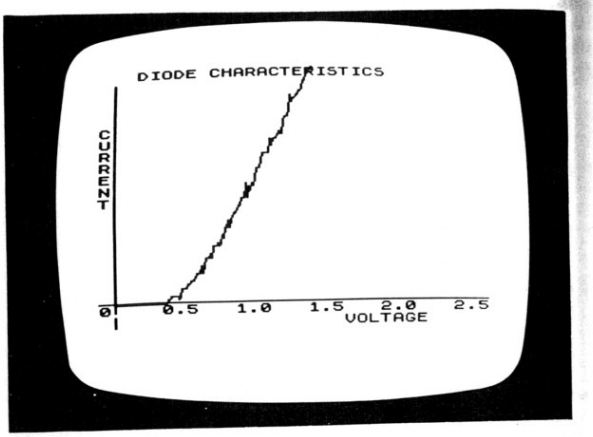


Plate 18 Diode characteristics

been produced by the DAC. One example is the current-voltage characteristics of a transistor, which may be plotted for different values of the bias current. Using transducers it would even be possible to plot pressure-volume curves of a gas at different temperatures (a three dimensional CRO).

The arrangement shown in Figure 5.8 allows the characteristics of three types of diode to be plotted automatically on the same graph. I-V PLOTTER (21) will carry out this task. The LED is particularly suitable for this, since it has a high turn-on voltage and it also lights up when it starts to conduct (Plate 18). Note how the output from the DAC is used to produce the steadily increasing voltage.

Programs like this allow a large number of measurements to be made in the science laboratory. Because the graphical results are quickly available, it is easier to see the science, before it gets lost in the mathematics.

Since the DAC can produce alternating voltages (with PROGRAMMABLE OSCILLATOR), it is possible to carry out experiments on phase lag and lead in reactive circuits in the same way. In such cases, it is the peak value of the alternating voltage that is required. This can be achieved with a simple diode rectifier and smoothing circuit. The values of R and C need to be chosen so that the time constant (R × C) is at least five times the period of the alternating voltage being

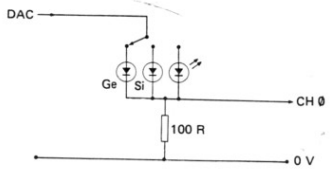


Figure 5.8 Diode characteristics

measured (i.e. R × C > 5/frequency). For accurate measurement it may be worth the extra cost to obtain an r.m.s. to d.c. converter (RS Components 308-786) instead.

Two most interesting applications of DAC and ADC techniques were described by Paul Beverley at the 1982 MUSE Annual Conference. The first of these consists in applying the voltage from a DAC to the input terminals of a chart recorder. When the DAC output voltage is ramped (with the DAC treated like a binary counter) the pen of the chart recorder moves steadily along. The pen is replaced by a photocell and made to scan along the diffraction pattern produced by a laser. The photocell is connected to an ADC channel and a plot of intensity against position is made automatically on the screen. The effect is magnificent!

The second application uses a DAC to produce a direct voltage, which is then

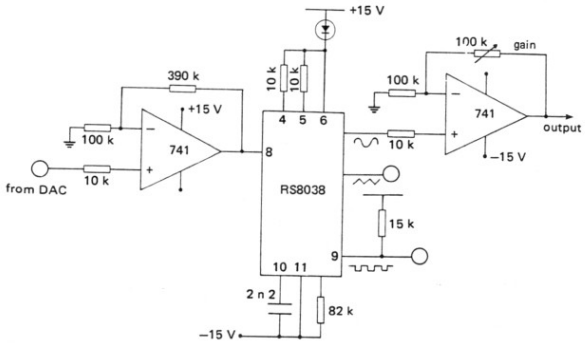


Figure 5.9 Waveform generator

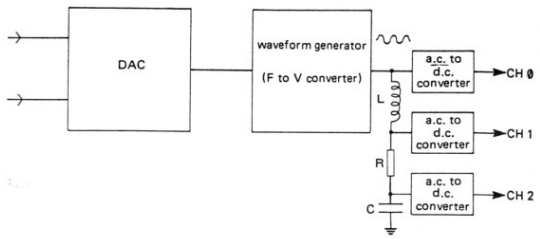


Figure 5.10 Spectrum analyser

fed to a waveform generator (RS Components 305-844) (Figure 5.9). The latter produces sine waves for feeding into a circuit, and square pulses that can be accurately counted by the microcomputer. The frequency of these sine waves is proportional to the direct voltage fed to its input terminal. By ramping the DAC voltage, a whole frequency spectrum is produced by the waveform generator. A range of 500 Hz to 20 kHz was produced with this arrangement.

The main problem of connecting the waveform generator to the DAC is that the direct voltage has to be applied to the former between its input and the +15 V line, not its 0 V line. This is solved by typing its positive rail to 0 V and using an initial op. amp. circuit to convert the voltage output from the DAC to the right levels. The second op. amp. is to buffer the output from the waveform generator before it is fed to a 30 W power amplifier (HY 60). The sine wave voltage is input to a filter circuit, the output from which is connected to an ADC channel via an r.m.s. to d.c. converter, thus measuring the output voltage from the filter (Figure 5.10). A plot of the output voltage against frequency gives the frequency characteristics of the filter circuit. The idea is such a beautiful application of hardware and software techniques that it forms a fitting note on which to end this chapter.

6 The Z80 microprocessor

'When I make a word to do a lot of work like that,'
said Humpty Dumpty, 'I always pay it extra.'
(Lewis Carroll, *Through the Looking Glass)*

The microprocessor is the manager of all the operations that the microcomputer undertakes. As in any organization, the best results are obtained by talking directly to the manager! Unfortunately this one does not speak English, communication with it is in machine code. This chapter is an introduction to microprocessors and includes a detailed examination of one particular device, the Zilog Z80.

Why use machine code?

We ought first to ask why anyone wants to write programs in machine code at all, isn't BASIC good enough? The answer is that BASIC is good enough for some purposes but not for all; there is no alternative if you want to have complete control over the microprocessor. With this control you gain speed; machine-code programs run up to 300 times faster than their BASIC equivalents. You also gain compactness; a machine-code program occupies only a fraction of the memory space needed to run an equivalent BASIC program. Thirdly, you gain freedom; you become independent of the operating system of your microcomputer and able to add extra facilities, which are not implemented by your machine. Finally, it even becomes possible to build your own microcomputer for a particular task, one that is self-contained with its own operating system, memory, program and microprocessor. Such a system is said to be **dedicated** and can be produced comparatively cheaply.

Many people start to learn how to write machine-code programs. Unfortunately there are so many things to be learned to begin with, that some get discouraged. After studying hexadecimal coding, addressing modes and indexation, the usual conclusion is that machine-code programming is too difficult. This introduction tries to overcome these initial problems, by reducing the number of ideas that have to be learned at the beginning. Each of the instructions of a microprocessor is described in a visual way, so that its effects can be more easily observed, thus making this introduction as easy as possible. Even so, machine-code programming is not simple!

Some people use the words 'microprocessor' and 'microcomputer' interchangeably, but they should be distinguished. The microprocessor is the silicon chip which acts as the brain of a computer. A microcomputer contains a

microprocessor, but it contains other chips too, especially memory and I/O chips. (I/O stands for **input-output**, and refers to devices used for getting information into and out of the microprocessor.) Usually a microcomputer will have a keyboard and a TV screen controller too, but this is not always true. Confusion arises because a dedicated system may contain a microprocessor and I/O and memory all inside a single package, called a **single chip microcomputer**. Nevertheless I reserve the word 'microcomputer' for complete machines like the Spectrum and the BBC microcomputer and 'microprocessor' for the processing unit inside the microcomputer that makes it work.

In this book we consider one particular microprocessor, the Zilog Z80, which is found inside many different microcomputers (ZX81, RML 380Z, Exidy Sorcerer and ZX Spectrum). There are several other microprocessors, another popular one being the Rockwell 6502, which is used in the PET, Apple and BBC microcomputers. The instruction set and the codes used for the Z80 are not the same as for other microprocessors, so, unfortunately, you will not be able to use this book to guide you in programming them.

To study machine-code programming, some sort of microprocessor development system can be used, but I am assuming that most readers will not have access to one of these. Instead, you may use a microcomputer for this purpose, but that is not until the next chapter. In this chapter we shall only be using a simulation of how the Z80 microprocessor behaves. One of the problems of real microprocessors is that they have to be programmed exactly in the right way, or they can cause the microcomputer to crash. The advantage of a simulation run from BASIC is that mistakes in programming can be trapped to prevent such disasters.

This program, called Z80 SIMULATION, uses the graphics capability of the microcomputer to show what happens inside the Z80 microprocesor. It was developed by Dr L. Firth of Paisley College of Technology and I am most grateful to him for allowing me to make use of it here. With its aid you can write machine-code instructions immediately and thus learn more quickly, what each instruction does. The listing of this program is given in the Appendix (Z80SIM 5). The simulation does not attempt to deal with all of the instructions that the Z80 can handle, only the more important ones.

Machine-code instructions and the microprocessor

In Chapter 2, we saw how a machine-code program stored in memory is executed by the microprocessor. Each byte contains an instruction in code and the microprocessor fetches each instruction in strict order and executes it. Some of the instructions tell the microprocessor to collect data and some other instructions tell it what to do with that data. Of course there has to be some clever way for the microprocesor to distinguish between a binary number that is an instruction code and a binary number that is data. We shall see later how this is done.

The Z80 microprocessor contains several memories of its own called **registers**.

Some of these are eight bits long and some sixteen. The address of the next instruction to be executed is contained in a sixteen-bit register called the **PROGRAM COUNTER** (PC), which is basically a binary counter. When the microprocessor is ready it fetches the next instruction from the address indicated by the PC. To do this it switches on some of the lines of the address bus to point to the correct place (or **location**) in RAM; this is called putting the address on the address bus. It then sends a signal to the addressed location, which says 'tell me what binary number you are storing'. This signal is called a **read signal** and the READ line of the microprocessor goes LOW, indicating that a memory read is taking place. The addressed location responds by copying its contents onto the **data bus**, and the microprocessor collects them from there. It has now fetched the binary code for its next instruction.

The microprocessor then decodes this binary code, to see which instruction is represented by it. Some instructions only affect the internal registers of the microprocessor, they are called **single-byte instructions**. The microprocessor simply executes these instructions straightaway. Some other instructions require two bytes before they can be executed. When the microprocessor has fetched the first byte and has decoded it, it knows if it has to fetch the rest of the instruction. The PC is increased by one (incremented) to point to the next address and the next byte is fetched from there. The first part of any instruction is the **operation** to be carried out (like ADD or AND) and the second part tells the microprocessor which data to use. This part is called the **operand**. The number of bytes needed for the whole instruction may be anything from one to four bytes.

Memory

We saw in Chapter 2 that some of the microcomputer's memory is RAM and some is ROM. Both are used for storing instructions for the microprocessor to execute. It is useful to imagine the microcomputer's memory as being like the stamp locations in an album, with each location representing one byte. A particular binary number (i.e. a stamp) can be put into any location or taken out of it at any time. It is, of course, necessary to know where any particular stamp is stored, so that it can be found again. Hence every location is given a different **address**. For our purposes there are 256 pages in the album and 256 places (**byte positions**) on each page. To refer to any particular location we must specify its page number and its byte number within that page. Thus the fifty-first byte on page thirty-one would be referred to as byte 51, page 31. To find its position with respect to the first byte in the whole album, we need to calculate 256*page number + byte number. This is called its decimal address.

We noted before how a sixteen-bit address can be contained in two eight-bit bytes, the high byte giving the page address and the low byte giving the address within that page. This is why some instructions take up three bytes – two bytes are needed to specify the sixteen-bit address of the data to be used.

In the 16K Spectrum, the RAM goes from page 64, byte 0 (or decimal address 16384) to page 127, byte 255 (or decimal address 32767). In the 48K Spectrum, the

RAM goes from page 64, byte 0 to page 255, byte 255 (or decimal address 65535). However the microcomputer uses some of the RAM for its own purposes, so not all addresses are available to the user. In both models of the Spectrum some pages of RAM are used for the screen memory and to store the 'system variables', which are used by the operating system as a sort of diary of events. When you write a BASIC program, the operating system stores it in a particular part of the memory and when you type RUN, the operating system automatically goes to the start of this program and begins to collect and interpret it.

It is more difficult to write a machine-code program. The programmer has to decide where to put the program in the memory and tell the microprocessor where the program has been placed. Thus there are two tasks to be performed:

1 Enter the instruction codes into their correct locations.
2 Tell the microprocessor where to go to execute these instructions.

Before you can do either of these tasks, you need to know what sort of instructions can be given to the microprocessor. The rest of this chapter is devoted to a description of the Z80 instruction set. In Chapter 7 we shall return to these two tasks so that you will be able to run real machine-code programs.

The registers of the Z80 microprocessor

The main register is the **ACCUMULATOR.** This is where the results of most calculations are stored. It is often referred to as the **A register**. Next in importance is the B register, which is most often used for counting. Then there is a C register, a D register, an E register, an H register and an L register. Because memory locations have sixteen-bit addresses, these registers are grouped into three high/low pairs as follows: the BC register pair, the DE register pair and the HL register pair. In each pair the first register holds the high byte of the address, while the second register holds the low byte. Thus if the HL register pair holds the address 32006, H holds the value 125 and L holds the value 6. As above, the decimal address is obtained from H*256 + L, which gives 32006 in this example.

The Z80 microprocessor also has several sixteen-bit registers, of which we shall concentrate solely on one - the X-INDEX. This has various uses, which will be described later. Unfortunately for the beginner, I have so far only described half of the available registers of the Z80 microprocessor, but I shall stop here and study the ones I have mentioned. Even that will take up the rest of this chapter.

The ACCUMULATOR is the most used register. The results of logic or arithmetic operations are stored in the ACCUMULATOR after they have been executed. The ACCUMULATOR is an 8-bit register, so it can store any binary number from 0000 0000 to 1111 1111. From the ACCUMULATOR this binary number (data) can be sent to other parts of the microcomputer, such as the TV screen or RAM. The B, C, D, E, H and L registers are all similar to the ACCUMULATOR - they too are eight-bit registers. They are often used as counters or stores for temporary data. Another important purpose is to point to different memory locations.

There are two other registers that are used by the microprocessor (although the programmer is not usually aware of them); these are the **PROGRAM COUNTER** and the **ADDRESS REGISTER**. Each memory location holds the eight-bit binary number which is the code for an instruction. As we have already noted, the PC is used to point to the next location, so that each instruction code can be fetched from the memory in strict order to be decoded by the microprocessor. Some simple instructions only need one byte to tell the microprocessor all that it needs to know. For example, the code 0100 0111 tells the microprocessor to copy the data in the ACCUMULATOR into the B register; no other information is required. Some instructions require two bytes. The code to tell the microprocesor to put the number 0001 1001 (25 in decimal) into the ACCUMULATOR is 0011 1110 0001 1001. The first byte (the **operation**) tells the microprocessor what to do, while the second byte (the **operand**) tells it what data to use.

Some instructions expect the data to be collected from a location in the memory. When the microprocesor wants to collect data from a particular location, it puts the address of that location into its ADDRESS REGISTER. This register is connected to the outside memory via the address bus. Each external location looks at the address bus, but only one location responds, the one that sees its own address on the address bus. This is like calling the class register in school; all the pupils hear the name being called out, but only the pupil with that name responds.

The addressed location can respond in two ways. If it is being read, then it places a copy of the data it contains onto the data bus. This data bus is connected to the DATA register in the microprocessor, so the data in the addressed location is copied into this DATA register. If the instruction to the microprocessor is to load the ACCUMULATOR with this data, then the DATA register transfers this data into the ACCUMULATOR. The whole instruction is called **loading the ACCUMULATOR from memory**. Note that the data is not removed from the addressed memory location, it is only copied into the DATA register. From there it is moved into the ACCUMULATOR and any data already in the ACCUMULATOR will be destroyed.

If the instruction is load the contents of the ACCUMULATOR into memory, the data moves the opposite way. A copy of the data in the ACCUMULATOR is first placed in the DATA register and it travels along the data bus to the addressed location in the memory. Only this addressed location will capture the data being sent. This is called a **write instruction**. Note that the data in the ACCUMULATOR is not destroyed by this instruction, it is only copied into the addressed memory location. Clearly though, any data that was in the addressed location before the write instruction, will be lost, replaced by the new data.

Since the microprocessor only handles eight-bit data, the DATA register is only an eight-bit register. The data bus thus consists of only eight lines, one for each bit of the data. The PROGRAM COUNTER and the ADDRESS register are sixteen-bit registers, because they are concerned with addresses rather than data. They

allow the microprocessor to collect data from any of 65 536 available addresses and the address bus consists of sixteen lines going to different parts of the microcomputer. To simplify matters, Z80 SIMULATION does not use all 65 536 addresses, but only the few from 32001 to 32007. These all have a high byte of 125 and a low byte from 1 to 7.

Mnemonic instruction codes

The instructions to the microprocessor are themselves binary numbers. The microprocessor interprets them according to a special code. For example, the code to instruct the microprocessor to load the decimal number 25 into the ACCUMULATOR is

0011111000011001

It is clear that codes like this are difficult to remember and it would be easy to make a mistake when programming a microprocessor with them. To make life easier a special language has been developed, called **mnemonic language**. The mnemonic for 0011111000011001 is LD A,25 (meaning load the ACCUMULATOR with the number 25). As you can see, the mnemonic is much easier to interpret than the binary code.

The instruction LD A,25 consists of two parts, the operation (LD), which tells the microprocessor what to do and the operands, which tell it what data to use or what registers to use. In this instruction the operand itself contains the data to be used, so it can be immediately transferred to the ACCUMULATOR. It is therefore called a **load immediate instruction**.

Another instruction is 'load from memory'. This has the mnemonic LD A,(32001). The operation has the same mnemonic (LD) and the same first operand (A), but the second operand is different, it is enclosed in brackets. This tells the microprocessor that the operand is not itself data but is an address, where the desired data can be found. LD A,(32001) means load the ACCUMULATOR with the data which is in the memory at the address number 32001 (i.e. at memory location 32001). The data is collected from location 32001 by putting the number 32001 on the address, bus and collecting the data via the data bus, as described above.

To write data into a memory location the same load instruction is used, but with the operands reversed. LD (32001),A means 'copy the data from the ACCUMULATOR into memory location 32001'. There is no instruction like LD A,25, because 25 is not an address, it is data. You can only store the contents of the ACCUMULATOR in an addressed location. This means that if you want to change the contents of memory location 32001 to the value 25, you must do it in two stages. First you must load the value 25 into the ACCUMULATOR with the instruction LD A,25 and then you must store it in location 32001 with the instruction LD (32001)A.

If you want the data in location 32002 to be copied into location 32001, you

must also do it in two stages. First you copy the data from location 32002 into the ACCUMULATOR with LD A,(32002). Next you copy it from the ACCUMULATOR into location 1 with the instruction LD (32001),A. To show this, load and run the simulation program (5) called Z80 SIMULATION, which is listed in the Appendix. The program displays the following registers:

ACCUMULATOR
B, C, D, E, H and L registers
X-INDEX
PROGRAM COUNTER
ADDRESS register

It also shows the flag register and the STACK, but we shall not deal with these just yet. The microprocessor is connected to the external memory via the data bus and the address bus. Only seven memory locations are shown, from 32001 to 32007. In a real microcomputer any of 65 536 locations can be addressed. In this respect this simulation is invalid, but the change to full sixteen-bit addressing is not too difficult, it is left until later.

At the bottom of the display is the **INSTRUCTION register** containing the current instruction. Normally this instruction has been fetched from the program memory at the address pointed to by the PROGRAM COUNTER. We shall, however, enter instructions one at a time, so the PROGRAM COUNTER will not actually be used in this way. Each instruction is shown in mnemonic language, so that it can be more easily understood, but remember that each instruction would really be stored as a binary number. After each instruction has been executed, the INSTRUCTION register will display it.

After loading the program, lock the keyboard to upper case by pressing CAPS-SHIFT and key 2 (CAPS-LOCK). Then type in the following instructions:

```
LD A,25
```

If you make a mistake while typing, you can rub it out with the DELETE key in the normal way. When you have typed LD A,25 correctly, press the ENTER key as usual. The simulation program will then attempt to execute your instruction. If you have typed it wrongly (for example if you have typed LDA,25 or LD A 25) the simulation program will tell you that your instruction is not valid by displaying ERROR. The commas and spaces are vital and must occur in the proper places, otherwise an error will be signalled. After entering this instruction, you should observe the number 25 enter into the ACCUMULATOR. The ADDRESS register will not be affected, because it is not used for this instruction.

Now enter LD (32001),A and you should see that the number 25 in the ACCUMULATOR is copied into location 32001. The original data in location 32001 is destroyed, but the 25 in the ACCUMULATOR is not lost. Because the address bus is used to do this, the corresponding ADDRESS register is affected. Finally copy the contents of location 32002 to location 32001. Enter:

```
LD A,(32002)
LD (32001),A
```

Continue this investigation for yourself. Try changing the operand 25 to other values in the range 0 to 255 and the operand addresses 32001 and 32002 to other addresses in the range 32001 to 32007 (values outside these ranges will produce an ERROR).

Data can be copied from any of the other internal registers into the ACCUMULATOR in much the same way. For example:

```
LD A,B
```

copies the data from the B register into the ACCUMULATOR, while

```
LD B,A
```

does the reverse. Try your own variations of this and you will discover that data can be copied from any of the A, B, C, D, E, H and L registers in to any of the same set of registers. In addition, any of these registers can be loaded with data directly. For example:

```
LD B,A
LD C,B
LD H,L
LD L,12
LD C,255
```

You can, however, only load from memory into the ACCUMULATOR. LD C,(32001) is not allowed.

As a test of your understanding, try the following problems, the solutions to which are given at the end of the chapter.

1 Type in a series of instructions to make the contents of location 32002 equal to 50.
2 Type in a series of instructions to make the contents of location 32006 equal to the contents of location 32007, but do not change the contents of location 32007.
3 Type in a series of instructions to make the contents of location 32001 equal to the number 1, the contents of location 32002 equal to 2 and the contents of location 32003 equal to 3.
4 Type in a series of instructions to make all of the internal registers contain 1.
5 Load the C register with the contents of location 32001. (Clue: use the ACCUMULATOR.)
6 Load location 32001 with the contents of the H register.
7 What is the difference between a 'write to memory' and a 'read from memory'? Which occurs when the instruction LD A,(32005) is executed?

Microprocesor arithmetic

In the Z80 microprocessor, addition is performed by adding data to the current contents of the ACCUMULATOR. The instruction ADD A,30 will add 30 to the existing contents of the ACCUMULATOR. The instruction ADD A,B will add the contents of the B register to the existing contents of the ACCUMULATOR. In both cases the result of the addition is left in the ACCUMULATOR and the original contents of the ACCUMULATOR are destroyed. It is not possible to leave the result of an addition in any of the other registers (although there is also a double byte ADD instruction, which can leave a result in the HL register pair; we shall look at this later).

To add together two numbers such as 5 and 6, we first of all execute the instruction LD A,5, followed by the instruction ADD A,6. You can try this for yourself using the simulation program. You will see that the result (11) is left in the ACCUMULATOR.

```
LD A,5
ADD A,6
```

It is not possible to add the contents of a memory location directly to the accumulator. One way of doing this is to load the first number and then keep this in one of the other registers, as follows:

```
LD A,(32001)
LD B,A
LD A,(32002)
ADD A,B
```

will add the contents of location 32001 to the contents of location 32002, leaving the result in the ACCUMULATOR.

Now enter each of these instructions in turn. After each one, note the effect on the contents of the ACCUMULATOR.

```
LD A,5
LD (32001),A
LD A,6
LD (32002),A
LD A,(32001)
ADD A,(32002)
LD (32003),A
```

Repeat this with some of your own numbers. Now try

```
LD A,255
ADD A,1
```

The result 0 remains in the ACCUMULATOR. A moment's thought will explain this. The largest number that the ACCUMULATOR can store is 1111 1111 (or 255 in decimal). If we try to exceed this number, it starts again from zero. (For the

mathematically minded, the microprocessor is counting in modulo 256.) This is like the milometer in a motor car: when the distance exceeds 99 999 miles, the milometer starts again from zero. Although it is possible to tell from the appearance of a car, whether it has travelled ten or 100 010 miles, the ACCUMULATOR does not age in the same way. To show that the ACCUMULATOR has exceeded 255 a special CARRY bit is used in the flag register. If the result of the addition is greater than 255 then this CARRY bit is set to logic 1. If the result of the calculation is not greater than 255, then this CARRY bit is cleared to 0. Check this by entering the following instructions:

```
LD A,100
ADD A,100
ADD A,100
```

The CARRY bit is particularly useful, since it enables the microprocessor to add large numbers, (it would be very inconvenient if we could not deal with numbers greater than 255). First of all, how does the microprocessor store such numbers? This problem has to be solved in the decimal system too, since a set of decimal digits can only count up to nine. To count higher numbers we use more sets of digits, arranged in columns and called hundreds, tens and units. The decimal number 23, is really $2 \times 10 + 3$.

Similarly, we can use two eight-bit bytes to store numbers larger than 256. This is not simple because the two columns are not tens and units, but 256s and units. The first column is called the **high byte** and the second is called the **low byte**. Converting a two-byte binary number to decimal requires the following formula:

```
decimal = 256 * high byte + low byte
```

A further complication is that the Z80 collects its number in the order low byte followed by high byte. We shall stick to this practice, even though we shall not be dealing with the microprocessor directly for some time yet. When written in this low byte/high byte order, the decimal number 4100 becomes 4,16, since $16 \times 256 + 4 = 4100$. Other examples are 3,12, which is $12 \times 256 + 3 = 3075$ and 250,255 which is $255 \times 256 + 250 = 65530$. To convert a decimal number to a two-byte number, divide the number by 256 – the integer part of the result is the high byte. Multiply this by 256 and subtract it from the original number to get the low byte.

Problems for you to try:

8 Convert each of the following low byte/high byte numbers to decimal:
   (i) 0,2
   (ii) 10,12
   (iii) 200,40
   (iv) 0,80
   (v) 96,234

9 Convert each of the following decimal numbers to low byte/high byte numbers:

   (i) 256
   (ii) 1024
   (iii) 8000
   (iv) 8192
   (v) 65535

Numbers larger than 255 are added in the following way. Each number is held in two successive locations, low byte and high byte. First the low bytes of the two numbers are added together and the result is stored. Then the high bytes are added together and the result is stored also. If the CARRY bit was set after the low-byte addition, it can be added in with the high bytes. The instruction to do this is **ADC**, meaning *add with CARRY*. In adding the low bytes, we do not want the CARRY bit to be added in. So we use the simpler form **ADD**, which ignores the CARRY bit.

Since we cannot store both the high byte and the low byte together in the ACCUMULATOR, we make use of the memory. This is illustrated in Figure 6.1. We put the number 4100 in the two locations 32001 and 32002, with the low byte (4) in location 32001 and the high byte (16) in location 32002. Then we put the number 510 into the next two locations (254 into location 32003 and 1 in location 32004).

Next we add together the low bytes of the two numbers using the ADD instruction (like adding up the units in a decimal addition). Because the result is greater than 255, the CARRY bit will be set (as in the decimal addition $5 + 8 = 3$, carry 1). We store the result of this low byte addition in location 32005.

Then we add together the high bytes using the ADC instruction. As we do this the CARRY bit from the low-byte addition is added in as well (as in decimal addition, when we get to the tens column we add in the carry from the units). The result of this is then stored in location 32006. The whole set of instructions for this double-byte addition is given below. Enter each of these instructions in turn. As each instruction is entered and executed, note what happens to the CARRY bit in the flag register and to the contents of the ACCUMULATOR. The first eight instructions are simply setting up the memory locations with the correct numbers.

```
LD A,4
LD (32001),A
LD A,16
LD (32002),A
LD A,254
LD (32003),A
LD A,1
LD (32004),A
LD A,(32001)
LDE, A
LDA, (32003)
ADD A,E
LD (32005),A
```

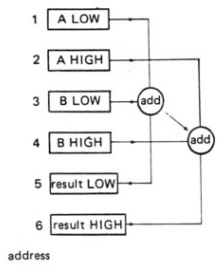


Figure 6.1 Two-byte addition

```
LD A,(32002)
LD A,E
LD A,(32004)
ADC A,E
LD (32006),A
```

The result is stored in locations 32005 and 32006, is it the result you expected?

Continue this investigation with large and small numbers. You will get the correct answer as long as the result is not greater than 65 535. What happens if the result is larger than this? (Clue: look at the CARRY bit when all the instructions have been executed.)

Problems for you to try:

10 Add together the numbers 45 and 54 (single byte addition) without using any external memory locations. (Clue: use the immediate mode.)

11 Add together the contents of locations 32004 and 32005 (single-byte addition) and put the result in location 32003.

12 Add together the numbers 450 and 540 using double-byte addition. Put one double-byte number into locations 32001 and 32002 and the other into locations 32003 and 32004. Then add the numbers and put the result in locations 32005 and 32006.

13 Put the single-byte number 225 into location 32001 and 200 into location 32002. Then add up the numbers and put the result into locations 32003 and 32004. The result is greater than 255, so be very careful about what happens to the CARRY bit.

14 Put the double-byte number 1000 into locations 32005 and 32006. Now add 1 in immediate mode to the contents of locations 32005 and 32006, storing the result in the same locations. Consider how you will cope with the situation where the low-byte addition results in the CARRY bit being set.

15 What two decimal numbers can be added together, using double-byte addition, to give the result 0? (Clue: there are 32 768 different answers!)

Subtraction

Subtraction can also be performed using the immediate mode or the register mode. The instruction SUB 1 will subtract one from the contents of the ACCUMULATOR, leaving the result in the ACCUMULATOR. Note that the ACCUMULATOR is the only register that can be used to store the result. For this reason it is not necessary to specify it in the mnemonics. Some books do this, writing the above instruction as LD A,1, but we shall stick to the convention in the Spectrum user guide – where only the ACCUMULATOR can be the first operand, it is not specified.

The effect on the CARRY bit is the same as for addition. If the second number is larger than the first, then one is borrowed from the next column. In the units column this 1 becomes 256, so the result in the ACCUMULATOR becomes larger than it was before. For example:

```
LD A,10
SUB 11
```

results in the number 255 being left in the ACCUMULATOR and one is borrowed from the next column. This 'borrow' is shown by the CARRY bit being set to 1. If there is no borrow as in the following case,

```
LDA,11
SUB 10
```

then the CARRY bit is cleared to 0 after the subtraction. Check these ideas by entering each of the above instructions, noting the status of the CARRY bit each time.

Note how the following instructions

```
LD A,0
SUB 1
```

leave 255 in the ACCUMULATOR, thus indicating that 255 is equivalent to –1 in this arithmetic.

The operation SBC automatically 'pays back' the CARRY bit (in the same way that ADC automatically adds in the CARRY bit). SBC can, however, be used with the HL register pair as the first operand, so it becomes essential to specify the ACCUMULATOR if the latter is intended to be the first operand. For example:

```
LD A,10
SBC A,1
```

will produce the result 9 in the ACCUMULATOR if the CARRY bit was previously cleared or 8 if the CARRY bit was previously set. To avoid errors in two-byte

subtraction, the first subtract instruction should use the operation SUB and subsequent subtractions on the higher byte should use the SBC. Check these ideas by entering each of the following instructions, noting the status of the CARRY bit each time.

```
LD A,10
SUB 11      (this leaves the CARRY bit set)
LD A,5
SBC A,4
```

In the last case the result is 0, not 1, because the CARRY bit is subtracted as well.

If the process involves double-byte subtraction, SBC ensures that the borrow is repaid during the high-byte subtraction. Enter each of the following instructions and observe their effect on the various registers:

Place the number 3,2 (decimal 515) into locations 32001 and 32002. Then subtract 5,1 (decimal 261) from the first number in immediate mode and place the result in locations 32003 and 32004.

```
LD A,3
LD (32001),A
LD A,2
LD (32002),A      (puts the number into the two bytes of memory)
LD A,(32001)
SUB 5
LD (32003),A      (low byte subtraction)
LD A,(32002)
SBC A,1
LD (32004),A      (high byte subtraction)
```

Problems for you to try:

16 Load a number into the ACCUMULATOR. Then subtract (SUB) this number from itself, leaving the result in the ACCUMULATOR. Do you get the result 0?

17 Place a single-byte number into location 32001 and another number into location 32002. Subtract the contents of location 32002 from the contents of location 32001, placing the result in location 32003.

18 Place a double-byte number in locations 32001 and 32002. Add this number to itself and put the result in locations 32003 and 32004. Then subtract the number in locations 32001 and 32002 from the number in locations 32003 and 32004, leaving the result in locations 32003 and 32004. What do you notice about this result?

19 Place 0 into the locations 32001 and 32002. Treat this as a double-byte number and subtract one from it in immediate mode, leaving the result in locations 32001 and 32002. What do you notice about the result?

Counting

Counting can be done by adding one repeatedly to the location being used as a counter, but it can also be done with the single instruction **increment**. The instruction **INC A** simply adds one to the contents of the ACCUMULATOR. Similarly **INC B**, **INC C** etc. do the same for the other registers. There is, however, an important difference with ordinary addition – INC does not affect the CARRY bit.

```
LD A,255
INC A
```

leaves 0 in the ACCUMULATOR, but the CARRY bit remains unaltered.

The **decrement** instruction, **DEC A**, is similar, but now the contents of the ACCUMULATOR are reduced by one instead. In this case too, no account is taken of the CARRY bit. This is not affected if a register is decremented below zero.

Both instructions are used a great deal in counting. It is often necessary in a program to repeat an instruction or a set of instructions several times (like the FOR . . . NEXT loop in BASIC). Suppose we want to repeat it eight times. The register being used as a counter is initially made equal to eight. After each cycle of the required instructions, this counter is decremented. When it reaches zero, the cycle has been repeated eight times. The B register is the one most often used for counting.

Investigate this set of instructions:

```
LD A,253
INC A
INC A
INC A
INC A
LD B,2
DEC B
DEC B
DEC B
DEC B
```

Now try this:

20 Place 0 in the B register and 3 in the ACCUMULATOR. Now increment the B register and decrement the ACCUMULATOR repetitively until the latter reaches zero. What value is left in the B register?

Incrementing and decrementing cannot be carried out directly on a memory location. To increment the contents of location 32001, say, it can first be loaded into the ACCUMULATOR, altered and then returned, like this:

```
LD A,(32001)
INC A
LD (32001),A
```

Logic instructions

As well as its arithmetic instructions, the microprocessor can also perform logic operations on data. Since the result of a logic operation can only be left in the ACCUMULATOR, the latter is not specified as the first operand. Logic operations are concerned with individual bits, so since a single byte of data consists of eight bits, the microprocessor has to perform eight logic operations at a time. Consider the series of instructions:

```
LD A,5
AND 6
```

The second data in this case is the binary number 0000 0110. This is ANDed with the data already in the ACCUMULATOR, which is the binary number 0000 0101. These two bytes are ANDed one bit at a time and the result is put into the ACCUMULATOR.

6 is	0	0	0	0	0	1	1	0
5 is	0	0	0	0	0	1	0	1
Result	0	0	0	0	0	1	0	0

The result has a logic 1 only where there is a logic 1 in both of the corresponding bit positions of the two bytes being ANDed. This is the bit 2 position, so the result of ANDing 5 and 6 is 4.

ANDing is a good way of clearing particular bits to 0 without affecting the other bits at the same time. If the ACCUMULATOR contained the value 3 (binary 0000 0011) and we wanted to turn bit 0 off, we could perform the instruction AND 254 (binary 1111 1110), which would only affect bit 0. (There are other, more direct ways of doing this.)

```
LD A,3
AND 254
```

Another use of the AND instruction is to **mask** an input (say from the Interspec) to look at one particular bit (say bit 0). If we load the contents of location 32005 into the ACCUMULATOR and perform the instruction AND 1, the result will be 1 if bit 0 of location 32005 was set and 0 if bit 0 was cleared. In a similar way,

```
LD B,127
LD A,128
AND B
```

leaves 0 in the ACCUMULATOR.

```
LD B,255
LD A,128
AND B
```

leaves 128 in the ACCUMULATOR.

Logical OR is carried out with the OR operation, which can also have an immediate mode or register mode operand.

```
LD B,5
LD A,6
OR B
```

The ACCUMULATOR contains 6 and this is ORed with 5, so the result is 7, as follows:

6 is	0	0	0	0	0	1	1	0
5 is	0	0	0	0	0	1	0	1
Result	0	0	0	0	0	1	1	1

There is a logic 1 in the result if there is a logic 1 in either of the corresponding bit positions of the two starting numbers.

One use of OR is to switch one particular bit on, without affecting the other bits. To turn on bit 7 of location 32005, we load the contents of location 32005 into the ACCUMULATOR, OR it with 1000 0000 (decimal 128) and store the result back in 32005.

```
LD B,128
LD A,127
OR B
```

leaves 255 in the ACCUMULATOR.

A less familiar logic instruction is EXCLUSIVE-OR. This gives a logic 1 output if the two inputs to the gate are different. The EXCLUSIVE-OR output goes to logic 0 if its two inputs are the same. It has the following truth table:

Input A	Input B	Result
0	0	0
0	1	1
1	0	1
1	1	0

The microprocessor operation which does this is **XOR**. This too can be used in the immediate mode or the register mode.

```
LD A,6
XOR 255
```

6 is	0	0	0	0	0	1	1	0
255 is	1	1	1	1	1	1	1	1
Result	1	1	1	1	1	0	0	1

XOR has one special property that makes it particularly useful. If the contents of location 32005 are loaded into the ACCUMULATOR and then EXCLUSIVE-ORed with 1111 1111, each bit that was previously on will be turned off, and each

bit that was previously off will be turned on. This can be seen from a comparison of the two numbers above. If the data collected from location 32005 is 6, the result shows a logic 0 in each bit position where it was previously a logic 1, and vice versa.

Try each of the following sets of instructions:

```
LD A,255        ;This instruction will switch all
LD (32005),A    ;bits of location 32005 on.
LD (32005),A
AND 16          ;This will switch off all
LD (32005),A    ;bits except bit 4.
OR 128
LD (32005),A    ;This will turn on bit 7 also.
XOR 240         ;This will turn bits 4 and 7 off and
LD (32005),A    ;bits 5 and 6 on.
```

Enter each of the following instructions in turn. Before each one, try to predict what the result in the ACCUMULATOR will be. Then see if you were correct.

```
LD A,170
AND 3
OR 16
LD B,A
AND 10
OR 15
XOR 10
AND 1
LD A,0
OR 1
AND 2
XOR B
```

Problems for you to try:

21 What is the result of ANDing 85 with 45?
22 What is the result of ORing 85 with 45?
23 What is the result of EXCLUSIVE-ORing 85 with 45?
24 How could you switch off bits 1 and 2 of location 32005 without changing the state of the other bits?
25 How could you switch bits 0, 1, 2, 3, 4, 5 and 6 of location 32005 on, yet not affect bit 7?

Indexed addressing

In a BASIC program we often use an array to store a set of related data, which may then be accessed by number. X(5) contains the fifth number of the array. In

machine code a similar system is used. The data is stored in successive bytes of memory and a special register is used as a pointer to point to each byte in turn. In the Spectrum, the X-INDEX is the easiest pointer to use. For example, location 32001 could contain the square of the number 1, location 32002 could contain the square of the number 2 and so on. Then, to find the square of a number in a machine-code program, we only have to look it up in this table. We do this with **indexed addressing**.

The instruction LD A,IX+5) loads the ACCUMULATOR with the contents of a memory location. The chosen location is obtained by adding the X-INDEX to the number specified (in this case, 5). This number is called the **displacement**. Thus if the X-INDEX is initially set to 32000, the chosen location would have the address 32000 + 5, which is, of course, location 32005. The contents of this location would thus be loaded into the ACCUMULATOR. The X-INDEX is loaded with its starting value with this instruction:

```
LD IX,32000
```

Then to collect the data from the fifth store, we say:

```
LD A,(IX+5)
```

The displacement can only be added to the base address of the index and cannot exceed 255, so only locations within 255 bytes of the base address can be reached. To go further, the value of the X-INDEX must be altered. Since the X-INDEX can have any value from 0 to 65535, then any desired location can be reached in this way. However, Z80 SIMULATION only displays the location 32001 to 32007, so it is not possible to give indexed addressing a full test. All load, arithmetic and logic instructions so far described can be used with indexed addressing as well as with the other addressing modes already described.

The advantage of indexing will not yet be apparent, because we have not discussed how to repeat a series of instructions. Let us first learn how to use this instruction. The following program will put the value 1 into location 32001, the value 2 into location 32002 and so on. Enter this series of instructions and see what happens each time. Note especially what happens to the ADDRESS register.

```
LD IX,32000
LD A,1
LD (IX+1),A
LD A,2
LD (IX+2),A
LD A,3
LD (IX+3),A
LD A,4
LD (IX+4),A
etc.
```

This program can be greatly simplified by keeping the displacement fixed and

by incrementing the X-INDEX each time instead. The program then becomes:

```
LD IX,32000
LD A,1
LD (IX+1),A     ;puts 1 in location 32001
INC A
INC IX
LD (IX+1),A     ;puts 2 in location 32002
INC A
INC IX
LD (IX+1),A     ;puts 3 in location 32003
INC A
INC IX
LD (IX+1),A     ;puts 4 in location 32004
```

This is the same program but using INC IX. Note how the same set of instructions is repeated over and over again. Clearly the machine-code equivalent of a FOR... NEXT loop will make this a very simple program, when we come to it. Can you repeat this procedure, but changing the load instructions to (IX+0) instead of (IX+1)? What difference does it make?

Rewrite the above program to read the contents of each memory location into the ACCUMULATOR, to add 1 and to store the result back in the same location. The program should use indexed addressing to point to each location in turn.

The PROGRAM COUNTER

Although Z80 SIMULATION is useful for demonstrating the different instructions available in the Z80 microprocessor, it is not possible to make it run a program. That is, it will not carry out a set of instructions automatically. It is as if in BASIC we could only enter statements one at a time into a microcomputer. We need a way of storing a whole series of instructions that the microprocessor can execute one by one. This is the only way that we shall be able to repeat a cycle of instructions for a given number of times. This will be shown in Chapter 7. Until then we can only simulate a machine-code program.

Until now we have not bothered particularly about the **PROGRAM COUNTER**, henceforth called the **PC**. This is a sixteen-bit counter that points to the address of the next instruction. Try each of these instructions and note the effect on the PC each time. Some (called **single-byte instructions**) increment the PC by one, while others (**double-byte instructions**) increase it by two. There are also three-byte and four-byte instructions, which are they?

```
LDA A,B
ADD A,C
SUB C
LD IX,32000
LD A,(IX+1)
```



```
LD (32001),A
AND (IX+1)
XOR (IX+2)
LD B,5
INC A
INC IX
```

The address in the PC starts at 32500, which is roughly where many real machine-code programs for the Spectrum begin. If you enter a very large number of instructions you could get this to go right up to 65535. Further increases cause it to reset to zero. 65 535 is the maximum number that a sixteen-bit register can hold. In the memory a segment of a program would be stored sequentially like this:

```
32500   LD IX,32000     ;four-byte instruction
32504   LD (IX+3),B     ;three-byte instruction
32507   LD A,25         ;two-byte instruction
32509   LD B,A          ;single-byte instruction
32510   etc.
```

Notice how the PC seems to be giving each instruction a number as in BASIC. But it is not at all like BASIC, these numbers are the address of the first byte of the instruction, some of which are four-, three-, two- and one-byte instructions. Each byte contains part of the code for the complete instruction. Here is the same program written out one byte at a time, to show the way that these codes are stored:

PC	Code	Instruction
32500	221	LD IX,32000
32501	33	
32502	0	
32503	125	
32504	221	LD (IX+3),B
32505	112	
32506	3	
32507	62	LD A,25
32508	25	
32509	71	LD B,A

The line numbers must be consecutive and none may be omitted. This is annoying when writing machine-code programs, because if you later want to insert another instruction, you have to move all the others down by one or two bytes (which is one of the reasons why BASIC is a better language than machine code). In BASIC, the next statement fetched is the one with the next highest number, and it does not matter if some numbers are omitted. The line numbers in machine-code programming represent the addresses in memory where the codes for the instructions are stored. They are the values taken by the PROGRAM

COUNTER to get each new instruction. Each time the PC executes an instruction, it is simply incremented to point to the next instruction. If we put this next instruction in the wrong place, the microprocessor will not notice, it will still fetch its next instruction from the next location in memory. It is quite possible that this wrong instruction collected by the microprocessor will cause the whole program to crash. Even worse, the microprocessor contains no error-checking procedures like BASIC, so it will carry on trying to interpret its instructions. Since data and instruction might be the same code, the microprocessor could get hopelessly lost.

Using the address of the program counter, it is common to write out machine-code programs like this:

```
32500   rpt     LD IX,32000     ;initialize X-INDEX
32504           LD (IX+3),B     ;store contents of B register
32507           LD A,25         ;initialize ACCUMULATOR
32509           LD B,A          ;save contents of ACCUMULATOR
32510   etc.
```

We are not bothered with how this program works, we are just looking at the method of writing it. The first column is the value of the PC as before, which is the address of the first part of each instruction.

The second column of the program listing is the name or **label** of the cycle of instructions to be repeated (rpt). This way of labelling the program is to show us where the **cycle** (or loop) begins. The microprocessor takes no notice of labels, because it uses the PC to determine where this loop is. Z80 SIMULATION likewise uses numbers to determine the next instruction. The label is only included for our information, it cannot be entered as any part of an instruction in Z80 SIMULATION. The third column has the mnemonic of the instruction as before. The remainder of the line, after the semi-colon, is the comment column. This is used to explain what is going on, rather like the REM statement in BASIC. Z80 SIMULATION will not recognize such comments, even if there is room to put them in, so these too should not be entered.

Program jumps

BASIC has two methods of jumping to a different part of the program, **GOTO** and **GOSUB**. There are exact equivalents in machine code too, **JP** (jump) and **CALL** (go to subroutine). The instruction JP 12000 loads the address 12000 into the PC and the next instruction is fetched from that address. Execution then continues line by line from this new position. JP therefore transfers control completely to this new part of the program. The microprocessor loses all knowledge of where it has come from and it has no way of getting back to it (unless, that is, the new part of the program sends it back with another JP instruction).

CALL 12000 puts the address 12000 into the PC in the same way, but the previous address of the PC is first saved in a special memory called the **STACK**.

The instructions beginning at line 12000 are then executed in order until the final instruction **RET** (just like RETURN in BASIC). Control then passes back to the main program at the address previously saved on the STACK.

JP and CALL are three-byte instructions so you might expect to see the PC increase by three when they are used. However, these instructions change the address in the PC, so you cannot really see this happen. Since the return address is a sixteen-bit address, it is saved on the STACK as a two-byte address (written in the low byte, high byte order). Note that the address saved on the STACK is not exactly the same as that originally in the PC. The reason for this is because the PC contains the address of the first byte of an instruction. By the time the PC has reached the end of the instruction, it points to a slightly different address.

Enter these instructions and watch especially the STACK and how the PC changes its address. Note that both the low byte and the high byte of the return address are stored on the STACK and note how the **STACK pointer (the arrow)** moves up and down, pointing to the last entry in the STACK. Note the relationship between the number pushed onto or pulled off the STACK and the PC address, when the CALL and the RET instructions are executed.

```
PC
xxxxx       JP 12000
12000       JP 10000
10000       JP 32000
32000       CALL 20000
20000       RET
32003       JP 30000
```

Do you see the difference between the JP and CALL instructions?

Problems for you to try:

26 What address would be left in the PC after the following instructions had been executed?

```
xxxxx       JP 12000
12000       JP 20000
```

27 What would a microprocessor do if it met this instruction?

```
12000       JP12000
```

Conditional jumps

In BASIC the IF . . . THEN statement allows the program to choose between alternatives.

```
1000  IF Y=0 THEN GOTO 5000
1010  X=2
```

If Y is zero at statement 1000, this causes a jump to line 5000. If Y is not zero, the

program continues with statement 1010. In machine code the conditional jump instructions have the same purpose. After nearly every instruction, a special bit in the flag register, called the **ZERO bit**, is changed. It is set to 1 if the result of the instruction is zero, it is cleared to 0 if the result is not zero. Watch the effect on the ZERO bit (Z) in the Z80 SIMULATION, when each of the following is executed:

```
LD A,0
LD B,1
OR B
XOR 255
AND 0
DEC A
DEC B
```

The **JP NZ** instruction (**jump if non-zero**) tests this ZERO bit and if it is cleared to 0 (i.e. the result of the previous instruction was not zero), then the jump is obeyed. If the ZERO bit is set to 1, then the result of the previous instruction was zero, so the jump is not obeyed and execution continues with the next line.

The **JP Z** instruction (**jump if zero**) is the opposite of this. The jump is obeyed when the ZERO bit is set and is not obeyed when the ZERO bit is cleared. Let us now see how this conditional jumping is used. It is assumed that the following program begins at 32500. You can get to this address by entering JP 32500. Remember not to type in the label nor the comment columns.

```
32500        LD B,5          ;Set counter
32502        LD IX,32000     ;Set pointer
32506  rpt   LD A,1          ;Get value
32508        LD (IX+1),A     ;Save value
32511        INC A           ;Next value
32512        INC IX          ;Next location
32514        DEC B           ;Dec counter
32515        JP NZ,32506     ;Repeat cycle
32518                        ;remainder of program
```

This program first sets a counter (the B register) to 5 and sets the X-INDEX to point to the memory location 32000. Then 1 is loaded into the ACCUMULATOR and stored in location 32001. Both the ACCUMULATOR and the X-INDEX are incremented and the counter is decremented. This instruction makes the B register equal to 4, which is non-zero, so the ZERO bit remains cleared. The JP NZ instruction thus succeeds and the PC returns to 32506 (the line called rpt). The next time this loop is executed, the B register contains three and the loop is repeated again. This continues until the B register reaches zero. Then the ZERO bit is set, the JP NZ condition finally fails and there is no jump back to rpt. So the program executes the loop five times, effectively copying the FOR . . . NEXT loop of BASIC.

An alternative type of jump is called a **relative jump**. In this case the operand is not an address, but the number of bytes to be skipped over. JR + 8 is an instruction to omit the next eight bytes. JR NZ + 8 means omit the next eight bytes if the ZERO bit is cleared. The operand in a relative jump is called a **displacement** and it is the number of bytes added to the PC. This displacement can be positive (a forward jump) or negative (a backward jump). For Z80 SIMULATION we signify this with the + or — symbols, which must be included. A real microprocessor has a special way of distinguishing positive and negative numbers, which we shall deal with later.

```
32500        LD B,5          ;Set counter
32502        LD IX,32000     ;Set pointer
32506  rpt   LD A,1          ;Get value
32508        LD (IX+1),A     ;Save value
32511        INC A           ;Next value
32512        INC IX          ;Next location
32514        DEC B           ;Dec counter
32515        JR NZ,-11       ;Repeat cycle
32517                        ;remainder of program
```

In this program the JR NZ,-11 instruction tells the microprocessor to go back eleven bytes to the address 32506, labelled rpt. JR NZ stands for jump if the result of the previous instruction is not zero. In this case the 'previous instruction' was **DEC B (decrement the B register)**. Since the B register starts at five, every time it is decremented it becomes smaller, but not equal to zero. So the jump condition is obeyed and the program jumps back to line 32506 each time. It does this by adding -11 to the PC, thus making it point to the previous address. After the fifth decrement, the B register finally becomes zero, so the ZERO bit is set and the condition of the jump instruction is not obeyed. This time the PC is incremented to 32517 and the next instruction is then fetched from this address instead.

The reason for jumping back eleven bytes and not ten is as follows. Look at what happens if the ZERO bit is set, so that the jump condition is not obeyed. The JR NZ,-11 instruction is a two-byte instruction, starting at address 32515. After it has fetched the operand (-11), the PC is equal to 32516. The jump condition fails, so this instruction has now been completed and the PC is incremented to point to the next instruction, which is at address 32517.

Now suppose that the B register was not zero so that the ZERO bit is cleared. In this case the jump condition will be obeyed and -11 will be added to the PC, which will thus become 32505, since 32516 + (-11) = 32505. This is the end of the current instruction, so the PC is incremented (to 32506) and the next instruction is fetched from line 32506. This is exactly where we want to be. The rule, therefore, is as follows: all relative jump instructions must jump to the address immediately before the desired address, so that when the PC is incremented prior to fetching the next instruction, it then points to the correct address.

Let us see how this applies to the following program, which achieves the same as the one above:

```
32500          LD B,5          ;Set counter
32502          LD IX,32000     ;Set pointer
32506    rpt   LD A,1          ;Get value
32508          LD (IX+1),A     ;Save value
32511          INC A           ;Next value
32512          INC IX          ;Next location
32514          DEC B           ;Dec counter
32515          JR Z,+3
32517          JP 32506        ;Repeat cycle
32520                          ;remainder of program
```

This time line 32515 is a forward jump JR Z,+3. This is after the instruction DEC B and will thus be obeyed whenever the B register is zero. This does not occur for the first five loops, so the PC is incremented to point to address 32517, which is a jump to address 32506. On the last loop the B register becomes zero so the jump is obeyed and the PC becomes 32519 (i.e. 32516 + 3). This is the end of the current instruction, so the PC is incremented to point to the next instruction at address 32520. Note once again that the displacement added to the PC makes it point to the address immediately in front of the desired address. This is to allow for the fact that the PC is incremented before the next instruction is fetched. Of all the ideas in machine-code programming, this is probably the most difficult to get right.

There are several other conditions that can be tested before a jump. The only other one we shall deal with now is testing the CARRY bit. JP NC means jump if the CARRY bit is cleared and JP C means jump if the CARRY bit is set. Otherwise their use is identical to the JP NZ and JP Z instructions. There are also the relative jump instructions JR NC and JR C, and you should be able to guess what they mean.

Comparison

So far we have only looked at counting down to zero; this is too restrictive. To enable us to count up as well, the ability to compare two sets of data is essential. The **CP (compare)** instruction performs this function. It does not need the first operand to be specified, since this is always the ACCUMULATOR. Suppose the B register contains the data 5, the instruction CP B carries out the following steps:

1 The data in the operand (in this case the B register) is subtracted from the data in the ACCUMULATOR and the result is stored internally. The data in the ACCUMULATOR is not changed.
2 If the operand data is equal to the ACCUMULATOR data then the result will be zero and the ZERO bit will be set, otherwise it will be cleared. Thus if CP B is followed by JP Z, the jump will be obeyed if the ACCUMULATOR also contains 5.
5 If the operand data is greater than the ACCUMULATOR data, then the CARRY bit will be set to 1, indicating that a borrow has occurred. If the

operand data is not greater than the ACCUMULATOR data then the CARRY bit will be cleared. These conditions can be detected by the jump instructions **JP NC (jump if the CARRY bit is cleared)** and **JP C (jump if the CARRY bit is set)**.

To summarize:
CP 5 followed by JP NC will jump if the ACCUMULATOR data is greater than or equal to 5.
CP 5 followed by JP C will jump if the ACCUMULATOR data is less than 5.
CP 5 followed by JP Z will jump if the ACCUMULATOR data is equal to 5.
CP 5 followed by JP NZ will jump if the ACCUMULATOR data is not equal to 5.

The CP operation can have an operand in the immediate, direct, register or indexed modes. In all cases the data obtained from the operand will be compared with the ACCUMULATOR data.

Try each of the following sets of instuctions. Make sure that you understand why the jump operands have the values they do. Try to predict what each program should do, then see if you were correct. You will have to re-enter the instructions in the loop (32508 to 32516) three times over, because Z80 SIMULATION will not remember them. In a real microprocessor this will not be necessary.

Program to place the square of the number 3 into location 32005

```
32500          LD C,0          ;Set result to zero
32502          LD B,3          ;Set counter
32504          LD E,B          ;Keep value
32505    loop  LD A,E          ;Get value
32506          ADD A,C         ;Add result
32507          LD C,A          ;Keep new result
32508          DEC B           ;Dec counter
32509          JR NZ,-6        ;Repeat loop
32511          LD (32005),A
```

The loop adds together the contents of the E register, called **value** and the C register called **result**. This loop is performed a total of three times, initially set by the counter. The final result at the end will thus be 3 + 3 + 3 or 3 squared, which is finally stored in location 32005.

Problem for you to try:
28 What would happen if the JR NZ,-6 instruction in line 32509 were replaced by JR NZ,-7 or by JR NZ,-5?

The CALL instructions can also be conditional, as follows:

```
CALL 2000 is an unconditional jump to a subroutine.
CALL NZ,2000 means jump to subroutine if the ZERO bit is cleared.
CALL Z,2000 means jump to subroutine if the ZERO bit is set.
CALL NC,2000 means jump to subroutine if the CARRY bit is cleared.
CALL C,2000 means jump to subroutine if the CARRY bit is set.
```

The return from subroutine instructions have similar forms, thus:

```
RET
RET NZ
RET Z
RET NC
RET C
```

Negative numbers

So far we have written -5, say, to indicate a backward jump. The microprocessor needs some other way of indicating whether a number is positive or negative. It does this by a coding technique known as twos complement. A clue to this concept lies in the single-byte addition programs we wrote. We tried the effect of adding 255 to a number and noted that it reduced the value of that number by one. (Yes, the CARRY bit is also set, but we take no notice of that.) This is standard practice in binary arithmetic and is known as **subtraction by twos complement addition**. This relies on the phenomenon that 256 is actually equivalent to 0 if the CARRY bit is ignored. Hence 255, which is one less than zero, is equivalent to -1, which is also one less than zero. A table of some of these equivalents shows this more clearly:

Positive decimal	Positive binary	Twos complement binary	Negative decimal
128	1000 0000	1000 0000	-128
127	0111 1111	1000 0001	-127
126	0111 1110	1000 0010	-126
125	0111 1101	1000 0011	-125
41	0010 1001	1101 0111	-41
40	0010 1010	1101 1000	-40
39	0010 1011	1101 1001	-39
38	0010 1100	1101 1010	-38
30	0001 1110	1110 0010	-30
20	0001 0100	1110 1100	-20
10	0000 1010	1111 0110	-10
2	0000 0010	1111 1110	-2
1	0000 0001	1111 1111	-1
0	0000 0000	0000 0000	0

An inspection of the table shows that we can now represent both positive and negative numbers with binary numbers, depending upon which form of binary coding us being used. Twos complement coding represents numbers in the range -128 to +127 only, and it is possible to tell the negative numbers, because their most significant bit (bit 7, at the left end) is always 1. For all positive numbers this bit is 0. Thus we need only test the most significant bit position to see if it is a 1 or a 0. The Z80 microprocessor is aware of this need and sets the **SIGN bit** in the flag register to tell us if a number is positive or negative. We do not have to bother

about this unless we want to make use of twos-complement coding. The numbers behave quite normally and it is up to us to decide what we want those numbers to represent. The operand of a relative jump instruction is the number of bytes of the machine-code program to be skipped over. This is not difficult to calculate, provided you remember that the program counter is incremented immediately before the next byte of an instruction is fetched from memory. So a jump must go to the byte immediately preceding the desired instruction.

In the case of forward branching, one counts up in the normal way until one reaches this preceding byte. The number obtained is the required operand. For example:

```
16100    AND 0
16102    JR Z,+2
16104    yyy zz
16106    ppp qq
```

After JR Z,+2, the PC is at memory location 16103. To this is added +2, giving 16105 and the PC is then incremented to 16106. The next instruction is fetched from 16106. Instruction ppp qq will be executed next after the jump instruction.

For backward jumping the operand should really be a twos-complement number, but Z80 SIMULATION has been programmed to accept negative numbers instead. Later we shall have to do this properly, but for the moment we can ignore this.

This idea of twos-complement coding is used for another type of conditional jump. If bit 7 is set to 1 after a previous instruction, then the number is regarded as negative, if bit 7 is 0 then the number is regarded as positive. So the operation **JR M (jump if minus)** will succeed if bit 7 is a 1 and the operation **JR P (jump if plus)** will succeed if bit 7 is cleared to 0. For example:

```
16100    OR 255
16102    JR M,+2
16104    yyy zz
16106    ppp qq
```

will cause instruction ppp qq to be executed next after the jump instruction, whereas

```
16100    OR 255
16102    JR P,+2
16104    yyy zz
16106    ppp qq
```

will cause instruction yyy zz to be executed next after the jump instruction. In both cases the microprocessor tests another bit in the flag register, called the **SIGN bit**. Z80 SIMULATION does not have room to display this bit, but it checks it nevertheless.

Bit instructions

The following program can be used to test bit 2 of location 32001 to see if it is HIGH or LOW:

```
LD (32001),A
AND 4
```

This may then be followed by a conditional jump instruction like JP Z. If the bit 4 is LOW, the jump will be obeyed. The Z80 has a whole set of instructions which carry out operations on single bits, thus avoiding the above, rather clumsy, method. For example:

```
BIT 7,B
```

tests bit 7 in the B register. If it is LOW, the ZERO bit is set, if HIGH, the ZERO bit is cleared. To set one particular bit in a register to zero, there is

```
SET 3,A
```

and to clear (or resest) a particular bit, there is

```
RES 5,C
```

These instructions are very useful for switching bits on and off in control situations.

Shift instructions

This set of instructions is often used in binary multiplication and division. Multiplying by ten with decimal numbers holds no terrors, we simply add a 0. Similarly, in binary, multiplication by two is accomplished by adding a 0. The instruction to do just that is **SLA (shift left arithmetic)**. This causes each bit in the specified register to move into the next position left, with 0 loaded into the lowest bit. If the number was originally greater than 127 (or negative in twos complement coding) then the 1 originally in bit 7 is shifted into the CARRY bit. For example:

```
LD A,81
SLA A
```

(The ACCUMULATOR now contains 162.)

```
SLA A
```

(The ACCUMULATOR contains 68 and the CARRY bit contains 1 (which is really 256), so that 256 + 68 = 324.)

Two-byte shifting can also be performed, by allowing the CARRY bit to be shifted into bit 0 of the high byte. This is done with the **RL** instruction (**rotate left**). This causes the CARRY bit from the low byte (if any) to be shifted into bit 0 of the high byte (Figure 6.2). For example:

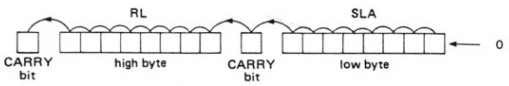


Figure 6.2 Right shifting on two bytes

```
LD C,181
LD B,0
SLA C
RL B
SLA C
RL B
SLA C
RL B
```

will cause the original two-byte number (181,0) to be multiplied by eight.

A similar set of instructions can be used to divide by two. This time the routine starts with the high byte and performs an **SRL** instruction (**shift right logical**) on it. Bit 7 becomes 0, bit 6 becomes equal to the previous value of bit 7, etc. and the contents of bit 0 are shifted into the CARRY bit. This instruction can be followed by an **RR** instruction (**rotate right**) and the CARRY bit is pushed into bit 7 of the low byte. Bit 0 of the low byte is pushed into the CARRY bit itself (Figure 6.3). (This is very useful for determining if the original number was odd or even, since only an odd number leaves the CARRY bit set to 1.)

In several cases so far we have carried out additions, etc. on two bytes separately. The Z80 instruction set contains a whole series of instructions that operate on two bytes at once. You will have noticed that the registers are grouped in pairs, called the **BC register pair, the DE register pair** and the **HL register pair**. In each case the first register contains the high byte and the second contains the low byte. The number of register pair instructions available is far less than for a single register, but they are correspondingly more powerful. For example, here is a program to add together two double-byte numbers:

```
LD HL,6000
LD DE,8000
ADD HL,DE
```

You will observe that the result is left in the HL register pair. In a sense the

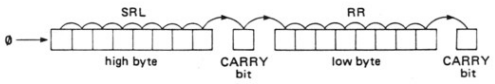


Figure 6.3 Left shifting on two bytes

HL register pair is acting like the ACCUMULATOR - it is the normal place for the result of a double-byte calculation.

It is possible to load a register pair directly from memory. But since the memory consists of single bytes, the register pair loads two successive bytes from the memory. For example:

```
LD BC,(32001)
```

loads the contents of location 32001 into the C register and the contents of location 32002 into the B register. (Note once again the low/high order.) The instruction

```
LD (32001),BC
```

does the reverse of this. This applies to the X-INDEX too.

A few arithmetic operations also exist for register pairs. We have already seen INC X and DEC X and these also exist as INC BC, DEC HL, etc. Above we introduced ADD HL,BC and this is paralleled by the following too:

```
ADD HL,DE
ADC HL,BC
SBC HL,DE
```

There is no simple SUB instruction when using register pairs. To be certain that the CARRY bit is not included when it should not be, it can be cleared by a dummy instruction, such as:

```
AND A
```

which does not affect the ACCUMULATOR, but does clear the CARRY bit.

The most valuable use of the register pairs is for a new and powerful addressing mode - **indirect addressing**.

```
LD HL,32001
LD A,(HL)
```

causes the contents of location 32001 to be loaded into the ACCUMULATOR. It works like this. The first instruction sets the HL register pair to contain 32001. In the instruction LD A,(HL), the (HL) is effectively replaced by (32001). So instead of loading the contents of HL, this instruction treats those contents as an address, from which the data is collected. This is why it is called an 'indirect' instruction. It is a powerful addressing mode, since it is easy to increment or decrement the HL register pair to point to the next memory location. In fact any number can be added or subtracted from the HL register pair, using the arithmetic instructions mentioned above.

The HL register pair can be used like this in place of any of the other registers. Here are a few examples:


```
BIT 5,(HL)
SET 3,(HL)
RES 1,(HL)
LD B,(HL)
LD (HL),A
ADD A,(HL)
SBC A,(HL)
AND (HL)
CP (HL)
INC (HL)
```

Indirect jumps can also be carried out.

```
JP (HL)
```

jumps to the address contained in the HL register pair. Since these contents can be adjusted easily, this gives the Z80 the facility for a computed GOTO.

The STACK

Another set of instructions is concerned with the STACK, which is a double-byte register. The instruction **PUSH BC** makes the stack pointer move down to point to the next STACK position and then pushes the contents of the BC register pair onto the STACK, for temporary storage. The reverse instruction **POP HL** will pull the contents of the current position off the STACK, place them in the HL register pair and make the stack pointer move up one. Try these examples:

```
LD BC,32000
PUSH BC
LD DE,21000
PUSH DE
POP BC
POP DE
```

and watch the movement of the stack pointer in each case.

This sequence swaps over the contents of the BC and DE register pairs. There is, however, a single instruction for swapping the HL and DE register pairs directly. This is:

```
EX DE,HL
```

This does not, however, work with any of the other register pairs.

No operation

The most mystifying instruction must surely be NOP (no operation). It simply causes the microprocessor to waste time. None of the registers is affected in any way, except for the PC, which is incremented to point to the next instruction. The main use of NOP is to adjust delay loops to get the correct delay time. Try it for yourself and see its lack of effect.

DIY

Try out your own ideas using Z80 SIMULATION. Apart from the restriction on memory locations (range 32001 to 32007 only) the simulation supports most Z80 instructions. A limited set of the instructions that can be implemented has been described in this chapter, so try them out for yourself. There is no doubt that the ability to handle the mnemonics properly does speed up the writing of machine-code programs.

Machine-code programming

This brief tour of the Z80 microprocessor has shown only some of the available instructions and their effects on the internal registers of the microprocessor. You should now be able to move on to the more exciting challenge of putting these instructions into a real machine-code program and seeing their overall effect.

Although we have used mnemonics, the microprocessor stores all the instructions and data as binary codes or numbers. When we start to write machine-code programs, we shall have to translate our mnemonics into these codes, so that the microprocessor understands. The way that this is done depends on the addressing modes used. The numbers may be in decimal or hexadecimal (see Chapter 2), so both are given.

i) Immediate mode The operand is the actual number to be loaded into the register.

Decimal code	Hex. code	Mnemonic
62,42	3E 2A	LD A,42

ii) Direct mode If we want to load the ACCUMULATOR with the contents of a particular memory location, we use the direct addressing mode, in which the operand is an address. The microprocessor goes to that address to find the number to be loaded into the ACCUMULATOR. Since there are so many different addresses, the operand consists of two bytes, the low byte of the address followed by the high byte. As an example, we could fetch the contents of address 32768 (8000H):

58,0,128	3A 00 80	LD A,(32768)

The brackets indicate the direct addressing mode. Note that the numeric code for LD is different from that in the immediate addressing mode.

iii) Indirect mode The HL register pair holds the address of the data.

33,2,0	21 02 00	LD HL,2
126	7E	LD A,(HL)

iv) Indexed addressing The final address is calculated by adding the X-INDEX to the displacement. The microprocessor then goes to this final address to get the desired data.

221,33,0,125	DD 21 00 7D	LD IX,32000
221,126,5	DD 7E 05	LD A,(IX+5)

The complete set of Z80 codes is not given here, reference should be made to the books in the Bibliography. Even then the lists of codes are not always helpful. By writing them in a particular way, a definite order can be discerned. The following lists show the decimal codes only.

LD instructions (eight bit)

Destination register	Source register B	C	D	E	H	L	(HL)	A
B	64	65	66	67	68	69	70	71
C	72	73	74	75	76	77	78	79
D	80	81	82	83	84	85	86	87
E	88	89	90	91	92	93	94	95
H	96	97	98	99	100	101	102	103
L	104	105	106	107	108	109	110	111
(HL)	112	113	114	115	116	117	118	119
A	120	121	122	123	124	125	126	127

Arithmetic and logic instructions (eight bit)

In each case the destination register is the ACCUMULATOR.

Instruction	Source register B	C	D	E	H	L	(HL)	A	N(direct)
ADD	128	129	130	131	132	133	134	135	198
ADC	136	137	138	139	140	141	142	143	206
SUB	144	145	146	147	148	149	150	151	214
SBC	152	153	154	155	156	157	158	159	222
AND	160	161	162	163	164	165	166	167	230
XOR	168	169	170	171	172	173	174	175	238
OR	176	177	178	179	180	181	182	183	246
CP	184	185	186	187	188	189	190	191	254

Other instructions (eight bit)

Destination register	Instruction INC	DEC	LD reg,number (direct)
B	4	5	6
C	12	13	14
D	20	21	22
E	28	29	30
H	36	37	38
L	44	45	46
(HL)	52	53	54
A	60	61	62

Solutions to problems

1
```
LD A,50
LD (32002),A
```
2
```
LD A,(32007)
LD (32006),A
```
3
```
LD A,1
LD (32001),A
LD A,2
LD (32002),A
LD A,3
LD (32003),A
```
4
```
LD A,1    or    LD A,1
LD B,1          LD B,A
LD C,1          LD C,A
LD D,1          LD D,A
LD E,1          LD E,A
LD H,1          LD H,A
LD L,1          LD L,A
```
5
```
LD A,(32001)
LD C,A
```
6
```
LD A,H
LD (32001),A
```
7 A 'write' places data from the ACCUMULATOR into a byte of memory. A 'read' takes the data from the memory and places it in the ACCUMULATOR. LD A,(32005) is a read instruction.

8 (i) 512
(ii) 3082
(iii) 10440
(iv) 20480
(v) 60000

9 (i) 0,1
(ii) 0,4
(iii) 64,31
(iv) 0,32
(v) 255,255

10
```
LD A,45
ADD A,54
```
11
```
LD A,(32004)
LD E,A
LD A,(32005)
ADD A,E
LD (32003),A
```
Note: E could also be any of B, C, D, H or L.

12
```
LD A,194
LD (32001),A
LD A,1
LD (32002),A
LD A,28
LD (32003),A
LD A,2
LD (32004),A
LD A,(32001)
LD E,A
LD A,(32003)
ADD A,E
LD (32005),A
LD A,(32002)
LD E,A
LD A,(32004)
ADC A,E
LD (32006),A
```
13
```
LD A,225
LD (32001),A
LD A,200
LD (32002),A
LD A,(32001)
LD E,A
LD A,(32002)
ADD A,E
LD (32003),A
LD A,0
LD E,0
ADC A,E
LD (32004),A
```
14
```
LD A,232
LD (32005),A
LD A,3
LD (32006),A
LD A,(32005)
ADD A,1
LD (32005),A
LD A,(32006)
ADC A,1
LD (32006),A
```
15 1 and 65535
2 and 65534
3 and 65533, etc.

16 LD A,1
SUB 1
17 LD A,5
LD (32001),A
LD A,9
LD (32002),A
LD A,(32002)
LD E,A
LD A,(32001)
SUB E
LD (32003),A
18 LD A,78
LD (32001),A
LD A,29
LD (32002),A
addition
LD A,(32001)
ADD A,78
LD (32003),A
LD A,(32002)
ADC A,29
LD (32004),A
subtraction
LD A,(32001)
LD E,A
LD (32003),A
SUB E
LD A,(32003)
LD A,(32002)
LD E,A
LD (32004),A
SBC A,E
LD (32004),A
The result is the original number.
19 LD A,0
LD (32001),A
LD (32002),A
LD A,(32001)
SUB 1
LD (32001),A
LD A,(32002)
SBC A,0
LD (32002),A
The result is 255,255 or −1 in twos complement coding.

20 LD B,0
LD A,3
INC B
DEC A
INC B
DEC A
INC B
DEC A
The B register contains 3.
21 85 AND 45 is 5
22 85 OR 45 is 125
23 85 XOR 45 is 120
24 LD A,(32005)
AND 252
LD (32005),A
25 LD A,(32005)
OR 127
LD (32005),A
26 The PC would contain 20000.
27 The microprocessor would continuously jump to location 12000 forever (like the BASIC statement 10 GO TO 10).
28 The microprocessor would jump to the middle of an instruction, which it would interpret wrongly. The program would probably crash.

7 Machine-code graphics

'If you want to get somewhere else, you must run at least twice as fast as that!'
(Lewis Carroll, *Through the Looking Glass*)

Microcomputers are not designed for running machine-code programs in the same way that they are for BASIC. There are generally more problems in entering, saving, loading and running such programs. In particular, machine-code programs contain no error-checking procedures like BASIC. If you ask the microcomputer (in BASIC) to divide by 0, it will stop and tell you that this is not possible. If you tell a microprocessor to jump to the wrong address, it will still jump and may cause a crash. This may mean that you lose all control of the machine and have to reset to regain control.

Crashes are quite common in machine-code programming and unfortunately the Spectrum cannot recover from such events without losing the program. This is not a disaster provided you saved the program on cassette tape before it was run. Cultivate the habit of saving and verifying every program before you run it, otherwise you will spend fruitless hours typing it in all over again.

Machine-code graphics

Machine-code graphics give a particularly good introduction to machine-code programming in general, as well as being important in their own right. The screen gives a visible record of the contents of the memory locations, so direct observations on the course of the program can be made. In Chapter 2 we looked at BASIC methods of making the *-character move around the screen. A machine-code program allows instructions to go directly to the microprocessor, which means that it doesn't have to waste time translating BASIC instructions, it can get on with the job immediately. Thus machine-code programs run several hundred times faster than their BASIC equivalents. For example, a routine to bounce a ball around the screen is perfectly satisfactory in BASIC (in fact a PAUSE has to be introduced to slow it down). So games like BRICKOUT can be written without machine code. But try moving a hundred molecules around the screen and the result is jerky and unacceptable. This chapter looks at those graphics applications that require machine-code programs and then offers some advice on machine-code programming in general.

A machine-code program to fill the screen with *-characters is not at all easy, so we revert to the method used in Chapter 2. We first fill every screen position with a

*-character, setting the attributes of each position, so that the *-characters become invisible. the following BASIC program does this:

```
10  INK 7:PAPER 7
20  FOR Y=0 to 21
30  PRINT AT Y,0;"********************************"
40  NEXT Y
```

A BASIC program to display these characters, by changing the ink to black in the ATTRIBUTES file, is as follows:

```
100  FOR X = 22528 TO 23295
110  POKE X,56:REM Turn ink to black
120  NEXT X
```

The machine-code PRINT statement is somewhat difficult to use, so let us keep this part of the program in BASIC. The second part is much simpler, consisting of sending the value 56 to each of 768 memory locations (the ATTRIBUTES). This can be achieved for the top row of stars thus:

```
        LD B,32          ;32 *-characters
        LD HL,5800H      ;First attribute position
        LD A,56          ;Black ink, white paper
next    LD (HL),A        ;Place * in screen position
        INC HL           ;Move to next position
        DEC B            ;Decrement counter
                         ;All positions done?
        JR NZ,-5         ;No,do next position
        RET              ;Yes, so finish
```

The program works like this. First the B register is initialized to count up to thirty-two (since each line on the screen has thirty-two characters). Next the HL register pair is initialized to point to the first ATTRIBUTE position (5800H is 22528 in decimal). Note that addresses should be left in hexadecimal, since they are then easier to convert to low byte, high byte pairs. The number 56, loaded into the ACCUMULATOR, produces black ink and white paper in each ATTRIBUTE position. We are now ready to begin.

The first memory location (first ATTRIBUTE position) is loaded with the contents of the ACCUMULATOR, using the indirect addressing mode. Then the HL register pair is incremented to point to the next position. The counter is decremented and, if it has not yet reached zero, the program jumps back to the label 'next' and repeats for the next ATTRIBUTE position. This continues for thirty-two loops (the top row on the screen), at which point the jump instruction fails and the program moves on to the RET instruction. This sends the Spectrum microcomputer back to BASIC, with the results of the machine-code routine clearly visible.

The listing for this program has a label column (containing the label 'next'), a mnemonics column and a comment column. Only the mnemonics are significant

for the process of turning the program into code. Each one has to be converted into the binary codes that the microprocessor understands. There are three ways of doing this, two of which involve hand compilation as follows. Each mnemonic is looked up in a table and converted into its correct decimal or hexadecimal code. Displacements are carefully converted to twos complement numbers and actual addresses calculated and split into their high byte, low byte components. Finally, the codes are entered into the memory from BASIC. Let us tackle each of these tasks, using decimal numbering rather than hexadecimal.

Coding

Each of the mnemonics in the program must first be converted to its proper decimal code. A complete list of all the codes is not given, but a selection was listed at the end of Chapter 6. For a complete list consult the books in the Bibliography or the table at the back of the Spectrum user guide. The result of this hand assembly is as follows:

```
6,32          LD B,32              ;32*-characters
33,0,88       LD HL,5800H          ;First attribute position
62,56         LD A,56              ;Black ink, white paper
119  next     LD (HL),A            ;Place * in screen position
35            INC HL               ;Move to next position
5             DEC B                ;Decrement counter
                                   ;All positions done?
32,251        JR NZ,-5             ;No, do next position
201           RET                  ;Yes, so finish
```

The next step is to decide where these codes are to be put in the memory. As noted in Chapter 6, short machine-code programs can be placed at the location 32000 upwards. To prevent BASIC from using these locations too, we first tell the Spectrum to keep clear of it. This is achieved with the BASIC statement:

```
CLEAR  32000
```

We must now put each code into its correct position. Let us first see where these are. Here is the program again (with the comment column omitted) showing where the codes will be placed:

```
32000    6,32          LD B,32
32002    33,0,88       LD HL,5800H
32005    62,56         LD A,56
32007    119  next     LD (HL),A
32008    35            INC HL
32009    5             DEC B
32010    32,251        JR NZ,-5
32012    201           RET
```

This is even more obvious if each byte is written out separately:


```
32000    6
32001    32
32002    33
32003    0
32004    88
32005    62
32006    56
32007    119
32008    35
32009    5
32010    32
32011    251
32012    201
```

Turning the mnemonics into machine codes in this way is called **compiling the program**. The next step is to get the codes into their chosen locations. It is possible to put each code into its proper place with a succession of POKE commands, but the following program makes this easier.

```
10   CLEAR 32000
20   FOR i = 32000 TO 32012
30   INPUT x
40   POKE i,x
50   NEXT i
```

The correct numbers would then be INPUT from the keyboard when this program is run.

To check that the codes are correct, another program can be used, as follows:

```
60   FOR i = 32000 to 32012
70   PRINT i, PEEK i
80   NEXT i
```

Now the codes are in place. Before the program can be run, we need to set up the screen, so that the top row is filled with invisible stars. This can be done with:

```
90   INK 7:PAPER 7
100  PRINT AT 0,0;"********************************"
```

Finally, to execute the machine-code program, we have to tell the microprocessor to go and collect its instructions from location 32000 onwards. There are several ways of doing this:

```
RANDOMIZE USR 32000
LET k = USR 32000
```

or any other statement that contains USR 32000 will do, such as

```
PRINT USR 32000
```

This executes the machine-code routine and, upon returning to BASIC, prints the current value of the BC register pair. This can be a useful check for a routine that seems to be working wrongly. We shall use

```
110  LET k = USR 32000
```

Control passes to the routine starting at location 32000 and continues until a RET instruction (code 201) is reached. This is why a machine-code program has to be perfect every time. If, for any reason, the microprocessor does not find the RET instruction, it will never return to BASIC. Pressing the keys has no effect and the screen may remain blank. The only solution is to reset, (switch off the microcomputer, wait a few seconds and then switch on again).

If you have not yet done so, enter, save, verify and run this program. You should observe that the *-characters all appear at the same instant (even though there is an initial delay while the invisible stars are printed).

To save the machine-code program on tape is just a matter of using the correct Spectrum syntax.

```
SAVE "starbytes" CODE 32000,13
```

The '32000' is the address of the codes and the '13' is the number of bytes to be saved. This information is also stored on the tape. Apart from this, the SAVE routine is exactly the same as for a BASIC program. However, we also need to save line 10 and lines 90 to 110 of the above program. These are not in the machine-code routine, yet they are essential if it is to work correctly. This is done in the usual way.

Reloading the program from the tape also needs to be done in two stages. First the BASIC part of the program is loaded in the normal way. Then the machine-code part is loaded with:

```
LOAD "starbytes" CODE
```

This collects the codes from the tape and puts them into the address originally specified, when they were saved. Instead of the usual signal 'Program:', the signal 'Bytes:' is given to show that a machine-code program is being loaded. After this the program can be run as before.

Automatic load and run

It is possible to make the BASIC program load its own machine-code routine. The technique is to save the BASIC part first including a LOAD . . . CODE instruction. To show this the whole of this program is written out again.

```
10  CLEAR 32000
20  LOAD "starbytes" CODE
```



```
90  INK 7:PAPER 7
100 PRINT AT 0,0;"********************************"
110 LET k = USR 32000
```

This should be saved first using the command

```
SAVE "stars" LINE 10
```

Then the machine codes should be saved with

```
SAVE "starbytes" CODE 32000, 13
```

making sure that the program name is the same as in the LOAD . . . CODE statement. To load and run, rewind the tape and type

```
LOAD "stars" or even LOAD " "
```

and both parts of the program will be loaded and run automatically. The whole process is messy and cumbersome, so if you want a much better way, read on!

A BASIC loader

The technique just described suffers from one major difficulty (in addition to the hassle). If a program needs to be debugged, how do you edit it? It is quite possible to make a few changes with a POKE into the location containing the offending byte. But often you need to insert extra instructions and all the existing codes must then be moved to new locations. This means that the machine codes and the BASIC program to load and run them must be saved again.

A much better method is to put the machine codes into DATA statements as an integral part of the BASIC program. For example:

```
110  FOR i = 32000 TO 32012
120  READ x
130  POKE i,x
140  NEXT i
150  DATA 6,32,33,0,88,62,56,119,35,5
160  DATA 32,251,201
```

Each DATA statement (except the last) should contain ten bytes, for later ease of checking. The complete program is now as follows:

```
10   INK 7:PAPER 7
20   FOR Y = 0 TO 21
30   PRINT AT Y,0;"********************************"
40   NEXT Y
100  CLEAR 32000
110  FOR i = 32000 TO 32012
120  READ x
130  POKE i,x
140  NEXT i
```

```
150   DATA 6,32,33,0,88,62,56,119,35,5
160   DATA 32,251,201
200   LET k = USR 32000
```

You may save and verify this program as if it were entirely in BASIC (which it is!). For an automatic start, it can also be saved with

```
SAVE "stars" LINE 10
```

This technique is much less bother than the one described above. It is true that it takes up far more memory, since each byte of code now takes up several bytes of memory. (The machine-code part of the above program requires over 120 bytes, yet it only consists of 13 bytes of code.) This vast consumption of memory does not matter unless you have a very long machine-code program. If that is the case, you should not be using either of these methods in the first place!

Machine-code monitor

It should be apparent that loading and running machine-code programs is a complicated business. The situation can be improved by using a machine-code monitor. This is a program that is loaded into the Spectrum beforehand. It allows machine codes to be entered in hexadecimal, which is much easier because most books on machine-code programming use hexadecimal. The best program that I have tried so far is the Picturesque monitor, available through Griffin and George. This allows memory locations to be loaded with the hexadecimal codes, checked and saved with the minimum of hassle. It also contains a hex.-dec./dec.-hex. converter as a useful addition.

To allow the insertion of extra code, the monitor lets you move chunks of machine code around from one location to another or to display the contents of any part of the memory. Even more useful is the ease with which machine-code programs can be run. At various stages throughout the machine-code program, the monitor allows for the insertion of breakpoints. The program can be run up to these points and the status of the different Z80 internal registers displayed. This makes it easier to locate bugs. Finally, the Picturesque monitor contains a disassembler, which is a useful check on whether the correct codes were entered in the first place. This takes the machine codes and turns them back into mnemonics.

Assembler

Even with its impressive aids, a monitor is not the best way of entering a machine-code program. An assembler makes life easier still. This is the third and best way of entering machine-code routines. An assembler is basically a program that allows machine-code routines to be written directly in mnemonics. The assembler does the job of working out the correct codes for each instruction. It also provides facilities for editing the mnemonics and for discovering where they need to be edited.

The Picturesque assembler/editor fulfils both functions. In the 48K Spectrum it is possible to have the machine-code monitor as well, thus giving the best of all worlds (apart from BASIC, that is). The Picturesque editor allows you to write, edit and save program lines, in much the same way that you can in BASIC. Lines of assembly mnemonics can be numbered for reference, so that extra lines can be inserted or redundant ones removed. It is possible to display any line or lines of the program and alter them as required.

The Picturesque assembler then compiles the program (turns the mnemonics into code), and it keeps the resulting code in any part of the memory, not necessarily the same place where it will eventually reside. This is because the assembler program normally sits at the top of the memory, which is precisely where machine codes usually go. The code is kept in a buffer, from where it can be relocated (put somewhere else) or saved on tape and verified. Syntax errors are reported during or after the compilation. This is an important difference between an interpreter (like the BASIC interpreter) and a compiler. In BASIC, syntax errors are not reported until they are reached. With a compiler, syntax errors are fatal. You cannot just compile the bits of the program that are correct; it must all be correct beforehand.

The great advantage of an assembler is that symbols and labels may be used. Symbols are like variable names in BASIC. Instead of writing LD B,32, we could declare the variable **max** and write LD B,max instead. This makes the machine-code programs more meaningful. labels are the names of lines to which we shall want to jump, such as the label 'next' in the program above. Spectrum BASIC objects if a variable is used before it has been **declared** (given a value). This often happens in machine code, because of the need to jump forward to a label that has not yet been given an address. The Picturesque assembler solves this by making a first pass through the mnemonics to assign values to all labels and symbols and a second pass to carry out the compilation.

In the program above, comments were included to explain what the program is doing. The Picturesque assembler allows such comments, provided they are enclosed in inverted commas and follow a semi-colon. It also accepts directives. These are instructions for the compiler, such as ORG, which tells it the ultimate location of the code or EQU, which allows a symbol to be given a value. For example, to declare the variable max, we say:

```
max EQU 32
```

To show what an assembly language program looks like, the stars program above is rewritten in the form acceptable to the Picturesque assembler.

```
1000                ORG 32000
1010    ;           "Stars"
1020    ;
1030    max EQU 32
1040    screen EQU 5800H
```

```
1050      char  EQU 56
1060      ;
1070      begin LD B, max
1080            LD HL, screen
1090            LD A,char
1100      next  LD (HL),A
1110            INC HL
1120            DEC B
1130            JR NZ,next
1140            RET
```

When this program is compiled, the assembler prints a report, which looks something like this:

```
32000 06 20       LD B,max
32002 21 00 58    LD HL,screen
32005 3E 38       LD A,char
etc.
```

General format

In listing machine-code programs throughout the remainder of this book, I am aware that readers might be using any of the three methods mentioned above. I have, therefore, adopted the policy of writing most programs in the following format:

```
max       = 32
screen    = 5800H
char      = 56

32000 06 20             LD B,max          ;32 *-characters
32002 21 00 58          LD HL,screen      ;First attribute position
32005 3E 38             LD A,char         ;Black ink, white paper
32007 77          next  LD (HL),A         ;Place * in screen position
32008 23                INC HL            ;Move to next position
32009 05                DEC B             ;Decrement counter
                                          ;All positions done?
32010 20 FB             JR NZ,next        ;No, do next position
32012 C9                RET               ;Yes, so finish

DATA 6,32,33,0,88,62,56,119,35,5
DATA 32,251,201
```

This is a compromise, but can be readily understood by users of any of the techniques described above. Those with an assembler can enter the mnemonics, those with a monitor can enter the hex. codes and those with neither can use a BASIC loader and the data statements, which contain the codes in decimal. All

BASIC programs listed in the Appendix use the BASIC loader method. Let us now begin machine-code graphics in earnest.

Screenprint

Our stars program placed the *-characters on the screen in BASIC with the statement PRINT AT row,column;"*". Let us now see how to do this in machine code. The Spectrum allows us to make use of its machine-code printing routines for ourselves. The technique is to send the correct codes to a special subroutine, which is entered with the instruction RST 10H. The codes are carried to this routine in the ACCUMULATOR. This machine-code program will do exactly the same as PRINT AT 0,0;"*".

```
32000   3E 16    LD A,16H    ;Send PRINT AT
32002   D7       RST 10H
32003   3E 00    LD A,00H    ;Send row
32005   D7       RST 10H
32006   3E 00    LD A,00H    ;Send column
32008   D7       RST 10H
32009   3E 2A    LD A,2AH    ;Send *-character
32011   D7       RST 10H
32012   C9       RET         ;Finish
```

The BASIC loader and program to run this routine is given below.

```
10   CLEAR 32000
20   FOR i = 32000 TO 32012
30   READ x
40   POKE i,x
50   NEXT i
60   DATA 62,22,215,62,0,215,62,0,215,62
70   DATA 42, 215, 201
100  RANDOMIZE USR 32000
```

The memory space from 32000 upwards has been reserved for our use, even though our machine-code program only uses thirteen bytes. This is because we shall want to add more code to this program later.

This technique can be used to print any character in any position, even a user-defined character. The code used for the character is its ASCII code. A full list of these for the Spectrum is given in the user guide (Appendix A). To make the routine more general, we need variables in place of the 0,0 values (row, column). We could use two memory locations for this, or two of the internal registers, such as the D and E registers. If we do this, the C register could be used to hold the ASCII code for the character to be displayed. The routine then becomes:

```
32000   3E 16    LD A,16H    ;Send PRINT AT
32002   D7       RST 10H
32003   7A       LD A,D      ;Send row
```

```
32004   D7      RST 10H
32005   7B      LD A,E      ;Send column
32006   D7      RST 10H
32007   79      LD A,C      ;Send character
32008   D7      RST 10H
32009   C9      RET         ;Finish
DATA 62,22,215,122,215,123,215,121,215,201
```

Now the parameters for the row, column and the character can be passed to this routine from outside. We can treat it as a subroutine, exactly as with BASIC. This program writes a whole name in the middle of the screen.

```
32500   16 0A       LD D,10         ;Row 10
32502   1E 10        LD E,16         ;Column 16
32504   0E 42        LD C,42H        ;Letter B
32506   CD 00 7D     CALL 32000      ;PRINT AT 10,16;"B"
32509   16 0A        LD D,10         ;Row 10
32511   1E 11        LD E,17         ;Column 17
32513   0E 6F        LD C,6FH        ;Letter o
32515   CD 00 7D     CALL 32000      ;PRINT AT 10,17;"o"
32518   16 0A        LD D,10         ;Row 10
32520   1E 12        LD E,18         ;Column 18
32522   0E 62        LD C,62H        ;Letter b
32524   CD 00 7D     CALL 32000      ;PRINT AT 10,18;"b"
32527   C9           RET             ;Return to BASIC
```

The BASIC loader and program to run this is given below:

```
10   CLEAR 32000
15   REM SUBROUTINE
20   FOR i = 32000 TO 32009
30   READ x
40   POKE i,x
50   NEXT i
60   DATA 62,22,215,122,215,123,215,121,215,201
100  REM MAIN PROGRAM
110  FOR i = 32500 TO 32527
120  READ x
130  POKE i,x
140  NEXT i
150  DATA 22,10,30,16,14,66,205,0,125,22
160  DATA 10,30,17,14,111,205,0,125,22,10
170  DATA 30,18,14,98,205,0,125,201
180  REM
500  REM RUN PROGRAM
510  RANDOMIZE USR 32500
```


This is still not very flexible, a better method would be to write all the required characters into succesive memory locations first. Another table could hold the row position and another the column position for each character. We could then just run down the tables, collect the values and send them to our printing subroutine. By using the X-INDEX as a pointer to the table, we could increment this to point to the next character each time. The table is kept in the memory locations 32256 (7E00H) upwards.

Location	Contents	Character
7E00H	42H	B
7E01H	6FH	o
7E02H	62H	b
7E03H	20H	(space)
7E04H	53H	S
7E05H	70H	p
7E06H	61H	a
7E07H	72H	r
7E08H	6BH	k
7E09H	65H	e
7E0AH	73H	s
7E0BH	20H	(space)
7E0CH	31H	1
7E0DH	39H	9
7E0EH	38H	8
7E0FH	33H	3

Location	Contents	Row
7E10H	0AH	10
7E11H	0AH	10
7E12H	0AH	10
7E13H	0AH	10
7E14H	0AH	10
7E15H	0AH	10
7E16H	0AH	10
7E17H	0AH	10
7E18H	0AH	10
7E19H	0AH	10
7E1AH	0AH	10
7E1BH	0AH	10
7E1CH	0AH	12
7E1DH	0AH	12
7E1EH	0AH	12
7E1FH	0AH	12

Location	Contents	Columns
7E20H	10H	16
7E21H	11H	17
7E22H	12H	18
7E23H	13H	19
7E24H	14H	20
7E25H	15H	21
7E26H	16H	22
7E27H	17H	23
7E28H	18H	24
7E29H	19H	25
7E2AH	1AH	26
7E2BH	1BH	27
7E2CH	13H	19
7E2DH	14H	20
7E2EH	15H	21
7E2FH	16H	22

Display program

```
32500 06 10          LD B,16          ;Display 16 characters
32502 DD 21 00 7E    LD IX,32256      ;Point to start of table
32506 DD 56 10  next LD D,(IX+10H)    ;Row
32509 DD 5E 20       LD E,(IX+20H)    ;Column
32512 DD 4E 00       LD C,(IX+00H)    ;Character
32515 CD 00 7D       CALL 32000       ;PRINT AT row,column;char
32518 DD 23          INC IX           ;Point to next character
32520 05             DEC B            ;All characters done?
32521 20 EF          JR NZ,next       ;No do next character
32523 C9             RET              ;Yes return to BASIC
```

The BASIC loader and program to run this is given below. This is the last time that this will be given, in future you will have to work out for yourselves what values to put in the FOR . . . NEXT loops.

```
10   CLEAR 32000
15   REM SUBROUTINE
20   FOR i = 32000 TO 32009
30   READ x
40   POKE i,x
50   NEXT i
60   DATA 62,22,215,122,215,123,215,121,215,201
100  REM MAIN PROGRAM
110  FOR i = 32500 TO 32523
120  READ x
130  POKE i,x
```



```
140  NEXT i
150  DATA 6,16,221,33,0,126,221,86,16,221
160  DATA 94,32,221,78,0,205,0,125,221,35
170  DATA 5,32,239,201
200  REM TABLES
210  FOR i = 32256 TO 32303
220  READ x
230  POKE i,x
240  NEXT i
250  REM CHARACTERS
260  DATA 66,111,98,32,83,112,97,114
270  DATA 107,101,115,32,49,57,56,51
280  REM ROWS
290  DATA 10,10,10,10,10,10,10,10
300  DATA 10,10,10,10,12,12,12,12
310  REM COLUMNS
320  DATA 16,17,18,19,20,21,22,23
330  DATA 24,25,26,27,19,20,21,22
500  REM RUN PROGRAM
510  RANDOMIZE USR 32500
```

Note that the data statements in lines 250 to 330 are in sets of eight.

It should not be too difficult to see how this process could be extended to fill large parts of the screen. The main difficulty over this is that the displacement in the LD D,(IX+disp) instructions can only go up 255. To overcome this, we can use the HL register pair as a pointer instead. This can be increased one page at a time, by incrementing the H register. In this way up to 256 characters could be displayed on the screen at once. Beyond this, it becomes easier to ignore the rows and columns and just fill all the places on the screen, even if many of these will contain blanks. The technique then, is to point to each character in turn, and to place it in the next screen position. At the end of one line, the row counter is incremented to point to the next.

By way of example, we return to the problem we originally started with, how to fill the whole screen with *-characters in machine code. For this we can use the print subroutine developed before (32000 to 32009) and keep the D and E registers as pointers to the rows and columns. The character to be printed can be obtained from a table, as indicated above. This table runs from 7A00H to 7CFFH, which means that the top of memory must now be set lower than 32000. CLEAR 31000 will give plenty of room. For this particular application, we always send the same character, so the X-INDEX is not strictly needed as we shall see later. We are, however, trying to show how a complete picture could be transferred. In such a case, the character at each position could be different. To fill this table with the *-character is quite easy.

```
400   FOR i = 31232 TO 31999
410   POKE i,42
420   NEXT i
```

Screenfill

```
32500  DD 21 007A       LD IX,7A00H     ;Initialize pointer
32504  16 00            LD D,0          ;Initialize row counter
32506  1E 00     nxrow  LD E,0          ;Initialize column counter
32508  DD 4E 00  nxcol  LD C,(IX+00H)   ;Get character
32511  CD 00 7D         CALL 32000      ;Print character
32514  DD 23            INC IX          ;Point to next character
32516  1C               INC E           ;Next column
32517  3E 20            LD A,32         ;End column?
32519  BB               CP E
32520  20 F2            JR NZ,nxcol     ;No, do next column
32522  14               INC D           ;Yes, move to next row
32523  3E 16            LD A,22         ;Last row?
32525  BA               CP D
32526  20 EA            JR NZ,nxrow     ;No, do next row
32528  C9               RET             ;Yes, return to BASIC
DATA 221,33,0,122,22,0,30,0,221,78
DATA 0,205,0,125,221,35,28,62,32,187
DATA 32,242,20,62,22,186,32,234,201
```

To make this program a little more sensible for filling the screen with the *same* character, line 32508 can be replaced by

```
0E 2A            LD C,42         ;get *-character
```

This line is placed at the beginning, since we do not want to waste time by executing it over and over again. Lines 32500 and 32514 can be omitted altogether, since we are no longer using the character table. Unlike BASIC, we cannot just omit these lines, because they are not just arbitrary line numbers; they are the locations of the codes and none may be omitted. Finally, note how the jump displacements have to be altered because some code has been omitted. These changes give a revised listing as follows:

Starfill

```
32500  0E 2A            LD C,42         ;Get *-character
32502  16 00            LD D,0          ;Initialize row counter
32504  1E 00     nxrow  LD E,0          ;Initialize column counter
32506  CD 00 7D  nxcol  CALL 32000      ;Print character
32509  1C               INC E           ;Next column
32510  3E 20            LD A,32         ;End column?
32512  BB               CP E
32513  20 F7            JR NZ,nxcol     ;No, do next column
32515  14               INC D           ;Yes, move to next row
```



```
32516  3E 16            LD A,22         ;Last row?
32518  BA               CP D
32519  20 EF            JR NZ,nxrow     ;No, do next row
32521  C9               RET             ;Yes, return to BASIC
DATA 14,42,22,0,30,0,205,0,125,28
DATA 62,32,187,32,247,20,62,22,186,32
DATA 239,201
```

Attribute change

A similar technique can be used to alter the attributes of each screen position. This is slightly easier, since we can change them directly, as we did at the beginning of this chapter. We cannot use exactly that routine, since the B register only goes up to 255 and we shall want to affect all 768 screen positions. We could use the X-INDEX as above, but we still have to count 768 bytes. But an astute observer would note that 768 is exactly three pages, so it is easier to use the HL register pair to point to each position in turn. The attributes start at 5800H and go up to 5AFFH, so that when the HL register pair reaches 5B00H, we have finished. Although there is no sixteen-bit comparison instruction, we only need to test the H register, which will contain 5BH at the finish.

```
32521  3E 5B            LD A,5BH        ;Set end position
32523  21 00 58         LD HL,5800H     ;Initialize pointer
32526  16 3F            LD D,63         ;White paper, white ink
32528  72        nxatt  LD (HL),D       ;Paint attribute
32529  23               INC HL          ;Next column
32520  BC               CP H
32531  20 FB            JR NZ,nxatt     ;No, do next position
32533  C9               RET             ;Yes, return to BASIC
DATA 62,91,33,0,88,22,63,114,35,188
DATA 32,251,201
```

This routine is designed to follow immediately after the print * routine. When run, the *-characters should be printed on the screen and almost immediately erased by this attribute fixing routine. We have now developed a machine-code routine for filling the screen with invisible *-characters, a process that originally took four lines in BASIC. But we only need to do this once, so the machine-code routine will now be discarded in favour of that BASIC program, there is no point in using machine code when we don't have to!

Large letters

An application of the PRINT AT routine is to create large characters that can be seen across the classroom. The technique is fundamentally quite simple. When creating user defined characters, we saw that each consists of eight rows of eight dots. If, instead of printing single dots, we print quarter-square blocks, each consisting of sixteen individual dots, then each character is sixteen times bigger and can be read at four times the normal distance. Each enlarged character will

occupy sixteen PRINT AT positions, so we need a table of sixteen bytes to hold the required quarter-square characters. Printing them is handled by the routine described above; the next problem is to choose which quarter-square characters are to be used.

The codes for each of the standard characters are stored in ROM in sets of eight beginning at location 6100H. Only characters with ASCII values from 32 to 127 are stored here, and a pointer (called CHARS) is kept in the system variables. (This can be altered to point to any other location: you could, for example, define a set of gothic or italic characters and use them instead of those provided.) To point to a particular set of codes, we load the HL register pair with the value 6000H and the DE pair with the ASCII value of the desired character. The DE pair is then multiplied by eight (three left shifts) and added to HL, which thus points at the first row of the desired set of codes. For example, if CHR$ 65 is chosen, 8 × 65 gives 520 or 0208H. The HL pair will thus point at 6208H, which contains the codes for A as follows:

Location	Contents
6208H	0
6209H	60
620AH	66
620BH	66
620CH	126
620DH	66
620EH	66
620EH	0

The first character that can be accessed in this way is CHR$ 32 and its codes reside in locations 6100H to 6107H. This is why CHARS points at a location 256 below this beginning of the character table. If user-defined graphics are used, we send fictitious 'ASCII' values starting from 0 and point to the actual beginning of the character table instead.

The BIGLETT routine scans each row of these defining codes in turn and where a bit is on, it sets up the appropriate quarter square in the sixteen-byte table. This is done by shifting the code for each row into the CARRY bit and testing it with JR NC.

The Spectrum user is fortunate in that the set of quarter-square characters is logically given the codes 128 (blank or space) to 143 (black). Subtracting 128 produces the numbers from 0 to 15. This means that the quarters have values 2, 1, 8 and 4 and all combinations can be made by adding these numbers together. BIGLETT keeps a note of which row and which column a particular dot is in and converts this into a single number from 0 to 15. It then adds the required quarter square to any that already exist and stores them in the table. From here they are transfered directly to the screen.

The ASCII value of the character to be printed is poked from BASIC into location 32272 (character) and the required PRINT AT position is poked into locations 32273 (column) and 32274 (row).

BIGLETT

```
32000  06 10          LD B,16          ;Clear 16 bytes for table
32002  21 00 7E        LD HL,7E00H      ;Point to first byte
32005  36 00          LD (HL),0
32007  23       nxbyt INC HL           ;Next byte
32008  10 FB          DJNZ,nxbyt       ;Repeat 16 times
32010  2A 36 5C        LD HL,(CHARS)    ;Initialize pointer
32013  3A 10 7E        LD A,(char)      ;Get character code
32016  5F             LD E,A
32017  16 00          LD D,0           ;Set up DE with code
32019  CB 23          SLA E            ;Mult by 2
32021  CB 23          SLA E            ;Mult by 4
32023  CB 12          RL D             ;MSB into high byte
32025  CB 23          SLA E            ;Mult by 4
32027  CB 12          RL D             ;MSB into high byte
32029  19             ADD HL,DE        ;HL points at codes
32030  06 00          LD B,0           ;Point to first row
32032  0E 00       nxrow LD C,0        ;Point to first column
32034  5E             LD E,(HL)        ;Get a row of bit codes
32035  CB 3B       nxcol ASL E         ;Shift bit into CARRY
32037  30 1F          JR NC,nosq       ;Is there a bit?
32039  78             LD A,B           ;Odd or even row?
32040  CB 3F          SRL A            ;Shift LSB into CARRY
32042  30 0D          JR NC,yeven      ;Check CARRY bit
32044  79             LD A,C           ;Odd or even column?
32045  CB 3F          SRL A            ;Shift LSB into CARRY
32047  30 04          JR NC,xeven1     ;Check CARRY bit
32049  16 04          LD D,4           ;Odd row, odd column
32051  18 13          JR cont          ;Continue
32053  16 08       xeven1 LD D,8       ;Odd row, even column
32055  18 0F          JR cont          ;Continue
32057  79             LD A,C           ;Odd or even column?
32058  CB 3F          SRL A            ;Shift LSB into CARRY
32060  30 04          JR NC,xeven2     ;Check CARRY bit
32062  16 01          LD D,1           ;Even row, odd column
32064  18 06          JR cont          ;Continue
32066  16 02       xeven2 LD D,2       ;Even row, even column
32068  18 02          JR cont          ;Continue
32070  16 00       nosq LD D,0         ;Blank
32072  E5          cont PUSH HL        ;Keep note of code
                                        address
32073  78             LD A,B           ;Convert row to table
                                        position
32074  E6 0E          AND 14           ;(row DIV 2)*2
```

```
32076  CB 27          SLA A          ;(row DIV 2)*4
32078  6F             LD L,A         ;Keep low byte of pointer
32079  79             LD A,C         ;Convert column to table
                                     position
32080  CB 3F          SRL A          ;Column DIV 2
32082  85             ADD A,L
32083  6F             LD L,A         ;Keep low byte of pointer
32084  26 7E          LD H,126       ;High byte of table
32086  7E             LD A,(HL)      ;Get current contents
32087  82             ADD A,D        ;Add new quarter square
                                     code
32088  F6 80          OR 128         ;Set MSB
32090  77             LD (HL),A      ;Save new contents
32091  E1             POP HL         ;Restore code pointer
32092  0C             INC C          ;Next column
32093  79             LD A,C         ;Eight columns done?
32094  FE 08          CP 8
32096  20 C1          JR NC,nxcol    ;No, do next column
32098  23             INC HL         ;Point to next row of code
32099  04             INC B          ;Next row
32100  78             LD A,B         ;Eight rows done?
32101  FE 08          CP 8
32103  20 B7          JR NC,nxrow    ;No, do next row
32105  ED 5B 11 7E    LD DE,(rowcol) ;Point to PRINT AT
                                     position
32109  21 00 7E       LD HL,32256    ;Point to table of bytes
32112  06 04     nxtr LD B,4         ;Four bytes per row
32114  3E 16     nxtc LD A,16H       ;Send PRINT AT
32116  D7             RST 10H
32117  7A             LD A,D         ;Send row
32118  D7             RST 10H
32119  7B             LD A,E         ;Send column
32120  D7             RST 10H
32121  7E             LD A,(HL)      ;Send quarter square
                                     character
32122  D7             RST 10H
32123  23             INC HL         ;Next byte
32124  1C             INC E          ;Next PRINT AT column
32125  10 F3          JR NZ,nxtc     ;Do next column
32127  14             INC D          ;Next PRINT AT row
32128  1D             DEC E          ;Restore beginning of row
32129  1D             DEC E
32130  1D             DEC E
32131  1D             DEC E
```



```
32132  7D             LD A,L         ;All bytes done?
32133  FE 16          CP 16
32135  20 E7          JR NZ,nxtr     ;Do next row
32137  C9             RTS
```

A BASIC program to load this routine is given below. This program prints each of the ninety-six characters on the screen in order for inspection. To appreciate them, stand at least four metres from the screen.

```
   1  CLEAR 32000
   2  GOSUB 1000
 100  FOR n = 32 TO 127
 110  LET c = INT (n / 32)
 120  LET r = n - c * 32
 130  LET s = INT (r / 8)
 140  LET t = r - s * 8
 150  POKE row,4 * s
 160  POKE column,4 * t
 170  POKE character,n
 180  PRINT AT 0,0
 190  LET l = USR 32000
 200  PAUSE 25
 210  NEXT n
 220  STOP
1000  FOR i = 32000 TO 32137
1010  READ x
1020  POKE i,x
1030  NEXT i
1100  DATA 6,16,33,0,126,54,0,35,16,251
1110  DATA 42,54,92,58,16,126,95,22,0,203
1120  DATA 35,203,35,203,18,203,35,203,18,25
1130  DATA 6,0,14,0,94,203,35,48,31,120
1140  DATA 203,63,48,13,121,203,63,48,4,22
1150  DATA 4,24,19,22,8,24,15,121,203,63
1160  DATA 48,4,22,1,24,6,22,2,24,2
1170  DATA 22,0,229,120,230,14,203,39,111,121
1180  DATA 203,63,133,111,38,126,126,130,246,128
1190  DATA 119,225,12,121,254,8,32,193,35,4
1200  DATA 120,254,8,32,183,237,91,17,126,33
1210  DATA 0,126,6,4,62,22,215,122,215,123
1220  DATA 215,126,215,35,28,16,243,20,29,29
1230  DATA 29,29,125,254,16,32,231,201
1300  LET character = 32272
1310  LET column = 32273
1320  LET row = 32274
1330  RETURN
```

The routine can be changed to print large versions of user-defined graphics by altering the address of the character codes from the system variable CHARS to UDG, so that line 1110 becomes:

```
1110    DATA 42,123,92,58,16,126,95,22,0,203
```

These characters have the codes 0 to 20. If undefined by the user, each is initially set up to its corresponding upper case letter. It should be clear that this technique makes the creation of chunky pictures much easier than by drawing each with successive PRINT AT statements. The picture is first drawn with user-defined graphics and then enlarged in the manner shown above. We shall return to this routine again when considering animated graphics.

Particle motion

One of the earliest applications of microcomputers in science was the use of fast machine-code animations to simulate wave motion and the movement of molecules etc. We have already seen how the screen can be filled with the *-character. Let us now look at how the motion of this character may be achieved in machine-code graphics. The obvious way of achieving horizontal motion is to paint the character successively one screen position further to the right each time as we did in the BASIC program in Chapter 2. Note that the *-characters must already be in position, so that we merely expose them by changing the ATTRIBUTE file.

The following program paints the *-character into the thirty-two contiguous positions at the top of the screen. It is similar to the program discussed before, except that this time a new instruction has been introduced. This is **DJNZ (decrement and jump if not zero)**. It combines the two separate instructions used previously DEC B and JR NZ. DJNZ decrements the B register and jumps to the required place if the B register has not reached zero. This instruction is the reason for using the B register as a counter on so many occasions.

Stars 2

```
max     = 32
screen  = 5800H
char    = 56
32000   06 20          LD B,max          ;32 *-characters
32002   21 00 58       LD HL,screen      ;First attribute position
32005   3E 38          LD A,char         ;Black ink, white paper
32007   77       next   LD (HL),A         ;Place * in screen position
32008   23              INC HL            ;Move to next position
32009   10 FC           DJNZ next         ;All positions done?
32011   C9              RET               ;Yes, so finish
DATA 6,32,33,0,88,62,56,119,35,16
DATA 252,201
```


When you run this program, you will not get motion but merely a set of stars. Why, though, do we not see the stars appear in succession, why do they all come at once? The reason is not hard to find, but it requires a little more knowledge about the microprocessor.

Because so many things are happening in the microcomputer, everything is under the control of the system clock, which beats away regularly at about 300 nanosecond intervals (a frequency of 3.5 MHz). Each pulse is called a T-state. Single-byte instructions require four T-states usually, so they take about one microsecond to be executed. If the operation requires an external location, then the execution time is increased. Some instructions need one extra byte for the operand while others need two. An example of a two-byte instruction is LD A,7CH and an example of a three-byte instruction is LD HL,7C00H. Two-byte instructions are generally executed in two microseconds, while three-byte instructions take about three microseconds (for the extra byte to be fetched and decoded). Thus it is not difficult to predict how long a particular program will take. The execution time for each instruction can be looked up in a table and the total time calculated. For our particular program, the number of T-states is as follows:

```
LD B,max        ;7
LD HL,screen    ;10
LD A,char       ;7
loop begins
LD (HL),A       ;7
INC HL          ;6
DJNZ next       ;8 if unsuccessful
                ;13 if successful
```

The first three instructions are only executed once, taking twenty-four T-states. The next three instructions normally take twenty-six T-states, except for the last, when the test condition fails. The number of T-states is then twenty-one. The total is thus 24 + 31*26 + 21, which is 851 T-states or about 250 microseconds. No wonder the *-characters appeared instantaneously! The solution is obvious, we must find a means of making the microprocessor waste time.

There is a single-byte instruction in the Z80 set, which performs just this function – NOP (no operation). It takes four T-states to execute and causes absolutely nothing else to happen. Unfortunately, we are looking for a much longer delay than this and must go elsewhere. The most efficient time-wasting technique is to ask the microprocessor to count up to some number each time before proceeding with the rest of its instructions. In BASIC this is known as a **delay loop**.

```
100   FOR T = 1 TO 1000:NEXT T
```

In machine code the simplest delay loop uses one of the internal register pairs and since we are using the HL-pair as a pointer, we shall have to use the DE-pair

instead. This causes its own problems, since none of the sixteen-bit INC or DEC instructions has any effect on the flags. How shall we know when we have finished counting? The solution of this problem is to split the DE-pair into separate bytes and to check them independently. If we count down, rather than count up, we can quite easily check when both the D and E registers are at zero as follows:

```
        LD DE,count   ;Initialize DE register pair
loop    DEC DE        ;6 T-states
        LD A,D        ;4 T-states
        OR E          ;4 T-states
        JR NZ,loop    ;12 T-states if successful
                      ;7 T-states otherwise
        etc.
```

The DE register pair is initialized to some count value, say 10 000. The DE pair is then successively decremented. It is tested to see when it reaches zero, by ORing its two halves together. If it is not equal to zero, then the program jumps back to the second instruction, labelled **loop**. When the DE pair is decremented on the 10 000th time, it becomes 0000 0000 0000 0000. ORing the two halves gives zero, so the looping is then terminated. The execution times for each instruction are shown in the comment column, and it can be seen that this loop takes 26 T-states per loop or about 7.5 microseconds. The total delay is thus 7.5 × count, which, in our case, gives 75 ms.

The time to place all thirty-two *s on the screen is thus increased to more than two seconds, which is appreciable. The number written into 'count' before the routine is called can be varied from 1 to 65 535. The total time needed to place all the *s on the screen can thus be varied from about 0.3 ms to several seconds. Longer delays than this are unnecessary, since the program would then be slow enough for BASIC.

How do we insert this delay routine into our machine-code program? It could be fitted in after the * has been sent to its screen position and before the pointer is incremented to the next position, but there is a strong reason for not doing that. It is possible that the routine for producing a delay will need to be used several times more, and every time we use it, it will have to be written out again. So a better technique is to place the delay loop in a separate subroutine. The memory locations in the delay program have been chosen to run almost from the end of the previous routine upwards (so that the BASIC loader becomes easier to write).

The *-painting program, that we started with, must now be altered to take account of this delay subroutine. In addition, each star must be erased from the screen after its has been placed there, to produce the illusion of motion. We do this by painting the attribute position with white ink soon after it was painted with black ink. I say 'soon after' and not 'immediately after' because we want to leave the *-character long enough to be able to see it. The best place is therefore after the delay subroutine as follows. The value for the timing variable (count) is written

directly into two successive locations (32510 or 7EFEH and 32511 or 7EFFH) from BASIC and this sets the speed at which the star moves across the screen.

Star 3

```
max       = 32
screen    =5800H
char      = 56
delete    = 63
count     =32510 and 23511

32000  06 20          LD B,max       ;32 *-characters
32002  21 00 58       LD HL,screen   ;First attribute position
32005  3E 38    next  LD A,char      ;Black ink, white paper
32007  77             LD (HL),A      ;Place * in screen position
32008  CD 32 7D       CALL delay     ;Wait
32011  3E 3F          LD A,delete    ;White ink, white paper
32013  77             LD (HL),A      ;Delete *
32014  23             INC HL         ;Move to next position
32015  10 F4          DJNZ next      ;All positions done?
32017  C9             RET            ;Yes, so finish
32018  00             NOP            ;Filler
32019  00             NOP
32020  00             NOP
32021  00             NOP
32022  00             NOP
32023  00             NOP
32024  00             NOP
32025  00             NOP
32026  00             NOP
32027  00             NOP
32028  00             NOP
32029  00             NOP
32030  00             NOP
32031  00             NOP
32032  00             NOP
32033  00             NOP
32034  00             NOP
32035  00             NOP
32036  00             NOP
32037  00             NOP
32038  00             NOP
32039  00             NOP
32040  00             NOP
32041  00             NOP
32042  00             NOP
```

```
32043  00                NOP
32044  00                NOP
32045  00                NOP
32046  00                NOP
32047  00                NOP
32048  00                NOP
32049  00                NOP
32050  ED 5B FE 7E delay LD DE,(count) ;Get delay time
32054  1B          loop  DEC DE
32055  7A                LD A,D        ;Check for finish
32056  B3                OR E
32057  20 FB             JR NZ, loop
32059  C9                RET           ;Finish on timeout
```

To illustrate how the various parts fit together, the complete BASIC program to load and run this routine is as follows:

```
 10  CLEAR 32000
 20  REM Get delay time
 30  INPUT "Enter delay (range 1 to 65000) ";delaytime
 40  LET count = 32510
 50  LET highbyte = INT(delaytime / 256)
 60  LET lowbyte = INT(delaytime - 256 * highbyte + .1)
 70  POKE count, lowbyte:POKE (count + 1),highbyte
 80  INK 7:PAPER 7
 90  FOR Y=0 TO 21:PRINT AT 0,Y;"********************************":NEXT Y
100  REM load machine code
110  FOR i=32000 TO 32059
120  READ x
130  POKE i,x
140  NEXT i
150  DATA 6,32,33,0,88,62,56,119,205,50
160  DATA 125,62,63,119,35,16,244,201,0,0
170  REM filler
180  DATA 0,0,0,0,0,0,0,0,0,0
190  DATA 0,0,0,0,0,0,0,0,0,0
200  DATA 0,0,0,0,0,0,0,0,0,0
210  REM delay subroutine
220  DATA 237,91,254,125,27,122,179,32,251,201
500  REM run machine code routine
510  LET k=USR 32000
```

So far we have only considered what happens when the pointer to the next screen position (the HL-pair) is increased. You can probably guess that if we were to decrease the pointer instead, then the star would move backwards across the

screen from right to left. The instruction to decrement the HL-pair is **DEC HL**, and when executed, the HL-pair points to the previous screen position, rather than the next one.

What we shall do, is wait until the star reaches the thirty-first screen position and then, instead of finishing as at present, we shall decrement the HL-pair successively until it reaches the beginning again. We can detect when it gets there in two ways, either by resetting the B register to thirty-two and decrementing it until it reaches zero, or by waiting until the HL-pair reaches its starting value of 5800H, i.e. when the L register reaches zero. We opt for the first method and the extra instructions to do this are listed below, starting from the location where they are different from the previous listing.

```
32017  06 20             LD B,32          ;32 positions
32019  21 1F 58          LD HL,581FH      ;Start at end of line
32022  3E 38       nxrev LD A,char        ;Black ink, white paper
32024  77                LD (HL),A        ;Place * in screen position
32025  CD 32 7D          CALL delay       ;Wait
32028  3E 3F             LD A,delete      ;White ink, white paper
32030  77                LD (HL),A        ;Delete *
32031  2B                DEC HL           ;Move to previous position
32032  10 F4             DJNZ nxrev       ;All positions done?
32034  C9                RET              ;Yes, so finish
```

```
160  DATA ..... 6,32,33
180  DATA 31,88,62,56,119,205,50,125,62,63
190  DATA 119,43,16,244,201,0,0,0,0,0
```

It is left as an exercise for you to work out exactly how to insert these extra codes into the program (although strong clues are given). They replace some of the 0s already sitting in the data statements.

Instead of a return to BASIC in line 32034, a jump to the start of the program would keep the star in continuous motion. But how then would we ever leave this program? It would continue for ever until the power is removed and this is not an elegant way to finish. A better way is to look at the keyboard to see if any key is being pressed and, if so, to return to BASIC with RET. If this keyboard routine is placed at the end of the main program it will only be effective when the star reaches the left side of the screen. A better way would be to place the keyboard routine inside the delay subroutine so that the keyboard will be checked more often. Unfortunately this means that we cannot then immediately return to BASIC with RET, because we are still in a subroutine. We must first pull a double byte off the STACK to get at the BASIC return address.

Keyboard-sensing routine

The instructions used in the keyboard-sensing routine have not yet been introduced. Fundamentally, they are the same as PEEK and POKE, but they operate on I/O devices rather than on internal memory. The actual instructions

are IN and OUT and they behave in the same way as the BASIC IN and OUT discussed in Chapter 4. To see if a key is being pressed, the keyboard is scanned with the instructions

```
LD A,n
IN A,(254)
```

where n depends upon which part of the keyboard is being scanned. When the IN A,(254) instruction is executed, the microprocessor places the current contents of the ACCUMULATOR onto the high byte of the ADDRESS register. It then places 254 into the low byte of the ADDRESS register and reads the data on the data bus. The keyboard is in eight sections and each section responds when a particular bit of the high byte address goes LOW.

LOW bit	High byte	Low byte	Decimal address	Keys
Bit 0	254	254	65278	CAPS to V
Bit 1	253	254	65022	A to G
Bit 2	251	254	64510	Q to T
Bit 3	247	254	63486	1 to 5
Bit 4	239	254	61438	0 to 6
Bit 5	223	254	57342	P to 7
Bit 6	191	254	49150	ENTER to H
Bit 7	127	254	32766	SPACE to B

The keyboard responds by putting a byte onto the data bus corresponding to the key being pressed. If no key is being pressed in the selected section, then all bits are HIGH and the value returned is 255. If the end key of the chosen section is being pressed, then bit 0 goes LOW, giving a value of 254, etc. This way it is possible to scan the whole keyboard to see which key, if any, is being pressed. In our case, we are only using this facility to tell a machine-code routine to stop, so it is irrelevant which key we choose. An obvious one is SPACE, since that is associated with BREAK. The instruction to detect this key is, therefore:

```
LD A,127      ;Select correct section
IN A,(254)    ;Scan this section
OR 224        ;Set the top three bits
CP 254        ;Is it the SPACE key?
JR Z,finish   ;Yes, so finish
etc.          ;No so continue
```

The reason for setting the top three bits is because these can have any value, depending upon the colour chosen for BORDER. A black border, for example, sets all three bits LOW, so the test for the SPACE key would then have to be CP A,30. It is easier just to set the three bits HIGH every time, so that this variation can be ignored.

This routine is to be placed at the end of the delay subroutine as follows:


```
32059  3E  7F   LD A,127     ;Select correct section
32061  DB  FE   IN A,(254)   ;Scan this section
32063  F6  E0   OR 224       ;Set the top three bits
32065  FE  FE   CP 254       ;Is it the SPACE key?
32067  28  01   JR Z,finish  ;Yes, so finish
32069  C9       RET          ;No so continue bouncing
32070  C1       POP BC       ;Pull address off STACK
32071  C9       RET          ;Return to BASIC
```

The extra decimal codes are as follows:

```
DATA 62,127,219,254,246,224,254,254,40,1,201,193,201
```

With this keyboard-sensing routine the star can now bounce back and forth until you stop it by pressing the SPACE key. At some speeds the motion of the particle is rather jerky because the screen refresh rate is out of synchronization with the display of the particle. There ought to be a way of preventing this by maintaining control over when the screen is refreshed, but I have yet to discover how. The full program is listed below for those still not quite sure how to convert an assembly listing into a BASIC loader.

Stars 4

```
10  CLEAR 32000
20  REM Get delay time
30  INPUT "Enter delay (range 1 to 65000) ";delaytime
40  LET count = 32510
50  LET highbyte = INT(delaytime/256)
60  LET lowbyte = INT(delaytime - 256*highbyte + .1)
70  POKE count, lowbyte:POKE (count+1),highbyte
80  INK 7:Paper 7
90  FOR Y=0 TO 21:PRINT AT Y,0;"**********************************":NEXT Y
100 REM load machine code
110 FOR i=32000 TO 32071
120 READ x
130 POKE i,x
140 NEXT i
150 DATA 6,32,33,0,88,62,56,119,205,50
160 DATA 125,62,63,119,35,16,244,6,32,33
180 DATA 31,88,62,56,119,205,50,125,62,63
190 DATA 119,43,16,244,201,0,0,0,0,0
200 DATA 0,0,0,0,0,0,0,0,0,0
210 REM delay subroutine
220 DATA 237,91,254,125,27,122,179,32,251,62
230 DATA 127,219,254,246,224,254,254,40,1,201
240 DATA 193,201
```

```
500  REM run machine code routine
510  LET k=USR 32000
```

Adding .1 to 256*highbyte in line 60 is to avoid rounding errors.

Molecular motion

The machine-code routine for MOLECULAR MOTION (34) is similar to its BASIC equivalent in ONEMOL (33). The top and bottom walls are coloured black (ATTRIBUTE = 0) and the side walls blue (ATTRIBUTE = 9), so it is easy for a molecule to determine which sort of wall it is colliding with. This collision is detected in this way. For each particle its current position is stored in one table (poslo and poshi) and its current direction in another (dirlo and dirhi). The direction is first added to the old position and the new position is checked to see if it is currently occupied. If it is occupied by another molecule (ATTRIBUTE = 56), the latter is ignored. (Two colliding particles of the same mass simply exchange velocities anyway, so, if the molecules cannot be distinguished, the effect is the same as if the molecules had ignored each other in the first place.) If the new position is occupied by a wall, the direction is altered to represent a different direction of motion and dirlo and dirhi are updated accordingly. If the new position is empty or only contains another molecule (ATTRIBUTE = 63 or 56) the old position on the screen is deleted (painted white) and the ink is changed to reveal the molecule in its new position. The values of poslo and poshi are then updated accordingly. This process is repeated for each individual molecule. The C register is used to point to the current position in each table and the required table is selected by altering the B register. Any particular table value is then obtained with LD A,(BC).

To simulate different temperatures a delay is introduced in BASIC between calling the molecule routine each time. The number of molecules is determined by the contents of the location called number. Normally this is a variable, although in the program as listed under MOLMOT (34), the number of molecules has been fixed at thirty-five. The more sophisticated programs, of which MOLMOT is a sampler, provide the ability to choose the number of molecules.

MOLECULAR MOTION

Note: codes are given as decimals in this listing.

```
number   =30592
poslo    = 7A00H to 7AFF
poshi    = 7B00H to 7BFF
dirlo    = 7C00H to 7CFF
dirhi    = 7D00H to 7DFF
30464 58,128,126        LD A,(number)  ;Get number of molecules
30467 79                LD C,A         ;Pointer to current molecule
                                       ;Get current position in HL
30468 6,122        nxmo LD B,poslo     ;High byte pointer to position
30470 10                LD A,(BC)      ;Get low byte of position
30471 111               LD L,A         ;Save in L
30472 4                 INC B          ;Point to poshi table
30473 10                LD A,(BC)      ;Get high byte of position
30474 103               LD H,A         ;Save in H
                                       ;Get current direction in DE
30475 4                 INC B          ;Point to dirlo table
30476 10                LD A,(BC)      ;Get low byte of direction
30477 95                LD E,A         ;Save in E
30478 4                 INC B          ;Point to dirhi table
30479 10                LD A,(BC)      ;Get high byte of direction
30480 87                LD D,A         ;Save in D
30481 25                ADD HL,DE      ;Find new position
30482 126               LD A,(HL)      ;Get current contents
30483 254,63            CP 63          ;Is it empty?
30485 202,13,120        JP Z,empty     ;Yes
30488 254,0             CP 0           ;Top or bottom wall?
30490 202,128,119       JP Z,horiz     ;Yes
30493 254,9             CP 9           ;Side wall
30495 194,13,120        JP NZ,empty    ;No, so ignore current contents
                                       ;Side wall – change direction
30498 123               LD A,E         ;Get current direction
30499 254,223           CP 223         ;North-west?
30501 40,22             JR Z,nw        ;Yes
30503 254,255           CP 255         ;West?
30505 40,24             JR Z,w         ;Yes
30507 254,31            CP 31          ;South-west?
30509 40,26             JR Z,sw        ;Yes
30511 254,225           CP 225         ;North-east?
30513 40,28             JR Z,ne        ;Yes
30515 254,1             CP 1           ;East
30517 40,30             JR Z,e         ;Yes
30519 17,31,0      se   LD DE,31       ;Go south-west
30522 195,0,120         JP dirend      ;Change of direction finished
30525 17,225,255   nw   LD DE,-31      ;Go north-east
30528 195,0,120         JP dirend
30531 17,1,0       w    LD DE,1        ;Go east
30534 195,0,120         JP dirend
30537 17,33,0      sw   LD DE,33       ;Go south-east
30540 195,0,120         JP dirend
30543 17,223,255   ne   LD DE,-33      ;Go north-west
30546 195,0,120         JP dirend
30549 17,255,255   e    LD DE,-1       ;Go west
30552 195,0,120         JP dirend
```

```
30555 0                 NOP
.....................................................................
                                        ;Horiz. wall - change direction
30592 123          horiz LD A,E         ;Get current direction
30593 254,31            CP 31           ;South-west?
30595 40,22             JR Z,hsw        ;Yes
30597 254,32            CP 32           ;South?
30599 40,24             JR Z,hs         ;Yes
30601 254,33            CP 33           ;South-east?
30603 40,26             JR Z,hse        ;Yes
30605 254,223           CP 223          ;North-west?
30607 40,28             JR Z,hnw        ;Yes
30609 254,224           CP 224          ;North?
30611 40,30             JR Z,hn         ;Yes
30613 17,33,0      hne  LD DE,33        ;Go south-east
30616 195,0,120         JP direnD       ;Change of direction finished
30619 17,223,255   hsw  LD DE,-33       ;Go north-west
30622 195,0,120         JP direnD
30625 17,224,255   hs   LD DE,-32       ;Go north
30628 195,0,120         JP direnD
30631 17,225,255   hse  LD DE,-31       ;Go north-east
30634 195,0,120         JP direnD
30637 17,31,0      hnw  LD DE,31        ;Go south-west
30640 195,0,120         JP direnD
30643 17,32,0      hn   LD DE,32        ;Go south
30647 195,0,120         JP direnD
30650 0                 NOP
.....................................................................
30720 122          direnD LD A,D        ;Save new direction
30721 2                 LD (BC),A
30722 5                 DEC B
30723 123               LD (BC),A
30724 5                 DEC B
30725 195,32,120        JP cont         ;Continue with next molecule
30728 0                 NOP
30729 0                 NOP
30730 0                 NOP
30732 0                 NOP
30733 0                 NOP
30734 54,56        empty LD (HL),56     ;Insert in new position
30736 229               PUSH HL         ;Save new position
                                        ;Delete old position
30737 6,122             LD B,poslo      ;High byte pointer to position
30739 10                LD A,(BC)       ;Get low byte of position
```



```
30740 111               LD L,A          ;Save in L
30741 4                 INC B           ;Point to poshi table
30742 10                LD A,(BC)       ;Get high byte of position
30743 103               LD H,A          ;Save in H
30744 54,63             LD (HL),63      ;Delete from old position
30746 225               POP HL          ;Retrieve new position
                                        ;and insert into position table
30747 124               LD A,H          ;Save high byte
30748 2                 LD (BC),A
30749 5                 DEC B
30750 125               LD A,L          ;Save low byte
30751 2                 LD (BC),A
30752 13                DEC C           ;Point to next molecule
30753 194,4,119         JP NZ,nxmol
30756 201               RET             ;Back to BASIC if all done
```

Moving pictures

The above techniques can also be applied to moving whole pictures around the screen. The method is identical to the BASIC methods discussed in Chapter 2, using the machine-code PRINT AT routine described earlier. When executed, this program demonstrates BASIC and machine code side by side, so that their respective performances can be compared.

PICMOVE

```
32000 ED 5B 20 7E       LD DE,(postn)   ;Point to PRINT AT position
32004 2A 24 7E          LD HL,(ptr)     ;Point to table of bytes
32007 ED 4B 22 7E       LD BC,(colrow)  ;Number of rows and columns
32011 3E 16        next LD A,16H        ;Send PRINT AT
32013 D8                RST 10H
32014 7A                LD A,D          ;Send row
32015 D8                RST 10H
32016 7B                LD A,E          ;Send column
32017 D8                RST 10H
32018 7E                LD E,(HL)       ;Send code for character
32019 D8                RST 10H
32020 23                INC HL          ;Next code
32021 1C                INC E           ;Next PRINT AT column
32022 10 F3             JR NZ,next      ;Do next column
32024 14                INC D           ;Next PRINT AT row
32025 3A 23 7E          LD A,(cols)     ;Number of columns in picture
32028 1D           restor DEC E         ;Restore beginning of row
32029 04                INC B
32030 B8                CP B            ;All done?
32031 20 FB             JR NZ,restor    ;Do next
```

```
32033 0D          DEC C        ;Move to next row
32034 20 E7       JR NZ,next
32036 C9          RTS
```

A BASIC program to load this routine is given below. This program defines a racing car and then moves it across the screen, first via BASIC and then via the above machine-code routine. It is left for you to judge whether there is any significant difference. If you judge that there isn't, then increase the size of the picture to, say, an 8 × 8 matrix. The starting address, where the bytes for this picture are stored, has to be poked into the pointer (pointerlo and pointerhi) and the routine also needs to be told the number of columns and rows in the picture. Note that this is in the reverse order, so that the number of rows is loaded into the C register and the number of columns into the B register. Used in this way the machine-code routine is clearly superior. Likewise the machine-code routine would be more useful when moving several cars at once.

```
  1  CLEAR 32000
  2  GO SUB  1000
 10  REM Define the picture
 20  GO SUB 2000
 30  REM Move car via Basic
 40  FOR x=0 TO 27
 50  PRINT AT 5,x;" ABAC" (Note: these are defined
 60  PRINT AT 6,x;" FG D" graphics characters)
 70  PRINT AT 7,x;" IHIE"
 80  PAUSE 2
 90  NEXT x
100  REM Load character codes into table
110  FOR i=32256 TO 32270
120  READ code
130  POKE i,code
140  NEXT i
150  REM initialize pointers
160  POKE rowcount,3:POKE colcount,5
170  POKE pointerlo,0:POKE pointerhi,126
200  REM Move car via machine code
210  FOR x=0 TO 27
220  POKE row,10:POKE column,x
230  LET l=USR 32000
240  PAUSE 2
250  NEXT x
260  STOP
1000  FOR i=32000 TO 32036
1010  READ x
1020  POKE m,x
```



```
1030  NEXT i
1100  DATA 237,91,32,126,42,36,126,237,75,34
1110  DATA 126,62,22,215,122,215,123,215,126,215
1120  DATA 35,28,16,243,20,58,35,126,29,4
1130  DATA 184,32,251,13,32,231,201
1200  LET column=32288
1210  LET row=32289
1220  LET rowcount=32290
1230  LET colcount=32291
1240  LET pointerlo=32292
1250  LET pointerhi=32293
1260  RETURN
2000  REM Define graphics characters
2010  FOR i=0 TO 8
2020  FOR j=0 TO 7
2030  READ code
2040  POKE USR CHR$ (i+144)+j,code
2050  NEXT j
2060  NEXT i
2100  DATA 255,255,255,24,24,255,128,128
2110  DATA 0,0,0,0,0,255,0,255
2120  DATA 0,0,0,0,0,192,32,16
2130  DATA 8,4,2,1,1,2,4,8
2140  DATA 16,32,192,0,0,0,0,0
2150  DATA 129,129,129,129,129,129,129,129
2160  DATA 255,33,113,121,121,113,33,255
2174  DATA 255,0,255,0,0,0,0,0
2180  DATA 128,128,255,24,24,255,255,255
2200  RETURN
3000  REM codes for car positions
3010  DATA 32,144,145,144,146
3020  DATA 32,149,150,32,147
3030  DATA 32,152,151,152,148
```

In both cases a 5 × 3 matrix is moved horizontally, so that the picture contains following blanks to eliminate the rear of the picture as it moves. Movement in other directions is more difficult, since the whole picture has to be erased first. The bytes of the picture are pointed at through locations 32292 and 32293, so it is a simple matter to point these at a set of fifteen blanks (CHR$ 32) and to obliterate the whole picture by calling the routine again at the same place.

High resolution plotting

It is occasionally necessary to plot points on the screen in machine code. Examples are STANDING WAVES (31) and LONGITUDINAL WAVES (32).

where the screen picture has to be changed quite often to give the appearance of motion. Let us first look at the algorithm used to do this.

The high resolution screen runs from 4000H (top-left corner) to 57FFH. As we saw in Chapter 2, each bit of each byte controls one pixel of the screen. The memory map of the screen is not contiguous. Running the following program shows that each character position (eight by eight bits) is made from eight separate bytes, but these bytes are not together in the memory. Thirty-two contiguous bytes form a line and the next thirty-two bytes form another line eight pixels further down the screen. After 2048 bytes the whole of the top third of the screen is filled and it is then the turn of the middle third.

```
100   FOR i = 16384 TO 22527
110   POKE i,255
120   PAUSE 1
130   NEXT i
```

The algorithm to plot the point (x,y) directly is as follows:

```
byte number = base address + third + block + line + position
```

The base address is 4000H in all cases. The equation is interpreted as follows. The byte which contains the point x,y must be located within:

a) a particular third of the screen
The top third ($y < 64$) adds 000H to the base address.
The middle third ($63 < y < 128$) adds 800H to the base address.
The top third ($127 < y < 192$) adds 1000H to the base address.
Mathematically this is described by

```
third = 800H * INT(y / 64)
```

b) a particular block (where a single character can be printed
The top block adds 00H to the base address.
The second block down adds 20H to the base address.
The third block down adds 40H to the base address.
This continues, with 60H, 80H, A0H, C0H and E0H being added.
In general the block positions are multiples of 20H for each block and there are eight of them in each third of the screen. This situation is described by

```
block = 20H * INT (y/8)
```

c) a particular line within any one block
The top line in a block adds 000H to the base address.
The next line in the block adds 100H to the base address.
The next line in the block adds 200H to the base address.
In general, this can be described by:

```
line = 100H * (y MOD 8)
```


(y MOD 8 means the remainder left over after y has been divided by 8.)

d) a particular horizontal position along the line
The leftmost position adds 1 to the base address.
The next position adds 2 to the base address.
The next position adds 3 to the base address.
In general, since each position has eight pixels:

```
position = INT(x/8)
```

The bit within the byte is quite simple, it just follows the normal bit pattern. The leftmost bit is 01H, the next is 02H, the next 04H, the next 08H and so on up to 80H. This is back to front for us, but it is not difficult to turn the byte round the other way, as we shall see.

There are two things that can be done to a bit, once it has been located – to turn it on or to turn it off (in both cases leaving adjacent bits unaffected). Let us first look at the machine-code routine which locates the bit. This is a subroutine called **find**. The C register holds the x-coordinate and the D register has the y-coordinate. This is not quite right because the top-left of the screen is now the origin (0,0). This is a situation that Apple II users have long been used too. For most programs it is not a serious problem, the point (x,192-Y) has to be plotted instead. For wave motion programs this complication is ignored completely.

On return from the subroutine, the HL register pair holds the address of the screen byte containing the pixel required. The ACCUMULATOR contains a particular number, which is later used to determine the actual bit itself.

Find

```
28960 7A       LD A,D          ;Determine which line
28961 E6 07    AND 7
28963 67       LD H,A          ;Equals 100H*(y MOD 8)
28964 7A       LD A,D          ;Determine which block
28965 E6 38    AND 38H         ;Equal to 8*INT(y/8)
28967 07       RLCA            ;Multiply by 2
28968 07       RLCA            ;Multiply by 2
28969 6F       LD L,A          ;Equal to 20H*INT(y/8)
28970 7A       LD A,D          ;Determine which third
28971 E6 C0    AND C0H         ;Equals 64*INT(y/64)
28972 0F       RRCA            ;Divide by 2
28973 0F       RRCA            ;Divide by 2
28974 0F       RRCA            ;Divide by 2
28975 C6 40    ADD A,40H       ;Add base address
28977 84       ADD A,H         ;Add in line byte
28978 67       LD H,A          ;Save high byte address
28979 79       LD A,C          ;Determine which position
28980 E6 F8    AND 0F8H        ;Equals 8*INT(x/8)
28982 0F       RRCA            ;Divide by 2
28983 0F       RRCA            ;Divide by 2
```

```
28984  0F       RRCA        ;Divide by 2
28985  85       ADD A,L     ;Add block
28986  6F       LD L,A      ;Save low byte address
28987  79       LD A,C      ;Determine which bit
28988  2F       CPL         ;Turn byte round
28989  E6  07   AND 7       ;Mask to get lowest bits
28991  07       RLCA        ;Multiply by 2
28992  07       RLCA        ;Multiply by 2
28993  07       RLCA        ;Multiply by 2
28994  C9       RET         ;Return to calling program
```

The routines to erase and plot the bit once it has been located are now discussed. They use the RES and SET instructions respectively. They work on a most peculiar principle. On return from the find subroutine, the ACCUMULATOR contains a number as follows:

If bit 7 is chosen, the ACCUMULATOR contains 00H
If bit 6 is chosen, the ACCUMULATOR contains 08H
If bit 5 is chosen, the ACCUMULATOR contains 10H
If bit 4 is chosen, the ACCUMULATOR contains 18H
etc.

To this is added 86H for an erase or C6H for a plot. This produces the correct machine code to erase or set the required bit in the RES or SET instruction. It is therefore inserted into the actual program, which is reached immediately afterwards and executed. This is an example of a program that writes its own code – a **self-modifying program**. Such programs are very bad, because (as you can see) they are very difficult to describe and even more difficult to debug if they are found to be faulty. In defence, I find this the fastest way to do it, and speed takes precedence over good programming style on occasions.

Erase

```
28928  CD 20 71   CALL find       ;Get byte and bitcode
28931  C6 86      ADD A,86H       ;Add to get RES machine code
28933  32 09 71   LD (28937),A    ;Put code in program
28936  CB ??      RES ?,(HL)      ;Execute this code
28938  C9         RET             ;Return to calling program
```

The address 28937 is where the code to erase the chosen bit is stored. Hence, we cannot at this stage say what that code should be, the first part of the program determines it and changes the value accordingly. To show this, a "?" has been placed in the program, in the appropriate place in the DATA statement, so an arbitrary code like 0 must be inserted. This will be changed before it is executed, so the actual number does not matter.

To plot the dot requires a similar procedure, but with the SET instruction instead.

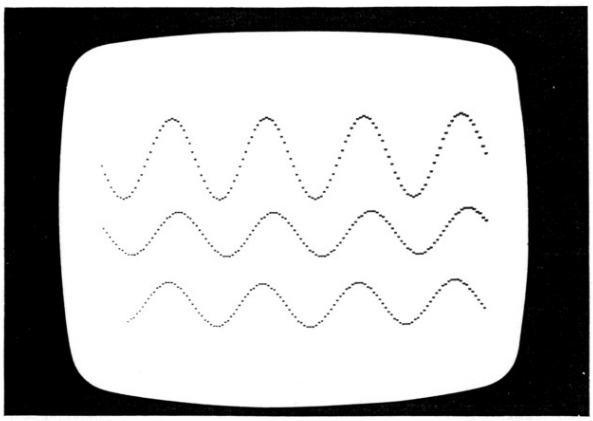


Plate 19 Standing waves

Plot

```
28944  CD 20 71   CALL find       ;Get byte and bitcode
28947  C6 C6      ADD A,C6H       ;Add to get SET machine
                                  code
28949  32 19 71   LD (28953),A    ;Put code in program
28952  CB ??      SET ?,(HL)      ;Execute this code
28954  C9         RET             ;Return to calling program
```

This routine is used extensively in the programs listed in the Appendix. STANDING WAVES (31) is a machine-code version of what is essentially a simple process. To create a wave on the screen we need to plot a sine wave, erase it and replot it one pixel to the left or right. In BASIC this takes far too long and wave motion is not apparent. Unfortunately, a machine-code routine to work out sines is beyond my capabilities. The solution is to use BASIC to work out the sines beforehand. These values are then stored in a table (sintbl), which is accessed in machine code using a pointer. If the X-INDEX points to the start of the table, then LD A,(IX+25) will retrieve the sine of 25 from the table. The table is loaded with the correct values by a program like this:

```
2000  REM Set up sine table
2010  REM contains 256 values
2020  FOR i=0 TO 255
2030  LET angle=i*PI/128
2040  LET val=SIN(angle)
2050  POKE (40000+1)=INT(20*val)
2060  NEXT i
```

This produces sines of amplitude 20. There has to be a different table for each different amplitude. Fortunately, the total number of amplitudes needed is usually sufficiently low that this can be done. STANDING WAVES uses a fixed amplitude just to show the principles (Plate 19).

The algorithm used to draw the waves is rather like that used to plot the molecules. For each x coordinate the present y-value of the wave is kept in a table (**opos**) and accessed via the BC register pair. This value is retrieved and passed to the erasing routine above. The current x position is then multiplied by a constant (called **wv1n**) and another constant (**time**) is subtracted to give the position so far reached in the table. The correct displacement at the current position is thus retrieved from the table. To this is added an offset to get the waves to the correct height and the new point is plotted. It is also put back into the table of positions ready to be erased the next time round.

Fundamentally, we are computing the wave displacement from the equation

```
displacement = amplitude*SIN(wvln*x-time)
```

Physicists will appreciate that the constant called 'wvln' is actually the reciprocal of the wavelength. This could be altered to change the number of waves that appear on the screen (and hence their wavelength). By adjusting 'time' at the completion of each cycle, the wave can be made to move through the table faster or slower. This is a means of adjusting the speed of the waves. The third variable (frequency) depends upon both speed and wavelength and cannot be independently varied.

The great advantage of this technique is the ease with which the wave can be made to travel backwards. The constant 'time' is added instead of subtracted to produce the result. The two displacements for the two waves are then added together to produce the standing waves. Close inspection of the listing in the STANDING WAVES program will reveal exactly how this is done. If you want both waves to travel in the same direction, producing interference, then 'time' must be added or subtracted from both. The program STANDING WAVES does not provide all these facilities, since it is only a sampler. The more sophisticated versions are available from Griffin Software. By way of recompense, LONGITUDINAL WAVES (32) was written after the Griffin package had been put into production, so it does not contain this interesting program. Longitudinal waves are very easy to derive from transverse waves. Instead of plotting the displacement in the vertical direction, it is added to the horizontal position and a line is drawn

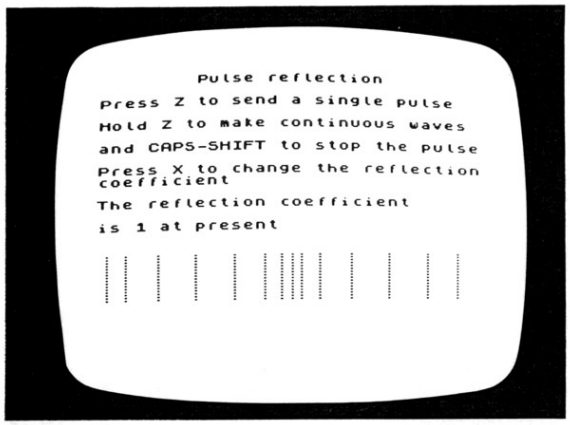


Plate 20 Longitudinal waves

in the required position (Plate 20). No full description is given, but interested readers can disassemble LONGITUDINAL WAVES to see how it is done.

Screen scroll

The layout of the screen makes it possible to shift each pixel into the neighbouring position, using the rotate instruction. This has several applications as we shall see later. I was first alerted to this possibility by Dr S. Rushbrook-Williams of the Microelectronics Educational Development Centre in Paisley.

The Spectrum screen is exactly 256 dots wide, which is perfect for machine code, since this is very easy to count with one of the Z80 internal registers. These dots are held in 32 successive bytes, so a routine to scroll the whole screen sideways is almost trivial.

```
31474  21 00 40        LD HL,4000H  ;Start of screen memory
31477  3E C0           LD A,0C0H    ;192 rows
31479  A7       nxrow  AND A        ;Clear the CARRY bit
31480  06 20           LD B,20H     ;32 bytes per row
31482  CB 1E    nxbyt  RR (HL)      ;Shift right one bit
31484  23             INC HL       ;Move to next byte
```

```
31485  10 FB     DJNZ nxbyt    ;32 bytes done?
31487  3D        DEC A         ;Yes, do next row
31488  20 F5     JR Z,nxrow    ;All rows done?
31490  C9        RET           ;Yes, return to BASIC
```

This routine is listed below as a BASIC program, which can be entered and RUN like any other. Line 210 'calls' the scroll routine to shift the whole screen one dot to the right. This version of the program works with a 48K or a 16K Spectrum.

A particularly useful application of this routine is to add together two sine waves to demonstrate interference or beats. The two component waves and their sum are plotted at the zero x coordinate and the screen then scrolls sieways to display the full wave. In practice the wave motion is rather jerky, but this is because so much of this program is still in BASIC. A full machine-code version has been developed, which is available from Griffin Software.

```
  1  CLEAR 31400
  2  FOR i= 31474 TO 31490
  3  READ x
  4  POKE i,x
  5  NEXT x
  6  DATA 33,0,64,62,192,167,6,32,203,30
  7  DATA 35,16,251,61,32,245,201
 10  REM Select the frequencies
 20  PRINT "Enter the frequency of wave 1"
 30  PRINT "(recommended range 10 to 20)
 40  PRINT
 50  INPUT f1
 60  PRINT "Enter the frequency of wave 2"
 70  PRINT "(recommended range 10 to 20)
 80  PRINT
 90  INPUT f2
100  REM Main program
110  FOR a = 0 TO 10000 STEP 0.01
120  REM Plot the first wave
130  LET y1 = 15*SIN(a*f1)
140  PLOT 0,y1+88
150  REM Plot the second wave
160  LET y2 = 15*SIN(a*f2)
170  PLOT 0,y2+140
180  REM Plot the sum of the waves
190  PLOT 0,y1 + y2 + 38
200  REM Scroll the screen
210  RANDOMIZE USR 31474
220  NEXT a
```


There are many other uses for this routine, for example, a multi-channel chart recorder. The data from an analogue to digital converter would be plotted instead of the waves. This is discussed in the next chapter.

The facility on the ZX Spectrum for loading data from tape into the screen memory is often used for creating pictures (see for example, the introductory Horizon tape). These pictures build up gradually line by line until the loading is complete. The result is less than satisfactory and some software houses prefer to load the bytes onto a black screen first and then switch on the ATTRIBUTES to expose the picture. Apart from the difficulty of using colour properly, this method is still not fast enough. What is required is a machine-code routine to transfer data from another part of the memory into the screen memory.

The following routine makes use of the 'block transfer' instruction of the Z80 microprocessor. The instruction LDIR transfers up to 65 535 bytes from one place to another in a fraction of a second. The Z80 internal registers first have to be set up to indicate the source, where the date comes from (called **SRCE**), the destination, where the data goes to (called **DEST**) and how many bytes should be transferred (called **NMBR**).

For our purposes, this destination is the **screen memory**. This consists of two parts, the **DISPLAY file**, which tells us which dots on the screen are turned on (and which are off) and the **ATTRIBUTES**, which hold the information about the foreground and background colour in each screen position. Fortunately, these two are contiguous, the DISPLAY file running from location 16384 to 22527 and the ATTRIBUTE file from 22528 to 23295. The destination thus starts at 16384 (4000H in hexadecimal) and continues for a total of 6912 (1B00H) bytes.

The source is the place where the picture bytes are stored prior to sending them to the screen. This will be different for 16K and 48K microcomputers. The 16K source address begins at 25088 (6200H), while the 48K source address begins at 57856 (E200H).

In Z80 assembly language the transfer routine is as follows:

```
LD BC,NMBR
LD HL,SRCE
LD DE,DEST
LDIR
RET
```

The LDIR instruction works as follows. The HL register pair contains the source address. The microprocessor goes to this address and collects one byte. The DE register pair contains the address where this byte is to be put, so the microprocessor sends the byte to that address. The HL register pair is then incremented to point to the next memory location. The DE register pair is also incremented to point to the next destination address. Finally, the BC register pair is decremented. The process is then repeated. When the BC register pair reaches zero, NMBR bytes have been transferred as required. A very short routine like this (twelve bytes) can go right at the top of memory. This is where the Spectrum keeps

a note of user-defined characters, so it is quite safe (unless you are using such characters). This is as follows:

16K . . . . . locations 32750 to 32761

48K . . . . . locations 65520 to 65531

Two versions of this program are given for the two different memory sizes:

16K version

```
1  FOR i = 32750 TO 32761
2  READ x
3  POKE i,x
4  NEXT i
5  DATA 1,0,27,33,0,98,17,0,64,237,176,201
```

48K version

```
1  FOR i = 65520 TO 65531
2  READ x
4  NEXT i
5  DATA 1,0,27,33,0,226,17,0,64,237,176,201
```

When run, this loader places the codes of the DATA statement into their correct locations. To execute the machine-code program, enter

```
RANDOMIZE USR 65520 (or 32750)
```

You should see the screen go black instantly. This is because you have transferred a set of 0s to the screen; there was no picture in the source memory to begin with. How do we put one there?

Painting a picture

The introductory Horizon tape contains a useful screen draw utility, which can be used to build a picture. Alternatively, you can write a program to PRINT blocks of colour, lines and any other user-defined characters you wish. When you have finished and the picture has been created, save it on tape. The command for this is

```
SAVE "picture" SCREEN$
```

This is equivalent to

```
SAVE "picture" CODE 16384,6912
```

which saves all 6912 bytes of the screen memory.

The syntax to recall the picture to the screen is

```
LOAD "picture" CODE
```


which automatically loads the bytes from the tape into the screen memory. You observe the picture being built up line by line.

Our improvement on this loads the picture bytes into a different area of memory and transfers them to the screen memory instantaneously. This other memory area must first be protected from BASIC, by altering the top of available memory locations. This is done with the CLEAR command.

16K version

```
CLEAR 25087
```

(reserves about 7000 bytes starting at location 25087)

48K version

```
CLEAR 57855
```

(reserves about 7000 bytes starting at location 57855)

The picture bytes can be inserted into this memory area by specifying the starting address at the time of loading, thus:

16K version

```
LOAD "picture" CODE 25088
```

48K version

```
LOAD "picture" CODE 57856
```

Finally, the USR function calls the machine-code routine and transfers the picture to the screen.

The transfer program

If all this seems complicated, here it is again in the one program. First enter and save this program as "transfer".

16K version

```
1  CLEAR 25087
2  FOR i = 32750 TO 32761
3  READ x
4  POKE i,x
5  NEXT i
6  DATA 1,0,27,33,0,98,17,0,64,237,176,201
7  LOAD " " CODE 25088
8  RANDOMIZE 32750
```

48K version

```
1  CLEAR 57855
2  FOR i = 65520 TO 65531
3  READ x
4  POKE i,x
5  NEXT i
6  DATA 1,0,27,33,0,226,17,0,64,237,176,201
```

```
7  LOAD " " CODE 57856
8  RANDOMIZE USR 65520
```

The picture should be created and saved as indicated above. Then the transfer program should be loaded and run. The screen will go blank as the Spectrum executes line 7 and tries to read data from the tape. Rewind the tape to the beginning of the picture bytes and play it back in the usual way. Line 7 does not specify any particular picture, so any properly prepared tape will be loaded by this line.

The same routine may be used to transfer several different pictures one after the other in rapid succession. This gives a cartoon effect, which can be used for animations. The 16K Spectrum has too little memory for this, but the 48K machine allows five such pictures to be stored in different sections of the memory, to be recalled at will. This is useful for flashing instant pictures, for example, the happy face or sad face to indicate a correct or incorrect response to a question.

Large digits

The first machine-code graphics routine I ever wrote was for displaying numbers in large digits. This routine has been used extensively ever since. It also appears in several of the programs listed in the Appendix. Fundamentally, it produces figures that are eight times their normal size. By using the standard seven-segment type display, 'chunkiness' in the displayed digits is avoided. Instead of switching a single pixel on or off, they switch whole ATTRIBUTE positions. The positions of the ATTRIBUTES to be turned on or off are held in a digits table (Plate 21).

The routine discussed below resides near the top of RAM in the 48K Spectrum, but it can be relocated. REACTION TIMER (6) uses the same routine in locations below 32768, which is the version of the routine for the 16K Spectrum.

Because there are thirty-two squares across the screen (nearly five times six) and there are twenty-four down the screen (three times eight) then the choice of a six by eight matrix allows fifteen different large digit positions on the screen, or three rows each of five digits. Even with a negative sign and a decimal point, this is enough. Each digit actually only occupies five columns and seven rows of the matrix, thus allowing a border to separate each character from its neighbour.

We use eight bytes to store the rows of any one large digit, using one bit for each column position in each row. If the bit at, say, position 7 is a 1, then the screen square corresponding to that position in the matrix is turned on (black square). If the bit in position 7 is a 0 then the corresponding screen position is turned off (white square). Thus a row of eight black and white squares can be stored in a single byte. In practice only five of these bits are used (bits 0 to 4), bits 5 to 7 are switched off. The sets of eight bytes for each digit are stored sequentially in a table called **dgttbl**. The first part of the program gets the digit code (which is passed via BASIC in a location called **dgtval** - 0FE02H), multiplies it by eight and enters

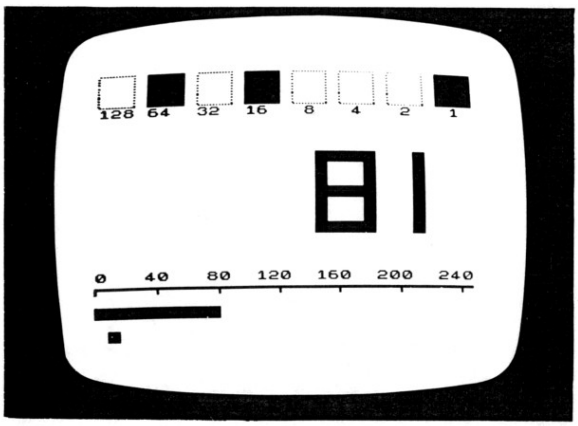


Plate 21 Attribute manipulation in machine code

bittbl to collect the eight bytes of the selected digit. These are kept in a set of eight temporary stores called **temp**.

The starting position for each large digit is specified by passing it through locations 0FE00H and 0FE01H. The full screen can be considered as having up to twenty-four columns on any of three rows. Then it is only necessary to pass two values corresponding to the ultimate row and column positions of the large digit. This is the position of its upper-left corner. These locations are loaded into the HL register pair, which then points to the required screen values position. Having obtained the bytes of the digit to be displayed and its screen position, it remains to look at each bit of each byte in turn and to send a black or a white character to the appropriate position on the screen. This is done using the **RRA** instruction (**rotate right ACCUMULATOR**) and looking at the CARRY bit to see if it is a 0 or a 1. The routine needs three counters, one to keep the screen position (HL pair), one to count the eight bytes of each digit (D register) and a third to keep track of the bits within each byte (E register).

BIGDIG

```
row    = 0FE00H
col    = 0FE01H
dgtval = 0FE02H
dgttbl = 0FD00H upwards
65040 2A 00 FE          LD HL,(0FE00H) ;Get screen location
65043 3A 02 FE          LD A,(dgtval)  ;Get digit code value
65046 37                SCF
65047 3F                CCF            ;Clear CARRY bit
65048 17                RLA            ;Multiply by eight
65049 17                RLA
65050 17                RLA
65051 4F                LD C,A         ;Point to correct digit
65052 06 FD             LD B,dgttbl    ;Point to digits table
65054 16 08             LD D,08H       ;Eight bytes
65056 0A          nxbyte LD A,(BC)     ;Get byte
65057 1E 06             LD E,06H       ;Six bits
65059 1F          nxbit  RRA            ;Rotate bit into CARRY
65060 30 04             JR NC,white    ;Black square?
65062 36 00             LD (HL),00H    ;Yes
65064 18 02             JR bitdone     ;Try next bit
65066 36 38       white LD (HL),38H    ;Send white square
65068 2C          bitdone INC L        ;Move to next position
65069 1D                DEC E          ;All bits done?
65070 20 F3             JR NZ,nxbit    ;No, do next bit
65072 F5                PUSH AF        ;Save ACCUMULATOR
65073 7D                LD A,L         ;Move down to next row
65074 C6 1A             ADD A,1AH
65076 6F                LD L,A
65077 F1                POP AF         ;Restore ACCUMULATOR
65078 0C                INC C          ;Next byte
65079 15                DEC D          ;All bytes done?
65080 20 E6             JR NZ,nxbyte   ;No, do next byte
65082 C9                RET            ;Yes, all done.
```

Conclusion

This chapter has come a long way and some readers may well feel that it is not for them. I did warn that machine-code programming was not easy, but no matter. Study the program listings to see the way that each routine is used and you should then be able to make use of them yourself, even if you cannot see how they work. Using the techniques discussed in this chapter, there is no reason why you cannot create your own machine-code graphics routines. It is a difficult art to master, but the rewards are well worth it. Commercial games packages rely heavily on

machine-code graphics, and it is against these that our pupils will compare educational software. What will be their response if we offer an inferior alternative?

8 Interfacing in machine code

'Now! Now! cried the Queen. 'Faster! Faster!'
(Lewis Carroll, *Through the Looking Glass*)

This chapter brings together previous ideas to produce useful routines for making fast measurements. First, let us look at the Interspec from the point of view of machine code. The addresses are exactly the same and the instructions used to access them are the assembly language equivalents of the BASIC IN and OUT instructions.

Relay outputs (OUT 63,8 in BASIC)

```
3A  08      LD A,8
D3  3F      OUT (A),63      ;Output bit 3 on, all others off
```

Switch inputs (LET status = IN 63)

```
DB  3F      IN A,(63)
50  nn nn   LD (status),A
```

Analogue input (OUT 31,1:PAUSE 1:LET X = IN 31)

```
3A  01      LD A,1
D3  1F      OUT (A),31      ;Select channel 0
CD  nn nn   CALL delay      ;delay for 100 microseconds
DB  1F      IN (A),31       ;Read selected channel
```

STOPCLOCK

Using machine-code versions of the programs discussed in Chapters 4 and 5 allows faster measurements and turns the microcomputer into a very powerful laboratory aid. One example of this is STOPCLOCK (7), the use of which was discussed in Chapter 4. This prints the current value of a centisecond clock on the screen in large digits, which is updated fifty times a second. This 'clock' is actually the internal clock of the Spectrum (known as the frame counter), which is accessed in machine code through location 23672. The Spectrum's own operating system updates this clock every fiftieth of a second.

The full BASIC program is listed in the Appendix. The following description may help to explain it.

Lines 29 to 44: the special codes for the large digits, the letters m, s and the decimal point. These are used by the BIGDIG subroutine, which is loaded by lines 10 to 22.

Lines 45 to 78 are the main timing routine.
Lines 80 to 87 contain the routine to display the contents of the minute, second and centisecond timers continuously.

The assembly language routine

6400 to 64095: the minute, second and centisecond stores are initialized to zero and then displayed on the screen by the subroutine called showtimes.
64096 to 64117: bits 0 and 1 of the user port are collected and stored in STAT. The program then sits at this point until one or other of these bits changes.
64118 to 64250: the centisecond store is cleared and the timing begins. Every fiftieth of a second the Spectrum operating system adds one to the frame counter in location 23672 (called CSLO). If this location contains a number less than five, the routine jumps to see if any key on the keyboard is being pressed. If not, the current time measured by the clock is displayed. If a key is being pressed, the showtimes routine is bypassed and the clock display freezes at its previous value although the timer itself continues to measure time. After the current time has (or has not) been displayed, the routine checks the inputs to see if any have changed. If so a return to BASIC is made. If not the routine goes back to check the current value of CSLO (**tmwait**).

When CSLO exceeds five, it is reset to zero and CSHI is incremented, thus effectively counting ten centiseconds. When it reaches ten, it too is reset and one second is added to the time. This process continues up to 100 minutes or until the input status changes. The clock stores continue to be incremented every two hundredths of a second even if the display is temporarily frozen. It would be more difficult to do this if a timing loop was being used to generate the time intervals.
64256 to 64376: the showtimes subroutine collects the contents of each of CSLO, CSHI, etc. and displays each in its correct position with the large-digits display subroutine discussed in Chapter 7. In this listing the codes are in decimal.

```
64000   62,0        LD A,0          ;Initialize clock stores
64002   50,0,252    LD (CSHI),A
64005   50,1,252    LD (SECLO),A
64008   50,2,252    LD (SECHI),A
64011   50,3,252    LD (MINLO),A
64014   50,4,252    LD (MINHI),A
64017   0           NOP
64018   0           NOP
64019   62,12       LD A,12         ;Display letter m
64021   50,2,254    LD (DGT),A      ;Send digit code
64024   62,24       LD A,24         ;Fifth position
64026   50,0,254    LD (POS),A
64029   62,88       LD A,88         ;Top row
64031   50,1,254    LD (ROW),A
```

```
64034  205,16,254        CALL display  ;Display in chosen place
64037  0                 NOP
64038  0                 NOP
64039  62,13             LD A,13       ;Display letter s
64041  50,2,254          LD (DGT),A    ;Send digit code
64044  62,24             LD A,24       ;Fifth position
64046  50,0,254          LD (POS),A
64049  62,89             LD A,89       ;Middle row
64051  50,1,254          LD (ROW),A
64054  205,16,254        CALL display
64057  62,10             LD A,10       ;Display decimal point
64059  50,2,254          LD (DGT),A    ;Send digit code
64062  62,6              LD A,6        ;First position
64064  50,0,254          LD (POS),A
64067  62,90             LD A,90       ;Bottom row
64069  50,1,254          LD (ROW),A
64072  205,16,254        CALL display
64075  62,0              LD A,0
64077  50,120,92         LD (CSLO),A   ;Zero centisecond digit
                                      ;This is the frame counter
64080  205,0,251         CALL showtim  ;Show current time
                                      ;Wait until an event occurs
64083  219,63            IN A,(63)     ;Get input status
64085  230,3             AND 3         ;Mask to get lower bits
64087  50,5,252          LD (STAT),A   ;Save in status store
64090  219,63      wait   IN A,(63)     ;Get input status again
64092  230,3             AND 3         ;Mask to get lower bits
64094  71                LD B,A        ;Keep in temp store
64095  58,5,252          LD A,(STAT)   ;Compare with previous status
64098  184               CP B
64099  40,245            JR Z,wait     ;Wait till input changes
64101  120               LD A,B        ;Alter status store
64102  50,5,252          LD (STAT),A   ;Initialize clock
64105  62,0              LD A,0
64107  50,120,92         LD (CSLO),A   ;Set frame counter to zero
64110  58,120,92 tmwait  LD A,(CSLO)   ;Wait for timeout
64113  254,5             CP 5          ;Five fiftieths?
64115  218,223,250       JP C,cont     ;No, check inputs
64118  62,0              LD A,0        ;Reset to zero
64120  50,120,92         LD (CSLO),A
64123  58,0,252          LD A,(CSHI)   ;Update next digit
64126  198,1             ADD A,1
64128  50,0,252          LD (CSHI),A
64131  254,10            CP 10         ;Ten tenths?
```



```
64133  218,223,250       JP C,cont     ;No, check inputs
64136  62,0              LD A,0        ;Reset to zero
64138  50,0,252          LD (CSHI),A
64141  0                 NOP
64142  0                 NOP
64143  58,1,252          LD A,(SECLO)  ;Update next digit
64146  198,1             ADD A,1
64148  50,1,252          LD (SECLO),A
64151  254,10            CP 1          ;Ten seconds?
64153  218,223,250       JP C,cont     ;No, check inputs
64156  62,0              LD A,0        ;Reset to zero
64158  50,1,252          LD (SECLO),A
64161  0                 NOP
64162  0                 NOP
64163  58,2,252          LD A,(SECHI)  ;Update next digit
64166  198,1             ADD A,1
64168  50,2,252          LD (SECHI),A
64171  254,6             CP 6          ;Sixty seconds?
64173  218,223,250       JP C,cont     ;No, check inputs
64176  62,0              LD A,0        ;Reset to zero
64178  50,2,252          LD (SECHI),A
64181  0                 NOP
64182  0                 NOP
64183  58,3,252          LD A,(MINLO)  ;Update next digit
64186  198,1             ADD A,1
64188  50,3,252          LD (MINLO),A
64191  254,10            CP 10         ;Ten minutes?
64193  218,223,250       JP C,cont     ;No, check inputs
64196  62,0              LD A,0        ;Reset to zero
64198  50,3,252          LD (MINLO),A
64201  0                 NOP
64202  0                 NOP
64203  58,4,252          LD A,(MINHI)  ;Update next digit
64206  198,1             ADD A,1
64208  50,4,252          LD (MINHI),A
64211  254,10            CP 6          ;100 minutes?
64213  218,223,250       JP C,cont     ;No, check inputs
64216  62,0              LD A,0        ;Reset to zero
64218  50,4,252          LD (MINHI),A
64221  0                 NOP
64222  0                 NOP
                                      ;Check SPACE key
```

```
64223   62,127      cont   LD A,127
64225   219,254            IN A,(254)
64227   246,224            OR 224
64229   254,255            CP 255
64231   194,110,250        JP NZ,tmwait
64234   205,0,251          CALL showtim   ;Display digits
64237   219,63             IN A,(63)      ;Check inputs
64239   230,3              AND 3
64241   71                 LD B,A
64242   58,5,252           LD A,(STAT)
64245   184                CP B
64246   202,110,250        JP Z,tmwait
64249   201                RET            ;Finish at this event
```

Subroutine to display current time (showtime)

```
64256   58,120,92          LD A,(CSLO)    ;Display CSLO (fiftieths)
64259   7                  RLCA           ;Multiply by two for centisecs
64260   50,2,254           LD (DGT),A     ;Send digit code
64263   62,18              LD A,18        ;Fourth position
64265   50,0,254           LD (POS),A
64268   62,90              LD A,90        ;Bottom row
64270   50,1,254           LD (ROW),A
64273   205,16,254         CALL display
64276   58,0,252           LD A,(CSHI)    ;Display CSHI
64279   50,2,254           LD (DGT),A     ;Send digit code
64282   62,12              LD A,12        ;Third position
64284   50,0,254           LD (POS),A
64287   62,90              LD A,90        ;Bottom row
64289   50,1,254           LD (ROW),A
64292   205,16,254         CALL display
64295   0                  NOP
64296   58,1,252           LD A,(SECLO)   ;Display SECLO
64299   50,2,254           LD (DGT),A     ;Send digit code
64302   62,18              LD A,18        ;Fourth position
64304   50,0,254           LD (POS),A
64307   62,89              LD A,89        ;Middle row
64309   50,1,254           LD (ROW),A
64312   205,16,254         CALL display
64315   0                  NOP
64316   58,2,252           LD A,(SECHI)   ;Display SECHI
64319   50,2,254           LD (DGT),A     ;Send digit code
64322   62,12              LD A,12        ;Third position
64324   50,0,254           LD (POS),A
64327   62,89              LD A,89        ;Middle row
```



```
64329   50,1,254           LD (ROW),A
64332   205,16,254         CALL display
64335   0                  NOP
64336   58,3,252           LD A,(MINLO)   ;Display MINLO
64339   50,2,254           LD (DGT),A     ;Send digit code
64342   62,18              LD A,18        ;Fourth position
64344   50,0,254           LD (POS),A
64347   62,88              LD A,88        ;Middle row
64349   50,1,254           LD (ROW),A
64352   205,16,254         CALL display
64355   0                  NOP
64356   58,4,252           LD A,(MINHI)   ;Display MINHI
64359   50,2,254           LD (DGT),A     ;Send digit code
64362   62,12              LD A,12        ;Third position
64364   50,0,254           LD (POS),A
64367   62,88              LD A,88        ;Middle row
64369   50,1,254           LD (ROW),A
64372   205,16,254         CALL display
64375   0                  NOP
64376   201                RET
```

Timing-loop routines

We saw in Chapter 4 how the relay outputs may be switched on for controlled intervals of time using simple PAUSE instructions in BASIC. The maximum rate at which these outputs can be switched on and off in this way is limited to about 100 Hz and this is inadequate for most purposes. A better way is to use machine-code delay loops, as we did when moving characters across the screen in Chapter 7. However we were not then interested in accuracy.

Since timing loops are used extensively for accurate measurement of short intervals, they will now be described. We shall use them to switch outputs rapidly on and off to produce sound in a suitable loudspeaker. This particular application (chosen only to illustrate the principles) is not a sensible way for making sound because BEEP already exists in Spectrum BASIC. It is, however, a useful way of producing square wave pulses and is the assembly-language routine for PULSER (15). The algorithm is as follows:

i) Switch the output on
ii) Delay for half-period
iii) Switch the output off
iv) Delay for other half-period
v) Go back to step i

Each machine-code instruction takes a specific number of T-states. In the Spectrum the T-states occur at 3.5 MHz so it is possible to calculate accurately

how long any set of instructions lasts. There are two major problems. Firstly, the ULA in the Spectrum is continually interrupting the microprocessor to update the frame counter. This causes unpredictable delays in the timing loop. To avoid this the interrupts from the ULA are prevented by disabling all interrupts with the DI instruction. Before returning to BASIC, the interrupts are enabled again with EI. Unfortunately, this does not always work, since the ULA in the Spectrum appears to have priority over the microprocessor. To avoid conflict between the two, they must be kept separate. The ULA only accesses the lower 32K of memory, so if the microprocessor is temporarily restricted to the top 32K, the interrupt problem is avoided. Thus all timing routines based upon the method of counting T-states must be located at the top of the memory.

The second problem is that conditional instructions take different times to execute, depending on whether the condition succeeds or fails. Generally, a condition that succeeds takes five T-states longer than one that fails. This has to be calculated and allowed for.

PULSER

```
64000 243              DI              ;Disable interrupt system
64001 62,255    rpt    LD A,255        ;All outputs on
64003 211,95           OUT (95),A
64005 237,75,0,251     LD BC,(length)  ;Pulse length (20 T-states)
64009 13        dly1   DEC C           ;(4 T-states)
64010 32,253           JR NZ,dly1      ;(7 or 12 T-states)
64012 16,251           DJNZ dly1       ;(8 or 13 T-states)
64014 62,0             LD A,0          ;All outputs off
64016 211,95           OUT (95),A
64018 237,75,2,251     LD BC,(time)    ;Time between pulses
64022 13        dly2   DEC C
64023 32,253           JR NZ,dly2
64025 16,251           DJNZ dly2
64027 62,127           LD A,127        ;Check keyboard for return
64029 219,254          IN A,(254)
64031 246,224          OR 224
64033 254,255          CP 255
64035 40,220           JR Z,rpt        ;No key, continue pulses
64037 251              EI              ;Restore interrupts
64038 201              RET
```

The length of the pulse is collected from locations 64256 (low byte) and 64257 (high byte) and placed in the BC register pair. C is decremented to zero and when it reaches zero the B register is decremented. This is better than decrementing the BC pair as a whole, since DEC BC does not check when the final result is zero. The total time elapsed is as follows. Suppose the location length contains 1000, (low byte 232, high byte 3). The pair of instructions in lines 64009 and 64010 above are first executed 232 times. For the first 231 times, the jump condition fails, so the

time taken is 4 + 12 T-states. On the last occasion the condition fails, so it takes 4 + 7 T-states. At this point the DJNZ dly1 instruction is executed, which takes 13 T-states (since it succeeds). The C register is then decremented 256 times more, 255 taking 16 T-states and one taking 11 T-states. This is repeated until the B register reaches zero and the timing loop ends. The time to load the BC register pair and the time to execute the instructions to switch the outputs must also be added in. This gives:

```
rpt     LD A,255        ;7 T-states
        OUT (95),A      ;11 T-states
        LD BC,(length)  ;20 T-states
```

First time:

```
dly1    DEC C           ;16*231+11 = 3707 T-states
        JR NZ,dly1
```

The following occurs three times

```
dly1    DEC C           ;16*255+11 = 4091 T-states
        JR NZ,dly1
```

Finally

```
        DJNZ dly1       ;13*2 + 8
        TOTAL           7+11+20+3707+3*4091+26+8 = 16052
```

The time before switching on again is similarly calculated, except that 43 T-states must be added for the keyboard sensing routine, as follows:

```
LD A,127        ;7 T-states
IN A,(254)      ;10 T-states
OR 224          ;7 T-states
CP 255          ;7 T-states
JR Z,rpt        ;12 T-states unless routine finishes
```

On this occasion the length of the timing loop is determined by the contents of locations 64258 and 64259 (called **time**). Assuming that time also contains 1000, then the time from the 'off' edge to the 'on' edge is thus 16052 + 43 or 16095 T-states. The pulse frequency is therefore 16052 + 16095 or 32147 T-states, which takes 9.185 milliseconds (each T-state takes 1/3.5 microseconds).

The basic routine makes several approximations to this process, but uses fundamentally the same calculation in reverse to determine what numbers to POKE into length and time before the routine is called.

The principle of measuring time intervals is similar. The inputs are read and stored in a memory location called **status**. The current state of the inputs is then monitored continuously and compared with status. Normally they will be the same, but when they are different, this is because an input has been activated. A clock is then started and the new status of the inputs is saved in status. When the inputs again change state, the current contents of the clock are noted and copied into a store. The time interval involved can then be calculated and displayed. We

saw in Chapter 4 how this routine was used with the internal clock to produce SIMPLE TIMER in BASIC (Example 12).

Accurate timing of short intervals is only possible using machine-code routines, since BASIC is too slow to respond to input changes. The use of timing loops for this purpose will now be described.

FAST TIMER

```
64000   243             DI          ;Disable interrupts
64001   1,0,0           LD BC,00H   ;Set timer to zero
64004   219,63  wtoff   IN A,(63)   ;Get input status
64006   230,1           AND 1       ;Mask to get bit 0
64008   32,250          JR NZ,wtoff ;Wait till bit goes off
64010   219,63  wton    IN A,(63)   ;Get input status
64012   230,1           AND 1       ;Mask to get bit 0
64014   40,250          JR Z,wton   ;Now wait till bit goes on
64016   3       count   INC BC      ;Begin timing
64017   219,63          IN A,(63)   ;Get input status
64019   230,1           AND 1       ;Mask to get bit 0
64021   32,249          JR NZ,count ;Continue till finished
64023   251             EI          ;Restore input status
64024   201             RET
```

The statement in line 800 of the BASIC program (listed in the Appendix) places the final contents of the BC register pair into the variable t automatically. From there the value is converted to a time interval and sent to the large-digit routine to be displayed.

The timing on this occasion is

```
count   INC BC          ;6 T-states
        IN A,(63)       ;10 T-states
        AND 1           ;7 T-states
        JR NZ,count     ;12 T-states
```

The total time per loop is thus 35 T-states, or ten microseconds. This is dependent on the accuracy of the Spectrum crystal oscillator, and if found to be inaccurate, the conversion factor in line 810 of the BASIC program will need to be changed accordingly.

The maximum count is 65535, giving a maximum measurable time of 655 milliseconds. If this is exceeded, a wrong answer will be displayed, since the routine has no mechanism for checking an overflow of the BC register pair.

For still longer time intervals, a three- or four-byte clock may be used. The method of incrementing this clock is not now visible, since compensatory delays are needed to allow for the occasions when the higher bytes are not incremented. It is better to use the technique of adding one unit to the clock during each loop instead. The CARRY bit from a low-byte addition can be added in to the next byte by adding in zero each time. The three bytes for the clock are kept in registers E, D

and L, called **clocklo**, **clockmid** and **clockhi** respectively. This 24-bit clock can count up to 16 777 216 and can measure times up to several minutes.

Count subroutine

```
count   LD A,clocklo        4 T-states
        ADD A,1             7 T-states
        LD clocklo,A        4 T-states
        LD A,clockmid       4 T-states
        ADC A,0             7 T-states
        LD clockmid,A       4 T-states
        LD A,clockhi        4 T-states
        ADC A,0             7 T-states
        LD clockhi,A        4 T-states
        IN A,(254)          10 T-states
        AND 1               7 T-states
        JR Z,done           7 T-states unless routine finishes
        IN A,(63)           10 T-states
        AND 3               7 T-states
        LD C,A              4 T-states
        CP status           4 T-states
        JR Z,count          12 T-states unless timing has finished.
```

It takes 106 T-states to complete this loop so the clock will increase by one unit approximately every thirty microseconds. The exact time was determined by accurate measurement over a long time period and the conversion factor in line 4005 of the BASIC programme (TSA, 9) was adjusted accordingly.

Once entered, this loop continuously counts the time. There are two ways of leaving the loop. If the SPACE key is pressed during the timing, then the keyboard detect routine will detect it and will terminate the loop. Alternatively, if there has been some change at either of the inputs so that it no longer compares with status, then the program goes off to find out what caused the change.

This routine could be used as it is, to measure short time intervals. With a photocell (connected to bit 0 of the switch inputs) and mounted inside a camera to measure how long its shutter remains open. However, the routine actually used in programs 9 to 14 has been made even more powerful by including extra facilities. Firstly, it allows up to thirty-two different time intervals to be measured consecutively. This means that it can be used for a variety of purposes, particularly the measurement of an a.c. frequency (which requires several cycles to be counted), the measurement of the speeds of a trolley as it runs down an inclined plane and the measurement of acceleration. TSA (9) uses this advanced-timing routine and the large-digits routine to display the results.

To allow the measurement of speed when studying the laws of collision between two trolleys, there must be two photocells. It is possible for the second trolley to begin a transit of its photocell before the first has finished crossing the other. Thus it must be possible to detect two inputs independently and to keep

their results separate. We still only need the one clock, but at the start or finish of a transit, the time on the clock is copied into a store. In fact, up to sixteen stores are available for each input and the pointers (**ptr**) keep track of which status change is currently being timed. Thus in the collision experiment it would be possible to have two trolleys approach from different directions, to collide in the middle and both go off in one particular direction at different speeds. This involves two events at one input and six events at the other, but the routine can easily cope with this.

The whole routine has a method for deciding how long it has to continue taking readings, since the number of events is kept in location 64247 (**evntctr**) beforehand. It also has an escape route, for the occasion when you run the program and find that the photocell is not working. This is achieved by the keyboard-detect routine.

The final part of the routine (**done**) is a means of converting the recorded clock times into time intervals. This is carried out for all of the stores even if most of them are empty.

```
Store=0FB00H (up to 0FBFFH)
evntctr=0FAF7(number of events)
status is B register
clocklo is E register
clockmid is D register
clockhi is L register

64000  243              DI
64001  6,0              LD B,00H          ;Set counter to 256
64003  221,33,0,251      LD IX,0FB00H      ;Clear all stores
64007  221,54,0,0 nxt    LD (IX+00H),0
64011  221,35           INC IX            ;Next store
64013  16,248           DJNZ nxt
64015  22,0             LD clocklo,0
64017  30,0             LD clockmid,0
64019  46,0             LD clockhi,0
64021  62,252           LD A,252          ;Set pointers to -4
64023  50,243,250       LD (ptr1lo),A     ;Save channel 1 pointer
64026  62,124           LD A,124
64028  50,245,250       LD (ptr2lo),A     ;Save channel 2 pointer
64031  62,251           LD A,251          ;Set high byte of pointers
64033  50,244,250       LD (ptr1hi),A
64036  50,246,250       LD (ptr2hi),A
64039  219,63           IN A,(63)         ;Get input status
64041  230,3            AND 3             ;Mask for bits 0 and 1
64043  71               LD status, A      ;Keep current status
64044  219,63           IN A,(63)         ;Get input status
64046  230,3            AND 3             ;Mask for bits 0 and 1
```



```
64048  79               LD C,A            ;Keep current status
64049  184              CP status         ;Same status?
64050  40,248           JR Z,wait         ;Wait till it changes
64052  0                NOP               ;Status has changed
                                          ;Determine which channel
64053  121        query LD A,C            ;Retrieve new status
64054  168              XOR status        ;Which channel?
64055  65               LD status.,C      ;Keep new status
64056  254,1            CP 1              ;Channel 1?
64058  40,8             JR Z,chan1        ;Yes
64060  254,2            CP 2              ;Channel 2?
64062  40,18            JR Z,chan2        ;Yes
64063  121              LD A,C            ;Both channels at once
64065  238,2            XOR 2             ;Ignore channel 2 this time
64067  71               LD status A
64068  58,243,250 chan 1 LD A,(ptr1)      ;Point to channel 1 counter
64071  198,4            ADD A,4           ;Update pointer
64073  50,243,250       LD (ptr1),A       ;Save pointer
64076  221,42,243,250    LD IX,(ptr1)      ;Set IX to pointer
64080  24,12            JR store          ;Store current time
64082  58,245,250 chan2 LD A,(ptr2)      ;Point to channel 2 counter
64085  198,4            ADD A,4           ;Update pointer
64087  50,245,250       LD (ptr2),A       ;Save pointer
64090  221,42,245,250    LD IX,(ptr2)      ;Set IX to pointer
                                          ;Save current clock
64094  221,115,0   store LD (IX+0),clocklo
64097  221,114,1        LD (IX+1),clocklo
64100  221,117,2        LD (IX+2),clocklo
64103  58,247,250       LD A,(evntctr)    ;All events done?
64106  61               DEC A
64107  50,247,250       LD (eventctr)
64110  40,28            JR Z,done         ;Quit if all events done
64112  123        count LD A,clocklo      ;Increment clock
64113  198,1            ADD A,1
64115  95               LD clocklo,A
64116  122              LD A,clockmid
64117  206,0            ADC A,0
64119  87               LD clockmid,A
64120  125              LD A,clockhi
64121  206,0            ADC A,0
64123  111              LD clockhi,A
64124  219,254          IN A,(254)        ;Check keyboard
64126  230,1            AND 1
64128  40,10            JR Z,done         ;Finish if SPACE pressed
```

```
64130   219,63          IN A,(63)       ;Check input status
64132   230,3           AND 3           ;Bits 0 and 1 only
64134   79              LD C,A          ;Save new status
64135   184             CP status       ;Has it changed?
64136   40,230          JR Z,count      ;Continue if no change
64138   24,169          JR query        ;Check which channel
64140   6,64      done  LD B,40H        ;64 stores to be altered
64142   221,33,248,251  LD IX,0FBF8H ;End store
64146   221,126,4 nxsub LD A,(IX+4)
64149   221,150,0       SUB (IX+0)
64152   221,119,4       LD (IX+4),A
64155   221,126,5       LD A,(IX+5)
64158   221,158,1       SBC A,(IX+1)
64161   221,119,5       LD (IX+5),A
64164   221,126,6 nxsub LD A,(IX+6)
64167   221,158,2       SBC A,(IX+2)
64170   221,119,6       LD (IX+6),A
64173   221,45          DEC IX
64175   221,45          DEC IX
64177   221,45          DEC IX
64179   221,45          DEC IX          ;Point to next block
64181   16,219          DJNZ nxsub
64183   251             EI
64184   201             RET
```

Fast digital to analogue conversion

In Chapter 5 we noted that the frequency of the alternating voltage produced by a DAC via BASIC was limited to a few hertz. I stated then that for higher frequencies it is necessary to do all the calculations in BASIC beforehand and store the results in the memory as individual bytes. These are then collected one by one from the memory and sent directly to the DAC using a machine-code routine. The waveform is created by BASIC before the machine-code routine is called. This gives a table of numbers between 0 and 255 held in a set of sixty-four locations called **store**. The machine-code routine outputs these numbers to the DAC one by one. A delay routine similar to that used before alters the rate at which the numbers are sent to the DAC and thus changes the frequency of the waveform. The length of this delay is loaded from BASIC into a location called **count** before the DAC output routine is called.

PROGRAMMABLE OSCILLATOR (16) is based upon this routine. As with the BASIC programs already discussed, different waveforms are produced by altering the defining equation. The waveform can be inspected by connecting the DAC output to a cathode ray oscilloscope or turned into sound with a suitable amplifier and loudspeaker.

PROGOSC

```
count = 0FA64
store = 0FAC0H to 0FAFFH
64000   243             DI
64001   33,192,250  rpt LD HL,store   ;Set pointer
64004   126         nxt LD A, (HL)    ;7 T-states
64005   211,127         OUT (127),A   ;11 T-states
64007   58,100,250      LA A,(count)  ;13 T-states
64010   61          dly DEC A         ;16*(count)-5
64011   32,253          JR NZ,dly     ;12 T-states unless end
64013   44              INC HL        ;Point to next data
64014   32,244          JR NZ,nxt
64016   62,127          LD A,127      ;Quit if SPACE pressed
64018   219,254         IN A,(254)
64020   246,224         OR 224
64022   254,255         CP 255
64024   40,231          JR Z,rpt      ;Continue if no key
64026   251             EI
64027   201             RET
```

The period varies from about 3750 T-states to about 265 000 in steps of 1024, depending upon the value in count. This gives a frequency between 13 Hz and 1 kHz approximately. Higher frequencies can be obtained by putting more than one cycle of the waveform into store to begin with, although this will reduce the resolution obtained. Even so, a mere eight voltage levels per waveform cycle still gives acceptable sound, in which case the frequency can be as high as 8 kHz.

This routine is incorporated in the program PROGRAMMABLE OSCILLATOR (16), which is very useful for producing alternating voltages. From an electronic engineering viewpoint, its waveform can have almost any shape, so it can be used to analyse the behaviour of filter circuits. For this purpose the output from the DAC can be boosted as described in Chapter 5.

Fast analogue to digital conversion

In Chapter 5 we saw how readings from the ADC may be plotted on the screen. If the measured voltages are changing rapidly however, BASIC is too slow and a machine-code routine is needed to collect the readings and to process them or store them for future use.

One possible application is to read the voltage input at two channels and to plot them directly on the screen with the plot routine described in Chapter 7. The result is rather like a cathode-ray oscilloscope in X–Y mode, with the advantage of producing a fixed display. It can thus be used for transient phenomena as before. The important part of the routine is as follows:

```
32000   62,0        nxt LD A,0
32002   211,31          OUT (31),A ;Select channel 0
32004   62,50           LD A,50    ;Delay for 100 microseconds
32006   61          dly1 DEC A
32007   32,253           JR NZ,dly1
32009   219,31          IN A,(31)  ;Read y-value
32011   47              CPL        ;Invert it
32012   254,176         CP 176     ;Too big for screen?
32014   56,2            JR C,inrng ;Acceptable value
32016   62,175          LD A,175   ;Reset to maximum value
32018   87          inrng LD D,A   ;Keep y-coordinate
32019   62,1         nxt LD A,1
32021   211,31          OUT (31),A ;Select channel 1
32023   62,50           LD A,50    ;Delay for 100 microseconds
32025   61          dly2 DEC A
32026   32,253          JR NZ,dly2
32028   219,31          IN A,(31)  ;Read y-value
32030   79              LD C,A     ;Keep x-coordinate
```

The remainder is the plot subroutine similar to that previously described.

The four-channel chart recorder (22) uses the same procedure, except that the values read from the ADC channels are first divided by eight and then plotted at various heights on the left of the screen. After each reading, the screen is scrolled to the right with the scroll routine described in Chapter 7.

Storage oscilloscope

The program STRGOSC (18) reads up to four channels in rapid succession. The minimum time between starting and completing a conversion is about 200 microseconds, which is the best that can be achieved considering the erratic nature of the Spectrum clock. Because of the time needed to collect the results, the minimum delay between readings is about 250 microseconds or 40 000 readings per second. To decrease this is simply a matter of increasing the delay loop. This provides a maximum delay of about fifty milliseconds (twenty readings per second). Lower rates than this can conveniently be handled in BASIC.

At full speed the time required to collect all 256 readings is only a few milliseconds. There thus has to be some means of telling the routine when to begin taking readings. A hardware solution is to use the relay contacts to begin the transient, but this can cause contact bounce problems (themselves of the order of a few milliseconds).

The software solution to this problem is based on the assumption that nobody is interested in the voltage until it starts to change. So the routine waits until it changes, before beginning to store readings regularly. In practice, ordinary fluctuations due to electrical noise means that the required change should be substantial, a change in the lowest four bits at least. This is a little complicated,

since the change may be positive or negative, so the absolute value of the change must be retrieved before the comparison can be made.

A faster and simpler technique is to compare the measured ADC value with a previously declared threshold value. When the measured value exceeds the threshold, then readings can begin. Unfortunately, this method means that the program cannot be used with, say, capacitor discharge, where the voltage approaches the threshold from the other direction. In such cases the JR NC instruction needs to be replaced by JR C. The 'wait for a change' method covers both eventualities, although it may take too long for some purposes (e.g. the light output from a flashgun). Which technique is used depends on the application. Only channel 0 is monitored in this manner, so channel 0 should be used to signal the first event from which readings are taken. The full assembly listing is given below.

STORAGE OSCILLOSCOPE

```
63996   243             DI
63997   33,6,251        LD HL,0FB06 ;Set pointer to stores
64000   62,0            LD A,0      ;Select channel 0
64002   205,100,250     CALL get    ;Collect ADC reading
64005   87              LD D,A      ;Save input voltage
64006   62,0        wait LD A,0     ;Select channel 0
64008   205,100,250     CALL get    ;Collect ADC reading
64011   95              LD E,A      ;Save new input voltage
64012   146             SUB D       ;Find difference
64013   48,1            JR NC,pos   ;Positive difference
64015   47              CPL         ;Negative so invert
64016   230,240     pos AND 240     ;Eliminate lower four bits
64018   40,242          JR Z,wait   ;Wait till change is larger
64020   115             LD (HL),E   ;Save first reading
64021   58,200,250 next LD A,(chan) ;Get number of channels
64024   254,2           CP 2        ;Two channels?
64026   56,38           JR C,one    ;No one channel only
64028   36              INC H       ;Next channel
64029   62,1            LD A,1      ;Select channel 1
64031   205,100,250     CALL get    ;Collect ADC reading
64034   119             LD (HL),A   ;Save input voltage
64035   58,200,250 next LD A,(chan) ;Get number of channels
64038   254,3           CP 3        ;Three channels?
64040   56,23           JR C,two    ;No two channels only
64042   36              INC H       ;Next channel
64043   62,2            LD A,2      ;Select channel 2
64045   205,100,250     CALL get    ;Collect ADC reading
64048   119             LD (HL),A   ;Save input voltage
64049   58,200,250 next LD A,(chan) ;Get number of channels
```

```
64052  254,4              CP 4          ;Four channels?
64054  56,38              JR C,three    ;No three channels only
64056  36                 INC H         ;Next channel
64057  62,3               LD A,3        ;Select channel 3
64059  205,100,250        CALL get      ;Collect ADC reading
64062  119                LD (HL),A     ;Save input voltage
64063  37                 DEC H
64064  37           three DEC H
64065  37             two DEC H
64066  62,127       one   LD A,127      ;Check keyboard
64068  219,254            IN A,(254)
64070  246,224            OR 224
64072  254,255            CP 255
64074  32,22               JR NZ,finish
64076  44                 INC L         ;Next set of readings
64077  40,19               JR Z,finish   ;Finish if all done
64079  58,201,250         LD A,(count)  ;Delay between readings
64082  40,6                JR Z,dlydn
64084  71                 LD B,A
64085  205,110,250  dly    Call delay    ;Delay for 100 microseconds
64088  16,251             DJNZ dly
64090  62,0        dlydn   LD A,0        ;Take next set of readings
64092  205,100,250         CALL get
64095  119                LD (HL),A
64096  24,179              JR next       ;Do next channel
64098  251          finish EI
64099  201                RET
64100  211,31       get    OUT (31),A
64102  205,110,250         CALL delay
64105  219,31              IN A,(31)
64107  201                RET
64108  0                  NOP
64109  0                  NOP
64110  62,50        delay  LD A,50
64112  61           dly2   DEC A
64113  32,253              JR NZ,dlys
64115  201                RET
```

The problems over timing led to the development of a faster ADC based upon the ZN448 device described in Chapter 5. This has its own inbuilt clock, so the erratic time-keeping of the Spectrum is eliminated. It is also much faster. A single delay loop provides data acquisition rates between 1000 and 50 000 readings per second. This is ideal for fast transient phenomena such as the light output from a flashgun.

The ZN448 normally has to be triggered to start a conversion. As used in the Griffin AD-pack, this is done automatically every ten microseconds. In the program FAST ADC, the data is read with IN A,(127). To achieve the maximum acquisition rate, the INI instruction is used. This reads the data from the I/O address contained in the C register, stores it in the location pointed at by the HL register pair, increments this pointer and decrements the B register, which acts as a counter – all this in fifteen T-states!

FAST ADC

```
64000  243                DI
65001  33,6,251           LD HL,0FB06   ;Set pointer to stores
64004  58,192,250         LD A,(count)  ;Get delay time
64007  95                 LD E,A        ;Save it
64008  6,250              LD B,250      ;250 readings in counter
64010  14,127             LD C,127      ;Set C to ADC read address
64012  219,127            IN A,(127)    ;Read ADC
64014  87                 LD D,A        ;Save reading
64015  219,127 nxtry      IN A,(127)    ;Read ADC again
64017  119                LD (HL),A     ;Save new reading
64018  146                SUB D         ;Compare with previous reading
64019  48,1               JR NC,pos     ;Positive result
64021  47                 CPL           ;Negative – so invert
64022  230,192 pos        AND 192       ;Eliminate lower six bits
64024  40,245             JR Z,nxtry    ;Wait till change is larger
64026  44                 INC L         ;Point to next position
64027  83           next  LD D,E        ;Get delay time
64028  21           dly   DEC D         ;Delay between readings
64029  32,253             JR NZ,dly
64031  237,162            INI           ;Take next reading, etc.
64033  32,248             JR NZ,next    ;Take next reading
64035  251                EI
64036  201                RET
```

It is almost impossible to make up a single data acquisition program to cover all desirable types of data. The advantage in developing your own routines is that they can then be made to fit any requirements. It is hoped that the discussions in this chapter will allow you to do this and thus make your microcomputer even more cost effective.

9 Dedicated systems

> 'I see you're admiring my little box,' the knight said in a friendly tone. 'It's my own invention.'
> (Lewis Carroll, *Through the Looking Glass)*

Permanent programs

Most microprocessors spend their time doing one set of tasks only. It is only the few that find their way into personal or school microcomputers, that are given different tasks from day to day at the whim of the programmer. The microprocessor inside a calculator has been preprogrammed to carry out calculations only. It will not be asked to play tunes or measure time intervals or temperatures etc. The microprocessor in a supermarket checkout will not be asked to play space invaders as well. The microprocessors in these systems are said to be dedicated to their one function. The programs that run these dedicated systems are usually frozen in ROM, because there is no need to change them once they have been written and debugged.

ROM is produced by a silicon chip manufacturer exactly as requested by the purchaser. The program is placed in the ROM by a process called **mask programming**. An individual ROM may contain 32 000 or more bytes, which is 256 000 different bits. Each bit of every byte in the ROM is initially switched on. Then, by a photographic process, each individual bit is marked, either as one to be left on, or one to be switched off. The final process then permanently switches each bit off, or leaves it on, according to the information printed on it. Once the program has been produced in this way, it is not possible to alter it later. The program remains in the ROM even if the power supply to the equipment is later switched off. When this ROM is coupled to a microprocessor, the latter will only carry out the program in the ROM.

The making of the masks for a ROM is a very expensive business and it is not done unless several thousand such ROMs are required. In the development stage, therefore, before the bugs have been ironed out, a different form of ROM is used, called programmable ROM. One version of this is especially useful; it is called **EPROM** or **electrically programmable ROM**. This allows programs to be burned in, just as with ROM, but it is also possible to erase this program and burn in a different one, if the first is found to contain bugs. The equipment needed to burn a program into an EPROM is not too expensive (fifty pounds or so for an add-on unit to the Spectrum), but it is probably not worth the average user getting such equipment. Local polytechnics and FE colleges usually possess it and are willing to let visitors make use of it under supervision.

EPROM enables the programmer to store machine-code routines permanently in his or her microcomputer, which can then be called from BASIC in the usual way (RANDOMIZE USR *nnnnn*). The exact value for 'nnnnn' depends upon where in the memory space the EPROM is placed. The most convenient EPROM is the 2516 (also known as the single-rail 2716), which can hold 2048 eight-bit bytes. A larger EPROM is the 2532, which can hold 4096 bytes.

Because the 48K Spectrum has so much memory, it has none to spare for an EPROM, although there are ways of overcoming this. The 16K Spectrum, however, has plenty of space for such EPROMS, addresses from 8000H to FFFFH are generally free. Thus EPROMs could be connected to the Spectrum and the routines contained in them could be accessed via machine code. Several commercial concerns make such add-ons for the Spectrum and also give details of how to make use of them (see Suppliers).

For physics teachers an obvious resident program for an EPROM is the timing routine discussed in the last chapter. To get such a routine into the EPROM, the hexadecimal code is usually typed into an EPROM burner and checked. Then a freshly erased EPROM is placed in one socket and the burn commences. Each binary code in turn is sent to its correct address and stored there by sending a voltage pulse along the program line. It takes one or two minutes for this process, after which the EPROM can be installed in the microcomputer. An EPROM can be erased again (for example, if a mistake has been made or if a better version has been developed) by exposing it to the correct dose of ultra-violet radiation. Any establishment with an EPROM burner will probably possess such a UV eraser too.

If this sounds a little complicated, then there is **EAROM (electrically alterable ROM)**. This too can be programmed and retains its program after the power has been switched off. Its program can be changed later, without having to erase the whole program first, since each memory location can be changed independently. EAROM is unfortunately much more expensive than EPROM, but does not need special equipment to program it. It is simply placed into an extension socket and treated like ordinary RAM, only it retains its program after the machine is switched off.

RAM has such tiny power requirements that a suitable battery can maintain a program in it for years after it has been programmed. Some RAM units therefore have a built-in battery to retain a program after the main power has been switched off. Here too, it is not necessary to buy special burner and eraser equipment, since the device behaves like any other RAM from the point of view of the microcomputer.

A stand-alone system

The above system is still just a microcomputer with some special resident routines. It would be possible, however, to buy a Spectrum, add an EPROM and use it purely as a dedicated system. Fundamentally this is what has happened to

some older microcomputers; for example my PET is now used exclusively for wordprocessing. However, the manufacturers of, say, video games, are not going to do it like this. For them most of the microcomputer, such as the BASIC interpreter and the keyboard are unnecessary. Their procedure is to design each system specially, using only those components necessary for the task required. This is also a technique which we can use too.

A dedicated system will need some means of collecting data and giving information back to the user. In a microcomputer this is the typewriter keyboard and video display. In a control system this could be a sensor and a few switches for input and an electromagnetic relay as an output (for example, in a system to open the garage doors automatically upon the arrival of a motor car). In both cases, however, some sort of input/output device will be needed and the techniques discussed in Chapter 4 are relevant in this context.

The system will also need a microprocessor and some RAM for storing variable data. The amount of RAM depends upon the system; a garage doors system may only need a few bytes, whereas a programmable electronic organ may need thousands. Finally the dedicated system will need its program stored in ROM or EPROM. Small dedicated systems can be developed around multi-purpose chips, such as the RRIOT (which contains ROM, RAM, input and output lines and a timer). The user's program is burned into the ROM when it is manufactured. This is combined with a microprocessor to give a two-chip system. The ultimate is a single chip containing all the RAM, ROM and I/O and the microprocessor as well. Such a chip is called a **single-chip microcomputer**.

These reductions in chip count give an obvious saving in cost, since the interconnections between the different parts of the system have already been made. All that is needed is a small printed circuit board to take the remaining components and a socket for the single-chip microcomputer and the complete system is ready.

This brings us right up to the present in microelectronic technology, since it is the use of such dedicated systems that is so profoundly affecting our lives. They are found in washing machines, sewing machines, knitting machines, motor cars, supermarket checkout points, video games and electric train sets. They control robots in factories, word processing equipment in the office and automatic stock delivery and despatch in the warehouse. What else they will do in the future is speculation, but I think it is safe to bet that those who can understand and program microelectronic systems are more likely to be employed than those who cannot.

This book was originally intended to be a part of a complete introduction to microelectronics, beginning with transistors and ending with complete single-chip microcomputers. This proved to be too ambitious and the emphasis was thus changed to using an existing microcomputer (in this case the Spectrum) to do most of the tasks that a microprocessor normally does. I do want to complete the picture, however, and to encourage some of you to build your own dedicated systems.

The first problem with any dedicated system is the number of connections needed. By the time an EPROM., an I/O device, RAM, address decoders and a microprocessor are connected together, the resulting forest of wires is quite alarming. A **printed circuit board (PCB)** is a much better proposition. It isn't difficult to make a PCB but, by the time the above devices have been added, the cost is well over fifty pounds. So much of the Spectrum is concerned with graphics, that it probably isn't sensible to use one of them in a dedicated system either, but this does not apply to the ZX81. The basic ZX81 now costs less than thirty pounds, so it seems far more sensible to use a ZX81, an I/O device and an EPROM instead. There are hundreds of simple systems that could be produced with this arrangement, although some expertise in machine-code programming and a good knopwledge of the Z80 are needed before such a project is tackled. The input to this system would be the normal keyboard, but there is probably little point in using a TV set for output. It is perfectly possible to connect a seven-segment display (like those in a calculator) to the I-pack and use that for feedback to the user. Different characters are displayed by controlling each segment of each digit independently. Calculator-style displays also have the advantage of being inexpensive. By alternating between upper and lower case letters, it is even possible to display enough letters of the alphabet to present such words as 'rEAdy', 'yES' and 'no'.

At first sight it looks as if the requirements of the eight-digit, seven-segment display are impossible to meet. The number of segments is actually eight, because of the decimal point, and this might imply that 8×8 or 64 lines are needed to drive all eight digits. In practice, only one digit is displayed at any one time and only the segments needed for that particular digit are switched on. This technique is called **multiplexing** and is the standard procedure for this type of display. (If the number 8888888888 is entered into a pocket calculator, which is then waved about, it becomes obvious that this is happening.) With this method we can use the same eight lines to run the segments for all of the digits and we only need eight more lines, one for each digit.

To provide temporary storage for input data and to allow a working space for the operating system some RAM is necessary. Rarely need this be more than the normal 1K, but a 16K RAM pack can just as easily be added to a dedicated system. The program needed to run the system will need routines for handling the display, for sending and receiving the data and for processing it and for interacting with BASIC. This might sound a great deal, but in fact machine code is very sparing in its use of memory, so the two kilobytes of a single 2716 EPROM are more than enough.

Originally I made several stand-alone systems based upon a 6502 system (see *The BBC microcomputer in science teaching*). I actually wrote the programs for these systems using an Apple II microcomputer. The address and data lines from an Apple connector socket were connected to a VIA and the outputs from this went to the keyboard, display and I/O lines of my system. The programs were written in the Apple's memory and when they had been debugged, the

hexadecimal codes were copied out by hand and changed to fit the dedicated system. They were then taken to Glasgow University and typed into their EPROM burner. The EPROM was then plugged into the final system. To my utter astonishment it worked first time! One of the systems I made using the Apple was a simple microprocessor tutor, now marketed by Griffin and George as the Mini-microprocessor. I have been hooked on dedicated systems ever since.

The method proposed for using the ZX81 is, however, very much easier to handle. Currently under development is a complete dedicated laboratory instrument. When finished, this will allow any measurements to be made through the I-pack interface and its peripherals (for some details, see *The ZX81 in science teaching*). More importantly, since such an instrument is based upon mass-marketed equipment, it is very much cheaper than a purpose built 'laboratory aid' (and more 'versatile' too!). I hope that others will follow this path and discover for themselves the joys of building dedicated systems.

Suppliers

At the time of writing several commercial interfaces are available for the ZX Spectrum microcomputer. Soon there will be an overwhelming supply. Unfortunately, the cost of an interface is no guide to its facilities. So what criteria should be used in selecting one for use in the science laboratory?

A useful laboratory interface would have a fast analogue converter, perferably with up to four channels. A data acquisition rate of at least 10 000 readings a second is needed for measuring transients. The inputs may be a.c. or d.c. and it should be possible to alter the sensitivity and the bias, so that, for example, the voltage across a capacitor could be measured as it discharged through an inductor. A useful facility would allow the alteration of the threshold level at which the measurements begin to be taken.

The interface ought also to provide a digital to analogue converter with sufficient power output to drive current through an LCR circuit or a lamp. Even better would be an a.c. output with controllable frequency as described in Chapter 5.

The digital side should have relay outputs for driving motors and heaters and TTL outputs for driving other integrated circuits. Inputs that can be driven directly from a switch or a photocell are also desirable. The minimum number of outputs is four and at least two inputs are needed. Eight of each is very nice if the expense can be justified. Ideally the interface should have a 5V output too, with sufficient current capability to drive LEDs, etc.

There is no commercially available interface that yet fulfils all of these requirements. Efforts are being made by Griffin and George to come up to the above specification and the Interspec (called the I-pack by Griffin and George) is the first of a series of stackable units, which, when put together, will meet most needs. A DAC-pack (digital to analogue converter) and an AD-pack (very fast analogue to digital converter) are already available and a conversion unit (Expand-pack) is planned, which will allow the measurement of the complete range of voltages and currents encountered in the laboratory.

For further details contact:

Griffin and George Ltd
Ealing Road
Wembley HA0 1HJ

Printer interface

The ZX printer is not good enough for readable listings. The listings in the Appendix were obtained with an Epson MX80 printer connected to the Spectrum via a Centronics interface, available from:

Kempston Microelectronics
180a Bedford Road
Kempston
Bedford MK42 8BL

Electronic components

Chapters 4 and 5 describe several interfacing circuits that can easily be made in school. Components are the biggest problem, but those mentioned are normally available from one of the following suppliers:

Farnell Electronic Components Ltd.
Canal Road
Leeds LS2 2TU

RS Components Ltd.
13–17 Epworth Street
London EC2P 2HA

Verospeed Components
Stansted Road
Boyatt Wood
Eastleigh
Hants SO5 4ZY

Bibliography

Introductory books are not included

BASIC

L. Poole and M. Borchers, *Some Common BASIC Programs*, Osborne/McGraw Hill, 1979

J. S. Coan, *Advanced BASIC*, Hayden Book Co. Inc., 1977

J. S. Gilder, *BASIC Computer Programs in Science and Engineering*, Hayden Book Co. Inc., 1980

R. E. Myers, *Microcomputer Graphics with Apple II examples*, Addison Wesley

Assembly language

L. A. Levanthal, *Z80 Assembly language programming*, Osborne/McGraw Hill

I. Logan, *Understanding your Spectrum*, Melbourne House, 1982

I. Logan and F. O'Hara, *The Complete Spectrum ROM disassembly*, Melbourne House, 1983

I. Sinclair, *Introducing Spectrum machine code*, Granada Publishing, 1983

W. Tang (ed.), *Spectrum machine language for the absolute beginner*, Melbourne House, 1982

Electronics

Malmstadt, Enk and Crouch, *Electronics and Instrumentation*, Benjamin/Cummings Pub. Co. Inc., California, 1981

Education

C. Doerr, *Microcomputers and the 3Rs*, Hayden Book Co. Inc., 1979

Program listings

Program 1 DIGITAL ELECTRONICS

```
 1 REM DIGITAL ELECTRONICS
 2 REM by R.A.Sparkes
 3 REM begun Sept. 12th. 1983
 4 PRINT AT 10,10;"Please wait"
 5 GO SUB 3000: REM define graphics
 7 CLS : PRINT AT 0,6;"DIGITAL ELECTRONICS"
 8 PRINT AT 2,0;"Choose a topic for study by"
 9 PRINT AT 3,0;"pressing one of these numbers."
10 PRINT AT 5,0;"1   Two input logic gates"
11 PRINT AT 7,0;"2   Three input logic gates"
12 PRINT AT 9,0;"3   The 2-to-4 decoder"
13 PRINT AT 11,0;"4  User-defined logic gates"
14 PRINT AT 13,0;"5  The bistable"
15 PRINT AT 15,0;"6  The binary counter"
16 PRINT AT 17,0;"7  The shift register"
17 PRINT AT 19,0;"8  The astable multivibrator"
18 PRINT AT 21,0;"9  The monostable multivibrator"
19 REM Initialize variables
20 LET s$="                                    "
21 LET a=0: LET b=0: LET c=0: REM input data from Interspec
22 LET olda=5: LET oldb=5: LET oldc=5: LET edge=0: LET cp=0: REM input status
23 LET p=11: REM pointer to truth table
24 LET inputs=63: LET outputs=63
25 DIM e$(4,30)
26 LET a$=INKEY$: IF a$<"1" OR a$>"9" THEN  GO TO 26
27 LET choice=VAL a$: REM menu value of choice
28 GO TO 100*choice
100 IF INKEY$<>"" THEN  GO TO 100
101 CLS : PRINT AT 0,7;"LOGIC GATES"
102 PRINT AT 2,1;"Select desired function by "
103 PRINT AT 4,1;"pressing one of these numbers"
104 PRINT AT 6,5;"1       AND"
105 PRINT AT 8,5;"2       OR"
106 PRINT AT 10,5;"3       NOT A"
107 PRINT AT 12,5;"4       EXCLUSIVE-OR"
108 PRINT AT 14,5;"5       EQUIVALENCE"
109 PRINT AT 16,5;"6       NAND"
110 PRINT AT 18,5;"7       NOR"
111 LET s$=INKEY$
112 IF s$<"1" OR s$>"7" THEN  GO TO 111
113 LET s=VAL s$: REM menu value of function
114 LET olda=5: REM reset old values
120 REM display functions and terminals
121 CLS
122 GO SUB 1000: REM display box
123 LET l=s*10+1090
124 GO SUB l
125 PRINT AT 4,12;p$
126 PRINT AT 4,23;"( )output"
127 PRINT AT 0,11;"LOGIC GATES"
128 GO SUB 1200: REM draw truth table
```



```
129 REM display inputs
130 IF s=3 THEN  GO SUB 5000
131 IF s<>3 THEN  GO SUB 5100
132 PRINT AT 21,0;"Press C to change or E to end."
150 LET a$=INKEY$
151 IF a$="c" OR a$="C" THEN  GO TO 100
152 IF a$="e" OR a$="E" THEN  GO TO 6
153 GO SUB 9000: REM Get input status
154 IF same THEN  GO TO 150
160 REM change input values on screen
161 IF s=3 THEN  PRINT AT 4,4;a: GO TO 170
162 PRINT AT 5,4;b
163 PRINT AT 3,4;a
170 REM display line of truth table
171 PRINT AT p,2;"      "
172 PRINT AT p,25;"      "
173 PRINT AT p,19;" "
174 LET p=12+2*b+4*a
175 PRINT AT p,2;"      "
176 PRINT AT p,25;"      "
180 REM calculate output data
181 LET l=s*100+2000: GO SUB 1
182 REM Send output to logic board
183 PRINT AT 4,24;vo
184 PRINT AT p,19;vo
185 OUT outputs,B*vo
186 GO TO 150
200 IF INKEY$<>"" THEN  GO TO 200
201 CLS : PRINT AT 0,2;"THREE-INPUT LOGIC GATES"
202 PRINT AT 2,1;"Select desired function by "
203 PRINT AT 4,1;"pressing one of these numbers"
204 PRINT AT 6,5;"1       AND"
205 PRINT AT 8,5;"2       OR"
209 PRINT AT 10,5;"3       NAND"
210 PRINT AT 12,5;"4       NOR"
211 LET s$=INKEY$
212 IF s$<"1" OR s$>"4" THEN  GO TO 211
213 LET s=VAL s$: REM menu value of function
218 LET olda=5: REM reset old values
220 REM display functions and terminals
221 CLS
222 GO SUB 1000: REM display box
223 LET l=1100*(s=1)+1110*(s=2)+1150*(s=3)+1160*(s=4)
224 GO SUB 1
225 PRINT AT 4,12;p$
226 PRINT AT 4,23;"( )output"
227 PRINT AT 0,11;"LOGIC GATES"
228 GO SUB 1400: REM draw three-input truth table
229 REM display inputs
230 GO SUB 5200: REM Add three inputs
232 PRINT AT 21,0;"Press C to change or E to end."
250 LET a$=INKEY$
251 IF a$="c" OR a$="C" THEN  GO TO 200
252 IF a$="e" OR a$="E" THEN  GO TO 6
253 GO SUB 9000: REM Get input status
254 IF same THEN  GO TO 250
260 PRINT AT 2,4;a
261 PRINT AT 4,4;b
262 PRINT AT 6,4;c
270 REM display line of truth table
271 PRINT AT p,1;"      "
```

```
272 PRINT AT p,26;"     "
273 PRINT AT p,19;" "
274 LET p=11+c+2*b+4*a
275 PRINT AT p,1;"     "
276 PRINT AT p,26;"     "
280 REM calculate output data
281 LET l=s*50+2750: GO SUB 1
282 REM Send output to logic board
283 PRINT AT 4,24;vo
284 PRINT AT p,19;vo
285 OUT outputs,8*vo
286 GO TO 250
300 IF INKEY$<>"" THEN  GO TO 300
301 CLS
302 PRINT AT 0,10;"2-TO-4 DECODER"
303 GO SUB 1070: REM display box
304 PRINT AT 4,0;"address"
305 PRINT AT 3,7;"A( )"
306 PRINT AT 5,7;"B( )"
307 PRINT AT 7,4;"data( )"
308 GO SUB 1600: REM draw data chart
309 PRINT AT 21,0;"Press E to end."
310 LET olda=5: LET l=2: LET p=0: REM initialize variables
311 PLOT 108,116: DRAW 42,p: PLOT 108,115: DRAW 42,p
312 LET a$=INKEY$: IF a$="e" OR a$="E" THEN  GO TO 6
313 GO SUB 9000: REM Get input status
314 IF same THEN  GO TO 312
319 PRINT AT 1,20;" "
320 PLOT 108,116: DRAW  OVER 1;42,p: PLOT 108,115: DRAW  OVER 1;42,p
321 PRINT AT 3,9;a
322 PRINT AT 5,9;b
323 PRINT AT 7,9;c
324 LET l=8-4*a-2*b
325 PRINT AT 1,20;c
326 LET p=55-1*8
327 PLOT 108,116: DRAW 42,p
328 PLOT 108,115: DRAW 42,p
330 REM change outputs
331 LET vo=(2^(2*a+b)) AND c  ,
332 OUT outputs,vo
333 GO TO 312
400 IF INKEY$<>"" THEN  GO TO 400
410 CLS : PRINT AT 0,7;"USER-DEFINED LOGIC"
411 PRINT AT 3,0;"This program allows you to make"
412 PRINT AT 5,0;"your own logic gates."
413 PRINT AT 7,0;"You may only use the variables"
414 PRINT AT 9,0;"A, B, C (or a, b, c), 0 or 1"
415 PRINT AT 11,0;"and say whether W, X, Y or Z"
416 PRINT AT 13,0;"is to produce the output."
417 PRINT AT 15,0;"You may only use the Boolean"
418 PRINT AT 17,0;"operators NOT, AND and OR and"
419 PRINT AT 19,0;"brackets ( ) in your expression."
420 PRINT AT 21,0;"Press SPACE to continue."
421 IF INKEY$<>" " THEN  GO TO 421
422 CLS : PRINT AT 0,7;"USER-DEFINED LOGIC"
423 PRINT AT 3,0;"Here are a few examples:"
424 PRINT AT 6,0;"Output:    z (or Z)"
425 PRINT AT 8,0;"Expression:  NOT(A AND b) OR 1"
426 PRINT AT 11,0;"Output:    x (or X)"
427 PRINT AT 13,0;"Expression:  0 OR (a AND NOT B)"
428 PRINT AT 16,0;"Operators NOT, AND and OR MUST ": PLOT 208,39: DRAW  OVER 1;
30,0
```



```
429 PRINT AT 18,0;"be typed as SINGLE KEYSTROKES"
430 PRINT AT 21,0;"Press  C  to continue."
431 IF INKEY$<>"c" AND INKEY$<>"C" THEN  GO TO 431
434 REM initialize boolean expressions
435 LET e$(1)="0"
436 LET e$(2)="0"
437 LET e$(3)="0"
438 LET e$(4)="0"
440 CLS : PRINT AT 0,7;"USER-DEFINED LOGIC"
441 GO SUB 1070
442 PRINT AT 3,7;"A( )"
443 PRINT AT 5,7;"B( )"
444 PRINT AT 7,7;"C( )"
446 LET w=0: LET x=0: LET y=0: LET z=0
449 PRINT AT 20,0;s$;
450 INPUT "Output w, x, y or z? (s to stop)";o$
451 IF CODE o$>90 THEN  LET o$=CHR$ ((CODE o$)-32)
452 IF o$="S" THEN  GO TO 6
453 IF o$>"Z" OR o$<"W" THEN  GO TO 450
455 INPUT "Boolean expression?                ";f$
456 PRINT AT 12+2*((CODE o$)-87),0;s$: PRINT s$
457 PRINT AT 12+2*((CODE o$)-87),0;o$;" = ";f$
460 LET a=0: LET b=0: LET c=0
461 LET olda=5: REM reset status
463 REM evaluate expression
464 IF o$="W" THEN  LET e$(1)=f$
465 IF o$="X" THEN  LET e$(2)=f$
466 IF o$="Y" THEN  LET e$(3)=f$
467 IF o$="Z" THEN  LET e$(4)=f$
470 PRINT AT 20,0;"Press  N  for a new expression"
471 PRINT AT 21,0;"or  E  to stop."
472 LET a$=INKEY$
473 IF a$="E" OR a$="e" THEN  GO TO 6
474 IF a$="N" OR a$="n" THEN  GO TO 449
475 GO SUB 9000: IF same THEN  GO TO 472
476 LET w=VAL (e$(1))
477 LET x=VAL (e$(2))
478 LET y=VAL (e$(3))
479 LET z=VAL (e$(4))
480 REM change screen logic levels
481 PRINT AT 3,9;a
482 PRINT AT 5,9;b
483 PRINT AT 7,9;c
484 PRINT AT 2,20;z
485 PRINT AT 4,20;y
486 PRINT AT 6,20;x
487 PRINT AT 8,20;w
490 REM change outputs
491 LET vo=8*z+4*y+2*x+w
492 OUT outputs,vo
495 GO TO 472
500 IF INKEY$<>"" THEN  GO TO 500
501 CLS : PRINT AT 0,8;"THE BISTABLE"
502 PRINT AT 3,0;"There are three main types"
503 PRINT AT 5,0;"of bistable, as follows:"
504 PRINT AT 7,0;"1      The SET-RESET bistable"
505 PRINT AT 9,0;"2      The D-input bistable"
506 PRINT AT 11,0;"3     The J-K bistable"
507 PRINT AT 14,0;"Press 1, 2 or 3 to select one."
508 LET s$=INKEY$
509 IF s$<"1" OR s$>"3" THEN  GO TO 508
510 CLS : GO SUB 1090: REM draw bistable
```

```
511 PRINT AT 2,27;"Q"
512 PRINT AT 8,25;"NOT-Q"
513 LET olda=5: REM reset status flag
514 PRINT AT 21,0;"Press C to change or E to finish"
519 GO TO 510+10*VAL s$
520 REM SET-RESET bistable
521 PRINT AT 3,3;"SET  ( )": PRINT AT 7,3;"RESET( )"
522 GO SUB 1700: REM Draw data chart
523 LET k$=INKEY$: IF k$="e" OR k$="E" THEN  GO TO 6
524 IF k$="c" OR k$="C" THEN  GO TO 500
525 GO SUB 9000: IF same THEN  GO TO 523
526 IF (a AND c) OR (NOT a AND NOT c) THEN  GO SUB 575: GO TO 523
527 LET v=(c AND NOT a) OR NOT (a AND NOT c)
528 GO SUB 570
529 GO TO 523
530 REM D-input bistable
531 PRINT AT 3,3;" D   ( )": PRINT AT 5,3;"CLK  ( )"
532 GO SUB 1900: REM draw data chart
533 LET v=0: GO SUB 570
534 LET k$=INKEY$: IF k$="e" OR k$="E" THEN  GO TO 6
535 IF k$="c" OR k$="C" THEN  GO TO 500
536 GO SUB 9000: IF same  THEN  GO TO 534
537 IF edge=0 THEN  GO SUB 575: GO TO 534
538 LET v=a: GO SUB 570
539 GO TO 534
540 REM J-K bistable
541 PRINT AT 5,3;" CLK ( )": PRINT AT 3,3;" J   ( )"
542 PRINT AT 7,3;" K   ( )"
543 GO SUB 1800: REM draw data chart
544 LET v=0: GO SUB 570
545 LET k$=INKEY$: IF k$="e" OR k$="E" THEN  GO TO 6
546 IF k$="c" OR k$="C" THEN  GO TO 500
547 GO SUB 9000: IF same  THEN  GO TO 545
548 IF cp=0 THEN  GO SUB 575: GO TO 545
549 IF NOT a AND NOT c THEN  GO SUB 575: GO TO 545
550 IF NOT a AND c THEN  LET v=0: GO SUB 570: GO TO 545
551 IF a AND NOT c THEN  LET v=1: GO SUB 570: GO TO 545
552 IF a AND c THEN  LET v=NOT v: GO SUB 570: GO TO 545
570 REM Change outputs on logic board and screen
571 LET vo=8*v+NOT v
572 OUT outputs,vo
573 PRINT AT 2,20;v
574 PRINT AT 8,20;NOT v
575 PRINT AT 3,9;a
576 PRINT AT 5,9;b
577 PRINT AT 7,9;c
580 RETURN
600 IF INKEY$<>"" THEN  GO TO 600
601 REM Binary counter
610 CLS : PRINT AT 0,10;"BINARY COUNTER"
611 GO SUB 1070: REM display logic board
612 PRINT AT 3,3;"INPUT( )"
613 PRINT AT 5,1;"RESET1 ( )"
614 PRINT AT 7,1;"RESET2 ( )"
615 PRINT AT 11,0;"Pulses are counted in binary"
616 PRINT AT 13,0;"when applied to the INPUT."
617 PRINT AT 15,0;"At least ONE of the RESET inputs"
618 PRINT AT 17,0;"must be LOW to this to happen."
619 PRINT AT 21,0;"Press  E  to finish"
620 LET prev=0: LET vo=4: LET olda=5
630 IF INKEY$="e" OR INKEY$="E" THEN  GO TO 6
631 GO SUB 9000: REM Get input status
```



```
632 IF same THEN  GO TO 630
633 IF b AND c THEN  LET vo=0: LET prev=0: GO TO 640
634 IF prev=1 AND a=0 THEN  LET prev=0: LET vo=vo+1
635 IF a=1 THEN  LET prev=1
636 IF vo=16 THEN  LET vo=0
646 GO SUB 650
647 GO TO 630
650 LET q=vo: LET z=0: LET y=0: LET x=0: LET w=0
651 IF q>7 THEN  LET q=q-8: LET z=1
652 IF q>3 THEN  LET q=q-4: LET y=1
653 IF q>1 THEN  LET q=q-2: LET x=1
654 LET w=q
655 PRINT AT 2,20;z
656 PRINT AT 4,20;y
657 PRINT AT 6,20;x
658 PRINT AT 8,20;w
659 OUT outputs,vo
660 PRINT AT 3,9;a
661 PRINT AT 5,9;b
662 PRINT AT 7,9;c
666 RETURN
690 STOP
700 IF INKEY$<>"" THEN  GO TO 700
701 REM Shift register
710 CLS : PRINT AT 0,10;"SHIFT REGISTER"
711 GO SUB 1070: REM display logic board
712 PRINT AT 5,3;" CLK ( )"
713 PRINT AT 7,3;"INPUT( )"
714 PRINT AT 11,0;"Each clock pulse shifts the data"
715 PRINT AT 13,0;"along by one bit."
716 PRINT AT 15,0;"The data is entered through the"
717 PRINT AT 17,0;"INPUT terminal."
719 PRINT AT 21,0;"Press  E  to finish"
720 LET vo=4: LET olda=5
725 GO SUB 650
730 IF INKEY$="e" OR INKEY$="E" THEN  GO TO 6
731 GO SUB 9000: REM Get input status
732 IF same THEN  GO TO 730
733 IF cp=0 THEN  GO SUB 660: GO TO 730
735 LET vo=vo+vo+c
736 IF vo>15 THEN  LET vo=vo-16
740 GO SUB 650
750 GO TO 730
800 IF INKEY$<>"" THEN  GO TO 800
810 CLS : PRINT AT 0,7;"ASTABLE MULTIVIBRATOR"
811 GO SUB 1090: REM display logic diagram
812 PLOT 0,153: DRAW 56,0: PLOT 88,64: DRAW 56,0
813 PRINT AT 3,0;"INHIBIT ( )"
814 PRINT AT 21,0;"Press  E  to finish"
815 PRINT AT 12,0;"The astable is free-running"
816 PRINT AT 14,0;"as long as INHIBIT is HIGH."
817 PRINT AT 16,0;"If this terminal goes LOW,"
818 PRINT AT 18,0;"the oscillations stop."
819 IF INKEY$="e" OR INKEY$="E" THEN  GO TO 6
820 OUT outputs,8
821 PRINT AT 2,20;"1"
822 PRINT AT 8,20;"0"
823 PAUSE 20
825 GO SUB 9000: REM Get input status
826 PRINT AT 3,9;a
830 OUT outputs,1
831 PRINT AT 2,20;"0"
```

```
832 PRINT AT 8,20;"1"
833 PAUSE 20
835 GO SUB 9000: REM Get input status
836 PRINT AT 3,9;a
838 IF INKEY$="e" OR INKEY$="E" THEN  GO TO 6
839 IF a=0 THEN  GO TO 830
840 GO TO 815
900 IF INKEY$<>"" THEN  GO TO 900
910 CLS : PRINT AT 0,5;"MONOSTABLE MULTIVIBRATOR"
911 GO SUB 1090: REM display logic diagram
912 PRINT AT 7,1;"INPUT2 ( )"
913 PRINT AT 5,1;"INPUT1 ( )"
914 PRINT AT 21,0;"Press  E  to finish"
915 PRINT AT 11,0;"The monostable is normally"
916 PRINT AT 13,0;"in the reset state."
917 PRINT AT 15,0;"A LOW-HIGH change at INPUT1 or"
918 PRINT AT 17,0;"HIGH-LOW change at INPUT2 will"
919 PRINT AT 19,0;"create a single output pulse."
920 REM begin
921 OUT outputs,1
922 PRINT AT 2,20;"0"
923 PRINT AT 8,20;"1"
924 GO SUB 9000: REM Get input status
925 LET prev=c
926 PRINT AT 5,9;"b"
927 PRINT AT 7,9;"c"
930 IF INKEY$="e" OR INKEY$="E" THEN  GO TO 6
931 GO SUB 9000: REM Get input status
932 IF same THEN  GO TO 930
933 IF edge THEN  GO TO 950
934 IF prev=c THEN  GO TO 926
935 IF c=0 THEN  LET prev=0: GO TO 950
936 GO TO 925
950 OUT outputs,8
951 PRINT AT 2,20;"1"
952 PRINT AT 8,20;"0"
953 PAUSE 20
954 GO TO 920
1000 REM display box
1010 PRINT AT 2,11;"tnnnnnnnnnnp"
1020 PRINT AT 3,11;"p          p"
1030 PRINT AT 4,11;"p          laaaaa   "
1040 PRINT AT 5,11;"p          p"
1050 PRINT AT 6,11;"uoooooooooop"
1060 RETURN
1070 REM display logic board
1071 PRINT AT 1,11;"tnnnnnnnnnnnp"
1072 PRINT AT 2,11;"p        ( )Z p"
1073 PRINT AT 3,11;"pa           p"
1074 PRINT AT 4,11;"p        ( )Y p"
1075 PRINT AT 5,11;"pa           p"
1076 PRINT AT 6,11;"p        ( )X p"
1077 PRINT AT 7,11;"pa           p"
1078 PRINT AT 8,11;"p        ( )W p"
1079 PRINT AT 9,11;"uooooooooooop"
1080 RETURN
1090 REM bistable
1091 PRINT AT 1,11;"tnnnnnnnnnnnp"
1092 PRINT AT 2,11;"p        ( )  p"
1093 PRINT AT 3,11;"pa            p"
1094 PRINT AT 4,11;"p             p"
1095 PRINT AT 5,11;"pa            p"
```



```
1096 PRINT AT 6,11;"p             p"
1097 PRINT AT 7,11;"pa            p"
1098 PRINT AT 8,11;"p         ( )  p"
1099 PRINT AT 9,11;"uooooooooooop": RETURN
1100 LET p$="  AND ": RETURN
1110 LET p$="  OR  ": RETURN
1120 LET p$="  NOT ": RETURN
1130 LET p$="EX-OR ": RETURN
1140 LET p$="EQUIV.": RETURN
1150 LET p$=" NAND ": RETURN
1160 LET p$=" NOR  ": RETURN
1200 REM draw truth table
1210 PRINT AT 8,8;"tnnntnnntnnnnnnp"
1220 PRINT AT 9,8;"p A p B poutput p"
1230 PRINT AT 10,8;"uooouooouooooooop"
1240 PRINT AT 11,8;"p   p   p       p"
1250 PRINT AT 12,8;"p 0 p 0 p       p"
1260 PRINT AT 13,8;"p   p   p       p"
1270 PRINT AT 14,8;"p 0 p 1 p       p"
1280 PRINT AT 15,8;"p   p   p       p"
1290 PRINT AT 16,8;"p 1 p 0 p       p"
1300 PRINT AT 17,8;"p   p   p       p"
1310 PRINT AT 18,8;"p 1 p 1 p       p"
1320 PRINT AT 19,8;"uooouoooooooooop"
1350 RETURN
1400 REM draw 3-input truth table
1410 PRINT AT 8,6;"faaaiaaaiaaaiaaaaag"
1420 PRINT AT 9,6;"c A c B c C coutputc"
1430 PRINT AT 10,6;"jaaabaaabaaabaaaaak"
1440 PRINT AT 11,6;"c 0 c 0 c 0 c      c"
1450 PRINT AT 12,6;"c 0 c 0 c 1 c      c"
1460 PRINT AT 13,6;"c 0 c 1 c 0 c      c"
1470 PRINT AT 14,6;"c 0 c 1 c 1 c      c"
1480 PRINT AT 15,6;"c 1 c 0 c 0 c      c"
1490 PRINT AT 16,6;"c 1 c 0 c 1 c      c"
1500 PRINT AT 17,6;"c 1 c 1 c 0 c      c"
1510 PRINT AT 18,6;"c 1 c 1 c 1 c      c"
1520 PRINT AT 19,6;"daaahaaahaaahaaaaae"
1530 RETURN
1600 REM draw data table
1610 PRINT AT 11,3;"tnnntnnntnnnnnnrnnnnnnnp"
1620 PRINT AT 12,3;"p A p B paddressq output p"
1630 PRINT AT 13,3;"uooouooouoooooosooooooop"
1640 PRINT AT 14,3;"p   p   p        q       p"
1650 PRINT AT 15,3;"p 0 p 0 p  00    q    W  p"
1660 PRINT AT 16,3;"p 0 p 1 p  01    q    X  p"
1670 PRINT AT 17,3;"p 1 p 0 p  10    q    Y  p"
1680 PRINT AT 18,3;"p 1 p 1 p  11    q    Z  p"
1690 PRINT AT 19,3;"uooouooouoooooosooooooop"
1699 RETURN
1700 REM draw bistable data chart
1710 PRINT AT 11,3;"tnnntnnntnnnnnrnnnnnnnp"
1720 PRINT AT 12,3;"p S p R p  Q  q NOT-Q  p"
1730 PRINT AT 13,3;"uooouooouoooooosooooooop"
1740 PRINT AT 14,3;"p   p   p     q        p"
1750 PRINT AT 15,3;"p 0 p 0 p no change    p"
1760 PRINT AT 16,3;"p 1 p 0 p  0  q    1   p"
1770 PRINT AT 17,3;"p 0 p 1 p  1  q    0   p"
1780 PRINT AT 18,3;"p 1 p 1 p no change    p"
1790 PRINT AT 19,3;"uooouooouoooooosooooooop"
1799 RETURN
1800 REM draw J-K bistable data chart
```

```
1810 PRINT AT 11,3;"tnnnntnntnrnnnnrnnnnnnnp"
1820 PRINT AT 12,3;"pCLK pJ pKq  Q q NOT-Q p"
1830 PRINT AT 13,3;"uoooouoouosoooosooooooop"
1840 PRINT AT 14,3;"p    p  p q    q       p"
1850 PRINT AT 15,3;"p m  p0 p0q no change  p"
1860 PRINT AT 16,3;"p m  p0 p1q  0 q   1   p"
1870 PRINT AT 17,3;"p m  p1 p0q  1 q   0   p"
1880 PRINT AT 18,3;"p m  p1 p1q changeover p"
1890 PRINT AT 19,3;"uoooouoouosoooosooooooop"
1899 RETURN
1900 REM draw D-inputbistable data chart
1910 PRINT AT 11,3;"tnnnntnnntnnnnnrnnnnnnnp"
1920 PRINT AT 12,3;"pCLK p D p  Q  q NOT-Q  p"
1930 PRINT AT 13,3;"uoooouoooouooooosooooooop"
1940 PRINT AT 14,3;"p    p   p    q        p"
1950 PRINT AT 15,3;"p ^  p 0 p  0  q    1   p"
1960 PRINT AT 16,3;"p ^  p 1 p  1  q    0   p"
1970 PRINT AT 17,3;"p    p X p  no change   p"
1980 PRINT AT 18,3;"uoooouoooouooooosooooooop"
1990 RETURN
2000 REM Determine outputs for each function
2100 REM  AND function
2110 LET vo=a AND b
2150 RETURN
2200 REM OR function
2210 LET vo=a OR b ,
2250 RETURN
2300 REM NOT function
2310 LET vo=NOT a
2350 RETURN
2400 REM EXCLUSIVE-OR function
2410 LET vo=1
2420 IF a=b THEN  LET vo=0
2440 RETURN
2500 REM EQUIVALENCE function
2510 LET vo=0
2520 IF a=b THEN  LET vo=1
2540 RETURN
2600 REM NAND function
2610 LET vo=NOT (a AND b)
2660 RETURN
2700 REM NOR function
2710 LET vo=NOT (a OR b)
2760 RETURN
2800 REM 3-input AND function
2810 LET vo=a AND b AND c
2820 RETURN
2850 REM 3-input OR function
2860 LET vo=a OR b OR c
2870 RETURN
2900 REM 3-input NAND function
2910 LET vo=NOT (a AND b AND c)
2920 RETURN
2950 REM 3-input NOR function
2960 LET vo=NOT (a OR b OR c)
2970 RETURN
3000 REM line graphics
3010 FOR i=0 TO 20
3020  FOR j=0 TO 7
3030  READ row
3040 POKE USR CHR$ (i+144)+j,row
3050 NEXT j
```



```
3070 NEXT i
3080 RETURN
3100 DATA 0,0,0,255,255,0,0,0
3110 DATA 16,16,16,255,255,16,16,16
3120 DATA 16,16,16,16,16,16,16,16
3130 DATA 16,16,16,31,31,0,0,0
3140 DATA 16,16,16,240,240,0,0,0
3150 DATA 0,0,0,31,31,16,16,16
3160 DATA 0,0,0,240,240,16,16,16
3170 DATA 16,16,16,255,255,0,0,0
3180 DATA 0,0,0,255,255,16,16,16
3190 DATA 16,16,16,31,31,16,16,16
3200 DATA 16,16,16,240,240,16,16,16
3210 DATA 128,128,128,255,255,128,128,128
3220 DATA 0,0,60,36,36,36,36,231
3240 DATA 255,0,0,0,0,0,0,0
3250 DATA 0,0,0,0,0,0,0,255
3260 DATA 128,128,128,128,128,128,128,128
3270 DATA 1,1,1,1,1,1,1,1
3280 DATA 255,1,1,1,1,1,1,1
3290 DATA 1,1,1,1,1,1,1,255
3300 DATA 255,128,128,128,128,128,128,128
3310 DATA 128,128,128,128,128,128,128,255
5000 REM Append inputs
5010 REM One input
5020 PRINT AT 4,1;"A ( )aaaaa"
5030 RETURN
5100 REM  Two inputs
5110 PRINT AT 3,1;"A ( )aaaaa"
5120 PRINT AT 5,1;"B ( )aaaaa"
5130 RETURN
5200 REM  Three inputs
5210 PRINT AT 2,1;"A ( )aaaaa"
5220 PRINT AT 4,1;"B ( )aaaaa"
5230 PRINT AT 6,1;"C ( )aaaaa"
5240 RETURN
9000 REM get input data
9010 LET a=0: LET b=0: LET c=0
9020 PAUSE 1
9025 LET q=(IN inputs)
9030 IF q>127 THEN  LET q=q-128
9031 IF q>63 THEN  LET q=q-64
9032 IF q>31 THEN  LET q=q-32
9033 IF q>15 THEN  LET q=q-16
9034 IF q>7 THEN  LET q=q-8
9035 IF q>3 THEN  LET c=1: LET q=q-4
9036 IF q>1 THEN  LET b=1: LET q=q-2
9037 LET a=q
9038 IF olda=a AND oldb=b AND oldc=c THEN  LET same=1: RETURN
9040 LET cp=0
9050 IF edge=1 AND b=0 THEN  LET cp=1
9090 LET edge=0
9100 IF oldb=0 AND b=1 THEN  LET edge=1
9108 LET olda=a
9109 LET oldb=b
9110 LET oldc=c
9111 LET same=0
9112 RETURN
```

Program 2 LOGIC TEST

```
  1 REM LOGIC TEST
  2 REM  by R.A.Sparkes
  4 PRINT AT 10,10;"Please wait"
  5 GO SUB 3000: REM define graphics
 10 DIM l(10): REM number of options
 20 LET a=0: LET b=0: REM input data from Interspec
 30 FOR m=1 TO 10: LET l(m)=0: NEXT m
 40 LET s=0: REM menu value of function
 60 LET inputs=63
 70 LET outputs=63
 80 LET score=0
 90 FOR m=1 TO 10
 95 GO SUB 5000
 99 LET attempt=0
100 CLS
102 PRINT "Question ";m
105 PRINT : PRINT "Which function"
106 PRINT : PRINT "is the board"
107 PRINT : PRINT "now making?"
108 PRINT : PRINT "Press a key from 0 to 9."
120 GO SUB 7000: REM draw gate
121 PRINT AT 11,0;"0. A AND B            1. A OR B"
122 PRINT AT 13,0;"2. NOT B              3. NOT A"
123 PRINT AT 15,0;"4.(NOT A AND B) OR (NOT B AND A)"
124 PRINT AT 17,0;"5.(NOT A AND NOT B) OR (A AND B)"
125 PRINT AT 19,0;"6. NOT(A AND B)   7. NOT(A OR B)"
126 PRINT AT 21,0;"8. NOT A AND B    9. NOT A OR B"
200 REM simulate the gate
220 LET olda=5: LET oldb=5: REM reset input status
230 GO SUB 6000
240 LET s$=INKEY$: IF s$="" THEN  GO TO 230
250 IF s$<"0" OR s$>"9" THEN  GO TO 210
260 LET s=(CODE s$)-48
340 REM check the response
350 IF s=n THEN  GO TO 400
355 LET attempt=1
360 REM wrong response
365 CLS
370 FLASH 1: PRINT AT 2,5;"WRONG": FLASH 0
375 PRINT
376 GO SUB 1000+s
380 PRINT AT 5,0;"Press key C to try again."
385 PRINT AT 7,0;"Press key A for the answer."
390 LET l$=INKEY$: IF l$<>"c" AND l$<>"A" AND l$<>"C" AND l$<>"a" THEN  GO TO 3
90
395 IF l$="c" OR l$="C" THEN  GO TO 100
396 CLS
397 PRINT "Check the truth"
398 PRINT : PRINT "for yourself"
399 GO TO 450
400 REM correct response
410 CLS
420 IF attempt=0 THEN  LET score=score+1
430 IF attempt=0 THEN  PRINT "Correct first time"
440 IF attempt=1 THEN  PRINT "Correct this time"
450 REM display gate
470 GO SUB 7000
480 GO SUB 1000+n
500 REM display truth table
505 GO SUB 1200
```



```
510 FOR a=0 TO 1
520 FOR b=0 TO 1
530 GO SUB fn
540 LET p=12+2*a+4*b
550 PRINT AT p,19;vo
560 NEXT b
570 NEXT a
580 PRINT AT 21,0;"Press Q for next question."
590 LET row=13
600 PRINT AT row,2;"     "
610 PRINT AT row,25;"     "
615 LET olda=5: LET oldb=5: REM reset input status
620 GO SUB 6000
630 LET row=12+2*a+4*b
640 PRINT AT row,2;"     "
650 PRINT AT row,25;"     "
660 IF INKEY$="q" THEN  GO TO 890
670 GO SUB 6000: REM read input status
700 IF same THEN  GO TO 660
750 GO TO 600
890 NEXT m
900 REM score routine
910 CLS
920 PRINT AT 3,0;"You got ";score;" correct."
930 IF score<4 THEN  PRINT AT 5,0;"This is poor. Repeat the test."
940 IF score<7 AND score>3 THEN  PRINT AT 5,0;"This is fair but not good.": PRI
NT AT 7,0;"You may wish to repeat the test."
950 IF score>6 THEN  PRINT AT 5,0;"This is a good score. Well done."
960 STOP
1000 PRINT AT 3,23;"AND": RETURN
1001 PRINT AT 3,23;"OR": RETURN
1002 PRINT AT 3,22;"NOT B": RETURN
1003 PRINT AT 3,22;"NOT A": RETURN
1004 PRINT AT 3,22;"EX-OR": RETURN
1005 PRINT AT 3,22;"EQUIV": RETURN
1006 PRINT AT 3,22;"NAND": RETURN
1007 PRINT AT 3,23;"NOR": RETURN
1008 PRINT AT 2,22;"NOT A": PRINT AT 4,22;"AND B": RETURN
1009 PRINT AT 2,22;"NOT A": PRINT AT 4,22;"OR B": RETURN
1200 REM draw truth table
1210 PRINT AT 8,8;"tnnntnnntnnnnnnp"
1220 PRINT AT 9,8;"p A p B poutput p"
1230 PRINT AT 10,8;"uooouooouooooooop"
1240 PRINT AT 11,8;"p   p   p        p"
1250 PRINT AT 12,8;"p 0 p 0 p        p"
1260 PRINT AT 13,8;"p   p   p        p"
1270 PRINT AT 14,8;"p 0 p 1 p        p"
1280 PRINT AT 15,8;"p   p   p        p"
1290 PRINT AT 16,8;"p 1 p 0 p        p"
1300 PRINT AT 17,8;"p   p   p        p"
1310 PRINT AT 18,8;"p 1 p 1 p        p"
1320 PRINT AT 19,8;"uooouooouooooooop"
1350 RETURN
2000 REM  AND function
2010 LET vo=a AND b
2050 RETURN
2100 REM OR function
2110 LET vo=a OR b
2150 RETURN
2200 REM NOT B function
2210 LET vo=NOT b
2250 RETURN
```

```
2300 REM NOT A function
2310 LET vo=NOT a
2350 RETURN
2400 REM EXCLUSIVE-OR function
2410 LET vo=1
2420 IF a=b THEN  LET vo=0
2440 RETURN
2500 REM EQUIVALENCE function
2510 LET vo=0
2520 IF a=b THEN  LET vo=1
2540 RETURN
2600 REM NAND function
2610 LET vo=NOT (a AND b)
2660 RETURN
2700 REM NOR function
2710 LET vo=NOT (a OR b)
2760 RETURN
2800 REM NOT A AND B function
2810 LET vo=(NOT a) AND b
2850 RETURN
2900 REM NOT A OR B function
2910 LET vo=(NOT a) OR b
2950 RETURN
3000 REM line graphics
3010 FOR i=0 TO 20
3020  FOR j=0 TO 7
3030  READ row
3040 POKE USR CHR$ (i+144)+j,row
3050 NEXT j
3070 NEXT i
3080 RETURN
3100 DATA 0,0,0,255,255,0,0,0
3110 DATA 16,16,16,255,255,16,16,16
3120 DATA 16,16,16,16,16,16,16,16
3130 DATA 16,16,16,31,31,0,0,0
3140 DATA 16,16,16,240,240,0,0,0
3150 DATA 0,0,0,31,31,16,16,16
3160 DATA 0,0,0,240,240,16,16,16
3170 DATA 16,16,16,255,255,0,0,0
3180 DATA 0,0,0,255,255,16,16,16
3190 DATA 16,16,16,31,31,16,16,16
3200 DATA 16,16,16,240,240,16,16,16
3210 DATA 128,128,128,255,255,128,128,128
3220 DATA 1,1,1,255,255,1,1,1
3240 DATA 255,0,0,0,0,0,0,0
3250 DATA 0,0,0,0,0,0,0,255
3260 DATA 128,128,128,128,128,128,128,128
3270 DATA 1,1,1,1,1,1,1,1
3280 DATA 255,1,1,1,1,1,1,1
3290 DATA 1,1,1,1,1,1,1,255
3300 DATA 255,128,128,128,128,128,128,128
3310 DATA 128,128,128,128,128,128,128,255
4000 REM collect input status
4010 LET a=0: LET b=0
4011 PAUSE 1: LET q=IN inputs
4012 IF q>127 THEN  LET q=q-128
4013 IF q>63 THEN  LET q=q-64
4014 IF q>31 THEN  LET q=q-32
4015 IF q>15 THEN  LET q=q-16
4030 IF q>7 THEN  LET q=q-8
4040 IF q>3 THEN  LET q=q-4
4050 IF q>1 THEN  LET b=1: LET q=q-2
```



```
4060 IF q>0 THEN  LET a=1
4070 LET same=0
4100 IF olda=a AND oldb=b THEN  LET same=1
4110 LET olda=a: LET oldb=b
4120 RETURN
5000 REM choose function
5010 LET n=INT (RND*10)
5020 IF l(n+1)<>0 THEN  GO TO 5010
5030 LET l(n+1)=1
5040 LET fn=2000+100*n
5050 RETURN
6000 REM organize the gate
6010 GO SUB 4000: REM collect input status
6015 IF same THEN  RETURN
6020 PRINT AT 2,18;a: PRINT AT 4,18;b
6030 REM compute output
6040 GO SUB fn
6050 REM send output
6060 OUT outputs,8*vo
6070 PRINT AT 3,30;vo
6080 RETURN
7000 REM draw gate
7010 PRINT AT 1,21;"tnnnnnnp"
7020 PRINT AT 2,16;"A( )ap      p"
7030 PRINT AT 3,21;"p      1( )"
7040 PRINT AT 4,16;"B( )ap      p"
7050 PRINT AT 5,21;"uoooooop"
7070 RETURN
```

Program 3 LEAST SQUARES PLOT

```
   1 REM least squares plot
1000 REM **********************
1010 REM
1020 REM      Collect data
1030 REM
1040 REM **********************
1050 REM
1060 CLS
1070 PRINT AT 0,6;"LEAST SQUARES PLOT"
1080 PRINT AT 3,0;"Enter the number of data pairs"
1090 INPUT "Number of readings ";numreadings
1095 DIM x(numreadings)
1096 DIM y(numreadings)
1100 PRINT AT 5,0;numreadings
1110 PRINT AT 7,0;"Enter each pair of readings"
1120 PRINT AT 9,0;"in the order x-coord, y-coord."
1130 FOR n=1 TO numreadings
1140 INPUT "x-coord= ";x(n)
1150 INPUT "y-coord = ";y(n)
1155 PRINT x(n);"   ";y(n)
1160 NEXT n
1170 CLS : GO SUB 8000: REM list readings
1180 PRINT : PRINT "Do you wish to change"
1190 PRINT : PRINT "any readings? Enter Y or N."
1200 INPUT a$
1210 IF a$<>"N" AND a$<>"n" AND a$<>"Y" AND a$<>"y" THEN  GO TO 1200
1220 IF a$="N" OR a$="n" THEN  GO TO 2000
1230 PRINT : PRINT "Enter the reference number for"
1240 PRINT : PRINT "the data pair you wish to change"
1250 INPUT m
1260 IF m>numreadings THEN  PRINT : PRINT "You did not enter this.": GO TO 1170
```

```
1270 PRINT : PRINT "Enter the new pair of readings."
1280 INPUT "x-coord = ";x(m)
1285 INPUT "y-coord = ";y(m)
1330 GO TO 1170
2000 REM ***********************
2010 REM
2020 REM       Determine axes
2030 REM
2040 REM ***********************
2050 REM
2060 CLS
2070 PRINT : PRINT "Enter the x-axis maximum"
2075 PRINT : PRINT "as a multiple 10."
2080 INPUT xmax
2090 PRINT : PRINT "Enter the y-axis maximum"
2095 PRINT : PRINT "as a multiple 10."
2100 INPUT ymax
2110 LET xscale=xmax/10
2115 LET yscale=ymax/10
5000 REM ***********************
5010 REM
5020 REM        Draw axes
5030 REM
5040 REM ***********************
5050 REM
5060 CLS
5070 PLOT 8,0: DRAW 0,175
5080 PLOT 0,8: DRAW 255,0
5100 FOR y=0 TO 10
5110 PRINT AT (21-2*y),0;y*yscale
5120 NEXT y
5130 FOR x=0 TO 10
5140 PRINT AT 21,(x*3);x*xscale;
5150 NEXT x
5200 REM ***********************
5210 REM
5220 REM    Linear regression
5230 REM
5240 REM ***********************
5250 REM
5270 LET xtotal=0
5280 LET ytotal=0
5290 LET sumxsquares=0
5310 LET sumxyproduct=0
5320 FOR n=1 TO numreadings
5330 LET x=24.5*x(n)/xscale
5340 LET y=16.5*y(n)/yscale
5360 LET xtotal=xtotal+x
5370 LET ytotal=ytotal+y
5380 LET sumxsquares=sumxsquares+x*x
5390 LET sumxyproduct=sumxyproduct+x*y
5400 PLOT x+6,y+8: DRAW 4,0
5410 PLOT x+8,y+6: DRAW 0,4
5420 NEXT n
5430 REM ***********************
5440 REM
5460 REM Calculate slope & intercpt
5470 REM ***********************
5480 REM
5490 IF numreadings<2 THEN  GO TO 5740
5500 LET slope=(numreadings*sumxyproduct-xtotal*ytotal)/(numreadings*sumxsquares
```



```
-xtotal*xtotal)
5510 LET intercept=(ytotal-slope*xtotal)/numreadings
5520 REM ***********************
5530 REM
5540 REM       Plot line
5550 REM
5560 REM ***********************
5570 REM
5580 REM plot minimum x-value
5585 LET x=0
5590 LET y=intercept+slope*x
5595 IF y<-8 THEN  LET x=x+10: GO TO 5590
5600 PLOT x+8,y+8
5610 LET x=245
5620 LET y=intercept+slope*x
5625 IF y>165 THEN  LET x=x-10: GO TO 5620
5630 DRAW x+8-PEEK 23677,y+8-PEEK 23678
5640 INPUT "Press ENTER to alter readings";p$
5650 CLS
5660 GO SUB 8000
5670 GO TO 1230
8000 REM List readings
8010 CLS
8020 PRINT AT 0,0;"n      x          y"
8030 FOR n=1 TO numreadings
8040 PRINT n;"      ";x(n),y(n)
8050 NEXT n
8060 RETURN
```

Program 4 MASTERMIND

```
 10 REM Mastermind
 20 DIM a(4)
 30 DIM b(4)
 40 DIM c(4)
 50 DIM r(25)
 60 DIM s(25)
 70 DIM t(25)
 80 DIM z(25)
100 PRINT AT 0,5;"MASTERMIND"
110 PRINT AT 3,0;"This game lets you guess"
120 PRINT AT 5,0;"four digits, chosen at random."
125 PRINT AT 8,0;"Press I if you want instructions"
126 PRINT AT 10,0;"otherwise press SPACE."
127 LET A$=INKEY$
128 IF A$=" " THEN  GO TO 300
129 IF A$<>"i" THEN  GO TO 127
130 CLS : PRINT AT 0,5;"MASTERMIND"
135 PRINT AT 2,0;"The game works like this."
140 PRINT AT 4,0;"If I pick the sequence 1 2 3 4"
150 PRINT AT 6,0;"and your guess is 4 2 6 3,"
160 PRINT AT 8,0;"then you score one BULL,"
170 PRINT AT 10,0;"because 2 is correct and is"
180 PRINT AT 12,0;"also in the correct position."
190 PRINT AT 14,0;"You score two COWS, because 4"
200 PRINT AT 16,0;"and 3 are correct digits, but"
210 PRINT AT 18,0;"they are in the wrong positions."
220 PRINT AT 21,0;"Press G for a game."
230 LET A$=INKEY$
240 IF A$<>"g" AND A$<>"G" THEN  GO TO 230
300 LET z=1
```

```
330 CLS
340 PRINT AT 0,0;"Choose the difficulty level."
350 PRINT AT 2,0;"This is the number of different"
360 PRINT AT 4,0;"kinds of digit I may choose from"
370 PRINT AT 6,0;"Pick one from the following list"
380 PRINT AT 8,0;"Level 4 (digits 1,2,3 or 4)"
390 PRINT AT 10,0;"Level 5 (digits 1,2,3,4 or 5)"
400 PRINT AT 12,0;"Level 6 (digits 1,2,3,4,5 or 6)"
410 PRINT AT 14,0;"Level 7 (digits 1 to 7)"
420 PRINT AT 16,0;"Level 8 (digits 1 to 8)"
430 PRINT AT 18,0;"Level 9 (digits 1 to 9)"
440 PRINT AT 21,0;"Press a number from 4 to 9.";
450 LET A$=INKEY$: IF A$="" THEN  GO TO 450
455 LET A=VAL A$
460 IF A<4 OR A>9 THEN  GO TO 330
500 CLS
530 FOR n=1 TO 4
540 LET a(n)=INT (RND*A)+1
550 NEXT n
560 PRINT AT 2,0;"Now make your guess."
570 PRINT : PRINT "Type out your next four digits,"
580 PRINT
600 PRINT "type 0000 to be told the answer."
610 FOR i=1 TO 4
620 LET b$=INKEY$: IF b$="" THEN  GO TO 620
625 FOR w=1 TO 20: NEXT w
630 IF b$=CHR$ (12) AND i>1 THEN  LET i=i-1: PRINT AT 20,5+2*i;" ": GO TO 620
640 IF CODE b$>57 OR CODE b$<48 THEN  GO TO 620
650 LET b(i)=VAL b$
660 IF b(i)>A THEN  PRINT AT 21,0;"That digit is out of range.":: GO TO 620
670 PRINT AT 21,0;"                                ";
680 PRINT AT 20,5+2*i;b(i)
690 NEXT i
700 IF b(1)=0 AND b(2)=0 AND b(3)=0 AND b(4)=0 THEN  GO TO 1140
710  LET r(z)=1000*b(1)+100*b(2)+10*b(3)+b(4)
720 LET y=0: LET x=0
730 FOR n=1 TO 4: LET c(n)=a(n): NEXT n
740 FOR n=1 TO 4
750 IF c(n)<>b(n) THEN  GO TO 770
760 LET x=x+1: LET c(n)=99: LET b(n)=100
770 NEXT n
780 FOR n=1 TO 4: FOR m=1 TO 4
790 IF c(n)<>b(m) THEN  GO TO 810
800 LET y=y+1: LET c(n)=99: LET b(m)=100
810 NEXT m: NEXT n
820 CLS
830 PRINT "GUESS    BULLS    COWS    GUESS NO."
840 LET s(z)=x: LET t(z)=y
850 FOR l=1 TO z: PRINT r(l);"       ";s(l);"       ";t(l);"       ";l: NEXT l
860 IF x=4 THEN  GO TO 890
870 LET z=z+1: IF z>15 THEN  GO TO 940
880 GO TO 570
890 PRINT : PRINT "Well done, you guessed correctly"
900 PRINT "You took "¡z¡" guesses."
910 PRINT "Press Y to try again."
920 LET q$=INKEY$: IF q$="" THEN  GO TO 920
930 IF q$="y" OR q$="Y" THEN  GO TO 330
935 STOP
940 CLS : PRINT : PRINT "You don't seem to know how"
950 PRINT : PRINT "to play. You should not just"
960 PRINT : PRINT "make wild guesses."
970 PRINT : PRINT "Use the information about"
```



```
980 PRINT : PRINT "BULLS and COWS to help you."
990 PRINT
1000 PRINT : PRINT "Only change one digit each time"
1010 PRINT : PRINT "then you can see if your score"
1020  PRINT : PRINT "goes up or down."
1030 PRINT : PRINT "Press Y for a new game."
1040 GO TO 920
1140 REM Give correct answer
1150 CLS
1160 PRINT : PRINT "My number is  "; a(1);a(2);a(3);a(4)
1170 PRINT : PRINT ;"Do you see your difficulty?"
1180 PRINT : PRINT "Try again.  Press Y."
1190 GO TO 920
```

Program 5 Z80 SIMULATION

```
  1 CLEAR 65000
  2 REM ZCODE by L.D.Firth April 1983
  3 PRINT AT 10,0;"When display shows,set CAPS LOCK"
  4 GO SUB 9000
  8 GO SUB 7000
 10 LET a$=""
 15 DIM r(24): REM registers
 16 LET x$="ABCDEHL"
 18 DEF FN r$(r)=("  "+STR$ r)(LEN STR$ r TO )
 19 DEF FN S$(r)=("   "+STR$ r)(LEN STR$ r TO )
 20 LET c=0: REM carry
 22 LET t=64
 24 LET b=256
 26 LET f=500
 28 LET q=32000
 30 LET z=c
 32 LET m=0
 35 LET a=32500: REM prog counter
 40 DIM m(10): REM memory
 45 DIM s(6): REM stack
 50 GO TO 505
 60 INPUT a$: REM instruction
 61 LET l=LEN a$
 62 RESTORE
 64 FOR n=1 TO 17
 66 READ d$
 68 IF l<LEN d$ THEN  GO TO 72
 70 IF a$<>d$ AND a$( TO LEN d$)=d$ THEN  GO TO 90
 72 NEXT n
 74 GO TO 495
 80 DATA "LD (","LD ","INC ","DEC ","AND ","OR ","XOR ","CP ","ADD ","SUB ","AD
C ","SBC ","JR","PUSH ","POP ","CALL ","RE"
 90 LET b$=a$(LEN d$+1 TO )
 95 GO TO 95+5*n
100 IF l<9 THEN  GO TO 495
101 IF b$( TO 4)="HL)," OR b$( TO 3)="IX+" THEN  GO TO 1000
102 IF b$="DE),A" OR b$="BC),A" THEN  GO TO 1000
103 IF b$( TO 4)="3200" THEN  GO TO 1025
104 GO TO 495
105 IF l<6 THEN  GO TO 495
106 IF b$="A,(BC)" OR b$="A,(DE)" OR b$(2 TO )=",(HL)" OR b$(2 TO 1-5)=",(IX+"
THEN  GO TO 1120
107 IF b$( TO 3)="IX," OR b$( TO 3)="BC," OR b$( TO 3)="DE," OR b$( TO 3)="HL,"
THEN  GO TO 1165
108 IF a$(5)="," AND b$>="A" THEN  GO TO 1100+50*(b$( TO 3)="A,(")
109 GO TO 495
115 IF l=5 THEN  GO TO 1300
116 IF b$="(HL)" OR b$( TO 1-6)="(IX+" THEN  GO TO 1430
117 IF b$="BC" OR b$="DE" OR b$="HL" OR b$="IX" THEN  GO TO 1470
119 GO TO 495
```

```
130 IF LEN b$<4 THEN  GO TO 1600
131 IF b$="(HL)" OR b$( TO 4)="(IX+" THEN  GO TO 1500
135 IF 1<7 THEN  GO TO 1700
136 IF b$="(HL)" THEN  GO TO 1670
137 IF b$( TO 4)="(IX+" THEN  GO TO 1660
138 GO TO 495
155 IF 1<7 THEN  GO TO 495
156 IF b$="A,(HL)" OR b$( TO 1-6)="A,(IX+" THEN  GO TO 2000
157 IF b$( TO 2)="A," THEN  GO TO 2100
158 IF b$( TO 3)="HL," THEN  GO TO 1800
159 GO TO 495
160 IF a$(3)=" " THEN  GO TO 2200
161 IF 1<5 THEN  GO TO 495
162 IF (a$( TO 5)="JRNC " OR a$( TO 5)="JRNZ ") AND 1>5 THEN  GO TO 2750
163 IF (a$( TO 4)="JRC " OR a$( TO 4)="JRZ ") AND 1>4 THEN  GO TO 2750
164 GO TO 495
170 GO TO 2400
175  IF a$(6)<="9" THEN  GO TO 2500
176 IF 1<8 THEN  GO TO 495
178 GO TO 2750
180 IF b$="T" THEN  GO TO 2600
181 LET c$=(b$+"    ")(3 TO 4)
182 IF c$="NC" OR c$="NZ" OR c$="C " OR c$="Z " THEN  GO TO 2760
495 LET a$=a$+": ERROR"
496 GO TO 502
501 LET a=a+1+(a$(1)>="0" AND a$(1)<="9")+(a$(1)="X")
503 PRINT AT 19,2;(a$+"                    ")( TO 18)
510 PRINT AT 15,14;FN S$(a);AT 11,14;FN S$(INT m+q);AT 7,12;FN r$(r(1));AT 3,1
FN r$(r(2));AT 3,6;FN r$(r(3));AT 7,2;FN r$(r(4));AT 7,6;FN r$(r(5));AT 11,2;F
r$(r(8));AT 11,6;FN r$(r(12));AT 15,3;FN S$(b*r(9)+r(24));AT 3,11;c;AT 3,14;z
515 PRINT AT 3+r(7),21;"1";AT 1+r(7),21;" ";AT 5+r(7),21;" "
520 FOR n=1 TO 7
526 PRINT AT n*3-2,25;FN r$(m(n))
530 IF n<>7 THEN  PRINT AT 2+n,17;FN r$(s(n))
535 NEXT n
540 LET dis=.1
600 GO TO 60
1000 LET v$=a$(9 TO 1)
1001 IF b$>"I" THEN  GO SUB 3500
1002 GO TO 3000
1010 LET m=b*r(CODE a$(5)-t)+r(CODE a$(6)-t)-q+dis
1015 IF m>.1 AND m<8 THEN  LET m(m)=v
1020 GO TO f
1025 IF a$(9)<="0" OR a$(9)>="8" THEN  GO TO 495
1027 LET m=VAL a$(9)
1030 LET b$=a$(10 TO 1)
1035 IF b$=",A" THEN  GO TO 1050
1040 IF b$=",IX" OR b$=",BC" OR b$=",DE" OR b$=",HL" THEN  GO TO 1070
1045 GO TO 495
1050 LET v=r(1)
1052 LET a=a+2
1055 GO TO 1015
1070 LET m(m+1)=r(CODE b$(3)-t)
1075 LET m(m)=r(CODE b$(4)-t)
1078 LET a=a+3
1080 GO TO f
1100 LET v$=a$(6 TO 1)
1102 GO TO 3000
1105 LET n=2.05
1106 LET v$=a$(4)
1107 LET val=v
1108 GO TO 3080
1110 LET r(CODE v$-t)=val
1115 GO TO f
1120 IF b$(4)="I" THEN  GO SUB 3500
1122 LET m=b*r(CODE a$(7)-t)+r(CODE a$(8)-t)-q+dis
1125 GO SUB 4010
1130 GO TO 1105
```



```
1150 IF b$( TO 7)<>"A,(3200" THEN  GO TO 495
1152 IF b$(8)>="1" AND b$(8)<="7" THEN  LET r(1)=m(VAL b$(8))
1155 LET a=a+2
1160 GO TO f
1165 IF a$(7 TO 1-2)="(3200" THEN  GO TO 1185
1170 LET v$="0"+a$(7 TO 1)
1172 LET n=2.7
1173 GO TO 3000
1175 LET a=a+1+(a$(5)="X")
1180 GO TO 1485
1185 IF a$(12)<="0" OR a$(12)>="8" THEN  GO TO 495
1187 LET m=VAL a$(12)
1188 LET r(CODE a$(5)-t)=m(m)
1190 LET r(CODE a$(4)-t)=(m<7)*m(m+1)
1193 LET a=a+3
1195 GO TO f
1300 LET v$=a$(5)
1301 LET n=4
1302 GO TO 3080
1305 LET v=v+1-2*(a$<"E")
1310 GO SUB 2800
1315 LET r(CODE a$(1)-t)=v
1320 GO TO f
1430 IF b$(2)="I" THEN  GO SUB 3500
1435 GO SUB 4000
1440 LET v=v+1-2*(a$<"E")
1445 GO SUB 2800
1450 LET m(m)=v
1455 GO TO f
1470 LET v=b*r(CODE a$(5)-t)+r(CODE a$(6)-t)
1475 LET v=v+1-2*(a$<"E")
1480 LET v=v+b*b*((v<0)-(v>=b*b))
1485 LET r(CODE a$(6-(n<3))-t)=v-b*INT (v/b)
1490 LET r(CODE a$(5-(n<3))-t)=INT (v/b)
1495 GO TO f
1500 IF b$(2)="I" THEN  GO SUB 3500
1510 GO SUB 4000
1550 GO TO 1605
1600 LET v$=b$
1602 LET n=7
1603 GO TO 3000
1605 LET d=4*((a$>"N")+(a$>"W"))
1610 POKE 65002,r(1)
1615  POKE 65004,d
1620 POKE 65006+d,v
1625 LET v=INT ((USR 65001)/b)
1630 GO SUB 2800
1635 LET r(1)=v
1640 GO TO f
1660 GO SUB 3500
1670 GO SUB 4000
1685 GO TO 1705
1700 LET v$=b$
1704 GO TO 3000
1705 LET v=r(1)-v
1710 GO SUB 2800
1720 GO TO f
1800 LET c$=a$(1-1 TO 1)
1802 IF NOT (c$="BC" OR c$="DE" OR c$="HL") THEN  GO TO 495
1805 IF n=10 THEN  GO TO 495
1810 LET v=b*r(8)+r(12)
1820 LET v=v+(b*r(CODE a$(8)-t)+r(CODE a$(9)-t)+c*(n>10))*SGN (67-CODE a$)
1830 LET c=(v>=b*b)+(v<0)
1840 LET v=v+b*b*((v<0)-(v>=b*b))
1850 IF n>10 THEN  LET z=v=0
1855 LET a=a+(n>10)
1860 GO TO 1485
2000 IF b$(4)="I" THEN  GO SUB 3500
```

```
2010 GO SUB 4000
2030 GO TO 2105
2100 LET v$=a$(7 TO 1)
2101 LET n=12
2102 GO TO 3000
2105 LET v=r(1)+(v+c*(a$(3)="C"))*SGN (67-CODE a$)
2110 GO SUB 2800
2120 LET r(1)=v
2130 GO TO f
2200 LET v$=a$(4 TO 1)
2202 GO TO 3000
2205 LET a=a+v
2210 GO TO f
2280 GO TO f
2400 LET r(6)=2*c+z: REM flagreg
2405 IF NOT (b$="IX" OR b$="AF" OR b$="BC" OR b$="DE" OR b$="HL") THEN  GO TO 4
5
2410 IF n=15 THEN  GO TO 2450
2415 LET v=b*r(CODE a$(6)-t)+r(CODE a$(7)-t)
2420 IF r(7)=6 THEN  GO TO f
2425 LET s(r(7)+1)=INT (v/b)
2430 LET s(r(7)+2)=v-b*INT (v/b)
2440 LET r(7)=r(7)+2
2445 GO TO f
2450 IF r(7)=0 THEN  GO TO f
2455 LET r(7)=r(7)-2
2460 LET r(CODE a$(5)-t)=s(r(7)+1)
2470 LET r(CODE a$(6)-t)=s(r(7)+2)
2475 LET flags=r(6)-4*INT (r(6)/4)
2480 LET c=INT (flags/2)
2485 LET z=flags-2*c
2490 GO TO f
2500 LET v$=a$(6 TO 1)
2502 GO TO 3000
2505 LET olda=a+3
2510 LET a=v-2
2515 LET v=olda
2520 GO TO 2420
2575 LET a=a+1
2580 GO TO f
2600 IF r(7)=0 THEN  GO TO f
2610 LET r(7)=r(7)-2
2620 LET a=b*s(r(7)+1)+s(r(7)+2)-1
2630 GO TO f
2680 GO TO f
2750 LET v$=b$(3+(b$(1)="N") TO )
2755 LET c$=b$( TO 2)
2760 IF c*(c$(1)="C")+z*(c$(1)="Z")+(NOT c)*(c$="NC")+(NOT z)*(c$="NZ") THEN
 TO 3000-400*(n=17)
2770 IF c$(1)<>"C" AND c$<>"NC" AND c$(1)<>"Z" AND c$<>"NZ" THEN  GO TO 490
2780 GO TO 970+100*n
2800 IF N>4 THEN  LET c=(v>=b)+(v<0)
2810 LET v=v+b*((v<0)-(v>=b))
2820 LET z=v=0
2825 RETURN
3000 FOR d=1+(d$="JR" AND v$(1)="-") TO LEN v$
3010 IF v$(d)<"0" OR v$(d)>"9" THEN  GO TO 3070
3020 NEXT d
3030 LET v=VAL v$
3040 IF d$="JR" AND INT (.5+v/b)<>0 THEN  GO TO 490
3050 IF v>=b+(n=2.7 OR d$="CALL ")*65280 THEN  GO TO 490
3060 GO TO 905+100*n
3070 IF n>12 THEN  GO TO 490
3080 FOR x=1 TO 7
3090 IF x$(x)=v$ THEN  GO TO 3200
3095 NEXT x
3100 GO TO 490
```



```
3200 LET v=r(CODE v$-t)
3210  GO TO 905+100*n
3500 LET dp=4+(n=2 OR n>8)*2+(n>1)
3505 IF b$(dp)<"0" OR b$(dp)>"7" THEN  GO TO 490
3510 LET dis=VAL b$(dp)
3520 IF b$(dp+1)<>")" THEN  GO TO 490
3522 LET a=a+2
3525 IF LEN b$<dp+3 THEN  RETURN
3530 LET v$=b$(dp+3 TO LEN b$)
3540 RETURN
4000 LET m=b*r(9)+r(24)-q+dis
4005 IF dis=.1 THEN  LET m=b*r(8)+r(12)-q
4010 IF m<1 OR m>7 THEN  GO TO 4050
4020 LET v=m(m)
4030 RETURN
4050 LET v=0
4060 RETURN
7000 REM machine code routine for logic instructions
7010 POKE 65001,62
7011 POKE 65003,24
7012 POKE 65005,230: REM AND
7013 POKE 65007,71
7014 POKE 65008,201
7015 POKE 65009,246: REM  OR
7016 POKE 65011,71
7017 POKE 65012,201
7018 POKE 65013,238: REM EXCLUSIVE-OR
7019 POKE 65015,71
7020 POKE 65016,201
7500 CLS
8000 PRINT AT 0,0;"faaaaaaaaaaaaaaaaaaaag faaagAdd"
8002 PRINT "c           C  Z STACK c c   c 1"
8004 PRINT "cfaaaiaaagfagfagfaaag c daaae"
8006 PRINT "cc   c   cc cc cc   c c faaag"
8008 PRINT "cdaaahaaaedaedaec   c c c   c 2"
8010 PRINT "c  B   C   Flagsc   c c daaae"
8012 PRINT "cfaaaiaaag faaagc   c c faaag"
8014 PRINT "cc   c   c c   cc   c c c   c 3"
8016 PRINT "cdaaahaaae daaaec   c c daaae"
8020 PRINT "c  D   E    ACC daaae c faaag"
8022 PRINT "cfaaaiaaag  faaaaaaag c c   c 4 "
8024 PRINT "cc   c   c  c       c c daaae"
8026 PRINT "cdaaahaaae  daaaaaaae c faaag"
8028 PRINT "c  H   L     ADDR REG c c   c 5"
8030 PRINT "cfaaaaaaag  faaaaaaag c daaae"
8032 PRINT "cc       c  c       c c faaag"
8034 PRINT "cdaaaaaaae  daaaaaaae c c   c 6"
8036 PRINT "c X-INDEX   PROG CNTR c daaae"
8038 PRINT "cfaaaaaaaaaaaaaaaaaag c faaag"
8040 PRINT "cc                    c c c   c 7"
8042 PRINT "cdaaaaaaaaaaaaaaaaaae c daaae"
8044 PRINT "daaaaaaaaaaaaaaaaaaaae 32000+"
8200 RETURN
9000 REM graphics
9001 RESTORE 9000
9002 FOR i=0 TO 11
9004 FOR j=0 TO 7
9010 READ row
9020 POKE USR CHR$ (i+144)+j,row
9030 NEXT j
9040 NEXT i
9100 DATA 0,0,0,255,255,0,0,0
```

```
9110 DATA 16,16,16,255,255,16,16,16
9120 DATA 16,16,16,16,16,16,16,16
9130 DATA 16,16,16,31,31,0,0,0
9140 DATA 16,16,16,240,240,0,0,0
9150 DATA 0,0,0,31,31,16,16,16
9160 DATA 0,0,0,240,240,16,16,16
9170 DATA 16,16,16,255,255,0,0,0
9180 DATA 0,0,0,255,255,16,16,16
9190 DATA 16,16,16,31,31,16,16,16
9200 DATA 16,16,16,240,240,16,16,16
9210 DATA 16,32,64,255,64,32,16,0
9220 RETURN
9400 INPUT r
9500 LET r$=("   "+STR$ r)(LEN STR$ r TO )
9600 PRINT r$
9700 GO TO 9400
```

Program 6 REACTION TIMER

```
  1 REM REACTION TIMER
 10 CLEAR 31473
 15 DIM g(4): REM digits for display
 16 PRINT AT 10,0;"Loading data, please wait."
 20 FOR i=31474 TO 31516
 30 READ x
 40 POKE i,x
 50 NEXT i
 60 DATA 42,0,125,58,2
 70 DATA 125,55,63,23,23
 80 DATA 23,79,6,126,22
 90 DATA 8,10,30,6,31
100 DATA 48,4,54,0,24
110 DATA 2,54,56,44,29
120 DATA 32,243,245,125,198
130 DATA 26,111,241,12,21
140 DATA 32,230,201
200 FOR i=32256 TO 32375
210 READ x
220 POKE i,x
230 NEXT i
240 DATA 31,17,17,17,17,17,31,0
250 DATA 4,4,4,4,4,4,4,0
260 DATA 31,17,16,16,31,1,31,0
270 DATA 31,17,16,30,16,17,31,0
280 DATA 1,1,1,9,31,8,8,0
290 DATA 31,1,1,31,16,17,31,0
300 DATA 31,1,1,31,17,17,31,0
310 DATA 31,16,16,16,16,16,16,0
320 DATA 31,17,17,31,17,17,31,0
330 DATA 31,17,17,31,16,16,16,0
340 DATA 0,0,0,0,0,0,4,0
350 DATA 0,0,0,30,0,0,0,0
360 DATA 0,0,31,21,21,21,21,0
370 DATA 0,0,30,2,30,16,30,0
380 DATA 17,17,17,31,17,17,17,0
390 DATA 0,0,31,16,31,1,31,0
500 CLS
510 PRINT AT 1,8;"REACTION TIMER"
520 PRINT AT 5,0;"The screen will go blank between"
530 PRINT AT 7,0;"5 and 10 seconds after"
540 PRINT AT 9,0;"you press 'B'."
```



```
550 PRINT AT 11,0;"Immediately the screen goes"
560 PRINT AT 13,0;"blank, you must press 'B'"
570 PRINT AT 15,0;"Your reaction time will then be"
580 PRINT AT 17,0;"displayed."
590 IF INKEY$<>"g" THEN  GO TO 590
600 LET max=500+INT (500*RND)
610 PRINT AT 20,0;"The screen will soon go blank."
620 FOR t=1 TO max
625 IF INKEY$="b" THEN  GO TO 900
627 NEXT t
630 CLS
640 POKE 23672,0
650 IF INKEY$="b" THEN  GO TO 700
660 IF PEEK 23672>100 THEN  GO TO 800
670 GO TO 650
700 LET rt=(PEEK 23672)/50
705 LET q=rt
710 GO SUB 1000: REM sort and display digits
720 PRINT AT 1,5;"Press 'C' for another go."
730 IF rt<0.2 THEN  LET a$="EXCELLENT": GO TO 770
740 IF rt<0.3 THEN  LET a$="GOOD": GO TO 770
750 IF rt<0.4 THEN  LET a$="FAIR": GO TO 770
760 LET a$="POOR"
770 PRINT AT 19,0;"Your reaction time is ";a$
780  IF INKEY$<>"c" THEN  GO TO 730
790 GO TO 500
800 PRINT AT 5,0;"WAKE UP !!!!!!!!!!"
805 PRINT AT 10,0;"I am not going to wait all day !"
810 PRINT AT 15,5;"Press 'C' for another go."
820  IF INKEY$<>"c" THEN  GO TO 820
830 GO TO 500
900 REM detect cheating
905 CLS
910 PRINT AT 5,0;"No cheating please !"
920 PRINT AT 8,0;"When you stop hitting 'B'"
940 PRINT AT 10,0;"I shall continue."
950 PRINT AT 15,5;"Press 'C' for another go."
960  IF INKEY$<>"c" THEN  GO TO 960
970 GO TO 500
1000 REM sort and display digits
1010 LET dp=0
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 3
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1200 FOR i=1 TO 3
1210 POKE 32002,g(i)
1220 POKE 32001,89
1230 POKE 32000,(6*i)
1240 LET z=USR 31474
1250 NEXT i
1300 REM display s
1310 POKE 32002,13
1320 POKE 32001,89
1330 POKE 32000,24
1340 LET z=USR 31474
1350 RETURN
```

Program 7 STOPCLOCK

```
 1 CLEAR 63999
 2 REM stopclock
 3 PRINT AT 10,0;"Loading data, please wait."
10 FOR i=65040 TO 65082
11 READ x
12 POKE i,x
13 NEXT i
14 DATA 42,0,254,58,2
15 DATA 254,55,63,23,23
16 DATA 23,79,6,253,22
17 DATA 8,10,30,6,31
18 DATA 48,4,54,0,24
19 DATA 2,54,56,44,29
20 DATA 32,243,245,125,198
21 DATA 26,111,241,12,21
22 DATA 32,230,201
25 FOR i=64768 TO 64895
26 READ x
27 POKE i,x
28 NEXT i
29 DATA 31,17,17,17,17,17,31,0
30 DATA 4,4,4,4,4,4,4,0
31 DATA 31,17,16,16,31,1,31,0
32 DATA 31,17,16,30,16,17,31,0
33 DATA 1,1,1,9,31,8,8,0
34 DATA 31,1,1,31,16,17,31,0
35 DATA 31,1,1,31,17,17,31,0
36 DATA 31,16,16,16,16,16,16,0
37 DATA 31,17,17,31,17,17,31,0
38 DATA 31,17,17,31,16,16,16,0
39 DATA 0,0,0,0,0,0,4,0
40 DATA 0,0,0,30,0,0,0,0
41 DATA 0,0,31,21,21,21,21,0
42 DATA 0,0,30,2,30,16,30,0
43 DATA 17,17,17,31,17,17,17,0
44 DATA 0,0,31,16,31,1,31,0
45 REM stopclock routine
50 FOR i=64000 TO 64376
51 READ x
52 POKE i,x
53 NEXT i
54 DATA 62,0,50,0,252,50,1,252,50,2
55 DATA 252,50,3,252,50,4,252,0,0
56 DATA 62,12,50,2,254,62,24,50,0,254
57 DATA 62,88,50,1,254,205,16,254,0,0
58 DATA 62,13,50,2,254,62,24,50,0,254
59 DATA 62,89,50,1,254,205,16,254
60 DATA 62,10,50,2,254,62,6,50,0,254
61 DATA 62,90,50,1,254,205,16,254
62 DATA 62,0,50,120,92,205,0,251,219,63,230,3,50,5,252
63 DATA 219,63,230,3,71,58,5,252,184,40
64 DATA 245,120,50,5,252,62,0,50,120,92
65 DATA 58,120,92,254,5,218,223,250,62,0,50,120,92
66 DATA 58,0,252,198,1,50,0,252,254,10
67 DATA 218,223,250,62,0,50,0,252,0,0
68 DATA 58,1,252,198,1,50,1,252,254,10
69 DATA 218,223,250,62,0,50,1,252,0,0
70 DATA 58,2,252,198,1,50,2,252,254,6
71 DATA 218,223,250,62,0,50,2,252,0,0
72 DATA 58,3,252,198,1,50,3,252,254,10
```



```
 73 DATA 218,223,250,62,0,50,3,252,0,0
 74 DATA 58,4,252,198,1,50,4,252,254,10
 75 DATA 218,223,250,62,0,50,4,252,0,0
 76 DATA 62,127,219,254,246,224,254,255,194,110,250
 77 DATA 205,0,251,219,63,230,3,71,58,5
 78 DATA 252,184,202,110,250,201
 79 DATA 0,0,0,0,0,0
 80 REM showtimes subroutine
 81 DATA 58,120,92,7,50,2,254,62,18,50,0,254,62,90,50,1,254,205,16,254
 82 DATA 58,0,252,50,2,254,62,12,50,0,254,62,90,50,1,254,205,16,254,0
 83 DATA 58,1,252,50,2,254,62,18,50,0,254,62,89,50,1,254,205,16,254,0
 84 DATA 58,2,252,50,2,254,62,12,50,0,254,62,89,50,1,254,205,16,254,0
 85 DATA 58,3,252,50,2,254,62,18,50,0,254,62,88,50,1,254,205,16,254,0
 86 DATA 58,4,252,50,2,254,62,12,50,0,254,62,88,50,1,254,205,16,254,0
 87 DATA 201
900 CLS
910 PRINT AT 0,0;"Start and"
920 PRINT AT 2,0;"stop the"
930 PRINT AT 4,0;"clock at"
940 PRINT AT 6,0;"switch"
950 PRINT AT 8,0;"input"
955 PRINT AT 10,0;"0 or 1."
960 PRINT AT 15,0;"Freeze the"
970 PRINT AT 17,0;"display by"
980 PRINT AT 19,0;"pressing"
990 PRINT AT 21,0;"SPACE."
1000 RANDOMIZE USR 64000
1010  PRINT AT 0,0;"          "
1020  PRINT AT 2,0;"          "
1030  PRINT AT 4,0;"Press R   "
1040  PRINT AT 6,0;"for another"
1050  PRINT AT 8,0;"measurement."
1060  PRINT AT 10,0;"          "
1070  PRINT AT 15,0;"Press E   "
1080  PRINT AT 17,0;"to finish."
1090  PRINT AT 19,0;"          "
1100  PRINT AT 21,0;"          "
1110 IF INKEY$<>"e" AND INKEY$<>"r" THEN  GO TO 1110
1120 IF INKEY$="r" THEN  GO TO 900
```

Program 8 FAST TIMER

```
 10 CLEAR 31473
 15 DIM g(5): REM digits for display
 16 PRINT AT 10,0;"Loading data, please wait."
 20 FOR i=31474 TO 31516
 30 READ x
 40 POKE i,x
 50 NEXT i
 60 DATA 42,0,125,58,2
 70 DATA 125,55,63,23,23
 80 DATA 23,79,6,126,22
 90 DATA 8,10,30,6,31
100 DATA 48,4,54,0,24
110 DATA 2,54,56,44,29
120 DATA 32,243,245,125,198
130 DATA 26,111,241,12,21
140 DATA 32,230,201
200 FOR i=32256 TO 32375
210 READ x
220 POKE i,x
```

```
230 NEXT i
240 DATA 31,17,17,17,17,17,31,0
250 DATA 4,4,4,4,4,4,4,0
260 DATA 31,17,16,16,31,1,31,0
270 DATA 31,17,16,30,16,17,31,0
280 DATA 1,1,1,9,31,8,8,0
290 DATA 31,1,1,31,16,17,31,0
300 DATA 31,1,1,31,17,17,31,0
310 DATA 31,16,16,16,16,16,16,0
320 DATA 31,17,17,31,17,17,31,0
330 DATA 31,17,17,31,16,16,16,0
340 DATA 0,0,0,0,0,0,4,0
350 DATA 0,0,0,30,0,0,0,0
360 DATA 0,0,31,21,21,21,21,0
370 DATA 0,0,30,2,30,16,30,0
390 DATA 0,0,31,16,31,1,31,0
400 FOR i=64000 TO 64024
410 READ x
420 POKE i,x
430 NEXT i
450 DATA 243,1,0,0,219,63,230,1,32,250
455 DATA 219,63,230,1,40,250
460 DATA 3,219,63,230,1,32,249,251,201
500 CLS
510 PRINT AT 0,8;"FAST TIMER"
520  PRINT AT 4,0;"The timing begins when switch"
530 PRINT AT 6,0;"input 0 goes HIGH and stops"
540 PRINT AT 8,0;"when input 0 goes LOW."
550 PRINT AT 15,5;"Ready to begin."
560 PRINT AT 10,0;"Maximum time interval: 655 ms"
800 LET t=USR 64000
810 LET q=t/100
815 CLS
820 GO SUB 1000
830 PRINT AT 2,0;" Press R for another reading."
840 IF INKEY$<>"r" THEN  GO TO 840
850 GO TO 500
1000 REM sort and display digits
1010 LET dp=0
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 3
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1200 FOR i=1 TO 5
1210 POKE 32002,g(i)
1220 POKE 32001,89
1230 POKE 32000,(6*(i-1))
1240 LET z=USR 31474
1250 NEXT i
1300 REM display s
1310 POKE 32002,13
1320 POKE 32001,90
1330 POKE 32000,24
1340 LET z=USR 31474
1400 REM display m
1410 POKE 32002,12
1420 POKE 32001,90
1430 POKE 32000,18
```



```
1440 LET z=USR 31474
1450 RETURN
```

Program 9 TSA METER

```
 1 CLEAR 63999
 2 REM TIME, SPEED & ACCELERATION METER
 3 PRINT AT 10,0;"Loading data, please wait."
10 FOR i=65040 TO 65082
11 READ x
12 POKE i,x
13 NEXT i
14 DATA 42,0,254,58,2
15 DATA 254,55,63,23,23
16 DATA 23,79,6,253,22
17 DATA 8,10,30,6,31
18 DATA 48,4,54,0,24
19 DATA 2,54,56,44,29
20 DATA 32,243,245,125,198
21 DATA 26,111,241,12,21
22 DATA 32,230,201
25 FOR i=64768 TO 64895
26 READ x
27 POKE i,x
28 NEXT i
29 DATA 31,17,17,17,17,17,31,0
30 DATA 4,4,4,4,4,4,4,0
31 DATA 31,17,16,16,31,1,31,0
32 DATA 31,17,16,30,16,17,31,0
33 DATA 1,1,1,9,31,8,8,0
34 DATA 31,1,1,31,16,17,31,0
35 DATA 31,1,1,31,17,17,31,0
36 DATA 31,16,16,16,16,16,16,0
37 DATA 31,17,17,31,17,17,31,0
38 DATA 31,17,17,31,16,16,16,0
39 DATA 0,0,0,0,0,0,4,0
40 DATA 0,0,0,30,0,0,0,0
41 DATA 0,0,31,21,21,21,21,0
42 DATA 0,0,30,2,30,16,30,0
43 DATA 17,17,17,31,17,17,17,0
44 DATA 0,0,31,16,31,1,31,0
45 REM Machine code timing routine
46 FOR i=64000 TO 64184
47 READ x
48 POKE i,x
49 NEXT i
50 DATA 243,6,0,221,33,0,251,221,54,0
51 DATA 0,221,35,16,248,22,0,30,0,46
52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
53 DATA 251,50,244,250,50,246,250,219,63,230
54 DATA 3,71,219,63,230,3,79,184,40,248
55 DATA 0,121,168,65,254,1,40,8,254,2
56 DATA 40,18,121,238,2,71,58,243,250,198
57 DATA 4,50,243,250,221,42,243,250,24,12
58 DATA 58,245,250,198,4,50,245,250,221,42
59 DATA 245,250,221,115,0,221,114,1,221,117
60 DATA 2,58,247,250,61,50,247,250,40,28
61 DATA 123,198,1,95,122,206,0,87,125,206
62 DATA 0,111,219,254,230,1,40,10,219,63
63 DATA 230,3,79,184,40,230,24,169,6,64
64 DATA 221,33,248,251,221,126,4,221,150,0
```

```
 65 DATA 221,119,4,221,126,5,221,158,1,221
 66 DATA 119,5,221,126,6,221,158,2,221,119
 67 DATA 6,221,45,221,45,221,45,221,45,16
 68 DATA 219,251,201
120 DIM g(5): REM digits for display
125 DIM a(4): REM acceleration stores
130 DIM t(4): REM time interval stores
135 DIM s(4): REM speed stores
140 REM define large digits
141 POKE USR "A"+0,0
142 POKE USR "A"+1,0
143 POKE USR "A"+2,0
144 POKE USR "A"+3,255
145 POKE USR "A"+4,255
146 POKE USR "A"+5,0
147 POKE USR "A"+6,0
148 POKE USR "A"+7,0
150 POKE USR "B"+0,24
151 POKE USR "B"+1,24
152 POKE USR "B"+2,24
153 POKE USR "B"+3,24
154 POKE USR "B"+4,24
155 POKE USR "B"+5,24
156 POKE USR "B"+6,24
157 POKE USR "B"+7,24
160 POKE USR "C"+0,255
161 POKE USR "C"+1,255
162 POKE USR "C"+2,3
163 POKE USR "C"+3,3
164 POKE USR "C"+4,3
165 POKE USR "C"+5,3
166 POKE USR "C"+6,3
167 POKE USR "C"+7,3
170 POKE USR "D"+0,3
171 POKE USR "D"+1,255
172 POKE USR "D"+2,255
173 POKE USR "D"+3,192
174 POKE USR "D"+4,192
175 POKE USR "D"+5,192
176 POKE USR "D"+6,255
177 POKE USR "D"+7,255
400 REM BEGIN
410 CLS
420 PRINT "TIME, SPEED & ACCELERATION METER"
430 PRINT AT 3,0;"For ACCELERATION press 'A'"
440 PRINT AT 5,0;"For SPEED press 'S'"
450 PRINT AT 7,0;"For TIME INTERVAL press 'T'"
460 LET a$=INKEY$
470 IF a$<>"a" AND a$<>"s" AND a$<>"t" THEN  GO TO 460
480 IF a$="a" THEN  GO TO 700
490 IF a$="s" THEN  GO TO 600
500 REM TIME INTERVAL
505 CLS : PRINT AT 0,3;"MEASURING TIME INTERVALS"
515 GO SUB 9000
520 GO TO 535
525 PRINT AT 1,0;"Ready to take reading ";k
535 POKE 64247,2: REM two events
540 LET h=USR 64000
545 GO SUB 4000
550 LET q=t1
555 LET t(k)=q
560 GO SUB 1000
```



```
570 LET k=k+1
575 IF k>f THEN  GO TO 7000
580 GO TO 525
600 REM SPEED
605 CLS : PRINT AT 0,10;"MEASURING SPEEDS"
615 GO SUB 9000
620 GO TO 635
625 PRINT AT 1,0;"Ready to take reading ";k
635 POKE 64247,2: REM two events
640 LET h=USR 64000
645 GO SUB 4000
650 LET q=0.04/t1
655 LET s(k)=q
660 GO SUB 1000
670 LET k=k+1
675 IF k>f THEN  GO TO 7000
680 GO TO 625
700 REM ACCELERATIONS
705 CLS : PRINT AT 0,5;"MEASURING ACCELERATIONS"
715 GO SUB 9000
720 GO TO 735
725 PRINT AT 1,0;"Ready to take reading ";k
735 POKE 64247,4: REM four events
740 LET h=USR 64000
745 GO SUB 4000
750 LET t2=t2+(t1+t3)/2
752 LET q=0.04*((1/t3-1/t1)/t2)
755 LET a(k)=q
760 GO SUB 1000
770 LET k=k+1
775 IF k>f THEN  GO TO 7000
780 GO TO 725
1000 REM sort and display digits
1010 LET dp=0: LET sign=0
1015 IF q<0 THEN  LET q=ABS q: LET sign=1
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 4
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1120 IF sign=0 THEN  GO TO 1200
1130 FOR i=4 TO 1 STEP -1
1140 LET g(i+1)=g(i)
1150 NEXT i
1160 LET g(1)=11
1200 CLS
1205 FOR i=1 TO 5
1210 POKE 65026,g(i)
1220 POKE 65025,89
1230 POKE 65024,(6*(i-1))
1240 LET h=USR 65040
1250 NEXT i
1300 REM display s
1310 POKE 65026,13
1320 POKE 65025,90
1330 POKE 65024,22
1340 LET h=USR 65040
1350  IF a$="t" THEN  RETURN
1360 REM display m
```

```
1370 POKE 65026,12
1380 POKE 65025,90
1390 POKE 65024,16
1400 LET h=USR 65040
1410 PRINT AT 17,28;"aa"
1420 IF a$="a" THEN  PRINT AT 17,31;"c": PRINT AT 18,31;"d"
1430 IF a$="s" THEN  PRINT AT 17,31;"b": PRINT AT 18,31;"b"
1440 RETURN
4000 REM Calculate times
4005 LET conv=0.000031506
4010  LET base=64260
4020 LET t1=(65536*PEEK (base+2)+256*PEEK (base+1)+PEEK (base))*conv
4030 LET t2=(65536*PEEK (base+6)+256*PEEK (base+5)+PEEK (base+4))*conv
4040 LET t3=(65536*PEEK (base+10)+256*PEEK (base+9)+PEEK (base+8))*conv
4050 IF (t1+t2+t3)=0 THEN  LET base=base+128: GO TO 4020
4060 RETURN
7000 REM restart routine
7010 PRINT AT 0,0;"Press 'R' to restart"
7020 PRINT : PRINT "Press 'M' to recall readings"
7030 LET c$=INKEY$
7040 IF c$="r" THEN  GO TO 410
7050 IF c$<>"m" THEN  GO TO 7030
7060 IF a$="a" THEN  GO TO 7300
7070 IF a$="s" THEN  GO TO 7200
7100 REM List times
7110 CLS : PRINT
7120 FOR z=1 TO f
7130 PRINT : PRINT "Time interval (";z;") = ";INT (10000*t(z))/10000;" s"
7140 NEXT z
7150 PRINT : PRINT : PRINT "Press 'R' to restart"
7160 IF INKEY$<>"r" THEN  GO TO 7160
7170 GO TO 410
7200 REM List speeds
7210 CLS : PRINT
7220 FOR z=1 TO f
7230 PRINT : PRINT "Speed (";z;") = ";INT (10000*s(z))/10000;"  m/s"
7240 NEXT z
7250 GO TO 7150
7300 REM List accelerations
7310 CLS : PRINT
7320 FOR z=1 TO f
7330 PRINT : PRINT "Acceleration (";z;") = ";INT (10000*a(z))/10000;" m/ss"
7340 NEXT z
7350 GO TO 7150
9000 PRINT : PRINT "You may take 1, 2, 3 or 4"
9010 PRINT : PRINT "successive readings, which will"
9020 PRINT : PRINT "be stored as well as being"
9030 PRINT : PRINT "displayed."
9040 PRINT : PRINT "When you are ready to begin,"
9050 PRINT : PRINT "press one of these numbers."
9055 IF INKEY$<>"" THEN  GO TO 9055
9060 LET e$=INKEY$
9065 IF e$="" THEN  GO TO 9060
9070 LET f=VAL (e$)
9080 IF f<1 OR f>4 THEN  GO TO 9060
9090 PRINT : PRINT "O.K.  I'm ready."
9100 LET k=1
9200 RETURN
```


Program 10 ACCELERATION TUTOR

```
 1 CLEAR 63999
 2 REM ACCELERATION TUTOR
 3 PRINT AT 10,0;"Loading data, please wait."
10 FOR i=65040 TO 65082
11 READ x
12 POKE i,x
13 NEXT i
14 DATA 42,0,254,58,2
15 DATA 254,55,63,23,23
16 DATA 23,79,6,253,22
17 DATA 8,10,30,6,31
18 DATA 48,4,54,0,24
19 DATA 2,54,56,44,29
20 DATA 32,243,245,125,198
21 DATA 26,111,241,12,21
22 DATA 32,230,201
25 FOR i=64768 TO 64895
26 READ x
27 POKE i,x
28 NEXT i
29 DATA 31,17,17,17,17,17,31,0
30 DATA 4,4,4,4,4,4,4,0
31 DATA 31,17,16,16,31,1,31,0
32 DATA 31,17,16,30,16,17,31,0
33 DATA 1,1,1,9,31,8,8,0
34 DATA 31,1,1,31,16,17,31,0
35 DATA 31,1,1,31,17,17,31,0
36 DATA 31,16,16,16,16,16,16,0
37 DATA 31,17,17,31,17,17,31,0
38 DATA 31,17,17,31,16,16,16,0
39 DATA 0,0,0,0,0,0,4,0
40 DATA 0,0,0,30,0,0,0,0
41 DATA 0,0,31,21,21,21,21,0
42 DATA 0,0,30,2,30,16,30,0
43 DATA 17,17,17,31,17,17,17,0
44 DATA 0,0,31,16,31,1,31,0
45 REM Machine code timing routine
46 FOR i=64000 TO 64184
47 READ x
48 POKE i,x
49 NEXT i
50 DATA 243,6,0,221,33,0,251,221,54,0
51 DATA 0,221,35,16,248,22,0,30,0,46
52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
53 DATA 251,50,244,250,50,246,250,219,63,230
54 DATA 3,71,219,63,230,3,79,184,40,248
55 DATA 0,121,168,65,254,1,40,8,254,2
56 DATA 40,18,121,238,2,71,58,243,250,198
57 DATA 4,50,243,250,221,42,243,250,24,12
58 DATA 58,245,250,198,4,50,245,250,221,42
59 DATA 245,250,221,115,0,221,114,1,221,117
60 DATA 2,58,247,250,61,50,247,250,40,28
61 DATA 123,198,1,95,122,206,0,87,125,206
62 DATA 0,111,219,254,230,1,40,10,219,63
63 DATA 230,3,79,184,40,230,24,169,6,64
64 DATA 221,33,248,251,221,126,4,221,150,0
65 DATA 221,119,4,221,126,5,221,158,1,221
66 DATA 119,5,221,126,6,221,158,2,221,119
67 DATA 6,221,45,221,45,221,45,221,45,16
68 DATA 219,251,201
```

```
100 REM Define symbols, etc.
120 DIM g(5): REM digits for display
125 DIM a(4): REM acceleration stores
130 DIM t(4): REM time interval stores
135 DIM s(4): REM speed stores
140 REM define large digits
141 POKE USR "A"+0,0
142 POKE USR "A"+1,0
143 POKE USR "A"+2,0
144 POKE USR "A"+3,255
145 POKE USR "A"+4,255
146 POKE USR "A"+5,0
147 POKE USR "A"+6,0
148 POKE USR "A"+7,0
150 POKE USR "B"+0,24
151 POKE USR "B"+1,24
152 POKE USR "B"+2,24
153 POKE USR "B"+3,24
154 POKE USR "B"+4,24
155 POKE USR "B"+5,24
156 POKE USR "B"+6,24
157 POKE USR "B"+7,24
160 POKE USR "C"+0,255
161 POKE USR "C"+1,255
162 POKE USR "C"+2,3
163 POKE USR "C"+3,3
164 POKE USR "C"+4,3
165 POKE USR "C"+5,3
166 POKE USR "C"+6,3
167 POKE USR "C"+7,3
170 POKE USR "D"+0,3
171 POKE USR "D"+1,255
172 POKE USR "D"+2,255
173 POKE USR "D"+3,192
174 POKE USR "D"+4,192
175 POKE USR "D"+5,192
176 POKE USR "D"+6,255
177 POKE USR "D"+7,255
400 REM BEGIN
410 CLS
420 PRINT "ACCELERATION TUTOR"
430 PRINT AT 3,0;"Connect a single 40-mm card"
440 PRINT AT 5,0;"to a trolley."
450 PRINT AT 7,0;"Connect two photocells in series"
460 PRINT AT 9,0;"to switch input 0."
470 PRINT AT 11,0;"Let the trolley pass in front"
480 PRINT AT 13,0;"of both photocells."
490 GO TO 700
500 LET a$="t": REM TIME INTERVALS
520 IF 1$="t" THEN  LET q=t2
530 IF 1$="i" THEN  LET q=t1
540 IF 1$="f" THEN  LET q=t3
550 GO SUB 1000
560 IF 1$="i" THEN  PRINT AT 0,0;"Initial time interval": GO TO 770
570 IF 1$="f" THEN  PRINT AT 0,0;"Final time interval": GO TO 770
580 IF 1$="t" THEN  PRINT AT 0,0;"Time interval between speeds": GO TO 770
600 LET a$="s": REM SPEED
610 IF 1$="i" THEN  LET q=0.04/t1
620 IF 1$="f" THEN  LET q=0.04/t3
630 GO SUB 1000
640 IF 1$="i" THEN  PRINT AT 0,0;"Initial speed"
650 IF 1$="f" THEN  PRINT AT 0,0;"Final speed"
```



```
660 PRINT AT 1,0;"Press T for measured time"
670 PRINT AT 2,0;"Press M to return to main menu."
680 LET m$=INKEY$: IF m$<>"m" AND m$<>"t" THEN  GO TO 680
690 IF m$="m" THEN  GO TO 740
695 GO TO 500
700 REM ACCELERATIONS
710 PRINT AT 18,0;"READY TO MEASURE ACCELERATION"
720 POKE 64247,4: REM four events
730 LET h=USR 64000
740 GO SUB 4000
750 LET q=0.04*((1/t3-1/t1)/t2)
755 LET a$="a"
760 GO SUB 1000
765 PRINT AT 0,0;"Acceleration"
770 PRINT "Press T for time between speeds"
780 PRINT "Press F to display final speed"
790 PRINT "Press I to display initial speed"
800 PRINT "Press A to display acceleration"
805 PRINT "Press R to take new readings"
810 LET 1$=INKEY$
820 IF 1$<>"r" AND 1$<>"t" AND 1$<>"i" AND 1$<>"f" AND 1$<>"a" THEN  GO TO 810
830 IF 1$="t" THEN  GO TO 500
840 IF 1$="a" THEN  GO TO 750
850 IF 1$="r" THEN  GO TO 400
860 GO TO 600
1000 REM sort and display digits
1010 LET dp=0: LET sign=0
1015 IF q<0 THEN  LET q=ABS q: LET sign=1
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 4
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1120 IF sign=0 THEN  GO TO 1200
1130 FOR i=4 TO 1 STEP -1
1140 LET g(i+1)=g(i)
1150 NEXT i
1160 LET g(1)=11
1200 CLS
1205 FOR i=1 TO 5
1210 POKE 65026,g(i)
1220 POKE 65025,89
1230 POKE 65024,(6*(i-1))
1240 LET h=USR 65040
1250 NEXT i
1300 REM display s
1310 POKE 65026,13
1320 POKE 65025,90
1330 POKE 65024,22
1340 LET h=USR 65040
1350  IF a$="t" THEN  RETURN
1360 REM display m
1370 POKE 65026,12
1380 POKE 65025,90
1390 POKE 65024,16
1400 LET h=USR 65040
1410 PRINT AT 17,28;"aa"
1420 IF a$="a" THEN  PRINT AT 17,31;"c": PRINT AT 18,31;"d"
1430 IF a$="s" THEN  PRINT AT 17,31;"b": PRINT AT 18,31;"b"
```

```
1440 RETURN
4000 REM Calculate times
4005 LET conv=0.000031506
4010  LET base=64260
4020 LET t1=(65536*PEEK (base+2)+256*PEEK (base+1)+PEEK (base))*conv
4030 LET t2=(65536*PEEK (base+6)+256*PEEK (base+5)+PEEK (base+4))*conv
4040 LET t3=(65536*PEEK (base+10)+256*PEEK (base+9)+PEEK (base+8))*conv
4050 IF (t1+t2+t3)=0 THEN  LET base=base+128: GO TO 4020
4055 LET t2=t2+(t1+t3)/2
4060 RETURN
```

Program 11 CONSERVATION OF MOMENTUM

```
 1 CLEAR 63999
 2 REM CONSERVATION OF MOMENTUM
 3 PRINT AT 10,0;"Loading data, please wait."
10 DIM t(4,2): REM stores for time intervals
46 FOR i=64000 TO 64184
47 READ x
48 POKE i,x
49 NEXT i
50 DATA 243,6,0,221,33,0,251,221,54,0
51 DATA 0,221,35,16,248,22,0,30,0,46
52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
53 DATA 251,50,244,250,50,246,250,219,63,230
54 DATA 3,71,219,63,230,3,79,184,40,248
55 DATA 0,121,168,65,254,1,40,8,254,2
56 DATA 40,18,121,238,2,71,58,243,250,198
57 DATA 4,50,243,250,221,42,243,250,24,12
58 DATA 58,245,250,198,4,50,245,250,221,42
59 DATA 245,250,221,115,0,221,114,1,221,117
60 DATA 2,58,247,250,61,50,247,250,40,28
61 DATA 123,198,1,95,122,206,0,87,125,206
62 DATA 0,111,219,254,230,1,40,10,219,63
63 DATA 230,3,79,184,40,230,24,169,6,64
64 DATA 221,33,248,251,221,126,4,221,150,0
65 DATA 221,119,4,221,126,5,221,158,1,221
66 DATA 119,5,221,126,6,221,158,2,221,119
67 DATA 6,221,45,221,45,221,45,221,45,16
68 DATA 219,251,201
400 REM BEGIN
410 CLS
420 PRINT "CONSERVATION OF MOMENTUM"
430 PRINT AT 3,0;"Connect two photocells,"
440 PRINT AT 5,0;"one to switch input  0"
450 PRINT AT 7,0;"the other to switch input 1."
460 PRINT AT 9,0;"Place a 40-mm card on each"
470 PRINT AT 11,0;"trolley and allow them to"
480 PRINT AT 13,0;"collide in the normal way."
490 PRINT AT 15,0;"The speeds of the trolleys will"
497 PRINT AT 17,0;"be displayed for each photocell."
498 PRINT AT 19,0;"If there are not enough transits"
499 PRINT AT 21,0;"press SPACE to regain control."
500 REM TIME INTERVAL
505 FOR n=1 TO 4
506 LET t(n,1)=0: LET t(n,2)=0
507 NEXT n
510 POKE 64247,8: REM 8 events = 4 transits
520 LET h=USR 64000
530 GO SUB 4000
540 PRINT AT 21,0;"Press R for another measurement."
```



```
590  IF INKEY$<>"r" THEN  GO TO 590
600 GO TO 400
4000 REM Calculate speeds for four transits
4005 LET conv=0.000031506
4010 REM Channel 1 data
4020 LET base=64260
4030 LET t(1,1)=conv*(65536*PEEK (base+2)+256*PEEK (base+1)+PEEK (base))
4035 IF t(1,1)=0 THEN  LET base=base+128: GO TO 4100
4040 LET t(2,1)=conv*(65536*PEEK (base+10)+256*PEEK (base+9)+PEEK (base+8))
4045 IF t(2,1)=0 THEN  LET base=base+128: GO TO 4110
4050 LET t(3,1)=conv*(65536*PEEK (base+18)+256*PEEK (base+17)+PEEK (base+16))
4055 IF t(3,1)=0 THEN  LET base=base+128: GO TO 4120
4060 LET t(4,1)=conv*(65536*PEEK (base+26)+256*PEEK (base+25)+PEEK (base+24))
4065 IF t(4,1)=0 THEN  LET base=base+128: GO TO 4130
4070 GO TO 4200
4090 REM  Channel 2 data
4100 LET t(4,2)=conv*(65536*PEEK (base+26)+256*PEEK (base+25)+PEEK (base+24))
4110 LET t(3,2)=conv*(65536*PEEK (base+18)+256*PEEK (base+17)+PEEK (base+16))
4120 LET t(2,2)=conv*(65536*PEEK (base+10)+256*PEEK (base+9)+PEEK (base+8))
4130 LET t(1,2)=conv*(65536*PEEK (base+2)+256*PEEK (base+1)+PEEK (base))
4200 CLS
4205 LET position=5
4210 PRINT AT 0,5;"CONSERVATION OF MOMENTUM"
4220 PRINT AT 3,0;"Results at photocell 0"
4230 FOR n=1 TO 4
4240 IF t(n,1)=0 THEN  GO TO 4290
4250 PRINT AT position,0;"Speed(";n;")  =  ";0.04/t(n,1);" m/s"
4260 LET position=position+2
4290 NEXT n
4300 LET position=position+2
4320 PRINT AT position,0;"Results at photocell 1"
4325 LET position=position+2
4330 FOR n=1 TO 4
4340 IF t(n,2)=0 THEN  GO TO 4390
4350 PRINT AT position,0;"Speed(";n;")  =  ";0.04/t(n,2);" m/s"
4360 LET position=position+2
4390 NEXT n
4400 RETURN
5000 FOR i=64256 TO 65000
5010 PRINT i,PEEK i
5020 NEXT i
5030 NEXT i
6000 PRINT IN 63: GO TO 6000
```

Program 12 SPEED–TIME PLOTTER

```
 1 CLEAR 63999
 2 REM SPEED-TIME PLOTTER
20 REM define arrows
21 POKE USR "B"+0,8
22 POKE USR "B"+1,4
23 POKE USR "B"+2,2
24 POKE USR "B"+3,255
25 POKE USR "B"+4,2
26 POKE USR "B"+5,4
27 POKE USR "B"+6,8
28 POKE USR "B"+7,0
31 POKE USR "A"+0,16
32 POKE USR "A"+1,56
33 POKE USR "A"+2,84
34 POKE USR "A"+3,146
```

```
 35 POKE USR "A"+4,16
 36 POKE USR "A"+5,16
 37 POKE USR "A"+6,16
 38 POKE USR "A"+7,16
 45 REM Machine code timing routine
 46 FOR i=64000 TO 64184
 47 READ x
 48 POKE i,x
 49 NEXT i
 50 DATA 243,6,0,221,33,0,251,221,54,0
 51 DATA 0,221,35,16,248,22,0,30,0,46
 52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
 53 DATA 251,50,244,250,50,246,250,219,63,230
 54 DATA 3,71,219,63,230,3,79,184,40,248
 55 DATA 0,121,168,65,254,1,40,8,254,2
 56 DATA 40,18,121,238,2,71,58,243,250,198
 57 DATA 4,50,243,250,221,42,243,250,24,12
 58 DATA 58,245,250,198,4,50,245,250,221,42
 59 DATA 245,250,221,115,0,221,114,1,221,117
 60 DATA 2,58,247,250,61,50,247,250,40,28
 61 DATA 123,198,1,95,122,206,0,87,125,206
 62 DATA 0,111,219,254,230,1,40,10,219,63
 63 DATA 230,3,79,184,40,230,24,169,251,201,6,32
 64 DATA 221,33,120,251,221,126,4,221,150,0
 65 DATA 221,119,132,221,126,5,221,158,1,221
 66 DATA 119,133,221,126,6,221,158,2,221,119
 67 DATA 134,221,45,221,45,221,45,221,45,16
 68 DATA 219,251,201
120 DIM s(32): REM speed stores
130 DIM t(32): REM time interval stores
400 REM BEGIN
410 CLS
420 PRINT AT 1,5;"SPEED - TIME PLOTTER"
430 PRINT AT 3,0;"This program measures the"
440 PRINT AT 5,0;"speeds of a 16-toothed card"
450 PRINT AT 7,0;"as it passes in front of a"
460 PRINT AT 9,0;"photocell, connected to"
470 PRINT AT 11,0;"switch input 0."
480 PRINT AT 15,0;"Please release the card now."
490 REM  IF a$="s" THEN  GO TO 600
600 REM  MEASURE SPEEDS
610 POKE 64247,32: REM 32 events
620 LET h=USR 64000
630 CLS : PRINT AT 10,0;"All readings taken."
640 PRINT AT 13,0;"Please wait while the results"
650 PRINT AT 16,0;"are calculated."
700 REM Calculate times and speeds
720  FOR b=1 TO 32
730 LET timebase=64256+(b-1)*4
750 LET t(b)=(65536*PEEK (timebase+2)+256*PEEK (timebase+1)+PEEK (timebase))*0.
004
755 IF b=1 THEN  GO TO 790
760 LET s(b-1)=t(b)-t(b-1)
770 IF s(b-1)=0 THEN  GO TO 790
780 LET s(b-1)=100/s(b-1)
785 IF s(b-1)>165 THEN  LET s(b-1)=165
790 NEXT b
800 REM plot graph
805 CLS
810 REM DISTANCE-TIME PLOT
820 PLOT 10,0: DRAW 0,175
830 PLOT 0,10: DRAW 255,0
```



```
835 PLOT 10,10
840 FOR b=1 TO 32
850 LET y=b*5+5
860 LET x=10+t(b)
870 IF x>255 THEN  GO TO 900
880 DRAW x-PEEK 23677,y-PEEK 23678
900 NEXT b
910 PRINT AT 2,0;"a": PRINT : PRINT "D": PRINT "I": PRINT "S": PRINT "T": PRINT
"A": PRINT "N": PRINT "C": PRINT "E": PRINT AT 21,26;"TIME b"
915 PRINT AT 0,4;"Press S for SPEED-TIME graph"
920 IF INKEY$<>"s" THEN  GO TO 920
1000 REM SPEED-TIME PLOT
1010 CLS
1020 PLOT 10,0: DRAW 0,175
1030 PLOT 0,10: DRAW 255,0
1035 PLOT 10,10
1040 FOR b=1 TO 31
1050 LET y=10+s(b)
1060 LET x=10+t(b)
1070 IF x>255 THEN  GO TO 1100
1080 DRAW x-PEEK 23677,y-PEEK 23678
1100 NEXT b
1110 PRINT AT 3,0;"a": PRINT : PRINT "S": PRINT "P": PRINT "E": PRINT "E": PRINT
 "D": PRINT AT 21,26;"TIME b"
1115 PRINT AT 0,1;"Press D for DISTANCE-TIME graph"
1120 IF INKEY$<>"d" THEN  GO TO 1120
1130 GO TO 800
2000 PRINT AT 0,2;"DISTANCE-TIME graph (Press S)"
```

Program 13 FREQUENCY METER

```
  1 CLEAR 63999
  2 REM  FREQUENCY METER
  3 PRINT AT 10,0;"Loading data, please wait."
 10 FOR i=65040 TO 65082
 11 READ x
 12 POKE i,x
 13 NEXT i
 14 DATA 42,0,254,58,2
 15 DATA 254,55,63,23,23
 16 DATA 23,79,6,253,22
 17 DATA 8,10,30,6,31
 18 DATA 48,4,54,0,24
 19 DATA 2,54,56,44,29
 20 DATA 32,243,245,125,198
 21 DATA 26,111,241,12,21
 22 DATA 32,230,201
 25 FOR i=64768 TO 64895
 26 READ x
 27 POKE i,x
 28 NEXT i
 29 DATA 31,17,17,17,17,17,31,0
 30 DATA 4,4,4,4,4,4,4,0
 31 DATA 31,17,16,16,31,1,31,0
 32 DATA 31,17,16,30,16,17,31,0
 33 DATA 1,1,1,9,31,8,8,0
 34 DATA 31,1,1,31,16,17,31,0
 35 DATA 31,1,1,31,17,17,31,0
 36 DATA 31,16,16,16,16,16,16,0
 37 DATA 31,17,17,31,17,17,31,0
 38 DATA 31,17,17,31,16,16,16,0
```

```
 39 DATA 0,0,0,0,0,0,4,0
 40 DATA 0,0,0,30,0,0,0,0
 41 DATA 0,0,31,21,21,21,21,0
 42 DATA 0,0,30,2,30,16,30,0
 43 DATA 17,17,17,31,17,17,17,0
 44 DATA 0,0,31,16,31,1,31,0
 45 REM Machine code timing routine
 46 FOR i=64000 TO 64184
 47 READ x
 48 POKE i,x
 49 NEXT i
 50 DATA 243,6,0,221,33,0,251,221,54,0
 51 DATA 0,221,35,16,248,22,0,30,0,46
 52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
 53 DATA 251,50,244,250,50,246,250,219,63,230
 54 DATA 3,71,219,63,230,3,79,184,40,248
 55 DATA 0,121,168,65,254,1,40,8,254,2
 56 DATA 40,18,121,238,2,71,58,243,250,198
 57 DATA 4,50,243,250,221,42,243,250,24,12
 58 DATA 58,245,250,198,4,50,245,250,221,42
 59 DATA 245,250,221,115,0,221,114,1,221,117
 60 DATA 2,58,247,250,61,50,247,250,40,28
 61 DATA 123,198,1,95,122,206,0,87,125,206
 62 DATA 0,111,219,254,230,1,40,10,219,63
 63 DATA 230,3,79,184,40,230,24,169,6,64
 64 DATA 221,33,248,251,221,126,4,221,150,0
 65 DATA 221,119,4,221,126,5,221,158,1,221
 66 DATA 119,5,221,126,6,221,158,2,221,119
 67 DATA 6,221,45,221,45,221,45,221,45,16
 68 DATA 219,251,201
120 DIM g(5): REM digits for display
400 REM BEGIN
410 CLS
420 PRINT "FREQUENCY METER"
430 PRINT AT 3,0;"Connect the unknown frequency to"
440 PRINT AT 5,0;"switch input  0"
450 PRINT AT 7,0;"through a 'squaring' circuit"
460 PRINT AT 9,0;"if necessary."
500 REM TIME INTERVAL
510 POKE 64247,32: REM 32 events = 16 cycles
520 LET h=USR 64000
530 GO SUB 4000
540 GO SUB 1000
550 PRINT AT 0,0;"While input is still connected"
560 PRINT AT 1,0;"press F to stop."
570 PAUSE 50
590  IF INKEY$="f" THEN  STOP
600 GO TO 500
1000 REM sort and display digits
1010 LET dp=0: LET sign=0
1015 IF q<0 THEN  LET q=ABS q: LET sign=1
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 4
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1120 IF sign=0 THEN  GO TO 1200
1130 FOR i=4 TO 1 STEP -1
```



```
1140 LET g(i+1)=g(i)
1150 NEXT i
1160 LET g(1)=11
1200 CLS
1205 FOR i=1 TO 5
1210 POKE 65026,g(i)
1220 POKE 65025,89
1230 POKE 65024,(6*(i-1))
1240 LET h=USR 65040
1250 NEXT i
1300 REM display H
1310 POKE 65026,14
1320 POKE 65025,90
1330 POKE 65024,18
1340 LET h=USR 65040
1400 REM display z
1410 POKE 65026,15
1420 POKE 65025,90
1430 POKE 65024,24
1440 LET h=USR 65040
1450 RETURN
4000 REM Calculate times
4010 REM Add total time for first 32 events at input 0
4020 REM This is equivalent to 8 cycles
4050 LET conv=0.000031506/16
4060 LET base=64260
4070 LET total=0
4100 FOR c=0 TO 124 STEP 4
4110 LET time=65536*PEEK (base+c+2)+256*PEEK (base+c+1)+PEEK (base+c)
4120 LET total=total+time
4130 NEXT c
4200 LET q=total*conv
4210 RETURN
```

Program 14 PENDULUM PERIOD

```
  1 CLEAR 63999
  2 REM PENDULUM PERIOD
  3 PRINT AT 10,0;"Loading data, please wait."
 10 FOR i=65040 TO 65082
 11 READ x
 12 POKE i,x
 13 NEXT i
 14 DATA 42,0,254,58,2
 15 DATA 254,55,63,23,23
 16 DATA 23,79,6,253,22
 17 DATA 8,10,30,6,31
 18 DATA 48,4,54,0,24
 19 DATA 2,54,56,44,29
 20 DATA 32,243,245,125,198
 21 DATA 26,111,241,12,21
 22 DATA 32,230,201
 25 FOR i=64768 TO 64895
 26 READ x
 27 POKE i,x
 28 NEXT i
 29 DATA 31,17,17,17,17,17,31,0
 30 DATA 4,4,4,4,4,4,4,0
 31 DATA 31,17,16,16,31,1,31,0
 32 DATA 31,17,16,30,16,17,31,0
 33 DATA 1,1,1,9,31,8,8,0
```

```
 34 DATA 31,1,1,31,16,17,31,0
 35 DATA 31,1,1,31,17,17,31,0
 36 DATA 31,16,16,16,16,16,16,0
 37 DATA 31,17,17,31,17,17,31,0
 38 DATA 31,17,17,31,16,16,16,0
 39 DATA 0,0,0,0,0,0,4,0
 40 DATA 0,0,0,30,0,0,0,0
 41 DATA 0,0,31,21,21,21,21,0
 42 DATA 0,0,30,2,30,16,30,0
 43 DATA 17,17,17,31,17,17,17,0
 44 DATA 0,0,31,16,31,1,31,0
 45 REM Machine code timing routine
 46 FOR i=64000 TO 64184
 47 READ x
 48 POKE i,x
 49 NEXT i
 50 DATA 243,6,0,221,33,0,251,221,54,0
 51 DATA 0,221,35,16,248,22,0,30,0,46
 52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
 53 DATA 251,50,244,250,50,246,250,219,63,230
 54 DATA 3,71,219,63,230,3,79,184,40,248
 55 DATA 0,121,168,65,254,1,40,8,254,2
 56 DATA 40,18,121,238,2,71,58,243,250,198
 57 DATA 4,50,243,250,221,42,243,250,24,12
 58 DATA 58,245,250,198,4,50,245,250,221,42
 59 DATA 245,250,221,115,0,221,114,1,221,117
 60 DATA 2,58,247,250,61,50,247,250,40,28
 61 DATA 123,198,1,95,122,206,0,87,125,206
 62 DATA 0,111,219,254,230,1,40,10,219,63
 63 DATA 230,3,79,184,40,230,24,169,6,64
 64 DATA 221,33,248,251,221,126,4,221,150,0
 65 DATA 221,119,4,221,126,5,221,158,1,221
 66 DATA 119,5,221,126,6,221,158,2,221,119
 67 DATA 6,221,45,221,45,221,45,221,45,16
 68 DATA 219,251,201
120 DIM g(5): REM digits for display
400 REM BEGIN
410 CLS
420 PRINT "PENDULUM PERIOD"
430 PRINT AT 3,0;"Connect the photocell to"
440 PRINT AT 5,0;"switch input  0"
450 PRINT AT 7,0;"Allow the pendulum to swing"
460 PRINT AT 9,0;"in front of the photocell."
500 REM TIME INTERVAL
510 POKE 64247,5: REM five events
520 LET h=USR 64000
530 GO SUB 4000
560 GO SUB 1000
580 PRINT AT 0,0;"While pendulum is still swinging"
581 PRINT AT 1,0;"press F to stop."
582 PRINT AT 2,0;"Release pendulum for another"
584 PRINT AT 3,0;"measurement of the period."
586 PAUSE 50
590  IF INKEY$="f" THEN  STOP
600 GO TO 500
1000 REM sort and display digits
1010 LET dp=0: LET sign=0
1015 IF q<0 THEN  LET q=ABS q: LET sign=1
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 4
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
```



```
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1120 IF sign=0 THEN  GO TO 1200
1130 FOR i=4 TO 1 STEP -1
1140 LET g(i+1)=g(i)
1150 NEXT i
1160 LET g(1)=11
1200 CLS
1205 FOR i=1 TO 5
1210 POKE 65026,g(i)
1220 POKE 65025,89
1230 POKE 65024,(6*(i-1))
1240 LET h=USR 65040
1250 NEXT i
1300 REM display s
1310 POKE 65026,13
1320 POKE 65025,90
1330 POKE 65024,22
1340 LET h=USR 65040
1350  RETURN
4000 REM Calculate times
4005 LET conv=0.000031506
4010  LET base=64260
4020 LET t1=(65536*PEEK (base+2)+256*PEEK (base+1)+PEEK (base))*conv
4030 LET t2=(65536*PEEK (base+6)+256*PEEK (base+5)+PEEK (base+4))*conv
4040 LET t3=(65536*PEEK (base+10)+256*PEEK (base+9)+PEEK (base+8))*conv
4042 LET t4=(65536*PEEK (base+14)+256*PEEK (base+13)+PEEK (base+12))*conv
4046 LET q=t1+t2+t3+t4
4050 IF q=0 THEN  LET base=base+128: GO TO 4020
4060 RETURN
```

Program 15 PULSE GENERATOR

```
  1 CLEAR 63999
  2 REM PULSE OUTPUTS
 10  FOR i=64000 TO 64038
 20 READ x
 30 POKE i,x
 40 NEXT i
 50 DATA 243,62,255,211,95
 60 DATA 237,75,0,251,13,32,253,16,251
 70 DATA 62,0,211,95
 80 DATA 237,75,2,251,13,32,253,16,251
 90 DATA 62,127,219,254,246,224,254,255,40,220,251,201
100 CLS
150 PRINT AT 1,7;"PULSE OUTPUT"
160 PRINT AT 5,0;"Square pulses will be output"
170 PRINT AT 7,0;"through the TTL output port."
180 PRINT AT 9,0;"Enter the pulse length"
190 PRINT AT 11,0;"in microsecond units "
200 PRINT AT 13,0;"maximum = 300000, minimum = 30"
210 INPUT length
220 IF length>300000 OR length<30 THEN  GO TO 210
230 CLS : PRINT AT 1,7;"PULSE OUTPUT"
240 PRINT AT 5,0;"Pulse length ";length;" microseconds"
250 PRINT AT 9,0;"Enter the time between pulses"
260 PRINT AT 11,0;"in microsecond units "
270 PRINT AT 13,0;"maximum = 300000, minimum = 60"
275 PRINT AT 15,0;"This must be at least 30": PRINT AT 17,0;"microseconds great
er than"
```

```
276 PRINT AT 19,0;"the length of the pulses."
280 INPUT width
290 IF width>300000 OR width<60 OR (width-length<30) THEN  GO TO 280
300 LET byte=(length*3.5-100)/16
310 LET highbyte=INT (byte/256)
320 LET lowbyte=byte-256*highbyte
330 POKE 64256,lowbyte+1
340 POKE 64257,highbyte+1
350 LET byte=((width-length)*3.5-100)/16
360 LET highbyte=INT (byte/256)
370 LET lowbyte=byte-256*highbyte
380 POKE 64258,lowbyte+1
390 POKE 64259,highbyte+1
400 CLS
410 PRINT AT 1,7;"PULSE OUTPUTS"
420 PRINT AT 5,0;"Pulses of length ";length
430 PRINT AT 7,0;"and time between pulses ";width
440 PRINT AT 9,0;"(in microseconds) are now"
450 PRINT AT 11,0;"being output through"
460 PRINT AT 13,0;"the TTL output port."
500  PRINT AT 21,0;"Press SPACE for different values"
510  LET l=USR 64000
520 GO TO 100
```

Program 16 PROGRAMMABLE OSCILLATOR

```
  1 CLEAR 63999
  5 PRINT AT 5,0;"Loading data. Please wait."
 10 FOR i=0 TO 63
 20 POKE (i+64192),128+127*SIN (i*PI/32)
 30 NEXT i
100 FOR i=64000 TO 64025
110 READ x
120 POKE i,x
130 NEXT i
150 DATA 33,192,250,126,211,127,58,100,250,61
160 DATA 32,253,44,32,244,62,127,219,254,246
170 DATA 224,254,255,40,231,201
180 PRINT AT 8,0;"Enter the desired requency"
190 PRINT AT 10,0;"in the range 15 to 500 Hz."
200 INPUT freq
205  IF freq<15 OR freq>500 THEN  PRINT AT 12,0;freq;" is OUT OF RANGE. Try aga
in.": GO TO 200
210 LET p=INT ((3500000/freq/64-58)/16)+1
220 REM "p" is the number of delay loops to be executed
230 POKE 64100,p
240 CLS
250 PRINT AT 10,0;"Oscillations now being output"
260 PRINT AT 12,5;"Press SPACE to stop"
270 LET l=USR 64000
280 CLS
290 PRINT AT 8,0;"Press F for a new frequency."
300 PRINT AT 10,0;"Press E to finish."
310 IF INKEY$="f" THEN  GO TO 180
320 IF INKEY$<>"e" THEN  GO TO 310
```


Program 17 X-Y PLOTTER

```
  1 CLEAR 63999
  2 REM XYPLOTTER
100 FOR i=64000 TO 64086
110 READ x
120 POKE i,x
130 NEXT i
150 DATA 0,62,0,211,31,62,100,61,32,253,219
160 DATA 31,47,254,176,56,2,62,175,87,62
170 DATA 1,211,31,62,100,61,32,253,219,31
180 DATA 79,122,230,7,103,122,230,56,7,7
190 DATA 111,122,230,192,15,15,15,198,64,132
200 DATA 103,121,230,248,15,15,15,133,111,121
210 DATA 47,230,7,7,7,7,198,198,50,73
220 DATA 250,203,0,62,127,219,254,246,224,254,255,202,1,250,251,201
300 LET l=USR 64000
```

Program 18 STORAGE OSCILLOSCOPE

```
  2 REM ADC Graph Plot
  3 CLEAR 63995
  5 LET a$=""
 10 FOR i=63996 TO 64115
 20 READ x
 30 POKE i,x
 40 NEXT i
 50 DATA 243,33,6,251,62,0
 60 DATA 205,100,250,87,62,0,205,100,250,95
 70 DATA 146,48,1,47,230,240,40,242,115
 80 DATA 58,200,250,254,2,56,38,36,62,1,205,100,250,119
 82 DATA 58,200,250,254,3,56,23,36,62,2,205,100,250,119
 84 DATA 58,200,250,254,4,56,8,36,62,3,205,100,250,119
 86 DATA 37,37,37
 88 DATA 62,127,219,254,246,224,254,255,32,22
 90 DATA 44,40,19,58,201,250,40,6,71,205,110,250,16
100 DATA 251,62,0,205,100,250,119,24,179,251,201
130 DATA 211,31,205,110,250,219,31,201,0,0
140 DATA 62,50,61,32,253,201
150 CLS : PRINT AT 1,5;"STORAGE OSCILLOSCOPE"
160 PRINT AT 4,9;"by R.A.Sparkes"
170 PRINT AT 7,0;"Enter no. of channels required,"
180 PRINT AT 9,0;"minimum 1, maximum 4."
190 PRINT AT 11,0;"Channel 0 is the action channel"
200 INPUT n
201 IF n<1  OR n>4 THEN  GO TO 200
202 PRINT AT 16,0;"Press C for a central axis,"
203 PRINT AT 18,0;"or press B for an axis at the"
204 PRINT AT 20,0;"bottom of the screen."
205  LET c$=INKEY$: IF c$<>"b" AND c$<>"c" THEN  GO TO 205
210 IF n<1  OR n>4 THEN  GO TO 200
214 CLS : PRINT AT 1,5;"STORAGE OSCILLOSCOPE"
220 PRINT AT 5,0;"Enter the time interval between"
230 PRINT AT 7,0;"readings in microseconds."
240 PRINT AT 9,0;"The minimum time is equal to 200*
242 PRINT AT 11,0;"times the number of channels."
245 PRINT AT 13,0;"The maximum time is": PRINT AT 15,0;"50000 microseconds."
250 INPUT m
260 IF m<200*n OR m>50000 THEN  GO TO 250
265 LET m=INT (m/200)
270 POKE 64201,m-n+1
```

```
280 POKE 64200,n
290 PRINT AT 21,0;"I am ready to take readings."
300 LET z=USR 63996
310 IF a$="s" THEN  GO TO 590
320 IF c$="b" THEN  GO TO 400
330 CLS
340 PLOT 16,0
350 DRAW 0,175
355 PLOT 10,90
360 DRAW 240,0
370 PRINT AT 2,0;"+1"
375 PRINT AT 6,0;"+.5"
380 PRINT AT 10,0;"0"
385 PRINT AT 14,0;"-.5"
390 PRINT AT 18,0;"-1"
392 PRINT AT 8,0;"V"
394 PRINT AT 11,31;"t"
395 GO TO 590
400 CLS
410 PLOT 16,0
420 DRAW 0,175
430 PLOT 10,10
440 DRAW 240,0
510 PRINT AT 0,0;"2.5"
512 PRINT AT 4,0;"2.0"
513 PRINT AT 8,0;"1.5"
514 PRINT AT 12,0;"1.0"
515 PRINT AT 16,0;"0.5"
520 PRINT AT 20,0;"0"
530 PRINT AT 6,0;"V"
540 PRINT AT 21,31;"t"
590 FOR j=1 TO n
600 FOR i=1 TO 240
610 LET y=10+0.85*PEEK (64005+i+256*j)
620 IF y>175 THEN  LET y=175
630 LET x=i+15
635 IF i=1 THEN  PLOT x,y: GO TO 650
640 DRAW x-PEEK 23677,y-PEEK 23678
650 NEXT i
660 NEXT j
700 PRINT AT 21,5;"S for same, N for new."
800 LET a$=INKEY$
810 IF a$="" THEN  GO TO 800
815 IF a$="s" THEN  PRINT AT 21,5;"Ready to take readings.": GO TO 300
820 GO TO 150
```

Program 19 FAST ADC

```
  1 REM FAST ADC
  2 REM Graph Plot
  3 CLEAR 63999
 10 FOR i=64000 TO 64036
 20 READ x
 30 POKE i,x
 40 NEXT i
 50 DATA 243,33,6,251,58,192,250,95,6,250
 60 DATA 14,127,219,127,87,219,127,119,146,48
 70 DATA 1,47,230,192,40,245,44,83,21,32
 80 DATA 253,237,162,32,248,251,201
140 LET a$=""
150 CLS : PRINT AT 1,5;"FAST STORAGE OSCILLOSCOPE"
160 PRINT AT 4,9;"by R.A.Sparkes"
```



```
170 PRINT AT 7,0;"Press C for a central axis,"
180 PRINT AT 9,0;"or press B for an axis at the"
190 PRINT AT 11,0;"bottom of the screen."
200  LET c$=INKEY$: IF c$<>"b" AND c$<>"c" THEN  GO TO 200
210 CLS : PRINT AT 1,5;"FAST STORAGE OSCILLOSCOPE"
220 PRINT AT 5,0;"Enter the time interval between"
230 PRINT AT 7,0;"readings in microseconds."
240 PRINT AT 9,0;"The minimum time is equal to 12"
245 PRINT AT 11,0;"and the maximum time is"
250 PRINT AT 13,0;"1000 microseconds."
260 INPUT m
270 IF m<12 OR m>1000 THEN  GO TO 260
280 POKE 64192,1+INT ((m-11)*7/32)
290 PRINT AT 21,0;"I am ready to take readings."
300 LET z=USR 64000
310 IF a$="s" THEN  GO TO 590
320 IF c$="b" THEN  GO TO 400
330 CLS
340 PLOT 16,0
350 DRAW 0,175
355 PLOT 10,90
360 DRAW 240,0
370 PRINT AT 2,0;"+1"
375 PRINT AT 6,0;"+.5"
380 PRINT AT 10,0;"0"
385 PRINT AT 14,0;"-.5"
390 PRINT AT 18,0;"-1"
392 PRINT AT 8,0;"V"
394 PRINT AT 11,31;"t"
395 GO TO 590
400 CLS
410 PLOT 16,0
420 DRAW 0,175
430 PLOT 10,10
440 DRAW 240,0
510 PRINT AT 0,0;"2.5"
512 PRINT AT 4,0;"2.0"
513 PRINT AT 8,0;"1.5"
514 PRINT AT 12,0;"1.0"
515 PRINT AT 16,0;"0.5"
520 PRINT AT 20,0;"0"
530 PRINT AT 6,0;"V"
540 PRINT AT 21,31;"t"
590 PLOT 15,10+0.64*PEEK (64262)
600 FOR i=1 TO 240
610 LET y=10+0.64*PEEK (64262+i)
620 IF y>175 THEN  LET y=175
640 DRAW 1,y-PEEK 23678
650 NEXT i
700 PRINT AT 21,5;"S for same, N for new."
800 LET a$=INKEY$
810 IF a$="" THEN  GO TO 800
815 IF a$="s" THEN  PRINT AT 21,5;"Ready to take readings.": GO TO 300
820 GO TO 150
```

Program 20 DIGITAL MULTIMETER

```
 10 CLEAR 32255
 15 DIM g(4): REM digits for display
 16 PRINT AT 10,0;"Loading data, please wait."
 20 FOR i=32512 TO 32554
 30 READ x
```

```
 40 POKE i,x
 50 NEXT i
 60 DATA 42,0,125,58,2
 70 DATA 125,55,63,23,23
 80 DATA 23,79,6,126,22
 90 DATA 8,10,30,6,31
100 DATA 48,4,54,0,24
110 DATA 2,54,56,44,29
120 DATA 32,243,245,125,198
130 DATA 26,111,241,12,21
140 DATA 32,230,201
200 FOR i=32256 TO 32383
210 READ x
220 POKE i,x
230 NEXT i
240 DATA 31,17,17,17,17,17,31,0
250 DATA 4,4,4,4,4,4,4,0
260 DATA 31,17,16,16,31,1,31,0
270 DATA 31,17,16,30,16,17,31,0
280 DATA 1,1,1,9,31,8,8,0
290 DATA 31,1,1,31,16,17,31,0
300 DATA 31,1,1,31,17,17,31,0
310 DATA 31,16,16,16,16,16,16,0
320 DATA 31,17,17,31,17,17,31,0
330 DATA 31,17,17,31,16,16,16,0
340 DATA 0,0,0,0,0,0,4,0
350 DATA 0,0,0,30,0,0,0,0
360 DATA 0,0,31,21,21,21,21,0
370 DATA 0,0,30,2,30,16,30,0
380 DATA 17,17,17,31,17,17,17,0
390 DATA 0,0,31,16,31,1,31,0
400 CLS : PRINT AT 1,5;"DIGITAL MULTIMETER"
410 PRINT AT 5,0;"Voltage is measured on channel 0"
420 PRINT AT 7,0;"Current is measured on channel 1"
430 PRINT AT 9,0;"Do you wish to display"
440 PRINT AT 11,0;"   POWER or RESISTANCE ?"
450 PRINT AT 15,0;"    Press P or R."
455 LET k$=INKEY$
460 IF k$<>"p" AND k$<>"r" THEN  GO TO 455
470 GO SUB 2000: REM display V and A
480 IF k$="p" THEN  GO SUB 2300
490 IF k$="r" THEN  GO SUB 2200
500 REM take readings
510 REM current
520 OUT 31,1
525 PAUSE 1
530 LET c=IN 31
540 LET current=c*0.01142/5
545 LET q=current
550 POKE 32001,89
560 GO SUB 1000
600 REM voltage
610 OUT 31,0
620 PAUSE 1
630 LET v=(IN 31)-c
632 LET v=IN 31
634 IF v>254 THEN  PRINT AT 4,10;"OUT OF RANGE": PAUSE 10: PRINT AT 4,10;"
     ": GO TO 500
635 LET voltage=v*0.01142
640 LET q=voltage
650 POKE 32001,88
```



```
660 GO SUB 1000
670 IF k$="p" THEN  GO TO 900
700 REM calculate resistance
705 IF current<0.00001 THEN  LET resistance=0: GO TO 720
710 LET resistance=voltage/current
720 LET q=resistance
750 POKE 32001,90
760 GO SUB 1000
780 IF INKEY$="" THEN  GO TO 500
790 GO TO 400
900 REM calculate power
910 LET power=voltage*current
920 LET q=power
950 POKE 32001,90
960 GO SUB 1000
980 IF INKEY$="" THEN  GO TO 500
990 GO TO 400
1000 REM sort and display digits
1010 LET dp=0
1020 IF q>=1 THEN  LET q=q/10: LET dp=dp+1: GO TO 1020
1050 FOR i=1 TO 3
1060 LET digit=INT (q*10)
1070 LET q=q*10-digit
1080 IF i<dp+1 THEN  LET g(i)=digit
1090 IF i>=dp+1 THEN  LET g(i+1)=digit
1100 NEXT i
1110 LET g(dp+1)=10
1200 FOR i=1 TO 4
1210 POKE 32002,g(i)
1230 POKE 32000,(6*(i-1))
1240 LET h=USR 32512
1250 NEXT i
1440 RETURN
2000 REM PRINT V
2005 CLS
2010 PRINT AT 0,25;"      "
2020 PRINT AT 1,25;"      "
2030 PRINT AT 2,25;"      "
2040 PRINT AT 3,25;"      "
2050 PRINT AT 4,25;"     "
2060 PRINT AT 5,25;"     "
2070 PRINT AT 6,25;"    "
2100 REM PRINT A
2110 PRINT AT  8,25;"    "
2120 PRINT AT  9,25;"     "
2130 PRINT AT 10,25;"     "
2140 PRINT AT 11,25;"      "
2150 PRINT AT 12,25;"      "
2160 PRINT AT 13,25;"      "
2170 PRINT AT 14,25;"      "
2180 RETURN
2200 REM PRINT OHMS
2210 PRINT AT 16,25;"     "
2220 PRINT AT 17,25;"      "
2230 PRINT AT 18,25;"      "
2240 PRINT AT 19,25;"      "
2250 PRINT AT 20,25;"     "
2260 PRINT AT 21,25;"      "
2280 RETURN
2300 REM PRINT W
2310 PRINT AT 16,25;"       "
```

```
2320 PRINT AT 17,25;"       "
2330 PRINT AT 18,25;"       "
2340 PRINT AT 19,25;"       "
2350 PRINT AT 20,25;"      "
2360 PRINT AT 21,25;"      "
2380 RETURN
5000 OUT 31,1: PRINT IN 31
5010 OUT 31,0: PRINT IN 31
5020 GO TO 5000
```

Program 21 CURRENT–VOLTAGE PLOT

```
  1 REM CURRENT-VOLTAGE CHARACTERISTICS
 10 CLS
 20 PLOT 10,0
 30 DRAW 0,165
 40 PLOT 0,16
 50 DRAW 255,0
 60 PRINT AT 21,20;"VOLTAGE"
 70 PRINT AT 20,0;"0    0.5   1.0   1.5   2.0   2.5"
 80 PRINT AT 0,3;"DIODE CHARACTERISTICS"
 90 PRINT AT 5,0;"C": PRINT "U": PRINT "R": PRINT "R": PRINT "E": PRINT "N": PR
INT "T"
100 REM RAMP THE OUTPUT VOLTAGE
105 PLOT 10,10
110 FOR x=10 TO 255
115 OUT 127,x
120 OUT 31,0
130 PAUSE 1
140 LET y=12+(IN 31)*2.5
145 IF y>175 THEN  GO TO 160
150 DRAW (x-PEEK 23677),(y-PEEK 23678)
160 NEXT x
170 IF INKEY$="" THEN  GO TO 170
180 GO TO 100
```

Program 22 FOUR-CHANNEL CHART RECORDER

```
  1 CLEAR 31999
100 FOR i=32000 TO 32078
110 READ x
120 POKE i,x
130 NEXT i
140 DATA 33,255,87,55,63,30,32,203,22,43
150 DATA 29,32,250,62,63,188,32,241,6,126
160 DATA 14,4,121,211,31,30,100,29,32,253
170 DATA 219,31,203,63,203,63,203,63,87,10,146,87
180 DATA 230,7,103,122,230,56,7,7,198,31
190 DATA 111,122,230,192,15,15,15,198,64,132
200 DATA 103,203,254,13,32,210,62,127,219,254
210 DATA 246,224,254,255,40,178,201
220 POKE 32257,64: REM channel 1 on top row
230 POKE 32258,100: REM channel 2 on second row
240 POKE 32259,136: REM channel 3 on third row
250 POKE 32260,175: REM channel 4 on bottom row
260 CLS
265 PRINT AT 0,3;"Multi-channel oscilloscope"
266 PRINT AT 8,21;"Channel 1"
267 PRINT AT 12,21;"Channel 2"
268 PRINT AT 16,21;"Channel 3"
269 PRINT AT 20,21;"Channel 4"
270 LET l=USR 32000
```


Program 23 MECHANICS DRILL

```
  10 REM MECHANICS DRILL
  20 REM by R.A.Sparkes
 100 CLS
 110 PRINT AT 1,7;"MECHANICS DRILL"
 120 PRINT AT 4,0;"This program tests your ability"
 130 PRINT : PRINT "to solve equations in mechanics."
 140 PRINT : PRINT "First I should like to know"
 150 PRINT : PRINT "your name. Type it in and then"
 160 PRINT : PRINT "press ENTER. If you make"
 170 PRINT : PRINT "a mistake, you can rub it out"
 180 PRINT : PRINT "with the DELETE key (top-right)."
 190 PRINT : PRINT "Press the CAPS-SHIFT key at the"
 200 PRINT : PRINT "same time as this DELETE key."
 210 INPUT a$
 290 REM what sort of question
 300 CLS
 310 PRINT AT 0,7;"MECHANICS DRILL"
 320 PRINT : PRINT "You can choose questions on"
 330 PRINT : PRINT "three different equations:"
 340 PRINT : PRINT "1.    s = u*t + a*t*t/2"
 360 PRINT : PRINT "                2   2"
 365 PRINT "2.    v = u  + 2*a*s"
 370 PRINT : PRINT : PRINT "3.    v = u + a*t"
 380 PRINT : PRINT : PRINT "Press one of these numbers"
 390 PRINT : PRINT "to make your choice."
 400 LET n$=INKEY$: IF n$<>"1" AND n$<>"2" AND n$<>"3" THEN  GO TO 400
 410 IF n$="1" THEN  GO TO 1000
 420 IF n$="2" THEN  GO TO 2000
 430 IF n$="3" THEN  GO TO 3000
1000  REM  s=u*t+a*t*t/2
1005 LET attempts=0
1016 LET u=1+INT (RND*20)
1017 LET t=1+INT (RND*10)
1018 LET a=1+INT (RND*20)
1019 REM ask same question again
1020 LET attempts=attempts+1
1030 CLS : PRINT "      s = u*t + a*t*t/2"
1040 PRINT AT 4,0;"What is the value of s"
1050 PRINT : PRINT "if  u  has the value ";u;" m/s"
1060 PRINT : PRINT "    t  has the value ";t;" s"
1070 PRINT : PRINT "    a  has the value ";a;" m/s2"
1075 IF attempts>3 THEN  LET r$="Press SPACE for another problem": PRINT : PRINT
 : PRINT : PRINT "This seems to be too difficult.": PRINT : PRINT "The correct a
nswer is  ";true;" m": LET correct=1: GO TO 1130
1080 PRINT : PRINT : PRINT "Give your answer as a number"
1090 PRINT : PRINT "of metres, then press ENTER."
1110 INPUT ans
1120 LET true=u*t+a*t*t/2
1125 IF ABS (ans-true)/true<0.01 THEN  GO SUB 5000
1126 IF ABS (ans-true)/true>=0.01 THEN  GO SUB 6000
1130 PRINT : PRINT r$
1150 PRINT : PRINT "Press Q for a different equation"
1170 LET y$=INKEY$: IF y$<>" " AND y$<>"q" AND y$<>"Q" THEN  GO TO 1170
1180 IF y$="Q" OR y$="q" THEN  GO TO 300
1190 IF correct=1 THEN  GO TO 1000
1200 GO TO 1020
2000  REM  v=v = u+u + 2*a*s
2005  LET attempts=0
2006 LET u=1+INT (RND*20)
2007 LET a=1+INT (RND*20)
```

```
2008 LET s=1+INT (RND*20)
2020 REM ask same question again
2025 LET attempts=attempts+1
2030 CLS : PRINT "          2     2"
2040 PRINT "         v   = u   + 2*a*s"
2050 PRINT AT 4,0;"What is the value of  v"
2055 PRINT : PRINT "if  u  has the value ";u;" m/s"
2060 PRINT : PRINT "    s  has the value ";s;" m"
2070 PRINT : PRINT "    a  has the value ";a;" m/s2"
2075 IF attempts>3 THEN  LET r$="Press SPACE for another problem": PRINT : PRINT
 : PRINT : PRINT "This seems to be too dificult.": PRINT : PRINT "The correct an
swer is ";true;" m/s": LET correct=1: GO TO 2130
2080 PRINT : PRINT : PRINT "Give your answer as a number"
2090 PRINT : PRINT "of m/s, then press ENTER."
2110 INPUT ans
2120 LET t$=STR$ (SQR (u*u+2*a*s))
2121 LET true=VAL (t$(1 TO 5))
2125 IF ABS (ans-true)/true<0.01 THEN  GO SUB 5000
2126 IF ABS (ans-true)/true>=0.01 THEN  GO SUB 6000
2130 PRINT : PRINT r$
2150 PRINT : PRINT "Press 0 for a different equation"
2170 LET y$=INKEY$: IF y$<>" " AND y$<>"q" AND y$<>"0" THEN  GO TO 2170
2180 IF y$="0" OR y$="q" THEN  GO TO 300
2190 IF correct=1 THEN  GO TO 2000
2200 GO TO 2020
3000  REM  v = u + a*t
3005 LET attempts=0
3006 LET u=1+INT (RND*20)
3007 LET a=1+INT (RND*20)
3008 LET t=1+INT (RND*25)
3020 REM ask same question again
3025 LET attempts=attempts+1
3030 CLS : PRINT AT 0,5;"     v = u + a*t"
3050 PRINT AT 4,0;"What is the value of  v"
3055 PRINT : PRINT "if  u  has the value ";u;" m/s"
3060 PRINT : PRINT "    a  has the value ";a;" m/s2"
3070 PRINT : PRINT "    t  has the value ";t;" s"
3075 IF attempts>3 THEN  LET r$="Press SPACE for another problem": PRINT : PRINT
 : PRINT : PRINT "This seems to be too dificult.": PRINT : PRINT "The correct an
swer is ";true;" m/s": LET correct=1: GO TO 3130
3080 PRINT : PRINT : PRINT "Give your answer as a number"
3090 PRINT : PRINT "of m/s, then press ENTER."
3110 INPUT ans
3120 LET true=u+a*t
3125 IF ABS (ans-true)/true<0.01 THEN  GO SUB 5000
3126 IF ABS (ans-true)/true>=0.01 THEN  GO SUB 6000
3130 PRINT : PRINT r$
3150 PRINT : PRINT "Press 0 for a different equation"
3170 LET y$=INKEY$: IF y$<>" " AND y$<>"q" AND y$<>"0" THEN  GO TO 3170
3180 IF y$="0" OR y$="q" THEN  GO TO 300
3190 IF correct=1 THEN  GO TO 3000
3200 GO TO 3020
5000 REM Correct answer
5010 PRINT : PRINT "Well done ";a$
5020 LET r$="Press SPACE for another problem"
5030 LET correct=1
5040 RETURN
6000 REM Wrong answer
6010 PRINT : PRINT "This is not good enough."
6020 LET r$="Press SPACE to try again."
6030 LET correct=0
6040 RETURN
```


Program 24 INTEGRATED SCIENCE TEST

```
   1 REM integrated science test
  20 DIM s(5): DIM r$(5)
  40 PRINT AT 10,0;"Please wait till graphic symbols"
  50 PRINT : PRINT "are constructed."
  60 GO SUB 3000: REM construct graphic symbols
 100 GO SUB 1000: REM call instructions
 110 FOR n=1 TO 5: LET s(n)=0: NEXT n
 115 LET total=0
 120 LET err=0: GO SUB 4000: REM call question 1
 130 LET err=0: GO SUB 5000: REM call question 2
 140 LET err=0: GO SUB 6000: REM call question 3
 150 LET err=0: GO SUB 7000: REM call question 4
 160 LET err=0: GO SUB 8000: REM call question 5
 170 GO TO 9000: REM call test score
1000 REM instructions
1010 CLS : PRINT "How to run this program."
1020 PRINT "nnnnnnnnnnnnnnnnnnnnnnnn"
1030 PRINT : PRINT "After each question there are"
1040 PRINT "five possible answers, labelled"
1050 PRINT "A, B, C, D and E."
1060 PRINT : PRINT "Choose the best answer and then"
1070 PRINT "press one of these letters"
1080 PRINT : PRINT "Press ""X"" for the first question"
1090 PAUSE 0: LET k$=INKEY$: IF k$<>"X" AND k$<>"x" THEN  GO TO 1090
1100 RETURN
1500 REM print, collect and mark responses
1510 PRINT AT 17,0;
1520 PRINT "A. Heat energy"
1530 PRINT "B. Chemical energy"
1540 PRINT "C. Movement energy"
1550 PRINT "D. Potential energy"
1560 PRINT "E. Electrical energy"
1570 PAUSE 0: LET k$=INKEY$: IF k$="A"  OR k$="B" OR k$="C" OR k$="D" OR k$="E"
OR k$="a" OR k$="b" OR k$="c" OR k$="d" OR k$="e" THEN  GO TO 1600
1580 REM response not allowed
1585 PRINT AT 17,20;"Press only"
1590 PRINT AT 18,20;"A,B,C,D or E"
1595 GO TO 1570
1600 REM check response
1610 IF r$(n)=k$ THEN  GO TO 1650
1620 REM wrong
1630 LET x$="Sorry, that is not right.": LET z$="Try again."
1640 LET s(n)=0: LET err=1: RETURN
1650 REM correct
1660 LET x$="Well done, that is correct.": LET z$=""
1670 IF err=0 THEN  LET s(n)=1
1680 LET err=0
1690 RETURN
1700 REM display result
1710 FOR i=11 TO 21: PRINT AT i,0;"                                        ": NEXT i
1720 PRINT AT 12,0;x$
1730 PRINT : PRINT y$
1740 PRINT : PRINT z$
1750 PRINT AT 21,10;"Press ""X"" to continue."
1760 PAUSE 0: LET k$=INKEY$: IF k$<>"X" AND k$<>"x" THEN  GO TO 1760
1770 RETURN
3000 REM line graphics
3010 FOR i=0 TO 20
3020  FOR j=0 TO 7
3030  READ row
```

```
3040 POKE USR CHR$ (i+144)+j,row
3050 NEXT j
3070 NEXT i
3080 RETURN
3100 DATA 0,0,0,255,255,0,0,0
3110 DATA 16,16,16,255,255,16,16,16
3120 DATA 16,16,16,16,16,16,16,16
3130 DATA 16,16,16,31,31,0,0,0
3140 DATA 16,16,16,240,240,0,0,0
3150 DATA 0,0,0,31,31,16,16,16
3160 DATA 0,0,0,240,240,16,16,16
3170 DATA 16,16,16,255,255,0,0,0
3180 DATA 0,0,0,255,255,16,16,16
3190 DATA 16,16,16,31,31,16,16,16
3200 DATA 16,16,16,240,240,16,16,16
3210 DATA 128,64,32,16,8,4,2,1
3220 DATA 1,2,4,8,16,32,64,128
3240 DATA 255,0,0,0,0,0,0,0
3250 DATA 0,0,0,0,0,0,0,255
3260 DATA 128,128,128,128,128,128,128,128
3270 DATA 1,1,1,1,1,1,1,1
3280 DATA 255,1,1,1,1,1,1,1
3290 DATA 1,1,1,1,1,1,1,255
3300 DATA 255,128,128,128,128,128,128,128
3310 DATA 128,128,128,128,128,128,128,255
4000 REM  question 1
4010 BORDER 7
4020 CLS
4040 PRINT "1.   faaaaaaaaaiaaaaaaag"
4050 PRINT "     c          c       c"
4060 PRINT "     c          c       c"
4070 PRINT "     jaaag     mOl      c"
4080 PRINT "     c   c              c"
4090 PRINT "     c  ~c           c  c"
4100 PRINT "     c   c  tnnnnr   c  c"
4110 PRINT "     c   c  p    qfak   c"
4120 PRINT "     c   c  p    qc c   c"
4130 PRINT "     c   c  p    qc c   c"
4140 PRINT "    nnnnnnnnnnnnnnnnnnnn"
4200 PRINT "A lamp gives out light energy"
4210 PRINT "and also another kind of energy."
4220 PRINT "Which one ?"
4230 LET n=1: LET r$(1)="a"
4240 GO SUB 1500: REM  print, collect and mark responses
4280 LET y$="A lamp gives out heat as well as light"
4290 GO SUB 1700: REM display result
4320 IF err=1 THEN  GO TO 4000
4340 RETURN
5000 REM  question 2
5010 CLS : PRINT "2.": INK 2
5030 PRINT AT 9,0;"                                "
5035 INK 5
5040 FOR i=0 TO 23
5050 PRINT AT 5,i;"       o"
5060 PRINT AT 6,i;"  MILK  i"
5070 PRINT AT 7,i;"        "
5080 PRINT AT 8,i;"  O    O"
5090 NEXT i
5095  INK 0
5100 PRINT AT 11,0;"What kind of energy does the"
5110 PRINT "engine give to the van ?"
5130 LET n=2: LET r$(2)="c"
```



```
5140 GO SUB 1500: REM  print, collect and mark responses
5150 LET y$="The engine moves the van along."
5160 GO SUB 1700: REM display result
5170 IF err=1 THEN  GO TO 5000
5180 RETURN
6000 REM  question 3
6010 CLS : PRINT "3.": INK 2
6030 PRINT AT 9,5;"                        "
6035 INK 0
6040 PRINT AT 2,7;"q   p"
6050 PRINT AT 3,7;"q   p"
6060 PRINT AT 4,7;"q";: BRIGHT 1: INK 6: PRINT "   ";: BRIGHT 0: INK 0: PRINT "p
"
6070 PRINT AT 5,7;"q";: BRIGHT 1: INK 6: PRINT "   ";: BRIGHT 0: INK 0: PRINT "p
"
6080 PRINT AT 6,7;"q";: BRIGHT 1: INK 6: PRINT "   ";: BRIGHT 0: INK 0: PRINT "p
"
6090 PRINT AT 7,7;"q";: BRIGHT 1: INK 6: PRINT "   ";: BRIGHT 0: INK 0: PRINT "p
   ";: INK 4: PRINT "      "
6100 PRINT AT 8,7;"q";: BRIGHT 1: INK 6: PRINT "   ";: BRIGHT 0: INK 0: PRINT "p
   ";: INK 4: PRINT "      "
6110 INK 0: PRINT AT 11,0;"What kind of energy is stored"
6120 PRINT "in food ?"
6130 LET n=3: LET r$(3)="b"
6140 GO SUB 1500: REM  print, collect and mark responses
6150 LET y$="Food is chemical energy.        We can store this in our bodies."

6160 GO SUB 1700: REM display result
6170 IF err=1 THEN  GO TO 6000
6180 RETURN
7000 REM  question 4
7010 CLS : PRINT "4."
7040 PRINT AT 5,10;"           "
7050 PRINT AT 6,10;"           "
7060 PRINT AT 7,10;"     A     "
7070 PRINT AT 8,10;"  SPARKES  "
7080 PRINT AT 9,10;"  BATTERY  "
7090 PRINT AT 10,10;"          "
7100 PRINT AT 11,0;"What kind of energy does a "
7110 PRINT "battery contain ?"
7130 LET n=4: LET r$(4)="b"
7140 GO SUB 1500: REM  print, collect and mark responses
7150 LET y$="A battery stores its energy as  chemical energy. This turns intoele
ctrical energy only if it is connected into a circuit."
7160 GO SUB 1700: REM display result
7170 IF err=1 THEN  GO TO 7000
7180 RETURN
8000 REM  question 5
8010 CLS : PRINT "5."
8020 PRINT : PRINT "nnnl"
8030 PRINT "    l"
8040 PRINT "     l"
8050 PRINT "      l"
8060 PRINT "       l"
8070 PRINT "        l"
8080 PRINT "         l"
8090 PRINT "          l"
8095 PRINT "           nnnnnnnnnnnnnnnnnnn"
8100 PRINT AT 1,0;"O": PAUSE 80: PRINT AT 1,0;" "
8110 PRINT AT 1,1;"O": PAUSE 24: PRINT AT 1,1;" "
8120 PRINT AT 1,2;"O": PAUSE 24: PRINT AT 1,2;" "
8130 PRINT AT 1,3;"O": PAUSE 24: PRINT AT 1,3;" "
```

```
8140 PRINT AT 2,4;"O": PAUSE 20: PRINT AT 2,4;" "
8150 PRINT AT 3,5;"O": PAUSE 16: PRINT AT 3,5;" "
8160 PRINT AT 4,6;"O": PAUSE 12: PRINT AT 4,6;" "
8170 PRINT AT 5,7;"O": PAUSE 9: PRINT AT 5,7;" "
8180 PRINT AT 6,8;"O": PAUSE 6: PRINT AT 6,8;" "
8190 PRINT AT 7,9;"O": PAUSE 4: PRINT AT 7,9;" "
8200 PRINT AT 8,10;"O": PAUSE 3: PRINT AT 8,10;" "
8210 FOR i=11 TO 30
8220 PRINT AT 9,i;"O": PAUSE 2: PRINT AT 9,i;" "
8230 NEXT i
8300 PRINT AT 11,0;"What kind of energy is the"
8310 PRINT "ball losing.                                    nnnnnn"
8330 LET n=5: LET r$(5)="d"
8340 GO SUB 1500: REM  print, collect and mark responses
8350 LET y$="The ball is falling, so it is   losing potential energy."
8360 GO SUB 1700: REM display result
8370 IF err=1 THEN  GO TO 8000
8470 RETURN
9000 REM display test score
9010 BORDER 3: CLS
9015 PRINT
9020 PRINT "Question number     First answer"
9030 FOR n=1 TO 5
9040 PRINT AT 3+n,7;n;: PRINT TAB 20;
9050 IF s(n)=0 THEN  PRINT "Wrong"
9060 IF s(n)=1 THEN  PRINT "Correct"
9065 LET total=total+s(n)
9070 NEXT n
9090 LET q$="s"
9100 IF total=1 THEN  LET q$=""
9110 PRINT AT 15,0;"You scored ";total;" correct answer";q$
9120 PRINT "out of 5 questions."
9200 PRINT AT 20,0;"Press ""Y"" to repeat the test,   any other key to finish."
9210 PAUSE 0: LET k$=INKEY$: IF k$="Y" OR k$="y" THEN  GO TO 110
9220 CLS : BORDER 7
```

Program 25 ELEMENTS

```
  1 REM ELEMENTS
  2 REM by R.A.Sparkes
 10 GO SUB 9000: REM Load data into array
 20 DIM p$(15)
 30 DIM d$(15)
 50 PRINT : PRINT AT 1,10;"ELEMENTS"
 51 PRINT : PRINT : PRINT "ELEMENTS is a guessing game."
 52 PRINT : PRINT "You type in the missing letters"
 53 PRINT : PRINT "one by one. Each correct letter"
 54 PRINT : PRINT "takes you nearer to guessing"
 55 PRINT : PRINT "the whole element. You can have"
 56 PRINT : PRINT "up to eight incorrect guesses"
 57 PRINT : PRINT "after which, you will be "
 58 PRINT : PRINT "given the correct answer."
 60 PRINT AT 21,0;"Press SPACE to continue."
 63 IF INKEY$<>" " THEN  GO TO 63
 64 CLS
 65 PRINT AT 1,10;"ELEMENTS"
 66 PRINT : PRINT : PRINT "Type your name,"
 67 PRINT : PRINT : PRINT "then press ENTER."
 68 PRINT : PRINT
 90 INPUT a$
100 REM set up word
110 GO SUB 8000: REM Get an element
```



```
120 LET wordlength=LEN (w$)
130 FOR i=1 TO wordlength
140 LET p$(i)=w$(i TO i)
150 LET d$(i)="-"
160 NEXT i
170 LET guess=0
180 CLS : PRINT AT 11,12;
190 REM Print letter positions
200 FOR n=1 TO wordlength
210 PRINT ;"-";
220 NEXT n
250 REM Ask question
260 PRINT AT 1,0;a$;","
270 PRINT AT 4,0;"Guess a letter."
280 PRINT AT 0,19;"wrong guesses"
290 PRINT AT 2,27;guess
330 LET l$=INKEY$: IF l$="" THEN  GO TO 330
333 IF CODE l$<97 THEN  LET l$=CHR$ (CODE l$+32)
340 IF CODE l$<97 OR CODE l$>122 THEN  GO TO 330
360 LET flag=0
370 FOR i=1 TO wordlength
380 IF l$<>p$(i) THEN  GO TO 400
390 LET flag=1
395 LET d$(i)=l$
400 NEXT i
410 REM Construct word so far
415 LET g$=""
420 FOR i=1 TO wordlength
430 LET g$=g$+d$(i)
440 NEXT i
500 PRINT AT 11,0;"The word is ";g$
520 IF flag=0 THEN  PRINT AT 15,0;"Your letter is not in my word,": PRINT : PRI
NT "try again."
530 IF flag=1 THEN  PRINT AT 15,0;"                                    ": PRINT : PR
INT"                "
540 IF g$=w$ THEN  GO TO 800
550 IF flag=0 THEN  LET guess=guess+1
560 IF guess>8 THEN  GO TO 880
580 GO TO 270
800 REM Success
810 PRINT AT 1,0;"Well done, ";a$
820 PRINT AT 4,0;"the hidden element is"
830 PRINT AT 6,0;w$
850 PRINT AT 20,0;"Press SPACE for another word."
860 IF INKEY$<>" " THEN  GO TO 860
870 GO TO 100
880 REM too many guesses
890 CLS
900 PRINT AT 1,0;"No, ";a$
910 PRINT AT 4,0;"the hidden element"
920 PRINT AT 8,0;w$
930 PRINT AT 20,0;"Press SPACE for another word."
940 IF INKEY$<>" " THEN  GO TO 940
950 GO TO 100
8000 REM Get an element
8010 LET r=INT (1+RND$103)
8020 IF e$(r)="                " THEN  GO TO 8010
8030 LET w$=""
8050 FOR i=1 TO 15
8060 LET q$=e$(r)(i TO i)
8070 IF q$<>" " THEN  LET w$=w$+q$
8080 NEXT i
```

```
8090 LET e$(r)=""
8100 RETURN
9000 REM Load data into array
9010 DATA "actinium","aluminium","americium","antimony","argon"
9020 DATA "arsenic","astatine","barium","berkelium","beryllium"
9030 DATA "bismuth","boron","bromine","cadmium","caesium"
9040 DATA "calcium","californium","carbon","cerium","chlorine"
9050 DATA "chromium","cobalt","copper","curium","dysprosium"
9060 DATA "einsteinium","erbium","europium","fermium","fluorine"
9070 DATA "francium","gadolinium","gallium","germanium","gold"
9080 DATA "hafnium","helium","holmium","hydrogen","indium"
9090 DATA "iodine","iridium","iron","krypton","lanthanum"
9100 DATA "lawrencium","lead","lithium","lutetium","magnesium"
9110 DATA "manganese","medelevium","mercury","molybdenum","neodymium"
9120 DATA "neon","neptunium","nickel","niobium","nitrogen"
9130 DATA "nobelium","osmium","oxygen","palladium","phosphorus"
9140 DATA "platinum","plutonium","polonium","potassium","praseodymium"
9150 DATA "promethium","protactinium","radium","radon","rhenium"
9160 DATA "rhodium","rubidium","ruthenium","samarium","scandium"
9170 DATA "selenium","silicon","silver","sodium","strontium"
9180 DATA "sulphur","tantalum","technetium","tellurium","terbium"
9190 DATA "thallium","thorium","thulium","tin","titanium"
9200 DATA "tungsten","uranium","vanadium","xenon","ytterbium"
9210 DATA "yttrium","zinc","zirconium"
9300 RESTORE
9310 DIM e$(103,15)
9320 FOR n=1 TO 103
9330 READ e$(n)
9340 NEXT n
9500 RETURN
```

Program 26 CHEMICAL NAMES

```
 100 REM Introduction
 110 CLS
 120 PRINT "Hello."
 130 PRINT
 140 PRINT "I should like to know your name"
 150 PRINT "Please type it in."
 160 PRINT : PRINT "If you make a mistake,"
 170 PRINT "you can rub it out again"
 180 PRINT "with the DELETE key (top-right)."
 190 PRINT "Hold down the CAPS SHIFT key"
 200 PRINT "(lower left) at the same time"
 210 PRINT "and press the DELETE key for"
 220 PRINT "each leter you want to rub out."
 230 PRINT : PRINT "When you are satisfied, press"
 240 PRINT "the ENTER key (extreme right)"
 250 PRINT "to tell me that you are ready."
 260 INPUT AT 1,0;"Type here "; LINE n$
 270 IF n$<>"" THEN  GO TO 300
 280 INPUT AT 0,0;"Come on! You must have some name";AT 1,0;"Type it here "; LIN
E n$
 290 GO TO 270
 300 CLS
 310 PRINT "Hello ";n$
 330 PRINT : PRINT "I shall give you the chemical"
 340 PRINT "symbol for one of the elements."
 350 PRINT "You must type out the full name"
 360 PRINT "of that element and I will"
 370 PRINT "tell you if you are right."
 380 PRINT : PRINT "Remember that you can rub out"
```



```
 390 PRINT "any letters you type wrongly and"
 400 PRINT "don't forget to press ENTER to"
 410 PRINT "tell me that you are ready."
 420 PRINT AT 20,0;"Press ""C"" to get a question."
 440 PAUSE 0
 450 IF INKEY$<>"c" AND INKEY$<>"C"  THEN  GO TO 450
 460 RESTORE
 500 REM Question routine
 510 LET x=1+INT (RND*24.99)
 520 FOR i=1 TO x: READ s$: READ e$: NEXT i
 530 LET guesses=0
 535 PRINT AT 20,0;"
  "
 540 PRINT AT 19,0;"The symbol is ";s$
 550 PRINT : PRINT "What is the name of the element?"
 570 INPUT AT 1,0;"Type here   ";g$
 580 REM convert to lower case
 590  LET w$=""
 600 FOR i=1 TO LEN g$
 610 LET l$=g$(i)
 620 LET v=CODE l$
 630 IF v<97 THEN  LET v=v+32
 640 LET l$=CHR$ v
 650 LET w$=w$+l$
 660 NEXT i
 670 REM Check answer
 680 IF w$=e$ THEN  GO TO 800
 690 REM Wrong answer
 700 LET guesses=guesses+1: IF guesses>3 THEN  GO TO 900
 710 CLS
 720 PRINT "No, ";n$
 725 PRINT
 730 PRINT "That is not right."
 740 PRINT : PRINT "Perhaps the spelling is wrong."
 750 PRINT : PRINT "Press ""C"" to try again."
 760 PAUSE 0: IF INKEY$<>"c" AND INKEY$<>"C" THEN  GO TO 760
 770 GO TO 535
 800 REM  Correct answer
 810  CLS : PRINT "Well done, that is correct."
 820 PRINT : PRINT "Would you like another ?"
 825 PRINT
 830 PRINT "Press 'Y' for YES or 'N' for NO."
 840 PAUSE 0
 850 IF INKEY$="y" OR INKEY$="Y" THEN  GO TO 300
 860 IF INKEY$="n" OR INKEY$="N" THEN  STOP
 870 GO TO 840
 900 REM Three wrong guesses
 910 CLS : PRINT "You don't seem to know this one,": PRINT : PRINT n$;"."
 920 PRINT : PRINT "The name for ";s$;" is ";e$
 940 PRINT : PRINT "Do you see where you went wrong?"
 950 GO TO 820
1000 REM symbols and names
1010 DATA "Cu","copper"
1020 DATA "Sn","tin"
1030 DATA "Au","gold"
1040 DATA "S","sulphur"
1050 DATA "H","hydrogen"
1060 DATA "I","iodine"
1070 DATA "Ag","silver"
1080 DATA "Pb","lead"
1090 DATA "Cd","cadmium"
1100 DATA "B","boron"
```

```
1110 DATA "C","carbon"
1120 DATA "Cl","chlorine"
1130 DATA "K","potassium"
1140 DATA "Na","sodium"
1150 DATA "He","helium"
1160 DATA "Li","lithium"
1170 DATA "A","argon"
1180 DATA "Ne","neon"
1190 DATA "Co","cobalt"
1200 DATA "Si","silicon"
1210 DATA "Ge","germanium"
1220 DATA "As","arsenic"
1230 DATA "Se","selenium"
1240 DATA "O","oxygen"
1250 DATA "Mg","magnesium"
```

Program 27 ANALOGUE–DIGITAL SIMULATION

```
  1 CLEAR 31500
  3 PRINT AT 10,5;"Loading, please wait."
100 REM DEFINE OUTLINES
110 FOR j=97 TO 107
120 FOR i=0 TO 7
121 READ x
122 POKE USR (CHR$ j)+i,x
123 NEXT i
124 NEXT j
130 DATA 85,0,0,0,0,0,0,0
140 DATA 0,128,0,128,0,128,0,128
145 DATA 1,0,1,0,1,0,1,0
150 DATA 0,0,0,0,0,0,0,85
160 DATA 85,0,128,0,128,0,128,0
170 DATA 85,0,1,0,1,0,1,0
180 DATA 128,0,128,0,128,0,128,85
190 DATA 1,0,1,0,1,0,1,85
200 DATA 0,0,0,0,0,0,0,85
210 DATA 0,0,0,255,255,0,0,0
220 DATA 0,0,0,255,255,128,128,128
400 GO SUB 800
500 BORDER 6
550 CLS
560 PRINT AT 4,0;"128 64  32  16   8   4   2   1"
570 PRINT AT 1,0;"eaf eaf eaf eaf eaf eaf eaf eaf"
580 PRINT AT 2,0;"b c b c b c b c b c b c b c b c"
590 PRINT AT 3,0;"gih gih gih gih gih gih gih gih"
600 PRINT AT 18,0;"0   40   80  120  160  200  240"
610 PRINT AT 19,0;"kjjjjkjjjjkjjjjkjjjjkjjjjkjjjjk"
620 FOR n=0 TO 255
630 POKE 32512,n
640 RANDOMIZE USR 32100
650 PAUSE 5
660 NEXT n
700 INPUT "Enter a number (0 to 255) ";number
710 IF number<0 OR number>255 THEN  GO TO 700
730 POKE 32512,number
740 RANDOMIZE USR 32100
750 GO TO 700
800 FOR i=31744 TO 31831
810 READ x
820 POKE i,x
830 NEXT i
```



```
850 DATA 31,17,17,17,17,17,31,0
851 DATA 4,4,4,4,4,4,4,0
852 DATA 31,17,16,16,31,1,31,0
853 DATA 31,17,16,30,16,17,31,0
854 DATA 1,1,1,9,31,8,8,0
855 DATA 31,1,1,31,16,17,31,0
856 DATA 31,1,1,31,17,17,31,0
857 DATA 31,16,16,16,16,16,16,0
858 DATA 31,17,17,31,17,17,31,0
859 DATA 31,17,17,31,16,16,16,0
860 DATA 0,0,0,0,0,0,0,0
900 FOR i=32000 TO 32058
910 READ x
920 POKE i,x
930 NEXT i
950 DATA 24,89,5,0,0,0,0,0,0
951 DATA 0,0,0,0,0,0,42,0,125,58
952 DATA 2,125,55,63,23,23,23,79,6,124
953 DATA 22,8,10,30,6,31,48,4,54,0
954 DATA 24,2,54,56,44,29,32,243,245,125
955 DATA 198,26,111,241,12,21,32,230,201
1000 FOR i=32100 TO 32409
1010 READ x
1020 POKE i,x
1030 NEXT i
1100 DATA 33,4,127,62,0,50,3,127,50,4
1110 DATA 127,58,0,127,254,100,56,5,214,100
1120 DATA 52,24,247,45,254,10,56,5,214,10
1130 DATA 52,24,247,45,119,62,0,50,5,127
1140 DATA 58,4,127,50,2,125,167,40,7,62
1150 DATA 1,50,5,127,24,5,62,10,50,2
1160 DATA 125,62,12,50,0,125,62,89,50,1
1170 DATA 125,205,16,125,58,3,127,50,2,125
1180 DATA 167,32,11,58,5,127,167,32,5,62
1190 DATA 10,50,2,125,62,18,50,0,125,62
1200 DATA 89,50,1,125,205,16,125,58,2,127
1210 DATA 50,2,125,62,24,50,0,125,62,89
1220 DATA 50,1,125,205,16,125,0,0,0,0
1230 DATA 0,0,58,0,127,71,14,0,33,32
1240 DATA 88,121,7,7,95,22,0,25,120,23
1250 DATA 71,48,4,62,0,24,2,62,56,119
1260 DATA 44,119,44,119,44,44,44,44,44,44
1270 DATA 44,44,44,44,44,44,44,44,44,44
1280 DATA 44,44,44,44,44,44,44,44,44,44
1290 DATA 44,44,44,44,44,119,44,119,44,119,44
1300 DATA 44,44,44,44,44,44,44,44,44,44
1310 DATA 44,44,44,44,44,44,44,44,44,44
1320 DATA 44,44,44,44,44,44,44,44,44,119
1330 DATA 44,119,44,119,12,62,8,185,32,154
1340 DATA 0,0,17,160,80,33,160,90,58,0
1350 DATA 127,254,8,56,8,214,8,54,0,44
1360 DATA 28,24,244,54,56,58,0,127,230,7
1370 DATA 71,40,6,62,0,55,31,16,252,18
1380 DATA 20,18,20,18,20,18,20,18,20,18
1390 DATA 20,18,20,18,44,62,191,189,48,1
1400 DATA 201,54,63,24,245,0,0,0,0,0
1500 RETURN
```

Program 28 BYTE SIMULATION

```
 10 REM Bistable simulation
 20 REM by R.A.Sparkes
 30 REM 12/1/83
 40 REM
100 REM ##################
110 REM
120 REM  Initialize variables
130 REM
140 REM ##################
150 REM
160 DIM b(8): REM bits of the byte
170 GO SUB 9000: REM Define graphics characters
180 LET byte=0: REM initial value
200 REM ##################
210 REM
220 REM   Main program
230 REM
240 REM ##################
250 REM
260 GO SUB 1000: REM draw bistable
270 GO SUB 2000: REM Accept an input
280  GO SUB option: REM Execute chosen option
290 GO TO 270: REM Repeat sequence
1000 REM ##################
1010 REM
1020 REM  Draw bistable
1030 REM
1040 REM ##################
1050 REM
1060 CLS
1070 PRINT AT 1,12;"BYTE 32500"
1090 PRINT AT 4,8;"eababababababababaf"
1100 PRINT AT 5,8;"d d d d d d d d"
1110 PRINT AT 6,8;"gacacacacacacacah"
1120 RETURN
1500 REM ##################
1510 REM
1520 REM Display byte in
1530 REM binary and decimal
1540 REM
1550 REM ##################
1560 REM
1570 LET temp=byte
1580 FOR i=8 TO 1 STEP -1
1590 LET bitvalue=2^(i-1)
1600 LET b(i)=(temp>=bitvalue)
1610 IF b(i) THEN  LET temp=temp-bitvalue
1620 NEXT i
1700 PRINT AT 14,0;"       Decimal value = ";byte;"        "
1710 FOR i=1 TO 8
1720 PRINT AT 5,(25-2*i);b(i)
1730 NEXT i
1740 RETURN
2000 REM ##################
2010 REM
2020 REM  Accept an input
2030 REM
2040 REM ##################
2050 REM
2060 PRINT AT 16,0;"Press  P to enter a new number  "
```



```
2070 PRINT AT 17,7;"A to add a number"
2080 PRINT AT 18,7;"S to subtract a number"
2090 PRINT AT 19,7;"L to shift left"
2100 PRINT AT 20,7;"R to shift right"
2110 PRINT AT 21,7;"F to finish"
2200 LET a$=INKEY$
2210 IF a$<>"p" AND a$<>"P" AND a$<>"a" AND a$<>"A" AND a$<>"s" AND a$<>"S" AND
a$<>"l" AND a$<>"L" AND a$<>"r" AND a$<>"R" AND a$<>"f" AND a$<>"F" THEN  GO TO
2200
2220 PRINT AT 16,0;"                                "
2230 PRINT AT 17,0;"                                "
2240 PRINT AT 18,0;"                                "
2250 PRINT AT 19,0;"                                "
2260 PRINT AT 20,0;"                                "
2270 PRINT AT 21,0;"                                "
2280  REM This clears the option choice
2290 REM Now determine where the chosen option can be found
2300 IF a$="p" OR a$="P" THEN  LET option=3000
2310 IF a$="a" OR a$="A" THEN  LET option=4000
2320 IF a$="s" OR a$="S" THEN  LET option=5000
2330 IF a$="l" OR a$="L" THEN  LET option=6000
2340 IF a$="r" OR a$="R" THEN  LET option=7000
2350 IF a$="f" OR a$="F" THEN  LET option=8000
2360 RETURN
2500 REM ##################
2510 REM
2520 REM  Collect number
2530 REM  as an integer
2540 REM  between 0 and 255
2550 REM
2560 REM ##################
2570 REM
2580 PRINT AT 16,9;"what number (0 to 255)?"
2590 INPUT n$
2600 IF CODE n$>57 OR CODE n$<48 THEN  GO TO 2590
2610 LET number=INT VAL n$
2620 IF number<0 OR number>255 THEN  GO TO 2590
2630 RETURN
3000 REM ##################
3010 REM
3020 REM  Poke number into byte
3030 REM
3040 REM ##################
3050 REM
3060 PRINT AT 14,0;"POKE address with"
3070 GO SUB 2500: REM Collect number
3080 LET byte=number
3090 PRINT AT 14,0;"        POKE 32500,";byte;"          "
3093 PRINT AT 16,0;"                                "
3095 PAUSE 100
3100 GO SUB 1500: REM Display number
3110 RETURN
4000 REM ##################
4010 REM
4020 REM  Add a number
4030 REM
4040 REM ##################
4050 REM
4070 PRINT AT 16,5;"ADD"
4080 GO SUB 2500: REM Collect number
4090 LET byte=byte+number
4100 IF byte>255 THEN  LET byte=byte-256
```

```
4110 PRINT AT 16,5;"ADD ";number;"
4120 GO SUB 1500: REM Display number
4130 RETURN
5000 REM ******************
5010 REM
5020 REM  Subtract a number
5030 REM
5040 REM ******************
5050 REM
5070 PRINT AT 16,0;"SUBTRACT"
5080 GO SUB 2500: REM Collect number
5090 LET byte=byte-number
5100 IF byte<0 THEN  LET byte=byte+256
5110 PRINT AT 16,0;"SUBTRACT ";number;"
5120 GO SUB 1500: REM Display number
5130 RETURN
6000 REM ******************
6010 REM
6020 REM  Shift left
6030 REM
6040 REM ******************
6050 REM
6060 PRINT AT 16,8;"SHIFT LEFT"
6070 LET byte=byte+byte
6080 IF byte>255 THEN  LET byte=byte-256
6090 GO SUB 1500: REM Display number
6100 RETURN
7000 REM ******************
7010 REM
7020 REM  Shift right
7030 REM
7040 REM ******************
7050 REM
7060 PRINT AT 16,8;"SHIFT RIGHT"
7070 LET byte=INT (byte/2)
7080 GO SUB 1500: REM Display number
7090 RETURN
8000 REM ******************
8010 REM
8020 REM  Finish
8030 REM
8040 REM ******************
8050 REM
8060 STOP
9000 REM ******************
9010 REM
9020 REM  Define graphics chars
9030 REM
9040 REM ******************
9050 REM
9100 FOR i=0 TO 7
9110 FOR j=0 TO 7
9120 READ row
9130 POKE USR CHR$ (i+144)+j,row
9140 NEXT j
9150 NEXT i
9200 REM Defining characters
9210 REM Letter A = horiz line
9220 DATA 0,0,0,255,255,0,0,0
9230 REM Letter B = T-junction down
9240 DATA 0,0,0,255,255,24,24,24
9250 REM Letter C = T-junction up
```



```
9260 DATA 24,24,24,255,255,0,0,0
9270 REM Letter D = Vertical line
9280 DATA 24,24,24,24,24,24,24,24
9290 REM Letter E = Top-left corner
9300 DATA 0,0,0,31,31,24,24,24
9310 REM Letter F = Top-right corner
9320 DATA 0,0,0,248,248,24,24,24
9330 REM Letter G = Bottom-left corner
9340 DATA 24,24,24,31,31,0,0,0
9350 REM Letter H = Bottom-right corner
9360 DATA 24,24,24,248,248,0,0,0
9400 RETURN
```

Program 29 RADIOACTIVE DECAY

```
  1 REM radioactive decay
  5 DIM n(8,32)
 10 CLS
 20 REM Set up molecules
 30 FOR y=0 TO 6
 40 FOR x=0 TO 31
 50 PRINT AT y,x;CHR$ 79
 55 LET n(y+1,x+1)=1
 60 NEXT x
 70 NEXT y
 80 PLOT 0,10: DRAW 255,0
 90 PLOT 24,0: DRAW 0,115
100 PRINT AT 20,2;"0"
110 PRINT AT 17,1;"50"
120 PRINT AT 14,0;"100"
130 PRINT AT 11,0;"150"
140 PRINT AT 8,0;"200"
150 PRINT AT 21,3;"0  5 10 15 20 25 30 35 40 t/s";
160 LET count=224
170 PLOT 24,110
180 FOR x=24 TO 255 STEP 0.45
190 REM Let nuclei decay at random
200 LET xpos=INT (32*RND)
210 LET ypos=INT (7*RND)
220 IF n(ypos+1,xpos+1)=1 THEN  GO SUB 1000
300 DRAW x-PEEK 23677,(10+count/2)-PEEK 23678
310 NEXT x
320 PRINT AT 0,0;"Plot finished                    "
330 STOP
1000 REM A nucleus has decayed
1010 LET n(ypos+1,xpos+1)=0
1020 PRINT AT ypos,xpos;CHR$ 42
1030 LET count=count-1
1040 RETURN
```

Program 30 SUM OF TWO DICE

```
100 REM Sum of two dice
110 PRINT AT 0,6;"THE SUM OF TWO DICE"
150 DIM s(12)
160 PRINT AT 2,0;"TOTAL NUMBER OF THROWS = "
170 PLOT 4,0: DRAW 0,150
180 PRINT AT 5,0;"120"
190 PRINT AT 10,0;"80"
200 PRINT AT 15,0;"40"
```

```
 210 PRINT AT 20,0;"0"
 220 PLOT 0,10: DRAW 255,0
 300 PRINT AT 21,1;"2  3  4  5  6  7  8  9 10 11 12";
 500 RANDOMIZE
1000 LET total=0
1010 FOR i=2 TO 12: LET s(i)=0: NEXT i
1020 REM shake dice and add them up
1050 LET dice1=INT (1+(6*RND))
1060 LET dice2=INT (1+(6*RND))
1070 LET sum=dice1+dice2
1080 FOR i=2 TO 12
1090 IF sum=i THEN  LET s(i)=s(i)+1: GO SUB 2000
1100 NEXT i
1110 LET total=total+1
1120 IF total=800 THEN  STOP
1130 PRINT AT 2,26;total
1140 GO TO 1020
2000 REM update bar chart
2010  LET x=(i-2)*24+8
2020 LET y=s(i)+10
2040 PLOT x,y
2050 DRAW 7,0
2070 RETURN
```

Program 31 STANDING WAVES

```
  1 CLEAR 28927
  2 PRINT AT 10,0;"Loading data, please wait"
 10 FOR i=28928 TO 29097
 20 READ x
 30 POKE i,x
 40 NEXT i
 50 DATA 205,32,113,198,134
 60 DATA 50,9,113,203,0
 70 DATA 201,0,0,0,0
 80 DATA 0,205,32,113,198
 90 DATA 198,50,25,113,203
100 DATA 0,201,0,0,0
110 DATA 0,0,122,230,7
120 DATA 103,122,230,56,7
130 DATA 7,111,122,230,192
140 DATA 15,15,15,198,64
150 DATA 132,103,121,230,248
160 DATA 15,15,15,133,111
170 DATA 121,47,230,7,7
180 DATA 7,7,201,0,0
190 DATA 0,0,243,14,0
200 DATA 6,115,10,87,205
210 DATA 0,113,38,118,58
220 DATA 70,113,129,111,126
230 DATA 198,89,87,2,205
240 DATA 16,113,4,10,87
250 DATA 205,0,113,38,118
260 DATA 58,70,113,95,121
270 DATA 147,111,126,198,40
280 DATA 87,2,205,16,113
290 DATA 4,10,87,205,0
300 DATA 113,5,10,95,5
310 DATA 10,198,12,131,4
320 DATA 4,2,87,205,16
330 DATA 113,12,12,194,75
```



```
340 DATA 113,58,70,113,61
350 DATA 50,70,113,62,254
360 DATA 219,254,246,224,254
370 DATA 255,202,73,113,254
380 DATA 254,40,241,251,201
500 REM Set up wave table
510 FOR i=0 TO 255
520 POKE (i+30208),(15*SIN (i*PI/32))
530 NEXT i
550 CLS
600 RANDOMIZE USR 29000
```

Program 32 LONGITUDINAL WAVES

```
  1 REM LONGITUDINAL PULSES
  2 CLEAR 28671
 10 PRINT AT 10,0;"Loading data, please wait"
100 FOR i=29696 TO 29927
110 READ x
120 POKE i,x
130 NEXT i
140 FOR i=0 TO 127
145 POKE 29184+i,9*SIN (i*PI/64)
150 NEXT i
200 DATA 243,62,254,219,254,246,224,254,255,40
210 DATA 28,254,254,40,242,254,253,40,2,251
220 DATA 201,58,128,114,60,230,127,50,128,114
230 DATA 111,38,114,126,50,0,112,24,19,58
240 DATA 128,114,167,40,4,254,64,32,228,62
250 DATA 0,0,50,128,114,50,0,112,33,255
260 DATA 112,17,0,113,1,0,1,237,184,58
270 DATA 0,113,95,58,129,114,87,6,8,62
280 DATA 0,135,203,19,48,1,130,16,248,203
290 DATA 47,50,0,114,33,1,113,17,0,113
300 DATA 1,0,1,237,176,14,16,6,115,10
310 DATA 87,205,140,116,6,112,10,87,4,10
320 DATA 130,129,6,115,2,87,205,190,116,62
330 DATA 16,129,79,254,240,32,226,195,1,116
340 DATA 38,80,122,230,248,15,15,15,111,122
350 DATA 47,230,7,7,7,7,198,134,50,162
360 DATA 116,203,0,125,198,32,111,254,128,56
370 DATA 246,198,128,111,36,36,62,88,188,32
375 DATA 236,201,0,0,0,0,0,0,0,0
380 DATA 38,80,122,230,248,15,15,15,111,122
390 DATA 47,230,7,7,7,7,198,198,50,212
400 DATA 116,203,0,125,198,32,111,254,128,56
410 DATA 246,198,128,111,36,36,62,88,188,32,236,201
500 REM Collect reflection coefficient
510 CLS
520 PRINT AT 0,8;"PULSE REFLECTION"
530 PRINT AT 5,0;"Enter the reflection coefficient"
540 PRINT AT 7,0;"as 1, 0.5, 0, -0.5 or -1"
550 INPUT n
560 LET refcof=-INT (n*2+.1)
570 POKE 29313,refcof
580 PRINT AT 10,0;"Resetting wave table"
600 REM Reset wave tables
610 FOR i=28672 TO 29183
620 POKE i,0
630 NEXT i
1000 REM main program
1010 CLS
```

```
1020 PRINT AT 0,8;"Pulse reflection"
1030 PRINT AT 2,0;"Press Z to send a single pulse"
1040 PRINT AT 4,0;"Hold Z to make continuous waves"
1050 PRINT AT 6,0;"and CAPS-SHIFT to stop the pulse"
1060 PRINT AT 8,0;"Press X to change the reflectioncoefficient"
1070 PRINT AT 11,0;"The reflection coefficient"
1080 PRINT AT 13,0;"is ";n;" at present"
1200 LET l=USR 29696
1300 GO TO 500
```

```
   1 CLEAR 63999
   3 PRINT AT 10,0;"Loading data, please wait."
  10 DIM t(4,2): REM stores for time intervals
  46 FOR i=64000 TO 64184
  47 READ x
  48 POKE i,x
  49 NEXT i
  50 DATA 243,6,0,221,33,0,251,221,54,0
  51 DATA 0,221,35,16,248,22,0,30,0,46
  52 DATA 0,62,252,50,243,250,62,124,50,245,250,62
  53 DATA 251,50,244,250,50,246,250,219,63,230
  54 DATA 3,71,219,63,230,3,79,184,40,248
  55 DATA 0,121,168,65,254,1,40,8,254,2
  56 DATA 40,18,121,238,2,71,58,243,250,198
  57 DATA 4,50,243,250,221,42,243,250,24,12
  58 DATA 58,245,250,198,4,50,245,250,221,42
  59 DATA 245,250,221,115,0,221,114,1,221,117
```

Program 33 MOLECULAR MOTION – BASIC

```
   2 RANDOMIZE
  10 GO SUB 5000
  20 GO SUB 3000
  30 GO SUB 2000
 100 POKE pos,56
 120 GO SUB 1000: REM Get new position
 130 LET l$=INKEY$
 200 IF l$="d" THEN  LET direction=1
 210 IF l$="a" THEN  LET direction=-1
 220 IF l$="x" THEN  LET direction=32
 230 IF l$="w" THEN  LET direction=-32
 240 IF l$="e" THEN  LET direction=-31
 250 IF l$="c" THEN  LET direction=33
 260 IF l$="z" THEN  LET direction=31
 270 IF l$="q" THEN  LET direction=-33
 280 GO TO 100
1000 REM Get new position
1010 LET newpos=pos+direction
1020 LET screen=PEEK newpos
1030 IF screen=63 THEN  GO TO 1300
1040 IF screen=0 THEN  GO TO 1500
1050 IF screen=9 THEN  GO TO 1600
1300 POKE pos,63
1310 LET pos=newpos
1330 RETURN
1500 REM Horizontal wall
1510 IF direction=31 THEN  LET direction=-33: RETURN
1520 IF direction=32 THEN  LET direction=-32: RETURN
1530 IF direction=33 THEN  LET direction=-31: RETURN
1540 IF direction=-33 THEN  LET direction=31: RETURN
1550 IF direction=-32 THEN  LET direction=32: RETURN
1560 IF direction=-31 THEN  LET direction=33: RETURN
```



```
1600 REM Vertical wall
1610 IF direction=-31 THEN  LET direction=-33: RETURN
1620 IF direction=-33 THEN  LET direction=-31: RETURN
1630 IF direction=33 THEN  LET direction=31: RETURN
1640 IF direction=31 THEN  LET direction=33: RETURN
1650 IF direction=1 THEN  LET direction=-1: RETURN
1660 IF direction=-1 THEN  LET direction=1: RETURN
2110 LET pos=22785+INT (29*RND)+32*INT (13*RND)
2160 LET r=INT (RND*8)
2200 IF r=0 THEN  LET direction=1
2210 IF r=1 THEN  LET direction=33
2220 IF r=2 THEN  LET direction=32
2230 IF r=3 THEN  LET direction=31
2240 IF r=4 THEN  LET direction=-1
2250 IF r=5 THEN  LET direction=-31
2260 IF r=6 THEN  LET direction=-32
2270 IF r=7 THEN  LET direction=-33
2400 RETURN
3000 REM Do display
3010 CLS
3015 PRINT AT 0,8;"Molecular motion"
3020 PRINT AT 2,0;"q  w  e       Press these"
3025 PRINT AT 3,1;"a b c"
3030 PRINT AT 4,0;"ad o ed       keys to move in"
3040 PRINT AT 5,1;"f g h"
3045 PRINT AT 6,0;"z  x  c       each direction"
3060 INK 0: PAPER 0
3070 PRINT AT 7,0;"                                "
3080 PRINT AT 21,0;"                                "
3090 INK 1: PAPER 1
3100 FOR i=8 TO 20
3110 PRINT AT i,0;" "
3120 PRINT AT i,31;" "
3130 NEXT i
3140 INK 7: PAPER 7
3150 FOR i=8 TO 20
3160 PRINT AT i,1;"oooooooooooooooooooooooooooooo"
3170 NEXT i
3180  INK 0
3200 RETURN
5000 REM define arrow characters
5010 REM A is a
5011 POKE USR "a"+0,224
5012 POKE USR "a"+1,192
5013 POKE USR "a"+2,160
5014 POKE USR "a"+3,16
5015 POKE USR "a"+4,8
5016 POKE USR "a"+5,4
5017 POKE USR "a"+6,2
5018 POKE USR "a"+7,1
5020 REM B is b
5021 POKE USR "b"+0,16
5022 POKE USR "b"+1,56
5023 POKE USR "b"+2,84
5024 POKE USR "b"+3,16
5025 POKE USR "b"+4,16
5026 POKE USR "b"+5,16
5027 POKE USR "b"+6,16
5028 POKE USR "b"+7,16
5030 REM C is c
5031 POKE USR "c"+0,7
5032 POKE USR "c"+1,3
```

```
5033 POKE USR "c"+2,5
5034 POKE USR "c"+3,8
5035 POKE USR "c"+4,16
5036 POKE USR "c"+5,32
5037 POKE USR "c"+6,64
5038 POKE USR "c"+7,128
5040 REM D is d
5041 POKE USR "d"+0,0
5042 POKE USR "d"+1,0
5043 POKE USR "d"+2,32
5044 POKE USR "d"+3,64
5045 POKE USR "d"+4,255
5046 POKE USR "d"+5,64
5047 POKE USR "d"+6,32
5048 POKE USR "d"+7,0
5050 REM E is e
5051 POKE USR "e"+0,0
5052 POKE USR "e"+1,0
5053 POKE USR "e"+2,4
5054 POKE USR "e"+3,2
5055 POKE USR "e"+4,255
5056 POKE USR "e"+5,2
5057 POKE USR "e"+6,4
5058 POKE USR "e"+7,0
5060 REM F is f
5061 POKE USR "f"+0,1
5062 POKE USR "f"+1,2
5063 POKE USR "f"+2,4
5064 POKE USR "f"+3,8
5065 POKE USR "f"+4,16
5066 POKE USR "f"+5,160
5067 POKE USR "f"+6,192
5068 POKE USR "f"+7,224
5070 REM G is g
5071 POKE USR "g"+0,16
5072 POKE USR "g"+1,16
5073 POKE USR "g"+2,16
5074 POKE USR "g"+3,16
5075 POKE USR "g"+4,16
5076 POKE USR "g"+5,84
5077 POKE USR "g"+6,56
5078 POKE USR "g"+7,16
5080 REM H is h
5081 POKE USR "h"+0,128
5082 POKE USR "h"+1,64
5083 POKE USR "h"+2,32
5084 POKE USR "h"+3,16
5085 POKE USR "h"+4,8
5086 POKE USR "h"+5,5
5087 POKE USR "h"+6,3
5088 POKE USR "h"+7,7
5090 RETURN
```

Program 34 MOLECULAR MOTION – FAST

```
  1 CLEAR 30000
  2 RANDOMIZE
  5 PRINT AT 10,1;"Please wait, loading data."
 10 GO SUB 1000
 20 GO SUB 2000
 30 GO SUB 3000
 50 LET temp=3
```



```
 100 RANDOMIZE USR 30464
 110 PAUSE temp
 120 LET l$=INKEY$
 130 IF l$="" THEN  GO TO 100
 140 IF l$="m" THEN  LET temp=temp-1
 150 IF temp<1 THEN  LET temp=1
 160 IF l$="n" THEN  LET temp=temp+1
 170 IF temp>5 THEN  LET temp=5
 180 GO TO 100
1000 FOR i=30464 TO 30759
1010 READ x
1020 POKE i,x
1030 NEXT i
1100 REM machine codes in decimal
1120 DATA 58,128,126,79,6,122,10,111
1130 DATA 4,10,103,4,10,95,4,10,87,25
1140 DATA 126,254,63,202,13,120,254,0,202,128,119
1150 DATA 254,9,194,13,120,123,254,223,40,22
1160 DATA 254,255,40,24,254,31,40,26,254,225
1170 DATA 40,28,254,1,40,30,17,31,0,195
1180 DATA 0,120,17,225,255,195,0,120,17,1
1190 DATA 0,195,0,120,17,33,0,195,0,120
1200 DATA 17,223,255,195,0,120,17,255,255,195
1210 DATA 0,120,0,0,0,0,0,0,0,0
1220 DATA 0,0,0,0,0,0,0,0,0,0
1225 DATA 0,0,0,0,0,0,0,0,0,0
1226 DATA 0,0,0,0,0,0,0,0,0,0
1230 REM horizontal wall at 128,119
1250 DATA 123,254,31,40,22
1260 DATA 254,32,40,24,254,33,40,26,254,223
1270 DATA 40,28,254,224,40,30,17,33,0,195
1280 DATA 0,120,17,223,255,195,0,120,17,224
1290 DATA 255,195,0,120,17,225,255,195,0,120
1300 DATA 17,31,0,195,0,120,17,32,0,195
1310 DATA 0,120,0,0,0,0,0,0,0,0
1320 DATA 0,0,0,0,0,0,0,0,0,0
1330 DATA 0,0,0,0,0,0,0,0,0,0
1340 DATA 0,0,0,0,0,0,0,0,0,0
1350 DATA 0,0,0,0,0,0,0,0,0,0
1360 DATA 0,0,0,0,0,0,0,0,0,0
1370 DATA 0,0,0,0,0,0,0,0,0,0,0,0
1380 REM 'dirend' at 0,120
1390 DATA 122,2,5,123,2,5,195,32,120,0
1400 REM 'empty' at 13,120
1410 DATA 0,0,0,0,54,56,229,6,122,10
1420 DATA 111,4,10,103,54,63,225,124,2,5
1430 DATA 125,2,13,194,4,119,201,0,0,0
1500 RETURN
2000 REM make up tables
2050 LET mols=35
2100 FOR i=1 TO mols
2110 LET pos=22753+INT (29*RND)+32*INT (14*RND)
2120 LET poshi=INT (pos/256)
2130 LET poslo=pos-poshi*256
2140 POKE (122*256+i),poslo
2150 POKE (123*256+i),poshi
2160 LET r=INT (RND*8)
2200 IF r=0 THEN  POKE (124*256+i),1: POKE (125*256+i),0
2210 IF r=1 THEN  POKE (124*256+i),33: POKE (125*256+i),0
2220 IF r=2 THEN  POKE (124*256+i),32: POKE (125*256+i),0
2230 IF r=3 THEN  POKE (124*256+i),31: POKE (125*256+i),0
2240 IF r=4 THEN  POKE (124*256+i),255: POKE (125*256+i),255
```

```
2250 IF r=5 THEN  POKE (124*256+i),223: POKE (125*256+i),255
2260 IF r=6 THEN  POKE (124*256+i),224: POKE (125*256+i),255
2270 IF r=7 THEN  POKE (124*256+i),225: POKE (125*256+i),255
2280 NEXT i
2300 POKE 126*256+128,mols
2400 RETURN
3000 REM Do display
3010 CLS
3020 PRINT AT 0,8;"Molecular motion"
3030 PRINT AT 2,0;"Press M to increase temperature."
3040 PRINT AT 4,0;"Press N to decrease temperature."
3060 INK 0: PAPER 0
3070 PRINT AT 6,0;"                                "
3080 PRINT AT 21,0;"                                "
3090 INK 1: PAPER 1
3100 FOR i=7 TO 20
3110 PRINT AT i,0;" "
3120 PRINT AT i,31;" "
3130 NEXT i
3140 INK 7: PAPER 7
3150 FOR i=7 TO 20
3160 PRINT AT i,1;"oooooooooooooooooooooooooooooo"
3170 NEXT i
3180  INK 0
3200 RETURN
5000 FOR i=0 TO mols
5010 PRINT PEEK (124*256+i),PEEK (125*256+i)
5020 NEXT i
```

Program 35 GRAVITY

```
  1 REM GRAVITY
  5 LET acceleration=-10
  6 LET p$="      "
 10 CLS
 20 PRINT AT 0,8;"VERTICAL HEIGHT"
 30 PRINT AT 4,0;"This program prints the vertical"
 40 PRINT AT 6,0;"height reached by an object"
 50 PRINT AT 8,0;"thrown vertically upwards."
 60 PRINT AT 12,0;"Enter the initial speed"
 70 PRINT AT 14,0;"in the range 0 to 200."
 80 INPUT "Speed = ";initspeed
 90 CLS
100 REM VERTICAL HEIGHT
110 PRINT AT 0,0;"Acceltn.  Speed   Height   Time"
130 FOR t=0  TO 18
140 LET height=initspeed*t+0.5*acceleration*t*t
150 LET speed=initspeed+acceleration*t
160 LET number=acceleration: GO SUB 1000
170 PRINT AT t+2,3;truncate
180 LET number=speed: GO SUB 1000
190 PRINT AT t+2,10;truncate
195 IF height<1 THEN  LET h$="0"+p$
200 LET number=height: GO SUB 1000
210 PRINT AT t+2,18;truncate
220 LET number=t: GO SUB 1000
230 PRINT AT t+2,27;truncate
240 NEXT t
250 STOP
1000 REM Convert to four digits
1010 IF ABS number<0.01 THEN  LET number=0
1020 LET n$=STR$ number+"     "
```



```
1030 LET truncate=VAL n$( TO 4)
1040 IF number<0 THEN  LET truncate=VAL n$( TO 5)
1050 RETURN
```

Program 36 RESONANCE

```
  1 REM LCR Resonance
 10 REM Define ohm symbol
 11 POKE USR "a"+0,24
 12 POKE USR "a"+1,36
 13 POKE USR "a"+2,66
 14 POKE USR "a"+3,129
 15 POKE USR "a"+4,66
 16 POKE USR "a"+5,36
 17 POKE USR "a"+6,36
 18 POKE USR "a"+7,231
 20 REM Define micro symbol
 21 POKE USR "b"+0,66
 22 POKE USR "b"+1,66
 23 POKE USR "b"+2,66
 24 POKE USR "b"+3,100
 25 POKE USR "b"+4,88
 26 POKE USR "b"+5,64
 27 POKE USR "b"+6,64
 28 POKE USR "b"+7,64
100 CLS
110 PLOT 0,5
120 DRAW 255,0
130 PLOT 5,0
140 DRAW 0,175
150 PRINT AT 0,0;"                                "
160 PRINT AT 1,0;"                                "
170 PRINT AT 2,0;"                                "
180 INPUT "Inductance (millihenries) = ";L
190 PRINT AT 0,0;"Inductance = ";L;" mH"
200 INPUT "Resistance (ohms) = ";R
210 PRINT AT 1,0;"Resistance = ";R;" a"
220 INPUT "Capacitance (microfarads) = ";C
230 PRINT AT 2,0;"Capacitance = ";C;" bF"
240 IF R=0 THEN  LET R=.0001
250 LET E=50
260 PLOT 5,5
270 LET flag=0
280 FOR f=1 TO 2500 STEP 10
290 LET XL=f*L/1000
300 LET XC=1000000/(f*C)
310 LET X=XL-XC
320 LET Z=SQR (R*R+X*X)
330 LET I=E/Z
340 LET VC=I*XC
350 IF VC<2 THEN  LET f=2550: GO TO 400
370 IF VC>1500 THEN  LET flag=0: GO TO 400
380 IF flag=1 THEN  DRAW 5+INT (f/10)-PEEK 23677,5+INT (VC/10)-PEEK 23678
390 IF flag=0 THEN  PLOT 5+INT (f/10),5+INT (VC/10): LET flag=1
400 NEXT f
410 GO TO 150
```

Program 37 CAPACITOR DISCHARGE

```
  1 REM Capacitor Discharge
 10 REM Define ohm symbol
 11 POKE USR "a"+0,24
```

```
 12 POKE USR "a"+1,36
 13 POKE USR "a"+2,66
 14 POKE USR "a"+3,129
 15 POKE USR "a"+4,66
 16 POKE USR "a"+5,36
 17 POKE USR "a"+6,36
 18 POKE USR "a"+7,231
 20 REM Define micro symbol
 21 POKE USR "b"+0,66
 22 POKE USR "b"+1,66
 23 POKE USR "b"+2,66
 24 POKE USR "b"+3,100
 25 POKE USR "b"+4,88
 26 POKE USR "b"+5,64
 27 POKE USR "b"+6,64
 28 POKE USR "b"+7,64
100 CLS
110 PLOT 0,5
120 DRAW 255,0
130 PLOT 5,0
140 DRAW 0,175
150 PRINT AT 0,0;"                                "
160 PRINT AT 1,0;"                                "
170 PRINT AT 2,0;"V": PRINT AT 21,31;"t";
180 INPUT "Capacitance (microfarads) = ";C
190 PRINT AT 0,0;"Capacitance = ";C;" bF"
200 INPUT "Resistance (ohms) = ";R
210 PRINT AT 1,0;"Resistance = ";R;" a"
240 IF R=0 THEN  LET R=.0001
250 LET E=148
260 PLOT 5,E+5
270 LET time=0
280 LET charge=E*C: REM microcoulomb
290 LET voltage=E
300 LET timeinc=1
310 REM Begin iteration
320 LET current=voltage/R
330 LET charge=charge-current*timeinc
340 LET voltage=charge/C
350 LET time=time+timeinc
360 DRAW 5+time-PEEK 23677,5+voltage-PEEK 23678
370 IF time<250 THEN  GO TO 310
400 GO TO 150
```

Program 38 PROJECTILES

```
 30 LET g=10
 40 LET f=0
 50 LET angle=45
 60 LET speed=25
 70 LET mass=1
100 REM Collect initial conditions
110 CLS
120 PRINT AT 1,5;"PROJECTILE MOTION"
130 PRINT AT 3,0;"Initial conditions:"
140 PRINT AT 5,16;"Speed     ";speed
150 PRINT AT 7,16;"Angle     ";angle
160 PRINT AT 9,16;"Friction  ";f
170 PRINT AT 11,16;"Gravity  ";g
180 PRINT AT 13,0;"Press ENTER to confirm."
190 PRINT AT 15,0;"Press SPACE to change"
```



```
200 LET key=CODE INKEY$
210 IF key<>13 AND key<>32 THEN  GO TO 200
220 IF key=13 THEN  GO TO 900
230 REM Change values
240 PRINT AT 17,0;"Which do you want to change?"
250 PRINT AT 19,0;"Enter S, A, F or G"
260 LET 1$=INKEY$: IF 1$<>"S" AND 1$<>"s" AND 1$<>"A" AND 1$<>"a" AND 1$<>"F" A
ND 1$<>"f" AND 1$<>"G" AND 1$<>"g" THEN  GO TO 260
265 CLS
270 IF 1$="S" OR 1$="s" THEN  GO SUB 500
280 IF 1$="A" OR 1$="a" THEN  GO SUB 600
290 IF 1$="F" OR 1$="f" THEN  GO SUB 700
300 IF 1$="G" OR 1$="g" THEN  GO SUB 800
310 GO TO 100
500 REM Alter initial speed
520 PRINT AT 0,0;"Enter new starting speed        "
530 PRINT AT 2,0;"in the range 0 to 100.          "
535 PRINT AT 4,0;"                                "
540 INPUT speed
550 IF speed<0 OR speed>100 THEN  GO TO 500
560 RETURN
600 REM Alter initial angle
620 PRINT AT 0,0;"Enter new angle of projection "
630 PRINT AT 2,0;"in the range 0 to 90.           "
635 PRINT AT 4,0;"                                "
640 INPUT angle
650 IF angle<0 OR angle>90 THEN  GO TO 600
660 RETURN
700 REM Alter frictional drag
720 PRINT AT 0,0;"Enter new frictional drag "
730 PRINT AT 2,0;"in the range 0 to 9.            "
735 PRINT AT 4,0;"                                "
740 INPUT f
750 IF f<0 OR f>9 THEN  GO TO 700
760 RETURN
800 REM Alter gravitational acceleration
820 PRINT AT 0,0;"Enter new gravity "
830 PRINT AT 2,0;"in the range 0 to 20.           "
835 PRINT AT 4,0;"                                "
840 INPUT g
850 IF g<0 OR g>20 THEN  GO TO 800
860 RETURN
900 REM Select variable for investigation
910 CLS
920 PRINT AT 17,0;"Which do you want to investigate"
940 PRINT AT 19,0;"Enter S, A, F or G"
950 LET 1$=INKEY$: IF 1$<>"S" AND 1$<>"s" AND 1$<>"A" AND 1$<>"a" AND 1$<>"F" A
ND 1$<>"f" AND 1$<>"G" AND 1$<>"g" THEN  GO TO 950
955 IF 1$="S" OR 1$="s" THEN  LET hell=1500
960 IF 1$="A" OR 1$="a" THEN  LET hell=1600
965 IF 1$="F" OR 1$="f" THEN  LET hell=1700
970 IF 1$="G" OR 1$="g" THEN  LET hell=1800
980 CLS
990 GO TO hell
1000 REM Initialize variables
1010 LET timeincr=0.1
1020 LET x=0: LET y=40
1025 PLOT 255,40: DRAW -255,0
1030 LET gravity=-g
1035 LET drag=f/10
1040 LET xspeed=speed*COS (angle*PI/180)
1050 LET yspeed=speed*SIN (angle*PI/180)
```

```
1060 REM Iteration starts here
1070 LET xforce=drag*xspeed
1080 LET yforce=drag*yspeed
1100 LET xacceleration=-xforce/mass
1110 LET xspeed=xspeed+xacceleration*timeincr
1120 LET x=x+xspeed*timeincr
1200 LET yacceleration=gravity-yforce/mass
1210 LET yspeed=yspeed+yacceleration*timeincr
1220 LET y=y+yspeed*timeincr
1230 IF x>255 OR y>175 OR y<0 THEN  RETURN
1300 DRAW x-PEEK 23677,y-PEEK 23678
1310 GO TO 1060
1500 REM Investigate speed
1510 PRINT AT 0,0;"Investigating speed              "
1520 PRINT AT 2,0;"Present speed = ";speed;"       "
1525 PRINT AT 4,0;"                                "
1530 GO SUB 1000
1540 PRINT AT 2,0;"Press SPACE to change speed"
1550 PRINT AT 4,0;"Press C to change something else"
1560 LET k$=INKEY$: IF k$<>" " AND k$<>"C" AND k$<>"c" THEN  GO TO 1560
1570 IF k$="C" OR k$="c" THEN  GO TO 100
1580 GO SUB 500
1590 GO TO 1500
1600 REM Investigate angle
1610 PRINT AT 0,0;"Investigating angle              "
1620 PRINT AT 2,0;"Present angle = ";angle;"            "
1625 PRINT AT 4,0;"                                "
1630 GO SUB 1000
1640 PRINT AT 2,0;"Press SPACE to change angle"
1650 PRINT AT 4,0;"Press C to change something else"
1660 LET k$=INKEY$: IF k$<>" " AND k$<>"C" AND k$<>"c" THEN  GO TO 1660
1670 IF k$="C" OR k$="c" THEN  GO TO 100
1680 GO SUB 600
1690 GO TO 1600
1700 REM Investigate friction
1710 PRINT AT 0,0;"Investigating friction           "
1720 PRINT AT 2,0;"Present friction = ";f;"              "
1725 PRINT AT 4,0;"                                "
1730 GO SUB 1000
1740 PRINT AT 2,0;"Press SPACE to change friction"
1750 PRINT AT 4,0;"Press C to change something else"
1760 LET k$=INKEY$: IF k$<>" " AND k$<>"C" AND k$<>"c" THEN  GO TO 1760
1770 IF k$="C" OR k$="c" THEN  GO TO 100
1780 GO SUB 700
1790 GO TO 1700
1800 REM Investigate gravity
1810 PRINT AT 0,0;"Investigating gravity           "
1820 PRINT AT 2,0;"Present gravity = ";gravity;"              "
1825 PRINT AT 4,0;"                                "
1830 GO SUB 1000
1840 PRINT AT 2,0;"Press SPACE to change gravity"
1850 PRINT AT 4,0;"Press C to change something else"
1860 LET k$=INKEY$: IF k$<>" " AND k$<>"C" AND k$<>"c" THEN  GO TO 1860
1870 IF k$="C" OR k$="c" THEN  GO TO 100
1880 GO SUB B00
1890 GO TO 1800
```


Program 39 NEWTON

```
  1 REM Newton
  2 REM Satellite motion
100 CLS
110 PRINT AT 0,7;"SATELLITE MOTION"
120 PRINT AT 2,0;"The aim of this program is to"
130 PRINT AT 4,0;"set a rocket in orbit around the"
140 PRINT AT 6,0;"moon from a space station."
150 PRINT AT 8,0;"You can choose the initial speed"
160 PRINT AT 10,0;"and direction of the rocket."
170 PRINT AT 12,0;"Crashing the rocket or losing"
180 PRINT AT 14,0;"it in space causes a restart."
190 PRINT AT 21,0;"Press B to begin."
200 IF INKEY$<>"b" AND INKEY$<>"B" THEN  GO TO 200
210 CLS
220 REM Draw moon and space station
230 FOR i=0 TO 360 STEP 8
240 LET x=128+6*SIN (i*PI/180)
250 LET y=88+6*COS (i*PI/180)
260 PLOT 128,88
270 DRAW x-PEEK 23677,y-PEEK 23678
280 NEXT i
300 FOR i=0 TO 360 STEP 10
310 LET x=128+2*SIN (i*PI/180)
320 LET y=3+2*COS (i*PI/180)
330 PLOT 128,3
340 DRAW x-PEEK 23677,y-PEEK 23678
350 NEXT i
360 REM Next attempt begins here
370 PRINT AT 0,0;"                                "
380 PRINT AT 1,0;"                                "
400 INPUT "Enter speed (0 to 10) ";speed
410 PRINT AT 0,0;"Speed = ";speed
420 INPUT "Enter angle (-90 to 90) ";angle
430 PRINT AT 1,0;"Angle =  ";angle
440 REM Calculate current position and speed
450 LET x=128: LET y=5
460 LET xvelocity=speed*SIN (angle*PI/180)
470 LET yvelocity=speed*COS (angle*PI/180)
480 PLOT x,y
500 REM Iteration begins
510 REM The moon is at 128,88
520 REM First calculate the rocket-moon distance
530 LET xdisplacement=x-128
540 LET ydisplacement=y-88
550 LET parameter=xdisplacement*xdisplacement+ydisplacement*ydisplacement
560 LET distance=SQR (parameter*parameter*parameter)
570 IF distance<350 THEN  GO TO 360
600 REM Compute new speed
610 LET xvelocity=xvelocity-200*xdisplacement/distance
620 LET yvelocity=yvelocity-200*ydisplacement/distance
630 REM Compute new positions
640 LET x=x+xvelocity
650 LET y=y+yvelocity
660 IF x<0 OR x>255 OR y<0 OR y>175 THEN  GO TO 360
680 DRAW x-PEEK 23677,y-PEEK 23678
690 GO TO 500
```

Program 40 RUTHERFORD

```
  1 REM Rutherford
  2 REM Alpha particle scattering
100 CLS
110 PRINT AT 0,3;"ALPHA PARTICLE SCATTERING"
120 PRINT AT 2,0;"The aim of this program is to"
130 PRINT AT 4,0;"fire apha particles at random"
140 PRINT AT 6,0;"at a nucleus of gold."
150 PRINT AT 8,0;"The alpha particles are"
160 PRINT AT 10,0;"deflected by the nucleus and"
170 PRINT AT 12,0;"there could be a direct hit."
180 PRINT AT 16,0;"Press B to begin."
200 IF INKEY$<>"b" AND INKEY$<>"B" THEN  GO TO 200
210 CLS
220 REM Draw gold nucleus
230 FOR i=0 TO 360 STEP 10
240 LET x=128+2*SIN (i*PI/180)
250 LET y=88+2*COS (i*PI/180)
260 PLOT 128,88
270 DRAW x-PEEK 23677,y-PEEK 23678
280 NEXT i
360 REM Next attempt begins here
370 LET x=0
380 LET y=RND*175
400 LET xvelocity=10
410 LET yvelocity=0
480 PLOT x,y
500 REM Iteration begins
510 REM The nucleus is at 128,88
520 REM First calculate the particle-nucleus distance
530 LET xdisplacement=x-128
540 LET ydisplacement=y-88
550 LET parameter=xdisplacement*xdisplacement+ydisplacement*ydisplacement
560 LET distance=SQR (parameter*parameter*parameter)
570 IF distance<10 THEN  GO TO 360
600 REM Compute new speed
610 LET xvelocity=xvelocity+200*xdisplacement/distance
620 LET yvelocity=yvelocity+200*ydisplacement/distance
630 REM Compute new positions
640 LET x=x+xvelocity
650 LET y=y+yvelocity
660 IF x<0 OR x>255 OR y<0 OR y>175 THEN  GO TO 360
680 DRAW x-PEEK 23677,y-PEEK 23678
690 GO TO 500
```


Index