MAKING THE MOST OF YOUR HX20

Tim Hartnell

MAKING THE MOST OF YOUR HX20

Tim Hartnell

Interface Publications
London and Melbourne

CONTENTS

FOREWORD

Over the years, we have seen computers shrink from vast machines that occupied a whole suite of rooms, to tiny hand-held devices. Although the HX20 is not the smallest of these machines, it is one of the smallest units which is built for ease and immense flexibility of operation.

In this book, Tim Hartnell shows you how to tap the power of your HX20. After discussing how computers work, he gives an overview of the fascinating history of computers, right up to the development of computers like the one you now own. Then he leads you through programming the HX20, step by step.

You'll be shown what to expect from commercial software - business and personal - and shown how to develop your own programs. A number of major programs, including MINICALC, PERSONAL FINANCE, REVERSI, CHECKERS and CHESS are also included in the book, to demonstrate how powerful is the genie you now have at your command.

All in all, this fine book represents a resource which you're sure to turn to again and again in the coming months. With Tim Hartnell's help, you're ready now to make the most of your HX20.

Jeremy Ruston,
Kensington, 1983

(Mr Ruston is the creator of a major original structured language - Ruston BASIC - and author of a number of computer books including THE BBC MICRO REVEALED and PASCAL FOR HUMAN BEINGS.)

CHAPTER ONE - WELCOME TO THE MACHINE

The era of the personal computer has without doubt arrived. The HX20 can be used in an enormous number of ways, from games and self-instruction in programming, through to control and management of home finances, and for business use. The computer has come a long way since Charles Babbage's dreams for his 'Analytical Engine' — as he called his first mechanical computer — were lost in a maze of interlocking cogs and wheels.

The HX20 is one of the outstanding personal computers available today. This book aims to help you make the most of your HX20, whether it is the first computer you have ever owned, or you are trading up from another machine.

If you stick to buying commercial software, or just key in programs from books and magazines, you'll miss one of the real thrills of owning a computer — making it bend to your will. To do this, you need a degree of facility in programming. This book should help you develop such facility, and will point you on the way to developing programs of your own.

To help you think clearly about the machine you have in your hands, we're going to start with the basic question: "What is a computer?". The American National Standards Institute has defined a computer as a 'device capable of performing systematic and logic procedures, without intervention by a human operator during the run'. Wordy and convoluted as this definition may seem, it is actually quite accurate.

Let's start with the first part of the definition: a 'device capable of performing systematic sequences of operation upon data...'. 'Systematic' is the vital word. A computer follows instructions given to it by the human programmer.

Your HX20 will not suddenly take off on paths of its own, making its own decisions and acting on them, unless you have specifically written a program which gives it the appearance of being able to do this. Babbage's close friend and public supporter, Ada, the Countess of Lovelace (Lord Byron's daughter) once pointed this out quite strongly. In her argument against thinking machines — which has come to be known as "Lady Lovelace's Objection" — she said: "The Analytic Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform."

Very shortly, when we start the programming part of this book, you'll learn about line numbers, which precede each statement in a program. You'll also learn that, unless told to do otherwise, the computer will systematically perform what it is told to do in those lines, taking them one by one in numerical order.

So, your computer is only capable of systematic action. This systematic action need not be just satisfying routine business demands like sorting names and addresses, or adding up vast columns of numbers (although your computer will do this very rapidly, and accurately, for hour after tireless hour) but also for such tasks as playing board games such as Checkers and Reversi. So long as a task, such as maintaining a payroll, or playing Checkers,

can be broken down into a series of steps which can be clearly delineated, a computer can be taught to carry out the task.

Now playing Checkers may not seem, at first sight, like a task which is simply a series of operations, carried out systematically. When played by a human being, such games seem more the result of applying an intuitional judgement on the state of the board as a whole than carefully working through the series of steps.

You, as a human player, do not play by looking first at the top left hand corner of the board to see if you can capture from there, then move to the next square along to see if you can capture from there...and so on, until you've swept the whole board, and then start looking for non-capture moves which would threaten the other player's pieces. Instead, you just look at the board, and without too much thought, 'know' which is the best move to make.

But the computer cannot do this. It must have every task spelt out in detail, tiny step by tiny step. The Checkers program later in this book, to allow you to challenge the HX20 to a game, actually does follow a procedure such as I've decribed. But because the computer works so quickly and choses randomly from equally good alternative moves, you cannot tell how methodically it is working its moves out. Another program in this book allows the computer to 'write poetry'. Although the mechanism of creating its verse is reduced to a few simple lines of program, the HX20's output appears to be the work of a wilful genius. You will learn to write programs like this with the aid of this book;

programs which appear to move beyond the 'systematic sequences of operation upon data'.

So, the first thing we can learn from the ANSA definition is that your HX20 works only logically, and in strict accord with instructions it has been given by you, even if the output of the program does not always seem — at first sight — as if it is the product of a systematic, orderly process.

The next part of the definition explains that the 'systematic sequences of operation upon data' include 'numerous arithmetic and logic procedures'. This is really the heart of the matter. The computer can manipulate numbers, just as you or a pocket calculator can (although it is probably quicker than the calculator, and more accurate than you).

But as well as working with numbers, the computer can make decisions. As you'll learn a little further on in this book, the computer is capable of performing an IF...THEN evaluation. That is, it can be programmed to carry out the following sequence: IF factor A is true THEN do step B. IF it is raining, THEN gets an umbrella. IF the clock has stopped, THEN wind it up.

This decision-making can extend to include more than one factor at a time (IF factor A is true AND factor B is also true THEN carry out action C). Going back to our Checkers game, we will discover the computer

makes among other things, an evaluation of the following statement:

> IF the square on the board I am looking at contains one of my pieces, AND the square diagonally to the right of that contains one of my opponent's pieces AND the square beyond that is empty THEN capture the enemy piece.

A fairly long series of such decisions, made in a predefined order (so that, for example, capturing a king is considered more important than the capture of a non-promoted piece, or that a capture is more important than a simple non-capture move) allows your HX20 to appear as if it is thinking, and making human-like decisions, and acting on them.

Lady Lovelace, writing in 1842, was well aware of the decision-making power a computer could be given: "The engine is capable under certain circumstances of feeling about to discover which of two or more possible contingencies has occurred, and then of shaping its future courses accordingly."

Now we come to the final part of the American National Standards Institute definition. It says that the systematic sequence of operations, including arithmetic and logic procedures, must be carried out 'without intervention by a human operator during the run'. This is what makes your HX20 different from a pocket calculator. Start your computer up, respond to its questions (such as 'How many hours did employee 3 work this week') and — without further aid from the human operator — the Epson will process the information in accord with its

programming, and will output the results. The effect can appear miraculous, as a tiny piece of electronics processes numbers, words, positions in space or concepts, to arrive at accurate conclusions.

When you come to write your own programs, you'll find you need a clearly-defined target to aim at, and well-reasoned path to reach that target. As in most fields of human activity, certain ways of doing things have been found to be better than other ways which achieve more or less the same end.

In the field of computer programming, one approach to programming -- which makes for neat programs which are relatively simple to follow, and to debug (that is, to remove errors) -- is called 'structured programming'. When, a little later in the book, we get around to writing a program to play Noughts and Crosses on the HX20, we'll discuss the fundamental ideas of structured programming. They are covered in more detail towards the end of the book in the chapter called IMPROVING YOUR PROGRAMMING TECHNIQUE.

But, that comes later. For now, we've covered a definition of the computer which should help you understand, to some extent, the paths your HX20 follows to carry out its tasks. In the next chapter, we'll try to answer the question "How Does Your HX20 Work?".

CHAPTER TWO - HOW DOES YOUR COMPUTER WORK?

Human beings tend work in a number system based on 10. We have nine digits to work with (plus zero) and with these we can represent any number we want to. The position of a number indicates its value, with the position being made clear by the presence of trailing zeroes. That is, we know 90 is 10 times greater than 9, with the zero which follows the 9 in 90 making it clear which value the number represents.

Computers tend to work in a number system based on two, the binary system. The binary system uses only one and zeroes, with the sequence and position of digits indicating their value. Here are the numbers one to nine in our normal numbering system (base ten), with their binary (base two) equivalents underneath:

1	2	3	4	5	6	7	8	9
1	10	11	100	101	110	111	1000	1001

As you can see, binary numbers grow long very quickly, and they lack the 'instant recognition' our Arabic numerals possess. Why on earth should they be used? Simply because your computer is, in essence, a vast heap of switches, and a switch — as you know — can have two states, and two states only, off and on. If we use off to represent zero and on to represent one, we can represent any number with a long enough row of switches.

Your computer, of course, does not require you to know or understand binary arithmetic. It does the conversions from your base system to its own, and

back, as needed. The vast majority of computers in the world use binary arithmetic. In the early days of programming, programmers had to work with zeroes and ones, which meant writing a program was an immensely time-consuming and frustrating task. Any error in running meant a vast number of ones and zeroes had to be checked.

In those days, programmers really were a race apart. The intellectual strain must have been tremendous. Fortunately, the computer does most of the work for us now, in changing numbers from long strings of zeroes and ones into numbers we can understand.

But this discussion, while it indicates how the computer manipulates numbers, does not help explain how a computer 'thinks'.

Computer logic, computer decision-making power, is based on the discoveries of a turn of the century English mathematician, George Boole. His work is called Boolean Algebra.

The value of Boole's discoveries lies in the fact that he reduced decision-making to a mathematical process, a process that proved tailor-made for computer use. Your computer stores 'true' and 'false' as numbers.

Let me illustrate this. If you turn your computer on, and type in PRINT 1 = 1 (a true statement) you'll see the computer prints up -1, showing it recognizes 'truth' as -1. Now enter a false statement, such as PRINT 3 = 4. This time your computer will return a zero, indicating that a false condition is stored as a zero.

Try your computer with the following statements, to check that it works as required in all circumstances:

```
PRINT 100>7
PRINT 10=17
PRINT 10=10
PRINT 10+5=15
A=5
B=5
B=A
```

Now, as you'll see later on, your computer can make decisions which involve 'logical operators'. That is, it can see if statement A AND statement B is true; if statement A OR B is true; or if neither statements A NOR B are true. The computer can even handle long complex chains of logic such as determining if statement A AND statement B are true AND statement C OR statement D is false OR statement E is false. The heart of the computer's decision-making power, as opposed to its ability to manipulate numbers, lies in this ability to handle Boolean Algebra, and in its ability to translate these findings -- expressed mathematically -- into an answer we can understand.

This, of course, is not the whole picture. Let's look more closely at your computer. It has a means of accepting information from the outside world (the keyboard) and a means of translating that information into the ones and zeroes it needs.

It has a way of storing that information until it needs it, of comparing new information with previously-held information, and a way of outputting

(via the screen, or the printer) the information it has developed.

If, for example, you wanted your computer to add four numbers together, you might first tell it that the letter A represents the number 4, that B equals 5, C equals 9 and D equals 2. It would accept this information, storing it as a number of binary patterns. Then, you might say ADD A and B, then ADD C and D, and print out the results.

There is a fixed binary pattern in the computer when you buy it, which tells it how to add two numbers when so instructed, and how to output the results.

Next, you might get it to determine which of the two results (A plus B or C plus D) was the larger, and print out this information. The ability to do this is also stored as a binaary pattern supplied with your computer when you buy it. Based on the finding of whether A plus B or C plus D was the larger, the computer could be instructed to do something else...and so on.

Read through this greatly simplified explanation until it begins to make some sort of sense. You will then be in a position to start to appreciate how your computer works. In fact, you don't need to know how the computer works in order to be able to use it — anymore than you have to be able to understand the workings of your television in order to watch a picture — but having an introduction to the subject will aid your appreciation of the electronic genie you have at your command.

CHAPTER THREE - YOUR COMPUTER'S FAMILY TREE

The history of true computers is short, less than 40 years. However, the history of man's attempts to ease the burden of manipulating numbers stretches back to the beginnings of civilization.

In fact, possession of the abacus, an early counting frame, is taken by some historians to be evidence of the presence of a civilized society. In this chapter, we'll be looking at the early history of computers. The later history, spanning the last 25 years, is fairly well known. The earlier times are not as well documented so, I've decided to concentrate on the earlier story here. And what a story it is.

In the past 40 years, many arguments for and against greater dependence on computers have been made. Despite the arguments against them, the encroachment of computers into our lives has continued steadily. And rather than fearing them, some of us have invited them into our homes, as your new computer bears witness.

Your computer is one of the latest developments in a line of machines which men have used to expand their thinking abilities. From notches in a stick to record cattle numbers, through pebbles in grooves in the sand, to beads on wires, man has exercised his ingenuity to increase his capacity to handle numbers and information. A product such as your computer, which you have bought without undue sacrifice, would have seemed a foolish dream to most of the men who labored to produce aids to man's thinking.

And there have been many such men. The history of computational devices goes way back into the past.

The earliest steps on the path that stretches up to your computer were made by men whose names were never recorded. But as we get closer to our own era, in the last few hundred years, specific inventions, discoveries and ideas stand out. These are the ideas which have made it possible for you to buy such a technological wonder as your computer. we'll be looking at these men -- and their inventions — in this section of the book.

Parallels have been drawn between the Industrial Revolution and our own time, which may one day be called the era of the Computer Revolution (or the Information Revolution, or even the Post-Industrial Revolution). Within a little over 100 years, the Industrial Revolution transformed the face of Britain — where the revolution began — and then the world. All were caught up in it, and its effects were so broad and far-reaching that few, if any, who were there at the beginning had any conception of how total, irreversible and inescapable the changes it wrought were to be.

So it will be with the computer. Already few aspects of our lives are beyond some computer influence. The clothes we wear, the food we eat, the television we watch, the planes we fly and the cars we drive, even the music we listen to, would either not be, or would be more complicated and expensive, if it were not for computers.

The change wrought by computers to our society will affect nearly all of us, in most aspects of our

lives. The Industrial Revolution did the same. It became a mighty steam hammer, pummelling people's attitudes, needs and lifestyle in an unstoppable manner.

The real difference between the Industrial Revolution and the 'Computer revolution' in which we are caught up, is that we have the benefit of being able to look back and see what happened last time. We know a little of how forces shape societies and of the impact of change. Our observations of the Industrial Revolution spell this out very clearly. But though we are aware, to some extent, of what will happen to us as the current revolution rolls on, we are powerless to stop it, even it we wanted to do so.

Bringing a computer into your home is embracing and encouraging the revolution. An insight into the men and their activities that brought us to this point may be an aid in interpreting the nature of the changes yet to come. And the history of computers is fascinating in itself, a witness to man's ingenuity.

Man started exercising that ingenuity a long time ago.

It is likely that heaps of pebbles were used in some way in the earliest attempts by man to manipulate numbers, and record the results of those manipulations. Early civilizations in the Asian river valleys, and in Egypt, certainly made use of pebbles in this way. The development from loose pebbles, to placing them in grooves in the sand, and then in grooves in a wooden or stone tray, is fairly easy to imagine.

From here, it is not too big a step to see how it would have been more convenient if the pebbles could be fixed in some way to the frame, forming the first version of the device known as the abacus. This is a frame (generally of wood) which contains a number of wires along which pebbles can slide, with the position of the pebbles conveying numerical information. (The ubiquity of pebbles in the development of early computation devices is attested to by the fact that the word calculate is derived from the Latin calculus, a pebble.)

It appears that the abacus was developed independently in several parts of the world, including most of the areas surrounding the Mediterranean, where it was well-distributed thousands of years before the birth of Jesus. Contemporary historical records from the fifth century BC mention the abacus in use among the Egyptians and Greeks, and the Spanish conquistadors found the device in use in Mexico and Peru. The Romans had abaci as well, as references by Juvenal, Cicero and Pliny bear witness.

Even the Russians were in on the act, with an abacus-like device that differed only slightly from the ones in use in such places as China and Western Europe. From China, the abacus spread into Japan in the fifteenth or sixteenth century. It is interesting to note that China and Japan were the two principal areas where use of the abacus survived well into this century. Development of the Japanese abacus continued up until 1930.

It is not surprising, given the widespread nature of the abacus and its extreme usefulness which allowed it to survive for over 5,000 years, that many

commentators on the field have seen the abacus as among the most significant inventions ever developed by man.

As discussed in the previous chapter, we generally write numbers down in the decimal system, where there are nine digits, plus zero. The number of series, and their positions in relation to the other digits, assign values to the digits. The decimal system, as we said, is easy for us to use, but most computers (including yours) prefer to work in a two-base system, in which there are only two digits, zero and one. Up until the eighth century, the use of the clumsy Roman system for representing numbers (with V for five, X for 100 and L for 1000, for example) was in very common use. But the Arabian system, which is more or less the one we use today, was fast surplanting it, because of its greater elegance and simplicity.

In the first or second century in India, the use of position to designate value and of zero as one of the fundamental digits, was developed. Although both devices had been used much earlier, it was the Hindu mathematicians who first codified the use of these two concepts. The Indian system spread to Arabia due to the vast commerce between the two countries and the Arabs continued to develop and extend ways of using it.

In the ninth century, following a vist to India, a mathematician by the name of Alkarismi wrote his fundamental treatise on calculating, Al-gebr we'l mukabala. This book (from whose title we have derived the word algebra) was the main means by which knowledge of the Arabian/Indian system

gradually penetrated the West. The word **algorithm**, meaning the series of steps taken to solve a problem, is widely used in computer circles today. This word comes from **algorism**, a word in current in the West for several centures which meant 'to solve using the techniques of Alkarismi'.

However, although Alkarismi's book was the main means by which the West learned how to manipulate numbers in the Hindu-Arabian manner, its influence was not felt until around 1100 when a Christian monk — Adelard of Bath — spent a year in a Muslim university (disguised, of course) and after this year translated the book into Latin.

But although the techniques of number manipulation were becoming more widespread, they were very far from universal. A population that is largely illiterate is hardly likely to be numerate. Even the (relatively) highly-educated Pepys records in his diary that he was obliged to engage an instructor 'of whom I intend to learn mathematiques'.

So the number system was waiting to be used, in a world where the demand for mathematical facility was increasing. Given the situation that most people had little or no real mathematical skill, it is not surprising that any device which made arithmetical processes simpler, or more approachable, would be readily welcomed.

Therefore, when John Napier, who lived in a village near Edinburgh, Scotland, produced a simple mechanical aid to multiplication — know generally as "Napier's Bones" — it proved almost immediately popular. The device was a series of rods with

numbers on each face, which could be combined to aid the multiplation of large numbers. Napier's Bones did not form, despite the impression created by some histories of computers, a calculating 'machine'. They were, instead, a reference tool which held, in a particularly convenient form, a means of working out complex multiplication sums.

Even though the "Bones" were popularly accepted, Napier's real contribution to the history of mathematics lay in his invention of logarithms, which allowed numbers to be multiplied together by adding their logarithms, and similarly reduced division to a process of subtraction.

It was soon realized that the product of two numbers could be achieved by the simple mechanical expedient of adding together two lengths, where the lengths represented the numbers' logarithms. From this realization, it was a small step to the giant leap made by English parson William Oughtred, who produced the first primitive slide rule around 1620.

Pepys, still continuing his study of 'mathematiques', bought a slide rule some forty years later, and found it 'very pretty for all questions of arithmetic'. In fact the simplicity and elegance of the slide rule for multiplication and division ensured it a position of prime importance among those who needed to use mathematics, until it was superseded by the pocket calculator and computer, only a few decades ago.

Although Napier's Bones and the slide rule could be described, to some extent, as mechanical devices, the term should perhaps be reserved for those aids

to calculation which used the position of particular points on interlocking cogs and wheels to represent numbers.

Blaise Pascal (after whom the computer language Pascal is named) is generally credited with having invented the mechanical calculating device. In the 1640's, while living in Paris, Pascal completed his first device, designed to aid his father, who was a tax collector. The machine, Pascal assured potential purchasers, would allow them to "without any effort perform all the operations of arithmetic", thus avoiding the "work which has often times factigued your spirit when you have worked with the counters or with the pen".

Pascal blossomed early as a mathematical prodigy. At the age of 11, he duplicated for himself a considerable portion of Euclid's work, and at 16 published a paper on solid geometry.

Although he died when he was just 39, Pascal found time to invent — among other things -- the barometer, the hydraulic press and the wheelbarrow.

His mechanical calculator was made up of a group of wheels which could be set to represent a number, and which could be turned to indicate another number, arranged in such a way that the turning to the second number performed an arithmetical process between the two numbers, so that the final position of the wheels represented the result of the arithmetic performed. Although Pascal's machines added and subtracted, they were only able to perform multiplication and division by repeated addition and subtraction respectively.

Thirty years later, G. von Leibniz, who co-invented the calculus with Newton, built the first machine which was a major advance on the Pascal devices. Leibniz worked out a way to mechanise multiplication. His method was so sound it still lay at the heart of the mechanical calculators which were built just before the pocket calculator invaded.

Deservedly proud of his machine, Leibniz claimed it would be "desirable to all those who are engaged in computations...the managers of financial affairs, the administrators of others' estates, merchants, surveyors, geographers, navigators, astronomers, and those connected with any of the crafts that use mathematics". He foresaw a big market for his device, a prophecy which did not come to pass.

Computers, of course, appear to do more than manipulate numbers. They also make decisions, and the first attempt to reduce logical decision-making to a 'science' appears to have been made by Charles, the 3rd Earl of Stanhope. He first built a calculating device which, although it was simpler to operate than Pascal's, employed the same basic mechanism.

He followed this up with a gadget he called the 'Logic Demonstrator', in which two coloured slides, a grey one representing the first premise of the argument, and a red one representing the second premise — slid back and forth across each other. The device was intended to aid the reaching of a logical conclusion as a result of reading the slides' positions related to each other, and to a scale marked on the wooden frame which held the whole thing.

Although it is stretching the definition of 'machine' to embrace the Earl's device, its invention at least suggests that men had begun to think that, given time, a 'science of logic' would be discovered and when it had been, its rules would be sufficiently precise to enable a machine to execute them.

The inadequacy of engineering at the time had prevented the Leibniz machine from gaining commercial distribution. It was not possible in that age to machine metal consistently to the tolerances required. In the early 1800's, however, Charles Thomas of Colmar, Alsace, put on the market a calculating device based closely on Leibniz's design. For over half a century he conducted a steady trade, selling a couple a month to "merchants, surveyors, geographers" and the rest.

As was discussed in the opening section of this book, the crucial factor that divides a calculator from a computer is the automatic nature of its processing, the fact that the "systematic sequence of operations, including arithmetic and logic procedures" are, and this is the important thing, carried out "without intervention by a human operator during the run". The devices mentioned to date, obviously, demand constant human intervention.

Aware of this, and seized by the idea of producing a calculating device which would work independently of a human operator, Charles Babbage started in 1812 to build what he called a 'Difference Engine' which would, he hoped, produce more accurate logarithm tables than were currently available.

One of the most interesting (and in light of its effects, most poignant) moments in the history of computers occurred as follows. It is Babbage speaking: "One evening I was sitting in the rooms of the Analytical Society at Cambridge...with a table of logarithms lying open before me. Another member coming into the room, and seeing me half asleep, called out 'Well, Babbage, what are you dreaming about?' to which I replied 'I am thinking that all these tables might be calculated by machinery'."

Babbage knew that a table of values for any function could be deduced, to an acceptable degree of accuracy, by means of polynomials, where the numbers to be inserted in the tables were calculated by a series of additions. A 'Difference Engine', as Babbage called his idea, could carry out the additions needed and then print out the results.

The idea was sound, and Babbage's first device, which took him the best part of a decade to build, was sufficiently 'intelligent' to produce tables for quadratic functions to eight decimal places. The Royal Society and the government were impressed and gave him a grant to produce a device which would work to twenty decimal places. It seemed that Babbage was really on the way.

Sadly, the device was never completed. Babbage was let down by the inability of engineers of his day to produce parts as needed and by his own lack of concentrated effort. From this point on, Babbage's story is one of brilliant ideas wasted because they were not followed through to realisation. After the government had outlayed the enormous (for the time) sum of seventeen thousand pounds, and some twenty

years of effort had been made by Babbage, the government cut the money off. Disraeli is quoted as saying that the only value he could see in the machine was for calculating the vast amounts of money which had been squadered on it.

Babbage, however, was not disheartened. He had by now found a friend and confidante Ada, Countess of Lovelace, daughter of Lord Byron, whom we met in chapter one. To the end, she championed Babbage and his machine which she appeared to understand intuitively. Babbage himself said of the Countess that she seemed to understand the Engine "better than I do, and is far, far better at explaining it".

As well as the Countess to take his mind off the fact that his government funds had been stopped, Babbage had a new idea which had formed a decade before he finally stopped working on the Difference Engine. He was now looking beyond the first 'Engine' to a machine which had fired his imagination, a truly remarkable device he called the 'Analytical Engine'. His idea for this was perhaps the single most original and important idea in the entire history of computers.

The Difference Engine, remarkable as it was in concept, was designed to carry out a single task. The Analytical Engine, by contrast, would be able to turn its hand to any task of calculation which was set; that is, it could be programmed.

And Babbage's concept was so complete, his designs embody all the vital components of modern-day computers. This Engine fits well within the ANSI definition outlined at the start of the book.

Babbage's device could store numbers; contained a control unit to direct its multiplication of those numbers; allowed the operator to enter information and instructions; would carry out those instructions on the entered data without further intervention from the human operator; and could finally output the results of its work. Babbage even realised that his machine could be programmed with a series of punched cards similar to those he had seen in use controlling the weaving of patterned fabrics on Jacquard looms.

When you think about the length of time, in the modern history of computers, for which punched cards ('Do not spike, fold or mutilate') were used as programming devices -- and are still used in some college time-sharing systems — it can be seen that Babbage's ideas were very sound, even if he was not able to bring them fully to fruition.

Babbage designed his engine to work with two sets of cards. The first set controlled the arithmetic manipulation required (such as whether the machine was to add, or multiply, or whatever) and the second set contained the numbers with which the engine would do its work.

Babbage called the first set the operation cards and the second set he dubbed cards of the variables. The word 'variables', as you will learn a little later, is still used today for the numbers (and nowadays for words as well) which are changed within a particular series of operations, although the operations themselves do not change.

The Analytical Engine was a remarkable concept, as I

am sure you can appreciate. We, of course, have the benefit of hindsight, and can see it for the advance in human thought it represented. Few at the time recognized the potential intellectual power the engine represented. Lady Ada, as we've seen, was one of the few who saw what the device could be, and she wrote about it most eloquently: "The Analytical Engine weaves algebraical patterns just as the Jacquard loom weaves flowers and leaves". She was also the first, as we pointed out in chapter one, to discuss the creative power of machine intelligence: "The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with."

Despite the poetry of Ada's description, and her encouragement, the machine was never built. Babbage's plans were too ambitious. He had designed the Analytical Engine to work to fifty decimal places, a degree of accuracy which was absurdly beyond practical needs. The demands it placed on the builders of the enginer - coupled with the lack of engineering precision we have mentioned before - practically ensured the engine would prove impossible to build.

Babbage worked on it almost up to his death in 1871, growing more bitter as he aged. He gradually realised he was most unlikely to actually see the machine operating in his lifetime. His workrooms were full of partially completed devices. Not one, not even the original Difference Engine, was

finished because as soon as construction was well underway, Babbage's mind would alight on a new project, and the original one would be partially or completely abandoned.

However, many mathematicians and engineers studied the notes that he and Lady Lovelace had prepared, and one of them went ahead and built a Difference Engine that worked. Swedish engineer Georg Scheutz read about Babbage's machine in the Edinburgh Review and had one up and running by 1855. Babbage himself - now aged 63 - was among those who went to see it in action. Although he spoke well of the project, it must have hurt him mightily not only to see a working Difference Engine in action, but to learn that the British Government had actually bought one to help calculate life expectancy tables.

At the Centennial Exposition in Philadelphia in 1876, an American engineer named George Barnard Grant had exhibited a working Difference Engine. It was enormous, about the size of a Volkswagon 'Beetle', flattened out a little, and was powered by electricity, but it lay in clear lineal descent from Babbage's machine. The Grant monster was too big to even come into consideration as a machine for office (or, perish the thought, home) use, but the fact that it worked and drew such public attention, convinced Grant that he almost had something. The size of his gargantuan monster stood in the way of public acceptance. So, without much further ado, Grant set about making smaller units. He succeeded, and in due course managed to sell over one hundred of a device his called his 'Rack and Pinion Calculator'.

Twenty years after Babbage died, Herman Hollerith, then working with the U.S. Bureau of the Census, realised that the methods of collating and tabulating census results by hand were so slow that when the 1890 census was held (American law required a census each decade) work on the results of the previous census would not have been finished.

Hollerith knew of the work of Jacquard, the developer of the loom which Babbage considered. Many of the census questions required a 'yes' or 'no' answer, and Hollerith realised that if a card was used to represent each person who completed the census, the position on the card could represent a specific question, with a hole in that position to mean 'yes' and the lack of a hole to signify 'no'.

The proposal was accepted by the Census Bureau, after a contest with the proponent of another idea (of using coloured, coded cards) proved how fast the Hollerith system could be. The system was a roaring success. The census returns were converted into punched cards, and these were fed through a reading device which used electrodes dipping into a pool of mercury to complete connections. Each completed connection of a specific circuit represented a 'yes' answer and the results were tallied up continuously on little dials.

Forty years before this, the Babbage engines had founded on public and government ignorance and disinterest, and the lack of engineering skills. Now,

with the Hollerith tabulating machine, it was shown that both these barriers had - to a significant entent - ceased to exist.

Public acceptance of calculating machines was growing, increasing the chance of a workable device becoming a commercial success and thus making it worth a person's while to produce. Engineering skills, as well, were improving. The stage was set for mechanical calculating devices to come into their own.

What really helped draw the curtain back, though, was the development by Spanish mathematician Roman Verea of a system of 'direct multiplication'. Swiss engineer Otto Steiger realized that Verea's system would be considerably simpler to mechanise than ways which had previously been tried, and shortly produced a device called 'The Millionaire'. This calculator was first released in 1894 and retired, with honors, in 1935 when more than 4500 units had been sold.

The Millionaire was, however, difficult to use. American Door E. Felt realised that systems which accepted the entry of numbers in a slow, fiddly manner by the turning of wheels, or via a slide, were most unlikely to set the business world on its ear. So Felt developed a device with a typewriter-like keyboard, and at last the calculating machine was really off and running.

Felt also developed a primitive means of printing

the results of work done by his 'Comptometer', but the thunder of this was stolen by William Burroughs, who not only built his own calculating machine, complete with keyboard, but connected it to a very good printing device.

Despite the fact that Burrough's machine produced its results in a much more convenient form than did those produced by Felt, the hunger of business and government agencies for calculating machines was so vast that both men became millionaires as they pushed their machines into a market which appeared impossible to saturate. Hand-operated Burroughs adding machines lasted well into the 1970's. I remember using one myself - the only calculating device the company owned - to work out record returns in a radio station as late as 1976.

Back in the 1870's, the British physicist Lord Kelvin had designed a device to mechanically integrate variables in math problems, using a series of cogs and gears. He used it to predict British tides. It worked well, and save an enormous amount of human computation time. Many versions of the 'Kelvin tide predictor' were developed and used around the world.

In 1876, Kelvin published a paper that suggested that a more elaborate version of his tide machine should be able to cope with problems involving differential equations. He called his theoretical device a 'differential analyser' and while those who read his paper agreed that the idea was workable, no-one got around to actually building such a device for more than forty years.

It was a professor at the Massachusetts Institute of Technology, Vannevar Bush, who lead the men who developed a working differential analyser. The device was big, clumsy and slow...but it worked. The Bush machine was, in line with Kelvin's concept, built of interlocking gears, and Bush realised that all those gears meant the machine would always be very slow. So he decided to replace some of the gears with vacuum tubes, increasing its speed (and its size, for the tubes had to be well spread out to allow the heat they developed to escape) and its reliability. Bush was the first man to produce a partially-electronic computing device.

Two British physicists, Douglas Hartree and his student Arthur Porter, visited Bush in 1933. The moment he saw the Bush machine, Hartree realised it looked just like something built out of a big Meccano set. On their return to Britain, the men bought such a set for twenty pounds, and succeeded in building a differential analyser which actually worked. Although it was much smaller than the Bush machine, it produced answers within two per cent of those which the Bush device had produced, and the Meccano computer eventually solved several difficult problems.

Hartree later built a full-size analyser at Manchester University. Part of this unit can be seen in London's Museum of Science. When I went to see it, I was amused to see that the label on the exhibit points out that operators discovered they could get an insight into the problem being solved by watching the turning of the unit's gears. This nearly caused a disaster one day when an operator, engrossed in the turning wheels, managed to get his

tie caught up in the cogs. Fortunately, a colleague managed to disable the machine before the operator himself was 'differentiated'.

Despite the fact that binary arithmetic had sparked a great deal of theorectical interest among mathematicians, no-one had actually followed through with the idea of producing a machine which would work in binary. No-one that is, until Konrad Zuse, an engineering student at the University of Berlin in Charlottenburg, decided that not only would he build a working computer, but it would be one which worked in binary.

He gave up his job in 1936, much to his parents' horror, and told them from that now on he would be building a marvellous machine, right there in their apartment. Zuse took over a section of the dining room to build his first machine, and then - as his device grew in size and complexity - gradually worked his way out into the room until it all but swallowed it up.

Zuse's parents, much to their credit, let him get on with it, and the machine (which he called Z1) was eventually up and running. It worked in binary, accepting input from the operator via a keyboard, and put out its results in the form of little electric lights.

Z1 evolved into Z2. The mechanical switches were replaced by telephone switching relays, and instead of the keyboard (which had proved too slow) information was fed into the computer via holes punched in discarded lengths of film. Zuse then became involved in the army, followed by a stint working

for an aircraft design company. He built two or three devices to help with the calculations involved in the design of new aircraft, and all the while was working on his computer. Eventually Z2 became Z3.

Later, having developed a fourth machine (called, naturally enough, Z4) and wanting not only to save his machine from destruction but also to avoid capture by the Russians, he (and Werhner von Braun) trekked across Germany to temporary sanctuary in an Alpine village. The locals, however, thought he was a spy and that the curious machine he brought with him was one of Hitler's secret weapons. He was captured by the Allies but after questioning was released. He returned to Germany, where the ideas embodied in Z4 became the base of an electronics company he founded. It was eventually bought out by Siemens.

As we've seen, Vannevar Bush can get credit for first applying electronics to computing, and also for having sparked the development of computing devices in Britain in the twentieth century. But his contribution did not end there.

Claude Shannon, one of Bush's students, added to his meagre income by working part-time on Bush's analyser. One day, when discussing the problems they were having with relay circuits in the machine (which always seemed to need repairing), Bush suggested it might be possible to set down on paper exactly how the circuits worked, using symbolic logic to do so. (Symbolic logic had been developed by Alfred North Whitehead and Bertrand Russell, who in their monumental work **Principia Mathematica** took the work of George Boole - which we discussed earlier - and

from it constructed a solid thesis that logic was bound up in, and had its genesis in, mathematics.) Encouraged by Bush, Shannon started to look at the problem and eventually produced a very important paper: "A Symbolic Analysis of Relay and Switching Circuits".

The paper clearly explained how circuits could be wired to carry out mathematical tasks, such as adding and subtracting, and demonstrated that the same processes could be used for decision-making, following the thinking of symbolic logic. Shannon also discussed the enormous advantage binary numbers had over decimal numbers when a machine was needed to work with them. After he left college, Shannon went to work at Bell Telephone Laboratories, and began developing the theoretical concepts which became the base upon which modern computers were developed.

Also employed at Bell at the time was another mathematician, George Stibitz, who while working at home one day putting together circuits involving telephone relays, small lights and batteries, realised that the 'on' position of a relay could be used to represent the '1' in a binary number, while the 'off' position could be used to represent a '0'. It was the work of less than an hour to draw up, and construct a circuit which added two very small binary digits, carrying a digit where necessary into the next column (as you carry the tens digit into the next column when adding up a row of figures), and gave the result in terms of small lights. At that moment, on his kitchen table, Stibitz had built the cornerstone of the world's first true electronic calculator.

Stibitz drew up the design of a general purpose calculator, which appears to be fully workable, although he did not get around to building it. Later, Stibitz gave a talk on computers at Dartmouth College (where, some 25 years later, the language BASIC which your computer uses, was developed) and in the audience was one John Mauchly. It appears that the lecture was one of the factors which spurred Mauchly to take up the path that led him to eventually building the world's first proper electronic computer. We'll be looking at Mauchly's contribution shortly.

A young associate professor at Harvard, Howard H Aitken, now enters our story. Backed by the resources of IBM, Aiken's task was to build a computer which used magnetic relays. Aiken had come to the conclusion that punched card mechanical adding machines could be used to help produce a totally automatic computation machine.

And so he built one, the Automatic Sequence Controlled Calculator (ASSC), which was in fact the world's first true computer. It had taken over a century for Babbage's dream of 1832 to come to life.

ASCC used mechanical cogs, controlled by electrical relays, and information was fed into the unit via holes punched in paper tape. Constants could be set via switches on the front of the machine, or via the paper tape. Output from ASCC came as punched cards or in automatic typewritten form. Although it was very slow, taking some ten seconds to perform a division, it was the world's first completed computer, and it operated for some 15 years, day and night, at Harvard.

Meanwhile, in Britain, under the spur of war, a team of very bright men built a number of computer-like devices to help crack codes generated by the German Enigma Machine. It is believed that cracking the codes won the war for the Allies.

The Colossus, as the final code-breaking device was called, used paper tape for input and could accept some 5000 characters a second for processing. While they were aware that the code-cracking needs had spurred the development of their machines, three of the team (Turing, Good and Mitchie) soon realised that what they were doing in the development of 'intelligent machines' had implications far grander than the solving of the particular problem which had brought them together might suggest.

Their discussions led eventually to a number of published papers, among them the one in which the 'Turing Criterion' was first put forward. The Criterion for 'machine intelligence' was published by Turing in 1950. He said then that if you were dealing with something at the end of a wire that could be a machine or could be human, and you could not tell - from the responses coming to you over the wire - whether that with which you were dealing was human or machine, the 'thing' at the other end was, by definition, intelligent.

In America, military needs also came to bear on the development of computing devices. At the Moore School of Electrical Engineering, a top secret project under the control of John Mauchly (who was, you'll recall, present at the talk given at Dartmouth in 1940 by Stibitz) got under way to build a device which could produce the ballistic tables

for new weapons, a process which consumed much labor and time. Urged by a young engineer, J Presper Eckert, to use tubes in the design of any machine they proposed, Mauchly eventually came up with a plan in 1942 for a device which would cost $400,000.

A little earlier we discussed the ASCC which, despite its electromagnetic clutch relays, was basically a mechanical device. The first electronic computer - the Electronic Numerical Integrator and Calculator (ENIAC) - was built by Mauchly and Eckert, and was running tests only a year after ASCC was completed, and was finished the following year. ENIAC found its first employment with the military, working out bomb and shell trajectories.

ENIAC used no moving parts, except for those which accepted information from the operator, and those which output the results of the machine's deliberations. The electronics of the day were far from reliable, and the 18,000 tubes rarely performed for an hour without a breakdown. But even an hour's continuous work was worth the effort of keeping ENIAC going, because in that hour ENIAC could perform work which would take ASCC a week.

Whereas computers today are told what to do (that is, programmed) via electronic impulses generated in most cases initially by a keyboard, ENIAC had 'hard-wired' programs, with the route of the wires representing the steps needed to solve a problem either soldered into place, or connected via a maze of plugs and sockets.

In 1945, one of the ENIAC team met by chance (on a railway station) a leading mathematician, John Von

Neumann, who had worked on the original atom bomb, and found his work on future atomic weapons hampered by delays in getting routine figure work completed. He joined the Moore School group in an advisory capacity shortly after this meeting, just as they were starting work on ENIAC's son and heir, EDVAC.

Von Neumann's real contribution was to suggest that the computer **store programs within itself.** Now this may not seem very radical, but it marked a major change from the way all previous computing devices had worked. Before the Von Neumann idea, which was first applied in the US in EDVAC, programs were either 'hardwired' and thus were extremely difficult to change, or the instructions were fed in, where and when needed, via punched tape or cards.

Von Neumann's idea was simple, but profound, and affected the evolution of all computers from that point. Let the program be fed in, and then stored by the computer in its own memory. The computer could, of course, store more than one program at a time, so it was able to call up programs from within other ones, or switch to the program required virtually instantaneously.

These additional abilities multiplied the potential speed of the computer many, many times. The computer could now be seen less as a dedicated slave working blindly on a single problem, and more as a servant with a host of abilities, able to turn to each new task with just the right intellectual equipment for the job.

Despite the fact that Von Neumann is credited with inventing the concept of the stored program, and

despite the fact that it was from EDVAC that came the impetus to ensure all future computers would use the idea, two British computers - the Mark 1 at Manchester University and EDSAC at Cambridge University - were actually up and running, using a form of stored program access, two years before EDVAC got underway.

It was at this point, just as 1950 dawned, when the computer moved from being a one-off, largely experimental device to become a consumer product. In America, large firms such as Bell Telephone, Sperry-Rand and (of course) IBM began work on computers which would be sold as products. With some of these firms, the computer would eventually fade away as a product line. With others, such as IBM, the computer engulfed any other business in which the firm had been engaged.

Work continued in Britain, with Cambridge and the National Physical Laboratory (where Turing worked) leading the rest of the country, and in some areas, the rest of the world. Lyons (of Teashop and food fame) even got into the act, developing its own LEO system which was used within the company for routine adminstration from 1954.

Despite the developments made by these companies, the size of computers, their complexity, their unreliability and their cost put them well outside the reach of small businesses, and the thought of people actually owning a computer at home was barely even hinted at outside the realm of science fiction.

In fact, even as late as 1970, the book 'Man and the Computer' (Charles Scribner's Sons, New York),

written by John Kemeny, president of Dartmouth College and one of the developers of BASIC, included the wonderful phrase: "We may assume that by 1990 even a small business may have its own terminal". "Even" and "terminal". As late as 1970 (not long ago in 'history of earth' terms, but a time gulf away from the present in terms of the history of computers) one of the key people involved in making computer programming accessible to the public at large saw that one of the principle developments in computers would be that small businesses might - in 20 years - have access to computing power, via a terminal.

Of course, Kemeny's prediction was made from a climate in which time-sharing systems (a remote, generally very big, computer servicing a large number of users, by direct linkage over telephone wires to far-off keyboards) were all the rage, and there was no indication that the situation would change radically in the foreseeable future.

However, the seeds of the revolution which has put your computer in your home had been planted much earlier, in 1948, when three men working at Bell Laboratories (John Bardeen, Houser Brattain and William Bradford Shockley) invented the transistor, and scored the Nobel Prize for Physics for this invention. Dependence on tubes, on heat removal, on an army of repair technicians, on massive power supplies and specially reinforced floors to take the massive weight of the computer was removed at a stroke.

The concepts which lay behind transistor operation were not new. Over a century before the

Bardeen/Brattain/Shockley invention, Sir Michael Faraday had noted that the electrical conductivity of some materials changed when their temperature varied, and in 1874, a solid state radio signal detector was developed by Ferdinand Braun, professor of physics at Marburg. (Devices such as transistors, which are solid matter, in contrast to tubes which contain 'empty space', are called 'solid state' devices.) 'Crystal set' radios were early solid state decices, but when the apparently much more flexible vacuum tube was developed in the early 1900's, attention moved away from them and was concentrated on tube improvement.

The computer field grew massively during the fifties and sixties. Spurred on by the space program, which demanded that anything carried on board a space craft be as small as possible, Americans worked steadily at compressing the size of their computers.

When transistors were first developed, apart from the obvious facts that they were smaller than tubes and needed far less power, there was no real change in the way electronic devices were made. Essentially, a number of components were soldered onto a base, and connected by wires much as had tube devices been constructed. As solid state devices became more complex and each transistor took the place of more than one component, a trend developed to etch the 'wires' (as lines of solder-like conducting material) directly onto a plastic base, rather than handwire components onto a chassis. The circuits became more involved and smaller, but were still a far cry away from modern circuits. The circuit was etched from a large diagram which was photographically reduced.

The famous 'chip' (several of which lie within the case of your computer) was developed when it was realized that, with care, the circuit could be reduced a hundredfold, then over a thousandfold, using the base material to hold the components and their connections.

A series of tiny etchings, on pieces of very pure silicon, are now put together in a little stack which forms an entire computer circuit, complete with the electrical equivalents of many thousands of switches. Each layer acts as an electrical microcircuit, and when up to 20 or so are stacked together, an entire computer is created, needing only input and output devices, and a power supply to function intelligently.

Next time you are in London, you might like to make a point of visiting the Science Museum, where part of Babbage's original 'engine' is preserved, in a marvellous hall full of early calculators and computers.

Each time I visit that room in the museum, I pause for a moment before the Babbage cogs and wheels and reflect on the heartbreak they represent. None of Babbage's mechanical ideas are in your computer, but it bears witness to the soundness of the speculations of men like Babbage and Boole.

With what delight Babbage would seize your HX20, rush with it to the rooms of the Analytical Society in Cambridge, and while waving it in the air, say: "I was thinking that all those logarithm tables might be calculated by machinery. And I have found the most wonderful engine to do it...".

CHAPTER FOUR - FIRST STEPS IN PROGRAMMING

The BASIC language used by your computer is fairly easy to master. At the very least, you should be able to program with some facility within half an hour from now, even if you have never programmed before. Mastery of programming, of course, will take much longer.

We will follow a simple routine in working through the commands available on your computer. Firstly, a word which the computer recognises will be briefly explained, and then it will be shown in use. Short programs will be introduced, combining the current word with earlier learned words, so you can see how different words can be used to build up a computer program. From time to time, we will have much more major programs. Many of these major programs could, in fact, be entered by you right now, so that even if you do not fully understand how they work, you could still enjoy running the program.

PROGRAM LINES

All computer programs in BASIC (the computer language we are learning) are composed of a series of lines, each of which begins with a number. In general terms, the computer executes a program in order, starting with the lowest line number, and proceeding to the highest. From time to time, however, the computer will redirect action within the program (using such commands as GOTO). This will become clear very shortly.

From now on, I'll assume that your computer is turned on and running, and that you're going to

enter each program as you come to it. You'll find that you gain far more from the book, and will learn to program much more quickly, if you go to the trouble of entering the programs as you come to them, trying out the exercises described. This is a self-teaching text, and it requires some application from you if it is to do its work effectively.

PRINT, LIST, NEW and RUN

PRINT is one of the most often-used words in BASIC programming. It means more or less as you would expect. A PRINT statement tells the computer to PRINT something on the screen. Turn on your HX20, and once the menu appears (the menu is the initial display you'll get when you turn the machine on), touch the 2 key. The copyright notice will then appear, and below it you will see a symbol like a 'V' turned sideways. Underneath this is a little line. This line is known as the cursor. If you just type anything into the computer, you'll see the cursor moves one step ahead of you.

Press the RETURN key. The message 'SN Error' (syntax error) will appear, indicating that you've tried to get the computer to understand something it has not been programmed to understand.

The word PRINT, and the other words (such as RUN, NEW and CLS) which we'll be studying in this section of the book, are words the computer has been programmed to understand. Whenever you get a 'SN Error', check carefully to make sure you have spelt the relevant word correctly. Other things, as well as incorrectly spelt words, will give rise to the

'SN Error', but this is one of the more common reasons for it appearing.

Next to show PRINT in action, type in PRINT "TEST" (getting the double quote marks from the 2 key, holding down the SHIFT key to get them). The word TEST should appear under your line PRINT "TEST", and below the TEST you'll see the sideways V over the line cursor. This indicates that the computer has finished the task you set for it, and is now waiting for further instructions. Now, enter the following, pressing RETURN at the end of the each line.

```
10 PRINT "AMERICA"
20 PRINT "THE"
30 PRINT "BEAUTIFUL"
```

Once you've got it in your computer, type in the word RUN and press RETURN again. As you can see, the RETURN key is pressed every time you want the computer to act on what you have just entered.

You should something like the following on your screen:

```
AMERICA
THE
BEAUTIFUL
```

Now there is quite a bit we can learn from this short program. Firstly, as I pointed out a little earlier, program lines are numbered, and the program tends to run from the lowest program line up to the highest. There is no compulsion to number the program lines in tens, starting at ten, but you'll find it a good habit to get into, as numbering programs in tens gives you room to add new lines in between those you have already entered.

The PRINT statement determines how information will be printed on the screen. Change the second line of your program to read like the following line. You change lines simply by typing the whole line in again, with the required changes, then pressing RETURN, which makes the new program line take the place of the old one. There is, in fact, another way to modify program lines which does not demand you retype the whole line, but it could be a bit confusing to introduce that at this stage, so we'll come back to it a little later. Anyway, type in the following new line 20:

```
20 PRINT ,"THE"
```

This time, when you RUN the program, you'll notice the program output is as follows:

```
AMERICA
                THE
BEAUTIFUL
```

The comma before the quote marks in line 20 has moved the PRINT position across the screen.

You can use the comma in this way to format the print output, so - for example - you get a row of numbers all neatly lined up.

Now, get rid of that program in your computer by typing in the word NEW. You'll see that the cursor reappears almost instantly. Typing in NEW, then pressing the RETURN key, instantly clears the contents of the computer's memory. (Note that there is no way of getting these contents back after using NEW, so the command should be applied with care.)

You can check that the computer is empty by entering the word LIST, then pressing the RETURN key. You'll

see the cursor reappear. If there was a program in the computer, it would appear on the screen, line by line. We'll demonstrate this in a moment.

Before we do that, however, enter and RUN the following program:

```
10 PRINT "THIS IS"
20 PRINT "A TEST"
```

You should get this result:

```
THIS IS
A TEST
```

Now, retype line 10 so it reads as follows:

```
10 PRINT "THIS IS ";
```

You'll see that there is a space between the letter S and the closing quote marks, and there is a semi-colon (;) after the closing quotes. Now run the program again, and you should get this result:

```
THIS IS A TEST
```

As you can see, the words all run along on the same line. This is because the semicolon joins the end of one PRINT statement to the following one.

TAB, and a little more on LIST

The TAB (for 'tabulate') command, allows for very precise positioning of the start of a line of PRINT, and must be followed by a number in parantheses. This number dictates how many spaces across the screen the PRINT statement which follows it will be printed.

```
10 PRINT TAB(2);"2"
20 PRINT TAB(7);"7"
30 PRINT TAB(4);"4"
```

The 'control variable', the number within the parantheses, can be a variable, rather than a number, as the following program demonstrates (note that the term 'variable', and the use of things like the FOR in line 10, will be discussed in due course):

```
10 FOR J=1 TO 12
20 PRINT TAB(J);J
30 NEXT
```

The control variable can also be an expression:

```
10 A=2
20 B=9
30 PRINT TAB(A+B);B
```

Once you have this program in your computer, remember you can check it by typing in the word LIST and then pressing RETURN. Doing this will make the listing appear on the scren. This use of LIST is OK is your program is only three lines long, but if it is longer, you may want to study the lines as they go past, and you may feel the lines scroll past you too quickly to be studied carefully.

Above the 2 and 3 keys, you can see a red key marked BREAK. You can use this when a listing is scrolling past, or at almost any time during the execution of a program, to halt the computer's current action completely. If you do this, the message 'Break in 30' will come up, with the number of the line being executed when you pressed BREAK coming up on the screen in the place of the '30'.

However, once you've done this, you'll have to type in LIST again to get the computer to get on with the listing.

Another way of stopping the listing, so that you have time to study it properly, is to use the grey key marked PAUSE which is two to the left of BREAK. You'll see that when you do this, the listing will stop rolling past until you press another key, or the space bar.

LIST can be used to get just part of the listing. If you have, for example, a program of 10 lines, each numbered in tens (so the line numbers would be 10, 20, 30 and so on up to 100), you could get the last four lines of the program to appear on the screen simply by entering:

```
LIST 70 -
```

This lists all the lines following the hyphen, so this example would get the computer to list lines 70, 80, 90 and 100. In a similar way, to get all the lines up to a particular number, you use the following form:

```
LIST - 40
```

This will list all the lines from the start of the program up to and including the one you have specified.

If the lines you wish to study are not at the start or the end of the program, you can get them up on the screen by entering a command in the following form:

```
LIST 20 - 60
```

This will list lines 20, 30, 40, 50 and 60.

Another variation on LIST is LLIST which, instead of listing the program on the screen, lists it to the printer. Try it with the three-line program which you still have in your computer. Make sure the switch to the left below the printer, marked "PRINTER OFF ON" is turned to ON, then type in LLIST, to get the listing out on the printer.

We'll look at LLIST again, as well as other commands for controlling the printer a little later. LLIST was mentioned here to complete the section on the use of LIST.

VARIABLES

Nearly all the programs you'll ever write or see written for your computer will use variables. Variables are letters, or combinations of letters and numbers starting with a letter, which are given values during the course of a program execution and are, in effect, those numbers during the run of the program.

To clarify that somewhat mysterious statement, look at the next small program segment:

```
10 A=36
20 B=9
30 C=A+B
40 PRINT A;"+";B;"=";C
```

In this program, a value of 36 is assigned to the variable A (and acts as though it was the number 36 right through the program), a value of 9 is assigned to variable B, and the total of A plus B is assigned to varible C, as can be seen in line 40. Variables used in this way are called numeric variables.

Variables can change their values during the course of a program, as this example shows:

```
10 A=INT(RND(1)*10)+1
20 B=INT(RND(1)*10)+1
30 PRINT "A is";A
40 PRINT "B is";B
50 FOR T=1 TO 500:NEXT
60 PRINT
70 GOTO 10
```

Although a variable can be any letter from A to Z, variable names are not restricted to single letters. Any combination of letters and numbers, so long as they start with a letter, is an acceptable variable name, so C2PO, R2DEE2 and FOXHUNTER are all valid names.

You'll find that the use of explicit variable names can help to make programs easier to understand. For example, in the next program, which works out what percentage a small number is of a large one, the larger number is assigned to a variable called BIGNUMBER, the smaller one to a variable with the name LITTLENUMBER and the result of the calculating is assigned to a variable called PERCENTAGE. (Note that every letter of the variable name is significant so GOSHOWOFF is not the same as GOSHOWOGF or GOSHOWOF.) Here is the program listing:

```
10 BIGNUMBER=761
20 LITTLENUMBER=234
30 PERCENTAGE=LITTLENUMB
ER*100/BIGNUMBER
40 PRINT PERCENTAGE;"%"
```

As you can see, it is very easy to understand what is happening. You are unlikely to use such long names all the time, but there is no reason why you

cannot use abbreviated versions of names like those used in the percentage program, which will still tell you, when you're looking through the listing, what the variables represent. Here, for example, is another version of the same program, with shorter variable names. Note that it is still quite clear what the shorter names refer to:

```
10 BIGNO=761
20 LITNO=234
30 PER=LITNO*100/BIGNO
40 PRINT PER;"%"
```

The names can be further abbreviated, but still retain a link with that which they represent:

```
10 BN=761
20 LN=234
30 PR=LN*100/BN
40 PRINT PR;"%"
```

Many computers will only accept (or take note of) double letter names for variables (like GD, AS or TP) so if you are writing programs which you want to be able to run on other makes of computers, it may be worth getting into the habit now of using double-letter variable names.

So far we have been looking at numeric variables. There are also string variables. In computer jargon, a string is anything which is normally enclosed within quote marks in a PRINT statement. It does not necessarily have to be in a PRINT statement, but if it was, and it was a string, it would be within quote marks.

Therefore, the following are strings:

"THE SUN IS HOT"

"THE ANSWER TO THE PROBLEM IS"

"HOW MANY ANGELS CAN DANCE?"

The only difference between the variable assigned to a string, as opposed to that assigned to a number, is that the string variable name ends with a dollar sign. The following are valid string names:

A$ FACE$ D123456$ G6H4$

Here's a program showing string variables in use. Note that while the string itself must normally be enclosed within quote marks, the variable name does not have quote marks around it, even when used in a PRINT statement:

```
10 NAME$=" EBERNEZER"
20 VERB$="IS"
30 A$="FOR THE SENATE"
40 B$="THE CANDIDATE"
50 PRINT B$
60 PRINT A$
70 PRINT VERB$;NAME$
```

You'll notice that numeric variables are widely used within the programs in the first part of this book. String variables are not so common. Strings and string manipulation are discussed in detail a little later on in this section of the book, but you know enough now to recognise them when you come across them.

LOAD/SAVE (and TAPCNT)

Instead of typing in your programs each time you want them, you can save them on tape. To save a program on cassette, use the WIND command to move the tape to a new position. I find it best to work in thousands, and keep a written record of what is at each 1000 mark.

Simply type in WIND 1000 or WIND 2000 or whatever, and when the tape stops, type in SAVE "NAME", replacing "NAME" with whatever you've decided to call your program. Then, to load the program back into your computer, either start at the beginning of the tape with LOAD "NAME" or, if you know where your program begins, enter WIND 1000: LOAD "NAME" as a single command, then press RETURN. The computer will move to the designated part of the tape, and then it will load the program.

If you're not sure where you are on the tape, the command PRINT TAPCNT (standing for tape count) will print up the location. Try the following, to show TAPCNT. Type in WIND RND(1)*100 and press RETURN. Once the tape has stopped, type in PRINT TAPCNT and press RETURN again. You should get a number between 1 and 100.

CHAPTER FIVE - MORE PROGRAMMING AND THE MYSTERY OF LIFE

CLS

The next command we will look at is CLS. This stands for Clear the Screen, and is used when you wish to get rid of a program listing when a program starts running, rather than just have the listing scroll upwards as the program output begins appearing on the screen.

The following program - STAR COLONY - shows CLS in action. (It also contains a number of other program words which you will probably not understand at this point. They will be explained as the book progresses.)

STAR COLONY is based on the very popular program LIFE, which was first developed by John Conway when he was attending Cambridge University in the UK. The program seeks to simulate the birth, growth and death of a colony of cells. As we are using asterisks to indicate cells, and as these look vaguely like stars, I decided to call this variation of the program STAR COLONY.

Conway drew up the rules under which the stars in the colony evolve. You'll be pleasantly surprised to see just how effective these rules can be when applied, producing unexpectedly delightful designs.

The rules, which are applied by checking the stars surrounding each position on the screen, one by one. As a result of these checks, decisions are made as to whether or not there will be a star in that

position the next time the colony is reprinted. If you imagine that a particular star is sitting on a chess board, somewhere near the center of the board, you'll see that it has eight squares surrounding it. If, when the contents of the eight surrounding squares are counted, two or three stars are found and there is a star on the square which is surrounded by the other eight, this star will survive until the next generation.

If there are three stars surrounding the square being checked, and the central square is empty, a star will be 'born' in that square in the following generation. If there are four or more surrounding stars, then the central star will die before the next generation is printed.

The rules are applied all over the screen at once, and once the full 'galaxy' has been assessed, the next galaxy generation is printed.

Enter and run the following program, then return to the book for a discussion of it. Keep in mind that as well as the command CLS there are many other commands within the program. We'll be looking at such words as REM, GOTO and FOR/NEXT shortly, so it may be beneficial to try and begin to understand how they work in this program, so you'll be able to easily understand the explanations of them when they are presented.

Here, then, is the program listing for STAR COLONY:

```
10 REM Star Colony
20 CLS
30 DEFINT A-Z
40 PRINT"PRESS ANY KEY"
50 N=0
```

```
60 N=N+1:IF INKEY$=""THE
N 60
70 CLS
80 PRINT"Please stand by
"
90 RANDOMIZE N
100 DIM A(6,23),B(6,23)
110 FOR X=2 TO 5
120 FOR Y=2 TO 21
130 IF RND(1)>.5 THEN A(
X,Y)=1
140 B(X,Y)=A(X,Y)
150 NEXT:NEXT

170 GOSUB 200
180 GOSUB 310
190 GOTO 170
200 CLS
210 SOUND RND(1)*50,1
220 FORX=2 TO 5
230 FORY=2 TO 21
240 A(X,Y)=B(X,Y)
250 IF A(X,Y)=1 THEN PRI
NT"*";
260 IF A(X,Y)=0 THEN PRI
NT" ";
270 NEXT
280 IF X<5 THEN PRINT
290 NEXT
300 RETURN
310 REM New Galaxy
320 FOR X=2 TO 5
330 FOR Y=2 TO 22
340 C=0
350 IF A(X-1,Y-1)=1 THEN
 C=C+1
360 IF A(X-1,Y)=1 THEN C
=C+1
370 IF A(X-1,Y+1)=1 THEN
 C=C+1
380 IF A(X,Y-1)=1 THEN C
=C+1
390 IF A(X,Y+1)=1 THEN C
=C+1
```

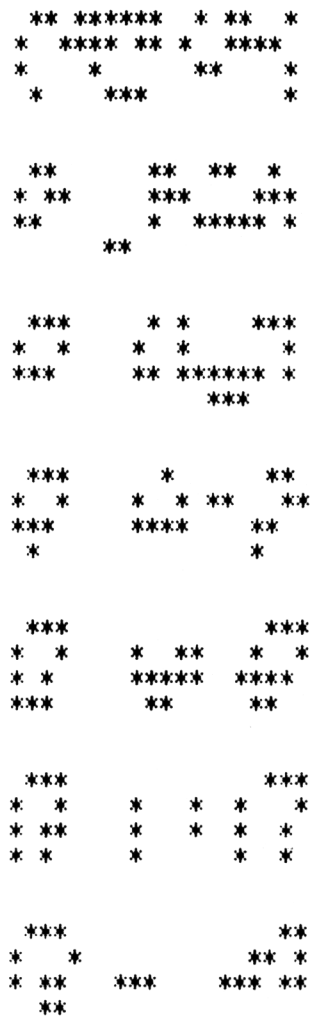
```
400 IF A(X+1,Y-1)=1 THEN
 C=C+1
410 IF A(X+1,Y)=1 THEN C
=C+1
420 IF A(X+1,Y+1)=1 THEN
 C=C+1
430 IF A(X,Y)=1 AND C<>2
 AND C<>3 THEN B(X,Y)=0
440 IF A(X,Y)=0 AND C=3
THEN B(X,Y)=1
450 NEXT:NEXT
460 RETURN
```

And here are some 'colonies' produced by it:

```
PRESS ANY KEY

Please stand by

* *  **   * ****  **
*   ********  * *  *
*** ***   * **** *
***  **   *    ***


 *  *   * * ****  **
* *     *       ** *
  *     *   **   **
* *** *    * * * *


 *       *   *******
  **    **  *   **  *
  *     *    ***
 ***          ** ***


  *      *      *  ***
 ***    **           *
        **   *     *
 ***         * ** *
```

PRESS ANY KEY

Please stand by



The program used to produce 'star colonies' on the printer is slightly different from the version to do so on the screen. Here is the listing to dump them to the printer:

```
10 REM Star Colony
20 CLS
30 DEFINT A-Z
40 LPRINT"PRESS ANY KEY"
45 LPRINT
50 N=0
60 N=N+1:IF INKEY$=""THE
N 60
70 CLS
80 LPRINT"Please stand b
y"
85 LPRINT
90 RANDOMIZE N
100 DIM A(6,23),B(6,23)
110 FOR X=2 TO 5
120 FOR Y=2 TO 21
130 IF RND(1)>.5 THEN A(
X,Y)=1
140 B(X,Y)=A(X,Y)
150 NEXT:NEXT

170 GOSUB 200
180 GOSUB 310
190 GOTO 170
200 CLS
210 SOUND RND(1)*50,1
220 FORX=2 TO 5
230 FORY=2 TO 21
240 A(X,Y)=B(X,Y)
250 IF A(X,Y)=1 THEN LPR
INT"*";
260 IF A(X,Y)=0 THEN LPR
INT" ";
270 NEXT
280 IF X<5 THEN LPRINT
290 NEXT
300 RETURN
```

```
310 REM New Galaxy
320 FOR X=2 TO 5
330 FOR Y=2 TO 22
340 C=0
350 IF A(X-1,Y-1)=1 THEN
 C=C+1
360 IF A(X-1,Y)=1 THEN C
=C+1
370 IF A(X-1,Y+1)=1 THEN
 C=C+1
380 IF A(X,Y-1)=1 THEN C
=C+1
390 IF A(X,Y+1)=1 THEN C
=C+1
400 IF A(X+1,Y-1)=1 THEN
 C=C+1
410 IF A(X+1,Y)=1 THEN C
=C+1
420 IF A(X+1,Y+1)=1 THEN
 C=C+1
430 IF A(X,Y)=1 AND C<>2
 AND C<>3 THEN B(X,Y)=0
440 IF A(X,Y)=0 AND C=3
THEN B(X,Y)=1
450 NEXT:NEXT
455 LPRINT:LPRINT:LPRINT
460 RETURN
```

As you can see, this listing is very similar to the first one given. All you need to do is modify lines 30, 80, 250, 260 and 280; and add 45, 85 and 455.

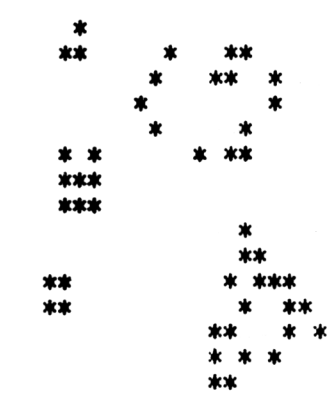
The colony size was set at four elements down and 20 across, because of the size of the screen. There is no reason why, if we are using the printer or an external screen to accept the program output, we cannot use a larger grid. One that is 20 by 20 works very well. In fact, the result of running it is more effective than the four by 20 version,

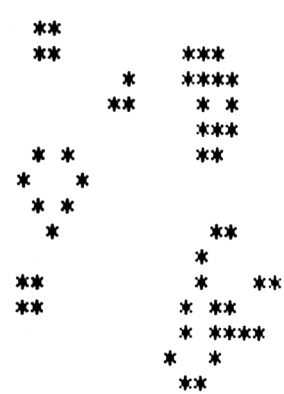
although it takes much longer to process each generation. Here is the further modification of the program, designed to produce a 20 by 20 grid on the printer:

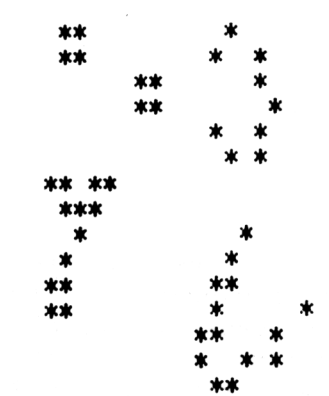
```
10 REM Star Colony -
   (large grid version)
20 CLS
30 DEFINT A-Z
40 LPRINT"PRESS ANY KEY"
45 LPRINT
50 N=0
60 N=N+1:IF INKEY$=""THE
N 60
70 CLS
80 LPRINT"Please stand b
y"
85 LPRINT
90 RANDOMIZE N
100 DIM A(23,23),B(23,23
)
110 FOR X=2 TO 21
120 FOR Y=2 TO 21
130 IF RND(1)>.5 THEN A(
X,Y)=1
140 B(X,Y)=A(X,Y)
150 NEXT:NEXT

170 GOSUB 200
180 GOSUB 310
190 GOTO 170
200 CLS
210 SOUND RND(1)*50,1
220 FOR X=2 TO 21
230 FOR Y=2 TO 21
240 A(X,Y)=B(X,Y)
250 IF A(X,Y)=1 THEN LPR
INT"*";
260 IF A(X,Y)=0 THEN LPR
INT" ";
270 NEXT
280 IF X<21 THEN LPRINT
290 NEXT
300 RETURN
```

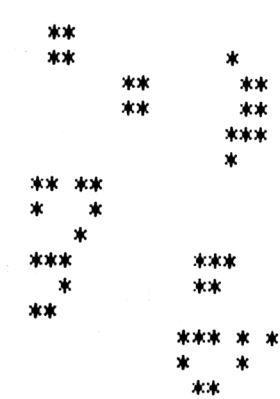
```
310 REM New Galaxy
320 FOR X=2 TO 21
330 FOR Y=2 TO 22
340 C=0
350 IF A(X-1,Y-1)=1 THEN
 C=C+1
360 IF A(X-1,Y)=1 THEN C
=C+1
370 IF A(X-1,Y+1)=1 THEN
 C=C+1
380 IF A(X,Y-1)=1 THEN C
=C+1
390 IF A(X,Y+1)=1 THEN C
=C+1
400 IF A(X+1,Y-1)=1 THEN
 C=C+1
410 IF A(X+1,Y)=1 THEN C
=C+1
420 IF A(X+1,Y+1)=1 THEN
 C=C+1
430 IF A(X,Y)=1 AND C<>2
 AND C<>3 THEN B(X,Y)=0
440 IF A(X,Y)=0 AND C=3
THEN B(X,Y)=1
450 NEXT:NEXT
455 LPRINT:LPRINT:LPRINT
460 RETURN
```

Because a much larger grid must be checked, this version of the program is very much slower than the two earlier versions, but the greatly improved results make the wait well worth while. To show this, here are some colonies in evolution, on the 20 by 20 galaxy:









Although it may be of limited interest and use at this point, I'll now go through the very first version of the program, more or less line by line, to outline what each line does.

I suggest you may wish to completely skip the explanation at this time, and return to it later on in your reading, after you've worked through some of the following section on programming. You'll find it will then make much more sense to you, and thus will be more valuable than it may be if you try to wade through it now, right at the beginning of your learning about computer programming.

We start with line 20, which clears the screen, removing whatever is left of the listing, and the word RUN from the screen. Line 30 ensures that all variables used, from A to Z, will be integer variables which ensures the programs runs more quckly than would be the case if the variables were floating point ones.

The routine from lines 40 to 60 sets up a loop which is only terminated when a key is pressed in response to the instruction "PRESS ANY KEY". Line 50 sets the variable N to zero, and line 60 increments this by one over and over again until the computer senses, through INKEY$, that a key has been pressed. The value of N obtained is used to seed the random number generator in line 90, thus ensuring that you get different colonies each time you run the program. Once you've pressed a key, line 70 clears the screen of the "PRESS ANY KEY" message, and replaces it with the words "Please stand by" while it creates the initial colony.

Line 100 dimensions two arrays, one to hold the colony which is currently displayed on the screen, and the other to hold the changes that are to be made to this colony for the following generation.

The routine from lines 110 to 150 sets up the starting colony. If the value of a particular element of the A array is zero, a blank (that is, no cell) will be printed. A value of one will cause a star-cell to be printed. Line 130 allots the 1's at random throughout the array, and the following line copies the contents of the A array, element by element into the B array. Note that the 'outside' elements of the array are not assigned, so the frame of the colony will always be blank. The elements in the frame are never checked, as the computer knows these are blank, and likewise are never printed.

Once the initial star colony has been generated, action goes to the subroutine starting at line 200. Another clear screen command is found here, this one for when the lines from 170 to 190 are traversed over and over again in an infinite loop. A beep is sounded by line 210 to alert you to the fact that a new colony is about to be printed, and the loop from 220 to 290 prints out the colony.

After a return to the '170 to 190' loop, action goes to the routine starting at line 310, where the stars surrounding each star on the grid are checked one by one, and the results of such checks stored in array B. You'll see that line 240, which is within the printout subroutine, ensures array B is copied into array A for the actual printout. Note that the number of neighbours for any cell is held in variable C.

Line 430 says, in effect: "If the square whose surroundings have been checked contains a star, and there are not either two or three stars in the surrounding eight squares, then that star should die for the next generation, its element within the array being set equal to zero".

In a similar way, line 440 says: "If there is no star in the sqaure whose surroundings are being checked, and the square has three neighbours, then a star should be born in that cell in the next generation. That is, its element within the array should be set equal to one".

After the return from that subroutine, the computer hits the GOTO message (line 190) which sends to back to line 170 to cover the two subroutines again. The process will continue until the BREAK key is pressed.

REM

The word REM stands for remark. This is a word or phrase which is used within a program simply to tell human beings looking at the program what the following section of the program is meant to be doing. REM statements are ignored by the computer, so you can use them to store any information which you feel could help you interpret what sections of the program are doing.

You'll find that REM statements are particularly useful when you return to a program after a long break. You may feel you will never forget what particular sections of a program do, and this belief will persist until you first try to fathom out the

workings of a program which you have not looked at for several days.

Here are two REM statements from the last version we gave of STAR COLONY:

```
10 REM Star Colony -
   (large grid version)
20 CLS

300 RETURN
310 REM New Galaxy
320 FOR X=2 TO 21
```

In these printouts, the REM statements are, of course, lines 10 and 310. Line 10 tells you what the program is, a fact which may not be immediately evident when you pick up one listing from a collection of ten or more of them. The second REM statement, line 310, explains that the following section of the program works out the status of the new colony.

Although you may feel REM statements are a bit of a waste of time with short programs, you'll find they come into their own with long programs. Certainly it is worth getting into the habit of using REM statements.

We'll now be looking at a number of commands, using the same program to show them in action. The program is 'Number Race', a title which will make sense once you've run the program. When you do, you'll see the numbrs 1 to 4 race each other across the screen, like four tiny horses.

At the beginning of the program, which is intended for two human beings (although a version in which you play against the computer can easily be created), each player has $100. The two players each

bet $5 on their choice of a horse to win the race, selecting one of the numbers 1 to 4 for a win. If the selected horse does win, $10 is paid.

The game continues, with race after race, until one of the players is broke. At this point, the solvent player is declared the winner.

CONT, GOTO

I suggest you enter the following program now, run it a few times, and then return to this text where the programming words CONT and GOTO are described.

```
10 REM Number race
20 DIM A(4),Z(2),X(2)
30 N=1:X(1)=100:X(2)=100
40 CLS
50 FOR T=1 TO 2
60 X(T)=X(T)-5:IF X(T)<1
 THEN 420
70 PRINT:PRINT
80 PRINT "Player";T;":"
90 PRINT "you've got $";
X(T)
100 PRINT TAB(2);"Which
number"
110 PRINT "will win (1 t
o 4)?";
120 LET A=VAL(INKEY$)
130 N=N+1:SOUND N MOD 50
,1
140 IF A<1 OR A>4 THEN 1
20
150 Z(T)=A
160 IF T=2 AND Z(1)=Z(2)
 THEN SOUND 1,1:PRINT:PR
INT "Number 1 has";Z(1):
FOR D=1 TO 1000:NEXT:GOT
070
170 NEXT
```

```
180 RANDOMIZE N
190 CLS
200 FOR T=1 TO 4:A(T)=T/
16:NEXT
210 FOR B=1 TO 4
220 LET A(B)=A(B)+RND(1)
230 PRINT TAB(A(B));B;
240 IF B<4 THEN PRINT
250 IF A(B)>17 THEN 300
260 SOUND A(B),0.5
270 NEXT
280 CLS
290 GOTO 210
300 PRINT:PRINT"The winn
er is";B
310 IF Z(1)=B THEN PRINT
 "And player 1 got it ri
ght":X(1)=X(1)+10
320 IF Z(2)=B THEN PRINT
 "And player 2 got it ri
ght":X(2)=X(2)+10
330 FOR T=1 TO 5 STEP RN
D(1):SOUND T,.5:NEXT
340 PRINT:PRINT
350 PRINT "Stand by"
360 FOR T=1 TO 5 STEP RN
D(1):SOUND T,.5:NEXT
370 PRINT "for a new"
380 FOR T=5 TO 1 STEP -R
ND(1):SOUND T,.5:NEXT
390 PRINT ".......RACE!!
"
400 FOR T=50 TO 45 STEP
- RND(1): SOUND T,.5:NEX
T
410 GOTO 40
420 PRINT:PRINT "Well, p
layer";T
430 PRINT "is broke, so
the"
440 PRINT "winner is";
450 IF T=1 THEN PRINT 2
ELSE PRINT 1
```

We'll look at the word CONT first of all. CONT stands for continue, and is used when - for some reason - you stop the program, and then wish to continue execution. As you have seen from running 'Number Race', after the two players have selected their 'horses', the four numbers run across the screen, emitting little beeps as they do so.

Run the program again, and once the race part of the program is underway, press the BREAK key. A message like 'Break in 230' will appear. Now type in CONT and press RETURN and you'll find the race picks up from the point where you stopped it. So CONT is very easy to understand. It stands for continue and is used whenever you wish to start execution again after BREAKing for some reason.

Note that you cannot use CONT if you change the program in any way before attempting to continue program execution.

GOTO

It is pretty obvious what GOTO means and does. You'll recall that I said earlier that programs in BASIC tend to be executed from the smallest line number to the highest. GOTO is one of the commands which allow you to break this orderly execution sequence from lowest to highest. When the computer comes across the word GOTO, it goes to the line number indicated.

You've probably realised that, while the race is being run, a small part of the program is repeated over and over again, much as there were parts of the STAR COLONY program which were repeated. If you

look at the sequence from lines 210 to 290, you'll see that these are the parts of the program which are executed time and time again while the race is underway. Line 290 (GOTO 210) is the one which sends action back to the beginning of the race loop. Line 410 (GOTO 40) sends action back to line 40 after one race has been won to collect the bets for the next race. Lines 290 and 410 are known as 'unconditional GOTO statements' because they always redirect action within a program.

We'll be looking in detail at the pair of words IF/THEN a little later in the book. For now, all you need to know is that these words impose conditions on the execution of the statement which follows them.

It is not necessary to include the word GOTO after a condition, as the computer will understand that it should be there.

That is, if the line means...

```
IF X = 3 THEN GOTO 50
```

...you can simply write...

```
IF X = 3 THEN 50
```

implied. Look at line 250. This checks the value of a variable called A(B) and if it finds it is more than 17, then tells the computer to GOTO line 300. As you can see, the word GOTO is not there after the THEN. The computer, however, understands the instruction which is intended, from the context in which it appears. You can see that the computer

does understand this from the action it takes when it finds A(B) is greater than 17. (You could also check it, I guess, by inserting the word GOTO in line 250 so that it comes between the 17 and the THEN. You'd see then it had no effect on the way the line operated.)

We'll be returning to the 'Number Race' program shortly, so save it on tape now so you'll be able to reload it later when it is needed.

ON...GOTO

This is a variation on the GOTO statement, which is used when you want to GOTO a series of destinations, depending upon the value held by a particular variable.

Here is a simple program, which shows ON...GOTO in action:

```
10 REM ON...GOTO DEMO
20 A=INT(RND(1)*4)+1
30 FOR B=1 TO 200:NEXT
40 ON A GOTO 50,70,90,11
0
50 PRINT "ONE"
60 GOTO 20
70 PRINT "TWO"
80 GOTO 20
90 PRINT "THREE"
100 GOTO 20
110 PRINT "FOUR"
120 GOTO 20
```

Line 20 generates the numbers 1, 2, 3 or 4 at random, assigning them to the variable A. Line 40, the important one for this discussion, sends action to 50 if A equals one (50 is the first number after GOTO), to 70 is A equals two (70 is the second destination), to 90 if A equals three and to 110 if A equals four. Line numbers 50, 70, 90 and 110 spell out the number in full, effectively changing a randomly-generated digit into a word.

ON...GOTO can also be very effective for 'writing' poetry as the next program shows.

```
10 REM ON..GOTO POETRY
20 CLS
30 A=INT(RND(1)*13)+1
40 FOR B=1 TO 200:NEXT
50 ON A GOTO 60,70,80,90
,100,110,120,130,140,150
,160,170,180
60 PRINT "LINGER ";:GOTO
 30
70 PRINT "ONLY ";:GOTO 3
0
80 PRINT "HEART ";:GOTO
30
90 PRINT "LOVERS":GOTO 3
0
100 PRINT "WANDERING":GO
TO 30
110 PRINT "PEACEFUL":GOT
O 30
120 PRINT "REACH OUT":GO
TO 30
130 PRINT "SAD ";:GOTO 3
0
140 PRINT "EYES":GOTO 30
150 PRINT "LIMPID ";:GOT
O 30
```

```
160 PRINT "WAITING":GOTO
 30
170 PRINT:PRINT:GOTO 30
180 PRINT "...";:GOTO 30
```

As you can see, the random number generated in line 30 directs the computer to various lines between 60 and 180, using ON...GOTO. The end of each print line contains a GOTO 30 which sends action back to get a new random number. Line 40 uses a dummy loop to slow things down enough to allow you to read the 'poems'. By changing all the words PRINT into LPRINT, you can get the computer to dump the result of its creative genius directly onto the printer, to create results like these:

```
SAD HEART SAD EYES
ONLY WAITING
LINGER ...WANDERING
SAD PEACEFUL
WANDERING
REACH OUT
ONLY ...ONLY LOVERS
LOVERS
ONLY PEACEFUL
SAD WAITING
ONLY REACH OUT


HEART SAD SAD EYES
LIMPID SAD PEACEFUL
WANDERING
LINGER REACH OUT
PEACEFUL
WANDERING
EYES
EYES
```

CHAPTER SIX - LOOPING THE LOOP

We've discussed CONT, GOTO and ON...GOTO, and now we'll look at words which are nearly always paired in programs, FOR and NEXT.

Essentially, FOR and NEXT control a loop which is executed the number of times specified in the FOR statement. The following program fragment should help make it clear:

```
10 REM FOR/NEXT
20 FOR A=1 TO 10
30 PRINT "THE US OF A NO
."; A
40 NEXT A
```

When you run it, you'll get this result:

```
THE US OF A NO. 1
THE US OF A NO. 2
THE US OF A NO. 3
THE US OF A NO. 4
THE US OF A NO. 5
THE US OF A NO. 6
THE US OF A NO. 7
THE US OF A NO. 8
THE US OF A NO. 9
THE US OF A NO. 10
```

THE US OF A NO. is printed out ten times, with each line ending with the value A has during that execution of the FOR/NEXT loop. Once you've run it a few times, change line 40 to read as follows:

```
40 NEXT
```

That is, you remove the A, the 'control variable'.

The computer knows which FOR the NEXT refers to, so the 'A' is not needed. However, when you have a number of loops, you may prefer to have the NEXTs paired with FORs, just so you can keep track of what is happening in the program.

You can nest loops (that is, have one or more inside the other) if needed. Here is an example of one loop nested inside another - the B loop is nested within the A loop. This routine prints out the multiplication tables from one times one to twelve times twelve.

```
10 REM NESTED LOOPS
20 FOR A=1 TO 12
30 FOR B=1 TO 12
40 PRINT A;"TIMES";B;"="
;A*B
50 NEXT B
60 NEXT A
```

It is important that the last NEXT that occurs in the program has the same control variable (the letter A, B or whatever) as the first FOR which is in the program. As you can see in the program above, the first FOR is A, and the last NEXT in the program is also A. The next control variable mentioned in the program is B, and the second last NEXT is also B. It is vital that you arrange your FOR and NEXT controls in this way, or the computer will get a little hot under the collar. You can demonstrate this by changing lines 50 and 60 to the following:

```
50 NEXT A
60 NEXT B
```

This will give rise to the error message 'NF Error in 60' (no FOR...). One way to get around this is to

remove the control variables from the NEXTs and let the computer sort them out. In this case, the final two lines of the program would read:

```
50 NEXT
60 NEXT
```

Another way of doing it is to delete line 60 (which you do by typing in 60, with nothing following, then pressing RETURN) and change line 50 to this:

```
50 NEXT B,A
```

As you'll see when you try this, the program works exactly the same in all cases, except when you get the control variables in the wrong order. You'll find the computer works fractionally more quickly when it does not have to check the correctness of the control variables. Therefore, the use of NEXT, without a control variable, is probably the simplest way to use FOR/NEXT loops. However, I suggest that now - at the beginning of your acquisition of programming skills - you stick to the version given in the first program, using NEXT B and NEXT A.

This will help you keep track of what is going on within the program.

FOR/NEXT...STEP

There is another aspect of FOR/NEXT loops which must be discussed, and that is the use of STEP. Clear your computer's memory by typing in NEW, then pressing RETURN, and enter the following program:

```
10 REM FOR/NEXT...STEP
20 FOR A=1 TO 20
30 PRINT A
40 NEXT A
```

When you RUN this, you'll see the numbers 1 to 20 printed on the screen. Now, change line 20 so it looks like this:

```
20 FOR A=1 TO 20 STEP 2
```

When you run this, instead of the computer printing out 1, 2, 3, 4, 5, 6... and so on down to 20, you'll see 1, 3, 5, 7, 9...down to 19. The computer has started counting from the first number quoted in the control statement (FROM A = 1 TO ...) to the closest it can get to the final number, counting up in two's. Change line 20 to this, and RUN the program again:

```
20 FOR A=1 TO 20 STEP 3
```

This time you should see 1, 4, 7, 10...etc.

So, from this we can learn that FOR/NEXT loops will step up in ones, unless another STEP size is specified. The STEP of one is the default case, which occurs when no other STEP size is specified.

The computer can also count downwards. There is no reason why a FOR/NEXT loop must go upwards. Try the following to see this:

```
10 REM FOR/NEXT...STEP
20 FOR A=20 TO 1 STEP -1
30 PRINT A
40 NEXT A
```

The STEP size does not have to be a whole number, as you'll discover if you change line 20 to this:

```
20 FOR A=20 TO 1 STEP -0.7
```

It is time now to return to our 'Number Race' program, to look at the FOR/NEXT loops in it. Either load the program back in, or refer to the listing given in the previous chapter.

The first loop, a 'T' loop, starts in line 50 and ends in line 170. Note that line 170 just reads NEXT. It could read NEXT T, but, as was pointed out, the computer knows which FOR the NEXT is referring to.

The first action within the loop is to subtract five dollars from each of the players' stakes, then - still within the loop - the players' bets for a winning horse are taken. Line 160 rejects the second player's chosen horse if it is the same as the first player's one. Line 180 seeds the random number generator, a topic we'll be discussing shortly, with a number related to how long it took the two players to choose their horses. 'Seeding' the random number generator, which is done with the RANDOMIZE command, ensures that numbers which are more or less genuinely random are produced. Without line 180, you'd find the same horse would tend to win every race.

Our next FOR/NEXT loop is held in line 200. Notice that this loop also uses a T as control variable. It is perfectly alright to use the same letter for different loops within a program, so long as the loops are not contained within each other. Because there is a slight bias towards horses with lower numbers (as they are checked in numerical order to see if their have crossed the finish line, and if horses one and four have both crossed this line, horse one will be awarded the race because it is checked first) the loop in line 200 gives the horses

with higher numbers a slight edge at the start of the race.

The race proper is held within the loop, with control variable B, from line 210 to 270. This runs over and over again, adding a little (line 220) to each horse's total, until one of the horses (see line 250) has a total which is greater than 17, when action is transferred to line 300 where the winner is proclaimed. Lines 310 and 320 check to see if either player had picked the winning horse, and if so, $10 is added to the lucky punter's stake. Line 330 uses another FOR/NEXT loop for a few sounds, and so do the following sections, gradually printing up (with musical accompaniment) "Stand by...for a new...RACE!!!!". Another loop, line 400, using STEP as we discussed earlier, produces more music and then action goes to line 40 to start the new race.

PRINTER CONTROLS

There are three commands to trigger the printer - LLIST, LPRINT and COPY. LLIST and LPRINT work much the same as the screen commands LIST and PRINT, except they write to the printer, instead of to the screen. (By the way, the use of 'to' in this sentence, as in 'write to the printer' is a peculiarity of computer talk. When someone speaking computer jargon says 'write to the printer' she means 'print out on paper', and by 'write to the screen' she means 'display on the screen'.) Where as LIST will make a listing scroll up on the screen, LLIST will print the whole thing on the printer.

You can write programs to use the printer exclusively, ignoring the screen. Here for example

is a 'mini derby' program, a somewhat simplified version of 'Number Race', which only uses the printer. The listing, of course, was dumped (more jargon) to the printer using LLIST.

```
10 REM MINI DERBY
20 N=1
30 LPRINT
40 LPRINT "PRESS A KEY T
O BEGIN"
50 N=N+1:IF INKEY$="" TH
EN 50
60 RANDOMIZE N
70 DIM A(4)
80 LPRINT
90 LPRINT
100 LPRINT"-------------
----------"
110 FOR B=1 TO 4
120 A(B)=A(B)+RND(1)*6
130 FOR T=1 TO A(B)
140 LPRINT ".";
150 NEXT T
160 LPRINT B;
170 IF A(B)>17 THEN 220
180 IF B<4 THENLPRINT
190 NEXT
200 LPRINT"-------------
----------"
210 GOTO 80
220 LPRINT:LPRINT "AND T
HE WINNER IS";B
```

And here is one run of the program. The screen remains blank (apart from the word RUN) throughout:

```
PRESS A KEY TO BEGIN

----------------------
... 1
.. 2
..... 3
 4
----------------------


----------------------
.... 1
...... 2
.......... 3
. 4
----------------------


----------------------
....... 1
......... 2
........... 3
..... 4
----------------------


----------------------
............. 1
............. 2
............... 3
...... 4
----------------------


----------------------
................. 1
AND THE WINNER IS 1
```

It appears that the lower numbers may enjoy an advantage, so we can add a line like the following to try and remove it:

```
75 FOR B=1 TO 4:A(B)=B:N
EXT
```

Now let's run it again:

```
----------------------
..... 1
...... 2
........ 3
....... 4
----------------------


----------------------
......... 1
......... 2
.......... 3
......... 4
----------------------


----------------------
............ 1
.......... 2
.......... 3
............... 4
----------------------


----------------------
............. 1
............. 2
............... 3
.................. 4
AND THE WINNER IS 4
```

At least the other numbers now seem to have a fighting chance.

The final printer command we'll use is COPY, which simply dumps whatever is on the screen to the printer. Type in COPY now, and see what you get. When I typed in COPY and pressed RETURN, I got this, the last few things I'd entered:

```
LLIST75
RUN
COPY
```

TIME$, DATE$

The Epson HX20 is supplied with a continuous internal clock and calendar, a pair of very useful features. When you first turn on the computer you'll see this:

```
CTRL/@ Initialise
1 MONITOR
2 BASIC
```

If you hold down the control key, marked CTRL and just above SHIFT on the left hand side of the keyboard, and press the @ key at the same time, you'll be asked to enter the date and time in the following format:

- month as two digits
- day as two digits
- year as two digits
- hour as two digits
- minutes as two digits
- seconds as two digits

Press RETURN after all this. If you make a mistake, press BREAK, then press the key to the left of BREAK marked MENU, to get back to the initial display to start again. Once you've entered the time and date,

return to the book. If you now type in PRINT DATE$, then press RETURN, the date will be printed up:

```
01/09/88
```

Type in PRINT TIME$ (or ? TIME$, using ? as an abbreviation for PRINT), and you'll get the time:

```
12:57:09
```

Now this is quite convenient if you just want to check the time and/or date from time to time, but if you want a continuous readout, and you don't mind humbling your computer to become a highly expensive digital clock, you can enter and run the following program:

```
10 REM Digital Clock
20 A$=TIME$
30 CLS
40 PRINT TAB(3);DATE$
50 PRINT
60 PRINT TAB(6);TIME$
70 IF TIME$=A$ THEN 70
80 GOTO 20
```

And if you decide you can do without the date, and simply want a tiny program to type in whenever you want to have your computer displaying the time, the following program will do:

```
10 A$=TIME$
20 CLS
30 PRINT TAB(6);TIME$
40 IF TIME$=A$ THEN 40
50 GOTO 10
```

For a bit of fun with this program, add the following two lines, which will cause the computer to beep every ten seconds:

```
35 A=VAL(RIGHT$(TIME$,2)
)
36 IF 10*(INT(A/10))=A T
HEN SOUND (A/2+5),1
```

Line 35 may look pretty terrifying, but don't worry about it now. We'll be looking at things such as RIGHT$ in due course, and you'll see then that even lines like 35 are fairly easy to understand.

It is possible to produce a more eleborate clock, which beeps every 10 seconds, and plays a little trill every minute. Here is the listing:

```
10 CLS
20 PRINT
30 PRINT TAB(6);TIME$
40 IF 10*INT(M/10)=M THE
N SOUND M,1
50 IF M=0 THEN FOR J=1 T
O 4:SOUND 12*J,1:NEXT
60 A$=TIME$
70 IF TIME$=A$ THEN 70
80 M=VAL(RIGHT$(TIME$,2)
)
90 GOTO 10
```

If you want to get really involved, you can produce a clock which makes continuously changing sounds, which are related to the time. You'll notice as well that the time moves across the screen every 10 seconds. (This is a clock for bored people.)

```
10 CLS
20 PRINT
30 N=10*INT(M/10)
40 PRINT TAB(M-N);TIME$
50 IF N=M THEN SOUND M,1
 ELSE SOUND 2*(M-N),0.5
60 IF M=0 THEN FOR J=1 T
O 4:SOUND 12*J,1:NEXT
70 A$=TIME$
80 IF TIME$=A$ THEN 80
90 M=VAL(RIGHT$(TIME$,2)
)
100 GOTO 10
```

CHAPTER SEVEN - CONTINUING UP THE LEARNING CURVE

Let's look back over the ground we've covered so far, As you can see, there is a great deal we have already done. By now, you should have a fair working knowledge of the following words, and may well be able to manipulate them with a degree of facility:

PRINT, LIST, NEW, RUN, TAB, CLS, REM, CONT, GOTO, FOR and NEXT, ON...GOTO and FOR/NEXT...STEP

As well as these words, you've learned about such things as program lines, numeric and string variables, how to use a semicolon in PRINT statements, and how to control the printer with LLIST, LPRINT and COPY.

We've come a considerable distance already, and in many ways the worst is now over! The 'learning curve' for BASIC is a gentle, upward slope, with the steepest part of the slope representing your first few hours of learning. Once you have those first few hours behind you - which you should have at this point - you'll discover that the additional things you learn interlock neatly with the material you've already mastered. This makes the learning process a satisfying, and not too demanding, one.

RENUM and DELETE

The computer is fitted with some very useful commands to aid you in programming. RENUM stands for renumber and allows you to ensure your program has neatly numbered lines before you save it on tape or disk, or dump it to the printer. It works very quickly.

Consider the following program, based on the digital clock one:

```
5 REM Digital Clock 2
7 REM A$ is assigned
8 REM to equal TIME$
10 A$=TIME$
15 REM CLS clears
16 REM the screen
20 CLS
30 PRINT TAB(6);TIME$
32 REM The next lines
33 REM make the 'beep'
35 A=VAL(RIGHT$(TIME$,2)
)
36 IF 10*(INT(A/10))=A T
HEN SOUND (A/2+5),1
40 IF TIME$=A$ THEN 40
50 GOTO 10
```

If we now type in RENUM, and press RETURN, in a few seconds we'll have this program:

```
10 REM Digital Clock 2
20 REM A$ is assigned
30 REM to equal TIME$
40 A$=TIME$
50 REM CLS clears
60 REM the screen
70 CLS
80 PRINT TAB(6);TIME$
90 REM The next lines
100 REM make the 'beep'
110 A=VAL(RIGHT$(TIME$,2
))
120 IF 10*(INT(A/10))=A
THEN SOUND (A/2+5),1
130 IF TIME$=A$ THEN 130
140 GOTO 40
```

Notice how RENUM not only changes the line numbers into steps of 10, but also changes the GOTO and GOSUB destinations to the correct (new) numbers. The program is renumbered in tens, starting with 10 as the first line number, unless you specify otherwise.

To show how you can specify a different numbering system, look at the following example. If you enter RENUM 7, 10, 13 will renumber the program with the first line number as 7, from line 10, in steps of 13. This produces the following, somewhat odd, program listing:

```
7 REM Digital Clock 2
20 REM A$ is assigned
33 REM to equal TIME$
46 A$=TIME$
59 REM CLS clears
72 REM the screen
85 CLS
98 PRINT TAB(6);TIME$
111 REM The next lines
124 REM make the 'beep'
137 A=VAL(RIGHT$(TIME$,2
))
150 IF 10*(INT(A/10))=A
THEN SOUND (A/2+5),1
163 IF TIME$=A$ THEN 163
176 GOTO 46
```

You specify the renumbering you desire as follows:

RENUM X, Y, Z where X is the number you want the first line to start with, Y is the number you want the renumbering to start from (it was 10 in the above example, so renumbering began from the first line) and Z is the step size.

If you decided just to renumber the last four lines, from 137, in steps of 99, and you wanted the line which is now 137 to become 1000, you'd enter RENUM 1000, 137, 99 to produce the following:

```
7 REM Digital Clock 2
20 REM A$ is assigned
33 REM to equal TIME$
46 A$=TIME$
59 REM CLS clears
72 REM the screen
85 CLS
98 PRINT TAB(6);TIME$
111 REM The next lines
124 REM make the 'beep'
1000 A=VAL(RIGHT$(TIME$,
2))
1099 IF 10*(INT(A/10))=A
 THEN SOUND (A/2+5),1
1198 IF TIME$=A$ THEN 11
98
1297 GOTO 46
```

These contrived examples indicate how RENUM is used, although you're most unlikely to want to renumber in this way. RENUM is useful not only for cleaning up listings before you save or dump them, but also for 'making space' in listings if you decide you wish to add extra lines at some point in a program and are running out of room for this. In the last listing, we have made a large space between lines 124 and 1000 where additional lines could be inserted.

DELETE, as you can probably imagine, gets rid of certain program lines. If you have the above program in your computer, and you entered DELETE 59 - 124 and pressed RETURN, you'd end up with this program:

```
7 REM Digital Clock 2
20 REM A$ is assigned
33 REM to equal TIME$
46 A$=TIME$
1000 A=VAL(RIGHT$(TIME$,
2))
1099 IF 10*(INT(A/10))=A
 THEN SOUND (A/2+5),1
1198 IF TIME$=A$ THEN 11
98
1297 GOTO 46
```

As you can see, lines 59, 72, 85, 98, 111 and 124 have vanished. The format for DELETE is the word DELETE followed by the first line number you wish to remove, followed by a hyphen, then the final line number you wish to remove.

MULTI-STATEMENT LINES

We've used these in several programs already. A multi-statement line is like the following:

```
10 FOR T=1 TO 20:PRINT T
:SOUND T,1:NEXT
```

As you'll see if you run this, it is a complete program in one line. This takes the place of the following four lines:

```
10 FOR T=1 TO 20
20 PRINT T
30 SOUND T,1
40 NEXT
```

Running this four-line program will demonstrate that it does exactly the same as the multi-statement line.

In general, I'd advise you to stay away from multi-statement lines, as they make errors harder to spot, and can make tracing a program through very difficult. There are also traps. For example, in the next two lines, the second statement (following the colon, which is always used to separate statements with a single line number) will never be executed:

```
10 GOTO 50:A=25

10 REM End of program:PR
INT "That's all, folks!"
```

The use of single or multi-statement lines is, in large measure, a question of taste. Because it is easier to decipher programs which have single statement lines, than it is to decipher those with multi-statement lines, I am generally in favor of sticking to single statement lines. However, you may well decide you prefer the compactness of programs which make good use of multi-statement lines.

There are times when multi-statement lines are desirable, or even essential. Two areas when they are desirable are when spacing out PRINT statements as follows (line 20):

```
10 PRINT "EAGLES"
20 PRINT:PRINT
30 PRINT "FLYING"
40 PRINT:PRINT
50 GOTO 10
```

or for 'dummy' loops for delay as follows (also line 20):

```
10 SOUND RND(1)*40,1
20 FOR T=1 TO 250:NEXT
30 GOTO 10
```

A FEW FUNCTIONS

By now, I'm sure you've realised that the programs we've looked at so far include many BASIC words which have not been explained. This is because, after a certain point, it becomes increasingly difficult to avoid assuming certain words are known, if worthwhile programs are to be introduced. Some of the words - such as INPUT - have pretty obvious meanings. However, even those which I assume you'll understand without long explanations will be covered in the coming pages. If, by now, you're able to work out what new words are likely to mean, you may well want to skim parts of the following, just to check that your guess was right.

SQR, INT and ABS

These three functions act on numbers to produce the results suggested by their names. SQR returns the square root of a number, INT the integer portion and ABS the absolute value of the number. You probably don't need any introduction to SQR. Type in the following and press RETURN:

```
PRINT SQR(4)
```

Of course, you'll get the result of 2, the square root of 4. The next program shows INT and ABS in use. Enter the program and run it, trying different numbers each time.

```
10 REM A FEW FUNCTIONS
20 GOSUB 130
30 PRINT "ENTER A NUMBER
"
40 INPUT A
50 GOSUB 130
60 PRINT "YOUR NUMBER"
70 PRINT "IS";A
80 GOSUB 130
90 PRINT "INT(A)=";INT(A
)
100 GOSUB 130
110 PRINT "ABS(A)=";ABS(
A)
120 END
130 FOR T=1 TO 500:NEXT
140 CLS
150 RETURN
```

This will not tell you too much if you just enter whole, positive numbers, as the program output will look like this:

```
ENTER A NUMBER

YOUR NUMBER
IS 45

INT(A)= 45

ABS(A)= 45
```

But enter a number like -34.567 and you'll see INT and ABS in action clearly:

```
ENTER A NUMBER

YOUR NUMBER
IS-34.567

INT(A)=-35

ABS(A)= 34.567
```

From this, you can see that INT returns the next lowest whole number (rather than rounding up, a difference as you'll see), and ABS returns the positive, whole number portion of the number upon which it is acting.

ROUNDING UP If you'd like to round numbers to the nearest whole number (so 2.54 becomes 3, rather than 2, as using INT would produce) use the following routine:

```
10 REM ROUNDING TO
20 REM NEAREST WHOLE
30 REM NUMBER
40 GOSUB 140
50 PRINT "ENTER A NUMBER
"
60 INPUT A
70 GOSUB 140
80 PRINT "YOUR NUMBER"
90 PRINT "IS";A
100 GOSUB 140
110 PRINT "ROUNDED,"
120 PRINT A;"IS";INT(A+0
.5)
130 END
140 FOR T=1 TO 500:NEXT
150 CLS
160 RETURN
```

As these examples show, positive numbers are rounded up to the nearest whole number, while negative numbers are rounded down:

```
YOUR NUMBER
IS 45.67

ROUNDED,
 45.67 IS 46
```

```
YOUR NUMBER
IS-45.67

ROUNDED,
-45.67 IS-46
```

To round a number, simply add 0.5 to it, then apply INT.

CHAPTER EIGHT - RANDOM NUMBERS AND DECISION MAKING

We've used random numbers several times already in this book, such as in the section on numeric variables for generating the initial 'population' in STAR COLONY and in the NUMBER RACE. Random numbers are very useful in games programs and simulation programs as they allow the computer to appear as if it is chosing from a number of alternatives.

By now you probably know the format for the production of random numbers. PRINT RND(1) will produce a number between zero and one.

This two-line program shows you the kind of numbers you get:

```
10 PRINT RND(1)
20 GOTO 10
```

Running it for a few seconds produces numbers like these:

```
 .591065
 .207991
 .550967
 .635863
 .10411
 .806334
4.29073E-02
 .953862
 .354289
```

These numbers, between zero and one, are not of much general use. Whole numbers (such as 5, 887 and 2345) prove of much greater use in many programs. However, the fractions can be used. Imagine we wanted to emulate the tossing of a coin. We could say that each time the random number produced was less than

0.5 it equalled a 'tail' and each time the random number equalled or was greater than 0.5, it represented a 'head'. Here's a little program to show this:

```
10 A=RND(1)
20 IF A<0.5 THEN PRINT "
HEAD"
30 IF A>=0.5 THEN PRINT
"TAIL"
40 GOTO 10
```

Run it for a few seconds and you'll get a result like this:

```
TAIL
HEAD
TAIL
TAIL
HEAD
TAIL
HEAD
```

We can now look at the mechanism by which the computer makes decisions. Examine lines 20 and 30 of the preceding program. They say IF the random number (the variable A) is less than 0.5 THEN PRINT "HEAD" (line 30), and IF the random number is greater than, or equal to, 0.5, THEN PRINT "TAIL" (line 30).

These two lines are examples of IF/THEN statements. They are easy to understand. IF a condition is true, THEN do something. IF/THEN can be followed by other things than a PRINT statement. For example, if we wanted the computer to make a high tone if it rolled a head, and a low tone for a tail, we could use the following program:

```
10 A=RND(1)
20 IF A<0.5 THEN SOUND 5
0,5
30 IF A<=0.5 THEN SOUND
1,5
40 GOTO 10
```

IF/THEN can also be followed by instructions to GOTO a certain line. The format for this is:

```
IF A<5 THEN GOTO 100
```

As you know, the word GOTO is optional in this case, as the computer will understand what is required. The format can be:

```
IF A<5 THEN 100
```

This has exactly the same effect as the earlier line.

There is another word we can use in connection with IF/THEN. This other word is ELSE. ELSE is used when you want the computer to do something IF a condition is fulfilled, and to do something ELSE if the condition is not fulfilled. We can demonstrate this with a variation on the HEAD/TAIL program:

```
10 A=RND(1)
20 IF A<0.5 THEN PRINT "
HEAD" ELSE PRINT "TAIL"
30 GOTO 10
```

Running this produces exactly the same result as running the initial version. There is no reason why the two conditions should be of the same type, as the following variation of that program shows:

```
10 A=RND(1)
20 IF A<0.5 THEN PRINT "
HEAD" ELSE SOUND20,5
30 GOTO 10
```

Let's get back to random numbers. If you'd been watching the output of the HEAD/TAIL program closely, you'd have noticed that there are almost

exactly the same number of heads as there are tails, which is as it should be. If the random number generator was producing genuinely random numbers, then there should be about as many numbers greater than 0.5 as there are numbers below 0.5. But how can we check this, and how can we be sure that the same sequence of 'random' numbers is not produced each time we run the program?

We'll attempt to answer those questions in a moment, but we must first discuss another aspect of random number use.

If you want whole numbers, rather than fractions, you have to manipulate the random number produced. This next program produces random numbers between zero and one:

```
10 PRINT RND(1)
20 GOTO 10
```

If you want numbers between zero and nine, you need to do the following:

```
10 PRINT INT(RND(1)*10)
20 GOTO 10
```

Note that the number you multiply by RND(1) is one more than the maximum randomly generated integer you want. To get numbers between one and ten, you just add one to the produced number, as follows:

```
10 PRINT INT(RND(1)*10)+
1
20 GOTO 10
```

Now we are in a position to check the randomness or otherwise of the numbers produced by the computer's random number generator. I ran the following program

twice and came up with the results with follow the program:

```
10 FOR B=1 TO 10
20 PRINT INT(RND(1)*10)+
1;
30 NEXT
```

```
 6  3  6  7  2  9  1  10
  4  6
```

```
 6  3  6  7  2  9  1  10
  4  6
```

As you can see, the 'random' numbers are not particularly random at all. Subsequent runs produced the same result. At first I thought that a longer run might produce numbers which more closely approached randomness, so I wrote the following program to produce 100 numbers at random between 0 and 9 and to count how often each number was produced. Because the total number of integers produced was 100, a count of each could be expressed as a percentage. Here is the program I used:

```
10 REM Random number
20 REM  distribution
30 DIM A(9)
40 FOR B=1 TO 100
50 C=INT(RND(1)*10)
60 A(C)=A(C)+1
70 NEXT
80 FOR B=0 TO 9
90 LPRINT B;" ";A(B);"%"
100 NEXT
```

And here are the results of running it twice:

```
0   9 %          0   9 %
1   13 %         1   13 %
2   8 %          2   8 %
3   15 %         3   15 %
4   6 %          4   6 %
5   12 %         5   12 %
6   14 %         6   14 %
7   10 %         7   10 %
8   8 %          8   8 %
9   5 %          9   5 %
```

As you can see, the longer runs do not indicate that the numbers become more random. There is a solution. Your computer generates random numbers by reading through a long, long list of numbers. The list is so long that it is virtually impossible to detect a sequence in it. If, however, the sequence is set back to its beginning each time RUN is entered (as has happened in the sample programs we've run), any pretence of randomness is lost.

The RANDOMIZE command is used to start the selection of numbers from different points in the list. The number which follows RANDOMIZE (known as the 'seed') tells the computer where in the list to begin. You can use RANDOMIZE like this, without a following number:

```
10 RANDOMIZE
20 PRINT RND(1)
30 GOTO 20
```

However, when you run it, the computer will print up

```
Seed?
```

and refuse to continue until you enter a number.

There are a couple of convenient ways to seed the random number generator, to ensure that the numbers produced are close to numbers chosen properly at random, which do not involve entering seed numbers and then pressing RETURN. One way is to use the delay taken to respond to an instruction as the seed. This is used in the STAR COLONY program. Here is a little routine to show it in action:

```
10 N=0
20 PRINT "PRESS ANY KEY"
30 PRINT "TO CONTINUE"
40 N=N+1
50 IF INKEY$="" THEN 40
60 RANDOMIZE N
70 PRINT N
```

As you'll see when you run this program, the longer you take to press a key, the higher the value assigned to N, the seed of the random number generator. Even though this is fairly simple, it is still not the most desirable way of seeding the generator, as some response - which is not really related to the program - is required from the user.

Fortunately, there is another way to solve the problem. The computer (as we discovered earlier) has a built-in clock. The 'seconds' value changes 60 times a minute, so if this value can be accessed it could be used to automatically seed the random number generator. I will not explain, at this point, how the line I'm about to give you, works (because we'll be discussing string handling shortly), but just give it to you to use.

If you enter the following program, you'll see the time printed up, then the second's value:

```
10 PRINT TIME$
20 PRINT RIGHT$(TIME$,2)
30 FOR T=1 TO 60:NEXT
40 GOTO 10
```

The RIGHT$(TIME$,2) extracts the second's value.

You'll recall we got the computer to print out 100 numbers generated at random between 0 and 9. If we add a variation of the RIGHT$(TIME$,2) to the program, as the means of seeding the random number generator, we should be able to produce a sequence of numbers different from the one we got twice in a row before.

Add the following line to the program which was given a few pages ago, and run it again:

```
25 RANDOMIZE VAL(RIGHT$(
TIME$,2))
```

Here are the results of a couple of runs produced by the program, with line 25 in place

0	12 %	0	10 %
1	12 %	1	14 %
2	7 %	2	7 %
3	9 %	3	10 %
4	8 %	4	13 %
5	11 %	5	9 %
6	11 %	6	11 %
7	10 %	7	8 %
8	5 %	8	8 %
9	15 %	9	10 %

Instead of showing the result as a list of percentages, we can get the computer to print up a bar chart to illustrate the number of times each number, between 0 and 9, is produced in a run of 100

numbers. This program will do it (a couple of sample runs follow the program listing):

```
10 REM Random number
20 REM  bar chart
30 RANDOMIZE VAL(RIGHT$(
TIME$,2))
40 DIM A(9)
50 FOR B=1 TO 100
60 C=INT(RND(1)*10)
70 A(C)=A(C)+1
80 NEXT
90 FOR B=0 TO 9
100 LPRINT B;
110 FOR C=1 TO A(B)
120 LPRINT CHR$(128);
130 NEXT
140 LPRINT
150 NEXT
```

```
0 ++++++++++
1 ++++++++++
2 +++++++
3 +++++++++
4 +++++++
5 ++++++++++
6 ++++++++++
7 +++++++++
8 ++++
9 ++++++++++++++
```

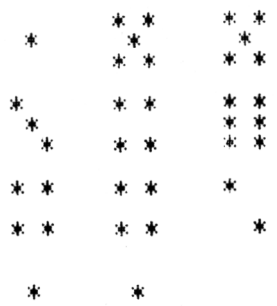
```
0 ++++++++++++
1 ++++++++++
2 ++++++++
3 ++++++
4 ++++++++++
5 ++++++++++
6 ++++++++
7 ++++++++++
8 ++++++
9 ++++++++++
```

DICE ROLLS

Random numbers can be used, as we have seen, to emulate randomly occurring events in 'real life', such as the result of tossing a coin a number of times. Dice throws produce, if the dice are unbiased, random numbers between one and six, and it is very easy to get the computer to emulate the throw of a single die, as this program illustrates:

```
10 RANDOMIZE VAL(RIGHT$(
TIME$,2))
20 A=INT(RND(1)*6)+1
30 CLS
40 SOUND 6*A,2
50 ON A GOSUB 80,100,120
,140,160,180
60 FOR T=1 TO 400:NEXT
70 GOTO 20
80 PRINT:PRINT " * "
90 RETURN
100 PRINT "*":PRINT:PRIN
T"  *"
110 RETURN
120 PRINT "*":PRINT" *":
PRINT"  *"
130 RETURN
140 PRINT"* *":PRINT:PRI
NT"* *"
150 RETURN
160 PRINT"* *":PRINT" *"
:PRINT"* *"
170 RETURN
180 PRINT"* *":PRINT"* *
":PRINT"* *"
190 RETURN
```

Here's what it looks like in action:



Many dice games require you to throw two dice at once, then add the pips. At first thought, you might feel the way to do this with the computer would be to get the computer to generate random numbers between 1 and 12. However, it does not work that way. Although the chance of a single die falling with any face showing is one in six (around 17%), the distribution of totals produced by two dice is as follows: total 2 (2.77%), total 3 (5.55%), total 4 (8.33%), total 5 (11.11%), total 6 (13.88%), total 7 (16.66%), total 8 (13.88%), total 9 (11.11%), total 10 (8.33%), total 11 (5.55%) and total 12 (2.77%).

The different numbers are due to the number of different ways a particular total can be reached. Seven, for example, can be reached in six different ways (1 + 6, 2 + 5, 4 + 3, 3 + 4, 5 + 2 and 6 + 1), while a total of two or of twelve can only be reached in one way (1 + 1 or 6 + 6).

If a random number generator is working properly,

and it is used to demonstrate how two dice might fall, we would expect the distribution of the totals to approach the theoretical distribution given in the preceding paragraph. We'll test it to see. Here is a program to throw two dice, followed by the results of two trials. Note that in each case the dice are thrown 100 times:

```
10 REM Throwing two
20 REM       dice
30 RANDOMIZE VAL(RIGHT$(
TIME$,2))
40 DIM A(12)
50 FOR B=1 TO 100
60 C=INT(RND(1)*6)+1
70 C=C+INT(RND(1)*6)+1
80 A(C)=A(C)+1
90 NEXT
100 FOR B=1 TO 12
110 LPRINT B;"-";A(B);"%
"
120 NEXT
```

```
1 - 0 %
2 - 2 %
3 - 4 %
4 - 7 %
5 - 9 %
6 - 17 %
7 - 13 %
8 - 18 %
9 - 14 %
10 - 7 %
11 - 8 %
12 - 1 %
```

```
1 - 0 %
2 - 4 %
3 - 6 %
4 - 7 %
5 - 10 %
6 - 15 %
7 - 16 %
8 - 15 %
9 - 9 %
10 - 11 %
11 - 5 %
12 - 2 %
```

As you can see, the distribution of totals is fairly

close to the predicted distribution. Certainly the totals show a bulge around the seven.

The program was modified to throw the dice 1000 times, with the idea that the total should more closely approach that which was predicted. Here is the result of throwing a pair of dice 1000 times:

1 - 0 %
2 - 3.2 %
3 - 6.4 %
4 - 7.8 %
5 - 11.4 %
6 - 13.5 %
7 - 16.1 %
8 - 13.8 %
9 - 12.2 %
10 - 7.7 %
11 - 5.5 %
12 - 2.4 %

As you can see, this is closer (as expected) to the theoretical distribution. Finally, the program was modified again, this time for 10,000 throws. Rounded up, the result is very close to the predicted result which shows that the random number generator in the computer is working well and, for all normal uses, can be assumed to be producing genuinely random numbers.

1 - 0 %
2 - 2.93 %
3 - 5.84 %
4 - 8.62 %
5 - 11.38 %
6 - 13.07 %
7 - 16.58 %
8 - 14 %
9 - 11.2 %
10 - 8.47 %
11 - 5.06 %
12 - 2.85 %

CHAPTER NINE - STRING HANDLING AND REVERSI

You may remember from our discussion on variables that strings are designated by variable names ending with a dollar sign. Strings are very useful parts of your working vocabulary in BASIC, as they can be manipulated to produce a number of valuable results.

The string functions we'll be looking at in this section are STR$, VAL, LEN, LEFT$, RIGHT$ and MID$. Although an expression containing some of these may look bewildering (as you'll recall if you think back to the RIGHT$ extraction of TIME$ used to seed the random number generator in the last chapter) they are easy to use if you tackle them carefully.

STR$

STR$ (which is often said aloud as 'string dollar' or 'string string', although I prefer the former) changes a number into an equivalent string. That is, it changes 45.6 into "45.6", a transformation which does not appear immediately to be one which has very much value. However, it is useful in come circumstances because strings are in some ways easier to manipulate than are numbers. I'll show you one use in a moment, but first we must look at two other string functions.

LEN

This function finds the length of a string, so if you told the computer to print LEN(A$) where A$ was "FULTON", it would return 6. If A$ was "45.6", then LEN(A$) would be 4, as the decimal point counts as one of the characters in the string.

VAL

This acts as the opposite of STR$. That is, STR$(42.6) is "42.6" while VAL("42.6") is 42.6.

We can use two of these to produce a program which lines numbers up to the right (instead of to the left, as is usually the case):

```
10 FOR J=1 TO 5
20 READ A
30 A$=STR$(A)
40 PRINT TAB(10-LEN(A$))
;"$";A$
50 NEXT
60 DATA 23.45,350.07,1.3
5,1444.75,23.74
```

This is what the output looks like:

```
   $ 23.45
 $ 350.07
    $ 1.35
$ 1444.75
   $ 23.74
```

Line 20 reads the number from the DATA statement, and line 30 changes this number (A) into a string (A$). Line 40 prints this, using a TAB setting of 10 minus the length (LEN) of the string.

LEFT$, RIGHT$ and MID$

These are used to extract required portions of strings. LEFT$ returns a designated length from the left of a string; RIGHT$ does the same from the rightmost character; and MID$ selects a string of a designated length from the middle of another string.

Here is a program to show LEFT$, RIGHT$ and MID$ in use:

```
10 A$="DAKOTA"
20 PRINT"LEFT$(A$,3)=";L
EFT$(A$,3)
30 PRINT"RIGHT$(A$,3)=";
RIGHT$(A$,3)
40 PRINT"MID$(A$,2,3)=";
MID$(A$,2,3)
```

And here is the result of running it:

```
LEFT$(A$,3)=DAK

RIGHT$(A$,3)=OTA

MID$(A$,2,3)=AKO
```

As you can see, when A$ is "DAKOTA", LEFT$(A$,3) prints out the three leftmost characters of the string, "DAK". RIGHT$(A$,3) returns the three rightmost characters of the string, "OTA". MID$(A$,2,3) gives three characters from the string, starting at the second character, "AKO".

ASC and CHR$

Before we move onto our next program, REVERSI (which is often known as Othello), in which MID$ is used to extract portions of the playing board (stored as a series of strings in an array), we need to look at a few other string-handling words, starting with ASC and CHR$.

The letters, numbers and symbols manipulated by your computer all have 'character codes'. These are known

as ASCII codes. ASCII stands for American Standard Code for Information Exchange, and is the most widely used encoding system for English language alphanumerics (that is, letters, numbers and symbols).

There are 128 upper and lower case letters, digits and some special characters. Although the exact character set varies from country to country (character number 35, for example, is the hash symbol in the US, while it is the pound sign in Britain), most of the symbols are standard.

ASC is used to get the ASCII code number from a symbol, and CHR$ is used to turn the code number into a symbol. The following program should make it clear:

```
10 PRINT "A";ASC("A")
20 PRINT "B";ASC("B")
30 INPUT "ENTER A LETTER
";A$
40 PRINT A$;ASC(A$)
50 C=ASC(A$)
60 PRINT C;"IS THE CODE"
70 PRINT "OF ";A$
80 PRINT "CHR$(C) IS ";C
HR$(C)
```

When you run it, you'll see something like this:

```
A 65
B 66

ENTER A LETTER?
T 84

 84 IS THE CODE
OF T

CHR$(C) IS T
```

We'll have a look at a program now which prints out the entire character set. The characters whose codes are below 33 are control characters which do not result in a character being printed out. Therefore, these have been left out of the program, which gets the computer to print out all the numbers from 33 to 159 with their corresponding symbols.

Here is the program:

```
10 REM HX20  characters
20 FOR A=33 TO 159 STEP
2
30 LPRINT A;" - ";CHR$(A
);"    ";A+1;" - ";CHR$(
A+1)
40 NEXT
```

And this is the result of running it:

```
33  - !     34  - "
35  - #     36  - $
37  - %     38  - &
39  - '     40  - (
41  - )     42  - *
43  - +     44  - ,
45  - -     46  - .
47  - /     48  - 0
49  - 1     50  - 2
51  - 3     52  - 4
53  - 5     54  - 6
55  - 7     56  - 8
57  - 9     58  - :
59  - ;     60  - <
61  - =     62  - >
63  - ?     64  - @
65  - A     66  - B
67  - C     68  - D
69  - E     70  - F
71  - G     72  - H
73  - I     74  - J
75  - K     76  - L
```

77 - M
78 - N
79 - O
80 - P
81 - Q
82 - R
83 - S
84 - T
85 - U
86 - V
87 - W
88 - X
89 - Y
90 - Z
91 - [
92 - \
93 -]
94 - ^
95 - _
96 - `
97 - a
98 - b
99 - c
100 - d
101 - e
102 - f
103 - g
104 - h
105 - i
106 - j
107 - k
108 - l
109 - m
110 - n
111 - o
112 - p
113 - q
114 - r
115 - s
116 - t
117 - u
118 - v
119 - w
120 - x
121 - y
122 - z
123 - {
124 - |
125 - }
126 - ~
127 -
128 - ┼
129 - ┴
130 - ┬
131 - ┤
132 - ├
133 - ─
134 - │
135 - ┌
136 - ┐
137 - └
138 - ┘
139 - ▒
140 - ▉
141 - ▄
142 - ▌
143 - ●
144 - o
145 - ♠
146 - ♥
147 - ♦
148 - ♣
149 - ♪
150 - ♫
151 - ♣
152 - ♯
153 - ♥
154 - ♿
155 - ↑
156 - ↓
157 - ×
158 - ÷
159 - ±

Another simple program, which you may like to run, will fill the screen (or your printing paper) with randomly chosen numbers, letters and symbols:

```
10 RANDOMIZE VAL(RIGHT$(
TIME$,2))
20 LPRINTCHR$(INT(RND(1)
*126)+33);
30 GOTO 20
```

The result will be something like this:

```
DK!B/ly."g:U}lHdn?U's ┼4
Ea=DgGM3█|/●↓↓┌█TUu&^F├大
大@'1?z@&;▄¦└1×?┌2+c+b█▒┐
}!$¥┬B'┴┬♥M大4×1♪S♦×o3>D¦
大Is]├}b/,N)/_[R8k♣¥g;fm/
2c9;×♠↓\J┐●Sft1^A2♪%',█?
H6p\.>1d;█\Ve4$1( mJ-|H÷
>peb-+"&(↓s┴Or>PK┼FM#Y/p
```

STRING$, SPACE$ and INSTR

Here are three more string functions for you. The first one, STRING$, is demonstrated by our next program:

```
10 REM STRING$ DEMO
20 RANDOMIZE  VAL(RIGHT$
(TIME$,2))
30 A$=STRING$(INT(RND(1)
*9+1),INT(RND(1)*126)+33
)
40 PRINT A$
50 GOTO 30
```

The program produces this sort of result:

```
VVV
55
%%
┤┤┤┤┤┤┤┤
YYYYYY
VVVVVVVV
```

The format of a STRING$ statement is STRING$(n,m) where n is the number of characters you want printed, and m is the ASC of the character you want. In the program, line 30 sets A$ equal to a random number of a randomly chosen character. It is very useful when, for example, you wish to set a string variable to something like a line of characters you wish to use to break up text output.

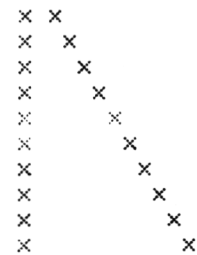
For example, if you wanted to 'rule off' every so often with a solid black line, you could enter the following lines, which produce the result shown below the listing:

```
10 A$=STRING$(24,140)
20 LPRINT A$
```

SPACE$ produces, as you might expect, a string made up of the number of spaces indicated by a number following the word SPACE$. That is, PRINT SPACE$(n) will produce n spaces. This may be of limited use, and difficult in some circumstances to detect, but a program like the following will show it in action:

```
10 FOR A=1 TO 10
20 A$="x"+SPACE$(A)+"x"
30 LPRINT A$
40 NEXT
```

It produces this result:



Strings on the HX20 can be **concatenated**, or added together, as occurs in line 20 of the program. Concatenation is shown more clearly in this program, which produces the result shown beneath line 40:

```
10 A$="#"
20 PRINT "A$ IS ";A$
30 B$=A$+A$+"HI"+A$
40 PRINT "B$ IS ";B$

A$ IS #
B$ IS ##HI#
```

INSTR looks for the first appearance of one string within another one, and gives a number which is the position where the designated string has been found, as this program shows:

```
10 A$="ABCDEFGH"
20 INPUT B$
30 PRINT "B$ IS ";B$
40 PRINT"INSTR(A$,B$) IS
 ";INSTR(A$,B$)
50 GOTO 20
```

If you enter any letter from A to H when prompted to do so (line 20), the computer will return a number which is the position of the letter within A$. If you enter a letter combination which is not within A$, you'll also get a return of zero. Here are some samples of output:

```
B$ IS A
INSTR(A$,B$) IS  1


B$ IS D
INSTR(A$,B$) IS  4


B$ IS K
INSTR(A$,B$) IS  0


B$ IS ABC
INSTR(A$,B$) IS  1


B$ IS ADS
INSTR(A$,B$) IS  0


B$ IS BCE
INSTR(A$,B$) IS  0


B$ IS BC
INSTR(A$,B$) IS  2
```

REVERSI

Now we get to the REVERSI program, in which MID$ features prominently. Not only is it used to extract portions of the playing board, but also counts the number of pieces each player has at the end of the game.

REVERSI (often called Othello) was invented in the 1880's, as a variation on Checkers. Players start with four pieces on a board which looks like this:

```
*12345678*
8........8
7........7
6........6
5...HC...5
4...CH...4
3........3
2........2
1........1
*12345678*
```

The H's are human pieces, while those of the computer are the C's. You can place a piece on any blank position (indicated by a dot) which is adjacent to an 'enemy' piece, and which has one of your own pieces beyond that. Once you place your piece on the board, every enemy piece in line between the one you've just placed and your other piece or pieces is converted into one of your own pieces. The 'conversions' occur in any direction, vertically, horizontally or diagonally.

I'll try to make that explanation a little clearer. Starting from the position above, I (as the human) placed a piece in the square with co-ordinates 5,6

(the two co-ordinates are entered separately, with RETURN between them, and the first co-ordinate is the number down the side; the second is the number across the top). Immediately, the computer piece (on 5,5) was converted into one of my pieces, to leave the board looking like this:

```
*12345678*
8........8
7........7
6........6
5...HHH..5
4...CH...4
3........3
2........2
1........1
*12345678*
```

Then it was the computer's turn to play. After a moment's deliberation, the computer moved into the square 6,6 and converted my piece on 5,5 leaving the board looking like this:

```
*12345678*
8........8
7........7
6.....C..6
5...HCH..5
4...CH...4
3........3
2........2
1........1
*12345678*
```

My response was to move into 3,4:

```
*12345678*
8........8
7........7
6.....C..6
5...HCH..5
4...HH...4
3...H....3
2........2
1........1
*12345678*
```

Without hesitation, the computer placed its piece on 3,3:

```
*12345678*
8........8
7........7
6.....C..6
5...HCH..5
4...CH...4
3..CH....3
2........2
1........1
*12345678*
```

The computer now had four pieces to my three. As you can see, the total number of pieces held by each player can vary widely from move to move, unlike games like chess, where - in general - the strength of each player changes only gradually.

It is interesting to note that Reversi was the first game in which a Grand Master was beaten by a computer. The victorious program, called The Moore, was written by a student at London University who

was not even sure how to play the game, nor what strategies would prove effective, when he began writing it.

As you can see, the game is fairly easy to learn. In fact, one commercial board version of the game bears the slogan 'A minute to learn, a lifetime to master', which seems to just about sum it up. All you need to know now is how the game ends. Each player continues in the way described, until all 64 squares are filled, or until neither player can move. At this point, the player with the most pieces on the board is declared the winner.

If you cannot move, you enter a zero instead of the 'position across' and the computer responds with a polite 'OK, please stand by', as it considers its next move.

Let's get back to the game we were considering. Several moves after the point at which we last saw it, the board looked like this. The computer had just moved into 1,3:

```
*12345678*
8........8
7........7
6..HHHHH.6
5..CCCH..5
4...CH...4
3..CHC...3
2..C.....2
1..C.....1
*12345678*
```

Anxious to prove the general superiority of human beings over silicon chips, I responded with a devastating move, placing my piece smugly in 5,2:

```
*12345678*
8........8
7........7
6..HHHHH.6
5.HHHHH..5
4...CH...4
3..CHC...3
2..C.....2
1..C.....1
*12345678*
```

I was now well ahead. A count of pieces at that point showed I had 12 pieces to the computer's mere five. But Reversi is a game of changing fortunes, and it is not a game in which either player can assume that material advantage halfway through a game suggests anything about the likelihood of eventual victory. Position can be far more important than winning a large number of pieces, and the program in this book plays purely on a knowledge of the most valuable positions.

The computer's response was to move into 4,6:

```
*12345678*
8........8
7........7
6..HHHHH.6
5.HHHHH..5
4...CCC..4
3..CHC...3
2..C.....2
1..C.....1
*12345678*
```

I placed my piece on 3,6 capturing not only the piece which had been on 3,5 but also that on 4,6:

```
*12345678*
8........8
7........7
6..HHHHH.6
5.HHHHH..5
4...CHH..4
3..CHHH..3
2..C.....2
1..C.....1
*12345678*
```

This small move, in which pieces in different directions are captured by a single move, shows why fortunes can change so swiftly in this game. The computer replied with a move on 2,6:

```
*12345678*
8........8
7........7
6..HHHHH.6
5.HHHHH..5
4...CHH..4
3..CHCH..3
2..C..C..2
1..C.....1
*12345678*
```

And I took the piece back by moving onto the bottom line, at 1,7:

```
*12345678*
8........8
7........7
6..HHHHH.6
5.HHHHH..5
4...HHH..4
3..CHHH..3
2..C..H..2
1..C...H.1
*12345678*
```

Later on in the game, although I continued to enjoy a slight numbers advantage, the computer was definitely putting the heat on me, as this series of exchanges show:

```
*12345678*
8.......C8
7......CC7
6..HHHCHC6
5.HHHCH..5
4..HHHCCC4
3..HHHHCH3
2..H..CHH2
1CCC..CHH1
*12345678*
```

```
*12345678*
8.......C8
7......CC7
6..HHHCHC6
5.HHHCH.H5
4..HHHCHH4
3..HHHHCH3
2..H..CHH2
1CCC..CHH1
*12345678*
```

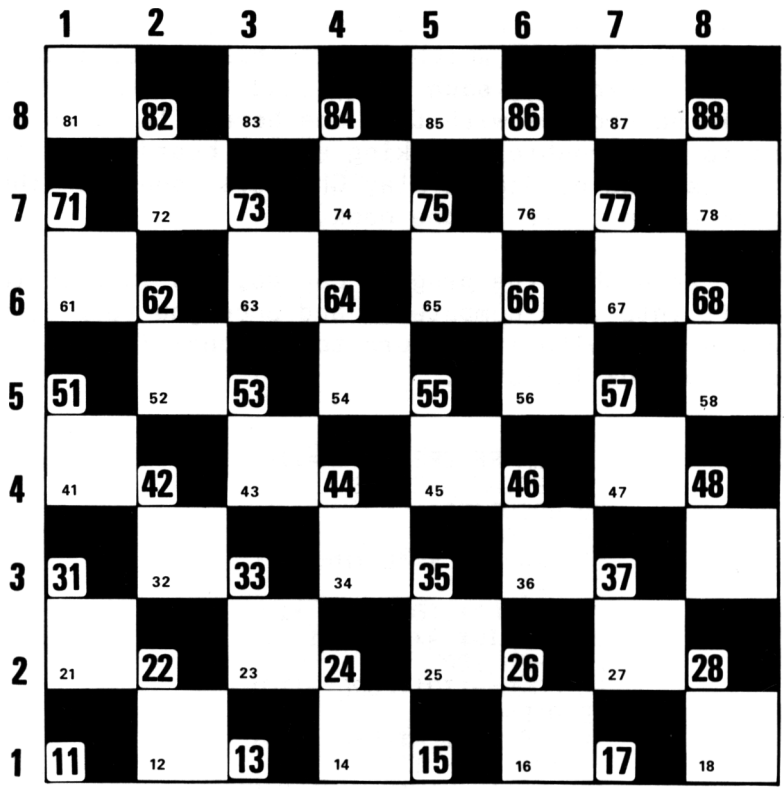
```
*12345678*
8.......C8
7......CC7
6..HHHCHC6
5CCCCCH.H5
4..HHHCHH4
3..HHHHCH3
2..H..CHH2
1CCC..CHH1
*12345678*
```

```
*12345678*
8.......C8
7.....HCC7
6..HHHHHC6
5CCCCCH.H5
4..HHHCHH4
3..HHHHCH3
2..H..CHH2
1CCC..CHH1
*12345678*
```

However, despite putting up my best efforts, the computer finally won, with 38 pieces to my puny 26.

One variation of this program was written which evaluated the strength of various positions, and then compared the number of pieces gained by each move of approximately equal worth. The computer then made the move which captured it the most pieces.

It was found, surprising as it may sound, that the program actually played worse when selecting the move which gave it the most pieces that it did when selecting a move which was equally strong in terms



of position, but game the computer the fewest number of pieces. And this method of play was slightly better than one in which the computer chose at random from a number of moves of approximately equal strength.

Although you can play the game using the board printed up on the screen, it takes a bit of practice to do so. Therefore, you may well prefer to mark up a checkers board with the appropriate numbers. You'll need a number of black and white pieces, about 40 of each. You may decide to buy a Reversi or Othello board set, which usually has pieces with a different color on each side, and to number the board in accordance with this one here. It is worth going to the trouble of making such a board, as it will also be needed to play Checkers against the program given later in the book.

Having seen what the program can do, it is time to enter it into your computer, and once you've played a game or two with it, return to the book for a discussion on how it works.

```
10 REM REVERSI (Othello)
20 DEFINT A-Z:CLS
30 GOSUB400
40 GOTO 200
50 B$="H":C$="C":H=0
60 K=1
70 A=T(K)+1:B=T(K+1)+1
80 IF MID$(A$(A),B,1)<>"
." THEN 180
90 FORC=-1TO1:FORD=-1TO1
:E=0:F=A:G=B
100 IF MID$(A$(F+C),G+D,
1)<>B$ THEN 120
110 E=1:F=F+C:G=G+D:GOTO
100
```

```
120 IF MID$(A$(F+C),G+D,
1)<>C$ OR E=0 THEN 160
130 MID$(A$(F),G,1)=C$
140 IF A=F AND B=G THEN
160
150 F=F-C:G=G-D:H=1:GOTO
 130
160 NEXT:NEXT
170 IF B$="C" OR H=1 THE
N 200
180 K=K+2:IF K<121 THEN
70
190 PRINT:PRINT"I cannot
 move":FL=1:FORZ=1TO500:
NEXT
200 CLS:IF B$="H" AND A>
0 AND FL=0 THEN PRINT "I
 moved in ";A-1;B-1:FORZ
=1TO400:NEXT
210 Z$=""
220 SOUND RND(1)*50,2
230 FORA=10 TO 2 STEP-1
240 PRINTTAB(4);A$(A)
250 FOR D=1TO500:NEXT:NE
XT
260 PRINTTAB(4);A$(1);
270 IF Z$="R" THEN RETUR
N
280 IF B$="C" THEN 50
290 PRINT:PRINT"Your mov
e"
300 PRINT"Enter 'R' to r
eprint"
310 PRINT"board,'C' to c
opy to"
320 PRINT"printer,'M' to
 move"
330 PRINT "'M' then 0, t
o pass";
340 GOSUB 760:IF Z$="R"
OR Z$="r" THEN GOSUB 220
:GOTO290
350 INPUT "Side number";
A:A=A+1:IF A<0 OR A>9 TH
EN 290
360 IF A=1 AND K>120 THE
N 450
```

```
370 IF A=1 THEN PRINT:PR
INT:PRINT:PRINT "OK, ple
ase stand by":GOTO 50
380 INPUT "Now across to
p";B:B=B+1:IF B<0 OR B>9
 THEN 380
390 B$="C":C$="H":GOTO90
400 DIM A$(10),T(120)
410 K=0:B$="H":FL=0
420 FOR Z=1TO10:READ Q$:
A$(Z)=Q$:NEXT
430 FOR Z=1TO120:READ T(
Z):NEXT
440 RETURN
450 PRINT:PRINT:PRINT"Th
e game is over"
460 C=0:H=0
470 FOR A=2 TO 9:FOR B=2
 TO 9
480 IF MID$(A$(A),B,1)="
C" THEN C=C+1
490 IF MID$(A$(A),B,1)="
H" THEN H=H+1
500 NEXT:NEXT
510 IF H>C THEN PRINT "Y
ou win!"
520 IF H<C THEN PRINT "I
 win!"
530 IF H=C THEN PRINT "I
t's a draw!"
540 FOR T=1 TO 1000:NEXT
550 PRINT "I scored ";C
560 FOR T=1 TO 1000:NEXT
570 PRINT "You scored ";
H
580 FOR T=1 TO 1000:NEXT
:PRINT:PRINT
590 PRINT "Press 'Y' for
"
600 PRINT "a new game,"
610 PRINT "'N' to end";
620 A$=INKEY$:IF A$<>"Y"
 AND A$<>"y" AND A$<>"N"
 AND A$<>"n" THEN 620
630 IF A$<>"N" AND A$<>"
n" THEN RUN
```

```
640 PRINT:PRINT:PRINT "O
K, thanks for
650 PRINT TAB(4);"playin
g"
660 END
670 DATA "*12345678*","1
.......1","2.......2",
"3.......3"
680 DATA "4...CH...4","5
...HC...5"
690 DATA "6.......6","7
.......7","8.......8",
"*12345678*"
700 DATA 1,1,1,8,8,1,8,8
,8,3,8,6,6,1,6,8,3,1
710 DATA 3,8,1,3,1,6,6,3
,6,6,3,3,3,6,6,4,6,5,3,5
,3,4,4,3,5,6
720 DATA 4,6,5,3,1,4,1,5
,8,4,8,5,4,1,4,8,5,1,5,8
,2,3,2,6
730 DATA 7,3,7,6,3,2,3,7
,6,2,6,7,2,4,2,5,7,4,7,5
,4,2
740 DATA 4,7,5,2,5,7,1,2
,1,7,8,2,8,7,2,1,2,8,7,1
,2,2,7,7,7,2,2,7
750 DATA 7,8
760 Z$=INKEY$
770 IF Z$<>"R" AND Z$<>"
r" AND Z$<>"M" AND Z$<>"
m" AND Z$<>"C" AND Z$<>"
c" THEN 760
780 IF Z$="C" OR Z$="c"
THEN GOSUB 810
790 PRINT
800 RETURN
810 FOR J=10 TO 1 STEP -
1
820 LPRINT TAB(4);A$(J)
830 NEXT
840 FOR J=1 TO 4:LPRINT:
NEXT
850 RETURN
```

Line 20 ensures that all variables used in the program will be integer variables, to maximise speed of execution. The screen is the cleared. Action is sent by line 30 to the subroutine beginning at line 400, the initialisation routine. Back after initialising, the program jumps to line 200 where the board is printed out.

Line 50 sets B$ to "H" and C$ to "C" (they are the other way around when it is the human's turn). Possible moves are held in the T array, and elements of the T array are selected by the variable K. That is, when K equals one, move T(1) will be considered.

Line 70 selects the first position to be considered, where T(K) + 1 equals the vertical co-ordinate, and T(K + 1) + 1 equals the horizontal co-ordinate. If line 80 (using MID$ as discussed before we introduced Reversi) finds that the position indicated does not contain a dot, indicating that the position is occupied, action moves to line 180, where K is incremented by two. If K is less than 121 (that is, all possible moves have not been discovered) the program goes back to 70 to consider the next move.

The routine from line 90 to 160 checks to see which pieces can be converted and makes the changes. Line 170 jumps over the routine for incrementing K, and line 190 (which points out "I cannot move") to 200, where the screen is cleared and the computer prints out the co-ordinates of its move.

The next section of the program prints out the board. There is a beep to let you know that the board will be printed (line 220) and then a loop is used to print out the elements of the A$ array. Line

250 slows the printing down, so you have a chance to see it. Line 270 returns action to the player move section if the board has just been reprinted at the request of the player, and line 280 sends action back to line 50 if C$ equals "C", which tells the computer the player's move is over.

The section from 290 to 330 is the menu of the possibilities open to the player after the board has been printed following a move by the computer:

```
Enter 'R' to reprint board,
      'C' to copy to printer,
      'M' to move,
      'M' then 0, to pass
```

Line 340 then turns the computer's attention to the subroutine beginning at line 760, which reads the keyboard to determine what the player wants to do. The player's vertical co-ordinate is accepted in line 350, and if the move was a zero (that is, A is one, as the one is added to the value entered by the human in line 350), line 360 checks to see if the computer has also passed (which will be the case if K is greater than 120) and if so, goes to line 450 where the game ends, and the pieces on the board are counted to see who has won.

Line 370 is actioned if the human has passed, and the computer has not. "OK, please stand by" appears on the screen and the computer starts looking for a new move. If the human has not passed with his or her first-entered number, the second (that is, horizontal) co-ordinate is accepted, the values of B$ and C$ are swapped, and action moves to the subroutine from line 90, which actually converts the

pieces. The same routine, you'll recall, was also used after the computer had moved.

We now get to the initialisation routine, to which we were sent by line 30. Two arrays are dimensioned in line 400 - the first one (A$) to hold the rows of the board and the second one (T) to hold the sequence of positions which are checked by the computer when it is looking for a move. Variables are initialised in line 410, then line 420 fills the A$ array, and line 430 fills the K array, before the RETURN sends attention back to near the beginning of the program.

"The game is over" declares the print statement in line 450, where indeed the 'end of game' routine begins. Variables C (to count the computer pieces) and H (for the human's) are set to zero in line 460, and the A and B loops (470 to 500) count how many pieces each player has on the board.

The result of the count is printed, along with the identity of the victor (lines 520 to 570). After a pause (line 580) the player is offered the option of starting a new game. If a new game is wanted, line 630 gets it underway. If not, "OK, thanks for playing" (lines 640 and 650) appears and the program terminates (line 660).

Lines 670 to 690 contain the DATA which is read into the A$ array (DATA, with its associated words READ and RESTORE, is on the agenda for discussion shortly.) The DATA for the T array, which determines the order in which squares are checked by the computer, is held in lines 700 to 750. (We'll get back to those in a moment.)

The final section of the program is a routine for reading the player's wishes after the board has been displayed, and (from line 810) for printing out the board is a hard copy has been requested.

You'll find that playing against the computer is a good way of improving your own skills at Reversi, and studying (perhaps even copying) the computer's strategy should help you advance fairly quickly to a good standard of play.

If you look at lines 700 to 750, you'll see the co-ordinates the computer uses when deciding where to move. This is the only strategy the computer has, a simple knowledge of the relative value of the positions on the board. However, armed with just this knowledge, the computer puts up a formidable game.

Here's a list of the positions, printed out by the computer, which shows the order in which it searches the squares. This order is not the only one it could have used, as several squares may have the same value at one time. You may wish to change the order around a bit in the DATA statements, and see what effect this has on the computer's game.

```
1 , 1     1 , 8     5 , 1     5 , 8
8 , 1     8 , 8     2 , 3     2 , 6
8 , 3     8 , 6     7 , 3     7 , 6
6 , 1     6 , 8     3 , 2     3 , 7
3 , 1     3 , 8     6 , 2     6 , 7
1 , 3     1 , 6     2 , 4     2 , 5
6 , 3     6 , 6     7 , 4     7 , 5
3 , 3     3 , 6     4 , 2     4 , 7
6 , 4     6 , 5     5 , 2     5 , 7
3 , 5     3 , 4     1 , 2     1 , 7
4 , 3     5 , 6     8 , 2     8 , 7
4 , 6     5 , 3     2 , 1     2 , 8
1 , 4     1 , 5     7 , 1     2 , 2
8 , 4     8 , 5     7 , 7     7 , 2
4 , 1     4 , 8     2 , 7     7 , 8
```

CHAPTER TEN - NOUGHTS AND CROSSES, AND AN INTRODUCTION TO STRUCTURED PROGRAMMING

Before we continue on with our step by step examination of the commands and functions available on your computer, let's take a look at a well-known game - NOUGHTS AND CROSSES - and discover how writing this program can serve as an introduction to the concept of 'structured programming'.

There are many ways to write a program to play Noughts and Crosses, the game in which the players each try to be the first get three of their symbol (either a nought or a cross) in a straight line on a three by three grid.

Most computer versions of this program suffer from one defect - they play too well, so it is impossible to beat them. The best you can do is draw, and once you've played a few games, you know exactly what the computer is going to do in a particular circumstance. These programs quickly become boring.

One way some programmers aim to give their programs fallibility is to program in a random 'mistake' from time to time, so the computer makes a stupid move (such as failing to complete a sequence when it is possible to do so) to allow the human to win. Again, such programs are unsatisfactory to play, and interest quickly wanes.

By contrast, as you've probably guessed by now, this version of the game allows you to win now and then (but not very often) and it is difficult to predict what the computer will do from certain positions, so

that its play does not appear repetitive (apart from the basic repetitive nature of the game). You'll find the computer is a good opponent, rarely allowing you to get the upper hand, but surrendering gracefully if you do.

After each game, the computer will print up the name of the winner, or that the game is a draw, play a little sequence of notes, then after a brief pause offer you a new game. As you'll see, there are several dummy loops to introduce slight pauses into the program, If you don't want these (and the game is very fast, even with them in) by all means delete them. If you find the sound routines annoying, by all means delete these as well.

Enter the program and play a few games with it, then get out of the program by using BREAK. You can then return to this book for a discussion on how the program works. This discussion will lead into an outline of the main ideas of structured programming, and an indication of how these ideas can help you write better programs.

The computer chooses who will go first, taking the first move about half the time. You move by touching the number which corresponds to the position where you want to move. The player input uses an INKEY$ routine, so there is no need to press RETURN after entering your move.

The positions, and their corresponding numbers, are:

```
1 2 3
4 5 6
7 8 9
```

Before we look at the listing, here are a couple of games. In the first game, the player moves first, with the computer going first in the second game. The computer won one game and drew the other. (The program used to print out the results is the same as the one given for printing on the screen, except that the PRINTs are changed to LPRINTs, a few LPRINTs are added to space things out, and the following line - 715 - is added to improve the appearance of the printout:

```
715 LPRINT:LPRINT:LPRINT
"----------------------"
:LPRINT

 ----------------------

 1  2  3     .  .  .
 4  5  6     .  .  .
 7  8  9     .  .  .
   Your move

 ----------------------

 1  2  3     .  .  .
 4  5  6     .  X  .
 7  8  9     .  .  .
   Please stand by

 ----------------------

 1  2  3     .  .  .
 4  5  6     .  X  .
 7  8  9     0  .  .
   Your move

 ----------------------

 1  2  3     .  .  .
 4  5  6     .  X  .
 7  8  9     0  .  X
   Please stand by
```

```
--------------------

1  2  3    0  .  .
4  5  6    .  X  .
7  8  9    0  .  X
  Your move

--------------------

1  2  3    0  .  .
4  5  6    X  X  .
7  8  9    0  .  X
  Please stand by

--------------------

1  2  3    0  .  .
4  5  6    X  X  0
7  8  9    0  .  X
  Your move

--------------------

1  2  3    0  .  .
4  5  6    X  X  0
7  8  9    0  X  X
  Please stand by

--------------------

1  2  3    0  0  .
4  5  6    X  X  0
7  8  9    0  X  X
  Your move

--------------------

1  2  3    0  0  X
4  5  6    X  X  0
7  8  9    0  X  X
  Please stand by

--------------------

1  2  3    0  0  X
4  5  6    X  X  0
7  8  9    0  X  X
It's a draw
```

Here's the game the computer wins:

```
I'll move first

---------------------

1  2  3     .  .  .
4  5  6     .  .  .
7  8  9     .  .  .

---------------------

1  2  3     .  .  .
4  5  6     .  0  .
7  8  9     .  .  .
  Your move

---------------------

1  2  3     .  .  .
4  5  6     .  0  .
7  8  9     .  X  .
  Please stand by

---------------------

1  2  3     .  .  0
4  5  6     .  0  .
7  8  9     .  X  .
  Your move

---------------------

1  2  3     .  .  0
4  5  6     .  0  .
7  8  9     X  X  .
  Please stand by

---------------------
```

```
1  2  3     .  .  0
4  5  6     .  0  .
7  8  9     X  X  0
  Your move

----------------------

1  2  3     .  .  0
4  5  6     .  0  X
7  8  9     X  X  0
  Please stand by

----------------------

1  2  3     0  .  0
4  5  6     .  0  X
7  8  9     X  X  0

I'm the winner!
```

And this is the listing, to allow you to challenge your computer to Noughts and Crosses:

```
10 REM NOUGHTS & CROSSES
20 DIM A(9)
30 RANDOMIZE (VAL(RIGHT$
(TIME$,2)))
40 CLS:FOR E=1 TO 500:NE
XT
50 IF RND(1)>.5 THEN PRI
NT "I'll move first":FOR
 E=1 TO 25:SOUND E,0.5:N
EXT:GOTO 90
60 GOSUB 700
70 GOSUB 430
80 GOSUB 620
90 GOSUB 700
100 GOSUB 430
110 IF A(5)=0 THEN A(5)=
1:GOTO 60
```

```
120 REM TO COMPLETE ROW/
BLOCK
130 PRINT:PRINT "  Pleas
e stand by";
140 D=1
150 B=1
160 IF B=1 THEN X=1:Y=2:
Z=3
170 IF B=2 THEN X=1:Y=4:
Z=7
180 IF B=3 THEN X=1:Y=5:
Z=9
190 IF B=4 THEN X=3:Z=7
200 C=1
210 IF A(X)=D AND A(Y)=D
 AND A(Z)=0 THEN A(Z)=1:
GOTO 60
220 IF A(X)=D AND A(Y)=0
 AND A(Z)=D THEN A(Y)=1:
GOTO 60
230 IF A(X)=0 AND A(Y)=D
 AND A(Z)=D THEN A(X)=1:
GOTO 60
240 IF B=1 THEN X=X+3:Y=
Y+3:Z=Z+3
250 IF B=2 THEN X=X+1:Y=
Y+1:Z=Z+1
260 IF C<3 THEN C=C+1:GO
TO 210
270 IF B<4 THEN B=B+1:GO
TO 170
280 IF D<2 THEN D=D+1:GO
TO 150
290 REM MOVE AT RANDOM
300 B=1
310 C=INT(RND(1)*9)+1
320 IF A(C)=0 THEN A(C)=
1:GOTO 60
330 B=B+1
340 IF B<21 THEN 310
350 B=0
360 B=B+1
370 IF A(B)=0 THEN A(B)=
1:GOTO 60
380 IF B<9 THEN 360
```

```
390 GOSUB 700
400 PRINT
410 PRINT "It's a draw";
420 GOTO 590
430 REM WIN CHECK
440 FOR B=1 TO 4
450 IF B=1 THEN X=1:Y=2:
Z=3
460 IF B=2 THEN X=1:Y=4:
Z=7
470 IF B=3 THEN X=1:Y=5:
Z=9
480 IF B=4 THEN X=3:Z=7
490 FOR C=1 TO 3
500 IF A(X)=A(Y) THEN IF
 A(Y)=A(Z) THEN IF A(X)<
>0 THEN 560
510 IF B=1 THEN X=X+3:Y=
Y+3:Z=Z+3
520 IF B=2 THEN X=X+1:Y=
Y+1:Z=Z+1
530 NEXT C
540 NEXT B
550 RETURN
560 PRINT
570 IF A(X)=1 THEN PRINT
 "I'm"; ELSE PRINT "You'
re";
580 PRINT " the winner!"
;
590 FOR X=1 TO 25:SOUND
X,0.5:NEXT
600 FOR X=50 TO 35 STEP
-1:SOUND X,0.5:NEXT
610 RUN

620 REM PLAYER MOVE
630 PRINT:PRINT "  Your
move";
640 A$=INKEY$
650 IF A$<"1" OR A$>"9"
THEN 640
660 B=VAL(A$)
670 IF A(B)<>0 THEN 640
680 A(B)=2
690 RETURN
```

```
700 REM PRINT OUT
710 CLS
720 PRINT "1  2  3   ";
730 FOR B=1 TO 9
740 IF A(B)=0 THEN PRINT
 " .  ";:SOUND 10,0.5:FOR
 E=1 TO 20:NEXT
750 IF A(B)=1 THEN PRINT
 " O ";:SOUND 20,1:FOR E
=1 TO 10:NEXT
760 IF A(B)=2 THEN PRINT
 " X ";:SOUND 30,2:FOR E
=1 TO 30:NEXT
770 IF B=3 THEN PRINT:PR
INT "4  5  6   ";
780 IF B=6 THEN PRINT:PR
INT "7  8  9   ";
790 NEXT B
800 RETURN
```

STRUCTURED PROGRAMMING

It's time now to have a look at how the program works, and introduce the idea of structured programming.

Line 20 dimensions the A array, which holds the board, and line 30 seeds the random number generator with a value derived from the time. As we saw earlier, this is a good way to ensure random responses are as close as possible to being truly random in a program where you need numbers produced over and over again within a very limited range.

The screen is cleared by line 40, there is a short delay (with the dummy E loop) and then line 50 determines which player will have the first move. If the computer decides it will have first move (that

is, the random number generated in line 50 is greater than 0.5) then action goes to line 90.

The most important part of the program is in the next few lines. These lines indicate a very important programming approach, one which is often given the title 'structured programming'. Structured programming calls for a 'top down' approach, in which the general intent of parts of the program is determined long before the specific code which will carry out specific actions is written.

The most important section of this program is lines 60 to 110. These lines are accessed time and time again during the program. The subroutine calls were written before the actual subroutines they called were written.

They do the following:

60 - calls the subroutine to print out the board
70 - calls the subroutine which checks to see if any player has won
80 - calls the subroutine to accept the player move
90 - calls printout subroutine
100 - calls the 'win check' subroutine
110 - checks to see if the center position is empty and if so, the computer moves to that position, then returns to line 60 where the loop begins again
120 - and following: this is the computer's move routine; once a move is made, action returns to line 60 to cycle through once again

A moment's thought will show that, with an approach such as this when important subroutines are called repeatedly from a loop, you gain at least two advantages:

1 - If you're not sure how to write a particular section of the program - such as the 'win check' routine - you can just put RETURN at the point in the program where the routine will eventually go, and get the rest of the program properly debugged before worrying about the routine. That is, you can check sections of the program, even when other sections are incomplete.

2 - You'll find it much easier to debug a program which is written in 'sections' like this, as the bug will probably only be within one section, and will subsequently be relatively easy to track down.

I urge you to use an approach like this whenever you can. We'll be looking at structured programming in greater detail in chapter 17, starting on page 313.

The routine from line 120 checks two things, both of which vital to winning Noughts and Crosses: whether there are two pieces in a row which can be completed to win, and whose pieces they are. If the computer finds that it has two pieces in line, it immediately places the third, then goes to the 'win check routine' which starts at line 700, via line 60. If it finds the two pieces are human ones, it moves to block. In either case, the computer moves into the vacant spot at the end of a row of three. The 'win check routine' then determines if the move was a winning one, and if not, knows it was a blocking move.

This routine continues until line 290 where, as the REM statement indicates, a move is chosen at random. The second from lines 300 to 340 chooses up to 20 moves at random.

If a possible move is not found in those 20 tries, then each square is checked in turn. If all the squares are found to be filled, the computer knows the game is a draw (because a win would have been detected earlier by the 'win check routine') and prints out its finding (line 420), then goes to 590, where a small tune is played before a new game begins.

The 'win check routine' starts at line 430. Firstly it checks possible wins horizontally, then vertically and finally diagonally. The routine exits at line 500 if a win is found. Normally, jumping out of a loop like this is not a good idea (in fact, it is considered very bad programming practice) but as it will only occur at the very end of a game, it is - I believe - acceptable at this point.

Generally, however, you definitely now jump out of loops in midstream. One alternative to exiting from a loop prematurely is demonstrated in the routine starting at line 120 where a series of...

```
IF X < 4 THEN X = X + 1: GOTO...
```

...is used instead (see, for example, line 270). If a win is not found, the RETURN sends action back to the master control loop. If, however, a win has been found, the lines from 560 to 610 print out the identity of the winner, plays a small sequence of notes, then starts a new game.

The player move routine starts at line 620. After printing up "Your move", the computer waits for a key to be pressed, using the INKEY$ routine in lines 640 and 650 to reject invalid moves. Once a number key between 1 and 9 is detected, the VAL equivalent of this key is assigned to variable B in line 660. As you'll recall from the earlier part of the book, when VAL is discussed, VAL turns a string into its numerical equivalent.

Line 670 checks to see if the chosen square is available (that is, that the designated eleement of the A array contains a zero). If it finds that is not, then action is returned to line 640 to get another move from the player. Once a valid move has been chosen, the relevant element of the A array is given the value two (a computer element has a value of one, an empty element is zero), and the RETURN sends attention back to the line following the one which sent the computer to the 'accept player move' subroutine.

The final section of the program, from line 700, prints out the board after each move. You may well want to modify this. As it stands, the computer is the O's, the human is the X's. Change the contents of the print statements in lines 750 and 760 if you wish to alter these. If you don't like the period for an empty position, change the contents of the quote marks in line 740 into whatever you desire.

Once you've played with this program a few times, you may be interested in writing a Noughts and Crosses program yourself, completely from scratch. Try not to follow the method I used (except for the

ideas of 'structured programming), but analyse the game from first principles, and try and create an original series of routines to play the game. It is not as difficult as you might think.

INKEY$

INKEY$ is a very useful command in many programs. It tells the computer to read the keyboard, and then return the key being pressed (or an empty string, "", if no key is being pressed) **without** you needing to press RETURN. This is very useful in programs which have menu selection, where you must, for example, enter a number or letter to choose from a number of menu items. If you look back to the Noughts and Crosses program listing, you'll see in lines 640 to 660 an INKEY$ routine for getting the number of the square into which you want to move.

It is similar to this one (although a little shorter):

```
10 A$=INKEY$
20 IF A$<"1" OR A$>"9" T
HEN 10
30 B=VAL(A$)
40 PRINT "B IS";B
50 PRINT
60 GOTO 10
```

When you run this, you'll see the computer will ignore any keys you touch except the unshifted number ones, and each time you press a key, it will print up a message like this:

```
B IS 1

B IS 5

B IS 3

B IS 9

B IS 8

B IS 5

B IS 6

B IS 3
```

You'll also find INKEY$ in use in many other programs in this book, such as the MINICALC one in the business section. It is a good idea to use INKEY$ when you can, as this is generally more effective than demanding that the user press RETURN after each key press.

WORD PROCESSOR

A very primitive word processor can be created within a few lines using INKEY$. This one is designed to work with upper case letters only. Use the hyphen key for a space, and the 1 key as a period. This program is limited, but it does allow you to use the HX20 as a sort of typewriter, dumping directly to the printer. You'll also see concatenation (addition) of strings, as discussed earlier in the book.

The program is quite fun to operate. If you want the computer to print out a line which is not complete (as it is, the computer waits till a line is longer than 20 characters before it dumps it to the

printer) just press the 2 key. You'll find the program prints a dash on the screen if you want a space, so that you can see you've indicated a space, but does not print the hyphen onto the printing paper.

```
10 REM Word processor
20 A$=""
30 PRINT
40 B$=INKEY$
50 IF B$="1" THEN A$=A$+
"."PRINT ".";
60 IF B$="2" THEN LPRINT
 A$:GOTO 20
70 IF B$="-" THEN A$=A$+
" ":PRINT "-";
80 IF B$<"A" OR B$>"Z" T
HEN 40
90 PRINT B$;
100 A$=A$+B$
110 IF LEN(A$)>20 THEN L
PRINT A$:GOTO 20
120 GOTO 40
```

```
THIS IS A DEMO TO
SHOW THE WORD
PROCESSOR PROGRAM
WORKING.  YOULL
FIND THAT ALTHOUGH
IT IS A TRIFLE
LIMITED IT CAN STILL
BE FUN TO OPERATE.
```

ADDING MACHINE/CASH REGISTER

INKEY$ can turn your HX20 into a simple-to-operate adding machine. You simply enter you number, digit by digit, including decimal points if you like. When

a particular number is finished and you're ready to enter the next one, press the "N" key (for 'next') and the display will scroll up, making way for the next number. When you want a total, press "T" and the numbers will automatically be added.

```
10 REM Adding machine
20 DIM C(30)
30 B$=""
40 A$=INKEY$
50 IF A$="T" THEN TTL=VA
L(B$):GOTO160
60 IF A$="N" THEN GOTO12
0
70 IF A$="." THEN B$=B$+
".":PRINT ".";
80 IF A$<"0" OR A$>"9" T
HEN 40
90 PRINT A$;
100 B$=B$+A$
110 GOTO 40
120 Q=Q+1
130 C(Q)=VAL(B$)
140 PRINT
150 GOTO 30
160 FOR D=1 TO 30
170 TTL=TTL+C(D)
180 NEXT D
190 PRINT:PRINT
200 PRINT "THE TOTAL"
210 PRINT "IS";TTL
```

If you'd prefer the computer to print out the numbers as you enter them onto the printer, as well as onto the screen, to make the computer more closely emulate an adding machine, use the following version of the program, which automatically ranges the output around the decimal place, for neat output:

```
10 REM Adding machine
20 REM with Print-out
30 DIM C(30)
40 B$=""
50 A$=INKEY$
60 IF A$="T" THEN TTL=VA
L(B$):GOTO190
70 IF A$="N" THEN GOTO13
0
80 IF A$="." THEN B$=B$+
".":PRINT ".";
90 IF A$<"0" OR A$>"9" T
HEN 50
100 PRINT A$;
110 B$=B$+A$
120 GOTO 50
130 Q=Q+1
140 Z=LEN(STR$(INT(VAL(B
$))))
150 LPRINT TAB(10-Z);B$
160 C(Q)=VAL(B$)
170 PRINT
180 GOTO 40
190 FOR D=1 TO 30
200 TTL=TTL+C(D)
210 NEXT D
220 PRINT:PRINT
230 LPRINT
240 LPRINT "THE TOTAL"
250 PRINT "THE TOTAL"
260 LPRINT "IS";TTL
270 PRINT "IS";TTL
```

```
      365.76
       23.45
     1000
       23.7

THE TOTAL
IS 1412.91
```

In its present form, the program can only add 30 numbers together. If you feel this is not enough, then change the 30 in lines 30 and 190 to, say, 100 or whatever number you feel will cover your needs.

CHAPTER ELEVEN - DATA, ARRAYS AND MORE DECISIONS

READ/DATA and RESTORE

This is an extremely useful trio of words which are usually found together within a program. Basically, the READ command is used to access information stored in a DATA statement, and RESTORE is used to indicate where - in a series of DATA statements - we wish to start accessing the information.

This program should make it clear:

```
10 FOR A=1 TO 10
20 READ B
30 PRINT B
40 NEXT A
50 RESTORE
60 SOUND 1,1
70 FOR C=1 TO 400:NEXT C
80 FOR A=1 TO 10
90 READ B
100 PRINT B
110 NEXT A
120 DATA 12,76,85,333,23
,43,65,88,2893,1
```

When you run this, you'll see the numbers in the DATA statement, line 120, printed out one by one as the computer goes through the first A loop, then again as it goes through the second one. When you run a program, it automatically RESTOREs to the first DATA statement in the program. When the computer comes across a RESTORE statement which is

not followed by a number, it RESTOREs to the first DATA statement in the program.

Here is a program to show RESTORE in action directly:

```
10 FOR A=1 TO 10
20 READ B
30 PRINT B
40 IF A=5 THEN SOUND 1,1
:RESTORE 70
50 FOR C=1 TO 50:NEXT C,
A
60 DATA 1,2,3,4,5,6,7,8
70 DATA 11,12,13,14,15,1
6,17,18
```

When you run this, you'll see it print out 1, 2, 3, 4, 5, 11, 12, 14, 15, 16. The RESTORE moved the data pointer to the start of line 70, and the computer started reading data from this point.

Now, delete lines 40 and 70, so your program looks like this:

```
10 FOR A=1 TO 10
20 READ B
30 PRINT B
50 FOR C=1 TO 50:NEXT C,
A
60 DATA 1,2,3,4,5,6,7,8
```

When you run it, you'll get an error message, OD Error In 20, caused by you trying to read more items of data than were provided in the program. It doesn't matter how many DATA items you have, so long as there are at least as many as the number to be read.

There is another thing about READ/DATA which you may find a trifle surprising. It does not matter where

in the program the DATA is scattered. The program will find the DATA items, and READ them, as this program shows:

```
10 DATA 1,2
20 FOR A=1 TO 11
30 DATA 3,4,5
40 READ B
50 DATA 6,7
60 PRINT B
70 DATA 8,9,10
80 NEXT A
90 DATA 11
```

So far in this discussion we've been reading numeric items of data, matching up variables with numbers within a DATA statement. Strings can be used, as the next program shows:

```
10 FOR A =1 TO 5
20 READ H$
30 PRINT H$
40 NEXT
50 DATA HI,THERE,MR,EPSO
N,COMPUTER
```

Notice that you do not need quote marks around the strings in the DATA statement in this example. If, however, you wanted to include a comma after the word THERE, and a period after the MR, you'd have to make the DATA statement as follows:

```
50 DATA HI,"THERE,","MR.
",EPSON,COMPUTER
```

Try this, to see it in action. You'll need quote marks if the DATA statement contains leading or trailing spaces, or punctuation marks, which you wish to preserve.

Numeric and string information can be mixed within a program, so long as you ensure that the computer

comes to a number when it expects and needs one, and comes to a string when it needs one. The next program, for example, manages to provide the right kind of information at the right time:

```
10 FOR A=1 TO 5
20 READ H$,B
30 PRINT H$,B
40 NEXT
50 DATA WORD,1,THIS,3,SO
NG,777,HEAVY,43,GOLLY,5
```

The computer can read more than one item of DATA at a time, as the program shows. You'll see READ, DATA and RESTORE in action in many programs in this book. The CHESS program is one which depends upon them heavily.

DIM

The DIM statement is used when you wish to set up an array which will hold a list of numbers, and which can be accessed by referring to the element of the array.

Here's an example which should make it clear. For an array to hold numbers, we use a statement of the form DIM A(20) where we want the A array to hold 21 elements. The number of elements in an array is always one more than the number specified in paretheses in the DIM statement. Each element of an array is filled with a zero when you first DIMension the array. The next program DIMensions an array in line 10 and then, using READ statements, will fill each element of the array with a number. The final section of the program will print out each element, in a way which should make it clear what is

happening. Notice that the first element of the array is A(0), the second is A(1)...and so on up to A(20).

```
10 DIM A(20)
20 FOR B=0 TO 20
30 A(B)=INT(RND(1)*10)+1
40 NEXT B
50 FOR B=0 TO 20
60 PRINT "A(";B;")  IS";
A(B)
70 NEXT B
```

Here's the result of running it:

```
A( 0 ) IS 6
A( 1 ) IS 3
A( 2 ) IS 6
A( 3 ) IS 7
A( 4 ) IS 2
A( 5 ) IS 9
A( 6 ) IS 1
A( 7 ) IS 10
A( 8 ) IS 4
A( 9 ) IS 6
A( 10 ) IS 5
A( 11 ) IS 1
A( 12 ) IS 6
A( 13 ) IS 6
A( 14 ) IS 4
A( 15 ) IS 5
A( 16 ) IS 2
A( 17 ) IS 10
A( 18 ) IS 2
A( 19 ) IS 3
A( 20 ) IS 3
```

An array is very useful if you wish to choose from a number of possibilities at random, as this program shows:

```
10 DIM A(5)
20 FOR B=0 TO 5
30 READ A(B)
40 NEXT
50 PRINT A(INT(RND(1)*6)
)
60 FOR C=1 TO 150:NEXT
70 GOTO 50
80 DATA 999,2343,111111,
2,0,43
```

The arrays we have looked at so far have been one dimensional. A multi-dimensional array is set by following the letter name of the array with more than one subscript, as this program shows:

```
10 DIM A(4,4)
20 FOR B=0 TO 4
30 FOR C=0 TO 4
40 A(B,C)=INT(RND(1)*9)
50 NEXT C,B
60 FOR B=0 TO 4
70 FOR C=0 TO 4
80 PRINT B;C;">";A(B,C)
90 NEXT C,B
```

Note that if you need eleven, or less, elements of an array, you do not need to dimension the array, as the computer will provide for eleven elements, as soon as one of them is called within a program. Here's an example:

```
10 A(3)=999
20 PRINT A(3)
30 PRINT A(7)
40 A(0)=A(3)+A(3)
50 PRINT A(0)
```

The computer also provides for string arrays, with a dollar sign in the dimension statement, and in subsequent references to the array. The presence of the dollar sign indicates that it is a string, and not a numeric, array. In the next program, a string array is set up in line 10, filled with the loop from lines 20 to 40, then elements are selected randomly by line 50:

```
10 DIM A$(5)
20 FOR B=0 TO 5
30 READ A$(B)
40 NEXT
50 PRINT A$(INT(RND(1)*6
))
60 FOR C=1 TO 150:NEXT
70 GOTO 50
80 DATA WORD,TEST,STAND,
DOWN,UP,ACROSS
```

You'll see arrays in use in many of the programs in this book.

OPTION BASE

This may be used to declare the minimum value for array variable subscripts. If you don't use it, the first element of an array is always zero. Using OPTION BASE 1 will ensure that the subscript of the first element in the array is 1.

ERASE

ERASE is used when you wish to wipe out a previously dimensioned array. You use the the command ERASE followed by the variable name of the array, such as A. More than one

array can be erased within a single line by following the word ERASE with the names of the arrays, separated by commas, as ERASE n, m, o. Then the arrays can be redimensioned.

MAKING COMPARISONS

The computer can use statements such as AND, OR and NOT to compare two or more things, and can act on the result of that comparison.

AND

If two statements to be checked are joined by an AND statement, then the computer will act only if both statements are true. The next program shows this:

```
10 A=100
20 B=200
30 IF A=100 AND B=200 TH
EN PRINT "YES"
```

OR

Two statements joined by an OR will lead to a 'true' finding if either of them is true. Our next program illustrates this:

```
10 A=100
20 B=200
30 IF A=100 OR B=100 THE
N PRINT "YES"
```

AND and OR can be used in the same program line, as this next example shows. In this program, the

computer will report "TRUE" if A and B are greater than 10 or if A equals B. Try it with a number of sample values for A and B, until you are sure you understand what is going on.

```
10 INPUT"A";A
20 INPUT"B";B
30 IF A>10 AND B>10 OR A
=B THEN PRINT "YES"
40 GOTO 10
```

NOT

This is used to produce, in effect, an 'opposite' condition to be tested, as you'll see in this brief routine:

```
10 INPUT"A";A
20 INPUT"B";B
30 IF NOT A=B THEN PRINT
 "NO"
40 GOTO 10
```

ARITHMETIC OPERATORS

The operators are > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to) and <> (not equal to).

These can all be used in decision-making lines, when the computer is trying to determine whether a statement is true or false. Enter this next program, trying it a number of times with different values, to see the effect of the arithmetic operators. Notice that they can be 'chained' with AND, or OR, or NOT.

```
10 INPUT"A";A
20 INPUT"B";B
30 IF A>B THEN PRINT "A
IS BIGGER THAN B"
40 IF A<B THEN PRINT "A
IS SMALLER THAN B"
50 IF A<B OR A+B=10 THEN
 PRINT "A IS SMALLER THA
N B","OR A+B EQUALS 10"
60 IF A<B AND A>10 AND B
>20 THEN PRINT "A IS LES
S THAN B","AND A IS GREA
TER THAN","10 AND B IS G
REATER",,"THAN 20"
70 FOR T=1 TO 200:NEXT
80 GOTO 10
```

CHAPTER TWELVE - BUSINESS APPLICATIONS

Many computers are launched without significant software support from the manufacturer. Commercial software companies watch the launch of a new computer, trying to gauge when there will be enough machines in the marketplace to justify spending time developing programs, whether they be games or business application programs. In the meantime, while waiting for programs which suit your needs to be developed, you can either adapt existing published programs from books or magazines, or write your own material from scratch. It is likely you will move from adapting programs into the field of writing your own.

If you know your business is unusual, and that a specific program would be very useful, it may well be worth the trouble and expense of hiring a free-lance programmer to create a program for you or modify a program which is currently available. Otherwise, books and magazines will be among your program sources.

There are a number of things to keep in mind when you decide you'd like to buy software for your HX20. You may be lucky enough to find exactly the program you need, which simply has to be loaded in and then run. However, a program which is tightly locked to your present method of doing business may prevent you from changing and developing your method of operation if the need arises.

Despite any claims you see in the advertising of programs, it is improbable that exactly the right program for your present and future needs exists

ready for purchase off the shelf. You must be prepared to work on the program to some extent.

Several companies are developing programs which are open enough to be tailored for a number of applications but are still tightly written enough to be of real use. You'll find these advertised and reviewed in the computing magazines. You may well find it worth hiring a freelance programmer for a short time to do the adapting to make the program suitable for your needs, or - once you've worked through this book - you may feel up to the task yourself. The software company may well be able to help you locate a program to modify the program for your own needs. You will possibly also find that the 'backup service' provided by the company, although certainly not going as far as a rewrite to order, will include advice on how you can modify the program to your needs. Another approach is to advertise for someone to come in on a part time basis to help you modify the program.

WHAT CAN YOU BUY?

The following programs are among those which are most generally available for small business use:

- payroll
- inventory
- accounts receivable
- accounts payable
- general ledger accounting
- word processing

You may well find you can buy a package which includes more than one of these programs in a single

integrated form. Many of the spread sheet accounting programs have developed versions which can perform a number of related tasks, and you may find such a program is suitable for your needs. To help you evaluate the use of the HX20 in business, our program Minicalc, which follows this discussion, not only performs a useful forecasting task by itself, but also will give you hands-on experience in working with such programs.

The whole field of budgetary control and forecasting is growing and is one of the areas where computers find most ready acceptance, because their value is so easy to appreciate. Using percentages and ratios for the analysis of financial statements is a growing field, and a program to assist with this may prove very valuable.

MINICALC

The Minicalc program, which can be very useful for extrapolating trends, allows the use of LPRINT to give you a permanent hard copy of its output. You can, however, use it with PRINT so the results are just shown on the screen. You are given a choice at the start of each RUN. Minicalc offers one of the facilities provided by spread sheet programs.

If you have any stream of data which represents returns of events occurring in sequence, and which appear to indicate a fairly steady development, you'll find applications for this program.

You could, for example, plot the cost of running a car over a two year period, and assuming you kept the same car, and did not do something radical to it

like having an accident or replacing the motor, could predict with some certainty the running costs for the following year. Car repairs tend to follow a trend which could be characterised by slowly rising costs, partly due to inflation and partly to the increasing age of the car.

Similarly, the number of rejects on a production line, with constantly improving quality control earlier in the production process, should lead to a gradually decreasing rejection rate. Entering known figures for rejects into Minicalc could provide you with an indication of the reject rate for three, six and nine months ahead, assuming your quality control improvement continues. You may well find that such things as the number of man hours lost due to industrial accidents in your plant shows a downward trend. Minicalc is ideal for producing a forward projection of this trend.

Many relationships can be extrapolated with this program, and so long as you do not run the projection too far into the future, and you watch for absurd output with 'excess extrapolation', you should find the information of value.

An example of excess extrapolation would be to enter the growth pattern in passenger use of a privately-owned bus service, until it exceeded the number of people in the area served by the buses, or exceeded the number of people in the area who did not have easy access to alternative means of transport. To suggest that because your company showed a gross improvement in output of five per cent per month for the last six months that it was probable that this growth pattern would continue month after month for

five years, and on the basis of projected income from the firm in five years time make massive capital investment now, would be placing too much reliance on a relatively short period of data collection.

Despite these cautionary examples, you'll still find Minicalc a valuable planning tool, especially if you use it to project for time periods which are similar to the time periods over which your entered data has been collected. That is, if you have sales figures from a particular territory for 12 months, and you'd like to see how the next 12 months shape up, assuming gross factors remain much the same over the coming year as pertained during the year for which data is available, you could use Minicalc with some confidence. To project the next decade's figures from a single 12 months' return would not be wise.

However, even this long range forecast could be of benefit in highlighting, for example, the residual deterioration in sales from a certain territory. While a one per cent drop per month in sales over a six month period might not seem too critical and could no doubt be blamed on external factors, projecting this for a further five years could highlight the seriousness of the problem. For example, entering six months' sales figures into the program (assuming the figures were 100 units, 99, 98, 97, 96 and 95) would show an average deterioration of 1.04%. Projecting this trend would show figures of 84 after 12 months, 74 after 24 months and 65 after 36 months, a falloff of more than a third.

On the other side of the coin, the output of a

growing trend can be a very encouraging source of good news. Assuming, for example, you projected future days lost through strike action, after you have followed a year-long process of improving management/worker relations, and entered figures for the last four quarters of 145 hours, 136, 122 and 104 lost, you'd find that if the trend continued over the next four quarters you'd only lose 91 man hours, 80, 71 and 62 respectively. Even if you doubt the reliability of a straight line projection of this type, you will probably agree that at the very least it gives additional information with which to make management decisions, and even if limited, this can be of value.

Before I give you the program listing, here are a couple of examples of the printed output of the program. The first one shows the number of components rejected per month (out of each 100,000 made) on a particular production line. As you can see, the reduction per month is not particularly regular, although the overall trend is towards lower reject rates per month. The program averages the rate of change, and then assumes this rate of change will remain constant in the coming 10 months. Although the program listing and output refers to time periods called 'months', it can obviously be altered or taken to refer to any time period you desire, from nanoseconds to years.

```
Recorded figures:
Month 1  2987
Month 2  2980
Month 3  2964
Month 4  2953
```

```
Month 5  2948
Month 6  2937
Month 7  2906
Month 8  2899
Month 9  2876
Month 10  2870

------------------------

Difference between
months:
Months 1 - 2 -.24 %
Months 2 - 3 -.54 %
Months 3 - 4 -.38 %
Months 4 - 5 -.17 %
Months 5 - 6 -.38 %
Months 6 - 7 -1.07 %
Months 7 - 8 -.25 %
Months 8 - 9 -.8 %
Months 9 - 10 -.21 %

------------------------

Average change-.45 %

------------------------

Now a projection
   of change:

Month 1 - recorded
                2870
Month 2 - 2857
Month 3 - 2844
Month 4 - 2831
Month 5 - 2818
Month 6 - 2806
Month 7 - 2793
Month 8 - 2780
Month 9 - 2768
Month 10 - 2755

------------------------
```

The second sample shows sales (in units of $10,000) recorded by a suburban hifi store over a period of six weeks (and note that the reference to 'months' in the program output is taken to mean 'weeks'). Ignoring the effect of events like Christmas on sales patterns, the retailer was interested in learning what sort of sales she could expect over the next 24 weeks. Note that the program gives two options as the basis for projection - the figure for the last recorded month, or an average of all entered figures. The first projection shown here is based on the 'last month' figure, and the second one on the average per month figure. As can be seen, this second projection is more conservative, and may prove a more useful basis for decision-making.

```
------------------------

Recorded figures:
Month 1  27.64
Month 2  27.94
Month 3  28.05
Month 4  28.05
Month 5  28.96
Month 6  29.71

------------------------

Difference between
months:
Months 1 - 2  1.07 %
Months 2 - 3  .39 %
Months 3 - 4  0 %
Months 4 - 5  3.14 %
Months 5 - 6  2.52 %

------------------------

Average change 1.42 %

------------------------
```

```
Now a projection
   of change:

Month 1 - recorded
                  29.71
Month 2 - 30
Month 3 - 30
Month 4 - 31
Month 5 - 31
Month 6 - 32
Month 7 - 32
Month 8 - 33
Month 9 - 33
Month 10 - 34
Month 11 - 34
Month 12 - 35
Month 13 - 35
Month 14 - 36
Month 15 - 36
Month 16 - 37
Month 17 - 37
Month 18 - 38
Month 19 - 38
Month 20 - 39
Month 21 - 39
Month 22 - 40
Month 23 - 40
Month 24 - 41

----------------------
```

This is the projection based on the average return per week:

```
----------------------

Now a projection
   of change:

Month 1 - recorded
                  29.71
Month 2 - 28
Month 3 - 28
Month 4 - 29
Month 5 - 29
Month 6 - 30
```

```
Month 7 - 30
Month 8 - 30
Month 9 - 31
Month 10 - 31
Month 11 - 32
Month 12 - 32
Month 13 - 33
Month 14 - 33
Month 15 - 34
Month 16 - 34
Month 17 - 35
Month 18 - 35
Month 19 - 36
Month 20 - 36
Month 21 - 37
Month 22 - 37
Month 23 - 38
Month 24 - 38

----------------------
```

And this is the complete listing for Minicalc:

```
10 REM Minicalc
20 CLS
30 GOSUB 870
40 PRINT "Enter number o
f":PRINT"months for whic
h":PRINT "figures are":I
NPUT"available";M:IF M<2
 THEN 40
50 TT=0
60 CLS
70 DIM A(M),B(M)
80 IF Z=1 THEN LPRINT "R
ecorded figures:"
90 FOR A=1 TO M
100 PRINT "Enter figure
month";A
110 INPUT A(A)
120 IF Z=1 THEN LPRINT "
Month";A;A(A)
130 TT=TT+A(A)
140 NEXT
```

```
150 AV=TT/M
160 FOR B=2 TO M
170 B(B)=(100-(A(B-1)*10
0/A(B)))
180 NEXT
190 CLS
200 PRINT "Difference be
tween"
210 PRINT "months:"
220 PRINT
230 IF Z=1 THEN LPRINT:L
PRINT "----------------
-------":LPRINT
240 IF Z=1 THEN LPRINT "
Difference between":LPRI
NT "months:"
250 FOR A=2 TO M
260 PRINT "Months";A-1;"
-";A;INT(100*B(A))/100;"
%"
270 IF Z=1 THEN LPRINT "
Months";A-1;"-";A;INT(10
0*B(A))/100;"%"
280 NEXT
290 FOR T=1 TO 2000:NEXT
300 TT=0
310 FOR A=2 TO M
320 TT=TT+B(A)
330 NEXT
340 AVE=INT(TT*100/(M-1)
)/100
350 CLS
360 IF Z=1 THEN LPRINT:L
PRINT "----------------
-------":LPRINT
370 PRINT "Av. change:";
AVE;"%"
380 IF Z=1 THEN LPRINT "
Average change";AVE;"%"
390 FOR T=1 TO 2000:NEXT
400 CLS
410 PRINT "Now a project
ion"
420 PRINT "    of change
:"
430 IF Z=1 THEN LPRINT:L
PRINT "----------------
-------":LPRINT
```

```
440 IF Z=1 THEN LPRINT "
Now a projection":LPRINT
 "   of change:"
450 FOR T=1 TO 1500:NEXT
460 PRINT:PRINT"How many
 month's"
470 PRINT"projection do
you"
480 INPUT"want";NU
490 CLS
500 PRINT "Final month:"
;A(M)
510 PRINT "Av. per month
:";AV
520 FOR T=1 TO 1500:NEXT
530 PRINT "Projection on
:"
540 PRINT TAB(3);"1 - fi
nal month"
550 PRINT "or 2 - averag
e";TAB(7);"per month?"
560 A$=INKEY$
570 IF A$<"1" OR A$>"2"
THEN 560
580 D=VAL(A$)
590 CLS
600 IF Z=1 THEN LPRINT
610 E=(A(M) AND D=1)+(AV
 AND D=2)
620 PRINT "Month 1 - rec
orded"
630 PRINT ,A(M)
640 IF Z=1 THENLPRINT "M
onth 1 - recorded"
650 IF Z=1 THEN LPRINT ,
A(M)
660 FOR A=2 TO NU
670 E=E+AVE*E/100
680 PRINT "Month";A;"-";
INT(E)
690 FOR T=1 TO 500:NEXT
700 IF Z=1 THEN LPRINT "
Month";A;"-";INT(E)
```

```
710 NEXT
720 IF Z=1 THEN LPRINT:L
PRINT "----------------
-------":LPRINT:LPRINT:L
PRINT
730 FOR T=1 TO 1500:NEXT
740 PRINT:PRINT
750 PRINT "1 - Projectio
n again"
760 PRINT "2 - Output ag
ain"
770 PRINT "3 - Start aga
in"
780 PRINT "4 - To end";
790 IF INKEY$<>"" THEN 7
90
800 A$=INKEY$
810 IF A$="" THEN 800
820 IF A$="1" THEN 400
830 IF A$="2" THEN 190
840 IF A$="3" THEN RUN
850 IF A$="4" THEN CLS:P
RINT:PRINT:PRINT "Thank
you":END
860 GOTO 790
870 PRINT "Enter 1 if yo
u"
880 PRINT "want a copy o
n"
890 PRINT "the printer..
."
900 PRINT "2 if you don'
t";
910 A$=INKEY$
920 IF A$<"1" OR A$>"2"
THEN 910
930 Z=VAL(A$)
940 IF Z=1 THEN LPRINT:L
PRINT "----------------
-------":LPRINT
950 CLS
960 RETURN
```

Line 30 sends action to the subroutine from line 879, which allows you to select screen only output, or screen plus printer output. In line 40, you are asked to enter the number of months for which you have information available. You'll recall that the word 'month' can be taken to stand for any time period you like, although it should be noted that if you decide 'months' stands for 'weeks' in the input part of the program, 'months' must also stand for 'weeks' at the output section.

If your activities are habitually divided into other time periods than months, you may wish to enter the other time period as you enter the program, every time you come to 'month' or 'months' in the listing. If you have no intention of using printer output (although you'll be depriving yourself of a real benefit by deciding not to allow the option of a hard copy of results), you can omit any lines (such as 230) which begin "IF Z = 1 THEN...".

Line 50 sets the variable TT, which will hold the total figures for the period entered, to zero. The screen is cleared in line 60 and two arrays dimensioned to hold the entered information (array A) and the difference between subsequent months (array B). The loop from lines 90 to 140 accepts and totals the enetered data, and line 150 works out the average per month, assigning this value to the variable AV.

The loop in lines 160 to 180 works out the difference between months, in percentage terms, assigning this information to elements of the B array. Line 190 clears the screen again, in preparation for printing out this information in the loop from lines 250 to 280. Line 290 (as well as

lines 390, 450, 520, 690 and 730) uses a dummy loop (control variable T) to delay action of the program to allow the user to assimilate the information before continuing. The variable TT is again set to zero in line 300, and the loop from 310 to 330 sums all the percentage differences, then line 340 assigns the variable AVE to the average of the these.

After clearing the screen in line 350, line 370 prints out the result of this calculation (that is, the average of the percentage changes from month to month). The screen is cleared again (line 400) and the projection part of the program begins. You enter the number of months for which you want an extrapolation (line 480) and then you're asked whether you want the projection based on the average return per month, or on the final month's figures. An INKEY$ input is used at this time (lines 560 and 570) to remove the need to press RETURN after touching the digit indicating your choice. A routine like this is ideal for interpreting a choice from a menu when the choices can be indicated with a single key press.

Line 610 uses the computer's method of interpreting logic statements to assign E (the base upon which the projection will be built) to either the final month's return, or the average month's return, depending upon which one you've indicated as your choice. The loop from 660 to 710 actually makes (line 670) and then prints out the result of the extrapolation.

The menu between lines 750 and 780 outlines your choices once the program is over. You can:

- run the program again (changing the base if you like)
- see the program run again, without having to re-enter the initial data
- run the program from scratch
- terminate the run

If you've run the program once, and you had not selected the option for printout, but now decide it would be good to have a printout and do not want to re-enter the figures, select option four from the menu, which terminates the program. Then enter as a direct command Z = 1:GOTO 190 then press RETURN. You will not have a printout of your original data, but you will have the most important part of a run, the projection. Notice how INKEY$ is used in this section (as it was in lines 560 and 570) to remove the need to press RETURN after selecting one of the options. The final routine, from line 910, is the printer option selection routine.

PERSONAL FINANCE

Our next program - PERSONAL FINANCE - will not only prove useful in its own right, but your experience working with it should aid you in evaluating business software for your own needs. In fact, you may well be able to modify it to work as a budgetary control program for a small business, or learn enough from starting and running it to go on to develop a financial planning program of your own.

The program is designed to both help you balance

your checkbook, and to project future recurrent costs against your income. If you have, for example, to pay $399.50 per month on your housing loan, $250 per month on payments on your boat, and $22.56 for newpaper deliveries, you can enter these costs and your current balance, then get the computer to subtract the total cost from the balance. The program will accept deposits from you, and keep you informed of the running balance. It is designed to lead you step by step through looking at your checking account balance.

When you first run the program, you'll be presented with a menu of four choices:

1 - to start again
2 - deposit
3 - modify schedule
4 - stop

When you turn on the computer, press "1" as the option. 'Enter number of items to be paid for each month' is the next instruction you'll be given. It is best to be generous here, perhaps entering a higher number than the maximum number of items you think you'll need because although you can leave some items blank, you cannot add additional items once the program is running (although you can easily change an item - even a blank one - into another item, so blanks can be used for later additions).

Look at the sample run of the program which follows this introduction. Following it through will give you a pretty good idea of how the program runs. You'll find that, once the program is underway, it leads you neatly, step by step, through the needed

procedure. Once you've read through the sample run, and worked out what is going on, enter the program, and try running it. Then return to the book for a detailed explanation of what the various sections of the program do.

Here is the sample run:

```
Current balance:
    $ 0

1 - start again
2 - deposit
3 - modify schedule
4 - stop
Enter number of
items to be
paid for
each month
Enter name of
item number 1
MORTGAGE

And how much is
MORTGAGE
each month
MORTGAGE $ 345.67
Enter name of
item number 2
CAR

And how much is
CAR
each month
MORTGAGE $ 345.67
CAR $ 146.37
Enter name of
item number 3
SERVICES
```

```
And how much is
SERVICES
each month
MORTGAGE $ 345.67
CAR $ 146.37
SERVICES $ 75

Enter X if correct
C to change one
A to see list again
Enter last known
balance
$ 1287.98
Enter deposits,
one by one,
end with 'E'$ 1331.54
Enter deposits,
one by one,
end with 'E'$ 1457.21
Enter deposits,
one by one,
end with 'E'
Enter 1 to subtract
all of payments,
or 2 for menu

Cost schedule
totals $ 567.04


Balance before
current costs:
     $ 1457.21

Current balance:
     $ 890.17

1 - start again
2 - deposit
3 - modify schedule
4 - stop
Thank you
```

And here is the program listing:

```
10 REM Personal Finance
20 CLS
30 BAL=0
40 GOTO 830
50 PRINT "Enter number o
f"
60 PRINT "items to be"
70 PRINT "paid for"
80 PRINT "each month";
90 INPUT NUM
100 CLS
110 DIM A$(NUM),A(NUM)
120 FOR A=1 TO NUM
130 PRINT "Enter name of
"
140 PRINT "item number";
A
150 INPUT A$(A)
160 IF LEN(A$(A))<2 THEN
 150
170 PRINT "And how much
is"
180 PRINT A$(A)
190 PRINT "each month";
200 INPUT A(A)
210 FOR Z=1 TO A
220 PRINT A$(Z);" $";A(Z
)
230 NEXT
240 NEXT
250 PRINT "Enter X if co
rrect"
260 PRINT "C to change o
ne"
270 PRINT "A to see list
 again"
280 B$=INKEY$
290 IF B$<>"X" AND B$<>"
x" AND B$<>"C" AND B$<>"
c" AND B$<>"A" AND B$<>"
a" THEN 280
300 IF B$="X" OR B$="x"
THEN 480
```

```
310 IF B$="A" OR B$="a"
THEN PRINT: FOR Z=1 TO N
UM:PRINT A$(Z);"  $";A(Z
):NEXT:GOTO 250
320 PRINT "Enter number
of"
330 PRINT "item you wish
 to"
340 INPUT "change";B
350 PRINT:PRINT
360 PRINT "Enter new nam
e for"
370 PRINT "item number";
B
380 INPUT A$(B)
390 PRINT:PRINT
400 PRINT "And how much
is"
410 PRINT A$(B)
420 INPUT "each month";A
(B)
430 CLS
440 FOR A=1 TO NUM
450 PRINT A$(A);" -$";A(
A)
460 NEXT
470 GOTO 250
480 CLS
490 IF BAL<>0 THEN 520
500 PRINT "Enter last kn
own"
510 INPUT "balance";BAL
520 CLS
530 PRINT "$";BAL
540 PRINT "Enter deposit
s,"
550 PRINT "one by one,"
560 PRINT "end with 'E'"
;
570 INPUT Q$
580 IF Q$="" THEN 570
590 IF Q$="E" OR Q$="e"
THEN 620
600 BAL=BAL+VAL(Q$)
610 GOTO 530
```

```
620 PRINT "Enter 1 to su
btract"
630 PRINT "all of paymen
ts,"
640 PRINT "or 2 for menu
";
650 C$=INKEY$
660 IF C$<"1" OR C$>"2"
THEN 650
670 C=VAL(C$)
680 IF C=2 THEN 870
690 SD=0
700 FOR A=1 TO NUM
710 SD=SD+A(A)
720 NEXT
730 PRINT:PRINT
740 PRINT "Cost schedule
"
750 PRINT "totals $";SD
760 FOR T=1 TO 1500:NEXT
770 CLS
780 PRINT "Balance befor
e"
790 PRINT "current costs
:"
800 PRINT TAB(4);"$";BAL
810 BAL=BAL-SD
820 FOR T=1 TO 2000:NEXT
830 PRINT "Current balan
ce:"
840 PRINT TAB(4);"$";BAL
;
850 IF BAL<0 THEN SOUND
1,2:PRINT "Overdrawn"
860 FOR T=1 TO 5000:NEXT
870 PRINT:PRINT:PRINT
880 PRINT "1 - start aga
in"
890 PRINT "2 - deposit"
900 PRINT "3 - modify sc
hedule"
910 PRINT "4 - stop";
920 IF INKEY$<>"" THEN 9
20
930 C$=INKEY$
```

```
940 IF C$<"1" OR C$>"4"
THEN 930
950 PRINT:PRINT:PRINT
960 C=VAL(C$)
970 IF C=1 THEN CLEAR:GO
TO 50
980 IF C=2 THEN 530
990 IF C=3 THEN 430
1000 IF C=4 THEN PRINT:P
RINT"Thank you":END
1010 GOTO 830
```

Now let's have a detailed look at the contents of the program. The variable BAL holds the balance of your account, and this is set to zero in line 30, before line 40 sends action to the routine from line 830 which displays the opening menu. The routine from line 50 is the destination of item one from the menu ('to start again'). You are asked to enter the number of items to be paid for each month, and line 90 accepts this number, assigning variable NUM to it.

The screen is cleared (line 100) and two arrays dimensioned in line 110, one to hold the name for each item in the payment schedule, and the other to hold the corresponding value. The loop starting at line 120 allows you to enter the schedule names (line 150) and values (line 200), reprinting the list of items entered so far (using the loop from lines 210 to 230) after each new item is entered.

Once you've finished entering the items, you are given the chance to modify, correct or otherwise change the list. You touch the "X" key if you are happy with the list, the "C" if you wish to change

an item, and the "A" to see the list again. The routine in lines 280 and 290 reads the keyboard, rejecting inappropriate inputs. If you decide the list is OK (that is, you have entered an "X") the program continues on from line 480. Line 310 arranges for a reprint of the list, and the routine from 320 to 420 accepts a single modification of an item and its price. The computer then returns to line 250, to allow you to approve the change, and if necessary to make more.

Line 480 clears the screen and if the balance is not zero (that is, the variable BAL has a value other than zero, see line 490) the program jumps the small section in lines 500 and 510 which allows you to enter the 'last known balance'. Line 530 prints the balance, and then the deposit routine begins. The sequence from 530 to 610 is repeated, each time allowing a new deposit to be entered, and each time reprinting the balance as it stands after that deposit has been made. The series of deposits is terminated by entering "E".

Note how the input values for the deposits are accepted as strings, then VAL is used to reject input where just RETURN has been pressed without a preceding number (line 580) and, of course, caters for the "E" which signals that there are no more deposits to be entered.

Once the deposits have been added into the balance, you are given the choice of subtracting all the payments in your schedule from the balance, or of going to the menu. Going to the menu at this point is useful if you decide you want to add additional deposits, or you wish to modify the payment schedule

in some way before subtracting the total of the schedule from the balance. An INKEY$ routine is used here to get either "1" or "2".

The variable SD is used for the scheduled items, totalling them in the loop 700 to 720. The result of this loop calculation is shown in lines 740 and 750. There is a short delay in line 760 before the screen clears again. The 'balance before current costs' is printed, followed by a delay, then the current balance is shown. Line 850 ensures that if the account is overdrawn, the message "Overdrawn" appears next to the final (negative) balance, and a tone sounds. Line 860 pauses before scrolling up (line 870) to present the menu again.

OTHER APPLICATIONS

We will now look at a few additional financial tasks in which your HX20 can prove useful, giving a short program to carry out the job in each case, and an example of it in use.

PERCENTAGE CHANGE

If you know what a figure is, and wish to find out what that figure plus a percentage of it is (such as finding out how much your house will be worth in a

year, if an increase of 5.5% is expected) you can use the following:

```
10 REM Percentage change
20 INPUT "Current worth"
;C
30 INPUT "Percentage cha
nge";P
40 PRINT "Increase (I) o
r"
50 INPUT "decrease (D)";
D$
60 IF D$="D" THEN P=-P
70 N=(1+P/100)*C
80 PRINT "The new figure
"
90 PRINT "is ";N
```

Here are two examples of it in action:

```
Current worth?
 35700
Percentage change?
 5.5
Increase (I) or
decrease (D)?
I
The new figure
is  37663.5
```

```
Current worth?
 35700
Percentage change?
 10
Increase (I) or
decrease (D)?
I
The new figure
is  39270
```

SIMPLE INTEREST

There are four variables involved in simple interest:

- principal
- rate
- time
- interest

When you run the program, you enter the figures you have, entering a zero for the figure you do not know. The computer will use the correct version of the formula to produce the unknown value. Note that TIME must be entered in years (with decimal fractions allowed, such as 3.5 for three and a half years), and the RATE must be a single figure which represents the percentage rate (such as 7.25 for 7.25%).

Here is the listing:

```
10 REM SIMPLE INTEREST
20 FOR T=1 TO 500:NEXT
30 CLS
40 INPUT "PRINCIPAL";P
50 INPUT "RATE";R
60 R=R/100
70 INPUT "TIME";T
80 INPUT "INTEREST";I
90 IF P=0 THEN GOSUB 130
:RUN
100 IF R=0 THEN GOSUB 16
0:RUN
110 IF T=0 THEN GOSUB 19
0:RUN
120 IF I=0 THEN GOSUB 22
0:RUN
```

```
130 PRINT "PRINCIPAL IS"
140 PRINT I/(R*T)
150 RETURN
160 PRINT "RATE IS"
170 PRINT 100*I/(P*T);"%
"
180 RETURN
190 PRINT "TIME IS"
200 PRINT I/(P*R)
210 RETURN
220 PRINT "INTEREST IS"
230 PRINT P*R*T
240 RETURN
```

Generally, we will know the amount borrowed (the principal), the interest rate being charged, and the period over which the loan is made. The program will output the amount of interest which must be paid, as the following sample run indicates. We are borrowing $500, at 7%, for two and a half years, and want to know much interest will have to be paid:

```
PRINCIPAL?
 500
RATE?
 7
TIME?
 2.5
INTEREST?
 0
INTEREST IS
 87.5
```

Here the answer "INTEREST IS 87.5" comes up, so we must repay principal plus interest, that is $587.50.

In the next example, I want to borrow $1000 for 10 years, and am told to do so I must pay $175

interest. I can use the formula to find out what simple rate of interest I am being charged:

```
PRINCIPAL?
 1000
RATE?
 0
TIME?
 10
INTEREST?
 175
RATE IS
 1.75 %
```

The rate is a mere 1.75!

Finally, here is another example where I know the amount I wish to borrow ($5000) and the period (five years). I am told I must pay $6125 in total (that is, $1125 more than I borrowed). What interest rate am I being charged? The program produces the answer immediately:

```
PRINCIPAL?
 5000
RATE?
 0
TIME?
 5
INTEREST?
 1125
RATE IS
 4.5 %
```

The rate is 4.5%.

COMPOUND INTEREST

The formulae involved in working out sums involved with compound interest are not as simple as those

used in the previous example. Whereas simple interest is worked out simply as a percentage of the principal, with compound interest, the interest earned is added - from time to time - to the principal, so the whole amount attracts further interest.

The formula is (1 + INTEREST/100) raised to the power of the number of years, with the result multiplied by the starting principal. Here is a program to do it:

```
10 REM COMPOUND INTEREST
20 INPUT "PRINCIPAL";P
30 INPUT "TIME";T
40 INPUT "RATE";R
50 A=P*((1+R/100)^T)
60 PRINT "$";P;"BECOMES"
70 PRINT "$";A;"AFTER"
80 PRINT T;"TIME PERIODS
"
```

Notice the reference to 'time periods'. This is to allow you to work on situations where the compound interest is added monthly, or even daily, rather than annually, to the figure which is attracting interest. You must make sure, of course, that the time rate you enter in line 30 is the time rate which is relevant for the interest.

Here's the result of running the program with a principal of $1000 invested for seven years, at 8.25% annual compound interest:

```
PRINCIPAL?
 1000
```

```
TIME?
 7

RATE?
 8.25

$ 1000 BECOMES
$ 1741.79 AFTER
 7 TIME PERIODS
```

This second example uses $1000, at 10% annual compound interest, for 73 years. The result is rather surprising:

```
PRINCIPAL?
 1000

TIME?
 73

RATE?
 10

$ 1000 BECOMES
$ 1.05115E+06 AFTER
 73 TIME PERIODS
```

Yes, $1000 at 10% compound interest grows to over one million dollars in 73 years. If you deposited $1000 for each of your grandchildren now at that rate, they could all be millionaires in due course (although how much one million dollars would be worth in spending power at that time might not be very exciting).

Here's a final thought on compound interest which you can demonstrate with your computer. Dividing 70 by the interest rate will give the approximate

period it takes money to double itself. I'll demonstrate that by entering into the program the following figures:

- principal $1000
- time 70/10 (seven years)
- rate 10%

```
PRINCIPAL?
 1000

TIME?
 7

RATE?
 10

$ 1000 BECOMES
$ 1948.72 AFTER
 7 TIME PERIODS
```

REPAYMENTS ON MORTGAGE

Probably the biggest sum of money you will ever borrow is to buy your house. The formula used takes into account that the early repayments are almost entirely repaying interest, while the later ones are repaying more principal than interest.

This program will tell you what repayments should be on a housing loan, and also how much you will pay back altogether. This final figure is, I warn you, somewhat depressing.

```
10 REM HOUSING LOAN
20 INPUT "PRINCIPAL";P
30 INPUT "TIME";T
40 INPUT "ANNUAL INT. RA
TE";R
50 R=R/100
60 REP=((1+R)^T)*R*P/(((
1+R)^T)-1)
70 PRINT "ANNUAL REPAYME
NT"
80 PRINT "IS $";REP
90 PRINT "MONTHLY REPAYM
ENT"
100 PRINT "IS $";REP/12
110 FOR Z=1 TO 500:NEXT
120 PRINT "TOTAL TO BE"
130 PRINT "REPAID IS"
140 PRINT "$";REP*T
```

Here is the result of borrowing $45,000 for 25 years at 14.5% interest:

```
PRINCIPAL?
 45000

TIME?
 25

ANNUAL INT. RATE?
 14.5

ANNUAL REPAYMENT
IS $ 6753.78

MONTHLY REPAYMENT
IS $ 562.815

TOTAL TO BE
REPAID IS
$ 168844
```

The final sum repaid - $168,884 - seems a vast jump from the amount borrowed.

Here is the saving realised if a loan at half a per cent lower can be arranged:

```
PRINCIPAL?
 45000

TIME?
 25

ANNUAL INT. RATE?
 14

ANNUAL REPAYMENT
IS $ 6547.43

MONTHLY REPAYMENT
IS $ 545.619

TOTAL TO BE
REPAID IS
$ 163686
```

Over $5000 less overall is to be repaid, with a monthly payment that is some $17 cheaper.

Finally, if you want a version of the program which gives a hard copy printout of the results, use the following:

```
10 REM HOUSING LOAN
20 INPUT "PRINCIPAL";P
30 LPRINT "PRINCIPAL?"
40 LPRINT P:LPRINT
50 INPUT "TIME";T
60 LPRINT "TIME?"
70 LPRINT T:LPRINT
```

```
80 INPUT "ANNUAL INT. RA
TE";R
90 LPRINT "ANNUAL INT. R
ATE?"
100 LPRINT R:LPRINT
110 R=R/100
120 REP=((1+R)^T)*R*P/((
(1+R)^T)-1)
130 LPRINT "ANNUAL REPAY
MENT"
140 LPRINT "IS $";REP
150 LPRINT
160 LPRINT "MONTHLY REPA
YMENT"
170 LPRINT "IS $";REP/12
180 FOR Z=1 TO 500:NEXT
190 LPRINT
200 LPRINT "TOTAL TO BE"
210 LPRINT "REPAID IS"
220 LPRINT "$";REP*T
```

FILE HANDLING

RAM files

The HX20 features random access memory file data (RAM files) which can easily be changed. This information is not lost when you turn the computer off, and it can be used by more than one program (there is no way, in fact, to restrict the DATA to a single program).

The files are stored in a single alloted area of the HX20's memory, but may be divided (in terms of use, if not in fact) by the DEFFIL statement which holds the position of the first item stored for that particular file, and directs the action of subsequent PUT% (place data on file) and GET% (extract it) calls in that program.

RAM files are instantly accessible (in contrast to sequential files which we'll be looking at shortly) but the memory available for these files is limited, and you have to work out exactly what demands a particular file must satisfy when you set up the file.

The nature of the information you store effects how much memory you will need. Integer variables take up two bytes each, single precision variables four bytes and double precision variables double this, at eight bytes each. String variables consume one byte per character. Because string variables can be of any length, you must always put them last in a file.

CALCULATING MEMORY REQUIREMENTS

We reserve memory for the file with the command CLEAR n, m where n is the length of the character file needed and m is the total length of the file. The file defaults to a 200 byte area size and a RAM file storage size of 256 bytes. You need only use CLEAR if the file space needed will exceed the default values.

Next DEFFIL must be used to set up the length of each element of the file, and the position within the overall RAM file this particular file will occupy. DEFFIL takes the form of DEFFIL n, m where n is the length of an individual file and m is the address of the start of this file within the overall RAM file area.

SEQUENTIAL FILES

Whereas RAM files are stored within the computer and the contents can be accessed in any order, sequential files must be (as their name indicates) accessed in sequence. The files are put onto tape and the tape must be read through in the order in which the information was laid down. Sequential files can hold much more data than can RAM files, which offsets their lack of convenience.

To use a sequential file you need to (a) open the file; (b) put data into it using INPUT #, or get it out with PRINT #; and (c) close the file when you've finished with it.

LOGIN

The LOGIN command is used to direct the attention of the computer to the relevant file, from the five areas into which the BASIC memory space is divided, so that you can use commands like NEW and SAVE. LOGIN must be followed by an integer between 1 and 5. Using LOGIN clears all variables.

LOAD?"CASx.mmmm"

You can use LOAD? to see whether or not a data file specified by the "CASx:mmmm" has been recorded correctly.

MERGE

This is used to merge a file (which must have been

saved in ASCII format) into the program in the current memory area.

EOF

This signifies end of file, and must be followed by the relevant file number in parentheses.

LOF

LOF is followed by a number in parentheses, and returns the size of the file specified by the number. The relevant file must have been previously opened in the input mode on the HX20.

The function LOF can also return the remaining length of the file if the relevant file is in a ROM cartridge, and the number of data stored in the buffer if the file is in the RS-232C port.

FILES

This is used to get a directory of the names of files within the relevant memory. If you do not follow the word FILES with a device name within quote marks, it will give you all files within peripheral devices currently connected to your HX20. If nothing is connected, the computer will search the microcassette.

The command will display the name of a file when it is found, the type classification (where 0 means a BASIC program file, 1 means a data file and 2 indicates a machine language program file) and the recording format (A for ASCII, B for binary).

SAVE

This command can be used in five different ways, all of them concerned, as the name suggests, with moving a program or contents of a file from a volatile memory area to a more permanent location.

SAVE "mmmm"

This saves a program called mmmm onto tape. You get the program back by LOAD "mmmm".

SAVE "mmmm",A or SAVE "mmmm",V

This is much the same as the above version of SAVE, except that the file is saved in ASCII format. This takes up more room than the binary format which is otherwise used, but allows the file to be accessed later on. A file saved in the ASCII format may be used as a data file. If you use a V, instead of an A, then the microcassette will automatically rewind, and verify the program just saved.

SAVE "CASx:nnn"

This saves BASIC programs on the file specified by CASx. The command can be followed by a comma, and an A or a V. If you use an A, the file will be saved in ASCII format (the default condition is to save the program in compressed binary format, which takes less space, but does not allow certain kinds of access to the file, such as use of MERGE). You can read an ASCII-saved file as a data file. The V triggers an automatic verifying of the program just saved, if the program has been saved on a micro-cassette recorder.

SAVE "COM0:uvxyz",A

This variant of the SAVE command on the HX20 specifies the interface conditions of the RS-232C port with the 'uvxyz'. The command sends the program out via the port. The close quotes must be followed by an A, or the information will not be sent out in ASCII format and will therefore be useless. The five-character statement controls the following:

u - the bit rate which must be a number from 0 to 6 which specify the following bit rates: 0 - 110 bps; 1 - 150 bps; 2 - 300 bps; 3 - 600 bps; 4 - 1200 bps; 5 - 2400 bps; 6 - 4800 bps

v - the word length of one character data which must be either 7 for 7 bits/character or 8 for 8 bits/character

x - this is use to specify the method of parity check and can be either N (no parity check), E (even parity check) or O (odd parity check)

y - this is either 1 or 2 to specify the stop bit length

z - this determines the active control lines and needs four conditions to be specified, using HEX digits (full details on these are given in your manual)

The default condition (on warm start) of uvxyz is as follows:

u - 4800 bps
v - 8 bits/character
x - no parity check

y - stop bit length of 2
z - CTS, ignore; DSR, active; RTS; + potential active; CD, ignore

---

OPENCOMO

This is used to open a specified file. It is followed by four parameters as described above.

SAVEM "CASx:nnn",&abcde,&mnopq,&hijkl

This is used to save a machine code program, or specified memory contents, on a specified file. The first part of the command specifies the device and file name, with the first hex and second hex numbers (&abcde and &mnopq) being the top and bottom addresses of the memory area to be saved, and the third number being the starting address for execution of the machine code program. If the data you are saving is not a machine code program, then make the third address the same as the first one. You cannot leave it out.

TITLE

The TITLE command is one key to using your HX20 most effectively. It allows you to hold up to five complete programs 'permanently' within the computer, where they will be safe from NEW. The programs protected in this way are available directly from the menu. Each TITLEd program is added to the menu

after the standard items 1 (MONITOR) and 2 (BASIC) numbered in the order in which they were TITLEd. The programs held in this area can be edited independently.

Try the following to test TITLE. NEW the computer, then enter the following:

```
10 PRINT "RUNNING FROM M
ENU"
20 SOUND RND(1)*56,1
30 PRINT:PRINT
40 GOTO 10
```

Run it for a few seconds so you are familiar with it in action. Once you've seen it working, stop the program with BREAK. Now type in the following:

```
TITLE "TEST"
```

You can put anything you like (up to eight characters long within the quote marks. Now, press RETURN, then MENU to get back to the initial display. As well as the normal 1 MONITOR and 2 BASIC, you'll see 3, followed by the word you used to follow the TITLE command, like this:

```
1 MONITOR
2 BASIC
3 TEST
```

Press the 3 key, and the program will start running automatically. TITLEd programs are, as I said earlier, protected from NEW. You can easily prove

this. Enter NEW and you'll get this message:

PP Error

PP means 'protected program'. Entering TITLE "" will delete your titled program.

STAT

STAT (status) is related to TITLE. Type in STAT 1 and you'll get a status report of the program area where you have just put a TITLEd program. This command will return the name of the program, and its size. Notice that although the first titled program comes up as 3 on the startup menu, it is referred to with STAT as 1, because it is the first program stored under TITLE.

PROGRAMMING THE FUNCTION KEYS

There are five programmable function keys on the HX20. When you turn the computer on, as your manual explains, the five keys are programmed as follows (where ^M equals pressing the RETURN key after entering the word):

1 - AUTO
2 - LIST^M
3 - LLIST^M
4 - STAT
5 - RUN^M

Holding down SHIFT will change the keys to the following:

6 (that is, SHIFT and key 1) - ?DATE$:?TIME$^M
7 - LOAD

8 - SAVE
9 - TITLE
10 - LOGIN

It is very easy to change them to suit your own needs. When you turn the computer on (warm start) key 3, for example, is set to LLIST^M as indicated above. This means that touching key 3 is the same as typing in LLIST then pressing RETURN.

If you wanted key 3 to make a beep sound, such as would be produced by SOUND 2,2 then you would type in (either in direct mode, or as part of a program if you wanted the key to have a specific role within that program):

```
KEY 3,"SOUND 2,2" + CHR$(13)
```

Try it now. Type in the above (where '+ CHR$(13)' is the way you get a ^M onto the key). Now press key 3 and the tone will sound, just as you directed.

To prove it is there permanently, turn your HX20 off, then on again, and type in KEY LIST (or KEY LLIST) which should produce this result:

```
PF1  AUTO
PF2  LIST^M
PF3  SOUND 2,2^M
PF4  STAT
PF5  RUN^M
PF6  ?DATE$:?TIME$^M
PF7  LOAD
PF8  SAVE
PF9  TITLE
PF10 LOGIN
```

CHAPTER THIRTEEN - HANDLING THE GRAPHICS SCREEN (inc. SCREEN, PSET, PRESET, POINT, LINE)

The word SCREEN determines whether we wish to use the built-in screen, or the external one, for text or graphics. The LCD screen can display both graphics and text at once, although the external display can only show one or the other at a time.

When you turn the computer on (and you do not press CTRL/@ for a cold start) the computer's built-in screen will automatically be set for both text and graphics.

There are four pairs of numbers which follow SCREEN. The numbers, and their effect, are as follows:

COMMAND FORM:	RESULT:
SCREEN 1,0	External display - text LCD - graphic
SCREEN 0,2	LCD - text External - graphic in high-resolution mode
SCREEN 0,1	LCD - text External - graphics in color
SCREEN 0,0	Both text and graphics on LCD

The following program puts a picture on the built-in graphics screen first, then adds text. Note that line 10 is not needed when the computer is turned on

with a warm start.

This program also shows the PSET command in use. PSET is used to print a single dot onto the screen. The built-in screen is 120 points across and 32 points down, so you can PSET points within these limits. Note that the horizontal co-ordinate comes first, with the vertical one second.

```
10 SCREEN 0,0
20 CLS
30 FOR X=10 TO 30
40 FOR Y=5 TO 25
50 IF RND(1)>0.5 THEN PS
ET(X,Y)
60 NEXT Y
70 NEXT X
80 PRINT "THIS"
90 PRINT "IS A"
100 PRINT "TEST ";
110 GOTO 30
```

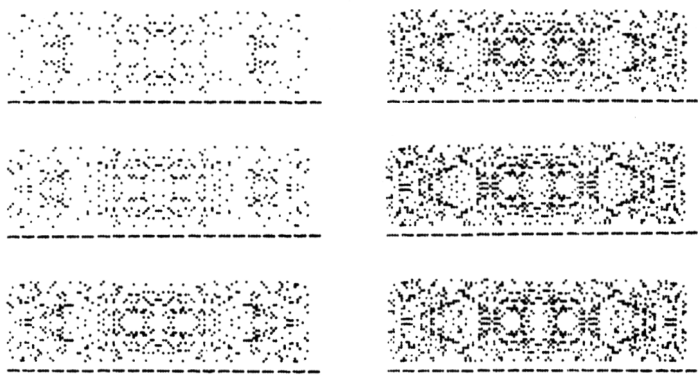
Here's a program which plots something a little more interesting than a random series of dots:

```
10 REM SUMMER WINE
20 SCREEN 0,0
30 CLS
40 RANDOMIZE VAL(RIGHT$(
TIME$,2))
50 A=INT(RND(1)*120)
60 B=INT(RND(1)*32)
70 PSET (A,B)
80 PSET (120-A,B)
90 PSET (120-A,32-B)
100 PSET (A,32-B)
110 GOTO 50
```

The quality of the output has to be seen to be believed. Add the following line if you want a permanent copy of your unfolding designs:

```
105 IF RND(1)>0.98 THEN
LPRINT:COPY:LPRINT"-----
----------------"
```

Here's part of a series of designs produced by this program:



Obviously, if you run this program long enough, the entire screen will turn black. The command PRESET is the opposite of PSET; it unplots a dot rather than plotting it. Here's a second program, Autumn Mist, which uses PSET and PRESET to create a constantly evolving design:

```
10 REM AUTUMN MIST
20 SCREEN 0,0
30 CLS
40 RANDOMIZE VAL(RIGHT$(
TIME$,2))
50 A=INT(RND(1)*120)
60 B=INT(RND(1)*32)
70 IF RND(1)>0.4 THEN GO
TO 130
80 PSET (A,B)
90 PSET (120-A,B)
```

```
100 PSET (120-A,32-B)
110 PSET (A,32-B)
120 GOTO 50
130 PRESET (A,B)
140 PRESET (A,32-B)
150 PRESET (120-A,B)
160 PRESET (120-A,32-B)
170 SOUND RND(1)*56,1
180 GOTO 50
```

As you'll hear, this program also has a sound routine (line 170) which produces random beeps as the program runs (one beep for every four dots erased). If you're running the program on the LCD screen, PRESET erases a dot from that screen. On an external display, PRESET resets the position to the background color.

Note that PSET and PRESET co-ordinates are different on the external display from the LCD co-ordinates. The range of co-ordinates on the LCD is 120 (horizontal) by 32 (vertical). The external display has, in the four color mode, a resolution of 128 (horizontal) by 64. The vertical range increases to 96 when the external display is operating in the high resolution mode. Note also that when operating in the four color mode, you continue to specify the co-ordinates as though you were working in the high resolution mode. The dots are not mapped one to one with the co-ordinates. The relationship between the designated co-ordinate and the actual point plotted is given by the following: Vertical position equals the integer value of two times the vertical co-ordinate divided by three.

POINT is used to determine the presence, or otherwise, of a plotted point at a specific position. The next program uses PSET to place some dots randomly on the screen, then uses POINT to see

how many of them can be found. When you run this, you'll hear a beep every time a dot is found.

```
10 CLS
20 RANDOMIZE VAL(RIGHT$(
TIME$,2))
30 FOR A=1 TO 300
40 PSET (RND(1)*120,RND(
1)*32)
50 NEXT A
60 IF POINT (RND(1)*120,
RND(1)*32)=1 THEN SOUND
RND(1)*30,1
70 GOTO 60
```

LINE is used to draw a line between two points. It can produce quite interesting displays, as this program illustrates:

```
10 REM RUSTONIAN
       TRIANGLES
20 CLS
30 L=RND(1)*120
40 M=RND(1)*32
50 N=RND(1)*120
60 O=RND(1)*32
70 P=RND(1)*120
80 P=RND(1)*32
90 A=RND(1)*3
100 B=RND(1)*3
110 C=RND(1)*3
120 D=RND(1)*3
130 E=RND(1)*3
140 F=RND(1)*3
150 LINE (L,M)-(N-L,O-M)
,PSET
160 LINE (L,M)-(P-N,Q-O)
,PSET
170 LINE (N-L,O-M)-(L-P,
Q-O),PSET
180 FOR X=1 TO 100:NEXT
190 IF L+A>31 OR L+A<0 T
HEN A=-A
```

```
200 IF M+B>31 OR M+B<0 T
HEN B=-B
210 IF N+C>31 OR N+C<0 T
HEN C=-C
220 IF Q+D>31 OR Q+D<0 T
HEN D=-D
230 IF P+E>31 OR P+E<0 T
HEN E=-E
240 IF Q+F>31 OR Q+F<0 T
HEN F=-F
250 L=L+A:M=M+B
260 N=N+C:O=O+D
270 P=P+E:Q=Q+F
280 CLS
290 GOTO 150
```

COLOR

This determines which colors will be shown on the external display. The command follows the format COLOR a, b, c where a, b and c must be in the range 0 to 3. Although there are eight colors which can be produced, the computer can only generate four at once. In the statement COLOR a, b, c the a specifies the foreground color, the b the background and c the color set. There are two color sets, each of which can produce four colors.

In color set 0 (i.e. when c is 0), the following colors can be produced:

0 - green
1 - yellow
2 - blue
3 - red

In color set 1, you can get these colors:

0 - white
1 - cyan
2 - magenta
3 - orange

The HX20 defaults (warm start) to color set 0, with a foreground color of 1 and background color of 0. This means you will get yellow writing on a green background.

MORE COMMANDS

LOCATE

This is a very useful command, which moves the cursor to the position of your choice on the virtual screen. The word LOCATE is followed by two numbers, separated by a comma. The first number is the horizontal co-ordinate of the cursor, while the second is the vertical co-ordinate. A third parameter may follow the vertical co-ordinate. If this third number is 0, it turns off the cursor, while a 1 will turn the cursor on.

CSRLIN

This returns the vertical position of the cursor on the virtual screen.

LOCATES

LOCATES is used to specify the position of the physical screen, moving it so that its upper left hand corner is in the location on the virtual screen specified by the two numbers which follow the word LOCATES. It is used in the form LOCATES n,m where n is the horizontal co-ordinate, and m the vertical one.

SCROLL

This command controls the rate and direction of scrolling. It takes the form:

SCROLL n,m,p,q

In this line, n controls the speed with which the LCD screen scrolls (the external display scroll speed is fixed).

The next one, m, must be in the range 0 to 9, with the higher digits indicating higher speeds. The second parameter, m, is the scroll mode. It must be 0 or 1, with 0 locking the screen at the left hand end of a line. Mode 0 will prevent the screen from moving 'sideways', even if the cursor moves far to the right, going off the screen. Mode 1 instructs the physical screen to follow the cursor.

The third parameter, p, is the scroll step size in characters. This is used for moving the screen right or left, when the CTRL key and one of the arrowed keys (above the RETURN key on the right hand side of the keyboard) is pressed. It must lie between 1 and 20 for the LCD and between 1 and 32 for an external display.

The final parameter, q, controls the number of lines which will be scrolled up or down when CTRL and P or CTRL and Q are pressed at once, or when the key marked SCRN (second from the right, on the top row) is pressed. The value for q must lie in the range 1 to 4 for the LCD and from 1 to 16 for an external display.

When you turn the computer on (warm start), it will be set as follows:

SCROLL 9,0,10,4 (LCD)
SCROLL 9,0,16,16 (external screen)

WIDTH

This command can be used in two ways. We will look at each in turn.

WIDTH "LPT0:",n

The LPT0 means the width of the output to the printer. The digit n must be within the range 1 to 255. The computer usually outputs 24 characters to the printer. If n is set to 255, the HX20 interprets this as a command not to send any automatic line feed commands.

WIDTH "COM0:",n

This controls the action of a printer connected to the RS-232C port. When you first turn the computer one (warm start), n is set to 80.

Try the following program:

```
10 REM PRINTER WIDTH DEM
O
20 INPUT "ENTER A NUMBER
 <24";A
30 IF A<1 OR A>24 THEN 2
0
40 WIDTH "LPT0:",A
50 LLIST
60 RUN
```

Here's what it produces when a value of 20 is entered for A in line 20:

```
10 REM PRINTER WIDTH
 DEMO
20 INPUT "ENTER A NU
MBER <24";A
30 IF A<1 OR A>24 TH
EN 20
40 WIDTH "LPT0:",A
50 LLIST
```

POS

This function tells you where the cursor (or printer head) is across a line. POS is used in a line by setting a variable equal to POS, where the word POS is followed by a number from 0 to 16, in parentheses.

If you use the statement A = POS (0) you'll get a number indicating the position (in number of character widths) of the cursor across the virtual screen, or of the print head across the paper. When the zero is replaced by any number between 1 and 16, you're told the number of (a) characters printed in the current line, plus the number of characters stored in the buffer which have not yet been printed.

CHAPTER FOURTEEN - SOME HANDY EXTRAS

We have come a long way in the short time we've been working with the Epson. If you've been saving the programs as you go along, you'll have started to build up quite a worthwhile library. Now is the time to start looking at computer magazines for programs written for other machines which you can adapt to your HX20. You'll be pleased to see how easily most of them can be converted to run on your computer once you alter the program's output to fit the HX20's screen. In this chapter, we'll be looking at some extra commands which you should find useful.

AUTO

This command is a handy one when you're writing programs, or entering them direct from a listing. It generates line numbers automatically, producing a new one each time you press ENTER. If you just enter AUTO, and then press ENTER, the Epson will assume you want to program in steps of 10, with the first line numbered at 10. Try it now, by typing in AUTO, then pressing ENTER. Follow this with a program line, press ENTER again, and you'll see 20 generated for your second line.

If you do not want to start with 10, and/or you do not want your line numbers to increment in tens, you need to follow the command with one, or two numbers. AUTO 25, 36 will start numbered at line 25 and increment the line numbers by 36 (so the line number sequence will be 25, 61, 97, 133 ...and so on). If you do not specify a second number (so your command is AUTO 20,) but the first number is followed by a comma, the last value used for the increment will be

used in this case. If the command does not have either a second number, nor a comma (AUTO 35) then an increment of 10 will be provided, starting the first line number at 35.

You can get out of the AUTO mode by pressing BREAK.

If you already have some lines written in a program before you go into AUTO, you may - from time to time - discover that the automatically generated line number is the same as a number already in the program. In this case, an asterisk will be printed after the line number. If you want to overwrite your original line, keep programming as normal. If you wish to keep your original line, press RETURN. The original line will remain in the program, and the next line number in the sequence will be generated.

FRE

This is used to tell you how much memory you have left. PRINT FRE(n), where n is any number, will return the number of free bytes. Take note, however, that programs need working space, and that arrays consume memory. Run the next program to show the 'memory-eating' effects of dimensioning an array:

```
10 PRINT FRE(1)
20 DIM A(2000)
30 PRINT FRE(1)
```

If the FRE is followed by a string, the command returns the memory left in the BASIC string area, as the following program shows:

```
10 PRINT FRE(1)
20 PRINT FRE("WORD")
30 PRINT FRE(A$)
```

SOUND

You've probably picked up how the SOUND command works by now. It is pretty simple. The command SOUND is followed by two numbers. The first defines the pitch of the note to be made, and the second its duration.

The pitch number may be from zero (a 'non-note' or pause) to 56 (the highest note). The duration can be from zero to 255, where each number represents one tenth of a second (so 2 represents a duration of 2/10ths of a second, and 10 equals one second).

If you run the next program, you'll hear what appears to be the same sequence of notes played twice, even though we only go through the loop (1 to 56) once:

```
10 FOR A=1 TO 56
20 SOUND A,1
30 NEXT
```

It sounds as if the sequence has been played twice because, for reasons best known to themselves, the designers of the Epson decided that a series of notes would be produced by the numbers 1 to 28, and then the notes which are half a tone higher than these would be represented by 29 through 56. That is, note 29 is just half a tone higher than note 1, while note 2 is half a tone higher than 29. If you find this confusing, you may prefer just to work with the notes from 1 to 28.

Here's a very short program which can be used at any time when you wish to demonstrate your computer. It

produces a cheerful, randomly-composed piece of 'electronic music':

```
10 SOUND RND(1)*56,RND(1
)*3
20 GOTO 10
```

If you want to use your computer as a kind of piano, enter and run the following program. The bottom row of keys is used as the piano keyboard:

```
10 A$=INKEY$
20 IF A$<>"Z" AND A$<>"X
" AND A$<>"C" AND A$<>"V
" AND A$<>"B" AND A$<>"N
" AND A$<>"N" AND A$<>"M
" AND A$<>"," THEN 10
30 IF A$="Z" THEN SOUND
1,1
40 IF A$="X" THEN SOUND
2,1
50 IF A$="C" THEN SOUND
3,1
60 IF A$="V" THEN SOUND
4,1
70 IF A$="B" THEN SOUND
5,1
80 IF A$="N" THEN SOUND
6,1
90 IF A$="M" THEN SOUND
7,1
100 IF A$="," THEN SOUND
 8,1
110 PRINT "♪♪",
120 GOTO 10
```

Here is the 'music' to play a well-known nursery rhyme:

```
ZZ BB NN B
VV CC XX Z
BB VV CC X
BB VV CC X
ZZ BB NN B
VV CC XX Z
```

SWAP

This command is used to exchange values between two variables. That is, if A equals 10 and B equals 99, SWAP A,B would make A equal 99 and B equal 10, as this short program illustrates:

```
10 A=INT(RND(1)*10)
20 B=INT(RND(1)*10)
30 IF A=B THEN 20
40 PRINT "BEFORE SWAP:"
50 PRINT "A=";A;
60 PRINT "  B=";B
70 PRINT "STAND BY FOR S
WAP"
80 FOR T=1 TO 500:NEXT
90 SWAP A,B
100 GOTO 50
```

SWAP can also be used on strings, as you can see from this routine:

```
10 A$=CHR$(INT(RND(1)*26
)+65)
20 B$=CHR$(INT(RND(1)*26
)+65)
30 IF A$=B$ THEN 20
40 PRINT "BEFORE SWAP:"
50 PRINT "A$=";A$;
60 PRINT "  B$=";B$
70 PRINT "STAND BY FOR S
WAP"
80 FOR T=1 TO 500:NEXT
90 SWAP A$,B$
100 GOTO 50
```

ATN

This function returns the arctangent and is used in the following form: A = ATN(n). It returns an answer in radians and works within the range minus pi/two to pi/two.

COS

This function returns the cosine and is used as follows: C = COS(n). It returns the answer in radians.

SIN

This function is used to get the sine of an angle, and is used in the following form: S = SIN(n). This also returns the answer in radians.

TAN

The tangent of a number of returned by this function, used as follows: T = TAN(n). Again, the function returns an answer in radians.

MOD

This function returns the remainder of a division. for example, 24 MOD 5 will return 4, because 4 is the remainder when 24 is divided by 5. The form of the command is n MOD m and the function returns the remainder when n is divided by m.

This program shows a few examples of it in use:

```
10 INPUT "FIRST NUMBER";
N
20 INPUT "TO BE DIVIDED
BY";M
30 PRINT N;"/";M;" IS";N
/M
40 PRINT N;"MOD";M;"IS";
N MOD M
50 FOR J=1 TO 500:NEXT
60 RUN
 876 / 9  IS 97.3333
 876 MOD 9 IS 3
 22 / 7  IS 3.14286
 22 MOD 7 IS 1
```

HEX$ and OCT$

HEX$ is used to change numbers from decimal form into hexadecimal form (that is, to base 16). The number is always given as a string, as follows: A$ = HEX$(nnnnnn). This short routine shows it in action:

```
10 REM HEX$ demo
20 CLS
30 INPUT "Enter a number
 ";N
40 PRINT "Your number is
 ";N
50 H$ = HEX$(N)
60 PRINT "which is ";H$
70 PRINT "in hexadecimal
"
80 PRINT
90 GOTO 30
```

```
Your number is  12
which is C
in hexadecimal
```

```
Your number is  32676
which is 7FA4
in hexadecimal
```

Note that the number must be in the range - 32768 to 65535 or a function call error will occur.

OCT$ works in the same way as HEX$, except that it returns a number to base eight, rather than to base 16.

LOG

This function returns the natural logarithm of a number, and is used in the form A = LOG(n).

TRON/TROFF

This pair of commands is used to turn a program tracer off and on. TRON is, naturally enough, TRacer ON, and TROFF is TRacer OFF. When you put the computer in the TRON mode (simply by typing TRON, then pressing RETURN, or by including it as a line in your program) and execute a program, you'll find the program will run as normal, exceot that the current line number being executed will be displayed on the screen within square brackets. TRON is designed to help you detect bugs within a program. You'll find it slows program execution down somewhat, so only use it in the debugging stage.

TROFF turns the tracer off. It can be entered directly or may be included within the program. NEW — or turning the computer off — will also disable TRON. Next time you have a program in your HX20, enter TRON to see the tracer in action.

DAY

You can set your computer to hold the day of the week as a number from 1 to 7. Number 1 can be set to any day which you find convenient so it can be used to trigger real time reminders within a diary program.

In such a program, it is easy to work out what number the current day is, counting from the day you have designated to be 1. For example, if you wanted Monday to be DAY 1, and today was Wednesday, you'd enter DAY = 3.

This is very useful for a 'memory tickler' program which you can store, and amend, under a TITLE. Then, you can just trigger this diary, and it will automatically print out on the printer, or the screen, what you need to do that day.

Here is the start of such a program:

```
10 REM REAL TIME DIARY
20 ON DAY GOSUB 1000,200
0,3000,4000,5000,6000,70
00
30 END
1000 REM MONDAY (MONDAY
= 1)
1010 LPRINT "CHECK LAST
WEEK'S SALE FIGURES"
1020 LPRINT "PHONE HQ FO
R THIS WEEK'S TARGET"
1030 LPRINT "AMEND PRINT
 SCHEDULES ACCORDINGLY"
1999 RETURN
2000 REM TUESDAY
2010 LPRINT "MEETING WIT
H ADMIN STAFF AT 10 AM"
```

```
2020 LPRINT "CHECK ON PR
OGRESS RE JOHNSON ACCOUN
T"
2030 LPRINT "CHECK RE DI
RECTORS MEETING, THURSDA
Y"
2999 RETURN
3000 REM WEDNESDAY
3010 REM...AND SO ON....
.
3020 REM .......
```

I happen to be writing this on a Tuesday. I set DAY (with Monday equal to one) when I first bought the HX20, so — if all is well — it should still be current. I can test this by entering PRINT DAY, and I get a return of 2. The DAY counter is working. When I trigger RUN, the computer prints out the following:

```
MEETING WITH ADMIN STAFF
 AT 10 AM
CHECK ON PROGRESS RE JOH
NSON ACCOUNT
CHECK RE DIRECTORS MEETI
NG, THURSDAY
```

As you can see from the listing, there are 999 possible storage spaces for each day, more than you are ever likely to need. You can easily add extra information for a particular day (you can find the current highest line number being used for, say, Wednesday, by entering LIST - 3999 and place your next item one more than the line number which was used before 3999) or delete it once the need for the item has moved.

DEF FN

This very useful programming tool enables you to define a function which can then be used throughout your program.

Suppose the program demanded the production of random numbers in various ranges from time to time during execution. You could create a function, called R, to produce random numbers in the range 1 to X, as follows:

```
10 DEF FN R(Z)= INT(RND(
1)*X + 1)
20 INPUT "ENTER A NUMBER
";X
30 Z=FN R(X)
40 PRINT Z
50 GOTO 20
```

The function may contain one (as above) or more variables. The following forms are valid:

```
DEF FN A (P,Q) = P*2 + SQR (Q)
DEF FN A (Z$) = INT (LEN(Z$)/2)
DEF FN A$(Z$) = RIGHT$ (Z$,2)
```

The DEF FN line must be passed over in program execution before a function call is made, as is demonstrated by the next program, which is a variation of the previous one to generate random numbers:

```
10 GOSUB 60
20 INPUT "ENTER A NUMBER
";X
30 Z=FN R(X)
40 PRINT Z
50 GOTO 20
60 DEF FN R(Z)= INT(RND(
1)*X + 1)
70 RETURN
```

DEFINT, DEFSGN, DEFDBL, DEFSTR

The DEF command, followed by INT, SGN, DBL or STR as above, allows you to define the kind of variables you want in a program. DEFINT A - Z is used to ensure that all variables used within the program are integer variables. This ensures that the program runs as quickly as possible. The computer handles integer variables more quickly than it does 'floating point' variables.

DEFSNG determines that variables will be single precision, that is, accurate to eight and a half decimal places. DEFDBL ensures the variables are double precision, that is they will hold numbers correct to fifteen and a half decimal places. The computer defaults to single precision.

DEFSTR declares that the variable names beginning with the letters which follow the DEFSTR will be string, rather than numeric, variables. Such variables do not have to be followed by a dollar sign, but must be treated within the program as if they did. This means, for example, that if you had declared A as a string variable, you could not include A/2 later in the program without generating a TM (type mismatch) error.

The range of letters which is covered by the declaration is determined by the letters following the DEF... If, as in CHESS and CHECKERS, we use DEFINT A - Z, it ensures that all variables will be integer variables. If you just want variable names beginning with a single letter to be variables of a

specific type, follow the declaration with that letter:

```
DEFDBL Q
```

For a range of letters, separate the two letters which lie at the extremes of the desired range with a hyphen:

```
DEFINT P - R
```

If you want to have two or more variable 'groups', separate the hyphenated pairs with commas:

```
DEFDBL A - C, P - R, X - Z
```

The computer treats all variables as single precision unless otherwise intructed.

```
10 PRINT 8/7
20 DEFSNG A - Z
30 Z= 8/7
40 PRINT Z
50 DEFDBL A - Z
60 Z=8/7
70 PRINT Z

 1.14286
 1.14286
 1.142857193946838
```

```
10 DEFSTR A-Z
20 A="SAUSAGE"
30 PRINT A
```

PRINT USING

The variations of PRINT USING enable you to control the kind of PRINT (and LPRINT) output produced by your HX20.

The general form of the command is PRINT USING "####";A,B, where each # represents a digit position. Each of these must be filled when a number is printed. If there are not enough digits, spaces will take the place of the missing digits.

When dealing with string fields, the following characters effect the final output: ! \ and &. With numeric fields, the characters are #, . (decimal point), +, -, **, $$, **$, ,(comma), ^^^^, _ and %.

The easiest way to understand them in action is to study the output of the following program. Run it several times, entering different strings, and numbers, until you understand how you can control the output so it is exactly as you want it to be:

```
10 REM Demo of PRINT USI
NG
20 INPUT "Enter a strin
g ";Z$
30 INPUT "and now a numb
er ";Z
```

```
40 ZN = - Z
50 CLS
60 PRINT "Your string is
 ";Z$
70 PRINT "Your number is
 ";Z
80 PRINT "!    ";
90 PRINT USING "!";Z$
100 PRINT "\    ";
110 PRINT USING "\    \";
Z$
120 PRINT "&    ";
130 PRINT USING "&";Z$
140 PRINT "#   ";
150 PRINT USING "#";Z
160 PRINT "##   ";
170 PRINT USING "##";Z
180 PRINT "###    ";
190 PRINT USING "###";Z
200 PRINT "#.##    ";
210 PRINT USING "#.##";Z
220 PRINT "+###.###    ";
230 PRINT USING "+###.##
#";Z
240 PRINT "##.##-     ";
250 PRINT USING "##.##-"
;ZN
260 PRINT "**#.##     ";
```

```
270 PRINT USING "**#.##"
;Z
280 PRINT "$$#.##     ";
290 PRINT USING "$$#.##"
;Z
300 PRINT "**$.##     ";
310 PRINT USING "**$.##"
;Z
320 PRINT "$$#._-     ";
330 PRINT USING "$$#._-"
;Z
```

Here's one result of running that program on the HX20:

```
Your string is HARTNELL
Your number is  1234.57
!   H
\   HARTN
&   HARTNELL
#  %1235
##  %1235
###   %1235
#.##   %1234.57
+###.###   %+1234.570
##.##-     %1234.57-
**#.##     %1234.57
$$#.##     %$1234.57
**$.##     %$1234.57
$$#._-     %$1235.-
```

Now that you've seen PRINT USING in action, you'll probably be able to think of a number of ways it can be used to improve the output of your programs. As an exercise, you could try making use of PRINT USING within the programs in the business chapter (chapter 12, starting on page 164).

USR

USR is used to call a machine language routine which has been previously defined by the DEFUSR statement. The letters USR are followed by a digit in the range 0 to 9 (referring to one of the maximum of ten user-defined functions, and assuming USR 0 if no digit is specified) and then an argument (by which you can transfer a value from BASIC to your machine language subroutine).

MEMSET

MEMSET is followed by an address in hex which is used to specify the lower limit of memory for use by the BASIC program, and to set the memory locations for your machine language programs. The value of MEMSET defaults to &H0A3F.

EXEC

This command is used to start the execution of a machine language subroutine. It is followed by the start address of the subroutine.

MOTOR

You use this to turn an external cassette recorder on and off. If the word MOTOR is not followed by either the words ON or OFF, the command will reverse the current state of the recorder. That is, if the cassette recorder is currently on, MOTOR will turn it off. MOTOR can be used within a program.

CDBL

This changes integers, and single precision numbers into double precision numbers, but without changing the number of significant digits. A hex number must be prefixed by &H, an octal number by &.

CINT

CINT, followed by a numeric expression in parentheses, as CINT(4/2*n), converts that expression, if it would otherwise be a single or double precision number, into an integer. The value of the numeric expression must between -32768 and 32767.

CSNG

CSNG, as you've probably guessed from the above, is used to change integers and double precision numbers into single precision ones. The conversion is done to six significant figures.

FIX

FIX returns the truncated integer part of a numeric expression which follows the word FIX.

ON ERROR GOTO

This command is followed by a line number, to which the program will branch, if there is an error in the program rather than stopping and displaying an error message. You can disable the error-trapping

completely by including the line ON ERROR GOTO 0 at the beginning of your program.

The variables ERL and ERR are used in connection with returning error messages. ERL stores the line number where the error occurred, and ERR stored the error code.

RESUME

RESUME is used after the program has stopped because of some error. If RESUME is used by itself, program execution will continue on from the line where the error occurred. RESUME NEXT will start at the line after this one. As an alternative to these, you can following RESUME by a line number, and execution will start from this number.

VARPTR

This is followed by the name of a variable in parentheses and returns the address of the variable or the array which is specified by the name.

PCOPY

This command allows you to copy a program from the workspace into a specified program area. The command is used in the form PCOPY n where n refers to the target program area and is a number from 1 to 5.

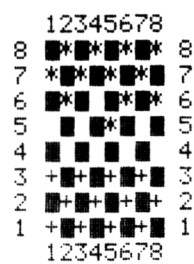
The HX20 will refuse to accept the command if you have no program in the workspace or if there is a program in the target area (either of these conditions will give rise to an FC - illegal function call - report), or if the target area is not big enough to hold the initial program (when you'll get an OM - out of memory - report). PCOPY will not work in any of these conditions.

CHAPTER FIFTEEN - PLAYING GAMES

We will take a break now from the serious side of the computer, and have a look at a game of Checkers. you'll find that the computer plays a fairly swift (if not exactly inspired) game, following the standard rules.

I suggest you use an external board, such as the one printed earlier in this book for use with Reversi, and make both your moves, and those of the computer, on the board. It is rather difficult to see the computer's moves, and to work out the best moves of your own, relying only on the built-in screen.

If, however, you decide to depend on the internal board (as I do when playing this game against the HX20 on board a plane), you'll find the program can help you in two ways. Firstly, you can get the HX20 to reprint the board at any time, by entering 0 when asked for your move. You can also get the computer to dump the current state of the game onto the printer. The output of the printer looks like this:



You get the computer to dump the current board to

the printer by entering "1" when asked for your move. If you wish to set up a board of your choice, perhaps giving the computer an advantage at the start of a game, or to give yourself an advantage, you enter "2" when asked for your move.

Your moves are entered as two numbers, a 'from' number and a 'to' number. You work out the number of the piece you want to move by entering the number down the side, then the number across the top of the board which refers to the square on which your piece is resting. You do in response to the prompt "FROM?". Having done this, the computer will prompt "TO?". You then enter the square you're moving to. If you're capturing, the computer will make the move, then print up the prompt "AGAIN (Y, N)?". You enter "Y" if you can jump again, and you'll be asked "TO?". The computer, of course, knows which square you're moving from.

It makes it multiple jumps automatically, reprinting the board after each move. You'll find it quite fascinating to watch the computer in action, as it prints on the screen all the moves it is considering. If it finds a move which it considers is quite a good one, it will make a beeping sound, and put the number of that move (it counts the good moves as it finds them) on the screen. Then, it will either capture if it can, or make one of the good moves.

The hierarchy of moves considered by this program are:

- can a capture be made?
- can an ordinary piece be transformed into a king

- can a move be made which does not place the moved piece in danger
- can a legal move be made

The computer scans the board, square by square, starting from the top left hand corner. If it finds a possible capture, it makes the move, not considering any alternatives. If it discovers a move which it can make which does not expose the moved piece to danger, it will store that move in the 'safe move repository' (which we'll discuss shortly when going through the program).

If no capture is found, the computer checks to see if it can promote a piece to king. If it finds it can, it makes this move. If such a move cannot be made, the computer returns to the 'safe move repository' and chooses a move at random from the stored moves. This ensures that the computer will not play the same game twice in a row. If no safe move is found, the computer chooses up to 300 moves at random, looking for a legal one. If it finds such a move, it makes it. If not, it concedes the game. The contest continues until either you, or the computer, manage to capture all of the opposing player's pieces.

Once you've entered your program into the computer, and played a few games, return to the book for a discussion on the contents of the listing.

```
10 REM CHECKERS
20 CLS:PRINT"Press space
 bar"
30 PRINT "to begin":N=0
40 N=N+1:IF INKEY$="" TH
EN 40
```

```
50 RANDOMIZE N:SOUND 5,5
60 IF N>59 THEN N=N/2:GO
TO60
70 SOUND N,4
80 GOSUB 1070
90 REM DELETE NEXT LINE
TO PREVENT HX20 FROM
HAVING FIRST MOVE
100 GOTO 130
110 GOSUB 680
120 GOSUB 850
130 GOSUB 680
140 GOSUB 160
150 GOTO 110
160 FOR M=1 TO 10:S(M)=0
:NEXT
170 SC=0:A=89
180 A=A-1
190 IF Q(A)<>C AND Q(A)<
>CK THEN 280
200 B=0:IF A<29 THEN B=2
210 B=B+1
220 M=A+N(B)
230 IF M>88 OR M<11 THEN
 280
240 IF 10*(INT(M/10))<>M
 THEN PRINTA:"to";M:"?"
250 IF (Q(M)=H OR Q(M)=H
K) AND Q(M+N(B))=E THEN
320
260 IF Q(M)=E THEN IF (Q
(M-11)<>H AND Q(M-11)<>H
K) THEN IF (Q(M-9)<>H AN
D Q(M-9)<>HK) AND Q(M+9)
<>HK THEN IF ((Q(M+22)<>
HK OR Q(M+18)<>HK) AND (
Q(M+9)<>C OR Q(M+9)<>CK
OR Q(M+11)=C OR Q(M+11)=
CK)) AND Q(M+11)<>HK THE
N GOSUB 470
270 IF B<2 OR (Q(A)=CK A
ND B<4) THEN 210
280 IF A>11 THEN 180
290 FL=0:IF Q(22)=C OR Q
(24)=C OR Q(26)=C OR Q(2
8)=C THEN GOSUB 1270
```

```
300 IF FL=1 THEN 650
310 GOTO 500
320 Q(M+N(B))=Q(A):Q(M)=
E:Q(A)=E
330 PRINT:PRINT"********
**********"
340 PRINT"From";A;"to";M
+N(B):SOUND 1,5:FOR T=1
TO 500:NEXT
350 FORQ=1TO4:PRINT TAB(
2*Q);"Got you!":FOR T=1
TO 200:NEXT:SOUND 5*Q,.5
:NEXT
360 CO=CO+1
370 GOSUB 680
380 A=M+N(B)
390 B=0
400 B=B+1
410 IF (A+2*N(B)<11 OR A
+2*N(B)>88) AND B<4 THEN
 400
420 M=A+N(B)
430 IF Q(M)=C AND B>3 TH
EN RETURN
440 IF (Q(M)=H OR Q(M)=H
K) AND Q(M+N(B))=E THEN
320
450 IF B<2 OR (Q(A)=CK A
ND B<4) THEN 400
460 RETURN
470 IF SC<10 THEN SC=SC+
1
480 PRINT ,"*";SC:SOUND
3*SC,1
490 S(SC)=100*A+B+20:RET
URN
500 IF SC=0 THEN 550
510 XC=INT(RND(1)*SC)+1
520 A=INT(S(XC)/100)
530 M=A+N(S(XC)-100*A-20
)
540 GOTO 650
550 SC=SC+1:A=INT(RND(1)
*88)+1
```

```
560 IF Q(A)<>C AND Q(A)<
>CK THEN 630
570 B=0
580 B=B+1
590 M=A+N(B)
600 IF M>88 OR M<11 THEN
 620
610 IF Q(M)=E THEN 650
620 IF B<2 OR Q(A)=CK AN
D B<4 THEN 580
630 IF SC<300 THEN 550
640 PRINT :PRINT"I conce
de the game":END
650 Q(M)=Q(A):Q(A)=E
660 PRINT"**************
******":PRINT"From";A;"t
o";M:SOUND7,5:FOR T=1 TO
1000:NEXT
670 RETURN
680 CLS
690 PRINT "HX20>";CO;" Y
OU>";HU
700 SOUND RND(30)+20,0.5
710 PRINT TAB(6);"123456
78"
720 FOR F=80 TO 10 STEP-
10
730 PRINT TAB(3);F/10;
740 FOR G=1 TO 8:PRINT C
HR$(Q(F+G));:NEXT
750 PRINT F/10:NEXT
760 PRINT TAB(6);"123456
78"
770 IF CO=12 OR HU=12 TH
EN 790
780 RETURN
790 IF HU=12 THEN PRINT:
PRINT "You have won":PRI
NT:PRINT"Congratulations
!":END
800 PRINT:PRINT"I win...
":PRINT:PRINT"Thanks for
 the game":END
```

```
810 REM 99 TO CONCEDE
820 REM 0 TO REPRINT
    BOARD
830 REM 1 TO LPRINT
  BOARD
840 REM 2 TO SET UP OWN
BOARD
850 INPUT"FROM";A
860 IF A=99 THEN PRINT "
Thanks for the":PRINT"sa
me...":END
870 IF A=1 THEN GOSUB 13
50:GOTO 850
880 IF A=0 THEN GOSUB 68
0:GOTO 850
890 IF A=2 THEN GOSUB 14
60:GOTO 850
900 INPUT"TO";B
910 Q(B)=Q(A):Q(A)=E
920 FOR T=11 TO 17:IF Q(
T)=C THEN Q(T)=CK
930 NEXT
940 FOR T=82 TO 88:IF Q(
T)=H THEN Q(T)=HK
950 NEXT
960 IF ABS(A-B)<12 THEN
RETURN
970 TY=RND(1)
980 IF TY<.3 THEN PRINT:
PRINT"Good move"
990 IF TY>.7 THEN PRINT:
PRINT"Got me!"
1000 HU=HU+1:Q((A+B)/2)=
E:GOSUB 680
1010 FOR T=82 TO 88:IF Q
(T)=H THEN Q(T)=HK
1020 NEXT
1030 PRINT:INPUT "AGAIN
(Y, N)";A$
1040 IF A$<>"Y" AND A$<>
"y" THEN RETURN
1050 GOSUB 680
1060 A=B:GOTO 860
```

```
1070 CLS
1080 H=43:HK=75:C=42:CK=
79:E=32:B=140
1090 OF=-99:DIM Q(99),N(
4),S(10)
1100 FOR M=1 TO 99:Q(M)=
OF:NEXT
1110 FOR M=1 TO 64
1120 READ D:READ G
1130 Q(D)=G:NEXT
1140 DATA 81,140,82,42,8
3,140,84,42,85,140,86,42
,87,140
1150 DATA 88,42,71,42,72
,140,73,42,74,140,75,42,
76,140
1160 DATA 77,42,78,140,6
1,140,62,42,63,140,64,42
1170 DATA 65,140,66,42,6
7,140,68,42,51,32,52,140
1180 DATA 53,32,54,140,5
5,32,56,140,57,32,58,140
1190 DATA 41,140,42,32,4
3,140,44,32,45,140,46,32
1200 DATA 47,140,48,32,3
1,43,32,140,33,43,34,140
,35,43
1210 DATA 36,140,37,43,3
8,140,21,140,22,43,23,14
0,24,43
1220 DATA 25,140,26,43,2
7,140,28,43,11,43,12,140
,13,43
1230 DATA 14,140,15,43,1
6,140,17,43,18,140
1240 FOR M=1 TO 4:READ N
(M):NEXT
1250 DATA -11,-9,11,9
1260 CO=0:HU=0:RETURN
1270 IF Q(22)=C AND Q(11
)=E THEN A=22:M=11:FL=1:
RETURN
```

```
1280 IF Q(22)=C AND Q(13
)=E THEN A=22:M=13:FL=1:
RETURN
1290 IF Q(24)=C AND Q(13
)=E THEN A=24:M=13:FL=1:
RETURN
1300 IF Q(24)=C AND Q(15
)=E THEN A=24:M=15:FL=1:
RETURN
1310 IF Q(26)=C AND Q(15
)=E THEN A=26:M=15:FL=1:
RETURN
1320 IF Q(26)=C AND Q(17
)=E THEN A=26:M=17:FL=1:
RETURN
1330 IF Q(28)=C AND Q(17
)=E THEN A=28:M=17:FL=1:
RETURN
1340 RETURN
1350 LPRINT TAB(7);"1234
5678"
1360 FOR F=80 TO 10 STEP
-10
1370 LPRINT TAB(4);F/10;
1380 FOR G=1 TO 8
1390 LPRINT CHR$(Q(F+G))
;:NEXT
1400 LPRINT F/10:NEXT
1410 LPRINT TAB(7);"1234
5678"
1420 FOR T=1 TO 4:LPRINT
:NEXT
1430 RETURN
1440 REM CREATE A BOARD
1450 REM The McKinna
   Option
1460 INPUT"Which square"
;Z:IF Z<11 OR Z>88 OR Q(
Z)=-99 OR Q(Z)=140 THEN
1460
1470 PRINT"Which piece?"
```

```
1480 PRINT "E - empty sq
uare"
1490 PRINT "C - comp. H
- human"
1500 PRINT "CK - comp. k
ing"
1510 INPUT "HK - human k
ing";X$
1520 Q(Z)=-42*(X$="C")-4
3*(X$="+")-75*(X$="HK")-
79*(X$="CK")-32*(X$="E")
1530 GOSUB 680
1540 PRINT "Enter P to p
lay"
1550 PRINT "or C to cont
inue"
1560 INPUT "to change bo
ard";X$
1570 GOSUB 680
1580 IF X$="P" THEN RETU
RN ELSE 1460
```

HOW THE PROGRAM WORKS

We'll now go through the program, line by line. Following through this explanation should give you an insight into (a) board game algorithms; (b) checkers' agorithms in particular; and (c) several programming techniques. Not only should your programming skills improve as a result of studying this material, but you should get a number of ideas you can apply to your own programs.

Line 10 labels the program, and line 20 clears the listing off the screen, and asks the player to "Press space bar to continue". In fact, just about any key would do at this point, but players seem to respond better to specific instructions than to things like "Press any key to continue". As well, there are some keys (such as NUM) which will not work at this point.

The last part of line 30, plus line 40, turns the time it takes the player to press the space bar into a number which is used to seed the random number generator. The variable N is set to 0 at the end of line 30, and 1 is added to this at the beginning of line 40. The computer then checks to see if a key is being pressed, and if not (that is, if INKEY$ equals "") it goes back to the start of line 40, where 1 is added to the value of N.

If the computer does find a key is being pressed, then the random number generator is seeded (line 50) and a tone (SOUND 5,5) sounds. You'll see that line 60 checks to see that the value of N is not so large that no additional tone would be sounded if this value of N was used as the first number in the SOUND

command. If it is greater than 59 (see the first part of line 60) then N is divided by two, and the value is checked again. Once the computer has reduced the value of N to less than 60, action moves to line 70, where a new note sounds (SOUND N,4).

The program proper now begins. Action moves to the subroutine starting at line 1070 where the variables are initialised. We'll be looking at that subroutine in due course. On returning from the sub-routine, the 'game loop' begins a cycle which calls each element of the program as a series of sub-routines, to actually play the game. You'll find that programming in this way, as was discussed when we played Noughts and Crosses, with the important parts of the program being called from within a perpetual loop, makes for much 'cleaner' programming, and also makes for programs which are much easier to debug than when written in other ways.

As you can see, the computer cycles from lines 110 to 150, then goes back to 110. The cycle begins by jumping into the middle of it (line 130) to give the computer the first move. As line 90 points out, you can give yourself first move by deleting line 100. Line 110 calls the subroutine starting at line 680. This prints out the board. Once the board has been printed, action goes to the subroutine from line 850 which accepts the player's move. This subroutine, as we shall see when we get there, also allows the player to choose to see the board again on the screen before moving; to print out the board; or to modify the contents of the board to his or her choice (by exercising what is known as 'The McKinna Option', named after the program tester who

suggested a 'modify the board routine' would be an interesting addition to the game).

After the player move has been made, within the subroutine beginning at line 850, line 130 sends the computer back to the routine from 680 to reprint the board. Then it is the computer's turn to move, with line 140 directing the program to the routine from line 160.

The speed of a program is dictated, to some extent, by the position within a program of often-called subroutines. The computer can take an appreciable time to find the start of a subroutine, and to find the return address. It searches through the program, line by line, from the beginning of the program every time a subroutine is called until it finds the one it wants. Therefore, the closer to the start of a program you place routines which are frequently called, the faster, generally speaking, the program will run.

The routines which determine the computer's moves are therefore at the beginning of the program, directly following the major game cycle loop. After the computer moves section comes the board printout, which is used many, many times within a game. This is followed by the 'accept the player's move' routine, where speed is not important. The 'safe move repository' (to be discussed shortly) also helps the program play quickly.

The next section of the program determines the computer's moves. Line 160 sets each element of the S array (which holds the 'safe moves') to zero, and the variable A which counts down in ones, and which

indicates the square being checked. The first square which is looked at (in the top left hand corner of the board) is number 88, one less than the initial value of A. You can see that line 180 subtracts one from the value of A, an operation which continues until the computer either runs off the bottom of the board (that is, if A becomes less than 11) or a capture is made (see line 280, which checks the value of A).

Line 190 checks to see if the selected square contains a computer piece (the variable C holds the ASCII value of the asterisk which is used for the computer's ordinary piece, and the variable CK holds the value of the letter O which is used for computer kings) and if it finds that the square does not have such a piece, directs action to line 280. Here, if A has a value greater than 11, directs action back to line 180, where A is decremented by one, and the search for a computer piece continues.

The variable B is set to zero in line 200. The N array (see line 220) is a four element array. Each element holds a number which specifies legal moves from any square. The numbers held are -11, -9, 11 and 9 (see DATA statement 1250) which, when added to the number of the square which holds a computer piece, represent a legal move.

Have a look at your numbered board. Imagine that the computer has a piece on square 55. If it is an ordinary piece (that is, not a king) it can move to square 44 (55 - 11) or to 46 (55 - 9). If the piece on square 55 is a computer king, there are two moves in addition to the -11 and -9 ones. The piece can move to square 64 (55 + 9) or to 66 (55 + 11).

You'll find this numeric relationship, between an occupied square and potential moves from that square, exists all over the board.

If the square being checked is less that 29 (that is, it is on the bottom row of the board) the computer does not bother checking the moves 'down' (which would take it off the board). It sets B equal to two, so when one is added to the value of B in line 210, the first 'king move' will be checked.

Line 220 adds the value of the element of the N array to the value of A, thus getting the number of the square into which the computer is considering moving. Line 230 checks to see if this move is actually on the board. If it is less than 11, or greater than 88 then it is, of course, off the board. If the number of the destination square does not end in zero (as it would if it were 20, 30 and so on, and thus 'off the board' to the left), the computer prints up the move it is considering, as "55 to 44?".

Next the computer checks (line 250) the contents of the square into which it is considering moving. If it holds a human piece (the variable H signifies a human piece, and holds the ASCII value of the plus sign which is used in the printed board to designate the human piece), or a human king (variable HK), and the square beyond this (M, the occupied square, plus N(B) again) is empty (the variable E holds the value of the blank, 'empty' space, ASCII 32), then the computer knows a capture can be made.

If it finds such a capture, it does not, as was mentioned before the program listing, bother looking

for further moves but proceeds to 320 to make the capture.

Line 260, the most complex in the program, looks to see first if the square into which the computer is considering to move is empty (that is, it is an E) and if it is, proceeds to check all the squares surrounding the one into which the computer is considering moving to see if the move, when made, would place the computer in danger of capture. If the computer decides that the intended move would not place it in danger, action goes to the subroutine from line 470 to store the move in the 'safe move repository'. However, even though a safe move has been found, the computer does not stop at this point but continues to search through the board, looking for additional safe moves, and for captures.

Line 270 checks the value of the piece on the square under consideration, and if it is a king (or B is less than two, meaning that only the first potential move designated by the N array has been checked) sends action back to line 210, where one is added to the value of B. Line 280 checks that the end of the board has not been reached (that is, that A has a value greater than 11) and if it has, sends action back to 180, where one is subtracted from the value of A.

Lines 290 and 300 control checking for potential promotion to a king. If no capture has been found, and regardless of the safe moves stored in the 'repository', line 290 sets a flag (variable FL) to zero, then checks the four squares (22, 24, 26 and 28) just before the final, 'promotion' row. If it finds a computer piece there, it sends action to the

subroutine at line 1270, which checks to see if there is an empty square into which the computer can move to promote its piece to a king. If it finds such a move, the flag FL is given a value of one to indicate that the promotion has been found. If it has (line 300), then action goes to line 650 which actually makes the move.

If a capture has not been found, and a king promotion cannot be made, the computer uses line 310 to go to the routine from line 500, where the safe moves are stored. We will discuss the safe moves storage and selection shortly.

Line 320 makes a capture move, setting the element of the Q array (which holds the state of the board) into which the computer is moving, M + N(B), to the value of the square from which the computer has moved, Q(A). The squares jumped over, Q(M), is then emptied (that is, its value is set to E) and the square which has been vacated by the computer move, Q(A), is also emptied.

Line 330 prints a line across the screen to warn you that a move has been made, and to alert you to watch it being printed, so you can change the pieces on an external board if you are using one. Line 340 prints out the move made, sounds the beep, and pauses for a short time, using the dummy T loop for the delay. Then, using line 350, it prints out the jubilant message "Got you!!" four times, moving slightly further across the screen each time (using TAB(2*Q)). Line 360 adds one to the computer's score, which is held by the variable CO.

Line 370 reprints the board, by going to the

subroutine starting at line 680. Then the 'number of square of the piece under consideration' is changed to equal the square the piece has just moved to. Lines 390 to 420 then check to see if a further jump can be made, and if one is discovered, line 440 sends action back to 320 to effect the capture. If the piece under consideration is an ordinary piece, line 450 makes sure the potential king moves for that piece are not examined. Line 460 returns to the master loop, where the board is reprinted, and the player's move requested.

If there is no capture, and an ordinary piece cannot be promoted to king, then the safe move repository, which comes next, is examined. Up to ten safe moves can be stored here. Line 470 checks to see if less than ten moves have been stored, and if so (that is, the variable SC has a value less than ten), SC is incremented by one. You'll recall that at the start of the move routine (line 160) the S array was filled with zeroes. This array holds the safe moves which have been found while the computer is looking for captures. Line 480 makes a beep to let you know that a safe move has been found, to keep you amused while waiting for the computer to move. Line 490 stores the 'from' and the 'to' squares as a single, unique number (which can be later decoded) and then returns to the subroutine which is examining the entire board.

Line 500 is checked if no capture and no king promotion moves are found. If SC equals zero, the computer knows there are no safe moves and line 500 sends action to the routine from line 550 which selects moves at random. Line 510 chooses one element of the A array at random, and lines 520 and

530 strip the unique number generated in line 490 back into 'to' and 'from' numbers. The actual move is made by line 650.

The routine from lines 550 to 630 chooses moves at random. The SC variable is used to count the moves. This was the safe move variable, and is used here to save introducing a new variable. Line 550 chooses a number between one and 88, and then checks to see if this element of the Q array (the array, you'll recall, which holds the entire board) is an ordinary piece or a king. If it finds it is either of these, lines 570 to 610 look to see if a move can be made by this piece, and if one is found, sends action to line 650 to make the move. If no move is found, the computer uses line 640 to concede the game.

Line 650 makes the move, if one has been found in the king-maker, safe or random category (the capture moves are made somewhat earlier in the program, at line 320). Line 660 tells you which move has been made, and sounds a warning beep so you'll know to look for the move. Again, the dummy T loop is used for a delay. Line 670 returns action to the main game loop.

The board is reprinted by the subroutine starting from line 680, the line which clears the screen. This line is, strictly speaking, not necessary, but I felt it gave a cleaner look to the output of the program if it started printing at the top of the screen, rather than always scrolling up from the bottom. Line 690 prints out the scores, and line 700 sounds a beep to let you know the board is being reprinted. Line 710 prints out the numbers across the top of the board, and lines 720 to 750 actually

print out the board, along with the numbers down the side. Line 760 prints the numbers across the bottom of the board. Line 770 checks to see if either the computer (variable CO) or the human (variable HU) has taken 12 of the opponent's pieces and if so, action to line 790. If neither of the two variables equals 12, line 780 returns from the 'print the board' subroutine.

If the human has won, line 790 acknowledges this, congratulates the player and then terminates the program. If the computer has won, it announces this, thanks the player for the game, and ends.

The next section of the program accepts the player's move. As you can see from the REM statements you enter 99 to concede the game (in response to the "FROM?" prompt), 0 to see the board reprinted on the screen, 1 to dump the board to the printer, or 2 to set up a board of your own choice (using the routine from line 1460, at the very end of the program).

If either 99, 0, 1 or 2 is entered, the computer acts as instructed, then returns to the FROM? prompt, to accept the player's real move. Line 900, as you can see, accepts the TO move, and line 910 actually makes the move. Lines 920 and 930 look to see if any computer piece can be promoted to king, and lines 940 and 950 do the same for the human's pieces. Line 960 looks to see if the difference between the 'from' and 'to' squares is less than 12. If it is, it knows that a capture has not been made, so returns to the main loop, where the board is reprinted, prior to the computer making its next move.

If, however, the computer discovers that a capture has been made, it selects a number at random between zero and one, assigns it to variable TY, and uses this to print out a congratulatory message to the player. Note that this does not happen every time a move is made. Line 1000 increments the human's score (variable HU) by one, and the second part of the line turns the square between the one move 'from' and that moved 'to', to empty. The last part of line 1000 sends action to the subroutine to reprint the board, and again a check is made to see if a human king has been created (lines 1010 and 1020).

Line 1030 asks if the player can move again, with a "Y" or "y" being interpreted as a 'yes' answer. If the player does not answer 'yes', action reverts to the main game loop again. If, however, the player can jump again, line 1060 sets the latest 'to' square to be the newest 'from' square, and then goes back to line 860 to accept a new 'to' destination.

The next section of the program is the initialisation routine, put near the end of the program as it is not accessed again. The screen is first cleared of the "Press space bar..." message, and some principle variables are assigned. H is the human piece, HK the human king, C is the computer piece and CK the computer king. E is the empty square, and the variable B is for the black squares. In line 1090, OF is the variable for 'off the board'. The Q array holds the board, and surrounding squares, the N array holds the possible moves from any position on the board, and the S array holds the safe moves. Line 1100 sets every element of the Q array to -99 (the variable OF), and the loop from 1100 to 1130 reads the Q array element number (D)

and the value to be placed in that element of the array (G).

The DATA statements from 1140 to 1230 contain the information to be held in the array. Line 1240 fills the N array, the array which holds the possible moves, using DATA from 1250. The variables to hold the human (HU) and computer (CO) scores are assigned in line 1260, when action is returned to near the start of the program.

The 'king-maker' routine occupies lines 1270 to 1340. The computer looks along the second back row to see if it has a piece there, and if it finds one, checks to see if it can move into the back row to gain a king. If it can, the flag FL is set to 1, to indicate that a king can be made. The routine from 1350 to 1430 dumps the board to the printer, in response to a '1' answer to the "FROM?" prompt.

The final routine in our program is 'The McKinna Option' which allows you to modify the board, once a game is underway, to your own specifications. A favorite modification is to give the human player a whole row of pieces less than the computer has. The routine is self-prompting and allows you to set up such things as challenging end games, to see how the computer reacts.

CHAPTER SIXTEEN - CHESS, THE FINAL FRONTIER

"It is not amazing that a bear can
be taught to dance well, but that
it can be taught to dance at all"

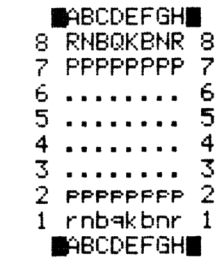
It is amazing that computers can be taught to play chess, and some of them do it brilliantly. Our program, 'Dancing Bear Chess', does not play particularly well. The main value and interest of this program, and the reason for including it in the book is that it is written in BASIC to fit the un-expanded HX20. As well, it plays fast, recognizable, (generally) legal but very weak chess.

It is included in this book because chess is among the most intriguing applications devised for a computer; because despite being in BASIC it plays swiftly; and because the study of how such a program works can indicate some approaches to the production of 'artificial intelligence' (or, at least, actions and reactions which appear intelligent) from a computer.

In the third chapter of this book, we discussed Alan Turing's thoughts on machine intelligence, and the Turing Criterion. This - as you'll probably recall - pointed out that an entity could be considered intelligent if, when interacting with it over a wire, the human being could not detect if the 'entity' was a machine or another person. The giveaway here would probably be the chess blunders the HX20 sometimes makes. But you could expect the same playing with a raw recruit to the game.

The program prints the board on the screen (and will

dump to the printer at any time) as follows:



Your pieces are the lower case letters at the bottom of the screen (the 'white' pieces) and the computer's pieces are at the top ('black'). The computer has the opening move. You indicate your move by entering the letter and number (as 'D2') of the piece you want to move, then press ENTER, then enter the letter and number (as 'D3') of the square you wish to move to.

The computer plays very quickly, with most moves made within 20 seconds. It also prints up on the screen the order in which it is considering moving the pieces (this order changes as the game progresses), and if it cannot find a capture move which it judges is worth making, it will tell you where in that list it will start looking for a move.

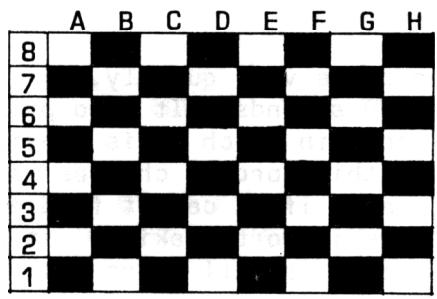
The HX20 chooses from this list using weighted random numbers, so it will never play the same game twice in a row, even when confronted with identical circumstances.

Once the computer has made its move, it signals this to you by making a series of sounds, and printing up on the screen the move it will make (along with the

time) and then continuing with the sound until you touch the "Q" key. It will then make the move, and reprint the board, before asking you to move.

This feature, of waiting till you say you are ready to see the move, allows you to walk away from the computer, then return when you are ready to see the move it has selected.

It is almost impossible to play this game without an external board and pieces (unless you are very patient and have an endless supply of printing paper). I have deliberately used standard (algebraic or continental) notation because it is fairly easy to buy a chess board which is already marked in this way. This is how the numbers and letters are arranged in standard algebraic notation:



There are two versions of the program given here. The first is generously supplied with REM statements, to allow you to unravel the program, and this is the version you should enter if you wish to exercise your mind by tackling the most rewarding task of improving the computer's play. If, however, you simply want a version of the program which will play as quickly as possible, and will be as simple

as possible to enter, use the second version. The notes given here refer to both versions, as I have not renumbered the shorthand version.

The computer plays as quickly as it does because of a few simple dodges. For a start, it always looks at the squares on the board in a predetermined order, an order which I devised which appeared to give the maximum trade-off between the quality of the computer's play and the speed of its reply. The second dodge is a further trade-off, in which the depth to which the computer searches for a possible reply to its intended move is kept to the absolute minimum, without introducing idiotic, completely random play.

I had been thinking of writing this program for about a year before I got around to doing it. After the Personal Computer World show in London in September, 1982, I had arranged a two week holiday in Wales and took with me a number of magazine articles on computer chess, and an introductory book on how to play chess written for children.

I started writing the first outline of the code in my notebook, then bought an exercise book to keep the outline (which was rapidly turning into a monster) into some sort of order. I realised that a chess game that was unable to get out of check would be worse than useless. For two days I tried to work out how the computer could (a) detect it was in check; and (b) once it had, work out how to get out of it. Then, when walking near the tiny coastal village of Amroth I worked out how to solve most of the check problem, and then seconds later realised that the same technique should work for any other

piece to detect both danger to itself in its present position, and the dangers it would face if it made the move it was contemplating.

The heart of the program's 'intelligence' (such as it is) lies in the routine I've called the "Amroth Defense", the subroutine which starts at line 410, and which is called time and time again when a move is contemplated. This routine is the real workhorse of the program, detecting that the player is in check, finding out whether or not the computer is in check, discovering a move (if such a move exists within its repertoire) to get out of check, looking for human pieces to capture, and so on. Despite the "Amroth Defense", the program plays like a real novice.

I knew the running speed of the program would be critical, and because programs with often-used subroutines placed at the beginning run fractionally faster than they do if such subroutines are towards the end, carefully structured the entire program so that the action would tend to flow smoothly within it. Non-critical routines (like the one which accepts the player's move, the one which reprints the board, the one which allows you to change sides, and the one which comments on your moves) are at the end. The master loop which calls the subroutines which do all the work is at the beginning, and the Amroth Defense is as close to the start of the program as I could place it.

I knew that the more elaborate the strategy I tried to build into the program, the longer it would take to move and the less flexible it play could be. I read through the children's "how to play chess"

book, looking for absolutely elementary strategy (such as backing up pawns with other pawns, and trying to possess the center of the board) and combined that with with my understanding of chess (including such rudimentary ideas as the one that knights should be developed before bishops) to create a program that, while under the influence of the random number generator, still managed to play with some apparent awareness of the implications of its actions.

Chess is one of the oldest of man's great games and remains the most fascinating and popular of them all. The intellectual challenge involved in teaching a machine to play chess has attracted many, many worked in the field. The game involves no chance, unless you count opportunities created by the blunder of the opponent) and appears an infinite resource for new and challenging play. The HX20 is not hidebound by the traditions of chess, so some of its openings may surprise, exasperate and (occasionally) delight you. Note that, in common with other 'primitive' chess programs, this program does not support en passant nor castling.

As was indicated by the board printout at the start of this chapter, the pieces are shown as letters, using capital letters for the computer's pieces (with the knight as "N" to distinguish it from the king) and lower case letters for your pieces (again with "n" for the knight). The black and white checkerboard on which chess is usually played is represented in this program by a series of dots. It is possible to write a simple routine to turn the appropriate squares black, but doing this does not aid interpretation of the board significantly, and

it slows down the printing of the board to a considerable extent. Therefore, I have opted for an eight by eight grid of simple dots to represent the battleground.

As you know, the point of chess is to get your opponent's king in check. When the king is in check it means that it could be captured on the next move. You are not allowed to capture a king, but signal check by entering a 'C' after the board reprints following your move, and waits for an input. The computer will look, anyway, to see if it is in check as one of its first priorities, but you still need to enter 'C', as this is used later (see lines 270 and 3010) for helping the computer to determine if it will concede the game.

The HX20 will attempt to get out of check by capturing the offending piece, or moving out of check. This is the one point where the computer program demonstrates it is only a dancing bear and not a ballerina. From time to time, it will commit the unpardonable sin of moving its king out of check by moving onto another square which is under attack. This happens rarely, and can either be taken as a sign that the computer concedes the game, or (if you feel merciful) you can leave it, and it will undo the damage with its next move.

The computer will signal 'check' to you in the majority of cases if it gets you in check, but it is worth concentrating on the game because, from time to time (again rarely) the computer will put you in check, but not tell you.

When you run the game, you'll see the board

reprinted after each move. When you make your move, the board will reprint, and the computer will then wait for an input before continuing. If you want the computer to move, just press ENTER (and you can get the same result by pressing ENTER twice after entering the 'to' square of the move, as the board will reprint and the computer then automatically continue on to make its move). Pressing "P" will copy the board to the printer (assuming the printer is turned on) and "X" will swap sides.

When you enter an "X", the HX20 carries out a mirror image swap, with your pieces being reflected in an imaginary mirror placed in the middle of the board, and the computer's pieces being swapped in a similar manner to your side, with the colors also being exchanged. That is, you still use lower case letters and the computer still uses upper case ones.

In our history chapter, we met Claude Shannon, and discussed the contribution he made by producing the important paper "A Symbolic Analysis of Relay and Switching Circuits" which explained how circuits could be wired to carry out mathematical and logical tasks. Shannon produced another important paper, Programming a Computer for Playing Chess, a decade later. In this, he pointed out that there were around 10 raised to the 120th power possible games of chess of 40 moves and it would take a computer 10 raised to the 90th power years to analyze to this depth at the rate of one game per microsecond. To drive home the size of the number, it is worth noting that there are less than 10 raised to the 120th power atoms in the universe.

Shannon concluded that there was no way a computer

could be taught to play chess purely by exhaustive analysis. He went on to outline how the two sides in a game could be evaluated. His evaluation function included rule of thumb relative values of pieces with the pawn as one; the queen, nine; rook, five; and bishop and knight three each.

Shannon suggested some refinement on these values, including putting an emphasis on the desirability of the center of the board to be under the control of the pawns if possible (a feature your program supports, with a sampling on squares to make moves which give the pawns controlling the center priority in the early stages of the game), the weakness of pawns in front of your own king after castling, and that - if possible - both rooks should ideally be on the same file to concentrate their attack.

Fully aware that position analysis of this type was of limited value, Shannon said that an ideal program (and, at that time, there were no chess-playing programs at all) would be aware that after a major capture such as one in which a queen is taken, the obvious response would be look for a possible capture in exchange. An evaluation function would not necessarily lead to this sort of move being made. Although when you play chess, or a similar game, you can see such things intuitively, a computer must be taught to recognise when a straight evaluation of each sides' strength, and a move based on a series of 'tactics' learned by rote, is not appropriate.

Shannon suggested that a program which always searched for moves to a fixed depth was a 'type A' strategy and said a 'type B' strategy, which allowed

the computer to reject some paths as 'obviously' not worth following, was required. In fact, most dedicated chess machines, and the better chess programs on the market, now use what may be called a 'Shannon B' strategy. Shannon said such a strategy could be developed by weighting the computer's assessment of a move so that it tended to pursue moves which captured, created or threatened check, before it looked at defensive moves.

A very primitive Shannon B strategy is used in our HX20 program in which after an "Am I in check?" routine is explored, the computer looks for its 'best' capture, using the generation of heavily-weighted random numbers to modify straight piece-value comparisons. For example, it knows that any capture by a pawn is likely to be of value (and certainly a capture of any piece other than a pawn is almost definitely of value, evaluated purely on a one-for-one capture basis, without looking at other implications of the move).

Therefore, the pawn capture routine from lines 2410 to 2480 uses (as does the rest of the program) the flag MM - for 'machine move' - to signal whether or not a move has been found, and having it set to one when a suitable move has been discovered. Line 2430 sets MM to one if any opponent piece is found to be under attack by a computer piece, and (in the second statement of line 2430) only changes this back to a zero if the piece under attack is a human pawn and the random number generated between zero and one is less than 0.2 This means, in practice, that the computer will always capture a piece which is not a pawn by a pawn if it can, and will capture a pawn by a pawn around 80% of the time. The 0.2 value, along

with the other values in random evaluation routines in the program, was first determined intuitively (that is, I took a guess) and then modified this by trial and error. You'll see another such weighting in line 1470, when 10% of the time (more or less) a non-capture move by a rook will be checked to see if the square it is considering is under attack, and a move will be made there if it is not.

Shannon also suggested that a computer could be programmed to recognise typical board positions, and would know which move had proved best in practice against that move. Only one serious attempt to implement this approach has been made. However, the program played badly at its first major public outing, a chess tournament in the early seventies, and the idea has not been used to a significant extent in any major chess program since that time.

The first program written to play a complete game of chess was developed by an IBM employee, Alex Bernstein, in the early 1950's, on one of his company's computers, the IBM 704. Bernstein's program asked a number of questtions, which determined which types of moves would be further considered. A similar approach is used in most programs which purport to display intelligence (including the Checkers and Noughts and Crosses programs in this book). His main question was the same as ours: "Is the king in check?" and he followed this with questions which included an understanding that the program would be encouraged to exchange pieces with the opponent when it was ahead, and to discourage it from doing so when it was behind.

However, it shared a weakness with DANCING BEAR CHESS. No question of the form "Can I give check?" was included, so the computer had no mechanism to aggressively pursue check. This is the kind of development you could include within the program when you start improving it.

We'll now look at our program, and I will explain the major routines and the thinking which lies behind them. After the screen has cleared when you first run the program, the message "Dancing Bear Chess (i) Hartnell Please stand by..." appears and remains on the screen while the initialisation procedure is carried out. Line 40 ensures that only integer variables will be used, a process which saves considerable operating time, as will be proved if you delete this line.

Line 3170 seeds the random number generator and a number of arrays which are used within the program are dimensioned in 3180. The A array holds the board, R holds the possible rook 'displacements' from the square currently occupied by a rook, B the bishop moves, N the knight moves, Q the queen moves (which are the bishop moves plus the rook moves). Z is used in the exchange routine when swapping sides, S for the order in which the squares on the board are checked, and T for holding the current position of the computer's pieces.

The variables are assigned in line 3200 for the human pieces, using obvious variable names (such as P for pawn and K for king). The E stands for 'empty'. The computer pieces (black) are assigned in line 3200, again using easily recognisable names (PB for the black pawn, BB for the black bishop and KB

for the black king).

The loop from 3230 to 3250 fills all elements on the major array, A, with 42, which will later be used to indicate to the computer that the move it is considering is 'off the board'. The next Z loop (3260 to 3290) fills most of the A array (the elements representing squares on the board) with their opening values (that is, it sets up the board for the start of the game).

The next section fills the potential move arrays with their 'displacements'. The knight moves are fed into the N array in the routine from lines 3470 to 3520, and as follows: rook - 3530 to 3610; bishop - 3670 - 3700; queen - 3710 to 3750 (note that the queen uses the DATA statements from the rook and the bishop); and king - 3760 to 3810. The Z loop used in lines 3820 to 3840 feeds the order in which squares are checked during the game into the S array.

Once this rather lengthy initialisation procedure has occurred, the computer makes a little music (line 3930) to let you know the game is about to begin, then returns to line 60. Here, the GOTO 90 sends action to that line to give the computer the opening move. After this, a standard cycle is followed. The board is printed using the subroutine from line 2760, the player move is accepted (line 80, sending to the subroutine from 3040) and the board reprinted. Then, the lengthy machine move procedure begins. This procedure, not surprisingly, occupies most of the program.

Firstly the move flag MM is set equal to zero. If

this becomes set to one at any time in the next two hundred odd lines, the computer has found a move and decided to make it. Line 110 holds the program for a moment (which, from the player's point of view, is just after the board has been reprinted following the acceptance of his or her move), waiting for the player to either simply press RETURN (at which point the computer will start looking for a move; "C" indicating the computer is in check; "P" to tell the computer to print out the board; or "X" to signal that a 'mirror swap' exchange (as explained earlier) is needed. After any of these, the computer will look for a move (although in the case of the exchange, will reprint the board before doing so, and give you the opportunity of requesting a printout at that point before looking for a move with its 'new' pieces).

The loop from 160 to 170 fills the T array with zeroes. The T array will hold the location of the computer's pieces, which saves a considerable amount of time in subsequent processing, as the computer does not have to continually 'rediscover' where its pieces are (as it does, for example, in the Reversi and Checkers programs). In line 180, U is set equal to zero. U is used to count the number of pieces the computer has on the board, so it knows how many pieces it must consider for moves. The complex-looking line 200 performs a number of tasks:

- it searchs through the squares on the board, in the order determined by the contents of the S array, to locate the computer pieces
- it counts them, using the variable U
- it places the pieces, in the order in which they have been discovered, into the T array

- it prints out the piece on the board
- if the piece is a king, variable KM (for 'king marker') is set to that square, so the computer knows where its king is at all times

The next line completes the Q array. If it discovers it only has two pieces on the board (that is, U is less than three) it sends action to line 2390 to concede the game. If you want the program to play to the bitter end, you can delete the last half of line 210.

Action now goes (via line 230) to the routine from 740, which checks to see if the king is in danger. Z is set equal to the king's position (held, you'll recall, by the king marker, KM). It checks the KM square against every possible attack, going to 1990 to try and get out of danger if it discovers it is in trouble.

It checks to see first if it is under attack from a knight, then a bishop, queen or rook (with a single routine), then for danger from a pawn. If it does not survive this battery of tests, which continue up to and including line 1000, then the computer goes - as I said - to 1990 if it finds trouble. Here, as the REM statement says, KING LOOKS FOR SAFE MOVE. If it finds a safe move, it goes to the small routine from 2690 to 2750 to make the move, and then goes back, via a zigzag routine, to the routine which prints the board. If it does not find a safe move in the long routine from 1990 to 2380 it concedes the game. The routine which acts on a safe move is from 2320 to 2350.

If it has discovered it is not in danger, 1010 sends

action back to line 240, SHUFFLE KING. This REM statement refers to lines 250 and 260, which move the king to the end of the pieces which will be considered from now on. The king is understandably reluctant to move its king if any other option is open to it.

Next the computer chooses a piece to move. Line 270 sets Q to equal zero, one or two (and the last part sets it to zero regardless, if you have told the computer it is in check, that is A$ = "C"). Line 280 adds one to this to be the piece it will consider first, and 290 sets Z equal to the number of that square (which, it knows from the routine in lines 200 to 210, contains the first piece it will consider moving). Jumping to 350, it goes to the subroutine which handles that piece's moves. Line 400 returns to 310, where a check is made to see if the 'machine move' flag equals one. If it does, a move has been decided upon, and the subroutine (after it has written its move on the screen "I moved from D7 to D5", and the player has pressed "Q" to signal the move has been noticed), the action returns to line 70 (via 320) to print the board before accepting the player's move.

If a move has not been made (that is, MM still equals zero), the computer checks in line 330 to see that Q (the number of the piece it has just considered) is lower than U (the total number of pieces the HX20 has on the board) and if it finds that Q is lower than U, it returns to 280, where Q is incremented by one, and the search continues with the next piece looking for a capture. As I said earlier when discussing the Shannon B search technique, the computer may well have found a

possible capture in its run through the routine, but the weighted random number generated told it not to make the move.

Once all the pieces have been checked to see if they can capture, line 340 sends the program to 2540 to make a non-capture move. Line 2550 generates a random number between zero and eight, using the two RND(1)'s to heavily weight the numbers in favor of those closer to zero. It does not stop the higher numbers being generated, but makes sure they appear less often than the lower ones. The end of 2550 checks to make sure the number generated (Q) is less than the number of pieces on the board (U), and -- if needed — goes back to the start of the line to get another starting value. When you run the program, the computer will print up all its pieces in the order in which it will consider them. The next line (2560) tells you where along that list it will now start looking for a piece to move.

Z is set equal to the piece held in T(Q) in line 2580 and the next section (much the same as the capture search section) sends the computer to the subroutine related to the piece Z. This routine is traversed until a move is found, and MM is set equal to one, triggering lines 2660 and 2670. If no moves have been found, the program 'falls through' lines 2650, 2660 and 2670 to trigger the "It's your game" message and the program terminates. The routine from 2920 to 2990 is used by the king to locate a move to escape check.

This covers the 'intelligent' parts of the program. The rest is reasonably self-explanatory and governs a number of house-keeping tasks:

2690 - 2750: Tells the player the move the computer is going to make, prints up the time, and returns

2760 - 2910: Prints out the board, holdiong at line 2780 if the computer has just moved until the player presses "Q" to signal that he or she has noted the move and is ready to proceed

2830 - 2840: These are within the board print routine, and serve to promote a pawn which has made it to the back row to queen. This routine is checked every time the board is reprinted. It does not significantly increase the time it takes to reprint the board

3040 - 3130: This accepts the player's move. If the player is capturing a computer piece (detected by line 3100) the action goes to the 'speech sub-routine' from 4170 which either makes a few beeps to acknowledge the capture and returns, or chooses from one of three messages ("Well done", "Good move" or "Got me"), holds the message for a few seconds (using the delay loop in line 4290) then returns to the player move routine

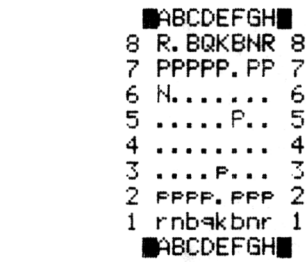
3950 - 4050: This covers the 'exchange', the mirror-image swap we've discussed. The best way to understand this is to play the game for a while, then enter an "X" to do the exchange. Having done this, print the board out, and you'll see the effect of the exchange. It is quite interesting to do this if you're being thoroughly beaten by the computer (and you want a second chance) or you are thrashing it and you want to see what it would do with your material

4060 - 4160: This routine simply dumps the program to the printer, when you've entered a "P", following your move

****

The program was modified slightly so that it would 'play against itself', swapping the pieces after each move, so it was considering its own previous move as an opponent. The play is undirected, without even the guidance that the computer receives when playing against a human, but it is fascinating to see how the computer plays. Its opening moves are not very promising:

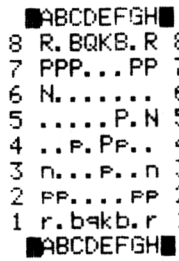
```
Dancing Bear Chess
   (i)-Hartnell
Please stand by...
      ABCDEFGH
    8 RNBQKBNR 8
    7 PPPPPPPP 7
    6 ........ 6
    5 ........ 5
    4 ........ 4
    3 ........ 3
    2 pppppppp 2
    1 rnbqkbnr 1
      ABCDEFGH


      ABCDEFGH
    8 RNBQKBNR 8
    7 PPPPP.PP 7
    6 ........ 6
    5 .....P.. 5
    4 ........ 4
    3 ........ 3
    2 pppppppp 2
    1 rnbqkbnr 1
      ABCDEFGH
```

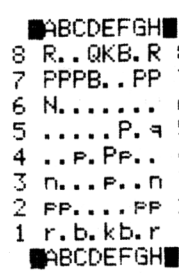


Not content with bringing out the queen's knight, the computer (as black, at the top of the board) brings out the king's knights as well:

Nonplussed, white elects to also develop a knight:

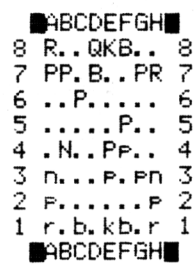
Later, black moves its king's knight into danger, and the white queen swoops:

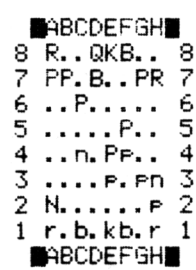




If she had stayed there, the queen might have been safe, but she foolishly elects to capture a pawn (!) and the black rook immediately pays her back:

Black now moves in on the attack, bringing first its queen's knight down, then following that with the queen herself:





```
  ABCDEFGH
8 R...KB.. 8
7 PP.B..PR 7
6 ..P..... 6
5 Q....P.. 5
4 ....Pp.. 4
3 ....p.pn 3
2 N......p 2
1 r.b.kb.r 1
  ABCDEFGH
```

CHECK!

By that stage, white was in real trouble, having thrown away much material in the early stages. I stopped the game, and started a new one, to see how white would fare this time:

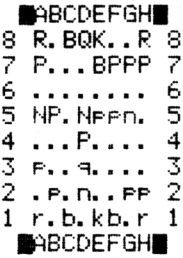
```
  ABCDEFGH
8 RNBQKBNR 8
7 PPP..PPP 7
6 ........ 6
5 ....P... 5
4 ...P.... 4
3 ..pp.... 3
2 pp..pppp 2
1 rnbqkbnr 1
  ABCDEFGH
```

```
  ABCDEFGH
8 RNBQKBNR 8
7 PP...PPP 7
6 ..P..... 6
5 ....P... 5
4 ...p.... 4
3 ...p.... 3
2 pp..pppp 2
1 rnbqkbnr 1
  ABCDEFGH
```

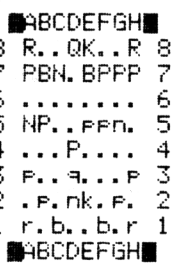
Things were settling down to a tussle of the pawns:

It was time to bring in the heavies:

White seems hell-bent on developing pieces:



But a fatal mistake is just waiting to happen, as white decides to take the king out for a breath of fresh air, perhaps to support the queen:



Watch closely now, and see how white puts the king back in obscurity, but not on its own square. As you'll see in a moment, this move is a disaster waiting to happen:

White now does the unthinkable, moving the queen from D3 to C3, right into the jaws of the pawn on D4. Not only does this throw the queen away, but it opens the way for check from the black queen, the full length of the board away:

CHECK!

Later in the game, having overcome this series of moves, white has managed to build a protective wall between the king and the outside world:

And sends forth a pawn to demolish firstly another pawn, and then then bishop on D6:

But nothing can compensate for its earlier blunder, and white conceded to black a few moves later. It is just as well the program plays black when it is playing against you.

Here is the first version of the program, liberally supplied with REMs and the like:

```
10 CLS
20 PRINT:PRINT "Dancing
Bear Chess"
30 PRINTTAB(3)"(i)-Hartn
ell":PRINT"Please stand
by...";
40 DEFINT A-Z
50 GOSUB 3140:REM INITIA
LISE
60 GOTO 90:REM  TO GIVE
MACHINE FIRST MOVE
70 GOSUB 2760:REM PRINT
BOARD
80 GOSUB 3040:REM PLAYER
 MOVE
90 GOSUB 2760:REM PRINT
BOARD
100 MM=0
110 INPUT A$:REM  CHECK
(C), EXCHANGE (X), PRINT
 (P)
120 IF A$="X" THEN GOSUB
 3960:GOTO 90
130 IF A$="P" THEN GOSUB
 4060
140 REM POTENTIAL CAPTUR
E CHECK
150 REM FIND COMPUTER PI
ECE
160 FOR Z=1 TO 16
170 T(Z)=0:NEXT
180 U=0
190 FOR Q=1 TO 64
200 IF A(S(Q))=BB AND A
(S(Q))<=RB THEN U=U+1:T(
U)=S(Q):PRINT CHR$(A(S(Q
)));:IFA(S(Q))=KB THEN K
M=S(Q)
210 NEXT Q:IF U<3 THEN 2
390
220 PRINT
230 GOTO 740
```

```
240 REM SHUFFLE KING
250 FOR Q=1 TO U:IF T(Q)
=KB THEN T(Q)=T(U):T(U)=
KB
260 NEXT Q
270 Q=INT(RND(1)*3)*((A$
="C")+1)
280 Q=Q+1
290 Z=T(Q)
300 GOSUB 350
310 IF MM=1 THEN GOSUB 2
690
320 IF MM=1 THEN 70
330 IF Q<U THEN 280
340 GOTO 2540
350 IF A(Z)=QB THEN GOSU
B 1020
360 IF A(Z)=RB THEN GOSU
B 1280
370 IF A(Z)=BB THEN GOSU
B 1540
380 IF A(Z)=NB THEN GOSU
B 1800
390 IF A(Z)=PB THEN GOSU
B 2410
400 RETURN
410 REM "CHECK?"
420 IF A(X)=107 THEN PRI
NT "CHECK!":Q=Q+1:GOTO29
0
430 IF X+9>88 THEN 450
440 IF A(X+9)<83 AND A(X
+9)>65 AND RND(1) <0.96
THEN RETURN
450 IF X-11<11 THEN 470
460 IF A(X-11)<83 AND A(
X-11)>65 AND RND(1) <0.9
6 THEN RETURN
470 REM Q/R/B CAPTURE CH
ECK
480 AD=0
490 AY=1
500 AX=X+Q(AY+AD)
510 IF AX<11 OR AX>88 TH
EN 540
520 AP=A(AX)
```

```
530 IF AP=Q OR AP=R AND
RND(1)>0.8 OR AP=B AND R
ND(1)>0.5 THEN RETURN
540 AY=AY+1
550 IF AY<8 THEN 500
560 AD=AD+7
570 IF AD<56 THEN 490
580 REM KNIGHT CAPTURE C
HECK
590 AY=1
600 AX=X+N(AY)
610 IF AX<11 OR AX>88 TH
EN 630
620 IF A(AX)=N THEN RETU
RN
630 AY=AY+1
640 IF AY<9 THEN 600
650 REM KING/PAWN CAPTUR
E CHECK
660 AY=1
670 AX=X+K(AY)
680 IF AX<11 OR AX>88 TH
EN 700
690 IF (A(AX)=K OR A(AX)
=P) AND RND(1)>0.1 THEN
RETURN
700 AY=AY+1
710 IF AY<9 THEN 670
720 MM=1
730 RETURN
740 REM KING CHECK
750 Z=KM
760 REM KNIGHT DANGER?
770 Y=0
780 Y=Y+1
790 X=Z+N(Y)
800 IF X<11 OR X>88 THEN
 820
810 IF A(X)=N THEN 1990
820 IF Y<8 THEN 780
830 REM ROOK/BISH/QUEEN
DANGER?
840 D=0
850 Y=0
860 X=Z+Q(Y+D)
```

```
870 IF X<11 OR X>88 THEN
 920
880 IF A(X)=B OR A(X)=Q
OR A(X)=R THEN 1990
890 IF A(X)<>E THEN 920
900 Y=Y+1
910 IF Y<8 THEN 860
920 D=D+7
930 IF D<56 THEN 860
940 REM PAWN DANGER?
950 X=Z+11
960 IF X>88 THEN 980
970 IF A(X)=P THEN 1990
980 X=Z-11
990 IF X<11 THEN 240
1000 IF A(X)=P THEN 1990
1010 GOTO 240
1020 REM QUEEN CAPTURE
1030 D=0
1040 Y=1
1050 X=Z+Q(Y+D)
1060 IF X<11 OR X>88 THE
N 1120
1070 IF A(X)=42 OR A(X)>
=BB AND A(X)<=RB THEN 11
20
1080 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410:IF MM
<>1 THEN 1120
1090 IF MM=1 THEN RETURN
1100 Y=Y+1
1110 IF Y<8 THEN 1050
1120 D=D+7
1130 IF D<56 THEN 1040
1140 RETURN
1150 REM QUEEN MOVE
1160 D=0
1170 Y=1
1180 X=Z+Q(Y+D)
1190 IF X<11 OR X>88 THE
N 1250
1200 IF A(X)<>E THEN 125
0
1210 IF RND(1)>0.5 THEN
GOSUB 410:IF MM=0 THEN 1
250
```

```
1220 IF MM=1 THEN RETURN
1230 Y=Y+1
1240 IF Y<8 THEN 1180
1250 D=D+7
1260 IF D<56 THEN 1170
1270 RETURN
1280 REM ROOK CAPTURE
1290 D=0
1300 Y=1
1310 X=Z+R(Y+D)
1320 IF X<11 OR X>88 THE
N 1380
1330 IF A(X)=42 OR A(X)>
=BB AND A(X)<=RB THEN 13
80
1340 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410:IF MM
=0 THEN 1380
1350 IF MM=1 THEN RETURN
1360 Y=Y+1
1370 IF Y<8 THEN 1310
1380 D=D+7
1390 IF D<28 THEN 1300
1400 RETURN
1410 REM ROOK MOVE
1420 D=0
1430 Y=1
1440 X=Z+R(Y+D)
1450 IF X<11 OR X>88 THE
N 1510
1460 IF A(X)<>E THEN 151
0
1470 IF RND(1)<0.1 THEN
GOSUB 410
1480 IF MM=1 THEN RETURN
1490 Y=Y+1
1500 IF Y<8 THEN 1440
1510 D=D+7
1520 IF D<28 THEN 1430
1530 RETURN
1540 REM BISHOP CAPTURE
1550 D=0
1560 Y=1
1570 X=Z+B(Y+D)
1580 IF X<11 OR X>88 THE
N 1640
```

```
1590 IF A(X)=42 OR A(X)>
=BB OR AX<=RB THEN 1640
1600 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410:IF MM
<>1 THEN 1640
1610 IF MM=1 THEN RETURN
1620 Y=Y+1
1630 IF Y<8 THEN 1570
1640 D=D+7
1650 IF D<28 THEN 1560
1660 RETURN
1670 REM BISHOP MOVE
1680 D=0
1690 Y=1
1700 X=Z+B(Y+D)
1710 IF X<11 OR X>88 THE
N 1770
1720 IF A(X)<>E THEN 177
0
1730 IF RND(1)>0.05 THEN
 GOSUB 410:IF MM<>1 THEN
 1770
1740 IF MM=1 THEN RETURN
1750 Y=Y+1
1760 IF Y<8 THEN 1700
1770 D=D+7
1780 IF D<28 THEN 1690
1790 RETURN
1800 REM KNIGHT MOVE
1810 Y=1
1820 X=Z+N(Y)
1830 IF X<11 OR X>88 THE
N 1870
1840 IF A(X)=42 THEN 187
0
1850 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410
1860 IF MM=1 THEN RETURN
1870 Y=Y+1
1880 IF Y<9 THEN 1820
1890 RETURN
1900 REM RANDOM KNIGHT
1910 Y=0
1920 X=Z+N(INT(RND(1)*8+
1))
```

```
1930 IF X<11 OR X>88 THE
N 1920
1940 IF A(X)=42 THEN 192
0
1950 Y=Y+1
1960 IF A(X)=E THEN GOSU
B 410
1970 IF MM=1 OR Y>20 THE
N RETURN
1980 GOTO 1920
1990 REM KING LOOKS FOR
SAFE MOVE
2000 YK=1
2010 Z=KM
2020 X=Z+K(YK):X1=X
2030 IF X<11 OR X>88 THE
N 2360
2040 IF A(X)=42 OR A(X)>
65 AND A(X)<83 THEN 2360
2050 IF A(X)>97 AND A(X)
<115 THEN 2360
2060 Z=X
2070 REM KNIGHT DANGER?
2080 Y=0
2090 Y=Y+1
2100 X=Z+N(Y)
2110 IF X<11 OR X>88 THE
N 2130
2120 IF A(X)=N THEN 2360
2130 IF Y<8 THEN 2090
2140 REM ROOK/BISH/QUEEN
 DANGER?
2150 D=0
2160 Y=1
2170 X=Z+Q(Y+D)
2180 IF X<11 OR X>88 THE
N 2230
2190 IF A(X)=B OR A(X)=Q
 OR A(X)=R THEN 2360
2200 IF A(X)<>E THEN 223
0
2210 Y=Y+1
2220 IF Y<8 THEN 2170
2230 D=D+7
2240 IF D<56 THEN 2170
2250 REM PAWN DANGER?
2260 X=Z+11
```

```
2270 IF X>88 THEN 2290
2280 IF A(X)=P THEN 2360
2290 X=Z-11
2300 IF X<11 THEN 2320
2310 IF A(X)=P THEN 2360
2320 X=X1:Z=KM
2330 MM=1
2340 GOSUB 2690
2350 GOTO 70
2360 YK=YK+1
2370 Z=KM
2380 IF YK<9 THEN 2010
2390 PRINT "I CONCEDE, C
HAMP!"
2400 END
2410 REM PAWN CAPTURE
2420 X=Z+9
2430 IF A(X)>=B AND A(X)
<=R THEN MM=1:IF A(X)=P
AND RND(1)<0.2 THEN MM=0
2440 IF MM=1 THEN RETURN
2450 IF Z=12 THEN RETURN
2460 X=Z-11
2470 IF A(X)>=B AND A(X)
<=R THEN MM=1:IF A(X)=P
AND RND(1)<0.2 THEN MM=0
2480 RETURN
2490 REM PAWN MOVE
2500 IF Z-10*(INT(Z/10))
=7 AND A(Z-1)=E AND A(Z-
2)=E AND(A(Z-13)=E OR A(
Z-13)=42) AND (A(Z+7)=E
OR A(Z+7)=42)THEN X=Z-2:
MM=1:RETURN
2510 IF A(Z-1)=E AND A(Z
-12)<98 AND A(Z+8)<98 TH
EN X=Z-1:MM=1:RETURN
2520 IF RND(1)<0.1 AND A
(Z-1)=E THEN X=Z-1:MM=1:
RETURN
2530 RETURN
2540 REM NON-CAPTURE MOV
ES
2550 Q=INT(RND(1)*RND(1)
*9):IF Q>U THEN 2550
2560 PRINT "Start piece:
";Q+1
```

```
2570 Q=Q+1
2580 Z=T(Q)
2590 IF A(Z)=PB THEN GOS
UB 2490
2600 IF A(Z)=NB THEN GOS
UB 1900
2610 IF A(Z)=BB THEN GOS
UB 1670
2620 IF A(Z)=RB THEN GOS
UB 1410
2630 IF A(Z)=QB THEN GOS
UB 1150
2640 IF A(Z)=KB THEN GOS
UB 1990
2650 IF MM=0 AND Q<U THE
N 2570
2660 IF MM=1 THEN GOSUB
2690
2670 IF MM=1 THEN GOTO 7
0
2680 PRINT"It's your gam
e":END
2690 REM COMPUTER MAKES
MOVE
2700 A(X)=A(Z)
2710 A(Z)=E
2720 PRINT "I moved from
"
2730 PRINT CHR$(INT(Z/10
)+64);(Z MOD 10);" to  "
;CHR$(INT(X/10)+64);(X M
OD 10)
2740 PRINT:PRINT TIME$;
2750 RETURN
2760 REM PROMOTE, PRINT
BOARD
2770 IF MM=0 THEN 2790
2780 A$=INKEY$:IFA$<>"Q"
 THEN SOUND RND(1)*50,1:
GOTO 2780
2790 CLS:PRINT TAB(6);"■
ABCDEFGH■"
2800 FOR X=8 TO 1 STEP -
1
2810 PRINT TAB(3);"■";X;
```

```
2820 FOR Y=10 TO 80 STEP
 10
2830 IF A(Y+1)=PB THEN A
(Y+1)=QB
2840 IF A(Y+8)=P THEN A(
Y+8)=Q
2850 PRINT CHR$(A(X+Y));
2860 NEXT Y
2870 PRINT X
2880 NEXT X:MM=0
2890 PRINT TAB(6);"■ABCD
EFGH■";
2900 SOUND 1,1:SOUND 2,1
:SOUND 1,1
2910 RETURN
2920 Z=KM
2930 QK=0
2940 M=Z+K(QK)
2950 IF A(M)=42 OR A(M)>
65 AND A(M)<83 THEN 3000
2960 IF MM=0 THEN 3000
2970 X=M
2980 KM=X
2990 RETURN
3000 IF QK<8 THEN 2940
3010 IF A$<>"C" THEN RET
URN
3020 PRINT "I CONCEDE TH
E GAME";
3030 END
3040 REM ACCEPT PLAYER M
OVE
3050 PRINT:INPUT "FROM (
LETTER,NO)";A$
3060 PRINT A$;" TO";
3070 INPUT B$
3080 X=10*(ASC(A$)-64)+V
AL(RIGHT$(A$,1))
3090 Y=10*(ASC(B$)-64)+V
AL(RIGHT$(B$,1))
3100 IF A(Y)>=75 AND A(Y
)<=82 THEN GOSUB 4170
3110 A(Y)=A(X)
3120 A(X)=46
3130 RETURN
```

```
3140 REM DANCING BEAR CH
ESS (i)
3150 REM HARTNELL Sep. 8
2 - Feb. 83
3160 REM INITIALISE
3170 RANDOMIZE VAL(RIGHT
$(TIME$,2))
3180 DIM A(99),R(28),B(2
8),N(8),Q(56),K(8),Z(88)
,S(64),T(16)
3190 REM HUMAN PIECES,
     WHITE
3200 P=112:R=114:N=110:B
=98:Q=113:K=107:E=46
3210 REM COMP. PIECES,
     BLACK
3220 PB=80:RB=82:NB=78:B
B=66:QB=81:KB=75
3230 FOR Z=1 TO 99
3240 A(Z)=42
3250 NEXT Z
3260 FOR Z=1 TO 64
3270 READ X:READ Y
3280 A(X)=Y
3290 NEXT Z
3300 DATA 18,82,28,78,38
,66,48,81
3310 DATA 58,75,68,66,78
,78,88,82
3320 DATA 17,80,27,80,37
,80,47,80
3330 DATA 57,80,67,80,77
,80,87,80
3340 DATA 16,46,26,46,36
,46,46,46
3350 DATA 56,46,66,46,76
,46,86,46
3360 DATA 15,46,25,46,35
,46,45,46
3370 DATA 55,46,65,46,75
,46,85,46
3380 DATA 14,46,24,46,34
,46,44,46
3390 DATA 54,46,64,46,74
,46,84,46
```

```
3400 DATA 13,46,23,46,33
,46,43,46
3410 DATA 53,46,63,46,73
,46,83,46
3420 DATA 12,112,22,112,
32,112,42,112
3430 DATA 52,112,62,112,
72,112,82,112
3440 DATA 11,114,21,110,
31,98,41,113
3450 DATA 51,107,61,98,7
1,110,81,114
3460 REM POTENTIAL MOVES
3470 RESTORE 3520
3480 REM KNIGHT
3490 FOR Z=1 TO 8
3500 READ N(Z)
3510 NEXT Z
3520 DATA 19,-19,21,-21,
-8,8,12,-12
3530 RESTORE 3580
3540 REM ROOK
3550 FOR Z=1 TO 28
3560 READ R(Z)
3570 NEXT Z
3580 DATA 10,20,30,40,50
,60,70
3590 DATA -1,-2,-3,-4,-5
,-6,-7
3600 DATA -10,-20,-30,-4
0,-50,-60,-70
3610 DATA 1,2,3,4,5,6,7
3620 RESTORE 3670
3630 REM BISHOP
3640 FOR Z=1 TO 28
3650 READ B(Z)
3660 NEXT Z
3670 DATA -11,-22,-33,-4
4,-55,-66,-77
3680 DATA 11,22,33,44,55
,66,77
3690 DATA 9,18,27,36,45,
54,63
3700 DATA -9,-18,-27,-36
,-45,-54,-63
```

```
3710 RESTORE 3580
3720 REM QUEEN
3730 FOR Z=1 TO 56
3740 READ Q(Z)
3750 NEXT Z
3760 RESTORE 3810
3770 REM KING
3780 FOR Z=1 TO 8
3790 READ K(Z)
3800 NEXT Z
3810 DATA 1,-1,11,-11,9,
-9,10,-10
3820 FOR Z=1 TO 64
3830 READ S(Z)
3840 NEXT
3850 DATA 46,56,36,66,47
,57,45,55
3860 DATA 37,67,35,65,28
,78,27,77
3870 DATA 44,54,26,76,38
,68,17,87
3880 DATA 18,88,34,64,25
,75,16,86
3890 DATA 48,24,74,15,85
,14,84,43
3900 DATA 53,33,63,23,73
,52,42,62
3910 DATA 32,83,13,72,22
,12,82,41
3920 DATA 51,31,61,21,71
,11,81,58
3930 FOR Z=1 TO 10:SOUND
 Z,1:SOUND28-Z,1:NEXT
3940 RETURN
3950 REM EXCHANGE
3960 FOR Z=11 TO 88:Z(Z)
=A(Z):NEXT
3970 FOR Z=11 TO 88:X=Z-
10*INT(Z/10)
3980 IF X=0 OR X=9 THEN
4000
3990 A(Z)=Z(Z+9-X*2)
4000 NEXT
4010 FOR Z=11 TO 88:M=A(
Z)
```

```
4020 IF M>=B THEN A(Z)=A
(Z)+PB-P
4030 IF M<=RB AND M>=BB
THEN A(Z)=A(Z)-PB+P
4040 NEXT
4050 RETURN
4060 LPRINT TAB(6);"■ABC
DEFGH■"
4070 FOR X=8 TO 1 STEP -
1
4080 LPRINT TAB(3);" ";X
;
4090 FOR Y=10 TO 80 STEP
 10
4100 LPRINT CHR$(A(X+Y))
;
4110 NEXT Y
4120 LPRINT X
4130 NEXT X
4140 LPRINT TAB(6);"■ABC
DEFGH■"
4150 LPRINT:LPRINT:LPRIN
T:LPRINT
4160 RETURN
4170 ON (INT(RND(1)*4)+1
) GOSUB 4190,4200,4230,4
260
4180 SOUND 1,1:SOUND 2,1
:SOUND 1,1
4190 RETURN
4200 PRINT "Well done"
4210 GOSUB 4290
4220 RETURN
4230 PRINT "Good move"
4240 GOSUB 4290
4250 RETURN
4260 PRINT "Got me"
4270 GOSUB 4290
4280 RETURN
4290 FOR I=1 TO 100:NEXT
:RETURN
```

Now here is the truncated version of that program:

```
10 REM DANCING BEAR
    CHESS (ii)
40 DEFINT A-Z
50 GOSUB 3140
60 GOTO 90
70 GOSUB 2760
80 GOSUB 3050
90 GOSUB 2760
100 MM=0
110 INPUT A$
120 IF A$="X" THEN GOSUB
 3960:GOTO 90
130 IF A$="P" THEN GOSUB
 4060
160 FOR Z=1 TO 16:T(Z)=0
:NEXT:U=0
200 FOR Q=1 TO 64:IF A(S
(Q))>=BB AND A(S(Q))<=RB
 THEN U=U+1:T(U)=S(Q):IF
A(S(Q))=KB THEN KM=S(Q)
210 NEXT Q:IF U<3 THEN 2
390
230 GOTO 740
240 FOR Q=1 TO U:IF T(Q)
=KB THEN T(Q)=T(U):T(U)=
KB
260 NEXT Q
270 Q=INT(RND(1)*3)*((A$
="C")+1)
280 Q=Q+1
290 Z=T(Q):GOSUB 350
310 IF MM=1 THEN GOSUB 2
690:GOTO 70
330 IF Q<U THEN 280
340 GOTO 2550
350 IF A(Z)=QB THEN GOSU
B 1020
360 IF A(Z)=RB THEN GOSU
B 1290
370 IF A(Z)=BB THEN GOSU
B 1550
380 IF A(Z)=NB THEN GOSU
B 1800
```

```
390 IF A(Z)=PB THEN GOSU
B 2410
400 RETURN
410 IF A(X)=107 THEN PRI
NT "CHECK!":Q=Q+1:GOTO29
0
430 IF X+9>88 THEN 450
440 IF A(X+9)<83 AND A(X
+9)>65 AND RND(1) <0.96
THEN RETURN
450 IF X-11<11 THEN 480
460 IF A(X-11)<83 AND A(
X-11)>65 AND RND(1) <0.9
6 THEN RETURN
480 AD=0
490 AY=1
500 AX=X+Q(AY+AD)
510 IF AX<11 OR AX>88 TH
EN 540
520 AP=A(AX)
530 IF AP=Q OR AP=R AND
RND(1)>0.8 OR AP=B AND R
ND(1)>0.5 THEN RETURN
540 AY=AY+1
550 IF AY<8 THEN 500
560 AD=AD+7
570 IF AD<56 THEN 490
590 AY=1
600 AX=X+N(AY)
610 IF AX<11 OR AX>88 TH
EN 630
620 IF A(AX)=N THEN RETU
RN
630 AY=AY+1
640 IF AY<9 THEN 600
660 AY=1
670 AX=X+K(AY)
680 IF AX<11 OR AX>88 TH
EN 700
690 IF (A(AX)=K OR A(AX)
=P) AND RND(1)>0.1 THEN
RETURN
700 AY=AY+1
710 IF AY<9 THEN 670
720 MM=1
730 RETURN
```

```
740 Z=KM
770 Y=0
780 Y=Y+1
790 X=Z+N(Y)
800 IF X<11 OR X>88 THEN
 820
810 IF A(X)=N THEN 1990
820 IF Y<8 THEN 780
840 D=0
850 Y=0
860 X=Z+Q(Y+D)
870 IF X<11 OR X>88 THEN
 920
880 IF A(X)=B OR A(X)=Q
OR A(X)=R THEN 1990
890 IF A(X)<>E THEN 920
900 Y=Y+1
910 IF Y<8 THEN 860
920 D=D+7
930 IF D<56 THEN 860
950 X=Z+11
960 IF X>88 THEN 980
970 IF A(X)=P THEN 1990
980 X=Z-11
990 IF X<11 THEN 240
1000 IF A(X)=P THEN 1990
1010 GOTO 240
1020 D=0
1040 Y=1
1050 X=Z+Q(Y+D)
1060 IF X<11 OR X>88 THE
N 1120
1070 IF A(X)=42 OR A(X)>
=BB AND A(X)<=RB THEN 11
20
1080 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410:IF MM
<>1 THEN 1120
1090 IF MM=1 THEN RETURN
1100 Y=Y+1
1110 IF Y<8 THEN 1050
1120 D=D+7
1130 IF D<56 THEN 1040
1140 RETURN
```

```
1160 D=0
1170 Y=1
1180 X=Z+Q(Y+D)
1190 IF X<11 OR X>88 THE
N 1250
1200 IF A(X)<>E THEN 125
0
1210 IF RND(1)>0.5 THEN
GOSUB 410:IF MM=0 THEN 1
250
1220 IF MM=1 THEN RETURN
1230 Y=Y+1
1240 IF Y<8 THEN 1180
1250 D=D+7
1260 IF D<56 THEN 1170
1270 RETURN
1290 D=0
1300 Y=1
1310 X=Z+R(Y+D)
1320 IF X<11 OR X>88 THE
N 1380
1330 IF A(X)=42 OR A(X)>
=BB AND A(X)<=RB THEN 13
80
1340 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410:IF MM
=0 THEN 1380
1350 IF MM=1 THEN RETURN
1360 Y=Y+1
1370 IF Y<8 THEN 1310
1380 D=D+7
1390 IF D<28 THEN 1300
1400 RETURN
1420 D=0
1430 Y=1
1440 X=Z+R(Y+D)
1450 IF X<11 OR X>88 THE
N 1510
1460 IF A(X)<>E THEN 151
0
1470 IF RND(1)<0.1 THEN
GOSUB 410
1480 IF MM=1 THEN RETURN
1490 Y=Y+1
1500 IF Y<8 THEN 1440
```

```
1510 D=D+7
1520 IF D<28 THEN 1430
1530 RETURN
1550 D=0
1560 Y=1
1570 X=Z+B(Y+D)
1580 IF X<11 OR X>88 THE
N 1640
1590 IF A(X)=42 OR A(X)>
=BB OR AX<=RB THEN 1640
1600 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410:IF MM
<>1 THEN 1640
1610 IF MM=1 THEN RETURN
1620 Y=Y+1
1630 IF Y<8 THEN 1570
1640 D=D+7
1650 IF D<28 THEN 1560
1660 RETURN
1680 D=0
1690 Y=1
1700 X=Z+B(Y+D)
1710 IF X<11 OR X>88 THE
N 1770
1720 IF A(X)<>E THEN 177
0
1730 IF RND(1)>0.05 THEN
 GOSUB 410:IF MM<>1 THEN
 1770
1740 IF MM=1 THEN RETURN
1750 Y=Y+1
1760 IF Y<8 THEN 1700

1770 D=D+7
1780 IF D<28 THEN 1690

1790 RETURN
1800 Y=1
1820 X=Z+N(Y)
1830 IF X<11 OR X>88 THE
N 1870
1840 IF A(X)=42 THEN 187
0
```

```
1850 IF A(X)>=B AND A(X)
<=R THEN GOSUB 410
1860 IF MM=1 THEN RETURN
1870 Y=Y+1
1880 IF Y<9 THEN 1820
1890 RETURN
1910 Y=0
1920 X=Z+N(INT(RND(1)*8+
1))
1930 IF X<11 OR X>88 THE
N 1920
1940 IF A(X)=42 THEN 192
0
1950 Y=Y+1
1960 IF A(X)=E THEN GOSU
B 410
1970 IF MM=1 OR Y>20 THE
N RETURN
1980 GOTO 1920
1990 YK=1
2010 Z=KM
2020 X=Z+K(YK):X1=X
2030 IF X<11 OR X>88 THE
N 2360
2040 IF A(X)=42 OR A(X)>
65 AND A(X)<83 THEN 2360
2050 IF A(X)>97 AND A(X)
<115 THEN 2360
2060 Z=X
2080 Y=0
2090 Y=Y+1
2100 X=Z+N(Y)
2110 IF X<11 OR X>88 THE
N 2130
2120 IF A(X)=N THEN 2360
2130 IF Y<8 THEN 2090
2150 D=0
2160 Y=1
2170 X=Z+Q(Y+D)
2180 IF X<11 OR X>88 THE
N 2230
2190 IF A(X)=B OR A(X)=Q
 OR A(X)=R THEN 2360
2200 IF A(X)<>E THEN 223
0
2210 Y=Y+1
2220 IF Y<8 THEN 2170
```

```
2230 D=D+7
2240 IF D<56 THEN 2170
2260 X=Z+11
2270 IF X>88 THEN 2290
2280 IF A(X)=P THEN 2360
2290 X=Z-11
2300 IF X<11 THEN 2320
2310 IF A(X)=P THEN 2360
2320 X=X1:Z=KM
2330 MM=1
2340 GOSUB 2690:GOTO 70
2360 YK=YK+1
2370 Z=KM
2380 IF YK<9 THEN 2010
2390 PRINT "I CONCEDE, C
HAMP!":END
2410 X=Z+9
2430 IF A(X)>=B AND A(X)
<=R THEN MM=1:IF A(X)=P
AND RND(1)<0.2 THEN MM=0
2440 IF MM=1 THEN RETURN
2450 IF Z=12 THEN RETURN
2460 X=Z-11
2470 IF A(X)>=B AND A(X)
<=R THEN MM=1:IF A(X)=P
AND RND(1)<0.2 THEN MM=0
2480 RETURN
2500 IF Z-10*(INT(Z/10))
=7 AND A(Z-1)=E AND A(Z-
2)=E AND(A(Z-13)=E OR A(
Z-13)=42) AND (A(Z+7)=E
OR A(Z+7)=42)THEN X=Z-2:
MM=1:RETURN
2510 IF A(Z-1)=E AND A(Z
-12)<98 AND A(Z+8)<98 TH
EN X=Z-1:MM=1:RETURN
2520 IF RND(1)<0.1 AND A
(Z-1)=E THEN X=Z-1:MM=1
2530 RETURN
2550 Q=INT(RND(1)*RND(1)
*9):IF Q>U THEN 2550
2570 Q=Q+1
2580 Z=T(Q)
2590 IF A(Z)=PB THEN GOS
UB 2500
```

```
2600 IF A(Z)=NB THEN GOS
UB 1910
2610 IF A(Z)=BB THEN GOS
UB 1680
2620 IF A(Z)=RB THEN GOS
UB 1420
2630 IF A(Z)=QB THEN GOS
UB 1160
2640 IF A(Z)=KB THEN GOS
UB 1990
2650 IF MM=0 AND Q<U THE
N 2570
2660 IF MM=1 THEN GOSUB
2690
2670 IF MM=1 THEN GOTO 7
0
2680 GOTO 2390
2690 A(X)=A(Z):A(Z)=E
2720 PRINT "I moved from
"
2730 PRINT CHR$(INT(Z/10
)+64);(Z MOD 10);" to  "
;CHR$(INT(X/10)+64);(X M
OD 10)
2750 RETURN
2760 IF MM=0 THEN 2790
2780 A$=INKEY$:IFA$<>"Q"
 THEN SOUND RND(1)*50,1:
GOTO 2780
2790 CLS:GOSUB 2890
2800 FOR X=8 TO 1 STEP -
1
2810 PRINT TAB(3);"■";X;
2820 FOR Y=10 TO 80 STEP
 10
2830 IF A(Y+1)=PB THEN A
(Y+1)=QB
2840 IF A(Y+8)=P THEN A(
Y+8)=Q
2850 PRINT CHR$(A(X+Y));
2860 NEXT Y:PRINT X:NEXT
 X:MM=0
2890 PRINT TAB(6);"■ABCD
EFGH■";
2910 RETURN
2920 Z=KM
```

```
2930 QK=0
2940 M=Z+K(QK)
2950 IF A(M)=42 OR A(M)>
65 AND A(M)<83 OR MM=0 T
HEN 3000
2970 X=M
2980 KM=X
2990 RETURN
3000 IF QK<8 THEN 2940
3010 IF A$<>"C" THEN RET
URN
3020 GOTO 2390
3050 PRINT:INPUT "FROM (
LETTER,NO)";A$
3060 PRINT A$;" TO";
3070 INPUT B$
3080 X=10*(ASC(A$)-64)+V
AL(RIGHT$(A$,1))
3090 Y=10*(ASC(B$)-64)+V
AL(RIGHT$(B$,1))
3100 IF A(Y)>=75 AND A(Y
)<=82 THEN GOSUB 4170
3110 A(Y)=A(X):A(X)=46:R
ETURN
3140 RANDOMIZE VAL(RIGHT
$(TIME$,2))
3180 DIM A(99),R(28),B(2
8),N(8),Q(56),K(8),Z(88)
,S(64),T(16)
3200 P=112:R=114:N=110:B
=98:Q=113:K=107:E=46
3220 PB=80:RB=82:NB=78:B
B=66:QB=81:KB=75
3230 FOR Z=1 TO 99:A(Z)=
42:NEXT Z
3260 FOR Z=1 TO 64:READ
X:READ Y:A(X)=Y:NEXT Z
3300 DATA 18,82,28,78,38
,66,48,81
3310 DATA 58,75,68,66,78
,78,88,82
3320 DATA 17,80,27,80,37
,80,47,80
3330 DATA 57,80,67,80,77
,80,87,80
```

```
3340 DATA 16,46,26,46,36
,46,46,46
3350 DATA 56,46,66,46,76
,46,86,46
3360 DATA 15,46,25,46,35
,46,45,46
3370 DATA 55,46,65,46,75
,46,85,46
3380 DATA 14,46,24,46,34
,46,44,46
3390 DATA 54,46,64,46,74
,46,84,46
3400 DATA 13,46,23,46,33
,46,43,46
3410 DATA 53,46,63,46,73
,46,83,46
3420 DATA 12,112,22,112,
32,112,42,112
3430 DATA 52,112,62,112,
72,112,82,112
3440 DATA 11,114,21,110,
31,98,41,113
3450 DATA 51,107,61,98,7
1,110,81,114
3490 FOR Z=1 TO 8:READ N
(Z):NEXT Z
3520 DATA 19,-19,21,-21,
-8,8,12,-12
3550 FOR Z=1 TO 28:READ
R(Z):NEXT Z
3580 DATA 10,20,30,40,50
,60,70
3590 DATA -1,-2,-3,-4,-5
,-6,-7
3600 DATA -10,-20,-30,-4
0,-50,-60,-70
3610 DATA 1,2,3,4,5,6,7
3640 FOR Z=1 TO 28:READ
B(Z):NEXT Z
3670 DATA -11,-22,-33,-4
4,-55,-66,-77
3680 DATA 11,22,33,44,55
,66,77
3690 DATA 9,18,27,36,45,
54,63
```

```
3700 DATA -9,-18,-27,-36
,-45,-54,-63
3710 RESTORE 3580
3730 FOR Z=1 TO 56:READ
Q(Z):NEXT Z
3780 FOR Z=1 TO 8:READ K
(Z):NEXT Z
3810 DATA 1,-1,11,-11,9,
-9,10,-10
3820 FOR Z=1 TO 64:READ
S(Z):NEXT Z
3850 DATA 46,56,36,66,47
,57,45,55
3860 DATA 37,67,35,65,28
,78,27,77
3870 DATA 44,54,26,76,38
,68,17,87
3880 DATA 18,88,34,64,25
,75,16,86
3890 DATA 48,24,74,15,85
,14,84,43
3900 DATA 53,33,63,23,73
,52,42,62
3910 DATA 32,83,13,72,22
,12,82,41
3920 DATA 51,31,61,21,71
,11,81,58
3940 RETURN
3960 FOR Z=11 TO 88:Z(Z)
=A(Z):NEXT
3970 FOR Z=11 TO 88:X=Z-
10*INT(Z/10)
3980 IF X=0 OR X=9 THEN
4000
3990 A(Z)=Z(Z+9-X*2)
4000 NEXT
4010 FOR Z=11 TO 88:M=A(
Z)
4020 IF M>=B THEN A(Z)=A
(Z)+PB-P
4030 IF M<=RB AND M>=BB
THEN A(Z)=A(Z)-PB+P
4040 NEXT:RETURN
4060 GOSUB 4140
```

```
4070 FOR X=8 TO 1 STEP -
1
4080 LPRINT TAB(3);" ";X
;
4090 FOR Y=10 TO 80 STEP
 10
4100 LPRINT CHR$(A(X+Y))
;
4110 NEXT Y
4120 LPRINT X
4130 NEXT X
4140 LPRINT TAB(6);"■ABC
DEFGH■"
4160 RETURN
4170 ON (INT(RND(1)*4)+1
) GOSUB 4190,4200,4230,4
260
4180 SOUND 1,1:SOUND 2,1
:SOUND 1,1
4190 RETURN
4200 PRINT "Well done"
4210 GOSUB 4290
4220 RETURN
4230 PRINT "Good move"
4240 GOSUB 4290
4250 RETURN
4260 PRINT "Got me"
4270 GOSUB 4290
4280 RETURN
4290 FOR I=1 TO 100:NEXT
:RETURN
```

CHAPTER SEVENTEEN - IMPROVING YOUR PROGRAMMING TECHNIQUE

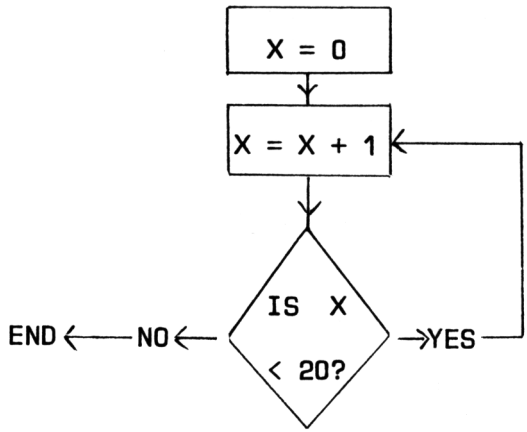
There will come a point in your development as a programmer when you'll have mastered the use of much of BASIC, and can now concentrate on writing better programs; programs which work after relatively little debugging, which are easy for others to understand and operate, and which are written logically and elegantly. Some hints as to how you can approach these aims are given in this chapter.

Your programs will be more likely to run first time if they are planned out carefully before you start entering code (and code is a synonym for program) into your computer.

A good way to start is to use a diagram which is often called a 'flow chart'. A flow chart is a series of boxes and other shapes, joined by lines, which show the flow of action and decision-making within the computer while the program is running.

The shapes used are not too important, and I suggest you stick to just two: a rectangle for most actions the computer must carry out, with a diamond shape each time the computer has to make a decision. The corners of the diamond can be used — as you can see in the diagram at the top of the following page — to cater for the alternatives facing the computer.

The diagram shows the flow chart of a program which sets the variable X equal to zero, then adds one to it. The value of X is checked. If X is founded to be less than 20, the program goes back to add one to X again. This continues until the value of X equals 20.



One advantage of using a flow chart is that you do not get locked, at an early stage of your work, into the peculiarities of the BASIC, its weaknesses and strengths, which is available on your computer. Instead, you concentrate on what you want to do. Of course, you may have to modify your plans slightly to accept limitations dictated by your BASIC, but you do not have to bend to these limitations (which may be, in part, imaginary) at the start of the process.

A flow chart is 'universal'. The same flow chart can be used as the basis of a program written on a computer furnished with a completly different BASIC, or even for a computer which has a language other than BASIC on board.

A flow chart models the flow of action and logic within a program, and is therefore very useful for picking up potential bugs at the earliest stages.

You may, for example, find that one condition for the program will test will never be fulfilled, possibly leading to the program being trapped in an infinite loop. Other parts of the code may be by-passed completely, because the condition which triggers entry into that part of the code will never be met.

Once you've devised a flow chart for your program, and you've run through it mentally a few times, so that the most obvious bugs are removed, you should reduce the flow chart to a series of subroutine calls. Although it seems pretty silly to do this for a simple program like our "SET X EQUAL TO ZERO, ADD ONE, CHECK IF IT'S LESS THAN 20" program, this method comes into its own with complex programs.

You start the program with a series of subroutine calls, with each action of the program being looked after by a separate subroutine. Then, if the steps within a program have to be performed several times in a particular sequence, the series of subroutine calls can be cycled through — over and over again — until a particular condition is met which signals the end of the run.

You'll recognize how useful this approach to programming can be when you get to the debugging stage of your program. If there is a bug, it is likely to be within a single subroutine, so it will be relatively easy to pin down the subroutine which contains the bug, rather than having to work right through the program trying to track it down.

Working with subroutine 'modules' in this way allows you to test sections of the program in isolation,

even before the entire program is finished. I'll try to make this statement clear by showing you the first part of a typical program to play Checkers. The program could start like this:

```
 10 REM CHECKERS
 20 GOSUB 9000: REM INITIALISE VARIABLES
 30 GOSUB 8000: REM PRINT BOARD
 40 GOSUB 7000: REM ACCEPT PLAYER MOVE
 50 GOSUB 8000: REM PRINT BOARD
 60 GOSUB 5000: REM COMPUTER MAKES MOVE
 70 IF (human has not won) AND (computer
       has not won) THEN 30
 80 IF (computer has won) THEN PRINT "I WIN"
 90 IF (human has won) THEN PRINT "YOU WIN"
100 END
```

You could quite a bit of this program running, and tested (such as the initialisation routine, printing the board and accepting the player's move), before you even turned your attention to how on earth you were going to get the computer to make its move.

You would then know — for example — that you would not need to waste any extra thought on whether or not an error in the board-printing routine was the cause of odd output. Having tested the board subroutine and the player move routine, you'd know that the error must be in the subroutine between lines 5000 and 6999, the subroutine in which the computer makes its move.

All you need to do is put a single PRINT statement, such as "THIS IS COMPUTER MAKING A MOVE" followed by a RETURN for incomplete subroutines, knowing that the program will accept that, and demonstrate the

direction the program flow is following, even if whole sections of code have not yet been written.

In general, I'd advise you to use this system of using a 'master loop' of subroutine calls within which you will 'hold' the entire program. If you look at the CHECKERS program in this book, you'll see that most of the action is held within the subroutine calls which fall between lines 110 and 140. You'll see that line 110, GOSUB 680, directs action to line 680 where the computer prints up the board and line 120, GOSUB 850, does the same with the routine which accepts the player's move.

I suggest you try and do as much writing of the program as possible **before** you turn the computer on, even though there is a great temptation to dive straight into the computer and start punching in code. You'll find that the discipline of writing it out by hand in advance will serve you in good stead and should, in the long run, produce a better program than might otherwise have been the case. Overall, you'll probably end up spending up less time on the program working in this way than you would if you began the process sitting at the computer keyboard.

It took me a while to learn this lesson. Although I had read suggestions along the lines of 'work out exactly what you're going to enter **before** you start at the computer' in several books, I tended to just jump right in without much prior thought.

Although I worked out rough flow charts, and had an idea what sort of display organisation I wanted, I certainly did not write much program out on paper

before starting at the computer. Then, I once found myself stuck for a two-week period without a computer and ideas for several programs just itching to be written. I had to write them out in an exercise book.

The relative ease with which the programs were debugged when they were eventually entered into the computer, and the complexity of the programs I wrote in this way (including my first Chess program) convinced me that this was the way I would work from then on. It is amazing how much cleaner a program can be if all the rough working out is done on paper, rather than on the computer screen.

At present, I tend to write the major 'call subroutines loop' first of all, but without line numbers, so the program contains lines like GOSUB PRINT BOARD and GOSUB INITIALISE. Next, I write each module (or subroutine) on a separate sheet of paper. Then, when the major subroutines have been written I shuffle them into an order which seems most logical.

All this, of course, occurs before any line numbers are written in. The subroutine modules are put in an order which ensures that the program structure is as logical as possible. I use arrows to indicate the destination of GOTO's within a module, and names for subroutine, as suggestion in the major loop. Later on, when the program has started to assume a firm shape, the lines are numbered (I always work in tens, starting at line 10) and the relevant GOTO and GOSUB destinations are added.

All programs have an 'end condition' at which point computation stops. It is worth putting a test for this end condition as part of the GOTO which sends the program back to start cycling through the major subroutine loop. This ensures that the cycle will continue until a particular condition is met, at which point the program 'falls through' that GOTO and continues on to the lines which signal the end of the program.

EXPLICIT INPUT PROMPS AND PRINT STATEMENTS

It is very useful, when writing a program, to keep in mind how the program will appear to a stranger when it is run by him or her for the first time. If there is an input prompt required, it is far more useful if the program prints up something like "HOW MANY HOURS HAS THE EMPLOYEE WORKED THIS WEEK?" instead of just "INSERT HOURS?" or the almost useless lone question mark.

The same suggestion applies to print output. It is far better that your program is written so that it prints out THE NUMBER OF HOURS WORKED ON FULL PAY THIS WEEK IS 27, rather than HOURS, FULL: 27 or an unsupported 27. Of course, providing explicit input prompts and output PRINT statements consumes memory as well as typing time when entering the program, but the contribution they make to the final program means the trouble involved is well worth it.

REM STATEMENTS

While exact PRINT statements and input prompts will help a person running a program make sense of it, REM statements can help make the program clear to

those who are examining the listing for the first time. REM statements (which are, as you know, ignored by the computer when a program is run) should be used to help illuminate the flow of logic within the program, and what is happening in certain places within it. This is especially important in parts of the program where decisions are made, or calculations are carried out.

Not only can REM statements be used to explain different parts of the code, but they can also be used to provide visual 'breaks', so that the various blocks of code which carry out certain tasks are visually separated from the rest of the program. A blank REM statement (the word REM standing alone in a program line) can be used for this. A row of asterisks is an effective alternative.

VARIABLES

It is worth considering the use of explicit names for variables, using either the word in full (such as HOURS as a variable name in a payroll program for hours worked) or an abbreviated version of this (such as HR) which has a fairly obvious meaning. You'll discover that this makes it easy, when working on a program, to keep track of your variables. This will help you with the initial debugging and later on as well if you need to improve or extend the original program.

Explicit variable names also help make your code more 'transparent' so other programmers can work out what the various parts of the program are intended to do. You'll also find it a great help to yourself when you return to the program at a later date. It

is surprising how code which seemed incredibly clear it terms of its purpose when you entered it, becomes exceedingly dense when you return to it after a long break.

CHECKING INPUT

Any input entered by the user should be checked by the program before it is accepted to ensure that incorrect data does not cause the program to crash at some future point. Whether you want string, or numeric input, it is often wise to allow for string input, which is first checked to see whether the entered material is acceptable and then, if necessary, the string can be changed into a number.

For example, if the user needs to enter a number between one and nine, a string can be accepted and then checked to ensure that it is not less than "1" amd more than "9" before being changed into a number with VAL.

As well as ensuring that the program will reject invalid input, you should check the program to see all the inputs which it does accept produce sensible answers when processed later on in the program. For example, make sure that your program does not accept zero as a possible number if the computer must later divide by this number. Similarly, if numbers are to be processed by a function, and then the result of this processing used for division, you must check that an apparently valid input does not turn into zero as a result of evaluation by the function.

If the information entered by the user is rejected, and a new input is requested, the program should

ideally point out why the original input was not suitable, or spell out again exactly what is required (such as "ENTER A NUMBER BETWEEN 1 AND 4"). You risk making users angry if input which appears valid is continually rejected without apparent reason.

DOCUMENTATION

The written material which accompanies a program is often called documentation. It is useful for a program to be supported by some documentation, however sketchy. The written information should explain, of course, what the program does, then go on to outline the flow of action within the program. The documentation should alert the user as to the kind of actions which will be required from him or her when running the prohram, and give an indication of the kinds of user input and reaction which will be accepted.

The format of the final output should also be discussed. A list of variable names can be included.

If there are ways in which the program can be developed, extended or improved, suggestions along these lines can be added to the documentation. Written references to any material which will help in understanding the algorithms used, or for giving suggested areas for program development, should also be included.

In many ways, it is reasonable to assume that the job of programming is not finished once the program is done. Without documentation, the job is only three-quarters complete. Documentation finishes the

task, adding a professional stamp to your work which allows the program you've written to be used most effectively.

Possibly the only time extensive documentation is not really required is when the program is 'menu-driven'. A program which uses REM statements extensively may not need very much in the way of documentation, especially if you can include a variable list, as a series of REM statements, at the end of the program.

Generally, however, you'll find it better to document a program externally, rather than rely on REM statements, or the various menu choices, to do the job for you. It is worth trying to write your program documentation so that it would make sense to someone who has not seen the program running.

This person should be able to get a very good idea, just from reading the documentation, of what the program does and how it does it; how it interacts with the user both in terms of accepting information and in presenting the results of its computations to the users; and how the program is organised as a whole.

Documentations for a major program should start with an introduction which quickly explains what is going on, and tells how to use the program. The later parts of the documentation can then discuss the program in greater detail. It is not good practice to force the user to wade through a vast amount of information in order to dig out the vital facts he or she needs to get the program running.

APPENDIX - SUGGESTIONS FOR FURTHER READING

As you've probably discovered by looking in your local bookstore, computer shop or library, the number of computer books is enormous, and growing by the hour. In this appendix, I will not even attempt a comprehensive overview of the field, but will simply list the books I have found personally worthwhile. The list includes books which have taught me about the flexibility and potential of computers. Others have acted as 'idea starters'. These should trigger in you - as they did in me - ideas for applications, programs and exploration.

BASIC BUSINESS SOFTWARE - Brooner, E.G. (Howard W Sams & Co. Inc., Indianapolis, Indiana, 1980)

BASIC COMPUTER PROGRAMS FOR BUSINESS, volume one - Sternberg, Charles D. (Hayden Book Co., Inc., Rochelle Park, New Jersey, 1980)

BASIC COMPUTER PROGRAMS FOR THE HOME - Sternberg, Charles D. (Hayden Book Co., Inc., Rochelle Park, New Jersey, 1980)

BASIC COMPUTER PROGRAMS IN SCIENCE AND ENGINEERING - Gilder, Jules H. (Hayden Book Co., Inc., Rochelle Park, New Jersey, 1980)

BASIC FUN - A KID'S GUIDE TO BASIC PROGRAMMING - Lipscomb, Susan Drake and Zuanich, Margaret Ann (Avon Camelot, New York, 1982)

BASIC, GETTING STARTED - David, William S. (Addison-Wesley Publishing Co., Reading, Massachusetts, 1981)

THE BASIC HANDBOOK - Lien, David A. (Compusoft Publishing, San Diego, California, 1979)

BASIC PROGRAMMER'S NOTEBOOK - Savage, Earl R. (Howard Sams & Co. Inc., Indianapolis, Indiana, 1981)

BASIC WITH STYLE - Nagin, Paul and Ledgard, Henry F. (Hayden Book Co., Inc., Rochelle Park, New Jersey, 1978)

THE BASIC WORKBOOK - Schoman, Kenneth E. Jr. (Hayden Book Co. Inc., Rochelle Park, New Jersey, 1977)

BEGINNER'S GUIDE TO MICROPROCESSORS - Parr, E. A. (Newnes Technical Books, Butterworth, London, 1982)

THE BEGINNER'S GUIDE TO BUYING A PERSONAL COMPUTER - Mansfield, Richard, et al (COMPUTE! Books, Greensboro, North Carolina, 1982)

BUSINESS CALCULATIONS - Bird, J.O., and May, A.J.C. (Longman, New York, 1980)

BUSINESS INFORMATION PROCESSING WITH BASIC - Struble, George (Addison-Wesley Publishing Co., Reading, Massachusetts, 1980)

CHESS AND COMPUTERS - Levy, David (Computer Science Press, Inc., Potomac, Maryland, 1976)

COMPUGUIDE, A CONSUMER'S GUIDE TO SMALL BUSINESS COMPUTERS - Eischen, Martha (dilithium Press, Beaverton, Oregon, 1982)

COMPUTERS FOR EVERYBODY - Miller, Merl and Willis, Jerry (dilithium Press, Beaverton, Oregon, 1982)

CONTINUING BASIC - Gosling, Peter (The MacMillan Press Ltd., Basingstoke, UK, 1980)

THE CP/M HANDBOOK - Zaks, Rodnay (Sybex, Berkely, California, 1980)

THE DAILY TELEGRAPH CALCULATOR BOOK - King, David (Daily Telegraph, London, 1979)

DEVELOPING COMPUTER SOLUTIONS FOR YOUR BUSINESS PROBLEMS - Petersohn, Henry H. (Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982)

80 PRACTICAL TIME-SAVING PROGRAMS FOR THE TRS-80 - Carroll, Charles J. (TAB Books, Inc., Blue Ridge Summit, Pennsylvania, 1982)

FINANCIAL ANALYSIS BY CALCULATOR - Mullish, Henry and Kestenbaum, Richard (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982)

GUIDE TO GOOD PROGRAMMING PRACTICE - Meek, Brian and Heath, Patricia (eds) (John Wiley & Sons, New York, 1980)

INTRODUCTION TO BUSINESS SIMULATION - Frazer Ronald (Reston Publishing Co. Inc., Reston, Virginia, 1977)

HOW DID WE FIND OUT ABOUT NUMBERS? - Asimov, Isaac (Walker and Co., New York, 1973)

HOW TO WRITE AN APPLE PROGRAM - Faulk, Ed (Reston Publishing Co. Inc., Reston, Virginia, 1982)

THE MAKING OF THE MICRO - Evans, Christopher (Victor Gollancz, London, 1981)

THE MARTIN BOOK OF MANAGEMENT CALCULATIONS - de Lisle, Christian (Woodhead-Faulkner Ltd., Cambridge, UK, 1979)

MATHEMATICS FOR PRACTICAL USE - Nielsen, Kaj L. (Barnes and Noble Books, Harper & Row, New York, 1962)

MICROCOMPUTERS AND THE THREE R's - A GUIDE FOR TEACHERS - Doerr, Christine (Hayden Book Company, Inc., Rochelle Park, New Jersey, 1979)

THE MOST POPULAR SUBROUTINES IN BASIC - Tracton, Ken (TAB Books Inc., Blue Ridge Summit, Pennsylvania, 1980)

PERSONAL COMPUTER - HOME, PROFESSIONAL AND SMALL BUSINESS APPLICATIONS - McGlynn, Daniel R. (John Wiley & Sons, New York, 1979)

THE PROGRAMMER'S BOOK OF RULES - Ledin, George Jr., and Ledin, Victor (Lifetime Learning Publications, Belmont, California, 1979)

SIMPLE BASIC PROGRAMS FOR BUSINESS APPLICATIONS - Alonso, J.R.F. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982)

A 60-MINUTE GUIDE TO MICROCOMPUTERS - Hollerbach, Lew (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982)

THE SMALL COMPUTER IN SMALL BUSINESS - Smith, Brian R. (Stephen Greene Press, Brattleboro, Vermont, 1981)

SOME COMMON BASIC PROGRAMS (3rd edition) - Poole, Lon and Borchers, Mary (Osborne/McGraw Hill, Berkeley, California, 1979)

USING MICRO-COMPUTERS IN BUSINESS, A GUIDE FOR THE PERPLEXED - Veit, Stanley S. (Hayden Book Company, Inc., Rochelle Park, New Jersey, 1981)

WHY DO YOU NEED A PERSONAL COMPUTER? - Leventhal, Lance A. and Stafford, Irvin (John Wiley & Sons, Inc., New York, 1981)

APPENDIX - GLOSSARY OF USEFUL WORDS

Accumulator - part of the computer's logic unit which stores the intermediate results of computations

Address - a number which refers to a location, generally in the computer's memory, where information is stored

Algorithm - the sequence of steps used to solve a problem

Alphanumeric - generally used to describe a keyboard, and signifying that the keyboard has alphabetical and numerical keys. A numeric keypad, by contrast, only has keys for the digits one to nine, with some additional keys for arithmetic operations, much like a calculator

APL - this stands for Automatic Programming Language, a language developed by Iverson in the early 1960s, which supports a large set of operators and data structures. It uses a non-standard set of characters

Application software - these are programs which are tailored for a specific task, such as word processing, or to handle mailing lists

ASCII - stands for American Standard Code for Information Exchange. This is an almost universal code for letters, numbers and symbols, which has a number between 0 and 255 assigned to each of these, such as 65 for the letter A

Assembler - this is a program which converts another program written in an assembly language (which is a computer program in which a single instruction, such as ADD, converts into a single instruction for the computer) into the language the computer uses directly

BASIC - stands for Beginner's All-purpose Symbolic Instruction Code, the most common language used on microcomputers. It is easy to learn, with many of its statements being very close to English

Batch - a group of transactions which are to be processed by a computer in one lot, without interruption by an operator

Baud - a measure of the speed of transfer of data. It generally stands for the number of bits (discrete units of information) per second

Benchmark - a test which is used to measure some aspect ofthe performance of a computer, which can be compared to the result of running a similar test on a different computer

Binary - a system of counting in which there are only two symbols, 0 and 1 (as opposed to the ordinary decimal system, in which there are ten symbols, 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9). Your computer 'thinks' in binary

Boolean Algebra - the algebra of decision-making and logic, developed by English mathematician George Boole, and at the heart of your computer's ability to make decisions

Bootstrap - a program, run into the computer when it is first turned on, which puts the computer into the state where it can accept and understand other programs

Buffer - a storage mechanism which holds input from a device such as keyboard, then releases it at a rate which the computer dictates

Bug - an error in a program

Bus - a group of electrical connections used to link a computer with an ancillary device, or another computer

Byte - the smallest group of bits (see bit) which makes up a computer word. Generally a computer is descibed as being 'eight bit' or '16 bit', meaning the word consists of a combination of eight or sixteen zeroes or ones

Central Processing Unit (CPU) - the heart of the computer, where arithmetic, logic and control functions are carried out

Character code - the number in ASCII (see ASCII) which refers to a particular symbol, such as 32 for a space and 65 for the letter 'A'

COBOL - stands for Common Business Orientated Language, a standard programming language, close to English, which is used primarily for business

Compiler - a program which translates a program written in a high level (human-like) language into a machine language which the computer is able to

understand directly

Concatenate - to add (adding two strings together is known as 'concatenation')

CP/M - stands for Control Program/Microcomputer, an almost universal disk operating system developed and marketing by Digital Research, Pacific Grove, California

Data - a general term for information processed by a computer

Database - a collection of data, organised to permit rapid access by computer

Debug - to remove bugs (errors) from a program

Disk - a magnetic storage medium (further described as a 'hard disk', 'floppy disk' or even 'floppy') used to store computer information and programs. The disks resemble, to a limited extent, 45 rpm sound records, and are generally eight, five and a quarter, or three and a half inches in diameter. Smaller 'microdisks' are also available for some systems

Documentation - the written instructions and explanations which accompany a program

DOS - stands for Disk Operating System (and generally pronounced 'doss'), the versatile program which allows a computer to control a disk system

Dot-matrix printer - a printer which forms the letters and symbols by a collection of dots, usually

on an eight by eight, or seven by five, grid

Double-density - adjective used to describe disks when recorded using a special technique which, as the name suggests, doubles the amount of storage the disk can provide

Dynamic memory - computer memory which requires constant recharging to retain its contents

EPROM - stands for Erasable Programmable Read Only Memory, a device which contains computer information in a semi-permanent form, demanding sustained exposure to ultra-violet light to erase its contents

Error messages - information from the computer to the user, sometimes consisting only of numbers or a few letters, but generally of a phrase (such as 'Out of memory') which points out a programming or operational error which has caused the computer to halt program executions

Field - A collection of characters which form a distinct group, such as an indentifying code, a name or a date; a field is generally part of a record

File - A group of related records which are processed together, such as an inventory file or a student file

Firmware - The solid components of a computer system are often called the 'hardware', the programs, in machine-readable form on disk or cassette, are called the 'software', and programs which are hard-wired into a circuit, are called 'firmware'. Firmware can be altered, to a limited extent, by

software in some circumstances

Flag - this is an indicator within a program, with the 'state of the flag' (i.e. the value it holds) giving information regarding a particular condition

Floppy disk - see disk

Flowchart - this is a written layout of program structure and flow, using various shapes, such as a rectangle with sloping sides for a computer action, and a diamond for a computer decision. A flowchart is generally written before any lines of program are entered into the computer

FORTRAN - a high level computer language, generally used for scientific work (from FORmula TRANslation)

Gate - a computer 'component' which makes decisions, allowing the circuit to flow in one direction or another, depending on the conditions to be satisfied

GIGO - acronym for 'Garbage In Garbage Out', suggesting that if rubbish or wrong data is fed into a computer, the result of its processing of such data (the output) must also be rubbish

Global - a set of conditions which effects the entire program is called 'global', as opposed to 'local'

Graphics - a term for any output of computer which is not alphanumeric, or symbolic

Hard copy - information dumped to paper by a printer

Hardware - the solid parts of the computer (see 'software' and 'firmware')

Hexadecimal - a counting system often used by machine code programmers because it is closely related to the number storage methods used by computers, based on the number 16 as opposed to our 'ordinary' number system which is based on 10)

Hex pad - a keyboard, somewhat like a calculator, which is used for direct entry of hexadecimal numbers

High-level langauges - programming languages which are close to English. Low-level languages are closer to those which the computer understands. Because high-level languages have to be compiled into a form which the computer can understand before they are processed, high-level languages run more slowly than do their low-level counterparts

Input - any information which is fed into a program during execution

I/O - stands for Input/Output port, a device the computer uses to communicate with the outside world

Instruction - an element of programming code, which tells the computer to carry out a specific task. An instruction in assembler language, for example, is ADD which (as you've probably guessed) tells the computer to carry out an addition

Interpreter - converts the high-level ('human-understandable') program into a form which the computer can understand

Joystick - an analogue device which feeds signal into a computer which is related to the position which the joystick is occupying; generally used in games programs

Kilobyte - the unit of language measurement; one kilobyte (generally abbreviated as K) equals 1024 bits (see bit)

Line printer - a printer which prints a complete line of characters at one time

Low-level language - a language which is close to that used within the computer (see high-level language)

Machine language - the step below a low-level language; the language which the computer understands directly

Mainframe - the term for 'giant' computers such as the IBM 307. Computers are also classed as mini-computer and microcomputer (such as the computer you own)

Memory - the device or devices used by a computer to hold information and programs being currently processed, and for the instruction set fixed within a computer which tells it how to carry out the demands of the program. There are basically two types of memory (see RAM and ROM)

Microprocessor - the 'chip' which lies at the heart of your computer. This does the 'thinking'

Modem - stands for MOdulator/DEModulator, and is a device which allows one computer to communicate with another via the telephone

Monitor - (a) a dedicated television-screen for use as a computer display unit, contains no tuning apparatus; (b) the information within a computer which enables it to understand and execute program instructions

Motherboard - a unit, generally external, which has slots to allow additional 'boards' (circuits) to be plugged into the computer to provide facilities (such as high-resolution graphics, or 'robot control') which are not provided with the standard machine

Mouse - a control unit, slightly smaller than a box of cigarettes, which is rolled over the desk, moving an on-screen cursor in parallel to select options and make decisions within a program. 'Mouses' work either by sensing the action of their wheels, or by reading a grid pattern on the surface upon which they are moved

Network - a group of computers working in tandem

Numeric pad - a device primarily for entering numeric information into a computer, similar to a calculator

Octal - a numbering system based on eight (using the digits 0, 1, 2, 3, 4, 5, 6 and 7)

On-line - device which is under the direct control of the computer

Operating system - this is the 'big boss' program or series of programs within the computer which controls the computer's operation, doing such things as calling up routines when they are needed and assigning prioritories

Output - any data produced by the computer while it is processing, whether this data is displayed on the screen or dumped to the printer, or is used internally

Pascal - a high level language, developed in the late 1960s by Niklaus Wirth, which encourages disciplined, structured programming

Port - an output or input 'hole' in the computer, through which data is transferred

Program - the series of instructions which the computer follows to carry out a predetermined task

PILOT - a high level language, generally used to develop computer programs for education

RAM - stands for Random Access Memory, and is the memory on board the computer which holds the current program. The contents of RAM can be changed, while the contents of ROM (Read Only Memory) cannot be changed under software control

Real-time - when a computer event is progressing in line with time in the 'real world', the event is said to be occurring in real time. An example would be a program which showed the development of a colony of bacteria which developed at the same rate that such a real colony would develop. Many games,

which require reactions in real time, have been developed. Most 'arcade action' programs occur in real time

Refresh - The contents of dynamic memories (see memory) must receive periodic bursts of power in order for them to maintain their contents. The signal which 'reminds' the memory of its contents is called the refresh signal

Register - a location in computer memory which holds data

Reset - a signal which returns the computer to the point it was in when first turned on

ROM - see RAM

RS-232 - a standard serial interface (defined by the Electronic Industries Association) which connects a modem and associated terminal equipment to a computer

S-100 bus - this is also a standard interface (see RS-232) made up of 100 parallel common communication lines which are used to connect circuit boards within micro-computers

SNOBOL - a high level language, developed by Bell Laboratories, which uses pattern recognition and string manipulation

Software - the program which the computer follows (see firmware)

Stack - the end point of a series of events which

are accessed on a last in, first out basis

Subroutine - a block of code, or program, which is called up a number of times within another program

Syntax - as in human languages, the syntax is the structure rules which govern the use of a computer language

Systems software - sections of code which carry out administrative tasks, or assist with the writing of other programs, but which are not actually used to carry out the computer's final task

Thermal printer - a device which prints the output from the computer on heat-sensitive paper

Time-sharing - this term is used to refer to a large number of users, on independent terminals, making use of a single computer, which divides its time between the users in such a way that each of them appears to have the 'full attention' of the computer

Turnkey system - a computer system (generally for business use) which is ready to run when delivered, needing only the 'turn of a key' to get it working

Volatile memory - a memory device which loses its contents when the power supply is cut off

Word processor - a dedicated computer (or a computer operating a word processing program) which gives access to an 'intelligent typewriter' with a large range of correction and adjustment features

APPENDIX - ERROR REPORTS

/0 11 - attempt to divide by zero
AO 52 - file of specified number is already open
BD 58 - data format in file is incorrect
BF 51 - file mode is incorrect
BN 50 - a file number is incorrect
BO 61 - buffer overflow
BS 9 - bad subscript in array (outside size of array defined, or wrong number of subscripts used)
CN 17 - can't continue
DD 10 - duplicate definition
DS 56 - direct statement in file
DU 60 - device unavailable
FC 5 - illegal function call
FD 55 - bad file descriptor
FN 23 - FOR without NEXT
ID 12 - attempt to enter a statement which is illegal in the direct mode
IE 54 - input past end, all data in file has been read
IO 53 - device I/O error
IU 59 - device in use
LS 15 - string too long (>256 characters)
MO 22 - missing operand
NE 63 - specified file does not exist
NF 1 - NEXT without a FOR
NO 57 - file not open
NR 19 - no RESUME statement in error-trapping routine
OD 4 - out of DATA
OM 7 - out of memory
OS 14 - out of string space
OV 6 - overflow
PP 62 - protected program

RG 3 - RETURN without GOSUB
RW 20 - RESUME without error
SN 2 - Syntax error
ST 16 - string formula is too complex
TM 13 - mismatch in the type of variable
UF 18 - a USR function is not defined
UL 8 - an error in the line number
UP 21 - there is an error with an undefined error code
WE 24 - WHILE without WEND (disk BASIC only)
WH 25 - WEND without WHILE (disk BASIC only)

MAKING THE MOST OF YOUR HX20

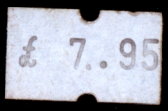
Over the years, we have seen computers shrink from being vast machines that occupied an entire suite of rooms, to exciting hand-held devices like the Epson HX20.

In this book, Tim Hartnell, author of over 30 top-selling computer books, shows you how to tap the immense power of your HX20.

In easy-to-follow steps, you'll be lead through programming the HX20 from first principles.

You'll also be shown what to expect from commercial software - business and personal - and shown how to develop your own programs.

A number of major programs, including MINICALC, PERSONAL FINANCE, REVERSI, CHECKERS and even CHESS are included in the book, to demonstrate how powerful is the genie you now have at your command.



ANOTHER GREAT BOOK
FROM
INTERFACE PUBLICATIONS

Making the most of your HX20

Hartnell

Interface