6809 MACHINE CODE PROGRAMMING

DAVID BARROW

6809 Machine Code Programming

6809 Machine Code Programming

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

First published in Great Britain by
Granada Publishing 1984

British Library Cataloguing in Publication Data
Barrow, David
6809 machine code programming.
1. Motorola 6809 (Microprocessor)——
Programming 2. Machine codes (Electronic
computer)
I. Title
001.64'25 QA76.8.M689

ISBN 0 246 12532-2

Typeset by V & M Graphics Ltd, Aylesbury, Bucks
Printed and bound in Great Britain by
Mackays of Chatham, Kent

Contents

Preface

The MC6809 was released to the world as the 'programmer's dream machine'. In fact Motorola, the manufacturers of the 6809, did a great deal of research to discover what the users of its predecessor, the 6800, wanted as their ideal computer. The 6809 was designed around their findings. It has a set of instructions that is more comprehensive and logically complete than any other processor in its class. For the skilled programmer, it is indeed a 'dream' of a machine.

For the newcomer to machine code, faced with 139 cryptically named instruction forms, the dream can be a nightmare. Learning to use the instructions effectively seems a near impossible task. How could anyone but a TEFAL scientist remember the different actions of every single instruction, let alone string them together to produce a program? Well, for a start there is more to programming than just knowing what instructions will or will not do. Important though that knowledge is, it can wait; the key to successful machine code programming is to have the right attitude of mind in the first place. And that is the approach I have taken in this book.

The introductory chapter tries to get past the concept of the microcomputer as nothing but a cold-blooded perfectionist with a heart of silicon. Computers are designed by people and the basic principles of their operation is not so alien to the way that we humans work as you might think. Seeing the computer as a microcosm which echoes human organisational methods is an essential first step in being able to use it with complete confidence. In the second chapter I take a look at how programs ought to be written to remove most of the mind-bending. It is an exercise which doesn't require a thorough knowledge of machine code but does need an understanding of what both computers and people can and cannot do.

Except for one chapter which deals with hardware, this book is a collection of program subroutines which I think you will find extremely useful, particularly for games programs. Though not covering all the 6809's instructions, the routines do include the vast majority of instruction forms and addressing modes – probably all that you will need to use in normal user software programming. I have documented the routines far more extensively than the sort of comments usually found in assembler listings to show how instructions are used to carry out clearly defined tasks. Reading and trying out fully explained code sequences is a more efficient and interesting way to learn machine code than attempting to understand how each instruction works in isolation. All the routines are written primarily for the Dragon 32 but most of them should work on other 6809 computers with little or no change.

This book would not have been written without the help of a great many people to whom I give my thanks – in particular to Richard Miles of Granada Publishing for his confidence in the book, to Alan Tootill who propelled me in this direction a long time ago, to my wife Chrissie for patiently rereading the manuscript at each minor revision, and not least to Karl and Sibelind who assisted my concentration the most by keeping unnaturally quiet. You can shout your heads off now, kids!

David Barrow

Chapter One
Into Machine Code

The psychologist G. A. Miller coined the phrase 'The Magical Number Seven, Plus or Minus Two' to describe the limitations of the human brain in recognising and retaining discrete items of information. It seems that whatever sense is involved – sight, sound, touch, taste, etc. – human beings are pretty accurate at dealing with small quantities of data. When asked to cope with more than about seven items at a time, however, we come unstuck and begin to make mistakes.

The concept of *at a time* is quite flexible and may refer to either simultaneous or sequential presentation of data. We can tell at a glance if a telephone number contains four, five or six digits and we may even be able to hold the image in our minds long enough to *read* the number from it. We can certainly remember a six-digit telephone number that is read out to us over, say, a three-second period but we may have difficulty remembering one read out over a thirty-second time span – especially if other things are going on at the same time.

Six-digit telephone numbers are quite easy but how would you get on with twenty-digit numbers? Could you remember the twenty digits for the forty seconds or so it would take to dial? Actually, you might not find this as difficult as you think. How do you see the telephone number 362436? As *three-six-two-four-three-six* or as *three-six ... two-four ... three-six*? If it was your girlfriend's number you might even see it as *thirty-six ... twenty-four ... thirty-six*. She, no doubt, has a similar method of remembering your number since the human brain has a trick or two up its cortex to get round the 7+/−2 limit!

Trick 1 is to group items together to form larger units but – and this is the important point – fewer of them. Trick 2 is to form associations between – well, between anything and everything that can be linked. The twenty digits of our hypothetical telephone number would get grouped into 2, 3, 4 or 5 digit sequences. The patterns inherent in the groups and our own private associations would keep the number in our minds long enough for us to dial it. After an initial period of inventing more suitable names for British Telecom, our 7+/−2 unit capacity brains would deal fairly efficiently with the larger numbers.

The way we function as thinking beings has a bearing on machine code programming. Computers are designed by people and the code that controls their operation is also ultimately a product of the human mind. Hardware organisation to some extent mirrors our mental capacity. Programming, whether in high level languages that sound like English or in machine code, is the act of translating our mental processes into a form which can be used by machines and (ought to be) readily understood by other people. Only by perceiving clearly the similarities and differences between people and computers/programs – and the abilities and limitations of both – can we interact effectively.

A question of address

Would it be a good idea if your house number, street, postal district, etc. formed an all-digit postcode that was also your telephone number, National Insurance number, credit card number, and so on? It would certainly alleviate the pressure on your memory but just think of the problems caused by such a system when you moved house!

We all need to be numbered or addressed in a multitude of ways for different purposes. Many of the ways say nothing about our location. Bank account numbers or National Insurance numbers bear no relation to our home address. Some of the ways in which we are addressed, however, give varying amounts of information regarding location. Your full telephone number tells me what exchange area you live in. Your postcode will take me to a small group of houses, one of which is yours. Your full address will take me to your front door. I will have to know your name, age or other details to single you out from Gran, Mum, Dad, half-a-dozen kids and a pet goldfish.

There are ways I might find you without having to know your actual address. 'Third house past the *Rose and Crown*' – before opening time, of course. As long as I know where the *Rose and Crown* is, and I go at the right time, I will find you.

Your bank account number might not say anything of your whereabouts but it could be made to do so. Most of the numbers we are burdened with cross-reference our name and address and it works the other way about, too. Your bank manager calls out, 'Get me Smith, P. D. Q.' – and in next to no time Smith's overdraft is staring mockingly up at the bank manager. Figure 1.1 shows the indirect route taken to get the Smith data.

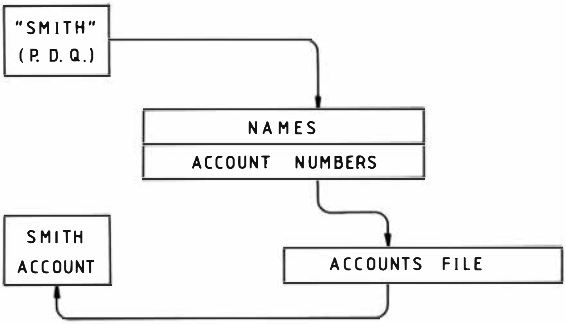


Fig. 1.1. Indirect addressing.

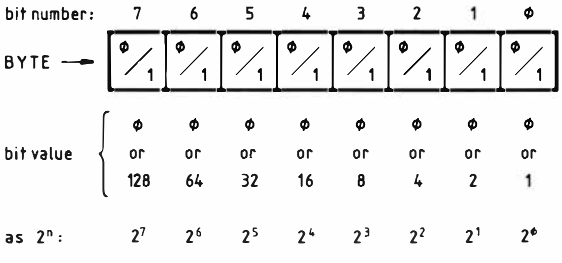
Our social organisation has come up with many different methods of directly or indirectly referring to people, places and things. It is not surprising that we build computers which utilise some of those methods.

Half a million bits

Computers based on processors like the 8080, Z80, 6502, 6800 and 6809 have, with all their memory intact, somewhat over half a million units of information inside them. In fact a little electronic jiggery-pokery will let us put a virtually infinite number of information units in these computers, though not all accessible at the same time. How it is done comes up later in the book but half a million units will keep us occupied for now.

All this information is useless if it is not organised in such a way that we can access and use any single unit that we want. Languages like BASIC come between us and the computer and provide us with a relatively simple but mysterious information storage facility called a *variable*. Some variables will hold numbers and others 'strings' of alphabetic characters. The latter type have to be given names ending in '$' for some obscure reason. But how BASIC actually stores the information inside the computer doesn't bother us: we type in 'Smith' and up pops the data on Smith – if we loaded the right program, of course. Somehow BASIC manages to extract the 'Smith' information from amongst those half million units.

The units of information are binary digits, called *bits*. Each can be either *reset* (0) or *set* (1) – not a very large amount of information! But just think how the decimal system works: one digit can tell us ten different things (0, 1, 2, 3, 4, 5, 6, 7, 8 or 9) but two digits can tell us a hundred things (00 to 99). Every time the number of digits in use is increased by one, the number of *states* is multiplied by ten. Binary works in the same way as decimal but with a multiplication factor of 2 instead of 10. A group of 8 bits has 256 different states (00000000, 00000001, 00000010, 00000011, 00000100, ..., 11111110, 11111111). Notice the use of *place value* in both the decimal and binary systems. Each next digit to the left is worth 10 (decimal) or 2 (binary) of its right neighbour and when a digit gets too full it is reset to 0 and there is a carry of 1 to the next higher place. In decimal the carry action occurs after 9 while it just happens to occur sooner in binary – after 1, in fact.



bit number: 7 6 5 4 3 2 1 Φ
BYTE
Φ/1 Φ/1 Φ/1 Φ/1 Φ/1 Φ/1 Φ/1 Φ/1
bit value
Φ or 128; Φ or 64; Φ or 32; Φ or 16; Φ or 8; Φ or 4; Φ or 2; Φ or 1
as 2ⁿ: 2⁷ 2⁶ 2⁵ 2⁴ 2³ 2² 2¹ 2^Φ

Fig. 1.2. Binary place value.

Eight-bit groups are so useful that they are treated as single larger units of information called *bytes*. Most machine code operations are carried out on bytes rather than on individual bits. The 8-bit byte grouping is the main organisation of computer memory. The machine code programmer has access to 14 bytes of information held inside the 6809 processor and up to 65536 bytes in memory chips.

Hexadecimal notation

With our 7+/−2 unit ability brains we can fairly reliably distinguish between binary numbers such as 11010010 and 11010110 (with a little practice, perhaps!) but when the sequences are extended to 16 bits or more easy discrimination becomes well nigh impossible.

Computers don't mind bits – but then they are made that way. We prefer to work with symbols which carry a greater amount of information per symbol so that we can use fewer of them, as in decimal. The decimal number 210 is much easier to read and understand than 11010010 which is the binary equivalent. But translation between decimal and binary is not all that straightforward. We cannot simply translate each decimal digit into a sequence of bits and then butt-join them.

An early attempt at reconciling the computer's preference for binary with human cognitive processes came up with *octal* or base 8 numbers (decimal is base 10). Three bits have eight different states which can be directly converted into the eight values (0 to 7) of an octal digit. This was doomed to failure, of course, since bytes have 8 bits. This meant that the leftmost octal digit never got above 3. Now base 16 is more or less the standard and works out quite neatly. Four bits have 16 states and each byte has two groups of four bits. Hexadecimal or *hex*, as the base is called, is not too difficult to work with once you get used to the letters A, B, C, D, E and F assuming another role as the digits following on from 9.

Two hexadecimal digits (a hex-pair) represent 8 bits or 1 byte. Four hex digits have 65536 different states – which is the exact number of possible 6809 memory locations. Any single memory location can be uniquely addressed by a 2-byte number (0000 to FFFF).

One further point. Whenever there is likely to be confusion as to whether a number is decimal or hex (it may even look like a name since hex uses some letters) it is usual to precede the hex by '$', as in ($0000 to $FFFF).

Table 1.1. Binary – hexadecimal – decimal

Binary	*Hex*	*Decimal*	*Binary*	*Hex*	*Decimal*
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15
			then		
10000	10	16	11000	18	24
10001	11	17	11001	19	25
10010	12	18	11010	1A	26
			and so on.		

The ingenious uses of ON and OFF

You can regard a computer as a glorified light switch. All that any of its bits can tell you is that it is ON (1) or OFF (0). Its electrical circuits are either *high* (probably +5 volts) or *low* (0 volts). But then you can regard a human being as a glorified amoeba. The glorification is that a person's body cells (or a computer's bits) are not entirely separate entities but are interdependent and have specialised functions. (At this point you might like to read Appendix A which describes the basic parts of a computer system generally and the 6809 processor in particular. On the other hand you might not like to read it.)

Why has no one designed a decimal computer? After all, decimal is our natural counting system, based as it is on our having ten fingers (including the thumbs). Actually decimal computers have been designed, built and used but binary computers are much simpler. They are easier to design, can make use of the electrical conductivity properties of such cheap and plentiful materials as silicon and are not as alien to our thought processes as the red herring about decimal fingers might have suggested (we don't use

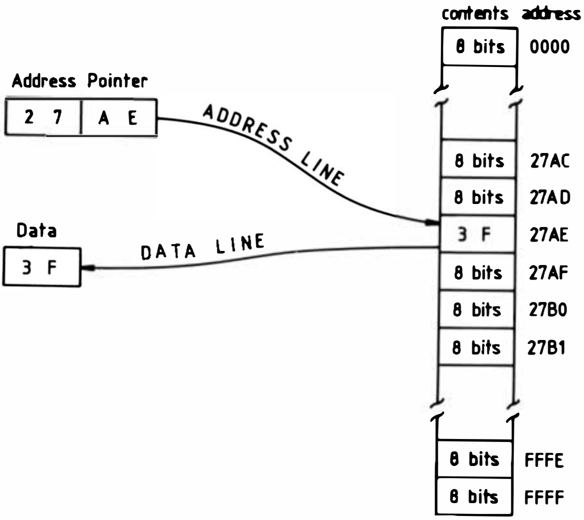


Fig. 1.3. Random Access Memory – direct addressing.

place value in finger counting – if we did it would have to be a binary system).

Did you read Appendix A – yes or no? Are you male or female, left-handed or right-handed? Is the weather wet or dry? Is the time day or night, a.m. or p.m.? Our cells, like the amoeba, reproduce by binary fission (splitting in two) and our neurons either fire or do not fire. We are very binary.

Binary decisions are fast, no 'hmmm – maybe'. At each point where a choice has to be made there is one simple test with only two possible results. A computer beating a path to the door of just one memory location out of 65536 does so in just 16 easy steps, working from a first test on bit 15 (the leftmost bit of a 16-bit address) down to bit 0, as in Table 1.2. The sequence is typical of a binary search process called the 'binary chop'. At each test exactly half of the remaining addresses, file records, list entries, or whatever are being searched are dropped from consideration.

Table 1.2. Binary address selection ($ABCD).

Bits tested	*Group in (hex)*	*Group out (hex)*
(high byte)		
1	8000 to FFFF	0000 to 7FFF
10	8000 to BFFF	C000 to FFFF
101	A000 to BFFF	8000 to 9FFF
1010	A000 to AFFF	B000 to BFFF
10101	A800 to AFFF	A000 to A7FF
101010	A800 to ABFF	AC00 to AFFF
1010101	AA00 to ABFF	A800 to A9FF
10101011	AB00 to ABFF	AA00 to AAFF
(low byte)		
1	AB80 to ABFF	AB00 to AB7F
11	ABC0 to ABFF	AB80 to ABBF
110	ABC0 to ABDF	ABE0 to ABFF
1100	ABC0 to ABCF	ABD0 to ABDF
11001	ABC8 to ABCF	ABC0 to ABC7
110011	ABCC to ABCF	ABC8 to ABCB
1100110	ABCC and ABCD	ABCE and ABCF
11001101	ABCD	ABCC

The chopping sequence of Table 1.2 is shown as a two-phase operation to highlight another important organisational feature of the 6809's memory: *pages*. Memory is divided into 256 pages which are numbered by the high order byte of the full two-byte address. The 6809 has a single-byte register which can be set to hold any page number and special Direct Page instructions which need only a 1-byte address to specify any one of the 256 different locations within the currently addressed page.

Having reached memory location $ABCD, we find that it contains 8 bits, each one either a 0 or a 1. What can this collection of bits stand for?

(1) An unsigned value between 0 and 255 ($00 to $FF).
(2) A signed value between −128 and +127 ($80 to $7F).
(3) Part of a larger number, perhaps 16 or 32 bits long.
(4) Part of an address (high or low order byte) pointing to another memory location.

(5) A collection of individual bits, each of which will light up a dot on the display screen if it is set.
(6) An ASCII character code (see Appendix D).
(7) A sequence of bits to cause branching in a program.
(8) Part of a machine code instruction.
(9) A voltage pattern for a D to A converter.
(10) Nothing at all.

The last item (10) possibly surprised you. If it did then that means you had *assumed* the contents of memory location $ABCD would mean something. Quite a lot of human assumptions are wrong but, since we survive as individuals and as a species, many of them must be more or less right. Assumption is a major factor in human thinking – it enables us to respond quickly to the real world. Computers don't make assumptions, they act on exact data; but programmers are apt to make the most unreasonable and disastrous assumptions.

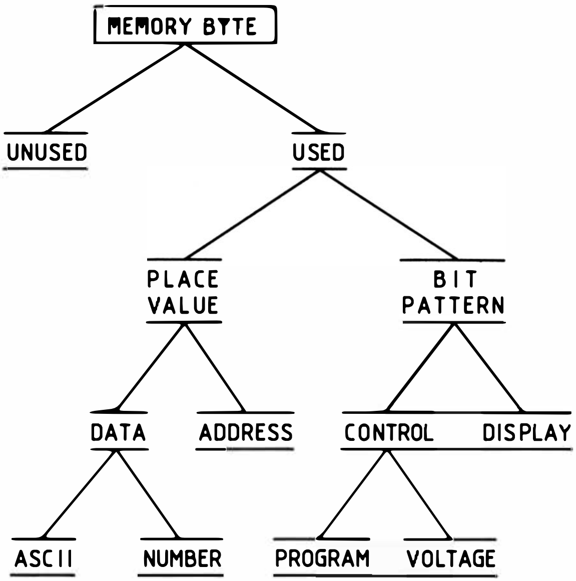


Fig. 1.4. Structuring information.

That you may have assumed that $ABCD contained valid data was probably my fault for presenting the possible uses (or non-use) of the memory byte as an unstructured list – useful, perhaps, for getting a few ideas down quickly and fitting the shape of a book page, but not much else. Lists of that type don't show up any groupings which emphasise the different relationships or associations between the entries. Even worse, they tend to obscure vital information, like the need to be sure that the location contains data and not just rubbish. Data (or information) structures, such as the *binary tree* (see Fig. 1.4), help to make explicit those facts which we might take for granted and also describe the connections between items.

Structure is the subject of the next chapter and the types of data use given in the list are dealt with at various places throughout the book. The list, by the way, is not definitive: the uses we can make of bits and bytes are limited only by our imagination.

Out of machine code

6809 machine code consists of a limited set of numerical instructions, from one to five bytes in length, which the processor decodes and then acts on. The processor uses the individual bits of the instructions while we see them as bytes, usually expressed in hex form. But people are not very good at associating pure numbers with specific operations (New York City police excepted – 'I have a suspected 159 on 5th and 12th') even though we might eventually learn to recognise instructions in numerical form.

All machine code programming is done in *assembly language* where each operation type is given a *mnemonic* (memory aid) abbreviation of the action, the CPU registers have names like A, X, PC, etc. and even addresses and data may have descriptive labels attached to them. An assembler will let you program in this symbolic form and then *assemble*, or translate, your symbolic *source program* into *object code* – the actual machine code.

If all you want to do is add a few simple machine code subroutines to your BASIC programs to speed up games or control external hardware, you should get by with the process known as *hand assembly*. You will still need to use assembler type mnemonics and labelling in writing your programs on paper but then you do the tedious job of translation into the numerical code form. DATA statements can be used along with a short READ ... POKE routine written in BASIC to store the instruction bytes in memory. EXEC, USR or similar BASIC functions should pass control to the address you specify.

Hand assembly of larger routines or complete machine code programs is very laborious. For work of this kind it is practically essential to use an assembler. The usual features of 6809 assemblers are described in Appendix B, and Appendix C gives the 6809 instruction set in both assembler and numeric code forms. If you are not familiar with 6809 instructions then this is a good time to read those two appendices.

Chapter Two
How to write Machine Code Programs

The title of this chapter is, perhaps, a little over-enthusiastic since the art of programming has filled several multi-volume books. All I can do here is throw you a few hints – some dos, don'ts, whys and why-nots – that may help to make machine code programming somewhat easier than it might otherwise be.

Machine code is a lot more difficult than BASIC for quite a number of reasons. Here are some of the main ones.

(1) All of the instructions are simpler (i.e. they do less) than BASIC commands, so you need more of them.
(2) BASIC deals with all the addressing for you. In machine code *you* have to decide where to put both programs and data.
(3) BASIC uses English words (more or less) and nice neat mathematical expressions and so is fairly easy to read even without REM statements. You need to be a machine to read machine code!
(4) BASIC programs can (but shouldn't) be written at the computer. Machine code *must* be written out on paper first. Even the best assemblers can show only a few instructions on the screen at any one time.

All these reasons suggest that, while you can get away with writing BASIC programs at the computer by spending an inordinate amount of time in EDIT mode, machine code must be approached in a more organised fashion. In fact the actual coding of the program should not take place until you have worked out a complete *structured design*. This might seem a little hard when possible code sequences are already suggesting themselves to you, but programs are far easier to change in the design stage than when a lot of code is already in place.

Another reason for delaying the writing of code until the design is complete is that it forces you to produce *documentation* for the program. Documentation produced in the design stage acts as a route map through the program. Without it large programs are extremely difficult to read and tracing the flow of control (the order in which various parts are executed) during debugging can be almost impossible.

Structure is basically of two types. The first type shows the breakdown of the program into dependent parts, each of which can be further subdivided until a complete tree structure exists of the program. This will not be a binary tree – each part can be divided into many smaller parts. The second type of structure is developed from the first and shows program flow.

Program structure

Computers are unaware of any logic more complex or detailed than that found in a single instruction. Each instruction is dealt with in isolation from, and without regard to, any other program instruction. Structure, then, exists only for the benefit of the programmer, not the computer.

Structuring a program is a top-down process. Don't be tempted to work out a section at quite a low level and then try to fit it in. Working from the top down means that at each stage you can forget about a large part of the program and concentrate your efforts on just one main branch. Follow the branching down religiously, keeping each part of the program separate. At some point in the proceedings you might notice that the branch you are working on includes processes that are the same as those of another branch already completed. It might seem a worthwhile idea to join these branches together at this point – but don't. When you come to design the program flow the identical parts can be written as a common subroutine or block of subroutines. Structurally, they are entirely independent.

There are no hard and fast rules about where and when to divide into parts. This is where programming skill comes in. Practice might not make perfect but it certainly helps you to see where a program can usefully be split up. Sometimes the divisions are obvious but at other times, especially in the higher levels, programs seem to defy you to split them into logically distinct parts. If divisions don't immediately suggest themselves then a rule of thumb is to aim at about half a dozen separate parts of equal weight. A short period of enlarging the scope of some parts at the expense of others, joining up thinned down parts or further splitting grossly enlarged parts should ensue. Eventually you will find the inherent structure of the program. Why half a dozen divisions? We are back to the 'Magical Number Seven, Plus or Minus Two'. It really is difficult to understand the structure of a program if all the splits are into a dozen or more parts, sub-parts, and so on. If that appears to be happening then have a rethink and see if you cannot produce a more readable structure. After all, the structure is there to help you find your way easily through the program.

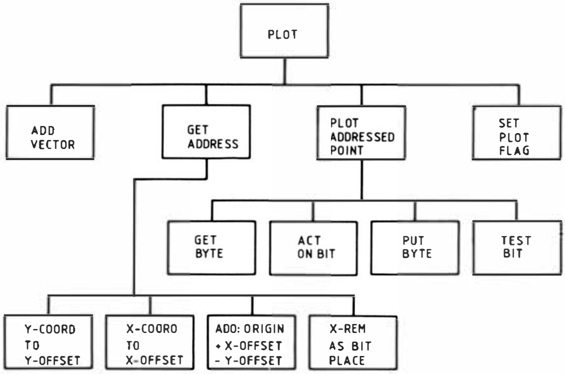


Fig. 2.1. PLOT structure.

Figure 2.1 shows a structure chart of the PLOT routine from the chapter on high resolution graphics. Subroutines of this kind sit right at the bottom of the structure tree (structure trees grow downwards) and usually comprise no more than three or four levels. The lowest level is at a point where each box represents only a dozen or so instructions at most and preferably only two or three. This is the point at which to stop dividing. Extending the design to another level would mean taking into account the actions of individual instructions. There are three (at least) good reasons for not doing that: (1) you might later want to write the program for another processor with very different instruction capabilities, (2) at this stage you cannot be sure which registers are going to be used nor the way in which data will be addressed or accessed, and (3) the structure chart only shows the interdependence of the parts, not the program flow which will directly affect coding.

Program flow

Flow charts describe the order in which each part of the program is dealt with by the computer and are a development from the structure charts. They don't stick to rectangular boxes but use several box shapes from a standard set of symbols. You can buy flow chart templates with about twenty different symbols on them. They

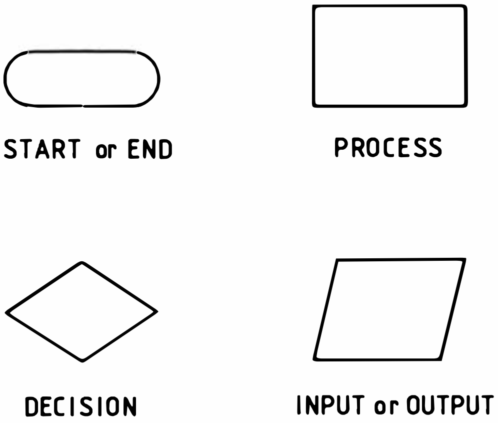


Fig. 2.2. Main flow chart shapes.

usually have labels giving the meaning or use of each symbol. Look for templates which conform to ISO Standard 1028, ANSI X3.5-1970 or BS4058. Figure 2.2 shows the four main symbols used in drawing flow charts. Other symbols are mostly concerned with differentiating media for storage or display – magnetic tape, visual display, punched card, etc. The shapes are meant to symbolise different actions and the order in which the actions are performed is described by a flow line which commonly has arrowheads to show the direction of flow. Arrowheads are, however, totally unnecessary for the flow lines in a structured flow chart and you should avoid them like the plague. They allow you to produce unconventional and pathological structures which are difficult to read and to code. Flow charts that don't rely on arrowheads have to be written, or drawn, to a set of standard *constructs* which help to make the subsequent coding clear, simple and quick.

There are three basic types of construct: *sequence*, *iteration* (looping) and *selection* (branching or decision). Through all these constructs the general direction of flow is downwards, entering at just one point (the top) and leaving at just one point (the bottom).

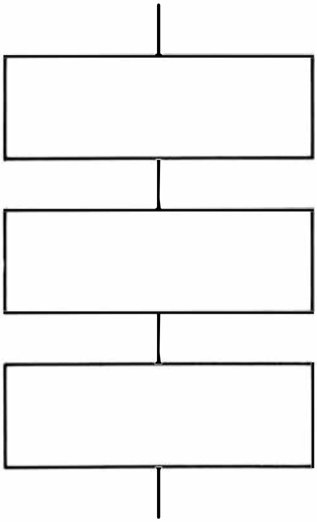


Fig. 2.3. Sequence.

Selection and iteration also have internal lines which flow left, right or upwards. Figure 2.3 shows a sequence of three processes. The flow goes straight down through each process in turn.

Selection, shown in Fig. 2.4, is a binary decision with only two possible results. Flow is either to the right or left, not both. The left and right paths do not have to be labelled since the result of a binary decision is always either *false*, *no*, 0 (to the left) or *true*, *yes*, 1 (to the right). The flow lines stretch out horizontally far enough to

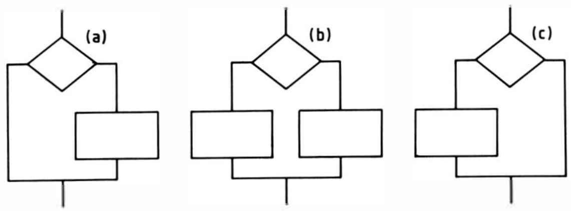


Fig. 2.4. Selection: (a) skip if false, (b) if then ... else, (c) skip if true.

accommodate the width of the different option boxes and then turn down. At the bottom they turn inwards, join in the centre and exit as a single line of flow.

Iteration is usually a REPEAT UNTIL or REPEAT IF function. The processes repeated have to be performed at least once, as in Fig. 2.5. The internal flow line leaves the end-test decision box on the left (repeat if result false) or on the right (repeat if result true) and moves out far enough to clear the process box. It then turns upwards until it reaches the start of the process to be repeated where it turns back in to join the downward flow line. A side join always means that the flow has come from the end of an iteration construct. The other line leaving the decision box does not come out horizontally from the other side, as it does in the selection construct. In this case it is a

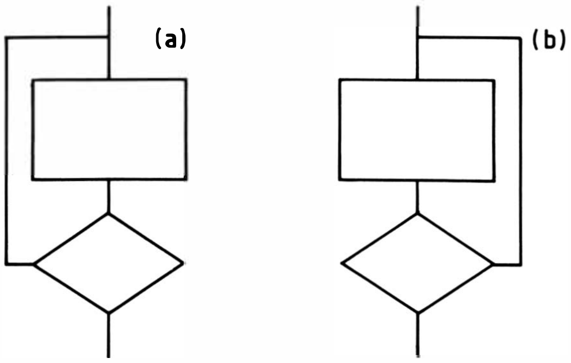


Fig. 2.5. Iteration: REPEAT UNTIL (a) true, (b) false.

fallthrough line, i.e. the test result matches the UNTIL (or does not match the IF) condition and flow falls through to the bottom of the construct. REPEAT UNTIL gives the fallthrough condition. REPEAT IF gives the looping condition.

The process inside a REPEAT construct is always performed at least once because the loop-test is at the end. Occasionally we need an iterative structure which will allow the process to be skipped entirely with a test right at the start of the construct, before the process box(es). This is a DO WHILE situation. While a certain condition applies, the process will be performed. The iteration ends as soon as the condition ceases to hold. DO WHILE is actually a composite construct made up of an initial selection for the possible 'skip process' and a normal REPEAT IF to determine if the condition holds for a repeat. Figure 2.6 (a) shows the composite structure of DO WHILE.

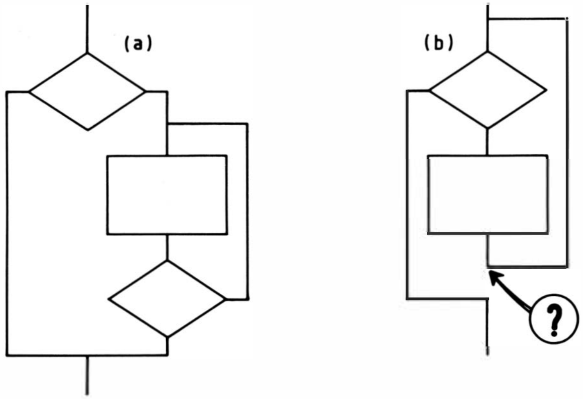


Fig. 2.6. DO WHILE true: (a) well structured, (b) pathological.

A pathological form of DO WHILE is often found in flow charts and is shown in Fig. 2.6(b). It does not conform to good structure standards in two ways: (1) the loop-back line could turn either right or left at the point indicated by the question mark and so could cross the exit line, and (2) the exit is from the side of the construct and could, without due care, give rise to extreme forms such as that in

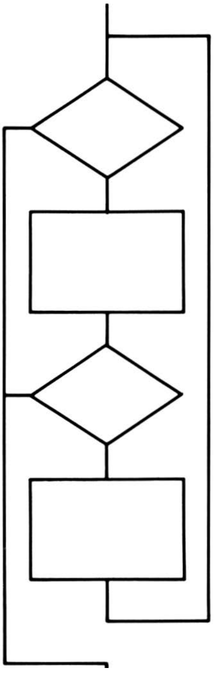


Fig. 2.7. A more pathological form of DO WHILE.

Fig. 2.7. The pathological DO WHILE is used quite a lot in coding with the justification that a simple 2-byte BRA instruction can replace a complex test and decision many bytes long. Optimization of this kind is often necessary when large programs have to be packed into a small amount of available memory. The flow chart design, however, should precede coding, particularly any optimization, so only the well-structured forms should be used at this stage.

Figure 2.8 shows the flow chart constructed from the PLOT structure chart (Fig. 2.1). All third level actions are incorporated in

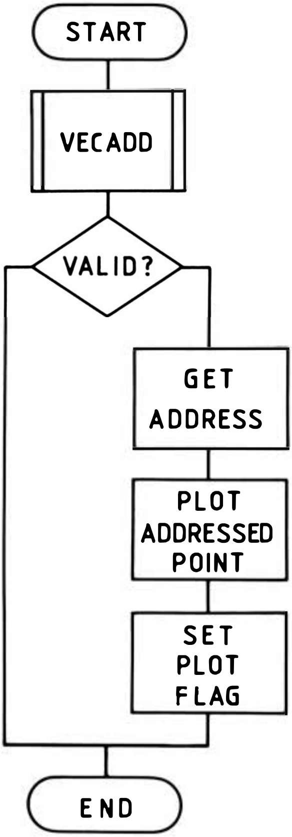


Fig. 2.8. PLOT flow.

the GET ADDRESS and PLOT ADDRESSED POINT process boxes. Flow charts can be drawn at various stages of the structured breakdown to illustrate more clearly how program control passes through any part of the program.

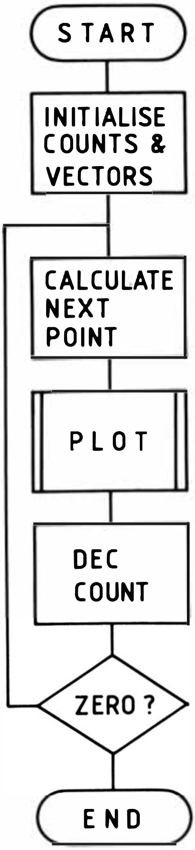


Fig. 2.9. LINE flow.

PLOT is used repeatedly by the routine LINE to plot each point on the line it draws. Figure 2.9 is a high level flow chart showing where PLOT fits in. The double sides of the PLOT process box show that it is a separate subroutine or independent procedure called by LINE. Even in a more detailed flow chart showing deeper levels the double sided box remains unexpanded – its structure and flow are shown on its own different chart.

Coding, testing, debugging

If you have prepared detailed structure and flow charts then this stage will be fairly straightforward and easy. If you have not you might end up in a tangle almost as soon as you start.

The lowest level boxes on your structure charts should name tasks that can be coded in a dozen instructions or less. These short sequences of code are the basic action routines. They perform some process or change on data fed to them from the next higher level and then pass the result back. All higher level tasks are concerned with some form of management – Which data? Which processes? What order? The entire edifice echoes the worker-management pyramid structure found in industry.

This distinction can prove important in coding and testing. Most of the bottom level routines can each be coded and tested in complete isolation from other parts of the program. Testing is a matter of inputting test data and checking output result for any errors. Test data is data at the extremes of the range of data the routine is designed to act on. For example, a routine which acts on ASCII hexadecimal digits and not other values would need to be tested with:

$2F (/)	$30 (0)	$39 (9)	$3A (:)
$40 (@)	$41 (A)	$46 (F)	$47 (G)

These are the codes at the end of, and immediately outside, the two groups used for hex digits. Other test data might be values in the range $80 to $FF which are not used as ASCII codes.

Routines at higher levels should be coded and tested from the top down. The highest levels will consist mainly of subroutine call instructions and branches selecting which lower parts to use. Here again test data can be fed to the routines being tested, this time from below. Many of the routines called by these levels might not yet be tested, or even written. They can often be emulated by a simple RTS (Return from Subroutine) instruction, possibly after setting or resetting necessary flags in the Condition Codes register by ANDCC

Table 2.1. Monitor commands.

Command	*Meaning*
Breakpoint	Insert Software Interrupt (SWI) at given address, saving replaced code byte.
Copy	Transfer a block of memory to new location.
Dump	Display a block of memory contents in hex.
Enter	Direct keyboard input of code or data bytes, either as hex-pairs or ASCII.
Exit	Exit monitor for other system software, e.g. BASIC or assembler.
Fill	Fill a block of memory with one value.
Go	Execute program at address in displayed PC register.
Jump	Execute program at given address.
Load	Load machine code program from tape or disk.
Register	Display contents of all registers, allowing them to be changed.
Save	Save machine code program on tape or disk.
Single-step	Execute program one instruction at a time, on key press, displaying all register contents and current instruction code at each step.
System	Alter computer system parameters, e.g. print speed, I/O rate, display mode.
Trace	Print control path of program during execution, i.e. address of every instruction executed.

and ORCC instructions. Such subroutine *stubs* are normally all that is needed to test the logic of the top levels.

The essential tools for coding, testing and debugging are an editor-assembler and machine code monitor. It is desirable to have these both resident in the computer in ROM form. Versions which have to be loaded into RAM from tape or disk can be corrupted by a faulty object program. The normal features of 6809 assemblers are described in Appendix B. Monitors are used on the assembled object code, not on the source program, allowing you to examine and change the contents of individual memory locations and registers. The features to look for in a monitor are given in Table 2.1.

Program documentation

Structure and flow charts form the major part of the documentation for the design stage. They often need to be supplemented by (a) definitions of the program, data and system on which the program is to run, and (b) decision tables showing conditions (cause) and actions (effect) for any complex decisions made in the program. A full description of this further documentation is beyond the scope of this book. For a full and very readable treatment of structure and documentation, I urge you to read the book by Tom DeMarco (see the *Further Reading* list).

The design documentation can help in producing documentation for the assembly language program, the names and descriptions of processes being transferred straight to the code routines. Documentation of the source program is essential – a bare list of several hundred assembler instructions is not much easier to understand than actual machine code.

Each clearly distinct part of the source program should have *header* information: its name, brief details of its action or task, the data input to it and output from it, any registers or memory changed by it, names of the subroutines that it calls, the maximum number of (hardware) stack bytes that it uses and the execution time in clock cycles if this is important. This information can be given on complete comment lines preceding the code.

Assemblers also offer the facility of adding comments after each instruction. Use this gift to the full. The comments written alongside the instructions should not just describe the individual actions of each instruction but should also make clear the full task performed by sequences of code. Using them as a rehydrated version of the mnemonics will not hold much water when, months later, you need to update the program and have to work out what the code is actually doing. The following routine L1234 is an example of atrocious program documentation.

```
L1234   LDA     #$FB    ;load A with 251
        STA     2,X     ;store A at (X+2)
        LDA     ,X      ;load A from (X)
        ANDA    #$20    ;AND A with 32
        BNE     L1235   ;branch if not equal
        BEQ     L1236   ;branch if equal
```

Rewriting it as CHKEYZ informs you of what the sequence is doing.

```
;CHKEYZ - test for Z key press. X = $FF00
CHKEYZ    LDA    #%11111011    ;write Z col. mask to out-reg.
          STA    2,X           ;at $FF02 and read rows-in
          LDA    ,X            ;from reg. at $FF00. Mask
          ANDA   #%00100000    ;out non-Z rows, then branch
          BNE    NOKEYZ        ;to "Z not pressed" or else
          BRA    KEYZ          ;to "Z pressed" routines.
```

Other differences between the routines are (a) CHKEYZ and the other labels used are abbreviations of the actions performed or the special entry conditions of the routines but L1234, L1235 and L1236 are meaningless, (b) binary numbers are used in CHKEYZ to draw attention to the fact that it is the bit-patterns and not numerical values that are being used, and (c) the use of BRA in CHKEYZ informs you that there is no fallthrough from the routine whereas this fact is not at all clear in the L1234 routine.

Now that the actions of the routine have been made plain it is obvious that the program is badly structured. Unless either NOKEYZ or KEYZ immediately follows CHKEYZ, the routine is doing two jobs - checking for Z key press and selecting from two processes. The use of indexed addressing to write to and read from the PIA registers at $FF02 and $FF00 is also unnecessary since those addresses are fixed, and it means that the routine depends on the X register being set at $FF00 on entry. These problems should have been sorted out during the program design stage. Good program documentation can highlight design errors - bad documentation would only hide them deeper.

Data

Purists regard *data* strictly as the plural of *datum* but computing convention has it as singular, collective, abstract or descriptive, so it is quite usual to say 'data is' rather than the grammatically correct 'data are'. Data can be of three types: *constant*, *variable* and an inbetween sort referred to as *parameter*. Our old friend PLOT can again be helpful as an illustration since it uses all three types of data.

Constant data never changes. It can be written straight into the program code if necessary - for example, LDA #$FB - but there are often good reasons why it shouldn't be. In PLOT, the conversion of the *y* coordinate to a vertical offset from the origin address requires, as constant data, the line increment value. This is the difference between the addresses of two vertically adjacent screen locations and while the line increment may not be the same on different computers it is a constant within any particular computer. PLOT also uses an eight-byte table of constant data. Each byte in the table has one set bit corresponding to one of the eight possible dot positions in one screen location. This too remains unchanged.

Variable data is that which can take different values every time a code sequence is executed. It is usually input to a routine as values held in accumulator or index registers. Sometimes a routine may have to pick up variable data from memory. The vector input to PLOT in the form of register values is variable and so are the co-ordinates of the last point plotted which PLOT reads from memory.

Parameters are data that define or limit the action of a routine. The vector and co-ordinates are really parameters but the term is often kept for data which changes less often. Parameters in PLOT include the origin address and the number of horizontal and vertical dots on the display area. These are constant for long periods, perhaps for the entire program, but they may be changed to use different screen pages (in the Dragon or TRS-80 Color Computer) or to limit the size of a *display window* on the screen.

In a complete program constant data and parameters may need to be accessed by several routines. Common data of this sort should be put in a reserved area. Computer operating systems are often fixed in ROM and cannot be changed so their *system variables* or *system parameters* are written to an area of RAM where they can be accessed, and altered if necessary, by any of the routines in the system. Corrupting this data can cause a system crash.

Constants limited to a single routine can be written into instructions as immediate data (Immediate addressing mode). If the same data is used several times during the routine it should be equated to a label in the source program before the first instruction. The assembler will insert the actual data in the instruction when it meets the label as operand.

```
BSDATA   EQU    10       ;base 10 data used in BASADJ
BASADJ   LDD    ,X       ;pick up value to adjust to
         CMPB   BSDATA   ;base, if lo-digit is less
         BCS    SADVAL   ;than base then skip, else
         SUBB   BSDATA   ;adjust by subtracting base
         INCA            ;and inc'ing next place digit
SADVAL   STD    ,X       ;re-store adjusted value
         RTS             ;and end routine.
```

BASADJ adjusts the low order digit of a two-digit value picked up in the D register (A with B) to a base BSDATA. If we want to use a different base (any base from 2 to 255) only the value equated to BSDATA need be altered.

Routines often need workspace for temporary storage of variables and intermediate results. If the workspace requirements are only small – say, half a dozen bytes or less – then the 6809 hardware stack is as good a place as any. The hardware stack pointer S can be used in exactly the same way as any of the index registers, X and Y, or the User stack pointer U with the Indexed and Indirect addressing modes. However, you should never use memory immediately *below* the current stack position. Any interrupt occurring will stack the contents of the entire register set (or just PC and CC in a fast interrupt), overwriting and corrupting your workspace. The Dragon and TRS-80 Color Computer Timer function works by an interrupt every $\frac{1}{50}$ second. If more than half a dozen bytes of workspace are needed a special area ought to be reserved immediately after the routine, or in a common area for use by several routines, using the assembler directive RMB.

Chapter Three

Number Crunching

Simple arithmetic is a fairly straightforward process which doesn't involve any computer hardware other than RAM and the processor itself, but it does bring in some commonly used and very important methods of memory addressing and program control so it is a good subject to start with.

The 6809 is an 8-bit processor and most of its instructions act on only one byte of data. It does have instructions to add, subtract and compare 16-bit (double byte) values but these are designed primarily for manipulating addresses. The processing of multi-byte values is usually best done inside a loop which deals with only one byte at a time although there are exceptional cases where 2-byte chunks can be processed.

MBADD adds the multi-byte binary number indexed by Y to that indexed by X and stores the result at a third location indexed by U. The values must all be the same length and this is input into the routine in B. An initial test is carried out on B to see if it is greater than $7F (127) and if it is then the routine aborts. This is because the accumulator offset indexing uses B as a signed value in the range $80 to $7F (−128 to +127) and if a negative value were used then memory below the addresses in U, X and Y would be changed. With only positive values of B valid, the loop end test is on the state of the negative flag N which is 0 for all valid B and goes to 1 immediately after the highest order bytes (at U, X and Y) have been processed and B is decremented to $FF (−1). Before exit, B is incremented to set the zero flag Z and reset N to show that the addition has been performed.

MBADD – Multi-byte binary addition

Stack – 1.

I/O – Value at X plus value at Y stored at U.
B indexes the low order bytes from X, Y, U

```
        (B = no. of bytes - 1). Invalid if B > $7F.
        Output Z=0, N=1: input B too big (B > $7F)
               Z=1, N=0: add done, C = any carry out.

MBADD   PSHS    A           ;save A contents while A used.
        TSTB                ;make sure B is valid (B < $80)
        BMI     MBAEND      ;end Z=0, N=1 if it is not.
        ANDCC   #%11111110  ;no carry in to addition.
;loop, processing each value place byte from lowest at
;R + B (R is U,X,Y) to highest at R + 0 (when B = 0),
;including carry from previous bytes addition.
MBALP   LDA     B,X         ;get 1st argument byte, add with
        ADCA    B,Y         ;carry byte from 2nd argument
        STA     B,U         ;and store to 3rd argument.
        DECB                ;index next higher order bytes
        BPL     MBALP       ;repeat till all added.
;set Z, reset N to show addition done. C unaffected by INC.
        INCB                ;set valid output flags.
MBAEND  PULS    PC,A        ;exit MBADD, restoring A.
```

MBADD shows the basic form for any process which picks up a string of bytes from one area, performs some change or transformation on them (perhaps with reference to a different string elsewhere) and then stores the new values in a different area. A few simple changes are all that is needed to make the routine do various other things. For example, changing ADCA B,Y to SBCA B,Y turns MBADD into MBSUB – multi-byte subtraction. The result of the operation need not go into a third area. Replacing STA B,U by STA B,X will put the result back to the first argument. Deleting ADCA B,Y from the routine turns it into a string transfer routine, moving up to 128 bytes from an area indexed by X to one indexed by U. Other methods of moving memory are shown in later chapters.

Multiplication and division

It is not very likely that you will need to multiply values up to 128 bytes in length but 8-bit and 16-bit multiplication and division routines are often needed. The 6809 is much more sophisticated than the other common 8-bit processors in that it boasts an 8-bit multiplication instruction MUL which executes very quickly in only 11 clock cycles. Eight-bit division and 16-bit multiplication and division can only be done by agonisingly slow software methods.

DIVAB is an 8-bit division routine (A remainder B := A / B). The action is the binary equivalent of the normal long division method, except that shift and rotate instructions are used to move the dividend (A) over the divisor (B) instead of B being shifted down under A as is normally done in a decimal paper-and-pencil long division. Being binary, the divisor can either not be subtracted from the dividend or can be subtracted only once at each digit place. The result of each subtraction that 'goes' is a set bit (1) and the result bit is 0 if the subtraction does not go. Since the dividend is being shifted out of A by one bit in each iteration, the quotient can be shifted into A as each bit is determined. After all eight dividend bits have been shifted from A into B, the complete 8-bit integer quotient is in A and B holds the remainder.

DIVAB - 8-bit unsigned binary integer division

Stack – 3.

I/O – Input A is the dividend, B is the divisor.
Output A is the integer quotient, B the remainder.

Notes – If the divisor is zero then output A= $FF and remainder B = input A. Division by zero is normally considered an error.

```
DIVAB     PSHS   B,CC       ;save flags, put divisor on stack
          LDB    #8         ;set up count for 8 bit-shifts
          STB    ,-S        ;on stack ("push" count).
          CLRB              ;clear accumulator/remainder.
;loop 8 times, attempt subtraction of divisor at each digit
;place, forming quotient one bit at a time.
DABLP     ASLA              ;shift next dividend bit to
          RORB              ;remainder, clearing quot. bit.
          CMPB   2,S        ;test if divisor can be subtracted
          BLO    DABLPT     ;from remainder, and only if it
          SUBB   2,S        ;can, subtract and set quotient
          INCA              ;bit at corresponding place.
DABLPT    DEC    ,S         ;repeat till all dividend shifted
          BNE    DABLP      ;and A now quotient.
;put remainder into stacked B (originally divisor) for
;pulling. Clear count byte, tidying stack for pull.
          STB    2,S        ;pulled B to be remainder.
          LEAS   1,S        ;bump S to remove count byte.
          PULS   PC,B,CC    ;exit, restoring registers.
```

Sixteen-bit division is done in exactly the same way but, of course, needs 16 iterations. Multiplication done bit by bit is somewhat the reverse of division; the multiplier is shifted out one bit at a time and

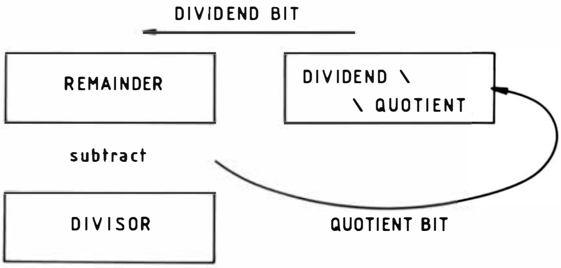


Fig. 3.1. Binary long division.

if the current bit is set then the multiplicand is added in to the partial product. The two operations are shown symbolically in Figs. 3.1 and 3.2.

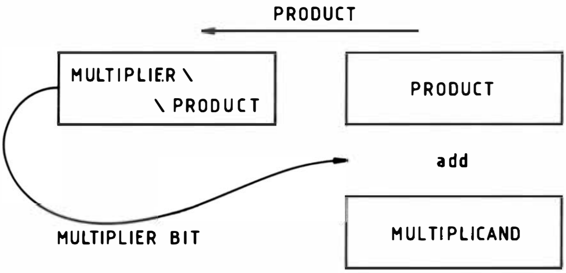


Fig. 3.2. Binary long multiplication.

DIVXY – 16-bit unsigned binary integer division
Stack – 8.
I/O – Input X is the dividend, Y is the divisor.
Output X is the integer quotient, Y the remainder.
Notes – Division by zero results in output X = $FFFF and Y = input X (dividend).

```
DIVXY   PSHS    Y,X,D,CC        ;save registers used.
        LDB     #$10            ;set up count for 16 bit-shifts
        PSHS   B               ;on stack, since A and B used
        CLRB                   ;as 16-bit accumulator/
        CLRA                   ;remainder (D).
;loop 16 times, try to subtract divisor at each digit place,
;forming quotient one bit at a time. Quotient shifts in as
;dividend shifts out.
DIVLP   ASL    5,S             ;shift next dividend bit through
        ROL    4,S             ;into remainder (D), clearing
        ROLB                   ;next quotient bit at bit 0 5,S.
        ROLA
        CMPD   6,S             ;test if divisor can be subtracted
        BLO    DIVLT           ;and skip (Q bit = 0) if not,
        SUBD   6,S             ;else it can so subtract and set
        INC    5,S             ;Q bit at corresponding place.
DIVLT   DEC    ,S              ;repeat until all dividend
        BNE    DIVLP           ;shifted. D now remainder.
;put remainder into stacked Y (originally divisor) for
;pulling quotient and remainder in X and Y. Clear count
;byte off stack so stack ready for pull.
        STD    6,S             ;remainder to stacked Y.
        LEAS   1,S             ;remove byte workspace off stack.
        PULS   PC,Y,X,D,CC     ;exit, restore with Q and rem.
```

MULXY – 16-bit unsigned binary integer multiplication
Stack – 8.
I/O – Input X is the multiplier, Y is the multiplicand.
Output X and Y contain the 32-bit product.

```
MULXY   PSHS   Y,X,D,CC        ;save registers, put arguments
        LDB    #16             ;on stack. Set up 16 loop count
        PSHS   B               ;on top of stack. Clear accum.
        LDD    #0              ;for forming product.
;loop 16 times, shifting partial product up one place and
;next multiplier bit out to carry flag C. If multiplier bit
;set then add multiplicand in at correct place. Partial
;product does not interfere with shifting out multiplier.
MULLP   ASLB                   ;shift partial product up
        ROLA                   ;through D and stacked multiplier
        ROL    5,S             ;which shifts up to accommodate
        ROL    4,S             ;and gets next place bit to C
        BCC    MULLT           ;no add in if place bit is 0,
        ADDD   6,S             ;else add multiplicand in to
        BCC    MULLT           ;correct place and take care of
        INC    5,S             ;any carry up through higher
        BNE    MULLT           ;order bytes of product, carry
        INC    4,S             ;won't reach multiplier bits.
```

```
MULLT    DEC     ,S            ;repeat until all multiplier
         BNE     MULLP         ;shifted out and processed.
;put low order two bytes of product into stacked Y (input
;multiplicand) for pulling 32-bit product in X and Y.
         STD     6,S           ;put product lo-bytes to stack
         PULS    B             ;clear loop count off stack.
         PULS    PC,Y,X,D,CC   ;exit, restore with XY = product.
```

MULXY uses the normal method of multiplication where there is no multiplication instruction to do the job. But since the 6809 does have an 8-bit MUL instruction a different method can be used – multiplying complete bytes at a time and adding the results in at the correct places. MBYBY does this for an 8-bit by 16-bit multiplication (B and Y := B * Y). The formula for the 24-bit product is:

(B * Yhibyte * 256) + (B * Ylobyte)

The method can be extended to a 16-bit by 16-bit multiplication or even further, but it is a long routine. The advantage is in speed – the byte method executes about twice as fast as the bit method. You pays your money and you takes your choice!

MBYBY – 8-bit by 16-bit unsigned binary integer multiplication
Stack – 5.
I/O – Input B is the multiplier, Y is the multiplicand.
Output B and Y contain the 24-bit product.

```
MBYBY   PSHS    Y,D,CC       ;save regs and stack arguments
        LDA     4,S          ;get multiplicand lo-byte and
        MUL                  ;mul by multiplier (in B) to get
        STD     3,S          ;part product to low 2 bytes.
        TFR     Y,D          ;get m'cand hi-byte to A and
        LDB     2,S          ;m'plier to B, then clear
        CLR     2,S          ;product hi-byte for add in.
        MUL                  ;mul hi-byte and do add in to
        ADDD    2,S          ;product high 2 bytes, back on
        STD     2,S          ;stack for pulling to B and Y.
        PULS    PC,Y,D,CC    ;exit, restore with product.
```

Pseudo-random numbers

True random numbers are very difficult to come by so the usual approach is to generate a series which exhibits minimum regularity. This subject has produced much discussion in the machine code series 'PCW SUB SET' in *Personal Computer World* (essential reading for assembly language programmers) where the conclusion was drawn that reasonably efficient 16-bit and 32-bit pseudo-random number generators could use the series

$$R_{i+1} = (1509R_i + 41) \bmod 2^{16}$$

and

$$R_{i+1} = (69069R_i + 41) \bmod 2^{32}$$

Modulus (or mod) 2^{16} arithmetic is extremely simple in machine code. It means the remainder left after dividing a number by 65536, and that is exactly what you get if you just take the two lowest bytes of any result as the answer and discard bits 16 upwards. Mod 2^{32} is the same but keeping the four lowest bytes.

Routines to generate both 16-bit and 32-bit random numbers on the 6809 appeared in 'PCW SUB SET' in May 1984. In both routines the numbers were held in memory indexed by U. But, of course, a routine can work on a 16-bit value input in a register and that is what RANDOM does in order to produce a pseudo-random number at maximum speed – ideal for determining the random attack patterns of alien invaders.

The constant multiplier of the last random number (or *seed*), 1509, can easily be factorised to simplify the calculations:

$$1509R = (2 * 3 * 256 * R) - (3 * 3 * 3 * R)$$

The multiplications then reduce to the quicker shift, add and subtract operations.

Random – 16-bit pseudo-random number generator
Stack – 3. *Clock cycles* – 75.
I/O – Input D is the previous random number or seed.
Output D is the new random number, negative flag N is set if D > \$7FFF, zero flag Z is set if D = 0.
Notes – Generator series $R_{i+1} = (1509R_i + 41) \bmod 65536$ is effected by using the identity, $1509 = (6 * 256) - 27$, and then using shift and addition instead of multiplication. Clock cycles are given against each instruction.

```
RANDOM  PSHS   D      ;7,   (S) = R
        ASLB          ;2,
        ROLA          ;2,   D = 2R
        ADDD   ,S     ;6,   D = 3R
        STD     ,S      ;5,   (S) = 3R
        ASLB            ;2,
        ROLA            ;2,   D = 2 * 3R
        PSHS    B       ;6,   (S) = 2 * 256 * 3R (hibyte)
        ASLB            ;2,
        ROLA            ;2,   D = 4 * 3R
        ASLB            ;2,
        ROLA            ;2,   D = 8 * 3R
        ADDD    1,S     ;7,   D = 9 * 3R
        STD     1,S     ;6,   (S+1) = 3 * 3 * 3R
        PULS    A       ;6,
        LDB     #41     ;2,   D = 2 * 256 * 3R + 41
        SUBD    ,S++    ;9,   ... - 9 * 3R. Tidy stack.
        RTS             ;5, exit, D = new R.
```

The same method can be used for a 32-bit random number generator with input and output in, say, the X and Y registers but the execution time will be more than double that of the 16-bit routine. RANDOM should prove adequate for any game since the series repeats only after 65536 different values.

The fact that it is a repeating series and will always produce the same sequence given the same starting value is a problem common to all pseudo-random number generators. Unless the computer system has some hardware device which can be assumed to have a different state each time a program is run – such as an on-board, real-time clock – then the usual method of finding a new seed is to seek keyboard input. Video display and keyboard reading are subjects for later chapters. For now, assume a message printing routine which asks for any key to be pressed and a routine to test for a keypress. The following program sequence will then produce a different value each time the program is run.

```
         JSR    REQUST   ;go print input request.
SEEDLP   ADDD   #1       ;continue to increment seed
         JSR    KEYCHK   ;until keypress check results
         BEQ    SEEDLP   ;in finding request met, then
;continue program with unique seed in D.
```

Chapter Four

PIAs, SAM and Folding Memory

When you use BASIC, the technical or hardware side of the computer is almost totally hidden. This helps to make BASIC programming the relatively easy job that it is but it does prevent you from taking direct command of the computer. Machine code programming, on the other hand, puts you in full control of the whole system. The catch is that you have to know more than just the particular language used by the microprocessor: you need to know how to use the other hardware devices in the system.

Microcomputer systems are technically quite complex. Even an inexpensive home computer has a lot more to it than just a microprocessor and a few memory chips – for example, the parts list of the TRS-80 Color Computer takes up six (large) pages of the Technical Reference Manual (available from Tandy stores, Catalogue number 26-3193). Twenty-nine of the parts are integrated circuits, including the MC6809E CPU, two MC6821 PIAs and a MC6883L SAM. The CPU is, of course, the microprocessor. The PIA and SAM chips are good examples of *Input/Output* and *System Control* devices which are discussed in general terms in Appendix A.

The 6820/6821 Peripheral Interface Adapter

The PIA is one of the most common *parallel* I/O devices used in 6809 systems. Parallel devices can input or output eight bits (one byte) of data simultaneously. They can also be used to emulate *serial* devices – which input or output a sequence of bits one at a time – by software control of just one line. The difference between the 6820 PIA and the 6821 PIA is only technical. As far as the programmer is concerned, the 6820and 6821 are the same. The Dragon and TRS-80 Color Computer each have two 6821 PIAs, occupying the same locations and configured similarly in both systems. The uses made by the Dragon of the PIA illustrate its extreme versatility so after a brief and, I hope, not too technical description of the PIA, we will look at what the Dragon does with it.

The PIA consists of two 8-bit ports, A and B, which can be considered identical for most purposes. Port A is usually configured for input and port B for output of data. Each port has three registers and occupies two memory locations as shown in Table 4.1, and this means that normal Read/Write memory is absent at the addresses used by the PIA. It is the usual practice to locate PIAs and other memory mapped devices well away from User-RAM. Nevertheless, you must always be careful, especially when using Indexed or Indirect addressing, not to write data accidentally to an I/O or control device or you could crash the system.

Since the Peripheral and Data Direction registers share the same address, both cannot be used at the same time. There is a good reason for this. The DR bits determine whether the corresponding PR bits are input or output. If a DR bit is *reset* (0) then the same-place bit in the PR is an *input* bit; if the DR bit is *set* (1) then the PR bit is an *output* bit. Consequently the PR can be set to include a mixed pattern of input and output bits. Once the system has been initialised to a specific PR input/output configuration, the DR can be hidden behind the PR to ensure that no accidental change takes place. Obviously there has to be some way of selecting which of the DR and PR you want to address and bit 2 in the port Control Register (CR-2) is used to switch between the Direction and Peripheral Registers, as shown in Table 4.2. RESET of the system (as at power-up) clears the CR, thus automatically selecting the DR ready for initialisation.

The following code sequence will configure a PIA port A so that bits 7 to 4 of the Peripheral Register are input lines and bits 3 to 0 are output lines. Note that bit 2 of the Control Register must be cleared (reset) and set using bit logic operations so that no other CR bits are affected.

```
LDA     CRA         ;clear CRA-2 to address DRA
ANDA    #%11111011  ;without changing any
STA     CRA         ;other CRA bits.
LDA     #%00001111  ;make PRA-7 to 4 input and
STA     DRA         ;PRA-3 to 0 output by
LDA     CRA         ;writing to DRA. Re-address
ORA     #%00000100  ;PRA without changing
STA     CRA         ;other CRA bits.
```

Table 4.1. PIA registers with Dragon addresses.

Port	*Registers*	*Dragon Addresses* PIA 0	PIA 1
A	Peripheral Register (PRA) *or* Data Direction Register (DRA)	$FF00	$FF20
	Control Register (CRA)	$FF01	$FF21
B	Peripheral Register (PRB) *or* Data Direction Register (DRB)	$FF02	$FF22
	Control Register (CRB)	$FF03	$FF23

Table 4.2. PIA register select (CR-2).

CR bit 2	*Register selected*
0	Data Direction Register
1	Peripheral Register

Now we can input and output up to eight bits of data at a time to a peripheral (keyboard, remote terminal, disk drive, etc.) by reading from or writing to a Peripheral Register in the PIA. The problem is knowing *when* to send the data or receive it. A 6809 CPU running at 2 MHz can transfer data from memory through an output configured PR at speeds in excess of 180000 bytes a second but the peripheral on the receiving end could be a slow printer tapping away at only 12 characters a second. So how can the PIA – the chip in the middle – reconcile the two?

The A and B ports are not, in fact, limited to just the eight I/O lines running from each Peripheral Register. There are two further lines to each port, connected this time to the Control Registers. These control lines are used for *interrupt* and *handshaking* signals and may also be used to output a steady voltage.

When interrupt and/or handshaking signals are used the fast CPU can get on with other processing tasks while waiting for a 'ready' signal from a slow peripheral. If an interrupt is used the peripheral can command the CPU to stop whatever it is doing and jump to a routine dealing with the peripheral's request, resuming the interrupted task when the request has been dealt with. If interrupts are not used the program being run must periodically check for a ready signal indicating that a handshake process has begun.

In an input handshake the peripheral puts data on the data lines and a ready signal on a control line (this may cause an interrupt).

Table 4.3. Control of PIA interrupts on control lines C1 and C2.

C1	*C2*	*Control Register (CR) bit use*
flags		
CR-7	CR-6	Transition (interrupt occurred) flags. Set (1) by control line transition. Reset (0) by CPU read of Peripheral Register.
control bits		
—	CR-5	C2 input select (CR-5 = 0). C1 always input.
CR-1	CR-4	Select effective transition: If CR-1 (CR-4) = 0 then high to low. If CR-1 (CR-4) = 1 then low to high.
CR-0	CR-3	Interrupt disable/enable: If CR-0 (CR-3) = 0 then interrupts disabled. If CR-0 (CR-3) = 1 then interrupts enabled.

After the CPU has read the data it sends an 'acknowledge' signal back to the peripheral. The peripheral does not send further data until it has received the data acknowledged signal. In an output handshake the peripheral puts a signal on a control line to say that it is ready to receive data. The CPU then puts data on the data lines and a data ready signal on a control line. In both cases it is the peripheral which sends the first control signal and does the waiting.

The PIA has an automatic mode where reading data from the PRA causes a data acknowledged signal to be sent and writing data to the PRB causes a data ready signal to be sent. This is the only difference between the A and B ports.

Control line 1 is input only and is usually tied to the IRQ or FIRQ interrupt lines of the CPU. Any transition (change in the voltage level) on it sets a flag in bit 7 of the PIA Control Register (CR-7). CR-7 is cleared only by a CPU read of the Peripheral Register. Two

Table 4.4. Automatic output signals on PIA Control line 2.

Control bits		*Control line 2 (CA2 or CB2)*
CRA-4	**CRA-3**	**CA2 signals**
0	0	Low (acknowledge signal) after CPU *read* from PRA until transition on CA1.
0	1	Low for 1 cycle after CPU *read* from PRA.
CRB-4	**CRB-3**	**CB2 signals**
0	0	Low (acknowledge signal) after CPU *write* to PRB until transition on CB1.
0	1	Low for 1 cycle after CPU *write* to PRB.

bits in the Control register, CR-1 and CR-0, are used to control interrupts on line 1. Control line 2 can be either input (CR-5=0) or output (CR-5=1). As an input line it behaves like line 1 except that CR-6 is the flag and CR-4 and CR-3 the control bits. Table 4.3 summarises PIA interrupt control.

When bit 5 of the Control Register is set (1), Control line 2 is an output line and CR bits 4 and 3 serve different purposes. Bit 4 determines if the signal will be automatic (CR-4 = 0) or software controlled (CR-4 = 1). Automatic signalling is shown in Table 4.4. With line 2 under software control the signal is constantly low (0 volts) when CR-3 = 0 and constantly high (usually +5 volts) when CR-3 = 1. Control line 2 can thus be used as an off/on switch by setting bits 5 and 4 of the Control Register and writing a switch bit to CR-3: 0 = OFF, 1 = ON.

PIA and the Dragon

The Dragon keyboard is a simple matrix connected to both Peripheral Registers of PIA 0 (see Table 4.5). Any single column can be activated by writing a zero to the corresponding bit of PRB, making sure that all other bits are ones. Bits 6 to 0 of the eight bits input from PRA will be all set (ones) if no key in that column is being pressed. If a key is being pressed then a reset (0) bit in the input data identifies the keyboard row. Bit 7 of the PRA is used for joystick comparison and may be 0 or 1. The whole keyboard can be scanned by iterating eight times and writing a zero to a different column in each iteration. This way, any combination of the 52 keys can be used simultaneously and identified.

Table 4.5. PIA I/O on the Dragon keyboard.

PIA 0 PRA	*PIA 0 PRB*							
Row input	*Column output to keyboard*							
from	7	6	5	4	3	2	1	0
keyboard	↓	↓	↓	↓	↓	↓	↓	↓
7 ←	*	*	*	*	*	*	*	*
6 ←	shift	*	*	*	*	break	clear	enter
5 ←	space	→	←	↓	↑	Z	Y	X
4 ←	W	V	U	T	S	R	Q	P
3 ←	O	N	M	L	K	J	I	H
2 ←	G	F	E	D	C	B	A	@
1 ←	/	.	-	,	;	:	9	8
0 ←	7	6	5	4	3	2	1	0

* not used.

TRS-80 Color Computer keyboard has a different arrangement.

The following two subroutines check for either the BREAK key being pressed (BRKCHK) or the key identified by row and column (KEYCHK). The pattern of keys pressed has to match exactly the input patterns or Z will be returned reset. To test for any keypress, make B = 0 and A = $FF then output Z will be set if no keys are pressed and reset if any are.

```
;BRKCHK - check BREAK press. Output: Z=1 if BREAK.
;subs: KEYCHK.
BRKCHK   PSHS   D           ;Save A & B. Load A & B with
         LDA    #%10111111  ;patterns to exclusively
         LDB    #%11111011  ;identify BREAK key
         BSR    KEYCHK      ;in KEYCHK action.
         PULS   PC,D        ;exit, Z=1 if BREAK.
```

```
;KEYCHK - check key press. Input: A, B = row, cols pattern.
;output: Z=1 if key(s) pressed match pattern.
KEYCHK    PSHS   D            ;Save A & B. Write cols to PRB
          STB    $FF02        ;from B. EOR read of PRA clears
          EORA   $FF00        ;equal bits, sets different bits.
          ANDA   #%01111111   ;clear unused line, bit 7.
          PULS   PC,D         ;exit, Z=1 if match.
```

Unfortunately, the keyboard is not the only device to use PRA and PRB. The joystick fire buttons are tied into PRA-0 and PRA-1, making it hazardous to attempt using both at once. Since it is unlikely that you would want to use joysticks and keyboard at the same time, though, the PIA bits can serve two functions. Similarly, PRB is used both to activate the keyboard and for sending data out to a printer - several control signals ensuring that the two functions are not confused. Tables 4.6 and 4.7 show the uses the Dragon makes of its two PIAs.

Table 4.6. Dragon PIA 0 uses.

Bits	*Uses*
PRA-0	Keyboard row input. R. joystick fire button.
PRA-1	Keyboard row input. L. joystick fire button.
PRA-2 to 6	Keyboard row input.
PRA-7	Joystick comparator.
CRA-0 } CRA-1 }	TV horizontal sync control bits.
CRA-2	DRA/PRA select.
CRA-3	(CA2) MUX select lo-bit. (sound, joysticks)
CRA-4	*set* } Makes line CA2 an output switch
CRA-5	*set* } under software control of CRA-3.
CRA-6	not used (CA2 is output).
CRA-7	(CA1) Horizontal sync interrupt flag.
PRB-0 to 7	Keyboard column output. Printer output.
CRB-0 } CRB-1 }	TV frame sync control bits.
CRB-2	DRB/PRB select.
CRB-3	(CB2) MUX select hi-bit. (sound, joysticks)
CRB-4	*set* } Makes line CB2 an output switch
CRB-5	*set* } under software control of CRB-3.
CRB-6	not used (CB2 is output)
CRB-7	(CB1) Frame sync interrupt flag. Timer.

Table 4.7 Dragon PIA 1 uses.

Bits	*Uses*
PRA-0	Cassette data input.
PRA-1	Printer strobe.
PRA-2 to 7	Six-bit D/A. (sound, joysticks)
CRA-0 } CRA-1 }	Printer acknowledge control bits.
CRA-2	DRA/PRA select.
CRA-3	(CA2) Cassette motor control.
CRA-4	*set* } Makes line CA2 an output switch
CRA-5	*set* } under software control of CRA-3.
CRA-6	not used (CA2 is output).
CRA-7	(CA1) Printer acknowledge flag.
PRB-0	Printer busy.
PRB-1	Single-bit sound. TV sound sensor.
PRB-2	16K/32K RAM select.
PRB-3 to 7	VDG control.
CRB-0 } CRB-1 }	Cartridge interrupt control bits.
CRB-2	DRB/PRB select.
CRB-3	(CB2) Sound enable.
CRB-4	*set* } Makes line CB2 an output switch
CRB-5	*set* } under software control of CRB-3.
CRB-6	not used (CB2 is output).
CRB-7	Cartridge interrupt (detect) flag.

The Dragon does not need a particularly complex system of interrupt and handshake signals so the designers have used Control line 2 of all four ports as switches of one kind or another. That is the reason why CR-4 and CR-5 are always set and CR-6 is not used. Writing a zero to CR-5 would make C2 an input line, disabling it as a switch. Writing a zero to CR-4 (with CR-5 = 1) would make output automatic and a signal would be sent out every time the CPU read from PRA or wrote to PRB. Changing the configuration of the C2 lines would disable sound, joysticks and cassette motor control.

The 2-bit configurations 00, 01, 10 and 11 can be written to CRB-3 and CRA-3 (Control Register bit 3 in both B and A ports of PIA 0) along with a single-bit *sound enable* (set CRB-3, PIA 1) to produce a four-state sound select system. The lines are tied to an analogue multiplexer (MUX) which selects between the 6-bit D/A converter, cassette, cartridge or a non-implemented fourth device as a sound source. The sound enable bit (CRB-3, PIA 1) is, of course, Control line 2 from the second PIA's B port used to switch on or switch off the sound. Control line 2 from the A port of PIA 1 is also used as a simple on/off switch, this time for computer control of the cassette motor (MOTORON, MOTOROFF).

This, I'm afraid, is where we must leave the Dragon's fascinating PIAs for now and move on to take a look at what SAM is doing. The Dragon and TRS-80 Color Computer books in the *Further Reading* list will tell you more about the uses made of the PIA by these two similar computers.

The Dragon and SAM

The use of the rather familiar term 'SAM' instead of his – sorry! *its* – grandiose title *Synchronous Address Multiplexer* (His Excellency, the Controller of the Dynamic RAM) reflects the fact that it is a very user-friendly chip, even though it performs a complex and sophisticated job behind the scenes. It is SAM that provides the two clock signals *E* and *Q* which keep the 6809 CPU ticking over at a steady rate. Because of this, SAM can be programmed to make the CPU run at different speeds. A word of warning here: not all Dragons can be speeded up – they are only guaranteed to work at 0.9 MHz.

The *E* clock cycle is used by SAM to control access to the dynamic RAM used for both program and screen memory. The 6809 CPU

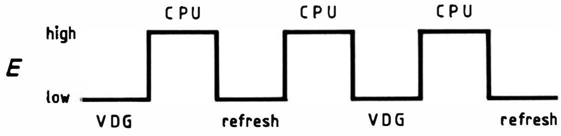


Fig. 4.1. CPU and VDG access to and refresh of dynamic RAM.

must access the memory every cycle and the Video Display Generator (VDG) must access it at least every two cycles. Also, dynamic RAM has to be *refreshed* every cycle or its contents will decay. The *multiplexing*, or interleaving of access, is performed by SAM allowing the CPU access on the high period of each *E* cycle and the VDG access on alternate low periods. VDG access also refreshes the RAM and in the remaining low periods the refresh is performed by SAM. All this is achieved by routing the address bus from the CPU through SAM and having SAM produce the necessary signals to get data on the data bus when it is needed by the VDG.

VDG access and refresh are *transparent* actions to the CPU and the programmer – they don't affect the execution of instructions in any way and they are beyond software control. Because SAM is used to emulate the VDG, however, it can be programmed to add an offset, in ½K increments, to the addresses used as screen memory. One drawback to this system is that SAM has also to be programmed to give the correct signals for the graphics mode currently operating in the VDG. Almost every time you alter the VDG mode by writing data to PRB-3 to 7 of PIA 1, the amount of memory used for display is changed and consequently you have to reprogram SAM – unless, of course, you are experimenting with strange graphics effects.

So how is SAM programmed? Not in such a complicated way as the PIAs but by a quite unusual method. SAM has sixteen single-bit registers spread out through 32 memory locations ($FFC0 to $FFDF) – two addresses for each bit! In fact the memory locations aren't there at all. SAM uses the bit patterns coming in on the address bus as a form of data and singles out $FFC0 through $FFDF as SAM control data bits. Bits 1, 2, 3 and 4 give the SAM control register number and bit 0 is the data to put in the addressed register. If bit 0 of the address is a zero the register will be cleared. If it is a one the register will be set. Any write operation to a SAM register will do the trick since the actual data written is ignored and it is the address used that matters. Table 4.8 gives the SAM register addresses for programming the display start address and other functions. Configuring SAM and the VDG (via PIA 1) is given full treatment for the TRS-80 Color Computer in *Color Computer Graphics*, available from Tandy. Although this is written primarily for programming in BASIC, addresses and data are given in hex and binary. And it is worth getting even if you own a Dragon since the machines are so similar.

Table 4.8 SAM registers.

Addresses (hex) *clear*	*set*	*Functions*	
FFC0	FFC1	Display mode. Three bit configuration.	
FFC2	FFC3	(see technical manual).	
FFC4	FFC5		
FFC6	FFC7	Display address offset (bit 9)	
FFC8	FFC9	bit 10	(write offset in the
FFCA	FFCB	bit 11	form xxxx xxx0 0000 0000
FFCC	FFCD	bit 12	where xxxx xxx are the
FFCE	FFCF	bit 13	address bits written
FFD0	FFD1	bit 14	to $FFD3 (high) down
FFD2	FFD3	bit 15	to $FFC6 (low) regs.)
FFD4	FFD5	Memory 'Page'. Keep this cleared.	
FFD6	FFD7	CPU rate (2-bit).	
FFD8	FFD9		
FFDA	FFDB	Memory size (2-bit).	
FFDC	FFDD		
FFDE	FFDF	Map type. Clear: 32K. Set: 64K.	

Block and tackle

When you consider that early mainframes got by quite happily with only 4K or so of memory, the 32K of the Dragon 32 seems quite a luxury. But while the mainframes might have been happy with only 4K their programmers were not, and even 32K on a home computer can be too restrictive for some jobs. Hence the trend to an increasing amount of immediate access memory and the advent of such beasts as the Dragon 64. But the popular 8-bit processors, such as the 6502, the Z80 and the 6809, have an address bus that is only 16 bits wide. As you must know by now, sixteen bits can address a maximum of 65536 locations – 64K of memory. So how can an 8-bit computer hold 64K or more of RAM and still have room for ROM operating systems? The answer, of course, is to put a few switches somewhere along the address bus which are operated by a memory-mapped device, and to write software that will switch to the different *banks* or *blocks* of memory as and when they are needed. Usually each block resides physically on a separate plug-in board so that the system can grow to keep pace with the user's requirements.

Bank or block switching means that you have to take a bit more care with your program design and coding. The extra memory may be there but you cannot have it all at once. There is the little matter of writing to a switch to access different parts of a program, various data storage areas or screen memory. Life can get complicated if you try and run a program resident in one block while the data it is supposed to be working on occupies another and not simultaneously addressable block. This is the memory-switch 'weatherhouse effect' and it can be very frustrating.

The weatherhouse effect shouldn't happen if you structure your programs and carefully define which blocks need to be concurrently addressed. The top level of your program should be the one to manage all the memory switching. If the program is clearly divided into separate *modules* – completely independent sub-programs – then it can occupy several blocks of memory. Only the top level *driver* program needs to be always on the bus – the modules can be switched on only when they are needed. Passengers?

The GIMIX gimmick

Dealing with large blocks of switchable memory is usually a matter for program design. When the switching cuts the display RAM into four blocks stacked on the same address space, however, the problem extends into the coding of graphics and other display access routines.

GIMIX manufacture a 6809 system which offers extended addressing facilities up to 1 *megabyte*. Before you dump your Dragon, though, I'd better warn you that the CPU board alone, with only 1K of scratchpad RAM, costs about the same as four sale-price 16K Color Computers. It isn't a home micro! The system accepts various memory boards, one of which is a 512 × 512 high resolution video board. The mapping is the normal horizontal line of 8 pixels (dots) to each byte, so the screen RAM is 64 bytes wide by 512 bytes high – 32K in all. However, the board occupies only 8K of address space. It is 'folded' into four bands.

Any one of the 262144 display dots is uniquely identified by two 9-bit co-ordinates. The horizontal (x) co-ordinate presents no problems since the 8K band addressed consists of 128 complete lines. The vertical (*y*) co-ordinate, on the other hand, has to be split before it can be converted into an address offset from the origin (see the chapter on high resolution graphics for how this conversion is done). The highest two bits (8 and 7) are used to select one of the four possible screen bands by writing them to a switch register. This leaves a 7-bit co-ordinate for conversion to the address offset and 7 bits is just enough to index any of the 128 addressed lines in the selected band.

If the valid 9-bit co-ordinate is in register D ($D < 512$) then the split is easily achieved by the three single-byte instruction sequence: ASLB : ROLA : LSRB : leaving the band select code in A and the 7-bit co-ordinate in B.

And that is how to unfold memory.

Chapter Five
Taking Control

The resident software in your computer should have routines which deal with the control of the system. Occasionally the manufacturer is kind enough to supply a list of these routines, the jobs they do, the input they require and their start addresses. Sometimes you can only find this information by disassembling the software and tediously working out the effect of long lists of uncommented instructions. It is worth the effort to do this as you may find that the computer is capable of doing much more than you thought it could.

Persor.ally, when writing machine code programs for home computers, I don't like to rely too much on firmware (system software on ROM) since it rarely does exactly what I want. It may do only part of the job or it may combine two or more tasks when my program needs only one of them. Also, routines which are part of a complete system are not usually as conscientious about conserving register values as I would like and they often assume that the index registers or the user stack pointer hold system addresses. The overheads of saving register values and loading registers with the necessary addresses before calling these routines can outweigh the benefits of using them. However, the main reason why I forego use of the supplied software is that I am then obliged to find out exactly how the computer hardware is programmed, written to or read from. Only then do I feel that I am in control of the machine.

The routines in this chapter program the Dragon's SAM and VDG chips, switch PIA C2 lines on or off and read the joysticks. Dragon BASIC contains code to do these things but possibly not in the way that you would like, or as quickly. They are not optimised for either speed or length so, if you need particularly fast operations for high-speed games programs, you can have a go at chopping out superfluous cycles. VIDEOM, for example, changes the text/graphics mode by picking up a value from a table of mode codes to write to SAM and the VDG. For absolute speed you could have a separate routine to set each graphics mode so as to make use of the rapid immediate data instructions. More importantly, if you experiment with hardware control routines you will get to know just what your computer can and cannot be made to do. So don't just use the routines supplied in ROM or given in this book without attempting to understand what they are doing.

Paging the video

VIDEOP is a routine to change the start address of the area of RAM used for screen memory. In the Dragon this address can be anywhere on a 512 byte boundary – a 16-bit address with the lowest nine bits all 0. The seven significant bits have to be written to the SAM registers occupying addresses $FFD3 down to $FFC6. The arrangement of these registers is given in Chapter 4.

Input to the routine is in the B register. The action of the routine is to isolate each bit in turn in bit 0 of the A register and use Accumulator offset addressing to write to even (if the bit is a 0) or odd (if it is a 1) addresses. Since the seven bits are the most significant bits of a 16-bit address, you might like to rewrite the routine to accept a full address in, say, the D register but use only the top seven bits. If you do that the text and graphics routines later in this book will have to be changed to meet the new standard.

VIDEOP – Video page addressing on the Dragon
Stack – 5.
I/O – B7 to B1 contains the ½K boundary number. B0 is ignored.
Notes – SAM is programmed by a write to an odd address if the bit is set, or to an even address if the bit is reset. Dragon memory $0000 to $03FF is used by the system and so is a small amount of memory at the top of user RAM. Safe start values for the highest resolution are $04 to $66.

```
VIDEOP   PSHS   X,D,CC      ;save registers used. Index SAM
         LDX    #$FFD2      ;at hi-bit register with X.
;write loop: X is decremented to index each address pair in
;turn with bit 0 of A determining write to even or odd addr.
VPLOOP   SEX                ;next addr bit all through A then
         ANDA   #%00000001  ;only in bit 0. Write to even
         STA    A,X         ;addr if 0, odd addr if 1.
         ASLB               ;next addr bit to bit 7 for SEX.
         LEAX   -2,X        ;move X to point to next SAM
```

```
CMPX   #$FFC4      ;addr-pair, repeating until
BNE    VPLOOP      ;7 bits written.
PULS   PC,X,D,CC   ;restore regs, exit VIDEOP.
```

Changing mode

Changing the text/graphics mode is more complex than changing the screen start address. Three bits have to be written to SAM at $FFC5 to $FFC0 and five bits have to be written to bits 7 to 3 of PIA 1 PRB to set the VDG.

VIDEOM gets these eight bits, combined in one byte, from a table. This table need not follow immediately after VIDEOM since

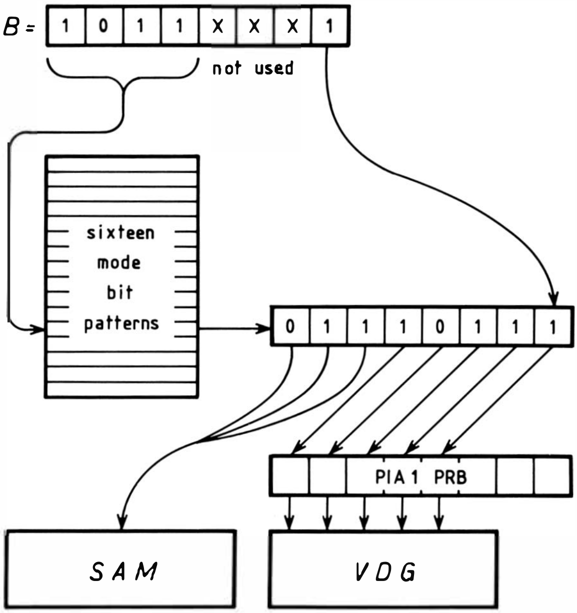


Fig. 5.1. Video mode selection.

the instruction LEAX VMTAB, PCR will load X with the address of VMTAB wherever it is. There are sixteen bytes in the table even though there are only thirteen different modes. The reason for this is that the high order digit of input B is used to index the table and it is shorter, quicker and a lot easier to include three repeated values in the table than to test for an illegal input value.

The lowest bit written to the PIA at PRB-3 is for colour set selection. In PMODE 4 this will be green on black if the bit is 0 or buff on black if it is 1. The table values have this bit (stored as bit 0) always reset and bit 0 from the input B value is merged to complete the group of five bits before they are written to the PIA. As a suggestion for improving the routine, you may like to write a separate module which writes only to PIA 1 PRB-3 to change the colour set. Then only PRB-7 to 4 should have table value bits written to them for mode selection.

VIDEOM – Video mode selection on the Dragon

Stack – 6.

I/O – Hi-nib B (high order digit of B) holds mode number, $0 to $F. B0 is the colour select bit: 0 = green, 1 = buff (or the associated colour groups).

Notes – Modes 0, 1, 2 and 3 are all Alphanumeric/inverse/semi-graphic-4 (text) mode. Mode names are given against the table but see the Dragon and Color Computer books listed in *Further Reading* for a complete description of the different modes.

```
VIDEOM    PSHS   X,D,CC        ;save registers used.
;first, index mode table and use hi-nib B as offset to pick
;up the correct mode byte, merging colour set bit with it.
          LEAX   VMTAB,PCR     ;point X to mode table start.
          TFR    B,A           ;get mode number from hi-nib B
          LSRA                 ;down into lo-nib A so that A
          LSRA                 ;gives the offset of required
          LSRA                 ;mode byte from table start.
          LSRA                 ;Then mask out all but colour
          ANDB   #%00000001    ;select bit in B and get mode
          ORB    A,X           ;merged with colour bit.
;second, highest 3 bits of mode byte written to SAM, moving
;lowest 5 bits up to B7 to 3 ready for PIA write.
          LDX    #$FFC4        ;index SAM mode addresses
VMLOOP    ASLB                 ;next mode bit into bit 0 of A
          ROLA                 ;and mask out other A bits so
          ANDA   #%00000001    ;X + A addresses even or odd
```

```
            STA     A,X             ;SAM addr to reset or set SAM
            LEAX    -2,X            ;mode bit. Repeat till 3 bits
            BNE     VMLOOP          ;written to SAM.
;third, write 5 mode bits to PIA 1 PRB to program VDG,
;not changing PRB-2 to 0. Bits 2 to 0 of B are all 0.
            PSHS    B               ;put mode on stack so bits 7 to 3
            LDA     $FF22           ;can be merged with PIA 1 PRB
            ANDA    #%00000111      ;after clearing old mode bits
            ORA     ,S+             ;out. Also remove mode from S
            STA     $FF22           ;write new mode to PRB/VDG.
            PULS    PC,X,D,CC       ;restore regs, exit VIDEOM
;table of conjoined 3-bit SAM and 5-bit VDG codes to
;set the Dragon graphics/text modes.
VMTAB       FCB     0,0,0,0         ;alpha/inverse/semi-graphics-4
            FCB     $02,$40         ;semi-graphics 6 and 8
            FCB     $80,$C0         ;semi-graphics 12 and 24
            FCB     $30,$32         ;true graphics 1F and 1T
            FCB     $54,$76         ;2F and PMODE 0
            FCB     $98,$BA         ;PMODE 1 and 2
            FCB     $DC,$DE         ;PMODE 3 and 4
```

Control switching

As stated in Chapter 4, all four of the Dragon's C2 control lines are configured as output switches. Those connected to PIA 0 are used for sound source selection (see Chapter 8) or joystick selection (dealt with later in this chapter). The two connected to PIA 1 are used for sound enable and cassette motor control.

SWITCH is a routine which writes new values to all CR-3 bits and so can deal with any C2 switching process with just one subroutine call. It takes advantage of the fact that the Dragon's PIAs do not exist only at their primary address locations but are each repeated eight times, so that PIA 0 can be written to at locations $FF1C to $FF1F as well as the normal $FF00 to $FF03. This makes all four Control Registers just two bytes apart from each other and the write can take place in a loop using the 2-byte auto-increment indexed addressing mode.

SWITCH – Write to all four PIA C2 lines on the Dragon

Stack – 5.

I/O – Bits 3 to 0 of input A contain the new values.
Bit 0 to PIA 0 CRA-3 (CA2) MUX SEL lo-bit.
Bit 1 to PIA 0 CRB-3 (CB2) MUX SEL hi-bit.

Bit 2 to PIA 1 CRA-3 (CA2) Cassette motor control.
Bit 3 to PIA 1 CRB-3 (CB2) Sound enable.

Notes – Repeat addresses of PIA 0 at $FF1D and $FF1F are used. No other CR bits are affected. All C2 lines are assumed to be output.

```
SWITCH   PSHS  X,D,CC      ;save registers used. Use X to
         LDX   #$FF1D      ;point to PIA 0 CRA.
;loop, get each Control Register contents in turn, set CR-3
;then reset it if A input bit is 0. Put back and index next.
SWLOOP   LDB   ,X          ;get current CR contents and
         ORB   #%00001000  ;always set CR-3 then shift
         LSRA              ;corresponding input bit out to
         BCS   SWNEW       ;carry, skip if set – job done –
         ANDB  #%11110111  ;else reset CR-3 to match input.
SWNEW    STB   ,X++        ;restore CR with new CR-3, bump
         CMPX  #$FF25      ;pointer to next CR, repeat till
         BNE   SWLOOP      ;4 bits written to 4 CR-3s.
         PULS  PC,X,D,CC   ;restore regs, exit SWITCH.
```

Joystick analog to digital read

The Dragon, of course, has a routine to test the current joystick positions. It is located at $BD52 (on my Dragon) and stores the joystick values in locations $015A to $015D. Registers U, X, D and CC are all changed during its execution, so if you do use the resident routine make sure that you push those registers first if they hold important values. It isn't a particularly quick routine – each joystick value (right horizontal, right vertical, left horizontal and left vertical) may be sampled up to ten times before the Dragon is happy with the result.

A lot of games use only one joystick, so it seems rather a waste of time always to test both. JOYCAB, with its two subroutine modules JOYAD and BUTTON, tests only one joystick. Which one, left or right, depends on the state of the carry flag C on input. It returns maximally useful information: C is set if the fire button is pressed, reset otherwise, for rapid BCC or BCS decisions and the six-bit joystick position values are in A7-2 (horizontal, *x*) and B7-2 (vertical, *y*) where left/right or up/down decisions can be made on the state of the negative flag N after TSTA or TSTB. Bits 1 and 0 of both accumulators contain the code for which joystick has been sampled. The modules JOYAD and BUTTON may each be called as routines in their own right to get just joystick or just fire button results.

Joystick sampling is an *analog to digital* conversion (A/D). The opposite process, *digital to analog* (D/A), is pursued at greater length in Chapter 8 but for now it is enough to know that a six-bit value written to the D/A converter (PIA 1 PRA-7 to 2) is output as a voltage which varies in direct proportion to the written digital value. The joystick horizontal or vertical movement affects the voltage allowed through a variable resistor and this is compared with the voltage output from the D/A. Bit 7 of PIA 0 PRA signals the result of the comparison. If the D/A output exceeds the joystick output then PRA-7 goes low (0), otherwise it is high (1). A/D conversion consists of a binary successive approximation of the D/A output to the compared voltage – in this case the joystick.

JOYCAB – Single joystick and fire-button read on Dragon

Modules –	JOYAD, BUTTON.
Subroutines –	SWITCH.
Stack –	5 + subroutine stack use.
I/O –	Input C = 0 for Right joystick read. C = 1 for Left joystick read. Output C = 1 if fire-button pressed, else C = 0. Bits 7 to 2 of A hold joystick horizontal (x) position value (%000000xx is far left). Bits 7 to 2 of B hold joystick vertical (y) position value (%000000yy is bottom). Bits 1 and 0 of both A and B hold joystick code: 00 = Right *x* 01 = Right *y* 10 = Left *x* 11 = Left *y*
Notes –	Modules JOYAD and BUTTON may be called as separate subroutines. The fire-buttons share PIA 0 PRA with the keyboard so it is inadvisable to use both simultaneously.

```
;JOYCAB: top level, converts input to correct form for
;module JOYAD read of x and y and components of requested joystick.

JOYCAB  LDB     #0              ;propagate carry through all bits
        SBCB    #0              ;of B and back into C, then also
        SEX                     ;through all A bits. Ensure bit
        ORA     #%00000001      ;0 of A set for y-component read.
        ANDB    #%11111110      ;B0 reset for x-component read.
        BSR     JOYAD           ;get y read in A, then
        EXG     A,B             ;into B, A getting x code
        BSR     JOYAD         ;then x read.
        BSR     BUTTON        ;fire-button state into C
        RTS                   ;and exit JOYCAB.
;
;JOYAD: low level, read joystick R or L, x or y, depending
;on A1,0: 00 = Rx, 01 = Ry, 10 = Lx, 11 = Ly. Output in
;bits 7 to 2 of A. A1,0 unchanged.
JOYAD   PSHS    CC            ;save carry flag. Mask out all
        ANDA    #%00000011    ;except code bits in A for merge
        JSR     SWITCH,PCR    ;and selection of correct joystick.
        STA     ,-S           ;"push" for later merge.
;binary successive approximation starting with $80 and if
;too small adding ½ each time, if too big subtract ½.
        LDA     #$80          ;start value, also put on stack
        STA     ,-S           ;for add/subtract value in loop.
;loop until 6-bit approximate value found.
VOLALP  STA     $FF20         ;output value to D/A and test
        TST     $FF00         ;comparator input to PRA-7
        BMI     VOLTAD        ;skip if approx. < joystick
        EORA    ,S            ;else a. > j. so clear last add
VOLTAD  LSR     ,S            ;halve the increment and add to
        ORA     ,S            ;approximation, then test if
        BITA    #%00000010    ;increment gone past 6-bit limit
        BEQ     VOLALP        ;looping until it does.
;A7 to 2 now contains 6-bit digital approximation to joystick
;position plus bit 1 set. Clear bit 1, removing increment
;from stack. Merge joystick code, removing code from stack.
        EORA    ,S+           ;clear A1 and get inc. off stack
        ORA     ,S+           ;merge code getting stack tidy
        PULS    PC,CC         ;for return to JOYCAB.
;
;BUTTON: low level, read right or left joystick fire-button,
;input C = 0 for R, C = 1 for L. Output C = 1 if button
;pressed, else C = 0.
BUTTON  PSHS    A             ;save A contents while A used to
        LDA     $FF00         ;get PIA 0 PRA for button bits.
        EORA    #%11111111    ;complement so press = 1, nopress
        BCC     BBTOC         ;is 0, no change to C so branch
        RORA                  ;can be made to shift out Left
BBTOC   RORA                  ;(bit 1) or just Right (bit 0)
        PULS    PC,A          ;into C. Return to JOYCAB.
```

Chapter Six
Versatile Graphics

The graphics commands in most computers are quite sophisticated but do they really do everything that you need? Try telling the Dragon to:

```
    LINE (0,0)-(255.255), PSET
```

If your Dragon is as daft as mine it will draw a line from (0,0) to (255,191). It won't even attempt to draw if any of the co-ordinates are less than 0 or greater than 255 – and that is not much good if you need a game shape to float on and off the screen.

The graphics suite in this chapter is somewhat rudimentary, doing only single-point plotting and straight-line drawing (no circle, box or fill routines) but it does have features which make it quite versatile. PLOT works on 16-bit co-ordinates ($8000 to $7FFF, decimal

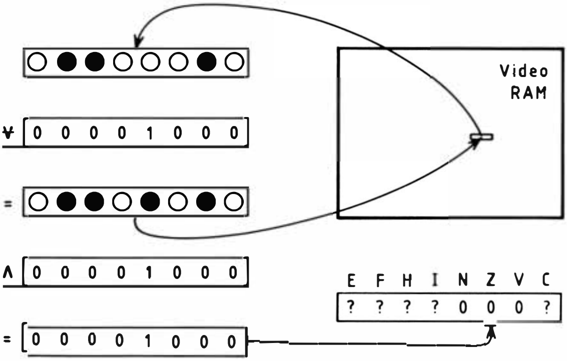


Fig. 6.1. Dot inversion and test in PLOT.

−32768 to +32767) so line figures can be drawn partly on or off the screen. Input to it, however, is by 8-bit *vectors* (offsets to the *x*,*y* coordinates of the last point plotted). This is quicker and more versatile since a string of vectors can be used to draw the same shape at different places on the screen by adjusting the start co-ordinates. Each *x* or *y* offset in the vector is in the range $80 to $7F (decimal −128 to +127) and this is quite adequate for most purposes.

The origin is at the bottom left of the display area, not the top left as in Dragon BASIC. The display area can be set to any size within the limits imposed by the screen size so a 'graphics window' can be set up on only part of the screen and the rest of the screen preserved for text.

A note on style

The graphics suite comprises several independent routines and the largest of these are split into modules. Three different forms of subroutine call are used to distinguish between the different structural relationships: (1) *BSR label* is used for routine-internal calls, that is from the top level to an integral but separately written part, (2) *LBSR label* is used for a call to a separate routine within the suite, and (3) *JSR label, PCR* is used when a routine not in the suite is called. Using these different forms can help to improve the readability of programs. However, problems crop up if a routine is large and the *BSR* form cannot be used for internal calls.

A high resolution graphics suite

PLOT – Modal, vectored plot

Modules –	PADDR, PLOTAP.
Subroutines –	VECADD, MBYBY.
Stack –	11 + subroutines.
I/O –	Vector (*x*,*y* offsets from last point) input in A,B. Essential parameters and variables in table PLTVAR. Mode determined by value at PLTVAR+15: 0 = TEST, 1 = INVERT, 2 = PLOT, 3 = UNPLOT. Output C = 0 if point outside display area. C = 1: Z = 1: point reset (off). C = 1: Z = 0: point set (on).

Notes – Written for variable sized display (maximum height of 65536 dots). Dot width must be a multiple of 8 (maximum 2040).

```
;PLOT: top level, decides if point is to be plotted.
PLOT    LBSR   VECADD       ;form new coordinates from vec
        BCC    PLTEND       ;+ last point. Only if in area,
        BSR    PADDR        ;convert coords to address and
        BSR    PLOTAP       ;plot using address info. Then
        ORCC   #%00000001   ;show plot has taken place.
PLTEND  RTS                 ;exit PLOT routine.
;
;PADDR: convert co-ordinates to absolute address and place bit.
;positive y coord is lower in memory than origin.
PADDR   PSHS   U,Y,D,CC     ;save regs used. Index PLOT
        LEAU   PLTVAR,PCR   ;variables from base U.
;convert y-coord to row LH byte address offset from origin.
        LDY    18,U         ;pick up y-coord and line-inc
        LDB    14,U         ;(addr diff between col. bytes)
        JSR    MBYBY,PCR    ;multiply them to get addr offset,
        STY    20,U         ;16-bit, put in 'address' variable.
;convert x-coord to byte offset along row.
        LDD    16,U         ;pick up x-coord and divide by
        LSRA                ;eight to give offset along row
        RORB                ;of byte location containing
        LSRA                ;required dot.
        RORB                ;(dot position in this byte
        LSRA                ;will be got from remainder
        RORB                ;of x-coord / 8).
;byte address is: origin − y-offset + x-offset.
        ADDD   8,U          ;x-offset + origin address
        SUBD   20,U         ;− y-offset gives byte addr
        STD    20,U         ;to 'address' variable.
;get set-bit byte giving dot-place in addressed byte.
        LDB    17,U         ;get x-coord lo-byte lowest
        ANDB   #%00000111   ;3 bits as rem. x-coord / 8
        LDA    B,U          ;indexing set-bit table to
        STA    22,U         ;get set-bit to 'set-bit' var.
        PULS   PC,U,Y,D,CC  ;restore, return to PLOT.
;
;PLOTAP: modal plot at addressed point. Exits with Z set
;if dot left clear, Z reset if dot is set.
PLOTAP  PSHS   U,D          ;save regs used. Index PLOT
        LEAU   PLTVAR,PCR   ;variables from base U.
        LDB    [20,U]       ;pick up display byte.
        LDA    15,U         ;get mode (0 to 3) and shift
        LSRA                ;out for branches on C and Z.
```

```
;TEST (0) is EQ,CC. INVERT (1) is EQ,CS.
;PLOT (2) is NE,CC. UNPLOT (3) is NE,CS.
         BEQ    PLAPIT          ;skip if Invert or Test, else
         ORB    22,U            ;use set-bit to set dot-bit.
PLAPIT   BCC    PLAPPB          ;skip if Test or Plot, else
         EORB   22,U            ;use set-bit to invert dot-bit.
PLAPPB   STB    [20,U]          ;put byte back to display.
         ANDB   22,U            ;use set-bit to test dot-bit.
         PULS   PC,U,D          ;restore, return to PLOT.
;
;PLTVAR: 23 bytes variables used by PLOT, VECADD, STMODE,
;STCRDS and GDRST. Display parameters here are initialised
;for Dragon graphics pages 5, 6, 7 and 8, as ¾ of full
;screen area in the centre.
PLTVAR   FCB    $80,$40,$20,$10 ;one set bit in each possible
         FCB    $08,$04,$02,$01 ;place in a byte.
         FDB    $32E4           ;origin. Bottom left of area.
         FDB    $00C0           ;display area x-dots (bytes * 8).
         FDB    $0090           ;display area y-dots (rows, lines).
         FCB    $20             ;line-inc (vertical byte diff).
         FCB    0               ;mode. Initialised to TEST.
         FDB    0               ;x-coord. (at origin).
         FDB    0               ;y-coord. (at origin).
         RMB    3               ;'address' and 'set-bit' vars.
```

VECADD – Add vector to co-ordinates and test against limits

Stack – 6.
I/O – Vector (x,y offsets) input in A,B.
Output C = 1 if new coordinates in display area,
else C = 0 if out of limits.
Notes – Lower limit is always 0.

```
VECADD   PSHS   U,Y,X           ;save regs used. Index PLOT
         LEAU   PLTVAR,PCR      ;variables from base U.
         LDX    16,U            ;pick up x and y coordinates
         LDY    18,U            ;in index registers X and Y
         LEAX   A,X             ;to use signed addition
         LEAY   B,Y             ;instructions to add 8-bit vector
         STX    16,U            ;offsets to 16-bit coords. Then
         STY    18,U            ;replace new coords.
;clear carry if either x or y coord is too big for display
;area. Also if too small as unsigned compare treats
;negative values as high positive values.
         CMPX   10,U            ;test x-coord against x-dots,
         BHS    VCAEND          ;skip, C=0, if outside limit.
```

```
        CMPY   12,U          ;else test y-coord with y-dots.
VCAEND  PULS   PC,U,Y,X      ;exit C=1 if both coords okay.
```

LINE – Modal, vectored straight line

Modules –	LINIT, LINEDO.
Subroutines –	PLOT.
Stack –	9 + subroutine.
I/O –	Vector input in A,B determines end-point of line drawn from last coordinate position.
Notes –	LINIT builds up an 8-byte table of variables to be used by LINEDO. For each point along the line, PLOT is called with a vector in A,B, the offsets being +1, −1, or 0.

```
;LINE: top level. Just calls modules.
LINE    BSR    LINIT         ;set up line vectors, counts, etc.
        BSR    LINEDO        ;use LINVAR values to drawline.
        RTS                  ;exit LINE routine.
;
;LINIT: compute LINE variables – absolute values of input
;offsets, sorted into Greater and Lesser; vectors for sending
;to PLOT: (a) Step-vec: both coordinates changed, (b) Normal-
;vec: just greater offset coordinate changed; dot-count:
;number of dots to plot; step-count: when to send step-vec.
LINIT   PSHS   U,D,CC        ;save regs used. Index LINE
        LEAU   LINVAR,PCR    ;variables from base U.
;store offsets. Initialise step/norm-vecs to +1.
        STD    ,U            ;store A,B offsets in offset
        LDD    #$0101        ;vars. Put +1 in both x and y
        STD    2,U           ;offsets in step-vec and
        STD    4,U           ;norm-vec initially.
;get absolute A offset, correcting vecs to −1 if necessary.
        TST    ,U            ;if end of line x-offset (A input)
        BPL    LITSTY        ;is positive then skip, else
        NEG    ,U            ;make it positive and
        NEG    2,U           ;correct step-vec and norm-vec
        NEG    4,U           ;to −1 for left-going line.
;do same for B offset.
LITSTY  TST    1,U           ;if end of line y-offset (B input)
        BPL    LIGXY         ;is positive then skip, else
        NEG    1,U           ;get absolute value and
        NEG    3,U           ;correct step-vec and norm-vec
        NEG    5,U           ;to −1 for down-going line.
;greatest e-o-l offset in ,U. Clear lesser offset norm-vec.
LIGXY   LDD    ,U            ;pick up x,y e-o-l offsets in A,B
```

```
         CMPA  1,U            ;test for y>x
         BLO   LIXLTY         ;skip to exchange if it is, else
         CLR   5,U            ;norm-vec y-offset is 0 for
         BRA   LIGLO          ;long horizontals.
LIXLTY   CLR   4,U            ;y>x so clear norm-vec x-offset
         EXG   A,B            ;for long verticals. Get greatest
LIGLO    STD   ,U             ;e-o-l offset in ,U. lesser in 1,U.
;dot-count is greater offset. Step count is< half of it.
         STA   6,U            ;dot-count is greater offset.
         DECA                 ;step-count starts at greatest
         LSRA                 ;integer< half greater offset.
         STA   7,U            ;
         PULS  PC,U,D,CC      ;restore, return to LINE.
;
;LINEDO: draw straight line using LINVAR variables.
;action: for dot-count, subtract lesser offset from step-
;count and if result positive then change greater offset
;coord else add greater offset to step-count and change
;both coords.
LINEDO   PSHS  U,D,CC         ;save regs used. Index LINE
         LEAU  LINVAR,PCR     ;variables from base U.
         TST   6,U            ;no line to draw if dot-count
         BEQ   LDEND          ;is 0 so end immediately, else
;test whether step necessary (both coords inc'd/dec'd)
LDDTLP   LDA   7,U            ;get step-count and subtract
         SUBA  1,U            ;lesser offset, step needed if
         BCS   LDSTEP         ;result gone below zero.
;positive: no step, change only greater offset coord.
         STA   7,U            ;re-store step-count and pick up
         LDD   4,U            ;norm-vec in A,B ready for
         BRA   LDPLOT         ;vectored plot.
;below zero: step, add greater offset to count, change both
LDSTEP   ADDA  ,U             ;coords. Add offset to make it
         STA   7,U            ;positive again before storing
         LDD   2,U            ;pick up step-vec in A,B for ...
LDPLOT   LBSR  PLOT           ;call to plot at vector A,B.
         DEC   6,U            ;repeat for dot-count, leaving
         BNE   LDDTLP         ;coords at line end point.
LDEND    PULS  PC,U,D,CC      ;restore, return to LINE.
;
;LINVAR: variables used by LINEDO and built up by LINIT
         RMB   8              ;see LINIT and LINEDO.
```

STMODE – Set plot mode
Stack – 2.

I/O – Input B is mode (0 to 3).
Notes – Not worth indexing PLTVAR with U.

```
STMODE  PSHS   B,CC            ;preserve regs.
        ANDB   #%00000011      ;mask out unused bits and store
        STB    PLTVAR+15,PCR   ;at mode variable in PLTVAR.
        PULS   PC,B,CC         ;restore regs, end.
```

STCRDS – Set coordinates
Stack – 1.
I/O – Input X,Y hold new x,y coordinates (16-bit).

```
STCRDS  PSHS   CC              ;STore instr. affects flags.
        STX    PLTVAR+16,PCR   ;write new co-ordinates to
        STY    PLTVAR+18,PCR   ;coord variables in PLTVAR.
        PULS   PC,CC           ;restore regs, end.
```

GDRST – Reset graphics display area
Modules – CRDRST, GDCLR.
Stack – 15.
I/O – No input needed, uses PLTVAR.
Notes – Clearing a full screen is just clearing each byte from one address to another. Clearing a window means clearing just one row (line) at a time inside a loop and this is slower. The modules, CRDRST and GDCLR may each be called separately.

```
;GDRST: top level. makes 'clear display area' and 'reset
;coords to origin' into just one subroutine call.
GDRST   BSR    CRDRST          ;reset coordinates to (0,0).
        BSR    GDCLR           ;clear display area.
        RTS                    ;exit GDRST routine.
;
;CRDRST: reset coords to origin by clearing.
CRDRST  PSHS   U,CC            ;save regs. Index coordinate
        LEAU   PLTVAR+16,PCR   ;variables in PLTVAR by U.
        CLR    ,U+             ;clear x-coord hi-byte, index
        CLR    ,U+             ;lo-byte, clear it and index
        CLR    ,U+             ;y-coord hi-byte, clear, index
        CLR    ,U              ;lo-byte and clear it.
        PULS   PC,U,CC         ;restore, return to GDRST.
;
;GDCLR: reset all bits inside display window. Nested loops,
;inner loop clears one display line, outer loop sets start
;address to each line in turn.
GDCLR   PSHS   U,Y,X,D,CC      ;save regs used. Index PLOT
```

```
        LEAU  PLTVAR,PCR       ;variables from base U.
;get bytes per row in X and no. of rows (lines) in Y.
        LDD   10,U             ;get x-dots and divide by 8
        LSRA                   ;to give number of bytes of
        RORB                   ;display locations in each
        LSRA                   ;line of the window.
        RORB                   ;x-dots must be a multiple
        LSRA                   ;of 8 or window not fully
        RORB                   ;cleared.
        TFR   D,X              ;X = bytes per row.
        LDY   12,U             ;Y = y-dots = no. of rows.
;get offset to move pointer up to next row, start at origin.
        LDD   #0               ;get negative line-inc to move
        SUBB  14,U             ;up screen = down in memory.
        SBCA  #0               ;line-inc stored as 1 byte.
        LDU   8,U              ;U now origin address as pointer.
;loops: clear rows from bottom to top of window.
GCRLP   PSHS  U,X              ;save LH address and byte count.
GCBLP   CLR   ,U+              ;clear row byte, pointing to next
        LEAX  -1,X             ;and repeat till all bytes in
        BNE   GCBLP            ;current row cleared.
        PULS  U,X              ;restore row LH address and byte
        LEAU  D,U              ;count. Move point up to next row
        LEAY  -1,Y             ;and repeat till all rows in
        BNE   GCRLP            ;window processed.
        PULS  PC,U,Y,X,D,CC    ;restore, ret to GDRST.
```

GSTRNG – Process a program embedded string of graphics commands

Subroutines – STMODE, VECADD, PLOT, LINE.

Stack – 8 + subroutines.

I/O – Stacked return address (to calling program) is used as pointer to the string of graphics commands which must immediately follow the BSR GSTRNG or JSR GSTRNG.

Commands:

0 string terminator, exit GSTRNG
1 STMODE – 1-byte PLOT mode follows
2 VECADD – 2-bytes, x,y vector, follows
3 PLOT – 2-bytes, x,y vector, follows
4+LINE – 2-bytes, x,y vector follows

Program return is to byte following terminator.

Notes – Graphics strings may be stored in an area separate from the program but each must have a preceding

JSR GSTRNG and have RTS after the null (0) terminator. The program then calls the string, not GSTRNG.

```
GSTRNG    PSHS   X,D,CC       ;save regs used. Get return address
          LDX    5,S          ;from stack as string pointer
          LDA    ,X+          ;get 1st code, point to next byte,
          BEQ    GSEND        ;exit if terminator, null string.
;command loop; 1st test for mode needing only 1 byte.
GSCLP     DECA                ;zero if command is STMODE
          BNE    GSNMOD       ;so test further if not,
          LDB    ,X+          ;get MODE and move point then
          LBSR   STMODE       ;go set new mode and
          BRA    GSNXT        ;go get next command.
;other commands have 2 bytes following which need to be
;passed on in A,B so the command code is put on stack.
GSNMOD    PSHS   A            ;put code on stack and pick up
          LDD    ,X++         ;vector in A,B moving pointer
          DEC    ,S           ;past them. Test for VECADD
          BNE    GSNVEC       ;and skip if not, else
          LBSR   VECADD       ;go add vector to coords, then
          BRA    GSCSNC       ;go clear stack, get next code.
GSNVEC    DEC    ,S           ;test for PLOT or LINE
          BNE    GSNPLT       ;skip if code was 4 or more
          LBSR   PLOT         ;go plot at vector then go clear
          BRA    GSCSNC       ;stack and get next command.
GSNPLT    LBSR   LINE         ;draw line, etc.
;clear command byte off stack when VECADD, PLOT or LINE
GSCSNC    PULS   A            ;crash prevention!
GSNXTC    LDA    ,X+          ;nextcommand, move pointer, and
          BNE    GSCLP        ;repeat if not null (0) terminator.
GSEND     STX    5,S          ;stack pointer for return to
          PULS   PC,X,D,CC    ;byte after terminator.
```

An example for the Dragon

GXMPLP is a short program showing how the graphics suite can be used. The first part of the program initialises SAM and the VDG, clears the display area (the window described in PLTVAR) and sets the plot mode to 1 (INVERT). Inverting dots has the useful property that if you go over the same points twice they are first set and then cleared. Put a delay between the setting and clearing and you have a 'frame', say about $^1/_{10}$ of a second. The next 'frame' can be plotted at a slightly different place – and hey presto! moving pictures!

GXMPLP and the routines called by it are all relocatable. Assemble them anywhere and use BASIC EXEC to call GXMPLP. The text screen reappears on return to BASIC.

```
;GXMPLP: graphics suite demonstration program for Dragon.
;stack - 11 + subs.
GXMPLP   PSHS   Y,X,D,CC         ;save regs used by program.
;initialise SAM and VDG (via PIA) by chapter 5 routines.
         LDB    #$1E             ;use graphics-pages
         JSR    VIDEOP,PCR       ;5, 6, 7 and 8.
         LDB    #$F1             ;set SAM and VDG for
         JSR    VIDEOM,PCR       ;PMODE 4, buff.
;set mode to Invert, clear window, init. coords & loop count.
         LDB    #1               ;mode 1 is INVERT dots.
         JSR    STMODE,PCR       ;
         JSR    GDRST,PCR        ;clear window.
         LDX    #$FFFF           ;set frame start-coords
         LDY    #$FFFF           ;to (-1, -1).
         LDD    #$0040           ;frame loop count.
;frame loop: set coords, draw shape, delay, draw shape.
GXFLP    JSR    STCRDS           ;set shape start coords
         BSR    GXMPLS           ;and draw string to set dots.
;delay for about 1/10 second.
         PSHS   X                ;save X (x-coord) for use as
         LDX    #$3000           ;delay loop counter.
GXDLP    LEAX   -1,X             ;do nothing but use up
         BNE    GXDLP            ;time for about 1/10 second
         PULS   X                ;then get x-coord back and
         BSR    GXMPLS           ;draw string in same place to
         LEAX   4,X              ;clear dots. Move shape start
         LEAY   3,Y              ;coords for next frame
         SUBD   #1               ;and repeat for 64 frames.
         BNE    GXFLP            ;
         PULS   PC,Y,X,D,CC      ;return to BASIC.
;
;GXMPLS: graphics command string. Last command before
;terminator is vector to set coords back to start point.
GXMPLS   JSR    GSTRNG,PCR       ;go process following string.
         FCB    $04,$1D,$00,$04,$00,$E3,$04,$E3,$00
         FCB    $04,$00,$1D,$03,$06,$EC,$03,$00,$FF
         FCB    $04,$04,$FC,$04,$08,$00,$00,$04,$04,$04
         FCB    $03,$00,$01,$02,$F8,$0A,$04,$FC,$F7
         FCB    $04,$08,$00,$04,$FC,$09,$02,$F5,$02
         FCB    $04,$08,$00,$04,$FE,$FC,$04,$FA,$04
         FCB    $02,$0E,$00,$04,$08,$00,$04,$FA,$FC
FCB     $04,$FE,$04,$03,$04,$FE,$03,$F3,$00
FCB     $02,$F8,$0A,$00 ;null (0) terminator.
RTS                     ;return to GXMPLP.
```

Optimising for the Dragon

The graphics suite doesn't operate as fast as it could. Indeed, the routines are quite a lot slower than those used in Dragon BASIC for several reasons. Here are the three most important, along with suggestions for increasing speed.

(1) The routines in the suite are written for structure, clarity and multi-system use, which you will have to sacrifice for optimum speed. Does PLOT need to have a top level and two lower levels as well as a subroutine call to VECADD? Write it as one long sequence and you will cut out the subroutine call instructions and the repeated register saving and U initialising – big cycle eaters all.

(2) Sixteen-bit coordinates are better than 8-bit – a line from the origin to (255,255) gets there, not to (255,191) – but they do slow things down. All is not lost, however. As plottable display coordinates must be in the range 0 to 255 ($FF) you can disregard the high order bytes of both coordinates when computing the address. The 'line-inc' doesn't change and so it can be written as immediate data in the routine instead of being picked up from PLTVAR. Try substituting the following sequence for instructions 3 to 13 in PADDR and see the speed difference.

```
LDB     19,U    ;get y-coord lo-byte (hi-byte
LDA     #32     ;must be $00 if valid) * line-
MUL             ;inc to give vertical offset
STD     20,U    ;from origin to "address".
LDD     16,U    ;get x-coord. Hi-byte = 0 so
LSRB            ;just divide lo-byte by 8 for
LSRB            ;valid byte offset along row
LSRB            ;in D.
```

(3) If you are happy with a top-left origin you can take advantage of the fact that multiplying the *y* coordinate by 32 (line-inc) can be done at the same time as dividing the *x* coordinate by 8 since y * 32 = y * 256 / 8. Lines 3 to 16 of PADDR can be replaced by the following code. Don't forget to change the origin address in PLTVAR and also change GDCLR so that D holds a *positive* line-inc value. GXMPLS will, of course, be upside down.

```
LDA    19,U   ;D gets y-coord * 256 +
LDB    17,U   ;x-coord then is divided
LSRA          ;by 8 to give both vertical
RORB          ;and horizontal address
LSRA          ;offsets from the origin
RORB          ;at the same time in one
LSRA          ;16-bit value. Ready for
RORB          ;adding origin address.
```

There are one or two other tricks that can be used to speed up PLOT and the other routines in the suite. If you discover them all you will find that 16-bit co-ordinates can be very nearly as fast as 8-bit on the Dragon – and much more versatile. Good hunting!

Chapter Seven
High Resolution Text

Owners of the Dragon or TRS-80 Color Computer should find this chapter very useful. The character set used in both these computers suffers from three gross disabilities: (1) it lacks lower-case letters and many standard symbols, (2) it is not ASCII coded, and (3) it cannot be used in the high resolution graphics modes. Not much can be done through BASIC about these problems but they can be entirely overcome in machine code. In fact we completely disregard the character set buried inside the VDG and instead use an area of memory to store character shaped bit patterns as a *user-definable* character set. Machine code is so fast and powerful that even having to write eight bytes to the screen for each character is quick enough to fill the Dragon's high resolution screen with 768 characters (24 lines of 32) in the blink of an eye.

The print routine

Like the plot routine of the last chapter, TPRINT can be set to four different modes of operation. The first (mode 0) is the normal print mode where the eight bytes containing the character bit patterns are simply written to eight vertically adjacent screen locations. In this mode the character sent to the print routine (source character) replaces that already on the screen (destination character). In the other three modes logical operations are used to produce various combinations of the source and destination characters. These modes are intended primarily for use with pre-defined graphics shapes rather than alphanumeric characters.

Mode 1 uses the logical AND to set (light up) each dot only if corresponding source and destination dots are both set. In mode 2 (EOR) any set bits in the source character patterns produce a change from set to reset, or reset to set, in the destination but source bits

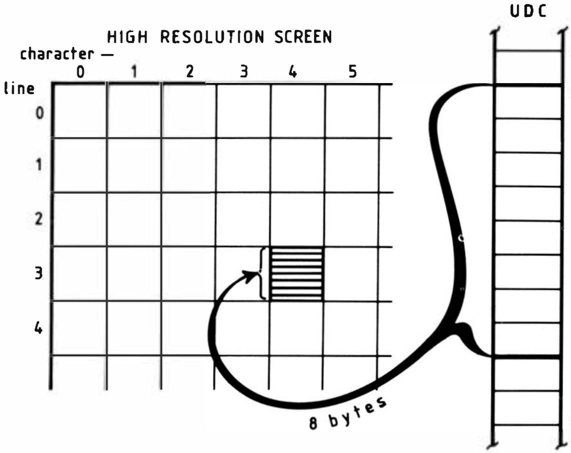


Fig. 7.1. Writing a UDC character to screen line 3, char. 4.

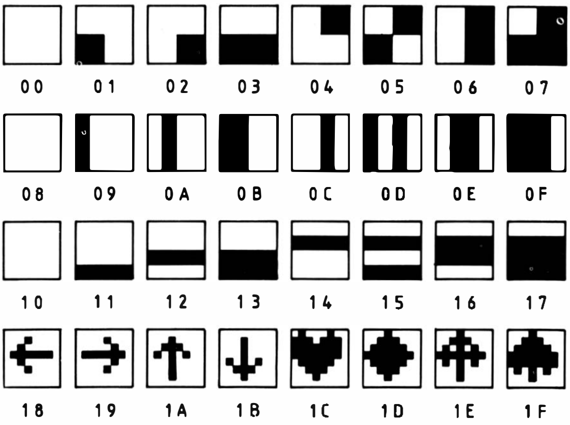


Fig. 7.2. UDC 'control code' graphics.

Table 7.1. Character bit-pattern hex codes.

ASCII		*Character bytes (hex)*							
Hex	*Dec*	*Top*	2	3	4	5	6	7	8
00	0	00	00	00	00	00	00	00	00
01	1	00	00	00	00	F0	F0	F0	F0
02	2	00	00	00	00	0F	0F	0F	0F
03	3	00	00	00	00	FF	FF	FF	FF
04	4	0F	0F	0F	0F	00	00	00	00
05	5	0F	0F	0F	0F	F0	F0	F0	F0
06	6	0F	0F	0F	0F	0F	0F	0F	0F
07	7	0F	0F	0F	0F	FF	FF	FF	FF
08	8	00	00	00	00	00	00	00	00
09	9	C0	C0	C0	C0	C0	C0	C0	C0
0A	10	30	30	30	30	30	30	30	30
0B	11	F0	F0	F0	F0	F0	F0	F0	F0
0C	12	0C	0C	0C	0C	0C	0C	0C	0C
0D	13	CC	CC	CC	CC	CC	CC	CC	CC
0E	14	3C	3C	3C	3C	3C	3C	3C	3C
0F	15	FC	FC	FC	FC	FC	FC	FC	FC
10	16	00	00	00	00	00	00	00	00
11	17	00	00	00	00	00	00	FF	FF
12	18	00	00	00	00	FF	FF	00	00
13	19	00	00	00	00	FF	FF	FF	FF
14	20	00	00	FF	FF	00	00	00	00
15	21	00	00	FF	FF	00	00	FF	FF
16	22	00	00	FF	FF	FF	FF	00	00
17	23	00	00	FF	FF	FF	FF	FF	FF
18	24	00	10	20	7E	20	10	00	00
19	25	00	08	04	7E	04	08	00	00
1A	26	00	10	38	54	10	10	10	00
1B	27	00	10	10	10	54	38	10	00
1C	28	44	EE	FE	FE	7C	38	10	00
1D	29	10	38	7C	FE	7C	38	10	00
1E	30	10	38	54	FE	54	10	10	00
1F	31	10	38	7C	FE	FE	54	10	00
20	32	00	00	00	00	00	00	00	00
21	33	10	10	10	10	10	00	10	00
22	34	24	24	24	00	00	00	00	00
23	35	24	24	7E	24	7E	24	24	00
24	36	10	3C	50	38	14	78	10	00
25	37	60	64	08	10	20	4C	0C	00
26	38	10	28	28	10	2A	44	3A	00

27	39	10	10	10	00	00	00	00	00
28	40	10	20	40	40	40	20	10	00
29	41	10	08	04	04	04	08	10	00
2A	42	10	54	38	10	38	54	10	00
2B	43	00	10	10	7C	10	10	00	00
2C	44	00	00	00	00	00	10	10	20
2D	45	00	00	00	7C	00	00	00	00
2E	46	00	00	00	00	00	00	10	00
2F	47	00	04	08	10	20	40	00	00
30	48	38	44	4C	54	64	44	38	00
31	49	10	30	10	10	10	10	38	00
32	50	38	44	04	18	20	40	7C	00
33	51	7C	04	08	18	04	44	38	00
34	52	08	18	28	48	7C	08	08	00
35	53	7C	40	78	04	04	44	38	00
36	54	1C	20	40	78	44	44	38	00
37	55	7C	04	08	10	20	20	20	00
38	56	38	44	44	38	44	44	38	00
39	57	38	44	44	3C	04	08	70	00
3A	58	00	00	10	00	10	00	00	00
3B	59	00	00	00	10	00	10	10	20
3C	60	08	10	20	40	20	10	08	00
3D	61	00	00	7C	00	7C	00	00	00
3E	62	20	10	08	04	08	10	20	00
3F	63	38	44	04	18	10	00	10	00
40	64	38	44	54	5C	58	40	3C	00
41	65	10	28	44	44	7C	44	44	00
42	66	F8	44	44	78	44	44	F8	00
43	67	38	44	40	40	40	44	38	00
44	68	F8	44	44	44	44	44	F8	00
45	69	7C	40	40	78	40	40	7C	00
46	70	7C	40	40	78	40	40	40	00
47	71	38	44	40	40	4E	44	3C	00
48	72	44	44	44	7C	44	44	44	00
49	73	38	10	10	10	10	10	38	00
4A	74	0E	04	04	04	04	44	38	00
4B	75	44	48	50	70	50	48	44	00
4C	76	40	40	40	40	40	40	7C	00
4D	77	C6	AA	92	92	82	82	82	00
4E	78	44	44	64	54	4C	44	44	00
4F	79	38	44	44	44	44	44	38	00
50	80	78	44	44	78	40	40	40	00
51	81	38	44	44	44	54	48	34	00
52	82	78	44	44	78	50	48	44	00
53	83	38	44	40	38	04	44	38	00

54	84	7C	10	10	10	10	10	10	00
55	85	44	44	44	44	44	44	38	00
56	86	44	44	44	28	28	10	10	00
57	87	82	82	82	92	92	AA	C6	00
58	88	44	44	28	10	28	44	44	00
59	89	44	44	44	28	10	10	10	00
5A	90	7C	04	08	10	20	40	7C	00
5B	91	7C	60	60	60	60	60	7C	00
5C	92	00	40	20	10	08	04	00	00
5D	93	7C	0C	0C	0C	0C	0C	7C	00
5E	94	00	00	10	28	44	00	00	00
5F	95	00	00	00	00	00	00	7C	00
60	96	20	10	08	00	00	00	00	00
61	97	00	00	38	04	3C	44	3C	00
62	98	40	40	78	44	44	44	78	00
63	99	00	00	3C	40	40	40	3C	00
64	100	04	04	3C	44	44	44	3C	00
65	101	00	00	38	44	7C	40	38	00
66	102	0C	10	10	7C	10	10	10	00
67	103	00	00	3C	44	44	3C	04	38
68	104	40	40	78	44	44	44	44	00
69	105	10	00	30	10	10	10	38	00
6A	106	10	00	10	10	10	10	10	60
6B	107	20	20	24	28	30	28	24	00
6C	108	30	10	10	10	10	10	38	00
6D	109	00	00	EC	92	92	92	92	00
6E	110	00	00	78	44	44	44	44	00
6F	111	00	00	38	44	44	44	38	00
70	112	00	00	78	44	44	78	40	40
71	113	00	00	3C	44	44	3C	04	06
72	114	00	00	2C	30	20	20	20	00
73	115	00	00	3C	40	38	04	78	00
74	116	10	10	7C	10	10	10	0C	00
75	117	00	00	44	44	44	44	3C	00
76	118	00	00	44	44	28	28	10	00
77	119	00	00	82	92	92	92	6C	00
78	120	00	00	44	28	10	28	44	00
79	121	00	00	44	44	44	3C	04	38
7A	122	00	00	7C	08	10	20	7C	00
7B	123	08	10	10	20	10	10	08	00
7C	124	10	10	10	10	10	10	10	00
7D	125	20	10	10	08	10	10	20	00
7E	126	00	00	32	4C	00	00	00	00
7F	127	18	24	20	78	20	22	7C	00

which are reset leave the destination unchanged. One important use of mode 2 is where the screen can be initially cleared to either white (green or buff in the Dragon) or black since the foreground character will always print as the reverse of the screen colour. Mode 3 is OR which overlays the source character on the destination.

TPRINT gets its source character dot patterns from a 1024-byte table, UDC (User Definable Characters), stored in RAM. Table 7.1 gives codes to produce the complete ASCII character set with various graphics shapes for the ASCII control codes $00 to $1F. You will probably find it easier to type large tables of numbers into the computer as hex digits, through a machine code monitor, rather than as FCB directives in an assembly language program.

ASCII uses only 7-bit codes with the highest bit of each byte reset (0), so bit 7 is used by TPRINT as an 'inverse character' flag. If the input character code byte has bit 7 set then the eight bytes picked up from UDC are complemented to change all 1s to 0s and all 0s to 1s before being written to screen memory. Ordinary characters have codes $00 to $7F and inverse characters have codes $80 to $FF.

You don't have to use the character patterns of Table 7.1, of course, since any shape which fits an 8 by 8 dot matrix can be written in UDC. You could even set up more than one UDC table in memory and switch the UDC address stored in TXTVAR (variables used by the text suite) between ASCII and pre-defined graphics. The text suite can be used as an easy way of moving small games shapes around on the screen.

TPRINT – Modal, high resolution display print

Modules –	TCHARY, TDISPX, TWRITE.
Subroutines –	TVALID, TRIGHT.
Stack –	4 + subroutines.
I/O –	Input B is character code. Bit 7 of B is character inverse flag (1 = inverse). Character dot patterns in RAM table UDC. Essential parameters and variables in table TXTVAR. Mode determined by value at TXTVAR+12: 0 = REPLACE, 1 = AND, 2 = EOR, 3 = OR. Output: character modally printed and cursor (TXTVAR+8,9) moved to next valid print position.
Notes –	Written for a variable sized high resolution display. Each character is an 8 × 8 dot matrix (one byte wide by 8 bytes deep) on the screen. Margins (undisplayed locations at the end of each screen row) are not allowed for.

```
;TPRINT: top level, saves registers used in modules,
;initialises U to index TXTVAR.
TPRINT    PSHS  U,Y,X,D,CC       ;save regs used. Index text
          LEAU  TXTVAR,PCR       ;variables from base U.
          LBSR  TVALID           ;ensure cursor is on screen.
          BSR   TCHARY           ;index UDC char patterns in Y.
          BSR   TDISPX           ;X indexes screen locations.
          BSR   TWRITE           ;move UDC char to screen and
          LBSR  TRIGHT           ;move cursor to next position.
          PULS  PC,U,Y,X,D,CC    ;exit TPRINT routine.

;TCHARY. Y points to first byte of 8 bytes in UDC giving
;dot patterns for character in B.
TCHARY    TFR   D,Y              ;save D in Y. Strip inverse flag
          ANDB  #%01111111       ;from char code. Code * 8 for
          LDA   #8               ;set of 8 bytes corresponding to
          MUL                    ;char code in UDC. Add UDC base
          ADDD  10,U             ;address to give char bytes
          EXG   D,Y              ;address in Y, restoring D.
          RTS                    ;return to TPRINT.
;
;TDISPX: X points to the top of 8 screen locations which
;correspond to the character position indexed by the cursor.
TDISPX    TFR   D,X              ;save D in X. Pick up no. of chars
          LDD   7,U              ;per line in A, line offset in B.
          MUL                    ;Address offset of top byte of
          ASLB                   ;leftmost char on cursor line is
          ROLA                   ;computed from
          ASLB                   ;cursor line offset *
          ROLA                   ;no. of chars per line *
          ASLB                   ;8 hi-res rows per character.
          ROLA                   ;Then add cursor char offset
          ADDB  9,U              ;to index top byte of 8 vertical
          ADCA  #0               ;locations char position. Add
          ADDD  ,U               ;screen start to give actual
          EXG   D,X              ;address into X, restoring D.
          RTS                    ;return to TPRINT.
;
;TWRITE: move 8 sequential bytes from UDC to 8 vertically
;sequential screen locations. Invert char bits if bit 7,B
;is set. Combine source (UDC) and destination (screen) bytes
;logically according to print-mode (TXTVAR+12).
TWRITE    SEX                    ;normal(A=$00), inverse(A=$FF).
          LDB   #8               ;pattern bytes count. Count and
          PSHS  D                ;norm/inverse used with S index.
          LDB   7,U              ;get screen row increment.
```

```
;write loop. 1st job: get UDC byte and perform inverting
;operation if bit 7,B was set (,S now = $FF).
TWLOOP   LDA   ,Y+               ;get UDC byte and index next.
         EORA  ,S                ;invert only if ,S=$FF.
;next job: test mode for logical combinations. After shift:
;REPLACE (0) is EQ,CC. AND (1) is EQ,CS.
;EOR (2) is NE,CC. OR (3) is NE,CS.
         LSR   12,U              ;combination test on mode.
         BEQ   TWRA              ;go REPLACE or AND
         BCC   TWEOR             ;go EOR
         ORA   ,X                ;combine all set bits in UDC and
         BRA   TWMEND            ;screen bytes.
TWEOR    EORA  ,X                ;complement screen bits if UDC
         BRA   TWMEND            ;bits are set, else leave.
TWRA     BCC   TWMEND            ;go replace, else result bit set
         ANDA  ,X                ;only if UDC and screen bits set.
TWMEND   ROL   12,U              ;restore mode.
;next job: put result byte to screen and move screen pointer
;down to next hi-res row.
         STA   ,X                ;replacement/combination to screen
         ABX                     ;add row-inc to move to next row
         DEC   1,S               ;and repeat until 8 bytes moved
         BNE   TWLOOP            ;from UDC, processed, written.
         PULS  PC,D              ;return to TPRINT.
;
;TXTVAR: 14 bytes variables used by most routines in the
;text suite. Display parameters here are initialised for
Dragon graphics pages 5, 6, 7 and 8.
TXTVAR   FDB   $1E00             ;screen start address.
         FDB   $1F00             ;2nd char line address.
         FDB   $3600             ;screen end + 1 address.
         FCB   $18               ;char lines (dot rows / 8)
         FCB   $20               ;line width (bytes per row)
         FCB   $00               ;cursor line offset (0 to 23)
         FCB   $00               ;cursor char offset (0 to 31)
         FDB   ????              ;address of UDC (you decide where)
         FCB   $00               ;print mode (at REPLACE)
         FCB   $00               ;clear mode: 0 = black, 1 = white.
```

Cursor control

TPRINT calculates the screen addresses from the cursor line and character (along the line) offsets stored in TXTVAR + 8 and +9. PRINT AT is easy to write into a program:

```
        LDA   #line          ;required line offset.
        LDB   #char          ;required character offset.
        STD   TXTVAR+8,PCR   ;set cursor for PRINT AT.
```

But in many cases a more useful method of adjusting the cursor position is to use control codes much like those used by ASCII. Eight codes, \$00 to \$07, or their 'inverse' equivalents, \$80 to \$87, are used by TCNTRL to select various actions.

\$04 to \$07 are used for single character or line shifts, TRIGHT, TLEFT, TDOWN and TUP. TDOWN is the same as 'line-feed' on a printer. Left movement is not allowed past the first character on any line and up movement can only go as far as the top line. Movement right or down can cause the screen to be scrolled. All four of these simple cursor movement routines exit through TVALID to ensure that the position is on the screen.

Null (\$00) is used by TCNTRL as a free value to which all printable characters (above \$07) are reduced for the *jump table*. Another use for the null character is as a string terminator. This is shown later in the chapter.

Codes \$03, \$02 and \$01 form a hierarchy of operations – TCARET ('carriage return', cursor to start of the line), THOME (cursor to start of the top line) and TCLEAR ('form-feed', cursor home and screen cleared). These three routines are combined in an optimised form as a single routine with three entry points. All of these control routines can be called directly by your program but accessing them by merely sending a control character through TCNTRL is by far the easier method.

TCNTRL – Control operation select routine

Subroutines –	TPRINT, TCLEAR, THOME, TCARET, TRIGHT, TLEFT, TDOWN, TUP.
Stack –	6 + subroutines.
I/O –	Code input in B is either a character to print, a null (return immediately) or a control code (see TCJMPT for control codes).
Notes –	The jump table can be extended to accommodate any more control routines used.

```
;TCNTRL: 1st part strips inverse bit from code, deals with
;null byte and reduces all non-controls to $00 for TPRINT.
TCNTRL  PSHS  X,B,CC        ;save regs used.
        ANDB  #%01111111    ;strip to normal char codes.
        BEQ   TCEND         ;exit immediately on null.
        CMPB  #7            ;test for control codes,
          BLS     TCINDX        ;skipping if B is a control,
          CLRB                  ;else index TPRINT branch.
;2nd part: X becomes jump table + 3 * control code to address
;correct long-branch instruction.
TCINDX    LEAX    TCJMPT,PCR    ;X becomes jump table base
          LEAX    B,X           ;address then add 3 * control
          ASLB                  ;code in B so X is address of
          LEAX    B,X           ;branch instr. to control routine.
          LDB     1,S           ;recover input character and
          JSR     ,X            ;call routine via jump table.
TCEND     PULS    PC,X,B,CC     ;end TCNTRL.
;
;TCJMPT: jump table for TCNTRL.
TCJMPT    LBRA    TPRINT        ;non-control codes.
          LBRA    TCLEAR        ;code 1
          LBRA    THOME         ;code 2
          LBRA    TCARET        ;code 3
          LBRA    TRIGHT        ;code 4
          LBRA    TLEFT         ;code 5
          LBRA    TDOWN         ;code 6
          LBRA    TUP           ;code 7
```

TCLEAR, THOME, TCARET – Clear screen and home cursor, home cursor or carriage return

Stack – TCLEAR: 6. THOME, TCARET: 0.
I/O – No direct input needed. TXTVAR used and affected. TXTVAR + 13 is "clear mode": 0 = black, 1 = white.
Notes – The full width of screen is cleared.

```
;TCLEAR: falls through to THOME.
TCLEAR  PSHS   U,X,D            ;save regs used. Index text
        LEAX   TXTVAR,PCR       ;variables from base X.
        LDU    4,X              ;start to clear from screen end.
        LDD    #0               ;0 to reset all screen bits
        TST    13,X             ;unless clear mode not zero
        BEQ    TCLOOP           ;for black screen, if not then
        LDD    #$FFFF           ;all bits set for white.
;clear loop pushes all reset or all set bits to screen RAM.
TCLOOP  PSHU   D                ;set/reset 16 bits moving pointer
        CMPU   ,X               ;down 2 bytes until at screen
        BNE    TCLOOP           ;start address when all cleared.
        PULS   U,X,D            ;restore regs and now home cursor.
;
;THOME: falls through to TCARET.
THOME   CLR    TXTVAR+8,PCR     ;cursor offset to top line and,
;
```

```
;TCARET: also end of THOME and TCLEAR.
TCARET CLR     TXTVAR+9,PCR  ;cursor offset to line start.
       RTS                   ;exit TCLEAR,THOME,TCARET.
```

TRIGHT, TLEFT, TDOWN, TUP - Single character cursor moves

Subroutines -	TVALID.
Stack -	0 + TVALID stack use.
I/O -	No direct input. TXTVAR used and affected.
Notes -	TDOWN may cause scrolling. TRIGHT may cause carriage-return, line-feed and scrolling.

```
;TRIGHT: move cursor one character space right.
TRIGHT   INC    TXTVAR+9,PCR  ;move cursor char offset
         BNE    TRVAL         ;okay unless 'wraparound' to
         DEC    TXTVAR+9,PCR  ;0, if so put it back
TRVAL    LBRA   TVALID        ;ensure valid screen position.
;
;TLEFT: move cursor one character space left.
TLEFT    TST    TXTVAR+9,PCR  ;if cursor char offset is not
         BEQ    TLVAL         ;already at leftmost position
         DEC    TXTVAR+9,PCR  ;move it back one space
TLVAL    LBRA   TVALID        ;ensure valid screen position.
;
;TDOWN: line-feed, cursor down to next line.
TDOWN    INC    TXTVAR+9,PCR  ;move cursor line offset
         BNE    TDVAL         ;okay unless 'wraparound' to
         DEC    TXTVAR+9,PCR  ;0, if so move it back
TDVAL    LBRA   TVALID        ;ensure valid screen position.
;
;TUP: cursor up one line.
TUP      TST    TXTVAR+8,PCR  ;if cursor line offset is not
         BEQ    TUVAL         ;already on top line then
         DEC    TXTVAR+8,PCR  ;move it up by one line
TUVAL    LBRA   TVALID        ;ensure valid screen position.
```

TVALID - Ensure cursor indexes valid screen position

Subroutines -	TSCROL.
Stack -	6 + TSCROL stack use.
I/O -	No direct input. TXTVAR used and affected.
Notes -	Excess cursor character offset causes setting to 1st position on next line. Excess line offset causes scrolling up by one line with line offset set to bottom line.

```
TVALID  PSHS   U,D           ;save regs used. Index text
        LEAU   TXTVAR,PCR    ;variables from base U.
```

```
            LDD    8,U              ;get cursor offsets in A,B.
;ensure valid char offset (in B).
            CMPB   7,U              ;char offset okay as long as it
            BLO    TVLINE           ;is less than chars per line,
            CLRB                    ;else reset to leftmost char
            INCA                    ;on next line, making sure that
            BNE    TVLINE           ;no 8-bit "wraparound" from $FF
            DECA                    ;to $00 occurs.
;ensure valid line offset (in A).
TVLINE      CMPA   6,U              ;line offset okay as long as it
            BLO    TVEND            ;is less than lines on display,
            LDA    6,U              ;else set at bottom line (1 less
            DECA                    ;than no. of lines) and scroll
            LBSR   TSCROL           ;display up one line.
TVEND       STD    8,U              ;put valid offsets back.
            PULS   PC,U,D           ;restore regs and exit.
```

TSCROL – Scroll display up one line, clearing bottom line
Stack – 8.
I/O – No direct input. TXTVAR used but not affected.
Notes – The full width of the screen is scrolled.

```
TSCROL      PSHS   U,Y,X,D          ;save regs used. Index text
            LEAU   TXTVAR,PCR       ;variables from base U.
;initialise pointers to top, leftmost bytes of first and
;second lines (1st and ninth hi-res dot rows).
            LDX    ,U               ;X is destination pointer
            LDY    2,U              ;Y is source pointer.
;scroll loop: move bytes from (Y) to (X), incrementing
;pointers to next bytes until source gone past screen RAM.
TSMLP       LDD    ,Y++             ;get source and bump pointer
            STD    ,X++             ;to destination, bump pointer
            CMPY   4,U              ;repeat until source pointer has
            BNE    TSMLP            ;gone past screen memory(end+1).
;initialise D for "clear" to black (mode 0) or white (mode 1).
            LDD    #0               ;$0000 to reset all bottom line
            TST    13,U             ;bits if mode 0
            BEQ    TSCLP            ;skip if it is,else D = $FFFF
            LDD    #$FFFF           ;to set bottom line bits.
;clear loop: set/reset bits until dest. gone past screen.
TSCLP       STD    ,X++             ;set/reset 2-bytes of destination
            CMPX   4,U              ;bumping pointer, until pointer
            BNE    TSCLP            ;at end + 1.
            PULS   PC,U,Y,X,D       ;restore regs and exit.
```

Strings and storage

Using a string handling routine is by far the best way to deal with large amounts of text. TSTRNG processes a string of characters and control codes terminating with a null byte ($00). So that as many codes as possible can be used for characters, TSTRNG will recognise control codes only if they are preceded by $80. Normally the codes are routed straight to TPRINT but on encountering $80 TSTRNG sends the next code through TCNTRL. The 'control code follows' code $80 never gets past TSTRNG.

TNSTR is a routine to fetch, or rather point to, 'named' strings held in memory. The name is really any 16-bit number, excluding $0000 which is used by TNSTR to recognise the end of the string table. You can, however, use two-letter ASCII codes for the names, and this does make programs more readable. Each string also has to have two bytes giving the string length which act as index to the following string. String tables should be set up as in this example:

```
STRTAB  FCC   'MS        ;string name "MS"
        FDB   $0008      ;length 8 bytes including null
        FCC   'Message   ;string contents
        FCB   $00        ;null string terminator
        FCC   'TX        ;string name "TX"
        FDB   $0005      ;5 bytes including null
        FCC   'text      ;string contents
        FCB   $00        ;null terminator
        FDB   $0000      ;table terminator.
```

A quite simple and readable program sequence is then all that is needed to first address any particular string and then get it printed:

```
        LDD    #'TX          ;string name to D
        LEAX   STRTAB,PCR    ;address table start
        JSR    TNSTR,PCR     ;go get string 'TX' address
        BCS    ERROR         ;error if no 'TX' string
        JSR    TSTRNG,PCR    ;go print string 'TX'.
```

The branch to an error handling routine is, of course, not necessary if you know that string 'TX' is in the table but it is good programming practice to take any possible error conditions into account.

TSTRNG -Character and control string handling routine
Subroutines – TCNTRL, TPRINT.
Stack – 4 + subroutines.

I/O – Input X points to the first byte of the string. Output X points to the byte following the string's null terminator.

Notes – Byte values $01 to $7F and $81 to $FF are normally passed on to TPRINT. Value $80 causes the next byte to be passed to TCNTRL. $00 causes exit from TSTRNG unless immediately following $80.

```
TSTRNG  PSHS   B,CC     ;save registers used.
        LDB    ,X+      ;get char and index next.
        BEQ    TSEND    ;exit immediately if null string.
;loop till terminator found.
TSLOOP  CMPB   #$80     ;is it 'control follows'?
        BNE    TSNOTC   ;skip if normal character
        LDB    ,X+      ;else get control char, bump
        LBSR   TCNTRL   ;pointer, and send char through
        BRA    TSNEXT   ;control select. Go get next.
TSNOTC  LBSR   TPRINT   ;normal chars printed.
TSNEXT  LDB    ,X+      ;get next char, bump pointer,
        BNE    TSLOOP   ;repeat till null terminator.
TSEND   PULS   PC,B,CC  ;restore and exit.
```

TNSTR – Index named string in string table

Stack – 2.

I/O – Input X addresses 1st byte of string table.
Input D contains string name
Output, string found: C = 0. D = string length
X = 1st string byte pointer.
Output, not found: C = 1. D = 0. X points to byte following table terminator.

Notes – Each string must be preceded by 4 bytes of information. Bytes 1 and 2 are the string name. Bytes 3 and 4 give the offset to the next string. Strings must end with a null terminator byte. The table must end with two null bytes.

```
TNSTR   PSHS   Y            ;save Y for use as holder of
        TFR    D,Y          ;requested name throughout.
;loop: first, test name, if found get length & set exit flag.
TNLOOP  CMPY   ,X++         ;test name, moving pointer past.
        BNE    TNTERM       ;if not name, skip to end test.
        LDD    ,X++         ;is right string so get length,
        ORCC   #%00000100   ;moving X to 1st byteand set Z
        BRA    TNLPND       ;so exit from loop occurs.
;test for table end reached, set 'not found' flag if it is.
TNTERM  LDD    −2,X         ;was'name'double-null terminator?
```

```
        BNE     TNNEXT    ;go get next if not, else set C
        ORCC    #%00000001 ;to show string not found and
        BRA     TNLPND    ;go exit loop (Z= 1).
;get length, add to pointer to index next string. Clear exit.
TNNEXT  LDD     ,X++      ;get length, moving pointer past
        LEAX    D,X       ;and add, indexing next string.
        ANDCC   #%11111011 ;clear Z for no exit from loop.
TNLPND  BNE     TNLOOP    ;repeat if not string or table end.
        PULS    PC,Y      ;restore and exit.
```

Optimising TPRINT

Since each text line takes up eight rows on the high resolution screen, the cursor address offset from the screen start is given by the formula:

(line offset * chars per line * 8) + char offset.

However, in the Dragon and Color Computer there are 32 characters to each line – each screen row uses 32 locations in PMODE 4. Since 32 multiplied by 8 is 256, the cursor address offset can be formed simply by picking up the line and character offsets as they are in TXTVAR. This makes the TDISPX module of TPRINT much shorter and quicker.

```
;TDISPX: Dragon / Color Computer version.
TDISPX  TFR     D,X   ;save D in X, pick up line and
        LDD     8,U   ;char offsets as full address
        ADDD    ,U    ;offset, add screen start to give
        EXG     D,X   ;cursor address into X, getting
        RTS           ;D back. Return to TPRINT.
```

Chapter Eight
Six Bits of Sound

System independent text or graphics suites are fairly easy to write since most computers use a similar form of memory mapped display. Sound generation, on the other hand, tends to be very hardware dependent. A lot of computers use programmable sound-effects chips. Some are limited to a simple monotone (sound-on, sound-off) or have no sound facilities whatsoever. The Dragon and TRS-80 Color Computer each have five possible sound production methods, except that one source is 'non-implemented'. The PIA switching of these is shown in Table 8.1. This diversity of sound creation methods makes it practically impossible to write generally applicable code.

Table 8.1. Dragon sound source selection

PIA 1 CRB-3 (sound enable)	*PIA 0 CRB-3 (MUX hi-bit)*	*PIA 0 CRA-3 (MUX lo-bit)*	*Sound source selected*
1	0	0	D/A
1	0	1	Cassette
1	1	0	Cartridge
1	1	1	not implemented
0	0 or 1	0 or 1	single bit sound

The routines in this chapter are written specifically for the Dragon's six-bit digital to analog converter (D/A) since this is by far the most versatile of the sources. They are divided into two suites showing different approaches to sound – static and dynamic. The second of these makes use of the Dragon's two timer interrupts which depend on the frequency of the alternating current mains power supply, which is 50 Hz in Britain. Because of this hardware dependency both sets of routines are unlikely to work on other computer systems and the second set may not work correctly in other countries where the a.c. frequency is different – in the United States of America, the frequency is 60Hz. This said, you will probably find that not too much rewriting of the routines will be needed to use them on other computers with a D/A and a high-frequency (5000 or more each second) interrupt.

The Dragon D/A

Bits 2 to 7 of PIA 1 PRA (at location $FF20) are output lines to the D/A. If a set bit is written to any of these PRA bits then the line goes high at about +5 volts. If a PRA bit is reset (0) then the line is low at 0 volts. At the other end of each line, in the D/A, the voltage is reduced in proportion to the binary place value of the line. This is achieved by parallel resistors which give double the resistance to each successively lower value bit. Once past the resistors, the lines join to produce one analog voltage which varies in proportion to the value of the six-bit digital number written to PRA-2 to 7.

A *digital value* is one which is incremented or decremented in discrete steps whereas an *analog value* can, theoretically, show an infinite number of gradations. The D/A output, however, depends on the digital write to the PIA and so can only have 64 different

Table 8.2. Dragon D/A output voltage.

PIA 1	*D/A bit*	*Digital value reset*	*Digital value set*	*Approximate voltage out low*	*Approximate voltage out high*
PRA-7	5	0	$20	0	2.288
PRA-6	4	0	$10	0	1.144
PRA-5	3	0	$08	0	0.572
PRA-4	2	0	$04	0	0.286
PRA-3	1	0	$02	0	0.143
PRA-2	0	0	$01	0	0.0715
				Sub-total:	
				Add constant:	0.25
				Total D/A output:	

voltage levels but this number is sufficient to obtain reasonably smooth sound changes. The output voltages don't range from 0 to +5 volts as you might expect but are limited to a safe middle range of about 0.25 to 4.75 volts. Table 8.2 shows how to calculate the output voltage for any digital value 0 to 63.

Since bits 2 to 7 of the PRA are used for the D/A instead of bits 0 to 5, it is sometimes worthwhile to think of the values as ranging from 0 to 252 ($00 to $FC) in increments of 4. This alternative approach affects the way in which the six bits to be used as a D/A value are selected from the eight bits in a byte before they are written to the D/A:

```
LOWAY  ASLA                 ;shift bits 0 to 5 up into
       ASLA                 ;2 to 7, clearing 0 and 1.
       STA    $FF20         ;write 6-bit to D/A.

HIWAY  ANDA   #%11111100    ;clear unused bits 0 and 1
       STA    $FF20         ;write 6-bit to D/A.
```

In a loop which increments or decrements the value output by 1, the 'low way' value is changed every iteration but the 'high way' value actually written to the D/A will be affected only every fourth iteration.

Creating waves

Rapid changes in air pressure cause our eardrums to vibrate and, after a process of bony amplification and neuron triggering, we hear sound. We discern the speed of the pressure changes as *pitch* and the difference between low and high pressures as *amplitude* or *volume*. These two dimensions to sound are represented graphically in Fig. 8.1 which shows one complete cycle of a sine wave. If the cycle of air pressure change this represents is repeated rapidly over a period of time we hear a constant note. Stretching the wave vertically makes the sound louder and stretching it horizontally results in a lower pitch – each second of the note contains fewer cycles.

We can use the D/A to create sound waves by applying different voltages to a loudspeaker, usually by way of an amplifier. The diaphragm of the speaker is controlled electromagnetically so that higher voltages cause a greater displacement from its rest position. Alternating between high and low voltages causes oscillation of the diaphragm which produces air pressure waves. The difference between the maximum and minimum voltages in a cycle translates

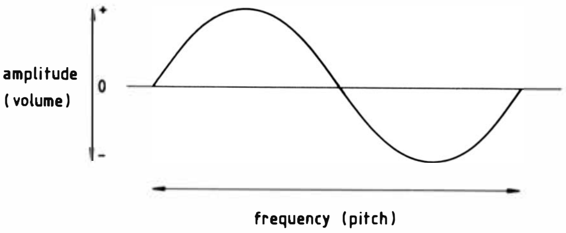


Fig. 8.1. Sine wave.

into pressure difference so we can control volume by voltage variation. Pitch is controlled by the time we take to complete each cycle of high to low voltage output.

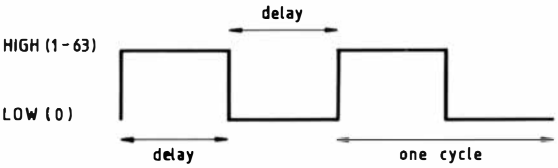


Fig. 8.2. Square wave.

Simple alternation between low and high voltage creates a 'square wave' with the volume dependent on the high voltage value and the pitch on the delay between changing state. This is shown in Fig. 8.2. Square waves are actually very good sounds since they are rich in harmonics at odd multiples of the fundamental frequency – for example, a 100 Hz square wave has the harmonic frequencies of 300, 500, 700 and so on. Square waves are not the only interesting wave shapes, however, and machine code programs operate fast enough to output several dozen different voltages in each cycle, not just the two necessary for square waves.

Relating the loudspeaker's diaphragm movement to Fig. 8.1, the curve can represent its displacement over a period of time. A sequence of values, taken from a wave-shape table, can be sent out through the D/A to position the diaphragm correctly at equal intervals of time along the cycle. The result is stepped rather than a smoothly curved wave form, but it is still quite a good approximation. Sound waves can be any shape and each has its own distinctive qualities. Table 8.3 is an example of some fundamental wave shapes coded in the highest 6 bits of each of sixteen bytes. The table is for the routine SOUND to use and $00 is a terminator, so any zero values within each shape are coded as $01 which will be masked to produce $00 by SOUND.

Table 8.3. Digital wave shapes (WAVTAB).

No.	*Values (hexadecimal)*															
0	FF	01	FF	01	FF	01	FF	01	FF	01	FF	01	FF	01	FF	00
1	F0	E0	D0	C0	B0	A0	90	80	70	60	50	40	30	20	10	00
2	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0	00
3	F0	10	E0	20	D0	30	C0	40	B0	50	A0	60	90	70	80	00
4	80	70	90	60	A0	50	B0	40	C0	30	D0	20	E0	10	F0	00
5	80	80	80	C0	FF	C0	80	40	01	40	80	C0	FF	C0	80	00
6	FF	FF	FF	FF	01	FF	FF	FF	FF	01	FF	FF	FF	FF	01	00
7	01	04	08	10	20	40	80	FF	FC	F8	F0	E0	C0	80	01	00

SOUND routine

The routine SOUND needs four parameters input in D and X. A must give the wave shape number and B the number of times the shape is to be repeated. The low order byte of X must have the frequency delay – a measure of the length of time between writing successive table values to the D/A. The high order byte of X is the volume control byte. Volume is graded from $\frac{1}{256}$ to $\frac{256}{256}$ of full volume. The digital value from the table is multiplied by this byte to produce a 16-bit value of which only the highest six bits are used. If the volume byte is $00 this is taken to mean 256.

An EQU directive sets the length of the table entries as far as SOUND is concerned. From this it calculates the start address of the requested waveshape. You can write a table with longer wave shapes and, if you do, WAVLEN will need to be set accordingly. Another way to get longer wave shapes is to set the null terminators to $01 – SOUND will continue to address successive table bytes until it reaches a $00.

SOUND – Wave shape sound routine
Stack – 8.

I/O – Input A = wave shape no. in WAVTAB. (0 to 255)
B = repeat count for wave shape. (1 to 256)
X-hi = volume (1 to 256)
X-lo = frequency delay (1 to 256)

Notes – Initialisation of the D/A as sound source, and sound enable must have taken place before SOUND. Both frequency-delay and repeat-count affect the length of the note played.

```
WAVLEN  EQU    $10            ;WAVTAB shapes byte length.
;initialisation by addressing 1st byte of correct entry in
;WAVTAB in X and getting repeat count in B.
SOUND   PSHS   X,D,CC         ;save regs used. Index base of
        LEAX   WAVTAB,PCR     ;wave table then calculate offset
        LDB    WAVLEN         ;to requested shape entry (A)
        MUL                   ;from table start and add to
        LEAX   D,X            ;pointer. X now at right shape.
        LDB    2,S            ;get repeat count back in B.
;shape repeat loop: end when B=0. Main action to save shape
;start address while processing shape for quick repeat.
SLOOP   PSHS   X,B            ;save shape start and repeat count.
        LDA    ,X+            ;get 1st shape val., index next,
        BEQ    SLPEND         ;but end shape if terminator.
;shape process loop: end on $00 table byte. Multiply table
;value by vol. and write to D/A.
SVALLP  LDB    6,S            ;get volume and if $00 then skip
        BEQ    SDTOA          ;as D already val. * vol. (256)
        MUL                   ;else D = val. * vol./256.
SDTOA   ANDA   #%11111100     ;mask out unused bits and write
        STA    $FF20          ;new value to D/A then delay ...
;frequency delay loop: determines pitch.
        LDB    7,S            ;get frequency delay from stack
SFREQ   DECB                  ;and loop until B=0 just using
        BNE    SFREQ          ;up time to get right pitch.
        LDA    ,X+            ;get next shape value and
        BNE    SVALLP         ;repeat till null terminator.
SLPEND  PULS   X,B            ;get shape start address and
        DECB                  ;repeat count back. Repeat till
        BNE    SLOOP          ;repeat count done.
        PULS   PC,X,D,CC      ;restore and exit SOUND.
```

Sound strings

Each set of sound parameters can, if given a name and placed in a table, form part of a string much the same as the text strings of Chapter 7. The routine SNSTR deals with sound strings rather like TNSTR and TSTRNG together deal with text.

The string table format is somewhat different for the sets of sound parameters since each element in the string has four bytes and not just one as in a text string. The two-byte name comes first, followed by two bytes giving the number of elements – this value has to be multiplied by four to index the next string in the table.

SNSTR also deals with initialisation of six-bit sound on the Dragon – and remembers to switch off the sound enable bit when the string has been played.

SNSTR – Play named sound string

Modules –	SGSTR, SDSTR.
Subroutines –	SWITCH, SOUND.
Stack –	12 + subroutines.
I/O –	Input D contains the name of the sound string. Output C=1 if sound played, C=0 for string not found or null string.
Notes –	The name in D cannot be $0000 as this is the end-of-table flag.

```
;SNSTR: top level, initialises pointer to string table, calls
;modules to address named string and, if found, play it.
SNSTR    PSHS   U,Y          ;save regs used. Point U to start
         LEAU   SSTAB,PCR    ;of sound string table.
         BSR    SGSTR        ;go find named string, but
         BCC    SNEND        ;end if not found, else
         BSR    SDSTR        ;go process it if found.
SNEND    PULS   PC,U,Y       ;restore, exit SNSTR routine.
;
;SGSTR: Input D = string name, U = SSTAB start address.
;out: C=0: U = SSTAB+3. C=1: Y = no. of elements, U is
;pointer to 1st byte.
SGSTR    PSHS   D            ;name to stack for comparison.
;get name and no. of elements. End if end-of-table.
SGLOOP   PULU   Y,D          ;get string header info.
         CMPD   #0           ;test for e-o-t flag and exit
         BEQ    SGLEND       ;with C = 0 if end reached.
;exit loop, string found, if name matches.
         CMPD   ,S           ;if string name = request name
         ORCC   #%00000001   ;then set string found flag C
         BEQ    SGLEND       ;and exit loop.
;move pointer to next string in table.
```

```
        TFR     Y,D             ;move no. of elements into D
        LEAU    D,U             ;and add four times to pointer
        LEAU    D,U             ;(4 bytes to each element)
        LEAU    D,U             ;so pointer now addresses the
        LEAU    D,U             ;name of next string.
SGLEND  BNE     SGLOOP          ;repeat till name matches or e-o-t.
        PULS    PC,D            ;restore name, ret. to SNSTR.
;
;SDSTR: Input U points to 1st byte, Y = no. of elements.
;out: C=0: null string (Y=0). C=1: string played, U at
;string + 1, Y = 0.
;first, test for empty string. If okay, switch sound on.
SDSTR   PSHS    X,D             ;save regs used.
        CMPY    #0              ;test for an empty string and
        BEQ     SDEND           ;end immediately if it is.
        LDA     #$08            ;else enable sound and select
        JSR     SWITCH,PCR      ;D/A at Dragon PIAs.
;process loop. Get parameters, call SOUND, till e-o-string.
SDLOOP  PULU    X,D             ;get SOUND parameters from string,
        LBSR    SOUND           ;indexing next set, and SOUND
        LEAY    -1,Y            ;them. Repeat for all elements
        BNE     SDLOOP          ;in string.
;disable sound and set string-played flag.
        LDA     #0              ;reset bit 3 for switching off
        JSR     SWITCH,PCR      ;sound enable.
        ORCC    #%00000001      ;flag string played in C.
SDEND   PULS    PC,X,D          ;restore, return to SNSTR.
```

Sound sample

SXMPL can be called from Dragon BASIC by an EXEC command. It sends the names of both strings in the example sound string table SSTAB to SNSTR. String $DEAF plays all eight wave shapes in the sample WAVTAB and string $EEEC plays the second shape at eight different pitches.

```
SXMPL   PSHS    D,CC            ;save regs used.
        LDD     #$DEAF          ;go play string "DEAF"
        JSR     SNSTR,PCR       ;
        LDD     #$EEEC          ;then string "EEEC"
        JSR     SNSTR,PCR       ;
        PULS    PC,D,CC         ;restore, return to Basic.

SSTAB   FDB     $DEAF           ;name
```

```
        FDB     $0008           ;8 elements (parameter sets)
        FCB     0,0,0,64        ;shape, rep-cnt, vol, fr-del.
        FCB     1,0,0,64        ;
        FCB     2,0,0,64        ;(try this string with the
        FCB     3,0,0,64        ;repeat-count, volume and
        FCB     4,0,0,64        ;frequency-delay parameters
        FCB     5,0,0,64        ;set to different values.
        FCB     6,0,0,64        ;Also change shape nulls to $01
        FCB     7,0,0,64        ;to get complex wave shapes.)
        FDB     $EEEC           ;name
        FDB     $0008           ;8 string elements.
        FCB     1,$20,$80,0     ;
        FCB     1,$24,$80,$E0   ;(see if you can find the
        FCB     1,$28,$80,$C0   ;frequency-delays which
        FCB     1,$33,$80,$A0   ;approximate to actual
        FCB     1,$40,$80,$80   ;notes in an octave.)
        FCB     1,$55,$80,$60   ;
        FCB     1,$80,$80,$40   ;
        FCB     1,$00,$80,$20   ;
        FDB     $0000           ;SSTAB terminator.
```

Dynamic HI-FI sound

The sound produced by the routine SOUND can be considered static since neither the frequency nor volume change throughout the duration of the note. This is useful for playing tunes but not much good for creating games sound effects. The best effects are produced when frequency and/or volume is dynamic. HIFI lets you set start values and increment or decrement values so both pitch and volume alter by a programmed amount every $\frac{1}{50}$ second.

Most Dragon owners know that the TIMER function is operated by an interrupt every $\frac{1}{50}$ second but many do not realise that this interrupt is synchronised to the video display logic which renews the TV picture fifty times a second. This is known as the 'Frame sync Interrupt' (FI). The display consists of 256 horizontal lines and the Dragon has yet another interrupt synchronised to the line timing – the 'Horizontal sync Interrupt' (HI). When enabled by setting bit 0 of PIA 0 CRA the HI causes an IRQ interrupt 12800 times a second (256 * 50), one every 69 or 70 clock cycles.

The part of HIFI which writes to the D/A is interrupt driven which means that it is not called as a subroutine by the main iterative part, HIMAP. Instead, every other HI signal causes the CPU to stop processing HIMAP, save all the registers on stack (excluding the stack pointer S) and process HIDIVE. The execution time of HIDIVE is longer than 70 clock cycles so a second interrupt is ignored, further interrupts being automatically disabled during the processing of one interrupt. HIDIVE decrements a frequency counter by 1 at each interrupt. Every time the counter reaches zero the D/A output is changed from low to high or from high to low. So HIFI outputs a square wave with a maximum frequency of 3200 Hz and a minimum of 12.5 Hz.

The FI signal causes the note length to be decremented by 1 and the note ends when this reaches 0. HIFI notes can be accurately timed from 0.02 to 5.12 seconds. FI also results in the volume and frequency values being adjusted by the input increment/decrement values. HIFI parameters can best be understood by reference to Table 8.4.

Table 8.4. HIFI sample string (HISS) hexdump. 160 bytes.

```
Offset    Parameter values

0000:     A3 20 00 F8 00 01 A3 40 80 01 80 FE A3 10 80 00
0010:     01 00 A2 08 00 00 A2 08 00 00 A2 08 80 00 A3 20
0020:     C0 00 01 00 A3 30 FF 00 40 00 A3 00 00 FF 00 10
0030:     A2 00 00 20 A3 00 00 10 10 00 A1 00 00 F0 A1 00
0040:     00 20 A1 00 00 E0 A3 00 00 F0 00 F0 A3 10 00 F0
0050:     00 00 A2 10 F0 00 A2 10 E0 00 A2 10 D0 00 A2 10
0060:     C0 00 A2 10 B0 00 A2 10 A0 00 A2 10 90 00 A2 10
0070:     80 00 A2 10 70 00 A2 10 60 00 A2 10 50 00 A2 10
0080:     40 00 A2 10 30 00 A2 10 20 00 A2 10 10 00 A2 10
0090:     08 00 A2 10 04 00 A2 10 02 00 A2 10 01 00 00 00
```

The first two bytes of each sound command are essential. If they are both nulls HIMAP ends. The second byte is the note length in $\frac{1}{50}$ second. The first byte tells HIMAP what other parameters are to be picked up. If bit 0 is set then volume and volume inc/dec are in the next two bytes. If bit 1 is set then the frequency delay and its inc/dec value follows. If both are set then the volume parameters are followed by the frequency parameters. If neither is set then no further bytes are picked up. Since only bits 0 and 1 of the command code are used, the other bits can take any value. In HISS they are used to identify the command bytes as A*x* (except for the byte at offset 0068 which is a frequency).

Look at offset 0030 to 003D in Table 8.4:

```
0030:   A2 00  = frequency follows. Note length 256.
0032:   00 20  = fr-delay 256, inc'd by 32 every 1/50 sec.
0034:   A3 00  = vol. & freq. follow. Length 256.
0036:   00 10  = volume 0, incremented by 16 every 1/50 s.
0038:   10 00  = fr-delay 16, constant (increment 0).
003A:   A1 00  = volume follows. Length 256/50 sec.
003C:   00 F0  = vol. 0. decremented by 16 every 1/50 sec.
```

HIJAN (just a noise) is a short program to send the address of HISS to HIFI. Call it from BASIC with an EXEC command and listen to 46.56 seconds of dynamic sounds.

```
HIJAN     PSHS    U              ;save the contents of U so it
          LEAU    HISS,PCR       ;can be used to send HISS
          JSR     HIFI,PCR       ;to HIFI for playing, then
          PULS    PC,U           ;restore, return to Basic.
```

HIFI – Interrupt timed dynamic sound routine

Modules –	HICUE, HIMAP, HICUT *and interrupt routine* HIDIVE.
Stack –	25 including IRQ entire register save.
I/O –	Input addresses 1st byte of a sound string. On exit, U = string + 1.
Notes –	Written to work on Dragon/TRS-80 Color Computer.

```
;HIFI: top level, just calls modules.
HIFI      PSHS    Y,X,D,CC       ;save registers used. Switch on
          BSR     HICUE          ;D/A sound and Horiz. Interrupt.
          BSR     HIMAP          ;go to parameter fetch loop.
          BSR     HICUT          ;switch off D/A and HI.
          PULS    PC,Y,X,D,CC    ;restore and exit HIFI
;
;HICUE: initialise high frequency interrupt and D/A.
HICUE     ANDCC   #%11101110     ;clear C, enable IRQ interrupts.
          LDA     #$08           ;enable sound and select
          JSR     SWITCH,PCR     ;D/A 6-bit sound.
          LEAX    HIDIVE,PCR     ;change IRQ jump address from
          STX     $010D          ;TIMER to HIDIVE.
          LDX     #$0003         ;ensure no sound output
          STX     HIWORD,PCR     ;until ready inside HIMAP.
          LDA     $FF01          ;enable Horizontal sync Interrupt
          ORA     #%00000001     ;by setting PIA 0 CRA-0
          STA     $FF01          ;not changing other bits.
          RTS                    ;return to HIFI.
```

```
;
;HIMAP: parameter fetch and inc/dec loop. Dependent on
;FI occurred flag C=1 or infinite loop at HIWAIT!
HIMAP   LDD    ,U++          ;get code & length, bump point,
        BEQ    EHIMAP        ;end if $0000 terminator.
;test bits 0 & 1 of command code:
;00 = note length only (already in B).
;01 = get volume byte and volume inc/dec byte in X.
;10 = get freq-delay byte and its inc/dec byte in Y.
;11 = get vol, vol-inc/dec, fr-del, fr-del-inc/dec in X & Y.
HILOOP  BITA   #1            ;test if volume follows and
        BEQ    HIFREQ        ;skip if not, else pick up
        PULU   X             ;vol, vol i/d in X, bump point.
HIFREQ  BITA   #2            ;test if frequency-delay follows
        BEQ    HIWAIT        ;skip if not, else pick up
        PULU   Y             ;fr-d, fr-d i/d in Y, bump point.
;infinite loop if C=0 on entry to HIWAIT. 1/50 second
;interrupt sets C to 1, so wait for interrupt to occur.
HIWAIT  BCC    HIWAIT        ;loop till C = 1, FI occurred.
;every 1/50 sec. adjust volume and frequency-delay by
;adding respective inc/dec bytes. Decrement note length
;if 0 get new parameters, else wait for next FI.
        PSHS   Y,X           ;put volume and frequency
        LDA    2,S           ;variables on stack for access
        ADDA   3,S           ;by accumulator for
        STA    2,S           ;parameter adjustment
        LDA    ,S            ;which produces dynamic
        ADDA   1,S           ;sound - continual change
        STA    ,S            ;of volume and/or frequency.
        DECB                 ;if note not finished
        BNE    HIWAIT        ;then wait for next 1/50 interrupt.
        LDD    ,U++          ;else get next command & length
        BNE    HILOOP        ;repeat till $0000 terminator.
EHIMAP  RTS                  ;return to HIFI
;
;HICUT: switch off high frequency interrupt and D/A sound.
HICUT   LDA    $FF01         ;disable Horiz. sync Interrupt
        ANDA   #%11111110    ;by clearing PIA 0 CRA-0
        STA    $FF01         ;not changing other bits.
        LDX    #$9D3D        ;change IRQ jump address from
        STX    $010D         ;HIDIVE to TIMER.
        CLRA                 ;switch off sound enable bit
        JSR    SWITCH,PCR    ;by clearing all C2 lines.
        RTS                  ;return to HIFI.
;
```

```
;HIDIVE: Interrupt routine, entered on both HI and FI
;interrupts. IRQ automatically saves all registers to
;stack so HIDIVE works on the stacked parameters and on
;2 variable bytes, HIWORD: hi-byte is a square wave D/A
;mask, lo-byte is the frequency count loaded from saved B.
HIDIVE  LDD   HIWORD,PCR  ;get D/A mask and fr-del.
        DECB              ;count down fr-delay and if
        BEQ   HIWAVE      ;0 then go change voltage out.
        NOP               ;else use up time so
        NOP               ;HIDIVE always takes same
        BRA   HIDAVE      ;no. of clock cycles.
HIWAVE  LDB   6,S         ;renew fr-delay count.
        EORA  #%11111100  ;low to high or high to low
HIDAVE  STD   HIWORD,PCR  ;restore changed variables.
        ANDA  4,S         ;get masked volume (or $00)
        STA   $FF20       ;and write it to D/A.
;set C if FI interrupt also occurred. Else C=0.
        LDA   $FF03       ;get PIA 0 CRB for FI flag in
        LSR   ,S          ;bit 7. Shift out stacked C
        ROLA              ;shift out FI flag and shift
        ROL   ,S          ;it in to stacked C flag
;read PRA and PRB to clear interrupt flags in CRA and CRB.
        LDA   $FF02       ;clear FI flag.
        LDA   $FF00       ;clear HI flag.
        RTI               ;return to interrupted HIMAP.
HIWORD  RMB   2           ;HIDIVE variables.
```

Chapter Nine

An Interrupt Driven Clock

Any computer system which can generate a regular interrupt at a frequency of 1 to 256 per second can have this on-screen clock. It has to be 'patched in' to the normal interrupt service routine. On the Dragon this means changing the address in the JMP instructions at locations $010C, $010D and $010E to the address of CLOCK which must end with a jump to the original destination of the interrupt – to $9D3D – if you still want the TIMER and PLAY functions. Don't expect mellifluous music with the clock in operation – all the tones have a very pronounced 50 Hz warble with a once-a-second hiccup. Set the time correctly by POKEing from BASIC.

CLOCK – Interrupt driven, on-screen, 24-hour, digital clock

Modules –	CDPRNT.
Stack –	Normal IRQ stacking + 2.
I/O –	None in. Registers U, X, D and CC are changed. Time written to memory-mapped display every second.
Notes –	If the normal interrupt routine ever uses the contents of the passed down registers then CLOCK should be written to PSHS and PULS U,X,D,CC.

```
IRQHZ   EQU   $32           ;interrupt frequency (50 for Dr.)
CLOCK   LEAU  CCOUNT,PCR    ;index frequency count down and
        DEC   ,U            ;dec it, exit CLOCK routine
        BNE   CLKEND        ;if second not up, else
        LDB   IRQHZ         ;renew counter for next
        STB   ,U            ;second count down.
;increment seconds with any carry to minutes, hours. BCD
;(Binary Coded Decimal) values used for speed and ease.
        CLRB                ;clear carry flag C so secs get
        LDB   #3            ;inc'd. Count for 3 values.
CVLOOP  LDA   ,-U           ;get time byte after moving
        BCS   CVLEND        ;pointer. Skip if no inc to do.
        ADDA  #1            ;inc time byte and correct to
        DAA                     ;BCD value then
        STA     ,U              ;put it back to string.
        CMPA    -3,U            ;compare with limit value and
        BLO     CVLEND          ;skip if not reached, else reset
        CLR     ,U              ;to 0. C clear for next byte inc.
CVLEND  DECB                    ;loop for seconds, minutes, and
        BNE     CVLOOP          ;hours, leaving U at hours.
;index screen RAM and print time to screen after converting
;BCD to ASCII decimal digits.
        LDX     -5,U            ;get screen address. Set count
        LDB     #2              ;in B for 3 loops, ending $FF.
CPLOOP  BSR     CDPRNT          ;print hrs, mins or secs
        TSTB                    ;if seconds just been printed
        BEQ     CPLEND          ;then skip, else
        LDA     #$3A            ;get colon ':' separator to
        STA     ,X+             ;screen, bumping pointer.
CPLEND  DECB                    ;repeat for hours, minutes
        BPL     CPLOOP          ;and seconds.
CLKEND  JMP     IRQRST          ;go to normal IRQ routine
;
;CDPRNT: module to get BCD value at U, convert to two
;ASCII decimal digits and put to screen at X.
CDPRNT  LDA     ,U              ;get BCD value,
        LSRA                    ;shift high order digit
        LSRA                    ;down into lo-nibble A,
        LSRA                    ;clearing hi-nibble A
        LSRA                    ;at same time, then add in
        ORA     #$30            ;ASCII digits hi-nibble and
        STA     ,X+             ;write to screen, bump pointer.
        LDA     ,U+             ;get lo-digit, bumping pointer,
        ANDA    #$0F            ;clear hi-nibble and add in
        ORA     #$30            ;ASCII digits hi-nibble, write
        STA     ,X+             ;to screen, bumping pointer,
        RTS                     ;and return to CLOCK.
;
;variables and parameters for CLOCK, with only label at
;interrupt frequency needed.
        FDB     $0400           ;screen RAM address ($0400 is
;at top left of Dragon's normal text screen).
        FCB     $24,$60,$60     ;hrs in day, mins in hr, secs in
;min. BCD values have to be written in looking like hex.
;These are the limits for comparison with variables ...
        FCB     0,0,0,          ;time variables (at midnight).
CCOUNT  FCB     IRQHZ           ;second counter.
```

Appendix A
6809 Architecture

Architecture usually refers to the make-up of the actual microprocessor, being anything from a simple list of the registers to a detailed mapping of the full logic. However, since the processor in isolation is about as much use as 1.5 kg of brain on a butcher's slab, it is more illuminating to describe it in relation to the computer system as a whole.

Figure A.1 shows a very much simplified block diagram of the relationship between the various components of a complete system. The number of devices and their linkage is far more complex – as a glance at the schematics for a real system will show you – but in general programmers are not too concerned with technical detail.

Peripherals stand outside the basic system and the computer can, in theory, work without any of them – though not to any useful effect. *Input/Output devices* are the sockets into which peripherals are plugged. At their simplest they are mere *ports* through which data is transferred. In their most complex form they are 'intelligent' or 'semi-intelligent' configurations which can perform much of the decoding of information to and from peripherals or linked systems.

Memory is an essential part of the system and is basically a set of numbered pigeon-holes for storing numbers. Memory is of two types: (a) ROM or Read Only Memory where the contents are fixed ('burnt-in' is the jargon term) and cannot be overwritten by new numbers, and is used for programs which have to be present in the computer on power-up; (b) RAM or Random Access Memory which can have its contents changed. Many computer systems have a *memory-mapped video* display where a portion of RAM is dedicated as *video-RAM* or *screen-memory*. These dedicated memory locations correspond to screen positions and their contents to dot-patterns appearing as characters or graphics on the screen. Some systems treat video display in the same way as printers requiring character codes to be sent through an I/O device.

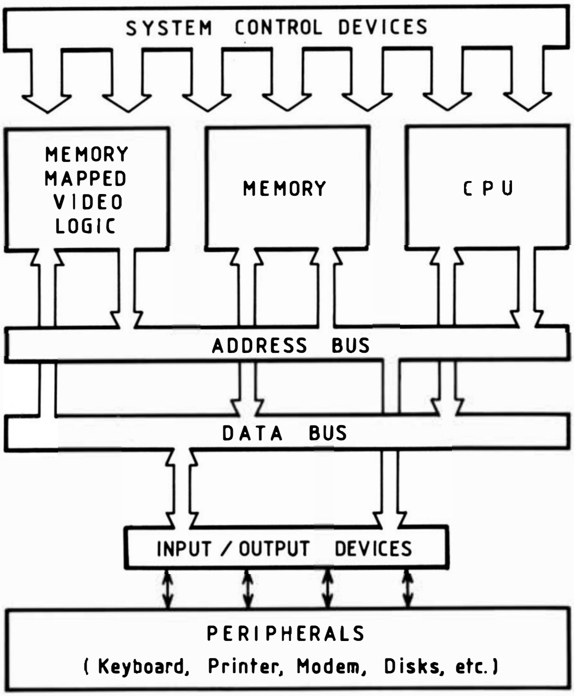


Fig. A.1. The main features of a computer system.

Address and *data buses* are networks of lines running through the system to carry information between the different parts. In an 8-bit computer the address bus is 16 lines wide and can hold any value between $0000 and $FFFF (0 to 65535 in decimal) and the data bus is 8 lines wide and can carry any value from $00 to $FF (0 to 255 decimal).

The *CPU* or central processing unit contains a small number of special memory locations referred to as *registers*, and *internal logic* which can perform a few simple operations on binary numbers. The CPU can also put values on to the address bus, put values on to the data bus, take values from the data bus and move values between the registers.

System control devices perform a variety of tasks mostly ensuring that the actions of all the other parts are synchronised. An example of the importance of this synchronisation is demonstrated by the need for both the CPU and video logic to access screen-memory at virtually the same time when a program is printing a message. The conflict is resolved by use of a *system clock* which regularly and frequently alternates between two states: high (or *active*) and low. The CPU is only allowed to access memory during one state and video logic during the alternate state. One high and one low state together constitute a *clock cycle* and the frequency of these cycles is used as the measure of how fast a computer operates. Since machine code instructions always operate in a given number of cycles the actual time taken by any instruction varies with the cycle rate. The TRS-80 Color Computer runs at 0.89 MHz (890000 clock cycles per second). More sophisticated, and expensive, business systems such as the SEED System 19 usually run at 2 MHz. The Positron 900 is advertised as having a *500 ns* cycle time (500 nanosecond cycles is the same as 2 MHz).

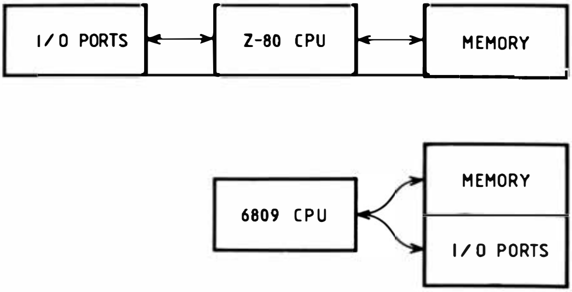


Fig. A.2. Z80 and 6809 memory and I/O ports addressing.

Computer systems are designed around the abilities and limitations of the processor (CPU) employed. The I/O ports of Z80 systems are tied directly to the processor and the Z80 has special instructions dealing specifically with I/O. The 6809 has no such specific I/O capabilities but can only address memory. Consequently I/O devices in 6809 systems have to be tied in to memory addresses (see Fig. A.2) and all I/O operations involve normal memory reference. Memory or port addressing is accomplished by putting a 16-bit address on the address bus. Data stored to memory or output has to be put on the data bus. Memory read or input takes data from the data bus.

All the timing of data movement, address decoding and most of the system control is taken out of the hands of the programmer who only has to provide the processor with a sequence of instructions in numerical form – machine code. The code will say what action has to be performed on data and *where that data is to be found*. This is the point where the CPU registers interact with the system.

The 6809 register set

Accumulators (A, B, D)

Accumulators are the registers in which the results of most

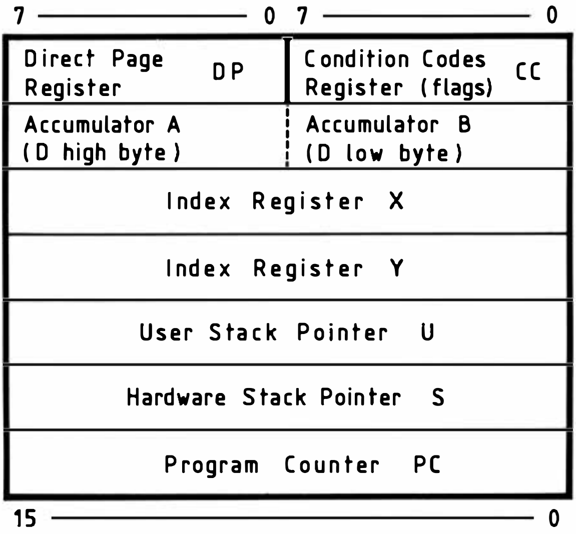


Fig. A.3. The 6809 register set.

operations are stored – especially arithmetic operations. A and B are both 8 bits long and capable of holding values 0 to 255 ($00 to $FF). D is a 16-bit accumulator formed by joining A and B. The 6809 has a few operations which act on 16 bits of data (mostly load and store operations) but as the data bus is only 8-lines, 16-bit data operations involve two accesses.

Pointers (X, Y, U, S)

Pointers are 16-bit registers which normally contain addresses. The Indexed/Indirect addressing modes of the 6809 cause the values held in the pointers to be put on the address bus, sometimes after the addition of a constant value (programmed in the instruction) or the value from an accumulator. As an example, the instruction LDA B,X causes the 16-bit address formed by adding the value of B to that of X to be put on the address bus. The system logic decodes this address to access just one memory location which results in the data held there being put on the data bus. The CPU receives the data from the data bus and stores it in the A register. All of this action is transparent to the programmer who thinks of the operation as *A←memory at X + B*. The pointers can also be used for 16-bit data store and load and limited arithmetic – designed for address manipulation.

Index Registers (X, Y)

X and Y are usually referred to as Index registers since that is their main function, as described above. The instructions LEAX n, X and LEAY n,Y also allow them to be used as 16-bit counters.

Hardware Stack (S)

As well as the normal pointer functions, S has a special function as stack pointer. Stack is an area of memory reserved for temporary storage of register values generally and for storage of program addresses during subroutine calls particularly. On a JSR or BSR instruction, the following actions occur:

> memory at S-1←Program Address low-order byte
> memory at S-2←Program Address high-order byte
> S←S−2
> Execution moves to Subroutine Address

At the end of the subroutine the program address is taken from memory at S:S+1, S has 2 added to it and the program continues. The instructions *PSHS register names* will cause a similar action but with the values of the registers named. Stack can be anywhere in memory since S is a 16-bit register and can therefore address any memory location.

User Stack (U)
U can be used in the same way as S for saving register values on a User Stack. Program addresses, however, are always saved to the Hardware Stack by subroutine call instructions.

Direct Page Register (DP)
DP is used by the processor as the high-order byte of memory addressed by the Direct Page addressing mode. The low-order byte of the address has to be written into the machine code instruction. The 6809 can address 65536 different memory locations. In hexadecimal the addresses run from $0000 to $FFFF and the leftmost two digits form the 'page number'. Pages thus run from page 0 to page 255 and each page contains 256 different locations. DP can be set to any page number.

Condition Codes Register (CC)
This is dealt with at length under *6809 Flags*. It is a collection of eight individual bits holding status and control information.

Program Counter (PC)
The Program Counter is the processor's own pointer register. It is the 16-bit register which holds the program address referred to in *Hardware Stack*. Machine code programs are nothing more than a sequence of 8-bit numbers – one number (byte) to one memory location – and the processor reads programs in the following way:

(a) the contents of the *PC* are put on the address bus
(b) the 8-bit value (instruction byte) is taken off the data bus
(c) the *PC* is incremented by 1
(d) the instruction byte is decoded to effect the correct action.

As machine code instructions are anything from 1 to 5 bytes in length the read-instruction sequence may be performed up to five times – each time the PC is pointed to the location of the next byte. When the processor performs the action of any instruction, the PC always contains the address of the location *after* the instruction. This fact is of absolute importance when you use instructions which depend on the value of the PC – Branches, PC-offset addressing, exchange and transfer with the PC.

6809 flags

The Condition Codes register (CC) is different from the other 6809 registers in that each of its eight bits is treated as a separate unit.

bit:	7	6	5	4	3	2	1	0
flag:	E	F	H	I	N	Z	V	C

Fig. A.4. The 6809 condition codes register.

Exceptions to this rule are the instructions ORCC, ANDCC, PSH, PUL, EXG and TFR. Five of the bits are used to flag the results of operations and the others as control bits. Table A.1 gives their uses.

Table A.1. 6809 Condition codes description.

Bit	Name	Description
0	C	*Carry*. Used to store a bit carried out of an arithmetic result when 8 (or 16 for double-byte arithmetic) bits are not enough. Also used as a ninth bit in rotate and shift operations.
1	V	*Overflow*. Shows if 2's complement arithmetic overflow has occurred. The overflow flag is the exclusive-OR of the carry-in with the carry-out of the result sign bit.
2	Z	*Zero*. Set if the result of an operation is zero (all bits reset).
3	N	*Negative*. Sometimes referred to as the *Sign* flag since it is a copy of the result sign bit. In 2's complement signed numbers, \$00 to \$7F is positive (0 to 127 decimal) and \$80 to \$FF negative (−128 to −1 decimal) and bit 7 is thus the sign bit. In 16-bit values the sign bit is bit 15.
4	I	*IRQ Interrupt mask*. Regular IRQ interrupts are disabled when *I* = 1, enabled (allowed to occur) when *I* = 0.
5	H	*Half carry*. Shows if any carry out of the low-order digit of an 8-bit addition occurred. Used

		by the processor during a DAA instruction.
6	F	*FIRQ Interrupt mask*. As *I* but for fast interrupts.
7	E	*Entire state*. IRQ, CWAI, SWI set $E = 1$ then save all registers on stack before dealing with the interrupt. FIRQ resets $E = 0$ and saves only *PC* and *CC*. *RTI* (Return from Interrupt) pulls *CC* off stack and tests *E* to determine whether all registers of just *PC* have to be restored.

6809 interrupts

As well as being connected to the address and data buses and various power and control lines, the 6809 CPU has four input lines which generate a particular type of response and three program instructions which emulate the same response. The response made by the processor is to save the state of the machine (i.e. register contents) on stack and pass control to one of a number of special service routines. The signal causing this response is known as an *Interrupt Request*.

Each type of interrupt requires its own service routine and the address at which that routine starts is stored in a reserved area of memory from $FFF0 to $FFFF. These addresses are known as *Interrupt Vectors*. Table A.2 gives the locations for the Most Significant Bytes and Least Significant Bytes of the vectors, the type of interrupt and brief descriptions.

Table A.2. 6809 Vectored interrupts.

MSB at	*LSB at*	*Type*	*Description*
FFFE	FFFF	RESET	Operating system start address on power on.
FFFC	FFFD	NMI	*Non-maskable interrupt*. Usually an emergency situation such as power drop. The response in this case would be to switch to a back-up battery.
FFFA	FFFB	SWI	*Software interrupt*. Instruction generated interrupt useful during program development for setting break-points. Also for control

			transfer between system and user programs.
FFF8	FFF9	IRQ	*Interrupt Request.* The IRQ line is usually tied to an I/O device such as the Peripheral Interface Adapter (PIA). The main use is for slow peripherals such as printers to give a *ready* signal to the CPU which meanwhile performs other duties.
FFF6	FFF7	FIRQ	*Fast Interrupt Request.* Higher priority, faster action equivalent of IRQ. Used for fast peripherals requiring quick response. FIRQ can interrupt IRQ unless disabled.
FFF4	FFF5	SWI2	As SWI.
FFF2	FFF3	SWI3	As SWI.
FFF0	FFF1	—	Not used.

Interrupt requests are *input* to the 6809. Corresponding *output* lines are used to signal to external hardware that an interrupt has been recognised and that the CPU is or is not ready to respond.

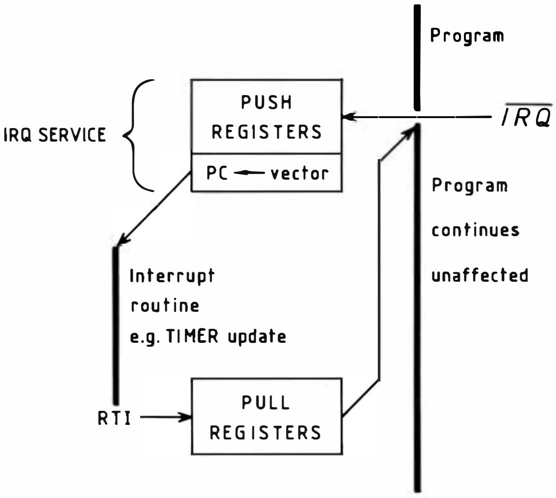


Fig. A.5. IRQ action.

IRQ and FIRQ may be enabled or disabled by the status of the I and F flags in the CC register. I/O devices using IRQ and FIRQ often have control bits which allow for interrupt enable/disable.

Interrupts are transparent to the interrupted program except that stack memory is used for register storage. If your program might be interrupted (e.g. by the timer on the Dragon or TRS-80 Color Computer) then you must ensure that there are 12 bytes of stack space below S for IRQ and 3 bytes below S for FIRQ. Figure A.5 shows the effect of IRQ.

Appendix B

6809 Assemblers

Conventions

Assembler programs (known as *source* programs – the actual machine code is referred to as the *object* program) are always written in a tabular fashion. The columns are called *fields*. Usually the fields are fixed to certain character positions on the line but some assemblers do allow a degree of latitude provided the correct *delimiters* (characters separating different fields) are used. All assemblers require a minimum of three fields with optional others. A printout of object code alongside source program may have as many as seven fields.

(1) *Location* Gives the address of the first byte of the machine code instruction. Usually in hexadecimal.
(2) *Code* Gives the machine code (1 to 5 bytes) in hexadecimal.
(3) *Line number* Optional in the source program but when used refers to the position of the instruction in the program.
(4) *Label* First necessary field in the source program. At assembly the label is used as equivalent to the address of the first byte of the instruction which is labelled.
(5) *Mnemonic* Operation name, e.g. ADDA.
(6) *Operand* or *Address* The data, register or memory reference part of the assembler instruction, e.g. −15,X or #$FE.
(7) *Comment* Optional in the source program. Description of what the program is doing. Necessary if you or anyone else wants to understand the program.

The normal delimiters are: (a) spaces – after a label, after the mnemonic and before a comment which follows an instruction, (b) commas – between operands, before register names in no-offset Indexed modes (e.g. ,X++), and (c) asterisk – before a complete comment line. The delimiters used in this book are standard except that comments are always preceded by a semicolon (;).

Labels are usually restricted to six characters and must start with an alphabetic character (A to Z) or a full stop (.). Some assemblers require all labels to begin with a specific symbol. The DASM assembler for the Dragon, for example, insists that all labels begin with '@'. Register names, mnemonics and assembler directives are not allowed to be used as labels.

Assembler directives

These are instructions to the assembler and are not translated into machine code even though they are written in the mnemonic field. Table B.1 gives the usual 6809 directives with their meaning.

Table B.1. Assembler directives.

Form	*Meaning*
ORG	Origin. Tells the assembler where in memory the object code has to start.
EQU	Equate label to data. The label can then be used in the source program operand field and the assembler will use the equated data in the object code.
RMB	Reserve Memory Bytes. Used to tell the assembler to leave a given number of locations free.
FCB	Form Constant Byte. Put a byte of data into memory at the current program location.
FDB	Form Double Byte. As FCB but two bytes of data.
FCC	Form Constant Character. Store the ASCII codes of the character(s) following FCC.
END	End of source program.

Assembler operand forms

The information which an assembler expects to find in the operand field of a source program corresponds closely to the addressing mode used, especially in the use of register names. Most assemblers, however, expect certain additional information and allow for an expanded range of expressions.

(1) *Labels* Labels can take the place of an actual address in the operand field. The assembler usually passes through a source program at least twice, the first time to build up a label-address table. Instructions requiring data will be given the data located at the label-addressing and instructions needing an address will be given the address. Program relative instructions will have the offset (label *minus* current position) calculated.
(2) *Data* The default case is decimal requiring the number only (optional preceding '&' for program clarity). Other forms are: (a) hexadecimal starting with "$", (b) octal starting with "@", (c) binary starting with "%", and (d) ASCII – single character or character string (depending on assembler sophistication) preceded by an apostrophe, e.g. FCC 'STRING. Some assemblers will also allow arithmetic expressions which reduce to 8-bit or 16-bit integers (fractions lost).
(3) *PC Relative* Since the normal Indexed form n,PC used with a label – say, DATTAB – would produce object code in which the value of DATTAB is used as the offset, a special assembler form n,PCR is allowed. In this case the distance from the instruction to DATTAB is calculated by the assembler and used as the offset.
(4) *Mode symbols* Immediate data requires a preceding hash sign '#'. If the hash is absent the assembler will interpret the data as an address. Assemblers automatically select Direct mode if the address given falls within the page indexed by the DP register and Extended mode if it does not. Direct mode can be forced by preceding the address with '<' and Extended mode by a preceding '>'. In Indexed offset addressing, the assembler automatically chooses the smallest offset form (none, 5-bit, 8-bit or 16-bit) consistent with the displacement. Preceding the operand with a '<' forces the 8-bit form and with a '>' the 16-bit form.

Some real assemblers

MACE – Editor, Assembler, Monitor for Dragon.
DASM – Assembler for Dragon/TRS-80 Color Computer. Allows assembler instructions to be embedded in BASIC programs. Associated monitor DEMON available together with DASM or on a separate cartridge.

DREAM – Editor, Assembler, Monitor from Dragon Data on cassette or cartridge. ALLDREAM version is a full development package with breakpoints, trace, disassembler, etc.
EDTASM – Editor, Assembler, Monitor package on cartridge for the TRS-80 Color Computer. Monitor-debugger part is ZBUG allowing breakpoint setting, etc.
RALLI – OmegaSoft relocatable assembler and linking loader
SA-92 MNEMONIC ASSEMBLER – Smoke Signal Broadcasting assembler. Allows multiple source files.

This list is, or course, by no means exhaustive. There are many assemblers on the market – several for each 6809 system – and almost all vary to a greater or lesser extent from the Motorola 6809 assembler standard.

Appendix C
6809 Instruction Set

First some facts and figures. There are 59 different *types* of instruction – LD, PSH, BLE, ASR, and so on. Taking into account the use of different registers to implement these types (CMPA, CMPB, CMPD, etc.) there are 139 *forms* of instruction. Many of these act on memory indicated by many different addressing modes and if we include these differences, we find that there are over 4800 instructions at our disposal. On sheer volume the 6809 totally overwhelms the two most popular 8-bit processors – the Z80 has about 700 different instructions and the 6502 a mere 151. Of course the number of instructions is not the only factor to determine the efficiency or power of a microprocessor. The execution time of the instructions, the ease of performing higher precision (i.e. 16-bit, 32-bit, floating point) arithmetic and flexibility in dealing with external logic – intelligent printers, hardware-decoded keyboards, etc. – are just a few of the many criteria you might use. However, the size and complexity of the instruction set is very important to the programmer.

Larger instruction sets contain a greater degree of redundancy. The instructions in the small repertoire of the 6502 each perform very different tasks. 6502 programming is consequently a rather mechanical job for the programmer who has very little choice in how he or she will actually code a program. The 6809 programmer, on the other hand, is faced with a bewildering three-dimensional array of instructions – dimensioned by type, form and mode many of which seem to do exactly the same operation. JSR n,PC does in three bytes and eight clock cycles what BSR n does in two bytes and seven clock cycles. And at the end of the subroutines we have just called, do we PULS PC or RTS? In an expansive mood we might even LDX ,S++ followed by TFR X,PC. The redundancy illustrated by these few examples is not an oversight of the 6809 design team, nor is it an unfortunate quirk of the three-dimensional structure of the language. Redundancy is *built in* to aid the programmer.

Rather like that extinct species of schoolteacher who defined his goal as instilling the three 'Rs' of reading, 'riting and 'rithmetic (but seemingly not spelling) in the minds of his pupils, the programmer is often concerned with the 'three Ss' of structure, speed and size. The story of the programmer who wrote a subroutine that was not only fast and compact but also so well structured that even his team leader understood it is probably apocryphal: most programs have to trade off two of the three Ss to achieve the third. The high redundancy of the 6809 instruction set allows us to write the same program in many different ways using varied proportions of the three Ss as circumstances dictate.

The superabandance of 6809 instructions is undoubtedly an aid to the experienced 6809 programmer who probably gives as much conscious thought to his choice of instruction as to his choice of words in a casual conversation in his own native dialect. For a beginner in the language it presents problems. A set with very few instructions is easy to learn: for each task there is one instruction that will do the job so no choices have to be made; the operation performed by each instruction is seen as clearly distinct from those of other instructions. The many instructions of the 6809 set seem to defy the 'one task – one instruction' classification which makes for easy learning and use. I mentioned earlier that the 6809 set has a three-dimensional structure based on *instruction type* (the sort of operation performed), *form* (in most cases this defines the register used) and *mode* (whether registers or memory or both are used and the way that memory is accessed). I then gave an example in which two instructions of both different type *and* different mode performed identical jobs. Clearly, if this sort of boundary crossing happens often – and it does – then learning the 6809 language and understanding its finer points only by type, form and mode is going to be very frustrating.

Tables C.1 to C.11 classify the instructions mainly by type. A few instructions are repeated in more than one group but generally each instruction has been consigned to a single group. The problem with this sort of single-entry grouping is that most instructions really belong to a number of diverse classes. For example, all instructions which use a specific register could be grouped together. To find if you can perform an 'Arithmetic Shift Left' on the X register you would refer to the 'Action on X' table and find that although you can ABX, LEAX, STX, LDX, PSHS X, etc. you cannot ASLX.

You might have found out more quickly by looking at a 'Rotate and Shift' table which would inform you that ASL can be carried out on the contents of the A and B registers and on a memory byte but not on X, Y, U, S, DP, D or PC. A useful table to compose would be one giving all the methods of jumping to and returning from a subroutine.

Compiling tables of related instructions for reference during coding is a sound method of learning what options are available for performing various tasks. The chances are that if your brain has gone to work on classifying the instructions to make the tables then you won't even need to look at the tables when you program – except to check up on result flags or instruction timings.

Putting the bytes together

Tables C.1 to C.11 give only the mnemonic and single-byte 'opcode' (in some cases a 2-byte opcode is needed and this is given). But 6809 instructions can be anything from 1 to 5 bytes long and the assembler mnemonic has often to be followed by one or more *operands* defining the registers and/or memory operated on. So how do you decide what can follow the mnemonic and opcode?

The *addressing mode* and a pinch of common sense will tell you exactly how to form the complete instruction.

(1) *Inherent* The operands are implied by the mnemonic and so the code is usually just 1 byte. A few instructions classed as Inherent do need a second byte giving certain information: CWAI needs an 8-bit value to AND with the CC register, EXG and TFR need two register names with their codes in the second byte, and PSH and PUL need the names of all the registers being pushed or pulled.
(2) *Direct* Either 2 or 3 bytes. The last byte gives the location within the page of memory indexed by the DP register. In assembler form this can be a number less than 256 or a label.
(3) *Extended* Either 3 or 4 bytes. The last two bytes specify a memory address. Assemblers will accept either a number less than 65536 ($FFFF or less) or a label.
(4) *Immediate* Operations using 8-bit registers expect a single byte of immediate data (i.e. a value written as the last byte of the instruction) and 16-bit registers need 16 bits of data. In assembly language the data could have been given a name by the EQU directive or have been put in a labelled memory location. In both of these cases the assembler will accept a label as operand.
(5) *Relative* Either an 8-bit or 16-bit signed offset to the PC written

as the last 1 or 2 bytes of the instruction. Assemblers will accept a label as operand and calculate the offset (and decide whether to use the Branch or Long Branch form).
(6) *Indexed/Indirect* All these modes need a post-byte immediately following the opcode, so look up Table C.13 for the post-byte. Table C.12 gives the operand form which corresponds to the particular mode and post-byte. The notes to Table C.12 tell you when any other bytes are needed. Assemblers will, as always, accept valid labels in place of addresses and offsets.

Key to the tables

Mnemonic	Acronym of the operation performed (e.g. BGE is Branch if Greater or Equal).
Action	Description of what the instruction will cause the processor to do.
Time	Execution of each instruction in system clock cycles. A 6809 running at 2MHz uses up two million cycles every second; each cycle is 500 ns (nanoseconds). The total execution time of instructions using the Indexed/Indirect modes is the sum of the cycles given against instruction (Tables C.1 to C.11) and Indexed form (Table C.12).
HNZVC	Flags affected by operations. Flag name: state depends on result. 0 always reset to 0. 1 always set to 1. ? state undetermined. – flag not affected by operation.
M	Single byte memory location.
M:M+1	Consecutive memory locations holding a 16-bit value. Pointers point to the high-order byte.
Code	Instruction opcode given as a pair of hexadecimal digits. May appear under various Addressing modes.
Imm., Dir., Ind., Ext.	Immediate, Direct Page, Indexed/Indirect and Extended Addressing modes. Other modes are Inherent and Relative.
←and→	Data (bit, byte or double-byte) is moved or assigned in the direction of the arrow.
$\wedge$	Logical AND (see below)
$\vee$	Logical INCLUSIVE OR (see below)

⊻	Logical EXCLUSIVE-OR (see below) AND, OR and EOR operate on corresponding individual bits only in the following way:

a	b	a^b	avb	a⊻b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Table C.1. 8-bit Accumulator Only Operations.

Accumulator A		Accumulator B								
Mnem	*Code*	*Mnem*	*Code*	*Action*	*Time*	H	N	Z	V	C
ASLA	48	**ASLB**	58	$C \leftarrow R_7 \sim R_0 \leftarrow 0$	2	?	N	Z	V	C
ASRA	47	**ASRB**	57	$R_7 \rightarrow R_7 \sim R_0 \rightarrow C$	2	?	N	Z	—	C
LSLA	48	**LSLB**	58	$C \leftarrow R_7 \sim R_0 \leftarrow 0$	2	—	N	Z	V	C
LSRA	44	**LSRB**	54	$0 \rightarrow R_7 \sim R_0 \rightarrow C$	2	—	0	Z	—	C
ROLA	49	**ROLB**	59	$C \leftarrow R_7 \sim R_0 \leftarrow C$	2	—	N	Z	V	C
RORA	46	**RORB**	56	$C \rightarrow R_7 \sim R_0 \rightarrow C$	2	—	N	Z	—	C
DECA	4A	**DECB**	5A	$R \leftarrow R-1$	2	—	N	Z	V	—
INCA	4C	**INCB**	5C	$R \leftarrow R+1$	2	—	N	Z	V	—
CLRA	4F	**CLRB**	5F	$R \leftarrow 0$	2	—	0	1	0	0
COMA	43	**COMB**	53	$R \leftarrow \overline{R}$	2	—	N	Z	0	1
NEGA	40	**NEGB**	50	$R \leftarrow \overline{R}+1$	2	?	N	Z	V	C
TSTA	4D	**TSTB**	5D	$R \leftarrow R$	2	—	N	Z	0	—

Notes:
1. $R_7 \sim R_0$ indicates that all bits are shifted by one place left or right.
2. $\overline{R}$ is the one's complement of R (i.e. R⊻$FF).
3. $R \leftarrow \overline{R}+1$ has the same effect as $R \leftarrow 0-R$.
4. R is either of Accumulator A or B.

Table C.2. 8-bit Memory Only Operations.

Mnem	*Imm* (—)	*Dir* (6)	*Ind* (6+)	*Ext* (7)	*Action*	H	N	Z	V	C
ASL	—	08	68	78	$C \leftarrow M_7 \sim M_0 \leftarrow 0$	?	N	Z	V	C
ASR	—	07	67	77	$M_7 \rightarrow M_7 \sim M_0 \rightarrow C$	?	N	Z	—	C
LSL	—	08	68	78	$C \leftarrow M_7 \sim M_0 \leftarrow 0$	—	N	Z	V	C
LSR	—	04	64	74	$0 \rightarrow M_7 \sim M_0 \rightarrow C$	—	0	Z	—	C
ROL	—	09	69	79	$C \leftarrow M_7 \sim M_0 \leftarrow C$	—	N	Z	V	C
ROR	—	06	66	76	$C \rightarrow M_7 \sim M_0 \rightarrow C$	—	N	Z	—	C
DEC	—	0A	6A	7A	$M \leftarrow M-1$	—	N	Z	V	—
INC	—	0C	6C	7C	$M \leftarrow M+1$	—	N	Z	V	—
CLR	—	0F	6F	7F	$M \leftarrow 0$	—	0	1	0	0
COM	—	03	63	73	$M \leftarrow \overline{M}$	—	N	Z	0	1
NEG	—	00	60	70	$M \leftarrow \overline{M}+1$	?	N	Z	V	C
TST	—	0D	6D	7D	$M \leftarrow M$	—	N	Z	0	—

Notes:
1. See notes to Table C.1.
2. Instruction times (clock cycles) are given at the column heads.
3. M is the addressed single byte of memory.

Table C.3. Test and Compare.

Mnem	*Imm* (2)	*Dir* (4)	*Ind* (4+)	*Ext* (5)	*Action*	H	N	Z	V	C
TSTA	(Inherent.	Code: 4D.	Time: 2)		$A \leftarrow A$	—	N	Z	0	—
TSTB	(Inherent.	Code: 5D.	Time: 2)		$B \leftarrow B$	—	N	Z	0	—
TST	—	0D	6D	7D	$M \leftarrow M$	—	N	Z	0	—
BITA	85	95	A5	B5	$A \wedge M$	—	N	Z	0	—
BITB	C5	D5	E5	F5	$B \wedge M$	—	N	Z	0	—
CMPA	81	91	A1	B1	$A-M$	?	N	Z	V	C
CMPB	C1	D1	E1	F1	$B-M$	?	N	Z	V	C
CMPD	1083	1093	10A3	10B3	$D-M:M+1$	—	N	Z	V	C
CMPU	1183	1193	11A3	11B3	$U-M:M+1$	—	N	Z	V	C
CMPX	8C	9C	AC	BC	$X-M:M+1$	—	N	Z	V	C
CMPY	108C	109C	10AC	10BC	$Y-M:M+1$	—	N	Z	V	C
CMPS	118C	119C	11AC	11BC	$S-M:M+1$	—	N	Z	V	C

Notes:
1. Basic instruction times (clock cycles) for BITA, BITB, CMPA and CMPB are given at the column heads.
2. Add 2 cycles for TST and CMPX in all modes.
3. Add 3 cycles for CMPD, CMPU, CMPY and CMPS in all modes.
4. Only status is affected by these instructions; memory and registers are unchanged.

Table C.4. Arithmetic and Logic.

Mnem	*Imm* (2)	*Dir* (4)	*Ind* (4+)	*Ext* (5)	*Action*	H	N	Z	V	C
ADCA	89	99	A9	B9	A←A+M+C	H	N	Z	V	C
ADCB	C9	D9	E9	F9	B←B+M+C	H	N	Z	V	C
ADDA	8B	9B	AB	BB	A←A+M	H	N	Z	V	C
ADDB	CB	DB	EB	FB	B←B+M	H	N	Z	V	C
ADDD	C3	D3	E3	F3	D←D+M:M+1	H	N	Z	V	C
SBCA	82	92	A2	B2	A←A−M−C	?	N	Z	V	C
SBCB	C2	D2	E2	F2	B←B−M−C	?	N	Z	V	C
SUBA	80	90	A0	B0	A←A−M	?	N	Z	V	C
SUBB	C0	D0	E0	F0	B←B−M	?	N	Z	V	C
SUBD	83	93	A3	B3	D←D−M:M+1	—	N	Z	V	C
ANDA	84	94	A4	B4	A←A^M	—	N	Z	0	—
ANDB	C4	D4	E4	F4	B←B^M	—	N	Z	0	—
EORA	88	98	A8	B8	A←A⊻M	—	N	Z	0	—
EORB	C8	D8	E8	F8	B←B⊻M	—	N	Z	0	—
ORA	8A	9A	AA	BA	A←AvM	—	N	Z	0	—
ORB	CA	DA	EA	FA	B←BvM	—	N	Z	0	—

Notes:
1. Instruction (clock cycles) for 8-bit operations are given a t the column heads.
2. 16-bit operations (ADDD and SUBD) take 2 cycles longer in all modes.

Table C.5. Register-Memory Transfer.

Mnem	*Imm* (2)	*Dir* (4)	*Ind* (4+)	*Ext* (5)	*Action*	H N Z V C
LDA	86	96	A6	B6	A←M	— N Z 0 —
LDB	C6	D6	E6	F6	B←M	— N Z 0 —
LDD	CC	DC	EC	FC	D←M:M+1	— N Z 0 —
LDX	8E	9E	AE	BE	X←M:M+1	— N Z 0 —
LDY	108E	109E	10AE	10BE	Y←M:M+1	— N Z 0 —
LDU	CE	DE	EE	FE	U←M:M+1	— N Z 0 —
LDS	10CE	10DE	10EE	10FE	S←M:M+1	— N Z 0 —
STA	—	97	A7	B7	M←A	— N Z 0 —
STB	—	D7	E7	F7	M←B	— N Z 0 —
STD	—	DD	ED	FD	M:M+1←D	— N Z 0 —
STX	—	9F	AF	BF	M:M+1←X	— N Z 0 —
STY	—	109F	10AF	10BF	M:M+1←Y	— N Z 0 —
STU	—	DF	EF	FF	M:M+1←U	— N Z 0 —
STS	—	10DF	10EF	10FF	M:M+1←S	— N Z 0 —

Notes:

1. Basic instruction times (clock cycles) for 8-bit transfers (A and B) are given at the column heads.
2. Add 1 cycle for transfers involving D, X or U in all modes.
3. Add 2 cycles for transfers involving Y or S in all modes.

Table C.6. Stack Operations.

Mnem	*Code*	*Time*	*Action*	H N Z V C
PSHS	34	5+	For each byte pushed:	— — — — — —
PSHU	36	5+	S/U←S/U−1: (S/U)←byte.	— — — — — —
PULS	35	5+	For each byte pulled:	— — — — — —
PULU	37	5+	byte←(S/U): S/U←S/U+1	— — — — — —

Notes:

1. All instructions require a post-byte with the following register-bit correspondence:

Bit:	7	6	5	4	3	2	1	0
Register:	PC	U/S	Y	X	DP	B	A	CC

2. Post-byte *set* bits result in the corresponding registers being pushed/pulled.
3. PSHS/PULS will push/pull U when bit 6 is set, PSHU/PULU will push/pull S when bit 6 is set.
4. Higher-bit registers are *pushed* before lower-bit, lower-bit registers are *pulled* before higher-bit.
5. Add 1 clock cycle to the basic instruction time for each *byte* (not register) pushed or pulled.
6. Flags (CC register) are unaffected only if no value is pulled to the CC register during PULS or PULU.

Table C.7. Register Exchange and Transfer.

Mnem	*Code*	*Time*	*Action*	H N Z V C
EXG	1E	7	Register 1↔Register 2	— — — — —
TFR	1F	7	Register 1↔Register 2	— — — — —

Notes:

1. Both instructions require a post-byte with the high- and low-order hexadecimal digits giving the codes for registers 1 and 2 respectively.
2. The register-digit correspondence is:

	(16-bit regs.)						(8-bit regs.)		
Digit: 0	1	2	3	4	5	8	9	A	B
Register: D	X	Y	U	S	PC	A	B	CC	DP

3. EXG or TFR is illegal between registers of differing length.
4. Flags (CC register) are unaffected only if the CC register is not the destination register.
5. The effect of codes 6, 7, C, D, E and F is undefined.

Table C.8. Pointer Manipulation.

Mnem	*Dir*	*Ind*	*Ext*	*Action*	H N Z V C
JMP	0E (3)	6E (3+)	7E (4)	PC←*EA*	— — — — —
JSR	9D (7)	AD (7+)	BD (8)	Stack ←PC	— — — — —
				PC←*EA*	
LEAX	—	30 (4+)	—	X←*EA*	— — Z — —
LEAY	—	31 (4+)	—	Y←*EA*	— — Z — —
LEAS	—	32 (4+)	—	S←*EA*	— — — — —
LEAU	—	33 (4+)	—	U←*EA*	— — — — —

Notes:
1. Instruction times (clock cycles) are given in parentheses.
2. All of these instructions are designed to deal with pointers to values rather than actual values. Hence the normal meaning given to each addressing mode does not apply. *One level of indirection is removed.*
3. *EA* is Effective Address. This is distinct from the notion of a 16-bit value as illustrated in the following examples:
   (a) LDX 2,Y will load X with the value *held in memory at Y+2:Y+3* but LEAX 2,Y will load X with the value (address) *held in Y* and add 2 to that value.
   (b) JMP $89AB (Extended mode) will load the PC with the value (address) $89AB NOT the value held in memory at $89AB and $89AC.

Table C.9. Program Relative Branching.

Mnem	*Code*	*Condition*				*Mnem*	*Code*	*Condition*			
		N	Z	V	C			N	Z	V	C
(a) Simple conditional											
BCC	24	—	—	—	0	**BCS**	25	—	—	—	1
BNE	26	—	0	—	—	**BEQ**	27	—	1	—	—
BVC	28	—	—	0	—	**BVS**	29	—	—	1	—
BPL	2A	0	—	—	—	**BMI**	2B	1	—	—	—
(b) Unsigned conditional											
BHI	22	—	0	—	0	**BLS**	23	—	—	—	1
								—	1	—	—
BHS	24	—	—	—	0	**BLO**	25	—	—	—	1
(c) Signed conditional											
BGT	2E	0	0	0	—	**BLE**	2F	—	1	—	—
		1	0	1	—			0	—	1	—
								1	—	0	—
BGE	2C	0	—	0	—	**BLT**	2D	0	—	1	—
		1	—	1	—			1	—	0	—
(d) Unconditional											
BRA	20	*always*				**BRN**	21	*never*			
(e) Subroutine											
BSR	8D	*always (after pushing PC to hardware stack)*									

Notes:

1. Flags (CC register) are unaffected by Branch instructions.
2. The flag states listed are those causing the branch. Where more than one configuration is given, any one of the patterns will cause branching.
3. The mnemonics and codes given are for the 8-bit offset forms of the Branch instruction. Each has a 16-bit *Long Branch* form with the following differences: (a) the mnemonic has an 'L' prefix (e.g. LBCC), (b) the code has a $10 pre-byte (e.g. $1024).
4. The instruction must be followed by a byte giving a signed value offset (−128 to +127). Long Branches must be followed by a 2-byte signed offset (−32768 to +32767).
5. *Action:* If condition true then PC←PC + offset.
6. *Time:* 8-bit form: 3 clock cycles (Branch or No Branch); 16-bit form: 5 cycles (No Branch), 6 cycles (Branch).
7. *Special cases:* BSR takes 7 cycles. A special form of LBRA (code $16) takes 5 cycles. A special form of LBSR (code $17) takes 9 cycles.
8. BRN (Branch Never) can be used for fine-tuning in precision timing situations or to mark a possible branch position during program development.

Table C.10. Interrupts.

Mnem	*Code*	*Action*	H N Z V C
CWAI	3C	CC Register ^ data: E Flag←1 Stack←PC,U,Y,X,DP,B,A,CC Wait for interrupt.	Note 1
SWI	3F	(Software Interrupt) E Flag←1 Stack←PC,U,Y,X,DP,B,A,CC F Flag←1: I Flag←1 PC←Interrupt Subroutine Vector stored in $FFFA and $FFFB	— — — — —
SWI2	103F	*As* SWI *except* F and I Flags unaltered *and* PC←Vector at $FFF4 and $FFF5	— — — — —
SWI3	113F	*As* SWI *except* F and I Flags unaltered *and* PC←Vector at $FFF2 and $FFF3	— — — — —
SYNC	13	Halt processing until interrupt. *If* interrupt disabled (F or I = 1) or interrupt request <3 cycles *Then* continue processing *Else* stack registers and transfer control to interrupt service routine.	— — — — —
RTI	3B	Return from Interrupt. CC←Stack *If* E Flag = 0 *Then* PC←Stack *Else* A,B,DP,X,Y,U,PC←Stack	Note 2

Notes:

1. CWAI requires a second (immediate data) byte to be logically ANDed with the CC register. The purpose of this is to clear either or both of the interrupt flags (F and I) before suspending operation to enable interrupts. Status (flags) thus depends on the result of the AND.
2. Status is restored to that before the interrupt occurred.
3. *Time:* SWI, SWI2 and SWI3 execute in 19, 20 and 20 clock cycles respectively. RTI takes 6 cycles if only the CC and PC registers have to be restored, 15 cycles if all registers are to be restored. CWAI and SYNC have time specifications of 20 and 2 cycles, respectively, but since both of these instructions wait for external events the timing is indeterminate.
4. Software interrupts are most often used for setting breakpoints and 'on error' service routines.
5. CWAI is used to provide an extremely quick response to an expected interrupt, since the state of the machine (registers) is saved before the interrupt occurs.
6. SYNC is used to synchronise operations to external events. Short pulse interrupt requests (less than 3 cycles) will simply restart program execution from the next instruction.

Table C.11. Program Control and Special Purpose Arithmetic.

Mnem	*Code*	*Action*	*Time*	*H N Z V C*
NOP	12	No Operation	2	— — — — —
ORCC	1A	CC←CCvdata	3	Note 1
ANDC	1C	CC←CC^data	3	Note 1
RTS	39	PC←Stack	5	— — — — —
ABX	3A	X←X+B (unsigned)	3	— — — — —
DAA	19	A←BCD adjusted A	2	— N Z 0 C
MUL	3D	D←A×B (unsigned)	11	— — Z — C
SEX	1D	D←16-bit B (A←sign B)	2	— N Z 0 —

Notes:
1. ORCC and ANDCC require a following byte of immediate data. Status is the result of the logical operation.
2. ORCC is used to set flags, ANDCC to clear them.
3. Note the difference between *ABX*, where B is an unsigned value in the range 0 to 255, and *LEAX B, X* where B is a signed value in the range −128 to +127.
4. After MUL the C Flag contains the sign of B. This is to facilitate rounding up the high-order result byte in A.
5. RTS may also be effected by *PULS PC*.
6. DAA is used following arithmetic operations on Binary Coded Decimal values to correct the result to BCD. Since it uses the Carry and Half-carry flags, it must be used before any other instructions alter the status.

Table C.12. Indexed Addressing Modes – Form and Timing.

Type	*Non-indirect* *Form*	*Time*	*Indirect* *Form*	*Time*
Constant Offset from R				
No offset	,R	0	[,R]	3
5-bit offset	p,R	1	*defaults to 8-bit*	
8-bit offset	n,R	1	[n,R]	4
16-bit	nn,R	4	[nn,R]	7
Accumulator from R				
A offset (8-bit)	A,R	1	[A,R]	4
B offset (8-bit)	B,R	1	[B,R]	4
D offset (16-bit)	D,R	4	[D,R]	7
Auto Increment/Decrement R				
Increment by 1	,R+	2	*not allowed*	
Increment by 2	,R++	3	[,R++]	6
Decrement by 1	,–R	2	*not allowed*	
Decrement by 2	,––R	3	[,––R]	6
Constant Offset from PC				
8-bit offset	n,PC	1	[n,PC]	4
16-bit offset	nn,PC	5	[nn,PC]	8
Extended Indirect				
16-bit address	*use Ext. Mode*		[nn]	5

Notes:
1. *R* is any of registers X, Y, U or S.
2. *Time* gives the number of clock cycles to be added to the basic instruction times given in Tables C.1 to C.11.
3. *p* is a 5-bit signed offset (range –16 to +15) encoded in bits 4 to 0 of the post-byte.
4. *n* is an 8-bit signed offset (range –128 to +127) which must follow the post-byte.
5. *nn* is a 16-bit signed offset (range –32768 to +32767 which must follow the post-byte.
6. The No offset, 5-bit offset, Accumulator offset and Auto Inc/Dec forms do not need data bytes following the post-byte.
7. Since the meaning of the Indirect Auto Inc/Dec Mode is to indirectly use a table of 16-bit addresses, an increment or decrement by 1 would be an error in programming and is not allowed.
8. The usual assembler forms for the Constant Offset from PC type are [*label.PCR*] and *label.PCR*. The assembler will calculate the offset at assembly time.

Table C.13. Indexed Addressing Modes – Post-byte Codes.

(a) Indexing by X, Y, U or S

R:	X	Y	U	S	R:	X	Y	U	S
0,R	00	20	40	60	−16,R	10	30	50	70
1,R	01	21	41	61	−15,R	11	31	51	71
2,R	02	22	42	62	−14,R	12	32	52	72
3,R	03	23	43	63	−13,R	13	33	53	73
4,R	04	24	44	64	−12,R	14	34	54	74
5,R	05	25	45	65	−11,R	15	35	55	75
6,R	06	26	46	66	−10,R	16	36	56	76
7,R	07	27	47	67	−9,R	17	37	57	77
8,R	08	28	48	68	−8,R	18	38	58	78
9,R	09	29	49	69	−7,R	19	39	59	79
10,R	0A	2A	4A	6A	−6,R	1A	3A	5A	7A
11,R	0B	2B	4B	6B	−5,R	1B	3B	5B	7B
12,R	0C	2C	4C	6C	−4,R	1C	3C	5C	7C
13,R	0D	2D	4D	6D	−3,R	1D	3D	5D	7D
14,R	0E	2E	4E	6E	−2,R	1E	3E	5E	7E
15,R	0F	2F	4F	6F	−1,R	1F	3F	5F	7F
,R+	80	A0	C0	E0	[,R+]	*not allowed*			
,RH	81	A1	C1	E1	[,RH]	91	B1	D1	F1
,−R	82	A2	C2	E2	[,−R]	*not allowed*			
,−−R	83	A3	C3	E3	[,−−R]	93	B3	D3	F3
,R	84	A4	C4	E4	[,R]	94	B4	D4	F4
B,R	85	A5	C5	E5	[B,R]	95	B5	D5	F5
A,R	86	A6	C6	E6	[A,R]	96	B6	D6	F6
n,R	88	A8	C8	E8	[n,R]	98	B8	D8	F8
nn,R	89	A9	C9	E9	[nn,R]	99	B9	D9	F9
D,R	8B	AB	CB	EB	[D,R]	9B	BB	DB	FB

(b) Program Counter Offset and Extended Indirect

	Four options					Four options			
n,PC	8C	AC	CC	EC	[n,PC]	9C	BC	DC	FC
nn,PC	8D	AD	CD	ED	[nn,PC]	9D	BD	DD	FD
nn	*use Extended Mode*				[nn]	9F	BF	DF	FF

Appendix D

ASCII Control and Character Codes

ASCII (American Standard Code for Information Interchange) control codes ($00 to $1F and $7F) were designed for terminal control. Most of them have no use within the stand-alone micro. A few are often used as cursor control codes by character and string print routines. The character codes ($20 to $7E) are almost always used for inter-computer communicating and often for file storage on tape or disk. They are not always used for internal character representation.

Table D.1. Control code meanings.

$00	NUL	Null	$10	DLE	Data Link Escape
$01	SOH	Start of Heading	$11	DC1	Direct Control 1
$02	STX	Start Text	$12	DC2	Direct Control 2
$03	ETX	End Text	$13	DC3	Direct Control 3
$04	EOT	End of Transmission	$14	DC4	Direct Control 4
$05	ENQ	Enquiry	$15	NAK	Negative Acknowledge
$06	ACK	Acknowledge	$16	SYN	Synchronous Idle
$07	BEL	Bell	$17	ETB	End Transmission Block
$08	BS	Backspace	$18	CAN	Cancel
$09	HT	Horizontal Tab	$19	EM	End of Medium
$0A	LF	Line Feed	$1A	SUB	Substitute
$0B	VT	Vertical Tab	$1B	ESC	Escape
$0C	FF	Form Feed	$1C	FS	Form Separator
$0D	CR	Carriage Return	$1D	GS	Group Separator
$0E	SO	Shift Out	$1E	RS	Record Separator
$0F	SI	Shift In	$1F	US	Unit Separator
$7F	DEL	Delete	$20	SP	Space

Table D.2. ASCII character codes.

L.S.	*Most significant hexadecimal digit*					
digit	2	3	4	5	6	7
0	SP	0	@	P	`	p
1	!	1	A	Q	a	q
2	"	2	B	R	b	r
3	#	3	C	S	c	s
4	$	4	D	T	d	t
5	%	5	E	U	e	u
6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(	8	H	X	h	x
9	)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[	k	{
C	,	<	L	\	l	|
D	—	=	M	]	m	}
E	.	>	N	↑	n	~
F	/	?	O	←	o	DEL

Appendix E

Some 6809 Computer Systems

The following list is just a small sample of the available 6809 systems. Most are designed as business machines or. boards for industrial control applications. In the home computer field, only Tandy/Radio Shack and Dragon Data have built complete systems around the 6809, although you can get 6809 add-on boards for other home/small business computers.

The TRS-80 Color Computer

This computer is built around the MC6809E processor with a clock speed of 0.894 MHz. Other on-board hardware devices include the MC6883 Synchronous Address Multiplexer, MC6847 Video Display Generator and two MC6821 PIAs. Video display is output to a normal television and is capable of thirteen different modes from text with a 32 × 64 pixel graphics capability to a 192 × 256 dot graphics without text. Sound can be software generated and output to a television speaker. Available with either 16K or 32K dynamic RAM.

Further details can be obtained from most Tandy dealers or from Tandy's Walsall head office on 0922 648181.

The Dragon 32 and 64

The Dragon is remarkably similar to the Color Computer and is available from many high street computer and electrical stores.

If you can't get the details, try Dragon Data on 0792 580651.

Positron 900 and 9000

The Positron 900 is based on the MC6809 and is built as a single board requiring the attachment of keyboard terminal. Dynamic RAM memory is expandable from 64K to 256K and up to 128K of on-board ROM is supported. The processor unit has four RS232C serial ports and an IEEE 488 interface. Up to seven processor units and eight disk drives can be networked using the Positron 9300 Network Controller which has its own 6809 processor.

The Positron 9000 Work Station incorporates the 9000/1 Main Processor Board and the 9000/2 Video/Keyboard. The keyboard is software decoded for easy user modification. The video output is Viewdata compatible with 24 rows of 40 characters on a 14 × 10 dot matrix. The screen is 1K memory mapped or 10K memory mapped for 240 × 240 pixel graphics, with mixed text. Video output is to composite video, direct video or channel 36. The board also has a 0 to 4kHz tone generator for output to an external speaker.

Further details are available from Positron Computers on 09252 29741.

SEED System 19/64DS5

This is based on a CPU board with a 6809 processor running at 2 MHz, a disk controller and 57K RAM. It has three RS232C serial interfaces, one parallel interface and one SASI interface for a Winchester disk add-on.

System 19 add-on boards include the SCB-69 CPU board with a 6809 processor at 2 MHz, 1K of scratchpad RAM and a 10 ms interrupt real-time clock with signals from months to 0.001 seconds, the SEED PTM-1 board based on the MC6840 timer with three 16-bit counters and associated control registers, a 256K dynamic RAM board and several D/A and A/D converters.

Details from Strumech Engineering Electronic Developments Ltd on 05433 78151 or 4321.

Windrush Micro Systems

Windrush supply a large range of systems and add-on boards intended mostly for industrial control or as development systems.

The Euro-3X development system features a 2 MHz 68B09 processor, 56K static RAM with battery back-up, two R6551A RS232 ports, two R6522 parallel ports, seven 16-bit timers (one MC68B40 and two R6522A) and a battery backed MM58167 clock-calendar.

The PRIVAC BT-1 512 by 480 intelligent graphics controller board has its own 6809 CPU running at 1.5 MHz. Text and graphics can be mixed with 43 lines of 83 characters in a 6 × 10 dot matrix character cell as default but are capable of being set to 15 different sizes in four orientations. Four 96 character sets are available. The board also includes an eight-channel 8-bit A/D converter for joystick, tracker ball or mouse control. The board has 6K of firmware, expandable up to 20K and communicates with the host board via only 4 bytes on the host memory map.

The GIMIX 6809+ CPU board includes jumper selectable clock speeds of 1, 1.5 or 2 MHz for the 6809 and clock speeds of 2, 3 or 4 for the optional 9511A or 9512 Arithmetic processor. It also has a 6840 programmable timer and a battery backed 58167 real-time clock.

Further details about these and many other boards and systems from Windrush Micro Systems on 0692 405189.

Further Reading

Books

One book like this can only provide you with a glimpse of the exciting and challenging field of machine code programming. The key to success in developing your new knowledge lies in hard work and receptiveness to ideas. The following list of books should provide you with a lot of the information you need to extend your programming abilities.

Sinclair, James and Barden are more or less introductory books but they do contain information about the Dragon and Color Computer (and about 6809 programming generally) that I did not have room for in this book. DeMarco and Leventhal are essential reading if you want to progress further – both of them give many useful references. Don't bother getting Knuth unless you are contemplating doing a course in computer science or are inordinately fond of mathematics. I have included the very informative psychology book to remind you that computers are built, programmed and used by people. Hofstadter and Spencer-Brown will both concentrate and expand your thoughts – read them together.

Motorola books

MC6809–MC6809E Microprocessor Programming Manual
Motorola Microprocessor Products Data Manual

Tandy (Radio Shack) books

Color Computer Graphics by William Barden Jr.
Cat. No. 62–2076
TRS-80 Color Computer Assembly Language Programming
by William Barden, Jr. Cat. o. 62–2077
TRS-80 Color Computer Technical Reference Manual
Cat. No. 26–3193

Brown, R. and R. J. Herrnstein. *Psychology*. London: Methuen, 1975.

DeMarco, Tom. *Structured Analysis and System Specification*. New York: Yourdon Inc., 1978.

Hofstadter, Douglas R. *Gödel, Escher, Bach: an Eternal Golden Braid*. London: Penguin Books, 1980.

James, Mike. *Anatomy of the Dragon*. Sigma Technical Press, 1983.

Knuth, Donald E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Massachusetts: Addison-Wesley, 1973 (2nd edition).

Leventhal, Lance A. *6809 Assembly Language Programming*. Berkeley: Osborne/McGraw-Hill, 1981.

Sinclair, Ian. *Introducing Dragon Machine Code*. London: Granada, 1984.

Spencer-Brown, G. *Laws of Form*. New York: E. P. Dutton, 1979.

Magazines

Reading magazines will keep you up to date with hardware developments. No magazine, as far as I know, is aimed solely at the writer of assembly language programs, though some do publish machine code listings and 'hex dumps' occasionally.

Personal Computer World has been running a series since 1980 called 'PCW SUB SET'. This publishes three or four machine code routines submitted by readers in each issue. Give it a try.

ROM

You can get a lot of ideas from the professionally written programs inside your computer system. Buy a disassembler (or write one) and find out how your BASIC is written. Just remember, though, that the software inside your machine is copyright so you can't use it commercially.

Index of Routines

Index

MACHINE CODE – POWERFUL . . . EFFECTIVE . . . ATTAINABLE!

At the heart of the Dragon, TRS-80 Color Computer and other computer systems, the 6809 microprocessor performs up to a million operations every second. The speed of interpreted BASIC, however, is measured in mere hundreds of actions per second. Machine code is the only way that you, the programmer, can harness the full power of the machine– for really fast games, accurate timing to thousandths of a second and total control of all functions.

This book introduces you to 6809 machine code, the professional programming methods that will save you time and frustration, and tells you how to take command of the support chips dealing with sound, graphics, keyboard and other input/output functions. Many essential routines are given with explanatory documentation to show the 6809 in action. Perhaps most importantly for Dragon owners, the software is provided to put text on a high resolution screen with a fully re-definable character set.

The Author
David Barrow is well-known to machine code enthusiasts as a prolific contributor to, and later presenter of, the machine code series PCW SUB SET in *Personal Computer World*. He is the co-author, with Alan Tootill, of *Z80 Machine Code For Humans* and *6502 Machine Code For Humans*.

Front cover illustration by Angus McKie

GRANADA PUBLISHING
Printed in Great Britain
0 246 12532 2

£7.95 net

BARROW 6809 MACHINE CODE PROGRAMMING GRANADA