machine code games routines for the commodore 64

essential routines for game design

paul roper



machine code games routines for the commodore 64

essential routines for game design

paul roper

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street
London WC2R 3LD

Copyright © Paul Roper, 1984
Reprinted 1984

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording and/or otherwise, without the prior written permission of the Publishers.

British Library Cataloguing in Publication Data
Roper, Paul
Machine code games routines for the Commodore 64.
1. Computer games 2. Commodore 64 (Computer) — Programming
I. Title
794.8'028'5425 GV1469.2

ISBN 0-946408-47-5

Cover design by Grad Graphic Design Ltd.
Illustration by Richard Dunn.
Typeset, printed and bound by Trintype, Nene Litho and Woolnough Bookbinding, Wellingborough, Northants.

CONTENTS

		Page
Author's Note		vii
1	Introduction	1
2	Further 6510 Theory	7
3	The Birth of a Game	23
4	Program Design	29
5	Coding the Program	39
6	Writing Your Own Routines	45
7	Testing and Debugging	53
8	Connections: Stringing the Game Together	61
9	Direct Routines	67
10	Indirect Routines	99
APPENDIX A: Utility Programs		133
APPENDIX B: Mini-assembler		145
APPENDIX C: Advanced Machine Characteristics		153
APPENDIX D: Laserbike		161
Index of Routines		167

Contents in detail

CHAPTER 1
Introduction
The process of writing games, how this book is set out, interrupts and the 64, machine code and BASIC.

CHAPTER 2
Further 6510 Theory
The workings of the 6510 and 6502 chips, addressing techniques, the stack, multi-bit operations, logical operations, scanning the screen, multiplication and shifts, data structures and tables.

CHAPTER 3
The Birth of a Game
Beginning to write games in BASIC and/or machine code, the essential ingredients of a good game, developing your game idea.

CHAPTER 4
Program Design
Drawing up charts to divide the game into a series of minor tasks, looking at top-down charts, how charts help in writing programs.

CHAPTER 5
Coding the Program
Functional routines, parameter passing, compiling routines, parameter clashes, assemblers.

CHAPTER 6
Writing Your Own Routines
How to write your own routines, turning algorithms into machine code routines, the development of a Spiral Screen Fill routine, algorithm design to stimulate your brain to solve problems and produce new routines.

CHAPTER 7
Testing and Debugging
The causes of bugs, keeping routines simple, testing the routines, debugging.

CHAPTER 8
Connections: Stringing the Game Together
What connections need to be made, iteration, DO–UNTIL loops, conditional branching, checking bugs in connectors.

CHAPTER 9
Direct Routines
Text and slow printing, Filling memory, Copying memory, Delays, Updating bombs, Scrolling, Interaction, Inverting and Explosions, Alternative sprite system.

CHAPTER 10
Indirect Routines
The interrupt, Sprites, Further uses of UDGs, Sound, Large-scale games, Fleet movements, Random numbers, Non-linear motion, Color memory.

APPENDIX A
Utility Programs
Recovering a crashed program, siting the program and memory management, entering machine code from BASIC, user-definable graphics and a UDG editor, sprites and a sprite editor.

APPENDIX B
Mini-assembler
BASIC assembler including a disassembler, program description, using the program, Mini-assembler listing, user-definable function keys, dynamic halt, quotes-mode alleviator, sound and sprite mute, Mini-toolkit.

APPENDIX C
Advanced Machine Characteristics
The keyboard and its buffer, BASIC text and variable storage, the CIAs, SID and VIC memory maps.

APPENDIX D
Laserbike
Complete game listing.

Author's Note

Here is a collection of routines, methods and techniques which are all useful in the construction of games. I have had great pleasure in compiling this list — I have discovered much about the machine that I didn't know before. There were many routines that couldn't be listed in this book for various reasons. The ones listed here, however, should suffice to give you a head-start in the world of machine code games programming. Don't stick rigidly to my methods and ideas: let yourself develop as you feel you should. I am totally self-taught and thus have my own particular style.

This book is designed to be more than just a collection of routines. All the ideas are explained so that you can confidently alter them, mix them and tease them apart. Don't just type them in and use them — do them justice.

Throughout the book I refer to both the 6502 and the 6510. This is a matter of loyalty. As far as programming is concerned, the two are identical. I had been programming the 6502 years before the 64 arrived, so I feel a certain amount of afinity for it.

On a final note I would like to thank my friends and family who bore with me and offered helpful suggestions while I wrote the book.

Program notes

In the computer listings in this book, the following symbols have been replaced. Please ensure that you key in the correct sign.

£ *appears as* \
← *appears as* —
↑ *appears as* ^

Commands which appear in the listings in square brackets are listed below:

[CLS] – Clear Screen
[HOM] – Cursor Home
[CD] – Cursor Down
[CU] – Cursor Up
[CL] – Cursor Left
[CR] – Cursor Right
[RVS] – Reverse On
[OFF] – Reverse Off
[FX] – Function Key X (eg [F1] , [F2])

CHAPTER 1
Introduction

The language of BASIC is remarkable in two ways: it's remarkably simple to learn and apply and it's remarkably slow. Machine code is also simple to learn (although much harder to write) but is remarkably fast. When the two are mixed a hybrid form of programming evolves: one that is simple to use yet fast. Ideally the best programs are written in 100 per cent machine code, but to write programs like this demands a lot of practice and experience. A mixture provides the best of both worlds. This book is about using machine code in games in this way. Writing pure machine code is simply a logical extension of some of the ideas, but don't rush it, it'll come naturally.

The problem with machine code is that it's difficult to be objective. The instructions themselves are so ridiculously elementary that any large task seems daunting to say the least. By contrast, BASIC lends itself well to the construction of longer, more complex programs.

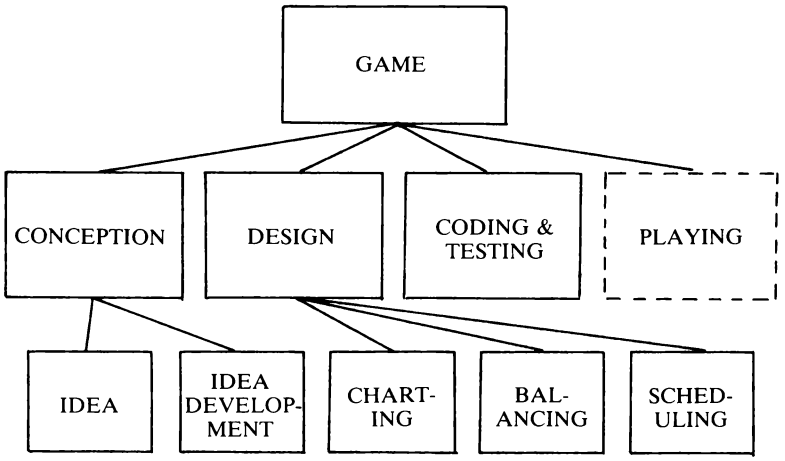
Look at it in this way. By now you probably 'think in BASIC' quite comfortably. When you start to learn French at school, the language seems very abstract but, after a time, the ability to think in the language itself becomes easier. The same is true of machine code. After a while you will develop a feel for the language proper. Mixing machine code with BASIC is a powerful way of learning. The more you do it, the bigger the role machine code will take in your games and, on top of this, you will minimise the amount of time needed to write a decent game. The 'balance of the game' is an important idea and the keynote in machine code programming is simplicity. Keep the amount of machine code down to start with and work up gradually, while keeping the machine code itself simple.

Writing games

Figure 1.1 illustrates the process of writing a game. Each stage, right from the first idea, is examined in detail. You start at the left and work steadily to the right. By the time you get there, you have a working game. This type of chart is known as a top-down chart (you'll be seeing a lot of charts so get used to them — although they are all very basic). In

the top-down chart, you start at the top and work down, fleshing out the process as you go. The result is a clear representation of the structure of a process.

Figure 1.1: The Game Production Process.



The first stage is the *conception*. Silly though it may seem to spend a lot of time thinking about the function of the game itself, you are going to expend a considerable amount of time and effort on the game so make this worthwhile by finding a good idea. The *idea* is the underlying principle of the game, whether it is stealing apples or destroying Klingon starbases. Then the idea is *developed* by building up a set of refinements. These might be things like mother ships and power-pills which liven up the game. Two things make a game attractive, good presentation and all the little finishing touches, so take your time at this stage.

Design is the next stage. Here, the game is *charted* so that we can see what has to be done. Then comes *balancing* where the BASIC-machine-code mix is decided. Next, with the machine code routines identified, *scheduling* begins. When writing in machine code, it's vital to document as you go. All the charts should be numbered and the routines all neatly listed along with their functions. All the documentation should be kept neatly in a file. Documenting does provide a great deal of satisfaction if it's done neatly. It's nice if, at the end of the day, you have a neat file containing all the details of the program. Throughout the book I make reference to the 'program files' and you will find that, if you do make use of one, it will ease a lot of the head-scratching moments when you need to know the details of how a routine passes data to and from itself.

Returning to scheduling, this simply means looking at the program files and planning a programme (not program) of *coding, testing* and documenting so that you can work to set deadlines.

Coding and testing is where the game really gets under way. This is the process of constructing routines to perform various operations and fiddling around until they work. After all this is done, the program is finished and playing can commence.

The routines

As the title says, this book contains a wealth of machine code routines. These perform a lot of simple and complex tasks that help take the pressure off BASIC programs. However they are not intended for you just to copy in and forget about. If you are going to use a routine, then take the trouble to see how it works — you may have to make alterations to it and, if you don't understand it, then modifying becomes impossible. As this was the intention, the routines are accompanied by a description of the topic, how the machine code program works and occasionally a few points on style in the programs. The routines themselves are annotated with short remarks and program flow is shown by arrows so that, at a glance, you can make cross references to the flowcharts that go with them. The result of all this support is that they are more than just routines, they are methods and techniques. The routine section itself is divided into two — direct and indirect. The direct routines perform fairly clearcut tasks such as filling memory or scrolling while the indirect section concerns itself with concepts (supported by routines) such as moving fleets of objects around the screen.

The book

Not all the routines in the book are contained in that section though. You will find them hidden all over (they are listed in the routine index at the end of the book) often just to prove a point to make it interesting. The appendices also contain more important information such as using the function keys as well as a collection of utility programs. The idea is that this book can be read and not used merely as a dry reference guide. I would suggest that you read the book before starting on a program as you may miss out on an idea which you could be using in a program.

Don't rush reading it through. I spend a lot of time 'dabbling around' with ideas, which is the only way to really get to know the machine and these routines. Play with them, alter them and tease them apart until you're really familiar with them. I find it all too easy to spend an hour happily toying around with ideas. If you're not really familiar with

machine code, then you won't understand its quirks; these will confront you and you won't know what's happening. It can be very depressing — a computer is a very strict teacher, and allows no mistakes.

Talking about mistakes, no machine code program is ever without them. This normally culminates in a hang-up. When this happens don't panic — just worry about your program. Often the RUN/STOP + RESTORE sequence will release you. Don't forget to make backup copies every half-hour or so and don't write a backup over the top of a previous backup — you will corrupt the old backup and acquire a write error with the new one. Don't forget to verify all your backups. It does take time with a tape system but it's worth it if you have a powercut. If the machine is still jammed, then a little routine in Appendix A (Crash Recovery) will pluck you out — have a look now.

At first, to make sure that the routines you need are in the book, you might try writing a game around a routine or two instead of writing the routines around a game. For instance, you might take the sprite homing routine (Chapter 10 — this homes sprites 1 to 7 on to the position of sprite 0) and write a chase game where the player controls sprite 0 with a joystick (also in Chapter 10) and has to perform certain tasks on the screen without colliding with a homing sprite. A program like this is easily written and will run quickly (especially if the routines are hooked on to the interrupt).

Interrupts

Interrupts are a major tool in the harnessing of the 64. You may not understand the term interrupt now, but by the end of the book you will. The interrupt is definitely the key to the machine. In the two routine chapters, many of the routines are specially designed to run on the interrupt, including a tune-player: when something is 'on the interrupt' it means that the 64 is automatically calling the routine for you. Thus you can have all eight sprites tearing around the screen while listing and editing the program with a flashing cursor (this will amaze your friends). All I can say is that, if you don't like using interrupts, then you won't capitalise on the 64. Some of the routines in Appendix B show you how to use the function keys properly to do some remarkable things.

Machine code

To finish off this introductory section, I'll just point out a few important facts. First, writing in machine code takes more time than BASIC. Programming starts off with great enthusiasm. However, after this there is a considerable period when you're not going to get results — the slump. Many would-be programmers give up at this stage, which results

in unfinished programs. Machine code programming is all about stamina, determination and confidence. Start off with simple routines and work up to the big ones.

Another important point is that there are an almost infinite number of ways of writing a program to perform a task. My routines are simply one way of doing this. Often you will be able to shorten them as they are designed to show you how they work and not to show off the 'clever tricks' that abound in machine code. The same idea goes for the process of games programming. Mine is simply a guideline and everybody develops their own particular style. Wait until you are confident and then develop your own style.

When writing your own routines keep them simple. If you look at the arrows on my routines, you will see they are few and are nested one inside the other. When branching is thrown in anywhere the bugs creep in. Keep it simple. Often badly-written routines arise because the programmer writes them at the keyboard. For the really simple ones this is alright but I have noticed that the screen itself doesn't help concentration. Most of my routines are sketched out on paper first and then transferred. As I have got into this habit I write my routines anywhere (anyway staring at a TV screen for too long is bad for you).

The question of assemblers must arise at this stage. While hand assembly is good for teaching the fundamentals, an assembler is vital for those longer routines. The best tactic is to get hold of a plug-in cartridge assembler. Personally I have several assemblers including a retired Acorn Atom which I sometimes use in parallel with my 64. If you must hand assemble then be wary of branch vectors (which can be tricky to calculate by hand). There is an assembler in Appendix B, at the back of the book, which doubles as a disassembler. Whether or not you have an assembler, the adressing mode representation in this book (shown in Table 2.2 in the next chapter) is probably not the one you are used to. The reason for the difference is an effort on my part to force you to comprehend the routines without blindly typing them in.

I say some pretty harsh words about BASIC in this book but this is not designed to stop you from using it. It remains the greatest computer language yet devised. Programs can be quickly and efficiently put together. Throughout the book I use BASIC to demonstrate some principles. Don't drop BASIC — it's great. If you're ready then start reading through the book. You'll find all sorts of useful information, so read on!

CHAPTER 2
Further 6510 Theory

Some of the methods used in this book may seem a little strange at first, and this chapter is designed to explain the theory behind these ideas. If you know about the workings of the 6510 and 6502 anyway, skip this chapter. Quite a few topics will be covered and it's imperative for effective programming that you are totally au fait with them. The best advice is to play around with these ideas until you can use and think of them instinctively. Mastery of anything is when you can do something without really having to think at all.

The ideas presented here are simply the ones I use. Don't just use them, try working with 32-bit or 64-bit numbers as opposed to 8 and 16 bits. If you stretch your abilities then success is guaranteed.

Addressing techniques

The 6510 processor has no less than 13 different addressing modes! Stop now and see how many you can remember. For every forgotten mode there is lost opportunity. Not only should you know them all but you should be able to decide instantly which is the best for the problem in hand. **Table 2.1** gives all the modes. How many do you know thoroughly?

The chances are you did pretty badly in that test. Returning to our earlier analogy, addressing modes are like tenses in French: you can get by on one or two but only just. Read on and discover the uses of the more obscure modes.

ABSOLUTE: Absolute mode is very important. It takes a 16-bit operand and uses that as the address. Many of the other modes are based on this one. In this book the absolute mode is denoted by just the operand, eg:

```
LDA 1024
```

which PEEKs the first byte of the screen. A close relative is the zero-page absolute mode which is identical except that it has only a one-byte operand which is a page-zero address (more about page-zero

Absolute	Two-byte operand used as a 16-bit address.
Immediate	One-byte operand used directly.
Implied	Single-byte function with one unique meaning.
Absolute X	Same as absolute, except X register added into the address.
Absolute Y	Uses the Y register instead.
Zero page	One-byte address giving 8-bit address in page zero.
Zero page X	X register added in again.
Zero page Y	Y register added in.
Accumulator	Single-byte instruction that affects accumulator.
Relative	Mode used by BRANCH instructions to ease program location.
Indexed indirect	Single-byte address into page zero to which the X register is added and the resulting address PEEKed to give a 16-bit address.
Indirect indexed	Single-byte address into page zero PEEKed to give 16-bit address. Y register then added in to index the address.
Indirect	Two-byte operand PEEKed to give 16-bit address. Used exclusively by the JMP instruction.

Table 2:1 6510 Addressing Modes.

later). A good assembler will, where required, automatically use this mode where required which is shorter and faster. Thus the code in zero-page absolute is shown the same way as normal absolute.

IMMEDIATE: Another old friend. The immediate mode always has one byte for the operand, the value of which is used directly. This is shown by the symbol '#' in this bok and is read 'hash'. This mode has no further 'indexed' extensions.

IMPLIED: These one-byte instructions perform some standard task. They are represented by just the relevant three-letter mnemonic.

RELATIVE: Branch instructions such as BEQ, BCC and BNE all use relative addressing. Irrespective of where they are encountered, they simply instruct the machine to jump to a new instruction by showing how far and in what direction this should be from the branch instruction. The beauty of this is that a program written totally in relative mode can be anywhere in the memory and run. A single JMP instruction within the code will mean that you will have to recalculate the jump address when you move. In the programs in this book I have used relative addressing everywhere. The routines supplied are 100 per cent portable.

The way to produce a relative branch even when you don't want a

condition to be true is to precede the branch instruction by the appropriate implied instruction:

```
CLC
BCC
or
SEC
BCS
```

Both of these will 'force' the branch. When you see this happening in the code, you'll understand what's going on. If you want to branch to a location that's out of range of the branch then you're probably doing some bad programming. If you can't avoid it then reverse the branch instruction (ie BCC becomes BCS and BPL becomes BMI) and place a JMP after it.

Look at this example:

```
BEQ LOOP
RTS
```

Suppose this gives an out-of-range error. The answer is the following code:

```
        BNE STOP
        JMP LOOP
:STOP   RTS
```

This is an 'extended' branch and should be used in preference to a leap-frogging technique, although locatability is lost.

At this stage it might be wise to discuss the ways in which variables and labels are presented in the book. Labels are preceded by a ':' sign and are written in a column well to the left of the code. Variables are simply short meaningful groups of letters that represent some value (eg LDA CHAR has CHAR as a variable). You could interpret this in two ways: either CHAR is the address of the value of CHAR and the instruction is thus absolute, or CHAR is the value itself making the instruction immediate. The choice is completely yours. For various reasons, I have used LLX for labels throughout the code as labels where X is any number. The routine is usually started by :LL0, and :ILL1 :ILL2, etc, appear in the order I used them when compiling the routine. As each routine uses the same labels you are forced to develop your own labels — an aid to success.

ZERO-PAGE THEORY: In this summary of addressing techniques, a

quick word on page-zero will not be out of place. Page-zero is the core of the computer's memory. It is the area which has modes devoted to it, and so these modes tend to be fast and effective. You will see that the 64 doesn't leave a lot of space for the programmer here but there's enough. The beauty of page-zero is that you only need a one-byte operand to address the 256 bytes in the page. The result is more space (this shouldn't be a consideration) more speed and versatility. If you look in the manual you'll find page-zero is full of all sorts of interesting things. Thus where possible use addresses in the range 2–255. Locations 0 and 1 contain ports and should be kept clear of.

ABSOLUTE X and Y: These are close relations of the absolute mode. A 16-bit address is given and to this is added the value of either the X or Y registers. This means that one instruction can produce 256 different addresses. This mode is used to produce very short fill and scroll routines amongst other things. The mode is represented by adding the suffix ',X' or ',Y' to the operand.

The instruction LDA 1024,X could PEEK any number from 1024 to 1279 merely by altering the X register. There is also a zero-page counterpart for this but only with the X register. (There is a Y register mode, but it's very limited.)

INDIRECT: This mode is used purely with the JMP instruction and is not used in the book, but it is very handy for picking up a vector. A 16-bit operand is used and this points to a pair of memory locations. These are then PEEKed and the address so obtained is jumped to. JMP (788) means look at 788 and 789, use their contents as a 16-bit address and jump to that address. The mode is signified here by enclosing the address in a pair of brackets.

ZERO-PAGE INDEXED MODES: These modes are useful, as the operative address can be changed to cover the entire memory of the 64. What happens is that the instruction points to a zero-page address. The value found at this address constitutes a 16-bit number which then becomes the operative address. On top of this, more flexibility is given by adding the values of the X and Y registers. There are two variations, where the value is added in before and where it is added in afterwards. One uses the X register and the other uses the Y.

INDEXED INDIRECT: This is the X register associated mode. The operand is a zero-page address to which the value of the X register is added (indexing). The resulting zero-page address is then PEEKed and the result of the PEEK supplies a 16-bit address (indirect). Look at **Figure 2.1** to see this in action.

Figure 2.1: Indexing and Indirection.

Indexing is when the X or Y register is added in.
Indirection is when an address is PEEKed to point to a new address.
Thus the difference between *indexed indirect* and *indirect indexed* is the order in which things are done.
Load accumulator at 100 in indexed indirect mode means:

Acc = PEEK(PEEK(100 + X)+PEEK(101 + X) * 256)

Load accumulator indirect indexed at 100 means:

Acc=PEEK(PEEK(100)+PEEK(101) * 256 + Y)

Each of the two modes is associated with a particular index register.

INDIRECT INDEXED: This is the Y register mode. The indirect PEEK is done first and the resulting address indexed by adding in the Y register. Both of these modes are very useful in manipulating tables of data as you will see in a later section.

This then completes our look at the possible addressing methods available.

Table 2.2: Assembly Language Representation.

No Hexadecimal numbers are used in this book. All the numbers are decimal. This table describes the representation of each mode in the book.

MODE	FORMAT
Implied	3-letter mnemonic only.
Accumulator	Suffix 'A', eg ASLA
Absolute	Mnemonic + operand (zero-page addressing is via default).
Absolute X or Y	Same as absolute but with ',X' or ',Y' appended.
Relative	
Relative	Mnemonic + label
Indirect indexed	Operand enclosed in brackets, ',Y' appended.
Indexed indirect	Operand + ',X' enclosed in brackets.
Immediate	Operand preceded by '#' sign.

These are slightly similar to the representation on the Atom.

The stack

The stack is the clever device which allows you to call JSR within JSR and eventually dig your way out with RTSs. This is not its only use. It

can also be used to pass parameters and store variables, but first let's see how it works.

The stack occupies page 1 of the 64's memory. A stack pointer maintains the position within the stack. Every time a JSR is encountered, the return address is stored on the stack, the stack pointer adds 2 and a JMP (effectively) is called to the subroutine address. On finding an RTS the stack pointer is lowered, an address is pulled off and a JMP is executed to this address. As the stack is 256 bytes long, it's possible to nest 128 consecutive JSR calls before overwriting the first return address: don't try to do this because all sorts of other things use the stack as well, and if it fills up you'll know all about it!

The instructions PHA and PLA mean PusH and PulL the Accumulator to and from the stack. Obviously if you do push something on, you'll have to wait until the stack pointer is in the right position to remove it. Manipulation of the stack pointer itself can be done with the aid of TSX (Transfer Stack pointer to X) and TXS which puts it back. If you are doing this, be careful not to alter any return address.

Using these ideas, we might store the value in the X register within a routine with no JSR, RTS or other stack operations between by using a TXA, PHA. Recovery is the opposite — PLA , TAX. Admittedly this is not very helpful but if we want to pass a parameter it can be a helpful way to send it. In the absence of a zero-page location, the stack provides a handy temporary storage facility. The processor status register can also be stored and removed from the stack by PHP and PLP. When an interrupt occurs, the stack is used to store all register values and the status register so that on return nothing seems to have happened.

Multi-bit operations

It's easy to add two 8-bit numbers. But as soon as we use more than eight bits, things begin to get a little more complicated. You probably know how to add 16-bit numbers: this section is to remind you and show you why things are done. The first thing we shall look at is addition. If you understand it now then write a 64-bit adding routine — this will test your knowledge.

The first thing to do is to clear the carry flag. This is because ADC means 'ADd with Carry'. A number is added into the accumulator and, if the carry flag is set, an extra 1 is added in as well (a smart way to add two numbers and increment the result is to use a SEC). Now if the last operation overflowed the accumulator, the carry flag is set to 1. When the next (more significant) bytes are added the carry is *not* cleared. If an overflow occurred, then an extra 1 must be added. The process is like adding decimal numbers: when a column exceeds nine you must carry a ten over into the next column where it appears as a one. Only the first

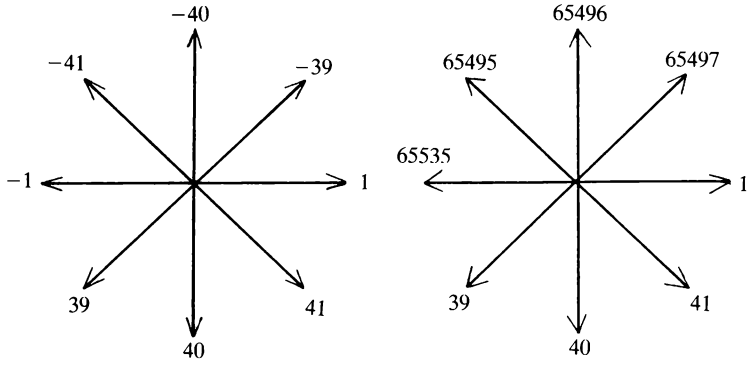
CLC is needed for the entire operation. The following code adds two 16-bit numbers found in 251–252 and 253–254:

```
LDA 251
CLC
ADC 253
STA 253
LDA 252
ADC 254
STA 254
```

Subtraction is performed in the same manner except that SEC is used to SEt the Carry beforehand. Addition and subtraction are vital in games as they allow us to move objects around the screen.

A *vector* is a number. In the manual you will see vectors which point to the start of ROM routines. In games terms, a vector is a number that is added into an address to simulate movement. When I speak of adding in a vector, I mean adding in a number to update a position. For example to move a sprite to the right you add in the vector 1 to its X coordinate. By definition, a vector has both magnitude and direction. Both of these can be contained within a number: a faster right movement vector would be to add 3 instead of 1. **Figure 2.2** shows the eight directions and the associated vectors you would use in BASIC. In machine code it would be a waste of time to have separate routines to add and subtract. This is overcome by using very large numbers to subtract. If you add in 65,535 then in 16 bits you have effectively subtracted 1.

Figure 2.2: Subtraction by Addition.



The second part of the figure should now make more sense. Instead of negative numbers, we add large numbers. They are in fact the 'two's

complement' of the negative numbers. This idea of using large numbers to perform subtraction is extensively used and so you should ensure that you fully understand it.

Comparing two numbers for similarity or difference is simple. You just use CMP (or CPX or CPY) and use BEQ (similar) or BNE (different). Working in 16-bits, a little more must be done. Often you must check to see if an address has been reached (or a missile has hit a ship) in which case a 16-bit comparison is needed. The process is simple: compare the two lower bits and the two higher bits separately. If the two lower bits don't match then the numbers must be different. If they do match then test the higher two bytes. Either they match (both numbers are the same) or they don't. The routine does this:

```
LDA 251
CMP 253
BNE UNEQUAL
LDA 252
CMP 254
BNE UNEQUAL
**Numbers are equal**
```

The two numbers are stored in 251–252 and 253–254. UNEQUAL is simply a label for the sake of demonstration.

These 16-bit operations are very important. They form the core of writing a game in machine code; without them you cannot move objects around with any degree of ease.

Logical operations

The AND/OR/EOR functions always seem a little strange as they have no apparent use. In fact they make the manipulation of individual bits within a byte very easy indeed. Often you will want to set just a bit in a byte or clear a few bits somewhere else without disturbing the rest. The key to this is the use of the logical operators. Each does its own little thing and **Figure 2.3** will refresh your memory as to their function.

AND: The result of AND is 1 only, and only if both the first input AND the second input are 1. Thus 1101 AND 1100 = 1100. The AND function is a *mask* operation. It enables you to mask out unwanted bits while retaining the important parts. Suppose we were interested in the lower three bits of an 8-bit number. To obtain these bits only, we mask with 1 + 2 + 4 = 7. Only where a 1 is present in the mask can a 1 show through. By this method we can do all sorts of things: checking whether a number is even or odd, or decoding a byte that contains more than one

Figure 2.3: AND-OR-EOR Truth Tables.

AND

IN	IN	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Mask

OR

IN	IN	OUT
0	0	0
0	1	1
1	0	1
1	1	1

Set

EOR

IN	IN	OUT
0	0	0
0	1	1
1	0	1
1	1	0

Toggle

piece of information, the AND function acts as a sieve through which only what you want drops through.

OR: The result of OR is 1 if either the first OR the second input is 1. This function performs the SET operation. If you want a bit set then you simply OR it with the relevant number. ORing with 0 leaves the number unchanged while ORing with 255 turns it all into ones. In the same respect the AND function is a CLEAR operation. Suppose we wish to set bit 4 of the number 10101010 then we simply OR with 10000:

```
10101010
00010000
--------
10111010
```

When POKEing important locations such as 0 and 1, it's vital to change only the bit that you want. Failing this, the machine will almost certainly crash.

EOR: Just the same as the OR function except that when both inputs are 1, the output is zero. EOR is a toggling function: if you continually EOR with a number, then the corresponding bits will go on and off and on again. This means that performing EOR twice leaves the number unchanged. One use is for toggling bit 7 of a character on the screen. This switches it to and fro between inverse video (flashing it). It can also be used to produce the two's complement of a number — EOR with all ones and then add 1.

Taking stock of these operations then we have a way of setting any bit, clearing any bit and toggling any bit. Consider the following application of controlling the tape motor:

```
LDA 1
ORA #48
STA 1
```
turns it off

This will have to be repeatedly done to hold the motor off.

Scanning the screen

As the book is designed for games purposes, a lot of the routines directly affect the screen. This means that they look at the screen and, depending on various instructions, alter its content. Thus the process of scanning the screen in machine code is worth thinking about. Normally the screen is scanned from top to bottom. But for some operations a reverse scan is needed (which is slower). The process of scanning the screen uses 16-bit theory, so make sure you understand the principles.

The normal scan is performed as follows. The address of the top lefthand corner is stored in a zero-page location (two consecutive bytes). A 16-bit increment is then used to increment the number and a 16-bit compare to see if the process is complete. The function of the routine is inserted into the loop and, as the address that we are working on is in page-zero, it's easy to interrogate the screen for information. The initial address is 1024 : 0,4 in the lo-hi format.

```
LDA #0
STA 251
LDA #4
STA 252
```

The increment is done with the aid of the INC instruction.

```
        INC 251
      ,BNE LLO
      ( INC 252
:LLO  ↘—
```

With the 16-bit compare, it doesn't matter whether we test the hi or lo-byte first. It's quicker though to test the lo-byte, as this will only be 'correct' on a few occasions. Another point to bear in mind is that we are testing the number immediately after incrementing it. This means that we must test for the first square *off* the screen as otherwise the bottom righthand corner will get neglected.

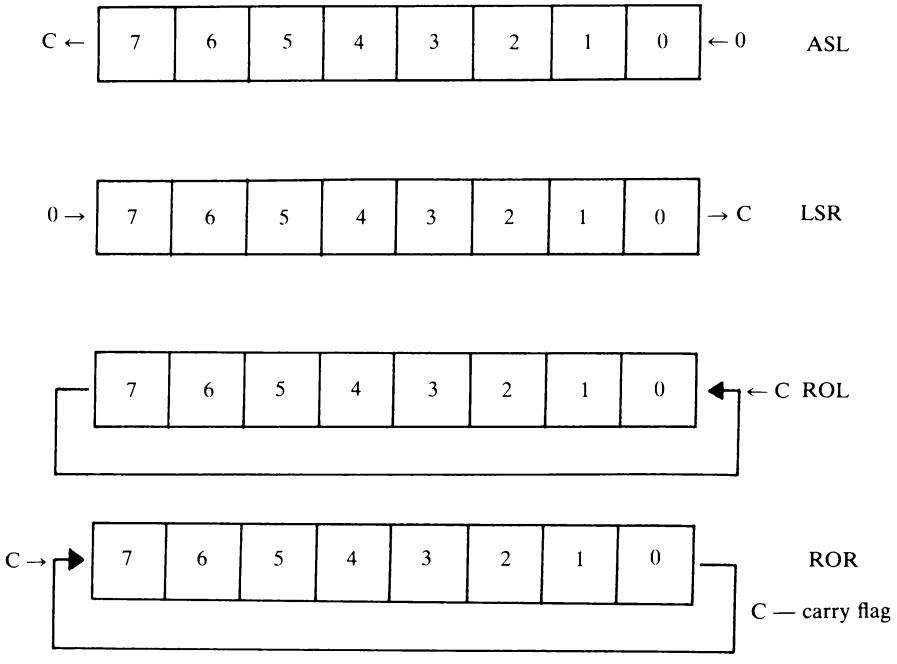
```
LDA 251
CMP #232
BNE [loop around again]
LDA 252
CMP #7
BNE [loop around again]
RTS
```

If we had incremented the number (used it to do something on the screen and then tested it), then we would test for the final address on the screen. You must look out for and be aware of these little points that recur in machine code programming.

Multiplication and shifts

Frequently multiplication has to take place in machine code. A typical problem might be to multiply a byte by 65. Where the multiplicand remains constant the problem is easily solved. Where the multiplier varies the topic goes beyond the scope of this book, although the principles remain the same. To multiply by, say, 65, split the number up into powers of 2 (64 and 1). Multiplying by 2 is easy so multiplying by a power of 2 is also easy.

Figure 2.4: Shifts and Rotates.



The shift instructions allow us to multiply and divide the numbers in a byte. **Figure 2.4** shows what happens when they are used. The ASL instruction is particularly handy for multiplying by 2. If you are doing a lot of this, then don't forget to catch the carry when it comes through. **Figure 2.5** shows how to do this.

The result of a*b is returned in T. First of all, copies of a and b are made to preserve their values. The multiplication is then performed. The first bit of X is tested by using the AND function. If it's a 1 then the value of Y is added into T. Then X is divided by 2 (shifted to the right) and Y multiplied by 2 (shifted to the left). If X = 0 then all the bits in X have been dealt with and the task is completed. If X<>0 then the first bit is again tested but it is not the same bit as before.

To do this with 16-bit precision demands 16-bit shifts both left and right, a 16-bit add and a 16-bit compare. In fact, it's not that hard at all but it's unlikely you'll want a multiply routine of this sort anyway. What Figure 2.5 does show though is the mechanics of a multiply. Suppose we want to multiply by 40. Let a = 40 then the only bits set in X will be 32 and 8. Thus we add together a × 2 three times (× 8) and b × 2 five times (× 32). Multiplying by 2 is done by a shift instruction; ASL and ROL can be used. Use ROL to perform a multi-bit shift as follows:

First clear carry	CLC
Next shift the first byte	ROL ADDR *or* ASL ADDR
Now shift the successive bytes	ROL ADDR+1
	ROL ADDR+2
	etc

Use ROR to divide by two in a multi-bit shift as follows:

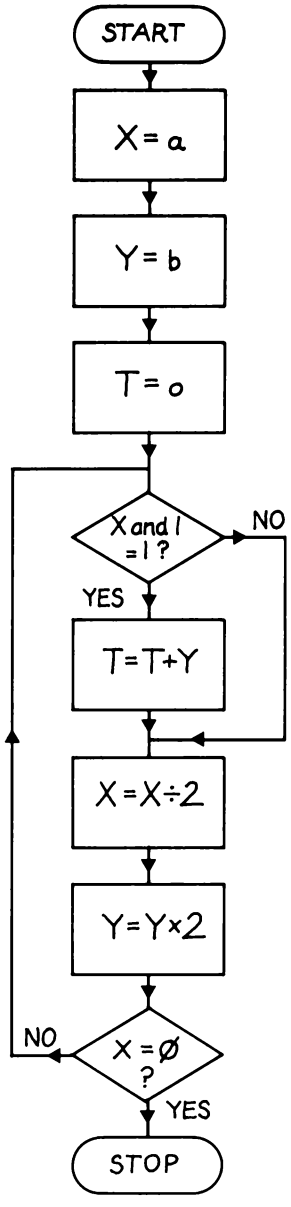
First clear carry	CLC
Next shift last byte	ROR ADDR+2 *or* LSR ADDR+2
Now shift successive bytes	ROR ADDR+1
	ROR ADDR

Data structures and tables

The simplest of data structures in BASIC is the single variable. This can easily be handled in machine code but, as soon as arrays are needed to control fleets of characters, problems arise. The use of this sort of array is demonstrated in the section on Fleet Movements (Chapter 10). An array is only a list of simple variables. The novice tends to shy away from them but in fact they are remarkably simple concepts.

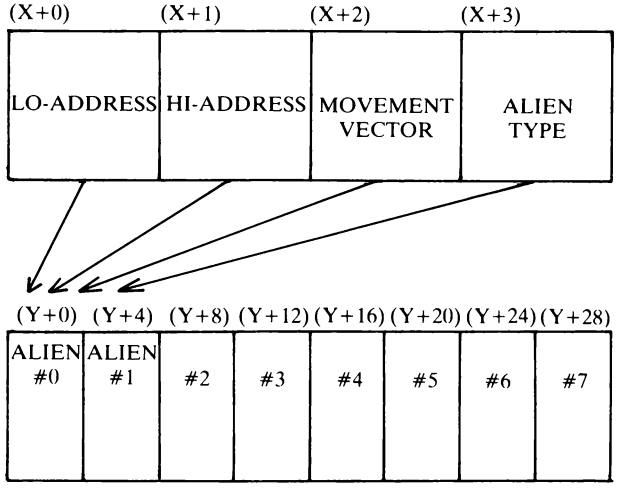
The first stage is to work out the dimensions of the array. What

Figure 2.5: Perform T = a*b.



information do you want held for each object? Some information, such as an address, will need two bytes whereas others will need only one. Let's suppose we want a table of 50 aliens containing an address, a movement vector and an alien type. This might be represented by the structure illustrated **Figure 2.6**. Each entry consists of four bytes. Next we decide where to site the table — find a suitable area of memory. Store the start of the table as a two-byte vector somewhere else (page-zero if possible) and store the number of objects (ie entries) somewhere else. To fill the table up with suitable data is a task for a COPY routine. This means that you can call up attack wave#1, wave#2, etc, with ease by simply instructing the copy to fetch the data from different areas of memory.

Figure 2.6: A Typical Table Structure.



Once we have defined the structure, manipulation is easy. Suppose the base of the table is held at 251 and 252. Then by:

```
LDY #0
LDA (251),Y
```

we can obtain the first byte of the table. If the structure is as above, then this is the lo-byte of an object's address. By using INY successive bytes can be accessed. If the overall length of the table is less than 256 then scanning is performed via the Y register. If it's larger then the best plan is just to add numbers to the address at 251 and 252 and use the Y register for reading each entry. To give an example let's suppose the objects in the tables run 0–49 and we want to find the type of alien number 12. As each object consists of four bytes, the object we are

interested in starts at Table + 12 * 4. Suppose 12 is in the accumulator. Two ASLAs multiply by 4 and TAY moves the value into Y. The alien type is the fourth (and final) byte of the object. Thus we follow with three INYs. A LDA(251),Y then supplies the type of object #12.

```
Get type of object in accumulator
ASLA
ASLA
INY
INY
INY
LDA (251),Y
RTS
```

As you can see, tables are very easy to use. You will find them in use in several of the routines in this book, including the screen-flash attribute routine (Inverting and Explosions, Chapter 9) where a table of 25 elements is used to determine whether or not to flash that line.

CHAPTER 3
The Birth of a Game

So you have decided to write a machine code game. No doubt you already have a rough idea of the game that you want, whether it is just another variation on the Space Invader theme or a totally new idea of your own. Whatever the case, don't rush through this first stage. Before you even think about moving on to the next part, put your thoughts down on paper in a neat and ordered fashion. There is no way that you are going to design a game (in machine code) if the basic idea varies from day to day following your whims. In short, make up your mind while you can. Once programming is under way, alterations will be costly both in terms of time and morale.

Speaking of morale, this is one of the most important factors when programming in machine code. There will be long periods of intense work with little reward. It is a real test of stamina and self-confidence. For this reason, it would be unwise for the beginner to attempt to write a game totally in machine code. Simply use machine code to speed up any slow sections of BASIC in a game. You may find that one or two routines are all that's necessary to considerably speed up a game that you have already written. (Indeed, one popular way of translating an idea into machine code is first to write a BASIC program to do the job and then to convert it stage by stage into code. I don't like this method if it is used to produce a pure machine code program as this results in rather badly written, difficult to edit and inefficient programs. It can, however, be a successful method for use on a small scale within a program.)

For the beginner, the best approach is to write a BASIC program leaving out the sections to be performed by machine code. The reason for this is that BASIC and machine code are two totally different languages. You can't attempt a literal translation. You need to go back to the problem in hand to write efficiently.

A classic example of this is the attempt to program a computer to translate one (human) language into another. As an experiment, the computer was told to translate a section of the Bible into one language, and then translate this into another. Finally this was translated into yet another language. The initial text read 'The spirit is willing but the flesh is weak.' The final product read, 'The liquor is good but the meat is

poor.' This serves to show the dangers of attempting a literal translation instead of considering the main issue.

How to write good games

This section is devoted to the design of a game that is both challenging and addictive. It is possible to brood for hours over various ideas and concepts without arriving at a satisfactory answer.

A careful study of the popular arcade games reveals various principles on which these games are based. All of the top games display these qualities to a certain extent, and I am certain that it is possible to 'work backwards' from these ideas to produce a game that is a winner.

The first factor is simplicity. Most arcade games are based round very basic ideas. This allows the novice a certain degree of success and eliminates the need to constantly refer back to the rules. The idea can be anything from collecting apples to climbing a mountain. Recent ideas have become very silly with titles such as *Attack of the Mutant Hamburgers*, but if it's fun then it works.

But there is a new genre on the rise, of which *Defender* is one. This is a complex game (as can be judged by the number of controls) but is very satisfying once mastered. However, it should be obvious that a complex game will be more difficult to write than a simple one, so do not attempt something of this calibre until you are sure of yourself.

The next factor of design is variety. Once again we have two types of games to discuss — the old and the new. Old favourites supply endless variation in that there are many ways to play them: in *Space Invaders*, for example, the invaders can be picked off in any order you like, each order resulting in a different 'end game'. The new genre of games provides variation via a series of different screens or 'attack waves'. Each screen provides a new and fresh challenge and there are often several ways of performing the task. By combining almost infinite variety with simplicity, a very sound game emerges which can be picked up quickly but which also maintains interest.

A third factor is the introduction, from time to time, of an element which adds a lot to the game (such as a mother-ship or fruit in a pac-man maze). Normally these offer a quick boost to the player's score at the expense of added danger. The arrival of a baiter in *Defender* adds to the excitement and helps keep interest in a finely executed game. These are often the stumbling blocks for players who can just about handle a game until a new element arrives to finish them off. In BASIC, this third factor is often neglected due to the severe loss of speed in order to control the action.

Undoubtedly, a gradual increase in difficulty is a prime feature in a game and this is the fourth factor. But note that difficulty does not make

the game: it simply serves as a barrier to the player and stimulates the will to player advancement.

An increase in difficulty can be achieved in two (sensible) ways. If the speed is gently raised, the player is forced to think faster and work with more coordination, resulting in the inevitable error. However, there is a limit to the degree of speed that can be introduced without resulting in a mere game of chance. If the enemy is made to become more intelligent as the game proceeds, the player is faced with increasing problems. This is sometimes achieved by altering the motion of the enemy from a somewhat random one to a more ordered fashion. By linking an increase in speed with this mutation of the enemy, there arises a natural score-barrier which the player cannot pass without practice. An attempt to effect this in BASIC usually fails miserably: an increase in speed is out of the question as the chances are that the game is already running at full speed, and increasing the intelligence of the enemy is out of the question as this would slow the game down even more. This results in a silly method being used (eg tripling the number of bombs being dropped), which forces the player into defeat. The only sensible course is to use machine code.

These four points (simplicity/variety/extra factor/difficulty) are the essential ingredients of a good game. To think up a winning idea you must first come up with an underlying objective, but keep it simple. Next think up a way of hindering the player and again keep it simple. Then introduce variety into the game by giving the player a range of different enemies and multi-solution tasks. Once this has been done, you have the foundation of the game. All that's left now is to introduce the third factor from time to time and produce a way of gradually increasing the difficulty of the game. Despite all I have said, game design demands imagination and an open mind. Ideas will often come unprompted, so be ready for them! Once you have a good idea don't waste it. Give it the treatment that it deserves and develop it into a fully-fledged arcade game.

Developing the idea

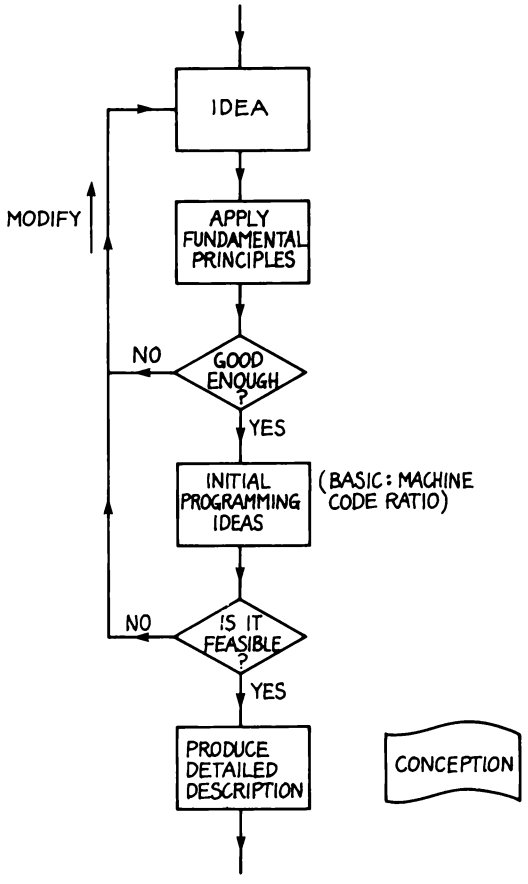
Once you have established the exact function of the game, you are nearly ready to move on to the next stage of program design. But first give a little thought to how you are going to achieve your aims. Be realistic. Although the graphics of the 64 are outstanding, it doesn't mean that it's easy to write a game with a lot of on-screen action. In fact it's more complicated than usual, because of the complexity of setting up the graphics, and because programmers tend to start off with too ambitious ideas. All that is really necessary at this stage is to think about how one thing is to be done in relation to another. For example, you

may decide that one class of enemy will be represented by UDGs while another class is shown using the sprite graphics facility.

This should help you identify possible areas for machine code intervention. It's unlikely that you will write games 100 per cent in machine code until you have had a fair bit of experience. You can use BASIC to initialise a game and give the player instructions: perhaps only a small part of the game will be assisted by machine code. The degree of jumping between the two will ultimately determine the time it takes you to complete the game.

A common method of producing above-average games is to write two versions of the game. The first one is the prototype and is made as good as is technically possible. Then the programmer spends time rethinking it, in an effort to further enhance the initial idea. These thoughts are reflected in the second and final version. This method, however, does

Figure 3.1: Conception.



have the grave disadvantage that it takes longer than ever to write the program. This is often avoided by writing the first program totally in BASIC and ignoring the lack of speed. At this stage (which can take only a day or two) both the idea and the techniques involved are developed so that the final product is as near perfect as possible. For the absolute beginner I would advise that he (or she) refrain from this method as it really is boring and unrewarding for the novice.

Once you think you have developed your idea in sufficient detail, make sure that it is not ambiguous and (more importantly) that it is presented in a tidy format for later reference. Make sure that you have achieved what you want. In the near future you are going to put in a fair bit of work on this program so be sure that you are going to do what you want to do and not what you allowed yourself to be roped into. Machine code is really a matter of stamina and achievement. If you're not really interested in what you are doing, then there's no way that you will finish, or even look like finishing, the game. You must be totally *au fait* with the game all the way through to ensure its survival. If you are confident that this stage has been satisfactorily completed then move on to the next section, but first have a break to allow the idea to fully mature in your mind. Whatever you do, don't rush it.

CHAPTER 4
Program Design

Once the detailed description of the game has been completed, the next stage is to draw up the charts that will divide the game into a series of minor tasks. You may well ask how this job is to be done, but unless it is, the game will never be completed.

In BASIC it's possible to write programs straight off the top of your head. In machine code the situation is different — for a lengthy program, there is no way round this stage of detailed specification. Indeed, even if only a section of the program is written in machine code, I would still advise you to stick with this stage. Once you get down to it, you will find it surprisingly simple.

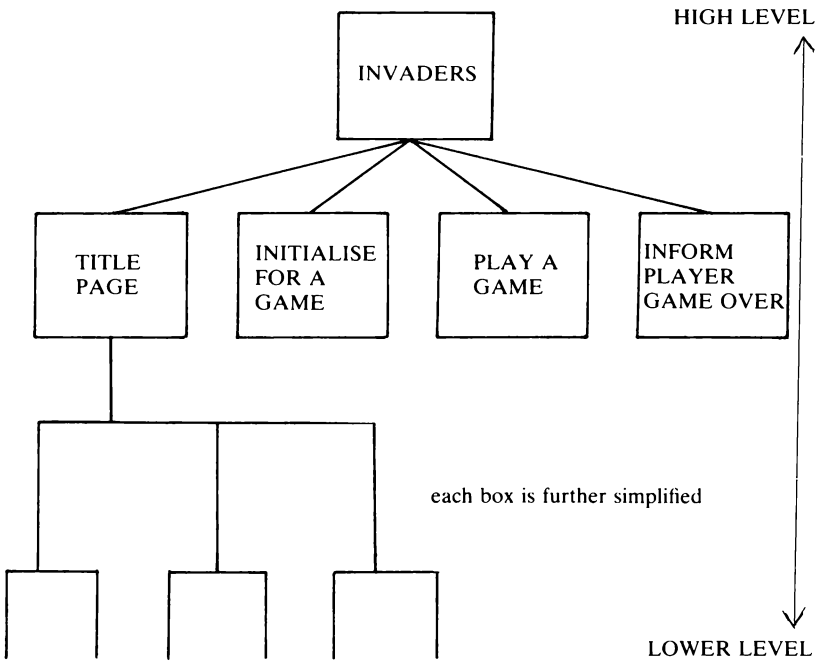
There are essentially two ways of describing a process with a visual representation: the flowchart and the top-down chart. Each has its virtues, although the use of a flowchart is often considered bad programming by some institutions. My system is first to take a clean sheet of paper and write in a box at the top of the page, in the middle, the title of the program. This is 'very high level': what I mean by this is that I (the human) understand it but the computer hasn't got the faintest idea of what the task is. The next stage is to take this box and divide it into a series of tasks not more in number than, say, four or five. Let us take for example the program *Space Invaders*.

I would divide this into the following series of tasks:

Display title page.
Initialise for a game.
Play a game.
Inform player that game is over.

These would then be fitted into a series of boxes *one* row down from the first box and connected as in **Figure 4.1**. Notice that the level of the description has dropped, in that it is more 'machine-orientated' than the first box. Subsequent stages simply mean taking each of these boxes and dividing and sub-dividing it further still, until the task of encoding it is little more than a straightforward exercise in coding. The final result will resemble the roots of a tree. At each stage a root splits into several other roots.

Figure 4.1: The Top-down Chart.



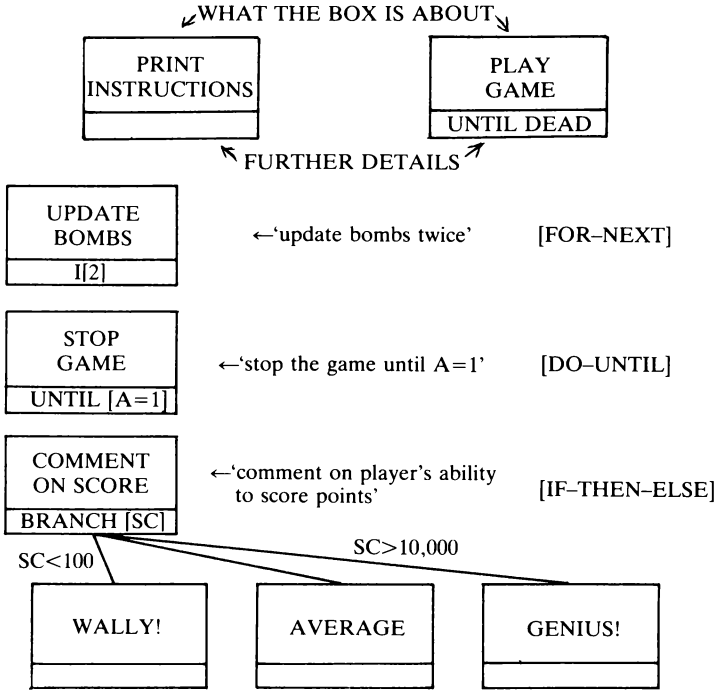
After a little practice you will know when the tree has the required amount of detail. Obviously there is no need to take things as far as the actual instructions themselves: I would stop the process when tasks became reduced to things like 'scroll screen left', as I know I am competent enough to write this in machine code. (This is where this book comes in: it will show you how to perform such tasks as scrolling the screen. Of course it cannot cover every eventuality and so I have included a chapter on 'algorithm design' — Chapter 6 — which is designed to stimulate your mind to devise methods of solving particular tasks.)

Now that we have obtained a basic insight into the process of program design, we can begin to expand on some ideas of designing top-down charts. If the program is to be at all complex, then the top-down chart will cover many sheets of paper. This means two things — we must be very neat and organised and we must have a flawless system for connecting one box to another. If the system is not flawless then the tree will become deformed. The other point is, how do we represent loops and branches in a top-down chart?

I'll discuss the second point first. It doesn't matter how you denote loops and branches in your top-down charts as long as you are consistent. Gone are the days (at least with micros) when one man

would describe the program and another would write it. You and you alone are writing this program. As long as you know what you mean then all is well. You will probably be aware of the notation used when using flowcharts (diamond box for decisions, rectangular box for processes, etc). A top-down notation is not normally shown by the shape of the boxes but by signs inside each box. I will show you my notation (which I didn't develop) but feel free to develop your own if you think that it's clearer. If you are doing this, then it might be a good idea to write down on a separate piece of paper exactly how your system works so that you don't deviate from it.

Figure 4.2: Top-down Schematics.

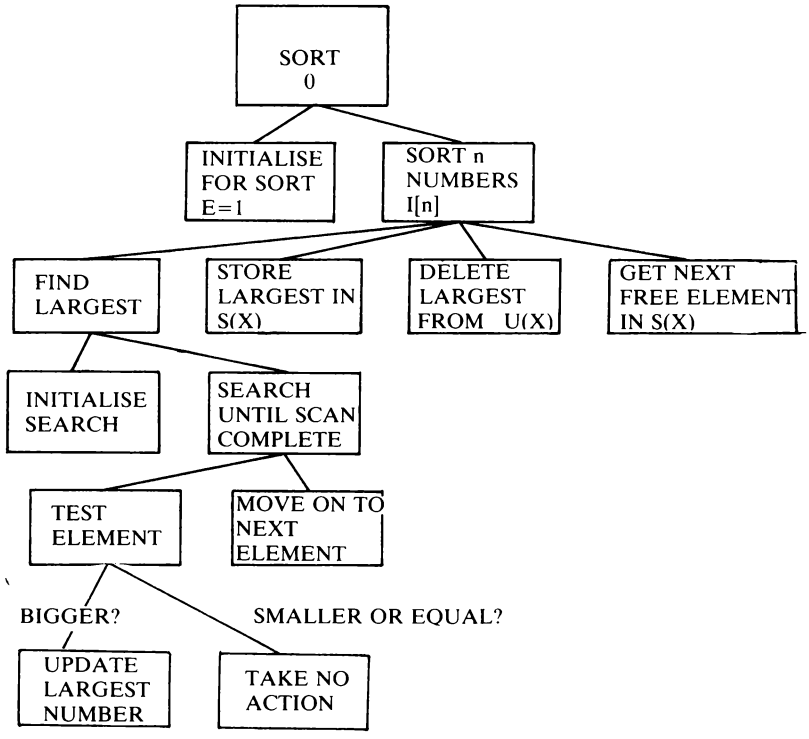


In my system, every box is divided up as shown in **Figure 4.2**. The top part contains the title and function of the box while the bottom part contains details of the way in which the 'box' will be executed. There are several distinct possibilities for this section. You may wish just one pass to be made. In this case I would normally leave the box empty or put in it the Greek symbol for nothing, '∅'. We may want a fixed number of passes: this is similar to the FOR–NEXT loop in BASIC. I show this by writing 'I[n]' meaning 'iterate n times'. You may wish the box to be repeatedly executed until some certain completion conditions arise: this

is similar to the DO–UNTIL instruction found in some BASICs. This is denoted by 'UNTIL [X]', where 'X' stands for a condition (eg SCORE = 1500). For the occasions where several possible events may occur, depending on some other event, I utilise the notation 'BRANCH ON [X]', where X is once again a condition. When this is in use, I write along the connecting line the value of the condition that will force the machine to take this path. Figure 4.2 should make all this clear to you.

We are now in a position to see how we can integrate all these ideas, so that we can draw a top-down chart for anything we like. Take the simple example of sorting a list of n numbers contained in the array U(X) into the array S(X). Let's use the following algorithm:

Figure 4.3: An Example Top-down Chart.



(1) Find the largest number in U(X).
(2) Store this value in the first empty element of S(X).
(3) Replace this number in U(X) with a very small number (effectively deleting it from U(X)).
(4) Repeat steps 1 through to 3 until all the elements in U(X) are very small.

This can be described in top-down form as shown in **Figure 4.3**. Bear in mind that we are demonstrating the principle of top-down design and therefore this example will be illustrated in BASIC. Before I explain it to you, try and see if you can understand how it works. The actual concept of sorting a series of numbers is quite complex, yet the use of a top-down chart breaks it down into easily-managed chunks.

Looking at Figure 4.3, the chart starts of with 'sort numbers'. Well this is a logical way to start. The '0' in the box is not really necessary, it's just there to show you what the box means. This task is divided into two: getting ready for the job and then actually doing it. The 'E = 1' in the left box means just that — assign the value 1 to variable E. This is clearly not a directive. The righthand box contains our iteration symbol 'I[n]'. A glance at the algorithm will remind you that we must perform the process of picking out the largest number and recording it as many times as there are numbers in the list. The symbol is telling us to repeat execution of its 'constituent' boxes n times.

With the next generation of boxes, it is impossible to guess from only one of them what the purpose of the whole chart is. This is a clear indication that our simplification of the problem is working. The four boxes comprise the four steps that must be repeated for every largest number found. Of the four boxes, it is not absolutely clear how we might program just one of them, so we further simplify this box only. The workings of this branch should be clear to you: notice the conditional branch box with suitable annotations along each arm.

Now we are in a position to program this chart. While this will be more objectively discussed later on, a short example will be given now. For simplicity we will program in BASIC. The principle involved here is known as 'tree-walking'. This means starting at the top and working down the leftmost arm until we arrive at a dead end (or 'leaf'). We then turn around, go back up and take the next route available at the next junction that hasn't yet been explored. In this way it's possible to walk the entire tree.

Starting at the top, we encounter first 'sort numbers', so we write the first line:

```
10      REM *** SORT NUMBERS ***
```

Next we come across E = 1, so this is naturally the second line.

```
20    LET E = 1
```

The next box is 'sort n numbers' so we make the third line:

```
30    GOSUB 1000
```

meaning that the subroutine starting at line 1000 will perform this function. It's only right that we should make this clear and so line 1000 becomes:

```
1000  REM *** SORT N NUMBERS ***
```

Travelling further along the tree, we arrive at 'find largest'. This itself diversifies and so we write:

```
1010  GOSUB 2000
```

followed by

```
2000  REM *** FIND LARGEST ***
```

This takes us to 'initialise search' — a leaf at last! This gives:

```
2010  LET L = 1
```

Next on our travels comes 'search' and note the directive to continue until L = 101. As this is not a leaf, we write:

```
2020  GOSUB 3000
```

followed by

```
3000  REM *** SEARCH ***
```

Now we get to 'test', which is a conditional branch. Taking each box in turn, we write:

```
3010  IF N(L) > LARGEST THEN LET LARGEST = N(L): LET P
      = 1
```

As nothing happens in the next box, we may as well ignore it. The walk now takes us to 'next L' which is a leaf so we write:

```
3020  LET L=L+1
```

This is the last box in the simplification of 'search' and so we can write:

```
3030  RETURN
```

We are now back at line 2020 where we placed a GOSUB. However, there was a directive telling us to repeat this operation until L = 101 so we write:

```
2030  IF L<>101 THEN GOTO 2020
```

followed by

```
3040  RETURN
```

We now walk back up the tree and finish off the 'find largest' subroutine by writing

```
2030  RETURN
```

We are back in the line 1000 subroutine and we see that we have three leaves so we write:

```
1020  S(E) = LARGEST
1030  N(P) = -9999
1040  E = E+1 (I have now dropped the optional 'LET')
```

Once again this is the final leaf of a box, so we finish off this box with

```
1050  RETURN
```

We now return to the first few lines we wrote and add

```
40  END
```

to halt the computer.

The program is now complete and ready for testing. You should have noticed how the top-down technique virtually wrote the program for us. All that was required was a little light thought, almost therapeutic!

It should now be apparent that a full program will have a very large top-down chart. This will be very difficult to draw on a single sheet of paper. The answer is to use a number of fairly large sheets (say A4) and to give each sheet an identification number: this might be two letters

from the game's name plus the number of the sheet, which will help you identify which game the chart belongs to. Thus for, say, Space Invaders we might have SI1, SI2, SI3, and so on. The reference number of the chart should be clearly written in a box at the top of the page. Make an index of these reference numbers and the purpose of the chart on a different sheet of paper, which will allow quick and easy cross referencing. To identify each individual box on the page, use the page code preceded by the letter 'B' (for box) with yet another code tagged on to distinguish between all the other boxes on the page. If you're really keen, then make up *another* index page with all these codes on it. All this paperwork will assist greatly in later stages of writing.

Once we have compiled a top-down chart, we are nearly in a position to start coding: we can see exactly what is required and where. Moreover, by looking carefully at the chart we can locate repetitions in the code. These are obviously ideal candidates for turning into subroutines. Yet more is done for us by the chart: it shows us how to connect each of the parts and also the natural way to test each section as it becomes available for integration into the rest of the program.

The efficient programmer should plan out all the work in a timetable so that he can see exactly where he is (and whether he is on schedule). Always allow yourself lots of time for each individual stage as machine code is never written without the arrival of bugs. A proficient BASIC programmer can write a full 10K program and have it in total working order in two days, but a machine code programmer will be pushed to write 1K of usable code in a day.

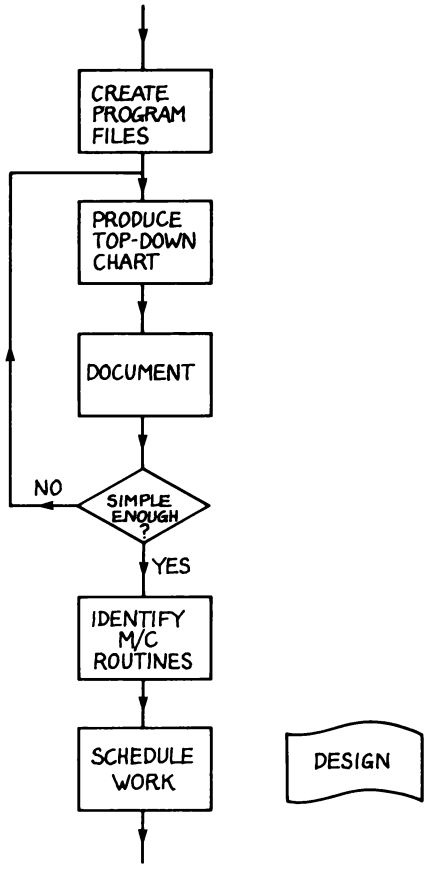
The stages in producing a schedule vary from programmer to programmer. Some programmers don't need to use too much organisation, but for a program in depth this can be mentally taxing. The first stage is to look through the chart and identify all the routines. Then, depending on how much of the program is to be in machine code, single out all the machine code routines. Compile a list of all these routines, including references to the chart. Now work through the chart, estimating how long each routine will take to code and test (this comes with experience) and, from this, organise your timetable. Obviously it is good to get into a habit of following the schedule tightly, so be generous with the estimates. You will have plenty of time to become a 'speed-merchant' later on.

Before moving on to the next chapter, try writing a few BASIC programs with the top-down method so that you have a chance to really appreciate the full value of the technique. This is what is known as 'structured' programming — the chart shows the program's structure. For efficiency and clarity I use a programmer's template with all the flowcharting symbols available, plus a pencil to rough out ideas. Once I have produced the prototype chart, it is transferred on to a much

smarter piece of paper. Using a template means that you get uniform boxes with square sides. You can either buy one or cut one out of cardboard.

As a final note to this chapter, bear in mind that in drawing up this chart you are laying the foundations of the game. It's well worth the time to build them securely, so take your time and double-check the tree when it's ready.

Figure 4.4: Design.



CHAPTER 5
Coding the Program

By this stage, you will have finished the groundwork and we can at last get on with the actual writing of the program itself. Taking stock of what has already been done, we have a detailed description of the program, and we have also compiled the top-down charts, which tell us exactly how to write the program. Furthermore we know which routines are to be written in machine code and so are ready to get on with writing them.

A major part of this book is taken up with a 'common routine' section (Chapters 9 and 10), which is a large selection of some of the common routines you are likely to want to use. These range from filling areas of memory with a code to scrolling the screen left and right (and other directions). Each routine is discussed in terms of subject matter and style, so that they are readily adaptable. This chapter shows you how to use these routines to the maximum, with the minimum of work. But don't think that the book will write the game for you: there will be some routines that you must construct yourself, and Chapter 6 is devoted to this aspect.

Those of you who are alert may have noticed that there are going to be two fundamentally different types of routine in our program. First, there is the purely functional routine which appears at every leaf on the top-down chart: this routine actually performs some noticeable task. The second type of routine structures the program: it links individual routines together and performs the inter-routine looping operations. The latter make up each node (ie junction) of the tree. The writing of these connecting routines is discussed in Chapter 8: the present chapter concerns itself with the purely functional routine only.

It is assumed that you already have a fair knowledge of machine code in that you understand the manipulation of the registers and how to perform the simple addition and subtraction operations which books on 6510 and 6502 code pride themselves on. As to actually writing something worthwhile, these books (with some exceptions) tend to leave you in the dark a little. This is because it's very difficult to be objective in machine code. The language lends itself very well to the construction of short, quick, repetitive tasks, whereas the writing of a complex program is a daunting task indeed. By the introduction of the

top-down chart, however, we have reduced our task into a series of very simple sub-tasks that lend themselves well to an objective style of programming. If this is not the case with your charts, then you simply haven't extended the tree in enough detail yet.

A vital instruction in machine code is the JSR/RTS combination. These two instructions give you the capability to call subroutines just as in BASIC. The only difference is that we can use a far higher level of nesting than in BASIC. The 6510 allows us to use up to 128 JSR calls without using a single RTS. This gives us the ability to call up to 127 subroutines within one subroutine. As a sneak preview of the next chapter, it would make sense to say that the 'connecting routines' work by calling each of their subsidiary routines with JSRs. If you are not totally clear on the topic of the JSR, go back and read up the relevant sections in your manual.

Parameter passing

The subject of parameter passing is a vital concept. Parameters are the pieces of information that a routine requires to do its job. The parameters of a PRINT statement in BASIC are the variables and any text that you wish to have printed. Consequently in machine code various routines will need some parameters. Some, however, won't need any. To give an example, consider a routine that fills a section of memory with a certain character. To perform this function the routine must know where to start and stop filling, and also what character to fill with. In BASIC there is only one way of parameter passing and that is to assign the values to be passed into a variable which the routine itself must interrogate. In machine code there are two distinct methods. Either the values required can be dumped into some pre-determined memory locations so that the routine can pick them up, or the data can be put on to the stack. For our purposes, we will use the first method as it is easier to implement.

To demonstrate this method, let's suppose we are going to use a Fill routine and so must pass two addresses and one code to the routine. As long as the parameters are sent and received in the same fashion, no problems will arise. We can store the two addresses in locations 251–252 and 253–254 as a pair of 16-digit numbers. As the code with which to fill the area is only a single byte we could send it in the X register (no particular choice, it could as easily have been the Y register). As long as the Fill routine knew about this, then there would be no problems. An important fact arises from this. If a routine is to be used with only one set of parameters throughout the program, then it would make sense to forget passing them at all and simply initialise them as part of the routine itself. It is up to you to spot when this sort of situation arises and

to act accordingly.

The same sort of reasoning is true for passing parameters back from the routine to the calling routine and then on into the next routine. This is a topic which will be discussed later in this chapter and in Chapter 8 in more detail, but for the time being we are ready to start looking at some of the routines that we must encode with a view both to parameter passing and to the actual function of the routine. Of course, there will be some routines that don't demand any parameters.

Compiling routines

Assuming that you are ready, we can start to compile some of the routines. The procedure for doing this isn't just to put pen to paper; you will be forced to think a bit about what you are doing. The system that I describe here shouldn't be strictly adhered to after a while; it's just to point you in the right direction, to help you get started. One of the key considerations at this stage is not to work too fast — you will not succeed, and you will only be disappointed. Working in machine code is a slow and laborious process: and be ready for the bugs, for they are the real test of your stamina.

The first stage of compilation is to study the routine to be encoded. Are you fully aware of its function? Is it simple enough to encode, or have you failed to simplify the tree to the required extent? You must know what you are trying to achieve right down to the last detail. Furthermore, you must be well aware of what parameters the routine requires.

The next stage is to try and find the corresponding ancillary routine (if there is one). You may find an ancillary routine with a close similarity or a very vague one to the routine for encoding. More often than not, the encoding routine will be a hybrid of two or more of the ancillary routines. If there is no real similarity then it looks like you are faced with the prospect of writing the routine from scratch: put it aside for the time being and move on to the next routine (the next chapter deals with writing from scratch).

In order to do this speedily, you would be well advised to get to know all of the routines fairly well, so that when you come to use one you already have a fair amount of confidence in it. But don't just leave it at that; make up your own routines and document them. Then start mixing them (mentally) and you may be amazed at some of the new routines you find yourself with.

Study each routine and establish the parameters it needs. These fall into two classes: in-going and out-going. In the case of in-going data, find out which (if any) are staying constant. This information may as well be entered by the routine itself rather than by the calling routine. All this

information should be carefully recorded in the program files so that it's available when you need it.

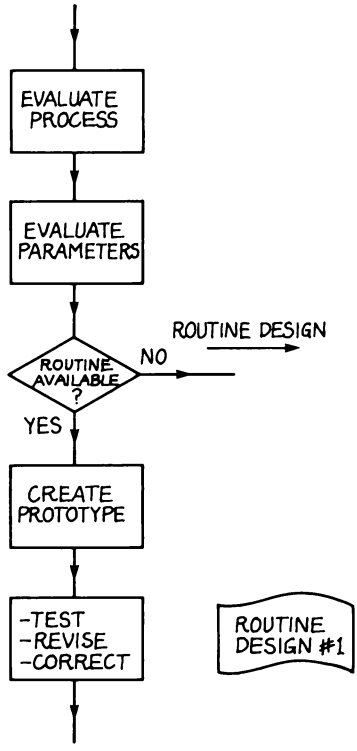
I must point out the danger of parameter clashes: these occur when more than one routine uses the same memory location for a different parameter. This results in loss of data and an erroneous entry for its replacement. Most of the time, clashes will not matter as the information carried in these locations won't be needed any more, but occasionally disaster will strike and it can be very difficult indeed to spot the fault as the routines work on their own, but result in a bug when run together, which is very frustrating. To prevent this, you must keep a detailed record of what memory locations are used, and where.

If you've followed the instructions so far you will now be in a position to write the routines (well, some of them, at least). Whether or not you own an assembler, you need a copy of all the routines you write so that these are instantly available. I would suggest copying the routines on to a ruled page so that everything is crystal clear: you can't afford any ambiguity.

The first version is the prototype — just make sure it works. It doesn't matter if it's a little inefficient. At this stage, you should not be using any of the clever tricks that abound in machine code programs: keep things plain, and try to avoid the use of JMP instructions within a routine as these render the code un-relocatable. There should be no call to jump further than 127 bytes, which is the limit for relative branches. To use a conditional branch as an unconditional branch, you force the branch by setting the flag which governs that branch. An example of this is to precede a BCC instruction with a CLC instruction. If you are hand assembling, then be wary of calculating these vectors as they are a common source of error.

Once you have produced the prototype routine, assembled, tested and corrected it, the next stage is to improve it. You may like it as it stands or you may have noticed some improvements that you can add. If so, go ahead by all means and change it — but don't forget to re-test it after the change.

Figure 5.1: Routine Design Number 1.



CHAPTER 6
Writing Your Own Routines

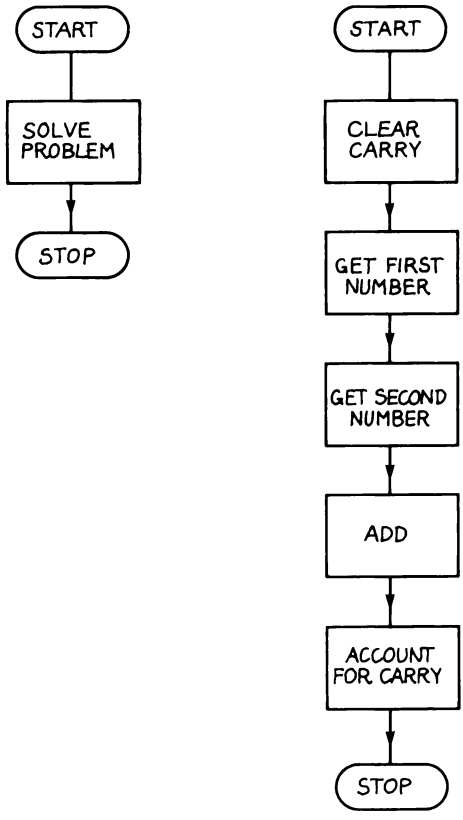
Not all the routines you are going to want can be written for you. You either have to try writing them, or shy away from machine code and write them in BASIC. At least make an attempt at machine code: even if you fail miserably, you will have gained valuable experience.

The first consideration is whether or not you know how to achieve what you actually want in normal terms. Suppose you want a routine to play chess, it's difficult enough describing the process in English, let alone converting it into machine code. If this is the case, then what you have is an algorithm problem — see the section on algorithm design later in this chapter. But don't expect instant success, inspiration may not come at once.

Well then, assuming you have got your algorithm worked out, we can consider turning it into an operative machine code routine. First, draw up a flowchart of the algorithm. For planning programs as a whole the flowchart is a poor tool, but for laying down an algorithm it is very effective. Try to make each box as simple as possible, but there is no need to get right down to the instructions themselves. What is wanted is something between the two flowcharts in **Figure 6.1**, which add two numbers together. Where necessary, indicate program flow lines with arrows. Where program control can take various paths, these should be kept to a minimum as otherwise the diagram can become cluttered. A major cause of failure in machine code (and other computer languages) is an over-complicated branching structure. This leads to unforeseen logical errors, leading to a very frustrated programmer. It should be possible to re-draw the flowchart several times, simplifying both the flow structure and the instruction content.

By now you should be well aware of parameters to be passed to and from the program. Note these down at the top and bottom of a sheet of paper. From the flowchart, the main part or 'spine' of the routine should be obvious. Don't worry about the other little bits, just copy down on the sheet the code which you think will perform this task. If you're not sure then type it in and check it. You will then be able to start adding in the other 'arms' of the flowchart. After each stage stop and check — don't do it all in one go. Once this is done you have written the routine. Look at the top and bottom of the sheet and add in the parameter passing code. This is just the prototype — if it works, then fine, but you

Figure 6.1: Over and Under-simplification.



may already have plans for a superior second version. The following sequence of charts and code shows this method in use.

The development of a Spiral Screen Fill routine

Filling the screen from top to bottom is child's play. Doing it from the centre outward is not so easy. Doing this in machine code certainly seems daunting at first. The solution, however, is painless. Using the Rectangular Fill routine (see Filling Memory, Chapter 9) simply draw larger and larger rectangles centred on the middle of the screen. The result of this is to fill the screen from the inside out.

If we look at the Rectangular Fill, we find it takes several parameters. First, it wants the address of the top lefthand corner of the rectangle. Then it wants the width and height, and last the code with which to fill the area. These are placed in the following locations:

251 & 252	Lo and hi addresses of the rectangle's corner
253	Width
254	Height
2	Code to fill with

Now the screen is 40 characters wide and 25 deep. If we increment the length of each side by 1 to produce the next rectangle, then the result will be asymmetrical. We must increment by 2. This means that the starting height is 1 (so we get 25 not 24). Following on from this, 12 increments will need to be made to fill the screen. This means a total of 13 rectangles. A little algebra tells us the starting size of the width:

$$12 \times 2 \text{ (there are twelve increments)} + \text{WIDTH} = 40$$
$$\text{WIDTH} = 40 - 12 \times 2$$
$$= 16$$

A little more algebra supplies us with the starting corner. After each increment, the corner moves to the left and up — diagonally north-west. We know the final position after 12 moves must be 1024 so:

$$\text{ADDRESS} - 12 \times - 41 \ (-41 \text{ is the vector}) = 1024$$
so
$$\text{ADDRESS} = 1024 + 12 \times 41$$
$$= 1516$$

Now we can draw the first flowchart (**Figures 6.2** and **6.3**).

Spiral Screen Fill

```
:LL0    LDA #236
        STA ADDR
        LDA #5
        STA ADDR+1
        LDA #16
        STA WIDTH
        LDA #1
        STA HEIGHT
:LL1    LDA ADDR
        STA 251
        LDR ADDR+1
        STA 252
        LDA WIDTH
        STA 253
        LDA HEIGHT
        STA 254
        JSR FILL
```

Figure 6.2: Preliminary Flowchart.

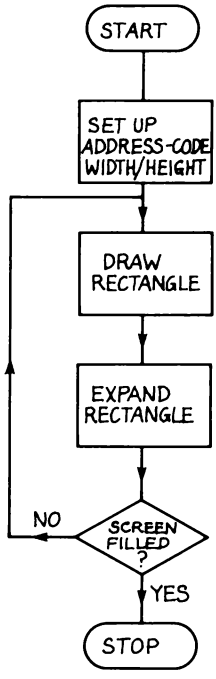
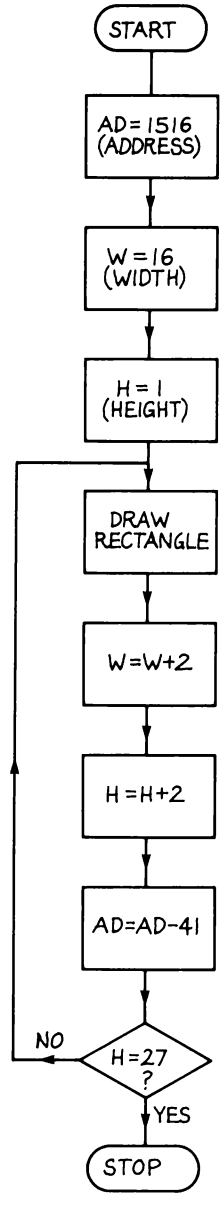
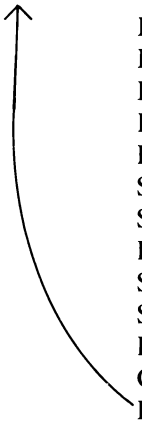


Figure 6.3: Coding Chart





```
INC WIDTH
INC WIDTH
INC HEIGHT
INC HEIGHT
LDA ADDR
SBC #41
STA ADDR
LDA ADDR+1
SBC #0
STA ADDR+1
LDA HEIGHT
CMP #27
BNE LL1
RTS
```

This routine then produces a screen fill from the centre outwards. It's certainly very verbose, but it works. Further refinements should be made only when the prototype routine has been produced. You may want a delay to help slow down the action — as always, it's a case of trial and error.

Algorithm design

Sooner or later you will turn to this page. Of course this book can't cover every eventuality: if every routine that could ever be written were to be listed here, then the whole magic of the world of computers would be broken. Computers are about virtually infinite possibilities. This is where this section comes in. It's designed to stimulate your brain so that you can solve problems and produce new routines to perform functions that perhaps only you have ever conceived of. The problem may be simple or it may be complex. Worse still, it might be infuriatingly simple in nature yet hard to solve. This chapter is about algorithm design. The word algorithm implies a way of doing something. This chapter aims at showing you how to create your own.

The first step (simple though it may seem) is precisely to define the problem. Includle every detail of the task. Now look at it and try to break it up into lesser tasks. In the section on program design, you were asked to do this but here we are looking for a more subtle connection between the tasks. Obviously, you simplified the task as far as possible then, but now the idea is to think in term of the two routines possibly working in parallel with each other. Consider the trivial example of keeping a randomly wandering object within a square. The problem can't be solved by suggesting that the particle never strays further than a set distance from a central point as this describes a circle: trying to find

an algorithm in terms of the distance won't work. The answer is to consider the vertical and horizontal distances separately. Don't expect the answer to jump out at once. This is simply a step in the right direction. Don't try to solve one algorithm on its own. Do several at once as you may find that one will help with another. The ability to think clearly and with a very open mind are key attributes that you will gain from algorithm design.

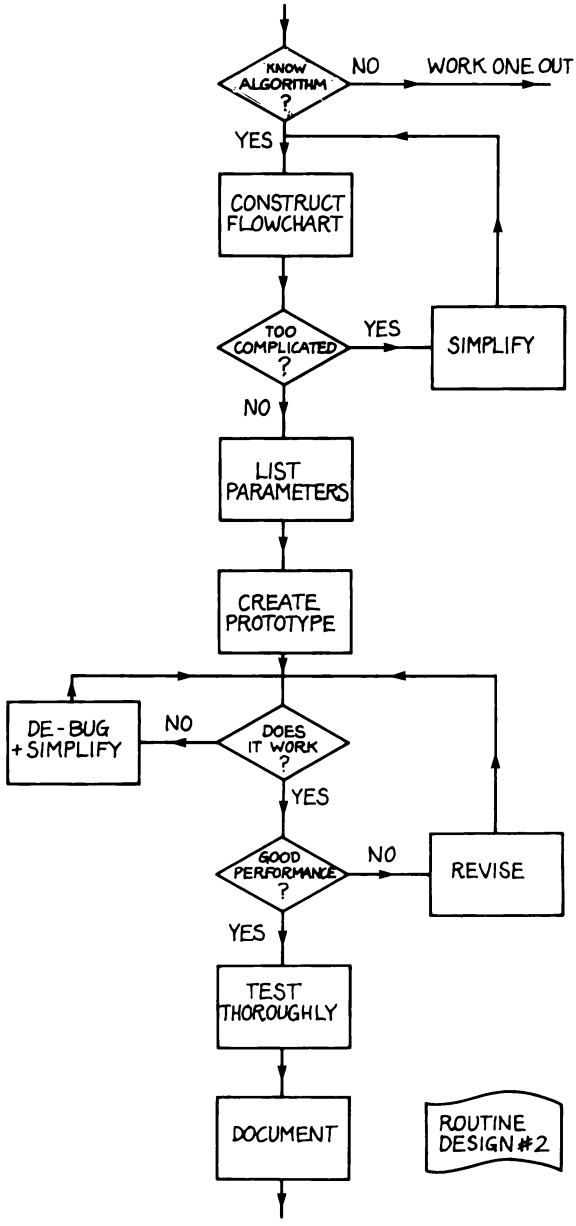
Once this first stage is under way (the whole thing is a dynamic process) there are various things you can try in order to find a solution. These are not the only methods, so don't refrain from using your own. One essential, however, is a doodling pad as it lets your mind wander into wider avenues than before. The aim is to reduce the problem into a series of clear and unambiguous stages suitable for coding (some people tend to forget what they are aiming at). It's important to keep the destination in mind, so every now and again re-read the definition of the problem.

The first thing to remember is that every problem has a solution. And the hardest ones sometimes tend to have very elegant solutions. Just as in the field of maths, help often comes from a section you thought had nothing to do with the problem. Consider the problem of creating a sense of movement into the screen (ie falling down a shaft) by drawing perspective lines and moving them outwards, slowly at the centre and quickly at the edges of the screen. Who would have thought that a solution lies in the use of UDGs (see Chapter 10). This brings up another point — there is more than one solution so don't fall into the trap of thinking that 'there's only one way to do it, and it has got something to do with this . . .'.

Don't be blinded by your destination. The chances are that the solution to the problem takes an indirect route there, so don't look for the single-stage solution. Magazines are a good source of ideas and programs — you'll be surprised what you can pick up by browsing through the listed programs and notes, which will show you how others solve problems.

There is normally a lazy way out of algorithm solving, in that it's normally possible to take each instance separately and write a routine for each. This is highly unsatisfactory but it does mean that you can get a sort of stand-in. There are circumstances when it is a perfectly good alternative. Take, for example, the task of providing trigonometrical formulae in machine code for 3D perspective projection. One way is to write floating-point routines and then use approximation formulae. This is very clearly a daunting task. The answer is that the ratios need only be supplied in 5° intervals anyway, and so you might as well use a table of angles and values instead. The result? — a fast routine that's quick and easy to write.

Figure 6.4: Routine Design Number 2.



I have a certain affinity for the flowchart in algorithm design. It allows me to be much more direct with ideas and it's easy to see how things are happening. This is really just a formal doodling. The process can be

quite fun although if you're not careful the diagram may get over-complicated. But I think the flowchart ought to have a place in the programmer's arsenal.

Manually performing the process you are trying to devise an algorithm for can often be enlightening. After a while you find yourself 'doing things without thinking'. It's at this stage that you must probe further in and enquire what is the underlying principle. Consider the eight queens problem:

Place eight queens on a chess-board so that no queen attacks any other.

Write an algorithm, suitable for use by a computer, to solve this problem. The task does seem a little overpowering at first, but if you attempt the solution manually you will find that it just takes a few minutes to see the light.

One of my pet ways of solving problems is the approximation method. I start off with an idea and gradually improve and refine it until I find the answer. The initial approximation takes the form of an educated guess at the solution and what factors may help solve it. The answer can turn out to be totally Heath Robinson due to all the alterations but that doesn't really matter — as long as it's 'watertight', it will work and you can always improve on it later. Remember this is just the prototype.

These then are some of the chief methods of algorithm design. Use them liberally and just let your mind wander. Inspiration may come to you at the most unexpected moments. It's unlikely that you will work things out instantly, but then again it's unlikely that you'll find yourself with too many difficult algorithms anyway. Most of the problems not supported by the service routines supplied will yield readily to decomposition.

CHAPTER 7
Testing and Debugging

No program in machine code is ever without its associated bugs. The longer it is the worse they are, and the more numerous. It is reasonable only to expect the shortest routines to work first time. This chapter is all about the hunting and killing of bugs. It's vital to remove them completely and kill them *dead*. The good programmer expects bugs (but hopes for none) and so is quite prepared to fight them.

Bugs come in all shapes, sizes and guises. They may be caused by errors, faulty algorithms or any one of a hundred causes. There is no such thing as the standard bug and sometimes it's difficult to even recognise the problem. One of the best debugging aids is the dry-run with paper and pencil but prevention is always better than cure, which is why this chapter comes in three sections. First how to keep your routines simple, second how to test them and third how to debug them. This chapter is possibly the most valuable in the whole book as it only takes a few bugs for you to lose heart and give up. Stand up and fight them: after all, if machine code programming was simple you wouldn't get so much satisfaction out of it. Trouble-shooting a program can even be fun if you enter into it in the right spirit.

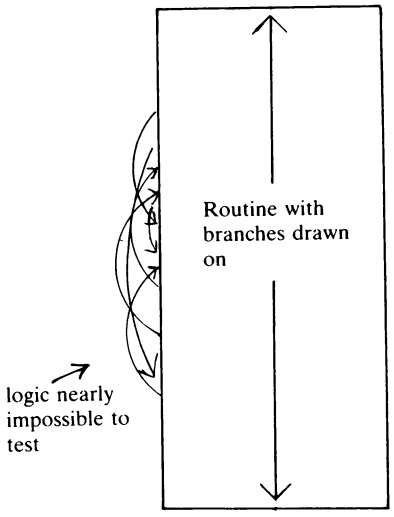
Keeping routines simple

A simple routine is a programmer's dream. All you have to do is keep it short and straightforward — no clever tricks. There should be as few branches as possible to minimise the total number of different paths. Don't hesitate to use a JSR in a routine even if this is the only time that routine is called. If the routine is therefore simpler, then the JSR is doing its job. Ideally a routine should be no more than, say, 25 instructions. This is so that it can all fit on the screen in one go. Any more than this, and start thinking about some JSRs.

Writing a program off the top of your head has a noticeably cramping effect on your thought processes — even in BASIC — as you can see only 25 lines at once. It's far easier to work from a paper listing than a screen — you can see the whole program at once. If you look at some of the routines in this book you will notice that they are kept short and

simple. The branches are also minimised. With very few exceptions, you will also notice that they are 'nested' one inside another. When control is passed to and fro between the different sections of a routine like the one in **Figure 7.1** it becomes very difficult even to test the routine. In fact, one mainframe program recently revealed a bug after over 10 years of flawless use! If you keep the branching logic simple then testing becomes far easier.

Figure 7.1: Haphazard Branching.



HAPHAZARD BRANCHING

Writing straightforward routines is largely a matter of practice. If you plan properly you will find your routines fall into place automatically, with simple branching. Following the top-down chart theory, the boxes become more complex as you move higher up the tree. If the leaves at the bottom are complex then who knows what the top will look like?

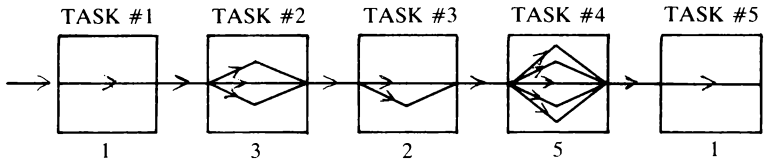
Testing the routines

The first law of debugging is 'You can never be sure a program is totally correct'. The second is, 'Some bugs are hard to find and easy to correct, some bugs are easy to find but hard to correct, and some bugs fall into both categories . . .'.

When testing a program, it is necessary to ensure that it works for all possible paths of flow through the program. What I mean by this is that a particular routine (say a keyboard scanning routine) may contain

various paths for the machine to follow. If one key is pressed, then it may execute one section of code, and if another key is pressed then another section may be executed. Thus there are two different 'paths' for the program to follow. To test this routine we must ensure that it can follow both paths correctly.

Figure 7.2: How Many Control Paths?



Consider **Figure 7.2**. This shows a program made up of five different sections. After completion of one section control moves on to the next. Under each box a number denotes the quantity of different paths the machine may take through each box. Thus the total number of different paths through the entire program will be the product of these numbers, ie $1 \times 3 \times 2 \times 5 \times 1 = 30$. To test this program fully, we must check that it can execute each of these paths correctly. This is obviously going to be tedious to say the least (bear in mind that a full program would have thousands of different paths). Therefore a system known as modular testing may be used. This entails testing each section individually through all possible paths. What this means is that, if we know that the section 'play' is totally correct (or appears to be), then we can forget about it. Using this method reduces the number of trials to the sum of the different paths, ie $1 + 3 + 2 + 5 + 1 = 12$.

Notice how, even in this trivial example, we have greatly reduced our work. This is often the natural way to test your code as, when you have just finished writing a section of code, the chances are that you won't have all the other routines finished to go along with it. Test the routine while your mind is still attuned to it.

This leads us on to the stage where we want to test a particular routine but don't know how to do this. First of all make sure that you have a copy on tape or disk and then proceed as follows:

1) Examine your charts for this section of code and pick out every branch or jump instruction. These are the only points where control may split in two. Now we aim to make the program travel along each of these paths by 'seeding' the program so that each of the branch and jump instructions are executed (not all at once but each in turn). Seeding the program is very simple — if you can't force the program to branch at a BCC instruction without obscure circumstances, then insert a CLC instruction before it (this may upset branch offsets): the program

will not know the difference. This method may be a bit ugly but it works — don't forget to remove the 'seed' after use. What this first stage has done is to show that all lines of control are operative.

2) The next stage is tougher on the program. The aim here is to confuse the routine by deliberately giving it foul data, such as it might receive while the program is running. It is up to you to be ruthless here. Often the programmer is well aware of an error in his thinking but pretends that it is not present. Confronted with this type of situation you must go back and correct your mistake as it will only return to haunt you later on.

If your routine comes through with flying colours, then congratulations! If, however, a bug rears its head then it is imperative that you debug the routine before continuing. Remember one thing — the computer doesn't make mistakes. If something doesn't work then it's your fault. Many's the time I have seen programmers notice an error and pretend that it's the machine's fault. Often a bug doesn't appear every time a program is run: if a bug appears and then hides for a while, don't fall into the trap of thinking that it really was the computer's fault. The rule is that if you see a bug once, then it exists and no amount of ignoring it will get rid of it. Once you admit that there is a bug in your code you're half-way to correcting it.

Debugging

So now you have a bug and you know roughly where it is. You may even have an inkling as to its cause. Apart from giving in to the pest, you have three options: you can correct the bug; you can rewrite the faulty section; or you can decide that the bug adds a new dimension to the game, as in, 'Avoid the white blobs on the screen as they are spiral vortices which if hit cause time to stop and the game ends', ie the machine hangs up. (OK this is taking it a bit far but you would be amazed at the number of programs which have been aided through divine intervention like this. You have probably done it yourself and, while you should guard against doing it too often, there are occasions when bugs fit the bill perfectly.) To give this underhand system a more technical name use the phrase 'If it doesn't work then document it (but don't tell anybody)'.

The favoured method is, of course, to correct the bug. However you may not be able to track it down. Worse still, you may have tracked it down but know that fixing it isn't going to be easy. It is at this (later) stage that you should resort to rewriting the code.

The first stage of debugging is to check that what you have entered on

the computer agrees with what you wanted to enter. A bug arising from a typing error or misprint is a gremlin (merely a distant cousin of the bug). Make sure you do this first, as otherwise you may tamper with the code and only later discover the misprint — a sheer waste of time.

The next step in debugging is to ask the question, 'Does it hang up?' If the routine goes into a loop then one of two things is probably happening. First, check all the branch instructions, both conditional and unconditional, as it is through these that the program can go into a loop. Often you will find that you have miscalculated a relative branch vector. If the code is all correct, then the next most likely cause is that 'completion conditions' are not occurring. Every section of code will exit the routine when the job is done. This is where the completion conditions come in: they inform the routine when the job is done. If completion conditions are not occurring then there is probably a small oversight in your code (ie the algorithm is probably correct) so the best solution here would be to dry-run the code with a pencil and paper and the error will give in without too much of a fight. If you find that your code still hangs up and neither of these explanations is the right one, then re-check the code with a fine-tooth comb: the error is in there somewhere. You will probably be best off with a dry-run as above. It's only a matter of perseverance, that's what makes the machine code programmer.

If however your code doesn't hang up but flatly refuses to work, then there is an error in your algorithmic reasoning. Just in case this isn't true, check the code for gremlins again and be especially careful over what addressing modes you wanted to use and what addressing modes you actually used. If all this is done and you still can't see the fault, then it's time to use some diagnostic aids. These will allow you to follow the program flow closely which, with luck, will enlighten you. There are a number of different aids so I would advise you to get to know them now, so that you are aware of the aid that will assist you most in your plight.

The first is a technique used on much larger computers. It involves listening to the noise the processor is making (ICL machines are fitted with a 'Hoot Volume' knob). On a mainframe this is detected and amplified so that the engineers can listen to the machine's activity. On the 64 this noise can be heard on the TV. All you have to do is turn up the volume and listen. Of course if you are using the SID then you won't be able to hear the 'hoot'. With the volume on, listen to the noise. Now enter this line of BASIC:

```
FOR A=1 TO 3000:NEXT
```

and press return. Hear the difference? This is the noise the machine

makes when in a loop. Of couse the note you hear depends on the rapidity of execution. Thus by turning up the 'hoot volume' you can listen in to the processor.

The second technique is a form of flagging. If your code is concerned with on-screen activity then, in order for you to see what's happening, it may be necessary for you to insert a delay into the main loop. This is easily done by use of the JSR instruction. Simply insert JSR (address of delay) into the loop and run the code. For details of how to write a delay, see the section in Chapter 9.

However, just slowing down the code may not be enough. You may wish to see exactly the value of a register or just how far across the screen a pointer has got. This is easily achieved by POKEing the information on the screen before the delay and then removing it afterwards. To show how far across the screen a pointer has got, I would advise POKEing the address it holds with a 160 (white square) and then removing this after the delay. When run, a white square will zip across the screen. If you want the value of a register, then use the absolute X or Y mode coupled with a base address of 1024. Once again, POKE a white square on before the delay and remove it. Thus the value of the register can be gleaned from the position of the white square. If the square is in the top lefthand corner of the screen then the register has value 0; the top righthand corner makes it value 39, and so on.

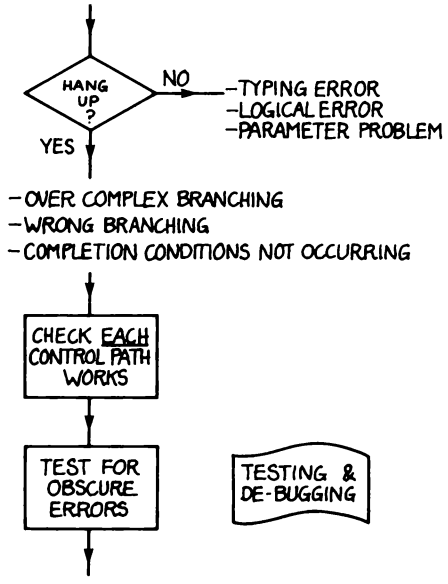
A far more elegant way of ascertaining register values is to use the neglected BRK instruction. This is a software-called interrupt and is non-maskable so you can be sure it works. Its use is simple: you simply sprinkle BRKs liberally throughout your program and, when encountered, they force an interrupt. The vector is contained at 790 and 791. On older CBM machines the BRK instruction traditionally pointed to the machine code monitor, but on the 64 there is no resident monitor so we aren't going to lose a lot. We can work the service routine two ways. We can store register values safely somewhere and then return to BASIC, or we can test for the function keys being depressed and act accordingly. If no key is held down then the program continues as normal: F1 slows the program down, F2 stops it, etc. All this is dealt with in the section on the interrupt and the Mini-toolkit in Appendix B. For the moment we shall limit ourselves to the former option.

```
:LL0    STA 820
        STX 821
        STY 822
        JSR 58251   (return to BASIC)
```

Site this routine somewhere, say 49152, and alter the BRK vector to the new address. On returning to BASIC you can discover register values by

PEEKing 820–822. A BRK is really a JSR to a preset address that takes only one byte of programming. For a more flexible system the best thing is to build it yourself — see Appendix B — and hopefully you can custom-design your own debugging aid. Good luck.

Figure 7.3: Testing and Debugging.



CHAPTER 8
Connections: Stringing the Game Together

After you have completed a number of functional routines you will notice that you are in a position to combine them. You can see this by looking at the top-down chart you drew earlier on. Only the leaves (final boxes) on the tree actually do something. All the other boxes simply connect them together in the desired structure. Once you have completed enough routines start connecting them, but only after they have been thoroughly checked and tested. If this hasn't been done, when a bug arises you won't know where to look. If you have checked that the individual routines do work, however, then you know that a bug is caused either by a parameter problem or by a faulty connection.

The first move is to look up the box on the tree and find out what connections must be made. Check each of the routines and see what information is passed to and from them. Check that the routines don't clash with each other. If problems do arise then you can either modify (and retest) the functional routines or patch up the fault in the connector. You should really have been aware of clashes when writing the routines anyway — that was why I emphasised the research into the exact function and nature of each routine.

Don't forget that further up the tree an old result (a parameter or data generated some time ago) may be called in for use. As long as you're aware of this then it's OK. Talking about moving further on up the tree, the connector box may not be a simple connection. It could require iteration or conditional branching or even a DO–UNTIL loop. This section will describe how to produce these constructs. Of course the first part is the one-pass connection. This simply entails each routine being called in turn: the connecting routine then returns to where it was called from.

Each routine is called with a JSR. Any parameters should be set up before this and afterwards a bit of tidying up may be necessary to prepare for the next JSR. At the end of the routine you store away any data needed in the future and then finish with an RTS. If some of the routines are simply one-offs, then you may already have written parameters into them. This is far preferable, as it improves the readability of the program. For instance:

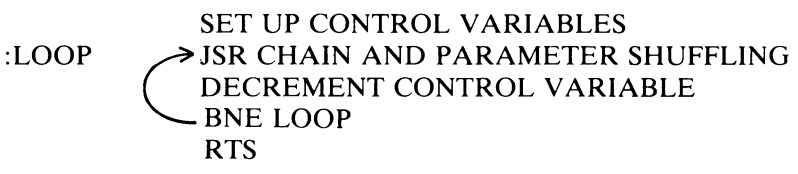
```
JSR KEYCHECK
JSR MOVESHIP
JSR MISSILE
RTS
```

is infinitely more clear than:

```
LDA #145
STA 121
LDA #34
STA 2
LDX #35
JSR KEYCHECK
LDA 2
AND 121
STA 121
JSR MOVESHIP
LDA 145
EOR #255
STA 2
LDY #40
TYA
JSR MISSILE
STA 2
RTS
```

isn't it? If you concentrate on austerity, your programs will almost write themselves. Thus the one-off box connection is simply a chain of JSRs with parameter activity in between and an RTS at the end.

Where iteration is required, things get a little more complicated. Ask yourself the question, what is the control variable? Then find out if the number of iterations stays constant or if this in turn is a parameter. Once you know this, you simply start the connector by setting the control value safely somewhere in memory. Then you proceed as normal for the one-off routine — JSR calls and parameter shuffling. At the end of this, decrement the control variable and use a BNE to loop back to the start of the JSR-calling section. The chances are that you will need less than 256 iterations, so only one-byte precision is required. Thus the construct is as follows:



The connector requiring a DO–UNTIL construct is much the same as the iterative one. The first stage is to find out what the operative condition (or conditions) is. All you have to do is write a routine (probably a compare) to test this condition and place this after the JSR chain. This is followed by a branch instruction which loops the program back if the condition hasn't been satisfied. Thus it looks like this:

```
:LOOP    ↗JSR CHAIN AND PARAMETER SHUFFLING
         (  TEST CONDITION
         ↖BRANCH ON NOT TRUE TO LOOP
          RTS
```

Testing for such conditions as 'less than' and 'greater than' comes under conditional branching. The conditional branch connector is possibly the most complex connector of the lot. Only the main branch conditions can be covered here. The resulting connector can get messy as far as branching is concerned, so be careful. There are two main types of routine here: where more than one routine can be called due to two conditions being satisfied, and where only one can be called. Each of these can be further split up by a connector which uses the same information on each condition and one which uses different information. What I mean by this is that one may branch to several sections depending on the player's score, whereas another may look at score, time left and other factors. Despite all this complication, any branching routine can be written as follows:

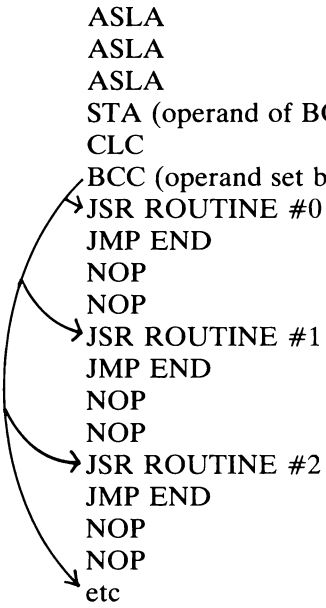
```
   TEST CONDITION #1
(  BNE
(  JSR ROUTINE #1
(  *
↘TEST CONDITION #2
(  BNE
(  JSR ROUTINE #2
(  *
↘etc.
```

The BNE here is being used to signify 'branch if condition not equal (not true)'. Thus each condition is tested in turn and, if true, the corresponding routine executed. The '*' shows where to insert a JMP (a forced branch is better) if you only want one routine run. This would change control at the end of the connector. This may be desirable in the following situation.

Suppose you have a ranking chart which comments on your score from one of five phrases. If you leave out the '*' part, then for each

phrase you will have to test for being greater than *and* less than another two numbers, to check if the score lies in a certain range. By simply starting at the top and using a 'greater than' check with a '*' jump, you only have to test for the 'greater than' condition.

Where routines are being run depending on a value in a limited range (say X = 1, 2, 3, 4, 5) then, by modifying a branch vector, you can eliminate the need to test for a condition. Take the situation when the value in the accumulator can range from 0 to 5, which dictates which routine to run. This can be achieved like this:



The number in the accumulator is multiplied by eight and the result stored in the operand byte of the BCC. This makes the program branch to the start of the corresponding JSR instruction. The two NOPs simply ensure that the JSRs are eight bytes apart.

Testing for conditions such as 'greater than' and 'less than' is easily achieved by use of the SBC instruction. Simply subtract one from the other and, by testing the carry flag or the negative flag, the relation can be evaluated. Testing for equality is, of course, a straight compare sequence.

If for some reason the connector doesn't work, the error shouldn't lie in the routines it calls — if you've tested them. Either the logic in the connector is wrong or the parameters are clashing. There's no excuse for missing a bug in this sort of routine. The construction is so simple that a

dry-run to keep track of parameter addresses should show the bug up fairly quickly. Once it works, don't forget to document the routine — it's going to be used later on when you've forgotten all about writing it.

CHAPTER 9
Direct Routines

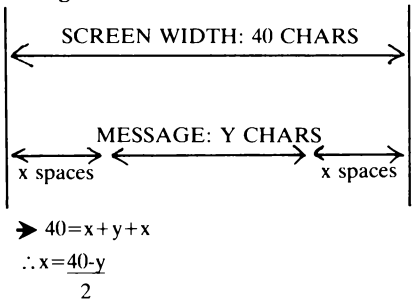
This chapter and the next one contain a large number of machine code routines, the 'common routines'. Each is accompanied by some text and suggestions or possible variations. They are more than just one-offs, they are methods. This means that a little thought on your part will produce many more routines. Play around with them, mix them and experiment. The Rectangle Fill (see Filling Memory) is a good example of versatility in an apparently 'normal' routine.

These aren't the only routines in the book, others crop up elsewhere but they are all listed in the routine index. Look out for more routines like these in magazines, among reader's programs. You may not be interested in the bulk of their programs, just in the odd one or two routines within those programs.

TEXT AND SLOW PRINTING

Most games, at some time or other, require messages to be sent to the player. These may be vital to the game or may simply add a degree of realism. One simple way to get text on to the screen is to use the COPY routine and simply copy out a chunk of memory with the relevant text contained in it. However, this method is very clumsy and wasteful of memory. Moreover, you may wish the text to appear gradually in a form of 'slowed-down' writing like the 'GAME OVER' messages at the end of many arcade games. And you may also want the message to be centred on the screen, so that it's pleasing to the eye. How are we going to achieve this? Well first let's review the problem in BASIC.

Figure 9.1: Text Centring.



$$\rightarrow 40=x+y+x$$

$$\therefore x=\frac{40-y}{2}$$

Let's say the message is contained in M$. Let's also assume that the cursor is already on the right line and all we have to do is slowly print out M$ along the centre of this line. The screen is 40 characters wide (**Figure 9.1**) and the length of M$ can be determined by LEN (M$). If the message is to appear centrally, then there must be the same number of spaces on the line before it, as after it. If we call this 'x' spaces, then we will see from the diagram that

$2x + LEN (M\$) = 40$

Solving for x gives us

$x = (40-LEN(M\$) \div 2)$

Thus the statement 'PRINT TAB (x);' will put the cursor in the position ready for printing, and the ';' will hold it there. Now all we have to do is PRINT each character in turn, using the MID$ command, and introduce a delay between each character. Try the following:

```
FOR C = 1 TO LEN (M$)
PRINT MID$(M$,C,1);
FOR DE=1 TO 100 : NEXT DE
NEXT C
PRINT
```

Don't forget the final PRINT, as otherwise the cursor will be left hanging after the last character of the message.

Slow printing

So that's how we do it in BASIC. In machine code, the message is simply a string of bytes terminated by a byte with a predetermined value (such as 255), where each byte represents the ASCII code of that character. Printing the characters is easy. All you do is set up a loop to read the data in sequence (ie put the value into the accumulator) and then call one of the ROM routines which prints a single character. To align the cursor to the correct position beforehand, call this character printing routine with some cursor controls (eg Home, cursor down × 5, cursor across ×15). Make sure that you are writing to the screen by calling the routine to clear all I/O channels, as otherwise you may find yourself dumping characters to the printer! A call to a delay routine while in the main loop will slow the process down so that the human eye can follow the process. Remember, messages of this sort are very good for breaking up the player's rhythm. Data input to this routine is: the

address of the text file, the text file itself and the duration of the delay (although this may already have been set). If you have a number of different messages, then it might be easier to have the slow printing routine called by, say, four or five subsidiary routines which set up the particular parameters for one message (they should also position the cursor). Then, when you want to print out a message, all you need do is make a call to one of the subsidiary routines. The routine returns nothing to the caller.

Here are the addresses of the ROM routines which I referred to:

CHROUT: This outputs to a device (in our case the screen) the ASCII character represented by the value in the accumulator. Its address is 65490 ($FFD2).

CLRCHIN: This restores all I/O channels to normal and means that, when CHROUT is used, the characters will go to the screen. Its address is 65484 ($FFCC).

Variations on this theme of printing characters might be to print a space between each character, thus 'expanding' the text and making it appear larger than it actually is. Compare the following:

GAME OVER G A M E O V E R

The second example is novel and therefore more pleasing, but don't overdo it. Other ideas might be to underline the message or, best of all, print it from the middle out. This is known as 'meta-printing' and can be very effective but it's a special effect so keep it for special things.

Meta-printing (BASIC listing)

```
10  L=LEN (M$) : IF L/2 = INT (L/2) THEN M$ = M$ + "  " : L =
    L + 1
20  X = INT ((40 - L)/2)
30  FOR P = (INT (L/2) + 1) TO 1 STEP -1
40  PRINT "[CU]" TAB (X + P - 1) MID$ (M$, P, ((INT (L/2) + 1)
    - P) * 2 + 1
50  NEXT P
```

This takes M$ and prints it from the centre outwards in the middle of a line.

The following routine prints out the message held in ASCII form starting at START and finishing with an entry of 13. The maximum length is 255 bytes:

```
:LL0    LDX #0
:LL1    LDA START,X     Get it
        JSR CHROUT      Print it
        INX             Move on
        *
        CMP #13         Return
        BNE LL1
        RTS
```

The '*' indicates where a call to a delay routine should be made if you want to include a slow printing effect. This is also the place to insert spaces to expand the text a bit. If you do this, make sure that you preserve the value in the accumulator.

If you want your messages to appear in a set position, then the best way of doing this is to precede the text with some cursor control codes. For example, you might have a message like this:

[HOM] [DOWN] [DOWN] [DOWN] GAME OVER

Using the following table to encode the control codes, the first four entries in the message table we get are 19, 145, 145, 145.

Cursor Home	:19	Clear Screen	:147
Cursor Left	:157	Cursor Right	:29
Cursor Up	:145	Cursor Down	:17

FILLING MEMORY

Filling memory with a certain value is a common request for the programmer as it shows up more than most the plodding speed of BASIC. By clearing the screen you get a free 'fill screen with 32' but BASIC goes no further than this. On the 64 a second screen-associated problem arises: if you clear the screen then the color memory is reset to the background colour. This means that, to put anything visible on the screen, you need two POKEs (video & color RAM). This problem does have another solution. Before clearing the screen you POKE 53281 with another value, thus changing the paper. Then you clear the screen and POKE 53281 with its former value. Be on the lookout for simple solutions like this to larger problems (see the previous section on Text and Slow Printing for clearing the screen).

There are essentially two main types of FILL: Block Fill where you simply give start and stop addresses, and Rectangle Fill where you specify the positions and dimensions of a rectangle (see **Figure 9.2(a)**). For filling small areas (<256 bytes) then a simple loop with the X or Y

registers in absolute register mode will do the trick: LDX#0, LDA #CHAR, :LOOP STA ADDR+X, DEX, BNE LOOP,RTS.

Block Fill

The following routine performs a straightforward Block Fill from the address in 251–252 to 253–254 (the final address in 253–254 will not be filled). A smarter routine using the zero-page plus Y mode is possible, but this routine is designed to be simple.

```
:LL0    LDY #0
:LL2    TXA
        STA (251),Y      Fill a byte
        INC 251
        BNE LL1          Move on
        INC 252
:LL1    LDA 251
        CMP 253
        BNE LL2
        LDA 252
        CMP 254
        BNE LL2
        RTS
```

In case you are wondering about the TXA, its purpose is to fill the X register with the character code to be POKEd.

Rectangle Fill

The Rectangle Fill is more complex but more useful. It's a fast way of producing a border: fill an area normally, and then fill a rectangle within this in a different code. The centre is overwritten producing a rectangle

Figure 9.2(a): Block Details.

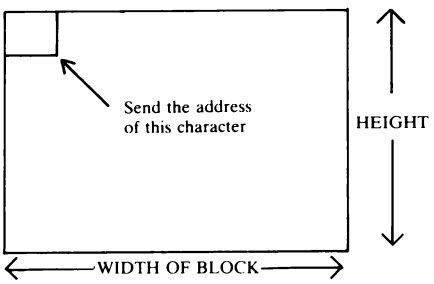


Figure 9.2(b): Obtaining a Border.

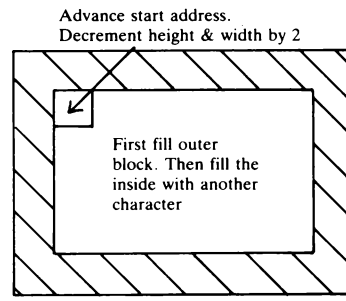
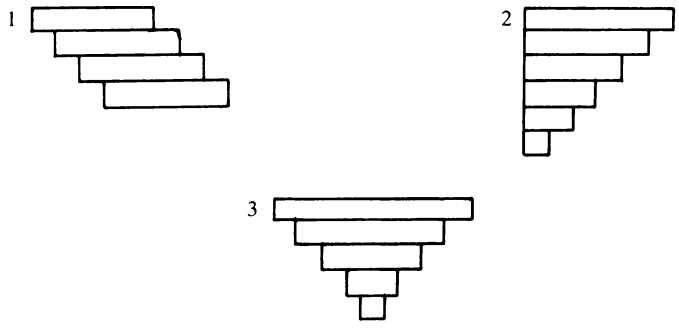


Figure 9.2(c): Complex Shapes with a Rectangle Fill.



1. Pretend screen is 41 chars wide
2. Decrement width each time
3. 41 char screen, decrement width by 2

with a border. As well as the code to plot, we need to know the address of the top left corner, the width and the height. One other small thing is also important; the width of the screen, so that each row of the rectangle is directly below the one above (a staggered effect can be quite eye-catching). With these in mind an algorithm might be to work out the address of the start of each row and plot as many characters as the width dictates. The following routine does just that:

```
:LL1    LDY #0
        LDA CHARCODE
:LL2    STA (251),Y          }
        INY                  }
        CPY WIDTH            }  Fill a line
        BNE LL2              }
        DEC HEIGHT
        BEQ LL3              Finished?
        LDA 251
        CLC
        ADC #SCREEN-WIDTH    Next line
        STA 251
        LDA 252
        ADC #0
        STA 252
        CLC
        BCC LL1
:LL3    RTS
```

The address of the top left corner goes in 251–252. The HEIGHT, however, does get reduced to 0 in the process so don't expect it to maintain its value on return. A triangle fill can be achieved by decrementing (or incrementing) the width each time round, but make sure that the WIDTH doesn't make the transition from 0 to 255 or vice versa! **Figure 9.2** shows the flexibility of this routine for producing a wide variety of shapes.

COPYING MEMORY

BASIC is notoriously slow for drawing pictures with: even the human eye can follow it quite easily. For the most part this doesn't really matter, but when switching from one display to another, or merely reprinting the background, it can turn a game very sour indeed. The solution, of course, lies in the use of machine code. Then the copying will take no more than a fraction of a second. The task is very simple — all you need to know is where to start and stop copying and also where to start putting the copied data. To explain this, suppose that from locations 49152–50151 lie the details of the title page of a game. To transfer this on to the screen all that we need to do is specify the start and stop positions as 49152 and 50151 and tell the routine to start copying the data found there at location 1024.

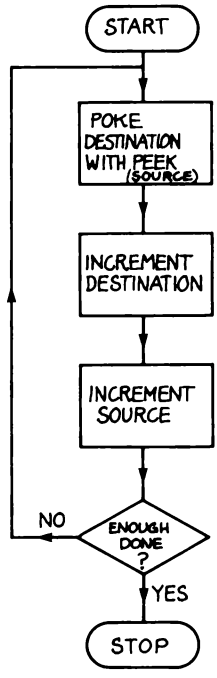
Most of the time you will be using the routine to copy screens or sections of a screen, and so it's likely that some or all of these parameters are staying constant. If so don't hesitate to write them into the routine itself. The other thing to watch out for is that the two areas must be totally different from each other. If any of the memory locations occur in both the section to be copied and the area to be copied to, then you may experience problems — this is the underlying principle of the SCROLL and is discussed under Scrolling later in this chapter. Taking stock of this routine then, we have the following algorithm:

1) PEEK the address of C
2) POKE A, C
3) C = C + 1
4) A = A + 1
5) IF C < E THEN GOTO (1)
6) RETURN

In this description, the following letters represent these factors:

C: This is the first address to be copied from. After each iteration, it is incremented so that next time round it's the next character's turn to be copied.

Figure 9.3: Copying Memory.

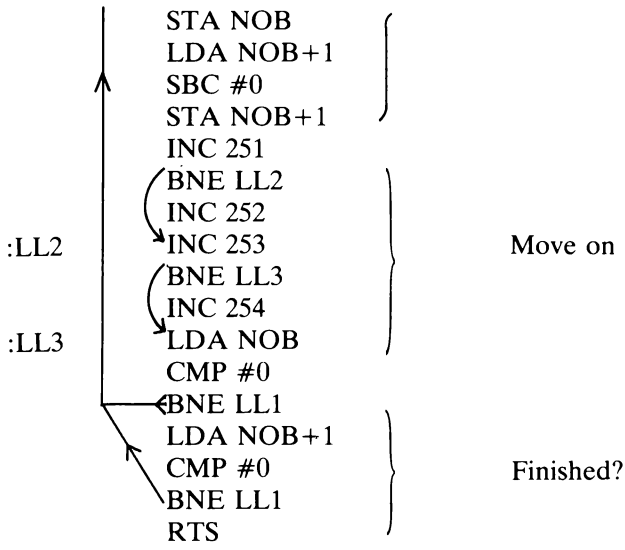


A: This is the first address to be copied to and is incremented in the same way as C.
E: This is simply the final address to be copied from.

Memory Copy

Copying screen layouts is not the only use for COPY routines. They may be used to set up arrays of alien ships in tables in preset attack waves, or perhaps re-locate a machine code program for you. The routine given below is for memory-screen copying, but could of course be easily altered. Nothing is returned by the routine in the way of parameters.

```
:LL0    LDY #0
:LL1    LDA (251),Y     Pick up data
        STA (253),Y     Copy it
        SEC
        LDA NOB
        SBC #1          Decrement no. bytes to be copied
```



NOB is the address of the number of bytes to be copied. Thus NOB and NOB+1 are the lo and hi bytes respectively. The actual copying proper is performed by the second and third instructions while the rest of the routine moves on to the next copying location. A variation on this might be to copy a 'window' instead of just a straightforward chunk of memory. This idea is used in the game 'Laserbike' (see Appendix D) where a 10 × 10 zone on the screen is copied on to from the playing zone.

Uses of the copy are manifold but, if the interrupt is used, then the possibility of creating a permanent background effect becomes possible. All that's needed is a copy of the backdrop somewhere else in the memory and the relevant hook-up routine.

DELAYS

Don't look here if your program is in BASIC! BASIC, as I have often said, is a very slow language — it puts delays in where you don't even want them by *interpreting* the language as opposed to *compiling* it. If you do want a sizeable delay in BASIC, then try the following method:

```
FOR A = 1 TO 1000 : NEXT
```

This will give a delay of just over one second. All it does is waste time by counting up to 1000 (in one second).

Working in machine code, you may find that your game really is too fast to play with any skill. It's in this situation that you need a delay. Just

insert it somewhere into the main loop and, every time the loop is executed, the delay will cause a pause. Of course we are talking here of delays of fractions of a second and not seconds but the underlying principle, of wasting time in a loop, applies. For some odd reason, the 6510 instruction set contains an instruction for doing nothing! It's known as 'No operation' and has the mnemonic NOP (implied). This instruction can be introduced into a time-wasting loop to double and triple the length of the delay, but first let's think about just why we need this delay.

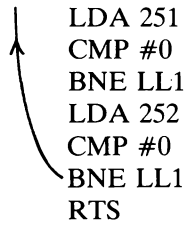
The very idea of a delay is to slow the game down, but speeding up the game (to a certain extent) as it progresses is one way of putting pressure on the player and forcing him into errors. Thus it would be nice to be able to vary the length of the delay. If the delay length is held in a 16-bit number somewhere, then the delay routine can PEEK the value required and, when you want to accelerate the pace of the game, you can easily reduce this value by subtracting some value from it. My solution to the problem is given below:

1) Read the length of the delay.
2) Decrement this value.
3) Perform some NOPs.
4) If the delay count is still > 0 then go back to (2).
5) Return.

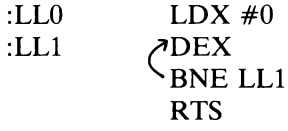
It is possible to predict accurately the length of the delay by adding up execution times outside the loop, and adding this to the loop time multiplied by the number of iterations, but I feel that the best method is to play it by ear and experiment with various values until you find a satisfactory solution.

The following program executes a delay of a length detailed by the 16-bit number in location 253–254. This value remains unaltered at the end of the routine.

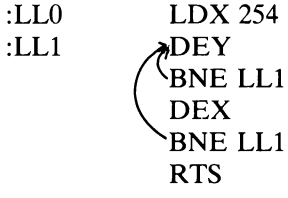
```
:LL0    LDA 253   \
        STA 251    |
        LDA 254    |  Set up
        STA 252   /
:LL1    LDA 251
        SEC
        SBC #1
        STA 251
        LDA 252
        SBC #0
        STA 252
```



Far simpler but less flexible is the following delay which utilises the X register:



Note that, although the register is loaded with a zero at the start, it is decremented before it arrives at the BNE, turning it into a 255. This could be extended a little by inserting a few NOPs in the main loop. By embedding a similar routine, but with the Y register, into this routine we can reach a compromise between the two routines:



The length of the delay is stored in location 254.

By hooking a delay routine on to the interrupt, it is possible to noticeably slow down your programs. But remember that this delay will be executed every sixtieth of a second, so don't make it too long. This is used to great effect in debugging programs. A hefty delay will slow down BASIC so much that clearing the screen can take seconds. (For some odd reason, clearing the screen is done from the bottom up rather than from the top down.) Anyway, use these three different delay routines where you think they are needed, but remember that, in general, it's more convenient to have a delay that doesn't destroy the length of the delay (the parameter dictating the duration of the hold).

UPDATING BOMBS

If you have ever written a version of *Space Invaders* or the like in BASIC, the chances are that the program was reasonably fast until the bombing routine was put in. The bombs always spoil the action in any BASIC arcade game as they take so long to update and check. Due to this, some people write games without bombs and instead limit the player to the number of shots he may use. This is a very poor substitute and leads to a boring game.

The obvious answer is to include a machine code routine to handle the bombing. All this does is search the screen for bombs, find them and move them on to the next position, first checking to see if they have crashed into something. If they have then this is flagged to the BASIC (or machine code) program. Once this is done, all that remains is to call the routine from BASIC with a single SYS call and check the contents of the flag to see if a bomb has struck. There are two ways in which the bombs can be moved. They can be kept on the screen and updated from there, or they can be kept in a display file and manoeuvred from there. It's up to you to decide which system to use. The simplest way is definitely the on-screen approach but this has the disadvantage that, if the bombs are accidentally overwritten, they are lost. In the on-screen system, all the BASIC program has to do is to POKE a bomb character on to the screen at the right position. The machine code will then pick it up and move it on from there. However, if you are scrolling the screen, the bombs will acquire a sliding effect as well (this is not necessarily a bad thing). The problem with the display file system is that, to eliminate a bomb, you must go through the hassle of subtracting one element from an array — a messy task. For details of the display file system, see the section on moving groups of objects (Fleet Movements, in Chapter 10).

We will assume (for simplicity) that your bombs are going to fall vertically (ie from top to bottom). Thus all the program has to do is to scan the screen for the relevant PEEK code, blank out the bomb at its present position and add 40, and plot it on again at the new position, first checking to see if it's going to hit something. If we think about it, there are several things it could hit. It could hit part of the enemy (ie its own side): this obviously doesn't matter and the bomb will just have to disappear. Or it could strike a protective hedge (like the ones in *Invaders*). If this is the case, then the normal practice is to erode the hedge and eliminate the bomb. The bomb could strike the bottom of the screen and, when this happens, the bomb has clearly finished its useful life. Thus we might as well destroy it. Some programmers like to have their bombs explode when this happens — this is just an extra refinement. The most important case is, of course, when the bomb strikes the player's ship. Differentiating between players' ships and other paraphernalia on the screen is largely up to the programmer. In

Space Invaders, one technique is to check only for a bomb-ship collision when the bomb is near the bottom of the screen. The only thing the bomb could hit on the line where the ship operates is the ship itself, and so all the program has to do is check for a collision when the bomb crosses this line. For more involved situations, the easiest method might prove to be just looking at the PEEK codes and comparing them with those of the ship and the other items.

This then gives the general idea of moving the bombs, but one or two finer points arise out of it. Firstly you must scan the screen in the opposite direction to that of bomb movement (this idea is met in scrolling as well) as otherwise you will pick up a bomb and move it on, only to pick it up again a fraction of an instant later (and move it on again). Of course this results in the bombs moving straight to the base of the screen in one scan. Secondly, don't forget to reset the collision flag. How this is done is important. You may want the bombs in the BASIC program to move three times as fast as everything else. This can be achieved by using:

```
SYS 826 : SYS 826 : SYS 826
```

(826 is the assumed address of the routine.) If this is the case, then it's no use the machine code routine resetting the flag: the flag might be lost if a collision occurred in the first one or two passes. In this instance it's the caller that must maintain the status of the flag, but bear in mind this is not always the case.

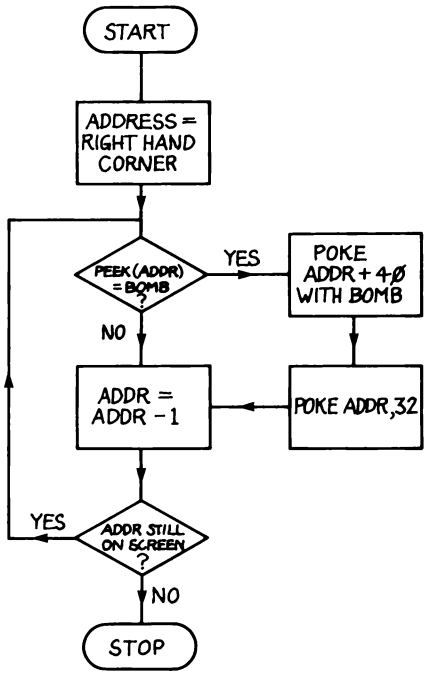
We are now in a position to set down the various stages involved in the bomb-updating process:

1) Reset the collision flag (may be omitted).
2) Scan screen in opposite sense to bomb motion.
3) If bomb has hit something other than the player's property, then apply rules of the game to the situation.
4) If the bomb has hit the player's ship, then set the collision flag.
5) Repeat (2) onwards until the screen has been scanned.

If the bomb does hit something then the programmer might want to trigger an explosion by tripping a gate in the SID.

To alter the direction of the bombs all you have to do is to alter the bomb vector, but bear in mind that if you make the bombs move into progressively lower memory locations then you will have to reverse the direction of the scan. To liven things up a bit you might like to try a 'spiralling' bomb routine where the bombs change shape as they fall. The best way to do this is to alter the UDG definition from one shape to another, every time the routine is called. Using this method I have

Figure 9.4: Bomb Update.



created bombs with teeth that gnash together as they fall. Ambitious programmers might like to add suitable sounds every time the bombs mutate. 'Homing' bombs can be produced for those games where the going gets really tough by taking each bomb in question and comparing chunks of its address with the 'target's' address. The art of doing this is discussed under Non-linear Motion, in Chapter 10. If you are operating this sort of homing bomb, however, you are going to have to establish a system for distinguishing between an updated bomb and one which hasn't been updated. One solution is to change the bombs from one character into another and then change back again for the next scan. It was in this way that I discovered the idea of spiralling bombs. The bomb routine demands no parameters: all you have to do is to call it and check the collision flag.

Fundamental Bomb Update routine

```
:LL0    LDA #191
        STA 251
        LDA #8      } Start at the base
        STA 252
        LDY #0
```

```
:LL1    LDA (251),Y     Get character
        CMP #36         Is it a bomb?
        BNE LL2
        LDY #40         Move down one line (+40 vector)
        STA (251),Y     Put it back
        LDY #0
        LDA #32
        STA (251),Y     Destroy the image
:LL2    LDA 251
        SEC
        SBC #1
        STA 251         Work backwards
        LDA 252
        SBC #0
        STA 252
        CMP #3
        BNE LL1         Done?
        RTS
```

This routine scans the entire screen and moves all bombs ($ signs) down the screen. If you want to move them somewhere else, simply assign the Y register a different value where shown. If you are interested in collisions, then insert a few instructions to PEEK the new locations before the bomb is moved in and set the collision flag accordingly. Anything other than '$' signs remain untouched.

The interrupt mixes well with this routine. Just plot the bomb on the screen and the machine will do the rest for you. (It can also be quite amusing to watch somebody trying to type a '$' sign on the screen when this is hooked on to the interrupt — every time they type it it falls to the bottom of the screen!) The following program shows how potent this routine can be even in BASIC. The POKE 53281,0 is merely there to change the colour of the background (otherwise the bombs won't show up). Type the program in and type RUN: this will load in the machine code. Then type RUN 1000 to see the routine in action.

Hail of Barbs

```
5    REM ** HAIL OF BARBS
10   DATA 169,191,133,251,169,7,133,252
20   DATA 160,0,177,251,201,36,208,10
30   DATA 160,40,145,251,160,0,169,32
40   DATA 145,251,165,251,56,233,1,133
50   DATA 251,165,252,233,0,133,252,201,3,208,221,96
60   P = 820
70   READ D : POKE P,D : P=P+1 : GOTO 70
```

```
1000  PRINT "[CLS]" : POKE 53281,0
1010  POKE 1024+INT(40*RND(1)),36 : SYS 830 : GOTO 1010
```

SCROLLING

Many games now available give the player a sense of movement by scrolling the screen in various directions. What this means is that the player isn't limited to playing in an area the size of the screen. The games also look much more attractive as there is more going on and therefore more variety. A SCROLL routine in BASIC is so slow that it takes several seconds. Clearly, then, SCROLL routines must be written in machine code, with the one exception of the automatic scroll executed by the PRINT staement when the bottom of the screen has been reached. The use of this facility is demonstrated in the program 'Asteroyds'. This can easily be accessed in machine code, either by calling the ROM routine directly or by sending a line-feed character to the print-character routine.

Asteroyds

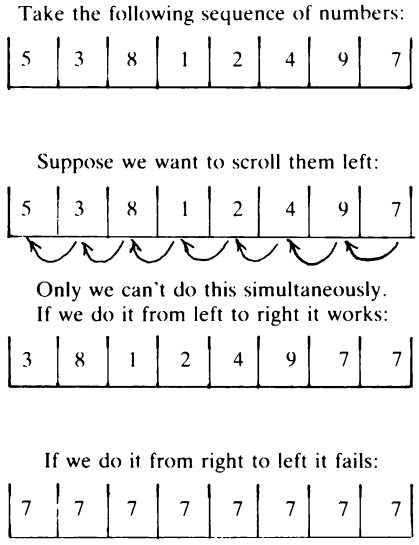
```
10 REM ASTEROYDS
20 DIM A$(4,4),A(4)
30 A$(1,1)="●"
40 A$(2,2)="╱╲"
50 A$(2,1)="╲╱"
60 A$(3,2)="╱‾╲"
70 A$(3,1)="╲_╱"
80 A$(4,3)="╱‾‾╲"
90 A$(4,2)="▌    ▐"
100 A$(4,1)="╲__╱"
110 A(1)=1:A(2)=2:A(3)=2:A(4)=3
120 IF AP=0THENGOSUB1000
130 PRINTTAB(P) A$(AN,AP):AP=AP-1:GOTO12
    0
1000 AN=INT(4*RND(1)+1):AP=A(AN)
1010 P=INT(33*RND(1)+1):RETURN
```

Despite what I said about BASIC's appalling lack of speed, in this instance it will serve us very well to show you the fundamental idea behind scrolling. To scroll to the left all you have to do is to copy each character into the character position to its left (see **Figure 9.5**). Type in the following program, clear the screen and draw a picture using the

graphics. Now RUN the program and watch. The picture is gradually shifted to the left. Here's the program:

```
10 FOR M=1024 TO 2022
20 POKE M,PEEK(M+1)
30 NEXT M
```

Figure 9.5: Scanning Directions

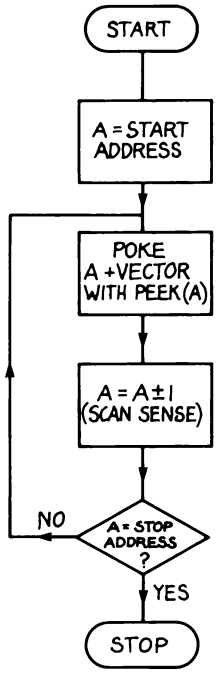


The '7' is moved all the way along.
Thus: SCAN IN THE OPPOSITE SENSE OF THE SCROLL

Now this little program brings two other vital facts to light. First, if something is on the extreme left of the screen (like your RUN statement) then it suffers 'wrap-around' and moves on to the extreme right of the screen. This will clearly be true for any scroll, it just depends on the direction. The other important fact that we can glean from the program is that you must 'scan' in the opposite direction to the scroll. We were scrolling to the left and the FOR–NEXT loop was scanning to the right. If you scan in the same direction as the scroll then the first piece of data is repeated throughout the area — try it and see. This is why a normal COPY routine won't work for all SCROLL directions.

Summarising then, a scroll is simply a localised copy (ie the destination is not far away), the copying must be done in the opposite direction to the movement and screen extremities may have to be cleared to prevent wrap-around.

Figure 9.6: Scrolling.



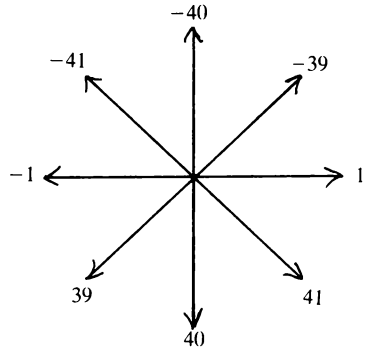
There are basically four scroll directions — up, down, left and right. These combine to produce the standard eight directions (**Figure 9.7**). For each of these directions, we can specify a number that represents in relative terms the position to which data is copied. For instance, when scrolling to the right this number is one — each memory location is copied into its address plus one. Scrolling left has the value minus one. So if this number is negative then the area must be scanned from left to right, and if it's positive it must be scanned from right to left (in descending memory order). These numbers are all shown in the diagram (although you should be totally familiar with them).

Putting together all these ideas, the algorithm becomes quite simply the following method:

1) M=PEEK (L)
2) POKE L+V,M
3) L=L+D
4) If not finished then (1)
5) RETURN

Where the following letters mean:

Figure 9.7: Scroll Vectors.



For vector<0 scan left to right
For vector>0 scan right to left

M: Temporary copying register.
L: Present position of scan.
D: Scan direction.
V: Scroll value.

To cover every direction you must have two routines — one scanning each way. Parameters to be passed to the routines are the start and stop addresses and the scroll value. Resist the temptation to scroll the entire screen as otherwise you won't have anywhere stable to plot things such as the score and number of lives left. One scroll technique which I am particularly fond of is the ditty shown in the following routine. It scrolls only 256 bytes (maximum), but is handy for scrolling messages across the screen or just generally moving things about in restricted areas. The VIC-II chip happens to support smooth scrolling. What this means is that in hi-res mode it will scroll the screen *bit by bit* as opposed to character by character. For details of how to achieve this see the *Programmer's Reference Guide.* You will still need a supporting character-scroll routine as the bit-scrolling routine only works for eight bits in any one direction. After that you must move all the memory on via the scroll routine given here and then you may return to the bit-scrolling routine. However, manipulating graphics in hi-res mode is a complicated procedure and so should be left until you have gained some experience.

256 Byte Continuous Scroll

```
:LL0        LDX #0
:LL1        INX
```

```
        LDA ADDR,X
        DEX
        STA ADDR,X
        INX
        BNE LL1
        RTS
```

This routine scrolls 256 bytes to the left, starting at ADDR. They are scrolled in a continuous loop so that everything goes round and round.

```
:LL0    LDX #0
:LL1    DEX
        LDA ADDR,X
        INX
        STA ADDR,X
        DEX
        BNE LL1
        RTS
```

This routine does the same, but to the right. Changing the LDX #0 to any other value will reduce the number of bytes scrolled. This will, however, remove the 'endless belt' effect. These scrolls are simple and quick — it's surprising what can be achieved with the X and Y registers.

Scroll into Lower Memory

The following routine performs a scroll into lower memory locations. Thus each byte is moved to a lower address, and so the area is scanned in the opposite direction — in increasing memory order. Each byte is moved into the next-lower address, resulting in an apparent movement to the left if watched on the screen. The symbols 'LO-START', 'HI-START', 'LO-STOP' and 'HI-STOP' refer to the lo and hi bytes of the start and stop addresses.

```
:LL0    LDA #LO-START   ⎫
        STA 251         ⎪
        LDA #HI-START   ⎬  Set up
        STA 252         ⎪
        LDY #0          ⎭
:LL1    INY             ⎫
        LDA (251),Y     ⎪
        DEY             ⎬  Scroll
        STA (251),Y     ⎪
        INC 251         ⎭
```

```
                BNE LL2
                INC 252
:LL2            LDA 251
                CMP #LO-STOP
                BNE LL1
                LDA 252          } Finished?
                CMP #HI-STOP
                BNE LL1
                RTS
```

The drawback with this routine is that characters on the lefthand edge of the screen suffer wrap-around. The solution is to clear this line before the scroll can take place. If this is done, then there is nothing to wrap-around. This routine should be called *before* the scroll or the lefthand column will appear ugly. The routine could be improved by using the X register to count the number of lines blanked, thus eliminating the need for the compare instructions. START and STOP are the addresses of the top and bottom lefthand corners of the screen respectively.

Line Blank routine

```
:LL0            LDA #LO-START
                STA 251
                LDA #HI-START    } Set up
                STA 252
                LDY #0
:LL1            LDA #32
                STA (251),Y      Blank out
                CLC
                LDA 251
                ADC #40          Move 1 line down (+40)
                STA 251
                LDA 252
                ADC #0
                STA 252
                LDA 251
                CMP #LO-STOP
                BNE LL1
                LDA 252
                CMP #HI-STOP
                BNE LL1
                RTS
```

To use these routines to scroll the entire screen left, use the following values for the symbols. If they are called one after the other, then the

total duration is still less than a twentieth of a second.

LINE BLANK : START = 1024, STOP = 1984
SCROLL : START = 1024, STOP = 2023

The interrupt can be used to great effect with the scroll. However, if the synchronisation is not correct, then the result will be a mess. I tend to hook-up only smaller scrolls on to the interrupt as they are often easier to handle. This sort of thing might be used to create an endless belt of fruit (or meteor debris) that is constantly in motion around a certain part of the screen. Scrolling is fun!

INTERACTION

Keyboard entry

In BASIC, the standard way of entering data is via the INPUT or GET

Table 9.1: Location 197 Keyboard Code Table.

No key depressed		64	
Shift/Commodore key have NO EFFECT			
SPACE		60	
RETURN		1	
F1		4	
F3		5	
F5		6	
F7		3	
','	(Move left)	47	Keys I use for movement
'.'	(Move right)	44	
'Q'	(Move up)	62	
'A'	(Move down)	10	
'1'		56	
'2'		59	
'3'		8	
'4'		11	
'5'		16	
'6'		19	
'7'		24	
'8'		27	
'9'		32	
'0'		0	

There is a relation between the ASCII code and the value but it's complicated, and far easier to know what's what.

statement. In machine code, these facilities are not easily available.

The easiest way to create machine-player interaction is to utilise the keyboard scanner that runs automatically on the interrupt (make sure that you have not disabled this). Every sixtieth of a second (during each interrupt) the keyboard is polled to check for any key depressions since the last check. If a key is being held down during this polling, then a value is stored in location 197 according to **Table 9.1**. If a new key has been pressed since the previous scan, then it's recorded in the buffer, which records the last ten keys pressed.

This buffer starts at location 631 and continues to 640: the present size of the buffer (ie how many characters it is holding) can be found at location 198 — if the buffer is empty, then this will contain a zero. However, as this book is describing the design of games, the programmer is far better off PEEKing location 197 as this tells him if a key *is* being held down. PEEKing the buffer will only tell you what keys *have* been pressed — whether this was a moment ago or last year. Don't forget that this buffer will soon fill up in the course of a game (as it only takes ten characters) and so, if you return to BASIC, be sure to empty it, either by this method:

```
FOR A=1 TO 10 : GET A$ : NEXT A
```

or perhaps the more elegant way:

```
POKE 198,0
```

This resets the buffer pointer.

Joysticks

Joysticks certainly allow games to become more exciting but don't just write games for joysticks or the keyboard alone — you don't know who's going to end up playing the game. (Paddles are so rare it's not worth writing them in, and as for light-pens, forget it.)

The two joystick ports reside at 56320 and 56321. Only the lower five bits are of any interest to us. Bit 4 is connected to the fire button so, to test for this state, simply use AND #16 followed by a BEQ or BNE. Bits 0–3 are concerned with the stick position. Read the port, AND #15 and look up the value obtained in **Table 9.2**. If the direction is stored as a vector, then it's a simple matter to add it into the address of your ship or whatever.

The value 255 subtracts one if there is no carry. The following routine moves sprite 0 around the screen via joystick signals and looks up the vector in this table:

[0,0,0,0,0,0,0,0,0,0,1,1,1,255,1,0,0,0,255,1,255,255,255,0,0,
0,0,1,0,255,0,0]

Table 9.2: Vector Table.

VALUE OBTAINED	DIRECTION	X VECTOR	Y VECTOR
0–4	NIL	.	.
5	Left+Down	1	1
6	Right+Up	1	255
7	Right	1	0
8	NIL	.	.
9	Left+Down	255	1
10	Left+Up	255	255
11	Left	255	0
12	NIL	.	.
13	Down	0	1
14	Up	0	255
15	NIL		

This routine, plus the sprite homing routine (see Non-linear Motion, Chapter 10) and a few rules, makes a simple but satisfying game. Try it and see; you'll be surprised how easily they work together. Anyway the following routine has no branches and runs straight through.

```
:LL0    LDA 56321       Get joystick position
        AND #15         Mask out rubbish
        ASLA            Multiply by 2
        TAX
        LDA TABLE,X     Look-up table
        CLC
        ADC 53248
        STA 53248
        LDA TABLE+1,X   Update X
        CLC
        ADC 53249
        STA 53249       Update Y
        RTS
```

You could modify this routine to run on the interrupt and move the cursor around the screen, to simplify the task of editing your programs. If you find that a one-pixel movement is too slow for what you had in mind, change ones to twos and 255s to 254s in the table, or just call the routine twice.

INVERTING AND EXPLOSIONS

Sooner or later in your game, you will want to be able to pretend (it's all make-believe) that the player's ship is exploding or that the doomsday bomb has just exploded. It's on this sort of occasion that you really want the whole screen to go up in a big firework display: for a few seconds the screen must totally freak out.

One way of achieving this is to 'invert' the screen. What is meant by this is that every pixel is examined, and if it's on, it's turned off, and vice versa. If this is done rapidly, the effect can be quite mesmerising (so don't overdo it) and is ideal for limiting the player's view of things when, for instance, he is flying through some asteroid debris. On top of this, if the border and background colours are switched fairly quickly, then this all adds to the merriment. The colour locations are 53280 and 53281 (screen and border): choose a colour number from 0 to 15.

The process of inverting the screen is a simple one (assuming you are in character mode) as any character in reverse field is simply that character inverted — look and see. Another added bonus from Commodore is that, to convert a number from normal to reverse field, all you have to do is add 128. Thus you can toggle to and fro by Exclusive ORing with 128. To bring the screen back to normal, just ensure that an even number of inversions has been performed. Problems will obviously arise if you are using UDGs or hi-res graphics, but these problems can be overcome. An inversion in hi-res, even in machine code, may appear sluggish, so the best bet here is to use the screen and border colour. Don't forget that the screen is only updated 60 times a second, so don't try anything too fast and, if you want to minimise 'snow', wait until a refresh signal arrives before inverting. The process of performing an inversion is thus surprisingly simple — just scan the screen and EOR every location with 128.

Other possibilities lie in flashing a line or a zone on the screen to attract the player's attention. To convert the supplied routine to do this, all you have to do is alter the start and stop addresses. An algorithm for this problem might look like:

```
1) I=PEEK (C)
2) I=I EOR 128
3) POKE C, I
4) C=C+1
5) IF C<=S THEN GO TO (1)
6) RETURN
```

Where C is the start address and S is the stop address. Nothing is returned in the way of parameters.

Inverting

The following routine inverts the entire screen once. If you only want one section inverted, then the answer is to change the start and stop addresses. If you EOR with a number other than 128, all sorts of things happen. The screen sudenly turns into a meaningless mess: as soon as the routine is called again, sanity is restored.

```
:LL0    LDA #0          Lo-start address
        TAX             ie LDX #0
        STA 251
        LDA #4          Hi-start
        STA 252
:LL1    LDA (251,X)     Get character
        EOR #128        Invert it
        STA (251,X)     Put it back
        INC 251         Move on
        BNE LL2
        INC 252
:LL2    LDA #7
        CMP #252        Hi-stop address
        BNE LL1         Finished?
        LDA #232
        CMP 251         Lo-stop address
        BNE LL1         Finished?
        RTS
```

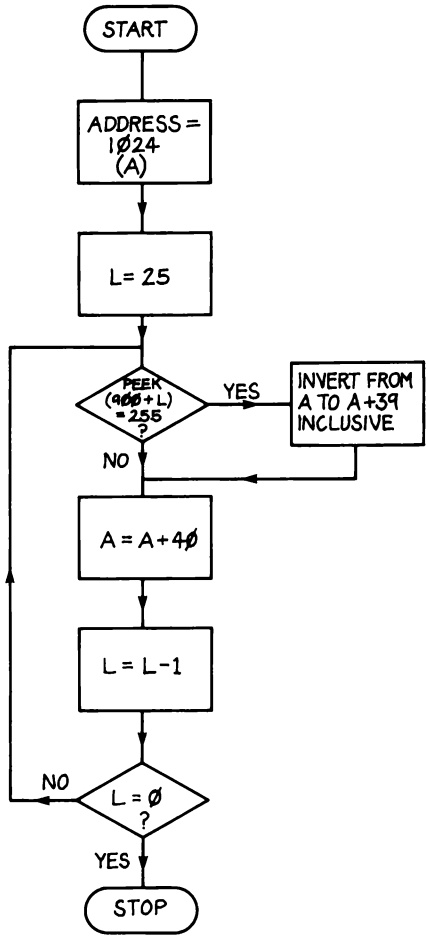
Hooking this up to the interrupt is amusing, to say the least, but if you want to invert something regularly, you really only want to have to write a line or two. The following routine does just that — it inverts 40 consecutive characters starting at START.

```
:LL0    LDX #0
:LL1    LDA START,X     Get it
        EOR #128        Invert it
        STA START,X     Put it back
        INX             Move on
        CPX #40         Finished?
        BNE LL1
        RTS
```

In fact it would be easier to start with x = 40 and then count down to

zero, eliminating the need for the compare. Now, if we had a routine like this for every line on the screen, with an option as to whether it flashed or not, and the whole thing hooked on to the interrupt, then we could have a flashing attribute system similar to the Spectrum and BBC machines (**Figure 9.8**)! The following routine does just that.

Figure 9.8: Attribute System.



Attribute Flasher

```
:LL0    DEC 254          Decrement interrupt counter
        BNE LL1          Have we counted 60?
        LDA #60          Yes, so flash & reset counter
        STA 254
        LDA #255
        STA 251
        LDA #3           Start at top of screen
        STA 252
        LDX #25          Do all 25 lines
:LL3    LDY #40          Each line is 40 characters
        LDA TABLE,X
        CMP #255         Flash this line?
        BNE LL4
:LL2    LDA (251),Y
        EOR #128
        STA (251),Y      Yes, so do so
        DEY
        BNE LL2
:LL4    CLC
        LDA 251
        ADC #40
        STA 251          Move on
        LDA 252
        ADC #0
        STA 252
        DEX
        BNE LL3          All done?
:LL1    RTS              Yes
```

Location 254 counts down from 60 to 0, decrementing each interrupt: thus there is one flash a second (in fact counting from 59 downwards produces a second's delay). When location 254 reaches 0, it scans the attribute table and, where it finds a 255 entry, inverts that line. The table is in reverse order. That is, the first entry corresponds to the 25th line and TABLE+1 corresponds to the 24th line and so on. The interrupt connection has not been included — it is simply a JMP to the start of the IRQ service routine.

AN ALTERNATIVE SPRITE SYSTEM

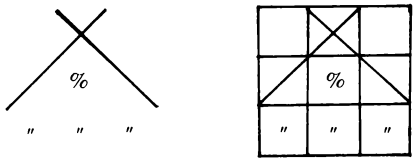
The resident sprite system moves objects around quite independently of

the screen content. A routine is presented here which manipulates character blocks around the screen, and which may be useful for moving large objects around. There is no limit to the size of the object or to the number of objects: The only restraint is that there is only one type of object (ie description).

The objects are described by a two-dimensional table. The first entry contains the POKE code of that element and the second contains the connection vector between that character and the next. **Figure 9.9** shows a typical symbol and the corresponding entries. This table is stored at addresses shown in the routine and, to use it, you POKE the address into the page-zero locations and call the routine. To delete the symbol you must set the delete flag as well, before calling. This flag is cleared on use. To move an object, then, the stages are as follows:

1) Delete the present location.
2) Add the vector to the address, thus moving the object.
3) Recall the routine.

Figure 9.9: Encoding a Sprite.



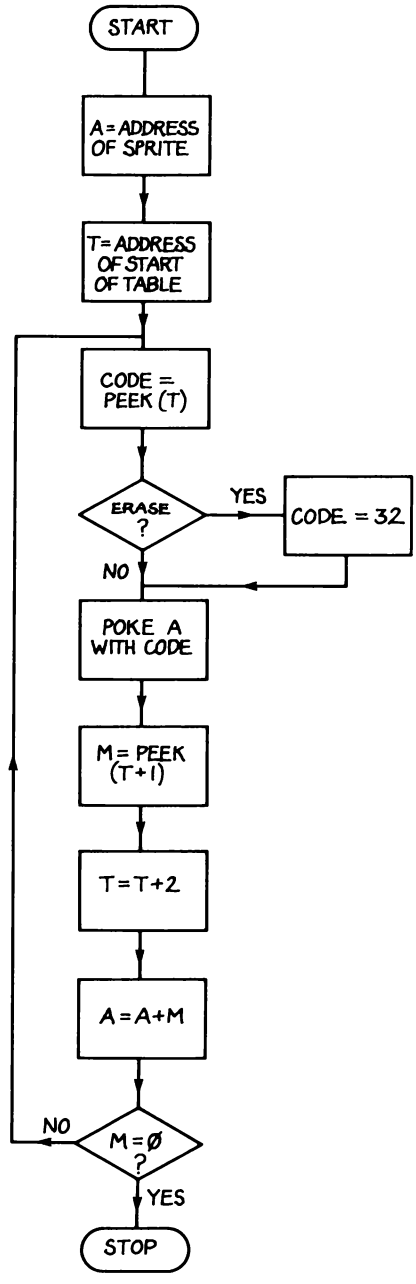
CODE	VECTOR	
86	39	' × '
78	1	' / '
37	1	' % '
77	38	' / '
34	1	' " '
34	1	' " '
0	—	' null entry'

1. Draw symbol
2. Lay grid over
3. Draw up table of codes & vectors
4. Poke tables into memory in the form: code, vector, code, vector . . . null eg [86,39,78,1,37,1,77,38,34,1,34,1,0]

Using this to move a fleet of objects means using a routine that performs the above steps in turn for each individual ship in the fleet. See the section on Fleet Movements (Chapter 10) for details of this process. More than one type of sprite can be used by this routine if you change the table addresses in between calls, but the chances are that this won't be necessary (**Figure 9.10**).

The following routine allows the manipulation of sprite-like objects as outlined. TABLE is the start address of the table and the end of the definition is marked by a character value of 0. The definition is limited to a maximum of 127 entries (this should be plenty).

Figure 9.10: Alternative Sprite System.



```
:LL0    LDA 253
        STA 251
        LDA 254          Copy address in
        STA 252
        LDX #0           Symbol table index
        LDY #0
        LDA TABLE,X
:LL2    CPY 2
        BEQ LL3          Delete symbol?
        LDA #32          Yes
:LL3    STA (251),Y      Plot/erase
        INX
        CLC
        LDA 251
        ADC TABLE,X      Move on to next character
        STA 251
        LDA 252
        ADC #0
        STA 252
        INX
        LDA TABLE,X
        BNE LL2          Finished?
        RTS
```

The address of the first character of the sprite is sent in locations 253 and 254. This is not altered on return, and so the routine can be repeatedly called without updating this. The PLOT/ERASE option is made by location 2. If this is anything other than a 0, the character is erased. However, on returning, location 2 is always reset to 0. Don't forget that this could be hooked on to the interrupt to maintain a certain picture on the screen or better still a vector system (see Sprites, Chapter 10). This routine is handy for plotting awkwardly-shaped objects when a lot have to be plotted (see Fleet Movements, Chapter 10).

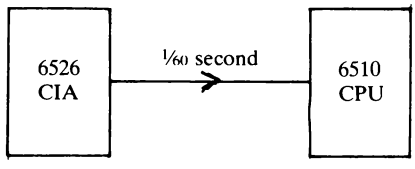
CHAPTER 10
Indirect Routines

THE INTERRUPT

What happens when the telephone rings? You stop what you are doing, pick it up, answer it and then resume what you were doing before. You have been 'interrupted'.

The same sort of thing is true with computers. They have control lines leading to the CPU which can generate an interrupt (**Figure 10.1**). When an interrupt is called, the computer stores the exact details of where it was and what it was doing on the stack. The interrupt vector contains the address of a routine and the machine jumps to this routine. The end of the routine contains an RTI — similar to RTS but meaning ReTurn from Interrupt. The activity of an interrupt routine is totally transparent to a normal program. In fact it hapens 60 times a second in BASIC alone!

Figure 10.1: CIA#1 Sends IRQ Signals to Check the Keys.

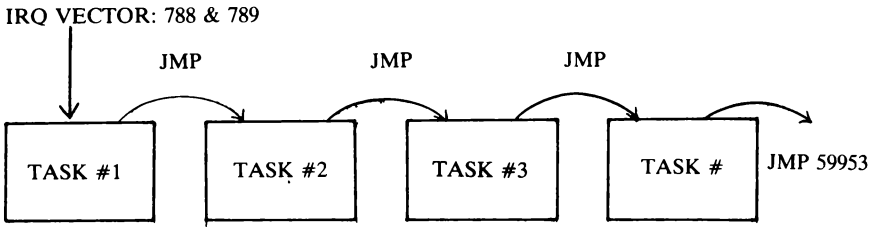


Every sixtieth of a second, the IRQ (Interrupt ReQuest) line generates an interrupt. The routine which services it makes the cursor flash and reads the keyboard to see what keys you are pressing (as well as performing other tasks). The address of this routine is held in 788 and 789 so, if you want to point it to your own routine, you can. One problem: if you do this the keyboard will become inactive, you will lose the cursor, and the clock will stop — your housekeeping routines have been swapped for your routine. Counter this by placing at the end of your routine not an RTI but a JMP 59953. This is the address of the start of the interrupt service routine and, of course, there's an RTI at the end of this.

In fact there are three different interrupts available. First, there's the IRQ which we have already discussed. Second, the NMI, and third the BRK. The NMI is very similar to the IRQ but will be ignored here. The BRK is an instruction itself and is very useful for debugging purposes (see Chapter 7 on testing and debugging). The rest of this section will be devoted to the IRQ.

This may seem all very well but you may be asking what advantage an interrupt system has over normal programming style. The answer is that it allows perfection of timing and synchronisation, allied with the beauty of being able to hook something on to the interrupt and then forget all about it. I talk of 'hooking' things on, as often the interrupt performs several tasks all 'chained' together (see **Figure 10.2**). It's easy to 'hook on' another task to the chain. So what sort of things can we do with the interrupt? The answer is that almost anything can be done with it.

Figure 10.2: Multiple Routines on the Interrupt.



For instance, you may want a particular event to occur every now and again: this might be playing successive notes in a tune or updating enemy ships on the screen. If this is the case, then a simple counting routine will initiate whatever you want after a certain time. Suppose for instance that to add atmosphere to your submarine game you want the sonar to make a 'blip' every four seconds. As the interrupt occurs 60 times a second, it must count 4 × 60 = 240. If we keep the count in location 254, then the program might look like this:

```
        DEC 254
        BNE LL1
        [Trip gate on sonar sound]
        LDA #240
        STA 254
:LL1    RTS
```

The interrupt can be used to do anything. The number of interrupts generated per second is also under user control. See the CIA section in Appendix C for details of this. Many of the routines in this book are designed to be interrupt-driven. These include sprite vectoring and programming the function keys.

Used properly, the interrupt is the key to the 64. The VIC-II chip has the capability to generate IRQs: sprite collisions are just one of several possible sources. Once interrupt routines are hooked on they can be forgotten about, making programming very simple. It's a bit strange at first, but you'll get used to it and be glad to have made the effort.

Hooking on

This describes briefly the process of hooking a program or routine on to the interrupt. It assumes that only one routine is to be added. If this is not the case, then change the JMP 59953 for the address of another routine to be hooked on.

1) Write the routine and place an RTS at the end.
2) Test it by using a BASIC loop to call it continually. This way the RUN/STOP RESTORE trick still works.
3) Replace the RTS with a JMP 59953.
4) Re-point the IRQ vector with one line of BASIC like this: POKE 788,LO : POKE 789,HI

If all is well, your routine is now smoothly operating 60 times a second.

SPRITES

Before the era of sprites, moving objects smoothly around the screen was a daunting task. The 64 is lucky enough to own a little family of eight sprites. With very little effort, they will perform for us, making the entire task of game writing much simpler. They exist on a plane either above or below the screen so that they don't actually move on the screen itself — they appear to glide over and under it. When they do cross each other, or something on the screen, we are informed by an entry in the sprite collision logs. Using sprites means that you can draw a detailed background and not worry about it becoming corrupted.

Initial defining can be speeded up through the use of a COPY or FILL routine. Full sprite manipulation details are available in the manual, and a handy chart is included in the Appendix to this book. Of special interest are the 'control' registers. These are one-byte registers with one bit allotted to each sprite (bit 3=sprite 3). These registers control which sprites actually appear and which sprites are expanded, and also record collisions amongst other things. See the SID and VIC memory maps in Appendix C for details. As the location of a sprite is recorded via a cartesian (X,Y) coordinate pair, homing becomes a very simple matter (see Non-linear Motion later in this chapter). To allow full span screen coverage, the X coordinate has its MSB in a separate control register. I

don't like using this as it complicates the programs, and also if it's not used there is a safe area on the screen for the score, etc.

Every now and again a sprite collides with something else. When this happens it's recorded in the collision logs by setting the appropriate bits. These are automatically reset when read, so you may have to copy this data when you read it. The sprite-hits-sprite log is 53278 and the sprite-hits-background is 53279. Using these registers and the enable register, it's easy to disable sprites that have collided. Suppose we want to remove all sprite-hits-sprite collisions. We do this by PEEKing the collision register. Changing all the ones to zeros and vice versa, and then ANDing the result into the enable register:

```
LDA 53278
EOR #255
AND 53269
STA 53269
```

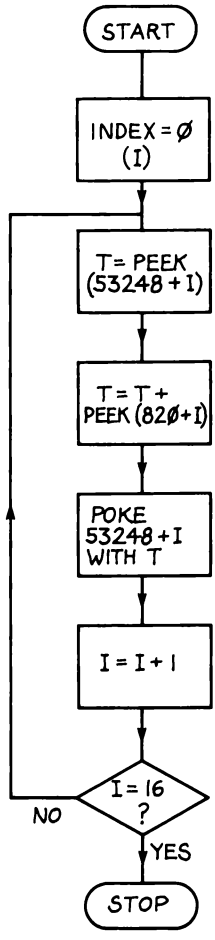
The removal of both types of collision is achieved by first ORing the collision registers together before the EOR instruction. Thus the maintenance of sprites is simple.

Only one thing really remains, the ability to set a sprite in motion and then make it keep on going of its own accord. This facility would be of enormous help in BASIC — a fast arcade 'blast everything that moves' program becomes possible. The routine that performs this feat is one of my favourite ones — it wasted an entire afternoon, playing around with different speeds and shapes of sprites (**Figure 10.3**). In 18 bytes, it is one of the shortest routines in the book:

```
:LL0    LDX #0
:LL1    LDA TABLE,X
        CLC
        ADC 53248,X
        STA 53248,X
        INX
        CPX #16
        BNE LL1
        RTS
```

A table of vectors is stored at TABLE to TABLE+15. For each sprite there is an X and Y vector. To effect backward movement, simply add the vector to 256 — eg −1 (go left) becomes 256+(−1) = 255. For really smooth motion, this should be hooked on to the interrupt, providing 60 updates a second. Try the following program on a friend. RUN it and, when the cursor returns, LIST the program — the sprites have been brought to life!

Figure 10.3: Perform Sprite Vectoring.




```
10  DATA 162,0,189,52,3,125,0,208,157,0,208,232,224,16,208,242
15  DATA 76,49,234
40  FOR P=49152 TO 49170 : READ D : POKE P,D : NEXT
50  FOR A=2040 TO 2047 : POKE A,15 : NEXT
60  FOR A=960 TO 1022 : POKE A,255 : NEXT
70  FOR A=820 TO 835 : POKE A,INT(4*RND(1)+1) : NEXT
80  POKE 788,0 : POKE 789,192 : POKE 53269,255
```

If you want to remove all collided sprites, then insert the removal routine between the BNE and after the RTS.

FURTHER USES OF UDGS

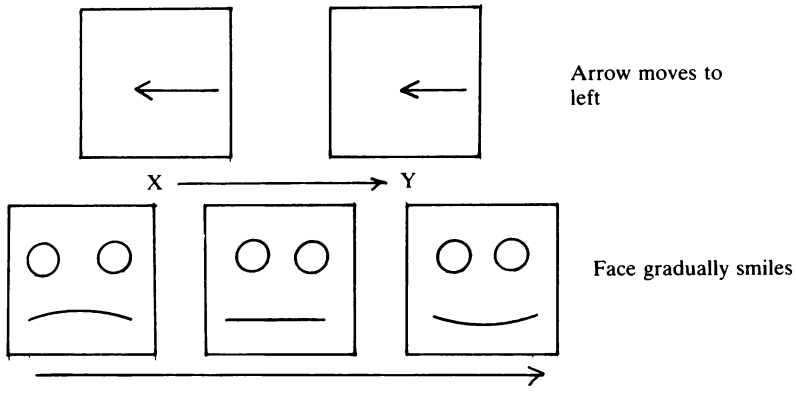
While UDGs are great for fine graphics, their application goes far deeper than just producing convincing aliens. Consider for a moment the function of a UDG: it allows you to create new characters in place of the old ones. Two things arise out of this: there is nothing to stop you from defining a pair of characters with the same definition, and there is nothing to prevent you altering the definition once it has been entered. Using these two ideas either together or singly, you can achieve very easily what previously verged on the impossible. If you don't fully understand UDGs, then read the section in Appendix A. Failing that, read the section in the *Programmer's Reference Guide* — it's vital that you understand fully the principles upon which the current section is based.

The idea of two identical graphics characters is not that new. Even before the advent of UDGs as a popular feature, the PETs possessed two identical characters. These were the space and the shifted space. Both appeared as a normal space but, in fact, reported PEEK codes of 32 and 96. This similarity presents no problem for the machine. Each PEEK code has one and only one corresponding bit matrix: the converse doesn't have to hold true.

This is all very well, but what is the use of it? A quick answer would be to say that it allows differentiation, which the player isn't able to make, between items. What this means is that you may want some aliens to be 'super-aliens'. Perhaps these are the only aliens that drop bombs. By creating look-alikes, they can be allowed to mix in with the rest of the gang. Using the shifted-space idea, it's possible to create invisible barriers. Checking for a crossing by address means alone is tedious — especially if there are several barriers. By employing invisible barriers the problem is eliminated. This idea can be extended even further: by surrounding an entity with an invisible boundary, you restrict movement within that area. So the look-alike UDG technique allows you to set up complex movement rules and character classes without giving too much away to the player. Despite all this, don't go to the extreme of creating invisible mines and other silly things — the player should have to survive by skill and not luck.

The principle of changing UDGs is yet more powerful, although a little harder to apply. The first use has its roots in the previous paragraph. By changing the definition of something that previously was a look-alike, you suddenly make it stand out. This idea could be used to give the player a quick glimpse of the opposition or even mark a development in the player's progression in a certain task. By performing an act, he is able to distinguish between one thing and another. This, however, shouldn't be attempted in BASIC, it really is far too slow: the transition from one form to another appears not instantaneously, but with a 'ripple' effect that cheapens the game.

Figure 10.4: UDG Switching and Matrix Transitions



The more advanced use of UDG-switching is that of changing the matrices so that an impression of motion appears. Suppose that character X has the pattern shown in **Figure 10.4** and is then transformed into character Y. Things have clearly moved to the left. Now suppose that there had been an entire line or even a whole screenful of character Xs. By altering the single matrix, you would have altered a multitude of parts on the screen simultaneously. The result? An impression of apparent motion. This removes the need for a scrolling routine at all (and looks smooth) if all you are after is an effect for a backdrop. Of course, this is simply the first and simplest application.

Consider the task of creating a screen display, simulating rushing through space in 3D. No simple scroll routine is going to help you here. Star debris must move slowly at the centre of the screen, accelerate, and be moving at a considerable rate when it gets to the edge. This task is not easy by normal methods. By applying UDG-switching techniques and employing a whole series of matrix transitions, the task becomes child's play — there is no messing about with differential screen movements.

What I have given here is a brief introduction to the world of UDG methods. The moral is to keep an eye open for the simple solution. The possibilities are endless and no doubt many of you will devise many more uses and tricks for the UDG: they are wonderful little things and shouldn't be kept out of the action when the going gets tough. It's wonderful to watch a player's face as the aliens mutate bit by bit into a deadlier foe (this could be achieved by BASIC). Furthermore, it won't affect your programs in the least so you might want to spice them up a bit. The same idea can also be used with sprites, except that one POKE is all that's necessary to change to an entirely new definition. Don't just use the UDGs and then forget about them for the rest of the game.

SOUND

Sound is a major element in virtually every available arcade-type game, as it gives the player a very satisfying feedback from what he is doing — it heightens the experience of the game. The traditional *Space Invaders* 'thump-thump' rhythm has been described as the heart-beat of the invaders. In fact, this noise works against the player as it is out of phase with the movement. As it does this, it upsets the player's rhythm and stops him from becoming totally synchronised with the game.

The 64 is particularly gifted in its sound, and possesses what is claimed to be the most flexible synthesiser available. Indeed it is a remarkable box but don't go all out to show off every little trick it can perform. The filters, for instance, have no real in-game value (or none that has been exploited). One of the nicest things is that you can set up a sound ADSR sequence so that, once you want to restart the sequence, all you have to do is trip the gate and forget about it.

There are essentially two ways of using sound in games: the real-time sound that occurs in a game to denote a gun-shot or explosion, and the annoying little ditty that the title page plays over and over again to attract attention. The subject of playing tunes will be discussed later, but for the time being the in-game sounds will be considered.

The device creating the sound on the 64 is known as the SID, for Sound Interface Device. This little chip is a computer in its own right. It will control up to three different noises simultaneously and allows very fine control of pitch, volume and colour. To use the SID effectively, you must know it well as at times it can be very uncooperative. The sound produced comes out of the television, or may be channelled into a hi-fi system through the monitor lead. Thus to eliminate the noise in a game, all the player has to do is to turn the TV or hi-fi sound down. This is very helpful for nocturnal users.

To use the SID, you need to set up a 'voice' with the appropriate values and then trip the gate whenever you want to hear it. Don't forget to POKE the volume control (54296) with 15 for maximum volume. There are three voices so this should allow three different sounds. If you want more, then you'll have to write a routine that enters the necessary values and then trips the gate. When I speak of 'tripping' the gate on a voice, what I mean is first of all to POKE the control register with a 0 and then re-POKE it with the waveform value. Merely POKEing with the waveform value is not enough. Thus use of the SID in a game is as follows:

1) Initialise the SID at the beginning of the game.
2) Simply trip the gate on the required voice when necessary.

It might be an idea to write the routine that trips each voice as a

subroutine (even though it's only a couple of bytes long) as this will save time. **Table 10.1** lists some possible values for various devices, but, for a full appreciation of the SID, refer to the *Programmer's Reference Guide* (especially to the section on the bell imitation).

Table 10.1: Some Sample Envelopes.

ADSR	WAVEFORM	SOUND	
0/0/240/0	17	ORGAN	
5/3/10/0	33	VIOLIN	
2/10/0/0	17	CHIME BAR	
5/5/0/0	129	STEAM ENGINE STROKE	Use hi frequency of 30 for these.
1/10/0/0	129	HEAVY GUN	
5/12/0/0	129	Explosion — Hi=10	

Produce phaser-like sound by making two sounds at very nearly the same frequency, thus obtaining 'beats'.

The following are obtained by using a lo-pass filter at resonance 15 and a voice hi-frequency of 30.

FILTER HI-FREQUENCY	SOUND
200	TORRENTIAL RAIN
140	LARGE FOREST FIRE
110	JET PLANE OVERHEAD

Playing a tune is fairly simple as long as only one voice is playing. As soon as you want chords and the like, the programming becomes complex. The best thing to do is to use one voice and, if you want the others to play as well, try making them mimic the first voice in a different waveform or several octaves higher: as I have suggested, the topic of using more than one voice with each playing a different part is beyond the scope of this section. It should prove a satisfying exercise for the reader.

To play a tune then, all that is required is a table of frequencies and durations (and a null entry for the last note). The procedure is simple — read a note and duration, set the frequency and trip the gate, wait for the prescribed duration and then move on to the next note. If you want to play an octave higher, then just double the frequency. In fact, you can multiply frequencies by any constant and the tune will still sound OK, but never try adding a constant as this will cause the tune to go sharp and flat in places. A program which plays a tune is listed and there is also some sample tune data. To play the tune all you need do is call the routine.

To sum up this section, sound in games is very easy. However working out the 'right' sounds for a game can be subjective. The best bet is to use a SID evaluation program (BASIC) and just toy around with

ideas until sounds begin to emerge. An evaluation program simply displays the voice characteristics, allows you to modify them and lets you hear what the sound is like. It can be short and simple, but very useful.

SID Evaluation

```
10  REM SID EVALUATION PROGRAM
20  POKE 54296,15 : POKE 54273,30
30  VC = [17,33,65,129]
40  INPUT "ADSR" ; A,D,S,R
50  POKE 54277,D+A*16
60  POKE 54278,R+S*16
70  POKE 54276,0 : POKE 54276,VC
80  GOTO 40
```

This program requests values for A, D, S and R, and then plays the sound generated by the envelope. Simply pressing RETURN (with no entry) leaves the envelope unchanged and plays the sound again.

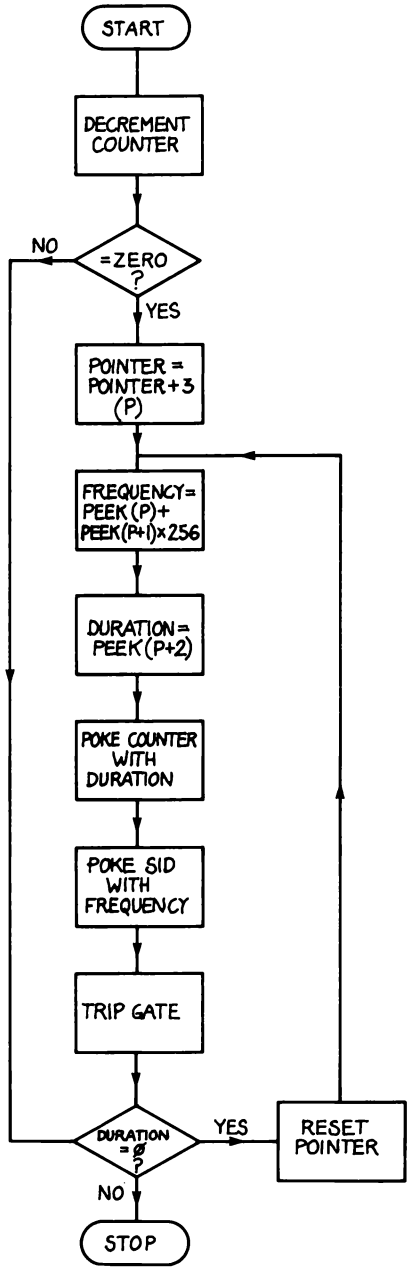
Key SID registers

Start of	Voice 1	54272
	Voice 2	54279
	Voice 3	54286
For each	Voice: (V=Start of that Voice)	
	V,V+1	Frequency
	V+2,V+3	Pulse waveform width
	V+4	Main control register
	V+5	Attack/decay
	V+6	Sustain/release
	Volume (0–15)	54296
	Random Nos.	54299

See Appendix C for a detailed chart.

The following routine plays a tune stored in a table. Each note is made up of three bytes — the lo and hi frequencies and the duration. If this is hooked-up to the interrupt (as it should be) then a duration of 60 means a note of length one second (**Figure 10.5**). The end of the tune is denoted by a duration of zero. When this is reached, the tune starts again: to avoid the end running into the beginning, place a delay at one end (sizeable duration, frequency 0). With this on the interrupt you can drive anyone crazy (so do so).

Figure 10.5: Interrupt-driven Tune Player.



```
:LL0        DEC 254              Decrement counter
            BNE LL1              Time for a new note?
            CLC                  Yes!
            LDA 251
            ADC #3               Move on in table
            STA 251
            LDA 252
            ADC #0
            STA 252
:LL2        LDY #0
            STY 54276            Cut sound
            LDA #[Waveform: 17,33,65,129]
            STA 54276            Trip gate
            LDA (251),Y
            STA 54272            Set lo-frequency
            INY
            LDA (251),Y
            STA 54273            Set hi-frequency
            INY
            LDA (251),Y
            STA 254              Set duration
            BNE LL1              Last note?
            LDA #LO-START        Yes, so repeat
            STA 251
            LDA #HI-START
            STA 252
            CLC
            BCC LL2
:LL1        RTS
```

HI and LO-START is simply the address of the start of the tune. For some 'state of the art' music, try running this program on totally random data — just like the real thing! For something a little more inspiring, sample the tune shown in **Table 10.2**.

LARGE-SCALE GAMES

Many games are fun on a normal-size screen, but by and by they tend to become somewhat limited by their physical size. If this is the largest playing area available, then it's difficult to hold the player's interest over a long period of time. There are, however, various ways round this problem.

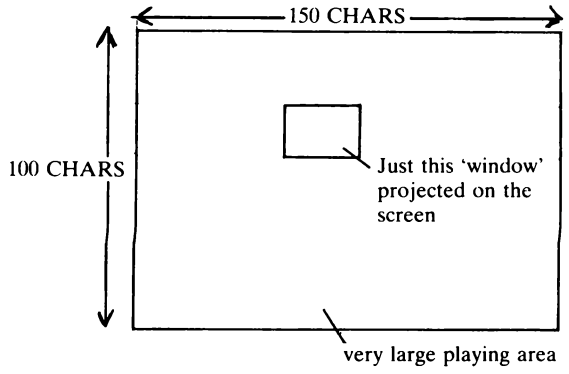
A very common method is to split the game up into several (or many)

Table 10.2: Sample Tune Data.
This data tries to resemble a tune. It is stored in the following order for the player program: lo frequency, hi frequency, duration. Duration is in seconds, so multiply by 60.

LO	HI	DURATION (multiply by 60)
96	22	1
49	28	1.5
19	28	0.5
156	26	1
96	22	1
223	29	1
223	29	1
49	28	1
156	26	1
49	28	1
135	33	1
135	33	1
165	31	1
135	33	3
49	28	1
162	37	1.5
135	33	0.5
223	29	0.5
49	28	1
156	26	1
96	22	1
31	21	1
49	28	1
156	26	1
96	22	1
96	22	1
31	22	1
96	22	3
0	0	255
0	0	0

separate screens. The player works his way through these screens in a predefined order, giving the impression that he is gradually travelling through the fantasy land. The other popular method is to make the screen act as a window on to a much larger playing area. This is done in *Defender* in one dimension and in 'Laserbike' (see Appendix D) in two dimensions. Games which supply the player with a view out of a window (**Figure 10.6**), such as in a space battle or racing track, are working in three dimensions. Working in three dimensions is very complicated and

Figure 10.6: The Screen Window.



only two-dimensional games will be considered here, but the possibility always exists.

Multi-screen games

The multi-screen game is easily achieved, the chief limiting factor being the amount of memory available. Each screen must be stored somewhere in memory, together with all the details of any 'gadgets' and special rules which apply on the screen. Each screen is brought on via a copying routine. Possibly several routines will have to be copied or acknowledged to maintain the game at this stage. What I mean is that one particular screen may demand a constant hail of bombs to drift across it. A dedicated routine would have to be written to take care of this. What this amounts to is that each screen consists of well over 1K of memory, even if it only occupies 20 lines of the screen. The rest of the memory is needed to perform housekeeping tasks. The memory limitation is not likely to be much of a problem, as it's difficult to think up more than a few original screens and gadgets anyway.

The method of using a window can be quite fun as, if you choose a particularly large playing area, the player can easily get lost. The first stage is to assign an area of memory to the 'world' on to which the window looks. This should be at least 2K. Then decide just how large the window should be. I consider that even half a screen is too much. In 'Laserbike' the player is limited to a 10 × 10 view of a 4K world. Next decide how you want the world to exist. The 'Laserbike' world is cylindrical — if you keep on going to the right then you loop back on yourself and re-appear on the left but one row down. This is because it is configured in memory like a collosal screen. If you keep on moving across you eventually meet the edge and wrap-around occurs. Of course, because we have the window view the 'edge' doesn't exist as

such. This effect doesn't hold true for the top and bottom: these are cordoned off with an uncrossable barrier. To eliminate the wrap-around, just draw a barrier vertically as well (this spoils the game slightly). Now you are in a position to write the routine to produce the window effect.

Let's see what information we have. We know how large the playing area is and this must factorise into two not too different numbers. Suppose the playing area is to be 80 × 80 bytes — a total of 6400 bytes. To see how this would be stored, think of it as a large screen. To go left and right, you would still add and subtract one, but moving up and down is now ±80 bytes. We know the size of the window, say 10 × 10 (these numbers don't have to be square). Now to draw the section of the world with the top lefthand corner having the address 8000, we must copy 8000–8009 to consecutive addresses starting from the screen address where we wish the window to be projected from. This, of course, is the address of the top lefthand corner of the window. To draw the next line we must go back to 8000 and add 80. This brings us one line down in the world. As before we now copy from 8080 to 8089 but starting at the window address + 40. We do this once for each line and the window is projected. **Figure 10.7** shows how this is done for a window of dimensions X,Y on a world W bytes across.

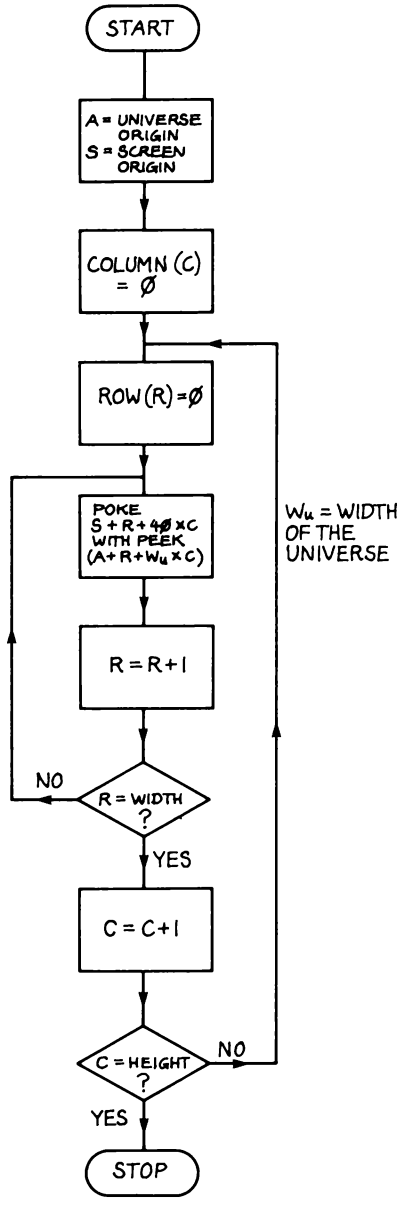
The program 'Laserbike' is based on the light-cycle race from *Tron*. When 'Laserbike' was first written in BASIC, a great delay was caused through clearing out the world between one round and the next. Of course filling a 4K area does take time. Thus the second routine was introduced and the game became satisfactory. Notice, however, that the world itself contains very little action: to coordinate action throughout a world of this size in BASIC would have ruined the game. This is why there is no enemy apart from the blazing trail you leave and the bombs.

The use of a window means that very ambitious games become possible. For instance, I am at present working on a game involving a hovercraft mounted with guns patrolling a sea/swamp/land zone with hazards such as minefields, gun batteries and tanks. A window routine is used to play the game over a wide area which should result in a game of fast action allied with skilful control of the hovercraft in difficult wind conditions.

2D Window Projector

The following routine produces on the screen a 'window' which has been copied from a larger area of memory. The address of the top lefthand corner of the window in the memory is stored at 251–252 and the address of the top lefthand corner on the screen is stored at 253–254.

Figure 10.7: Window Projection.



These do get altered during the routine and will therefore have to be reset. The width and height of the window and the 'width of the universe' are all variable.

```
:LL0    LDX #[Window Height]
:LL1    LDY #0
:LL2    LDA (251),Y          } Copy window
        STA (253),Y          }
        INY
        CPY #[Window Width]
        BNE LL2
        CLC
        LDA 251
        ADC #[Width of the Universe]   Next line
        STA 251
        LDA 252
        ADC #0
        STA 252
        CLC
        LDA 253
        ADC #40
        STA 253
        LDA 254
        ADC #0
        STA 254
        DEX
        BNE LL1                Finished?
        RTS
```

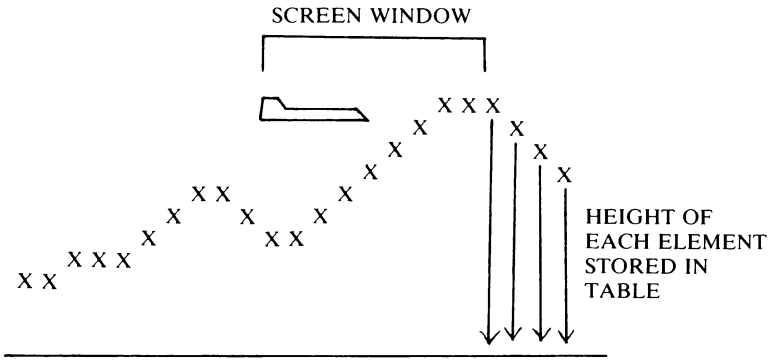
A *Defender* landscape window

The landscape in the game *Defender* is interesting in that it scrolls both ways. The player is restricted to a certain area — if he keeps on going to the right he will loop and come out on the left. The effect of this is that the game is played on an area many times larger than just the screen size itself. The question is how to create the landscape and know what is where as far as the game is concerned.

One solution is to store the landscape as a list of altitudes. If you choose a list 256 entries long, then you will get automatic wrap-around, and the playing area will have increased by a factor of 6.4 (see **Figure 10.8**). Each element on the picture can be thought of as being so many lines above the base of the screen. The entire landscape is listed as a series of heights which, when plotted like this, form a landscape. As you are using wrap-around, you will have to make sure that the two ends connect 'smoothly'.

Now look at the ship in Figure 10.8. Suppose that it is at position X on our landscape (0 <= X <= 255). As it's in the middle of the screen and we want to see the surrounding terrain, all we have to do is plot the

Figure 10.8: Projecting a Landscape.



values in the table in the range of, say, (X−15) to (X+15). To see if any hostile ships are going to be drawn just check their positions to see if they are in the range just described. This method makes the task of a *Defender* landscape very easy. The values are stored in a table 256 bytes long, a zero-page address points to the beginning of the table and, if you use the indexed X addressing mode, then you can have access to any entry in the table. On top of that, the wrap-around is automatic due to the crafty determination of the size. Let's look at a program to plot a landscape:

```
        TXA             Ship is at position X
        SBC #15         Plot to both sides of ship
        TAX
        LDY #30
:LOOP   LDA (TABLE,X)   Zero-page indirect mode
        JSR PLOT        Plot ship at coordinates
        INX               (Y,A)
        DEY
        BNE LOOP
        RTS
```

The program is almost self-explanatory. The table of altitudes starts at TABLE (zero-page vector). The routine is entered, with the X register containing the location of the ship. The program draws on 30 different parts of the terrain around the ship. The DEY could be replaced by a more complicated compare with the X register, but it's so much simpler for the plotting routine if it's given coordinates on the screen. The

plotting routine receives two pieces of data: the number of lines up to go from the base of the screen (A) and the number of lines to go across (Y). Simple arithmetic computes these positions and plots the necessary symbol. Don't forget to clear the screen first, though, or one landscape will overlay the next. When I first wrote this routine, it ran so fast that I could only see the landscape when I stopped the ship from moving. It's much faster than a normal scroll.

FLEET MOVEMENTS

There are few games in which the enemy doesn't consist of a number of similar aliens. (And those few games are normally fairly boring anyway.) Our problem is how to move around an entire fleet of ships in smooth formation such as in *Galaxian*. The answer is very simple. All you need is a routine to move one of them around and then use this routine to move all the clones around. After all, all the aliens are supposed to be identical anyway. If you think of an individual alien as a normal variable, then a fleet of them is simply an array. You can move them individually by simply altering any particular element, or you can affect them all at once. Don't shy away from using fleets in your games as the resulting one-ship game can get very boring indeed.

There are essentially two ways of producing fleet movement. Firstly there is the on-screen approach. This consists of having the aliens existing on the screen and using an update program to scan the screen and move them. The alternative is to keep all their positions and attributes in a table, and have a separate routine to plot them on to the screen. Each of these techniques has its advantages, and I shall discuss each in more detail.

The beauty of the on-screen method is that what exists on the screen is what exists in the game. If you blow something up with a bomb or a gun then, to remove it from the game, you must erase the image. If a table system is in use then erasure is not so simple: it is very infuriating to shoot down an alien, watch it explode and disappear, only to see it reincarnated the next second. The chief problem with the on-screen approach is that, if one of the ships is accidentally removed during some of the excitement, then the only record of its ever existing is in the player's mind, and causes a very poor sense of continuity. But the introduction of new matter on to the screen is very simple — you just plot it on wherever it is needed. When on-screen methods are in use then, just as in scrolling and bombing (which is a particular example), the scanning must be performed in the opposite direction to the movement as otherwise you will have no idea of what has and what hasn't been updated. Of course, one way round this problem is to change the character from one form into another, and then back again.

The method involving the use of a table is more fiddly but, overall, the superior method. What happens is that in the memory there exists a table of addresses of all the different characters. This table is in fact a machine code array and has the advantage that fleet movements can be easily coordinated. The characters don't have to be limited to one square each, they can be whatever you feel is right — all you need is a routine to plot a character at any address and you're away. (You may like to use my alternative sprite routine in the previous chapter, which is dedicated to manipulating large multi-square characters around the screen.) The advantage of using a system like this is that it's easy to assign each character a movement vector simply by using a two-dimensional rather than a one-dimensional array. All the update program has to do is to read through the table, pick up the vectors and add them on to the addresses. The removal of the previous image of the character may also be desirable to prevent the character leaving a trail. Using this vector method makes it a simple task to move asteroids around the screen in many different directions.

Using both of these methods, the speed of the enemy's movement can be finely controlled. You can move one or two of them at a time or move the whole lot. I personally like to move a few at a time to create the rippling effect of the *Space Invaders*. **Figure 10.9** shows the methods used for using both of the described techniques. The use of a table in machine code should already be straightforward for you, but just in case it isn't I'll describe the process now.

The first stage is to decide on the format of the table. For our purposes it might appear like that in **Figure 10.10**: three sections; first the address, then the movement vector, then the class of the alien. The last section of the table is so that you can control more than one type of alien at the same time within the same table. This uses four bytes per entry and so, to go from one entry to the next, all you must do is add four. The next stage is to decide where to put the table: find some suitable place, say 49152 onwards, and base it there. To scan the table, use a two-byte address contained in a pair of page-zero locations and use the indirect X zero-page mode to look at the table. With the zero-page address pointing to the first byte of the entry, you can obtain the address, vector and code of the character, simply by incrementing X. To move on to the next entry, zero X and add four to the memory location.

This then finishes our look at moving large numbers of objects around — it's quite simple after all. I'd like to make a few points before going on to the routines. Don't feel afraid to mix the methods by using an on-screen approach to move the bombs and a table to move the ships. This is a highly satisfactory method of producing the game, and an easy one. Don't limit your tables to one entry (ie address) only. As I have already shown, you can have much larger tables giving your characters

Figure 10.9: Moving a Fleet Sequentially, On-screen or by Table.

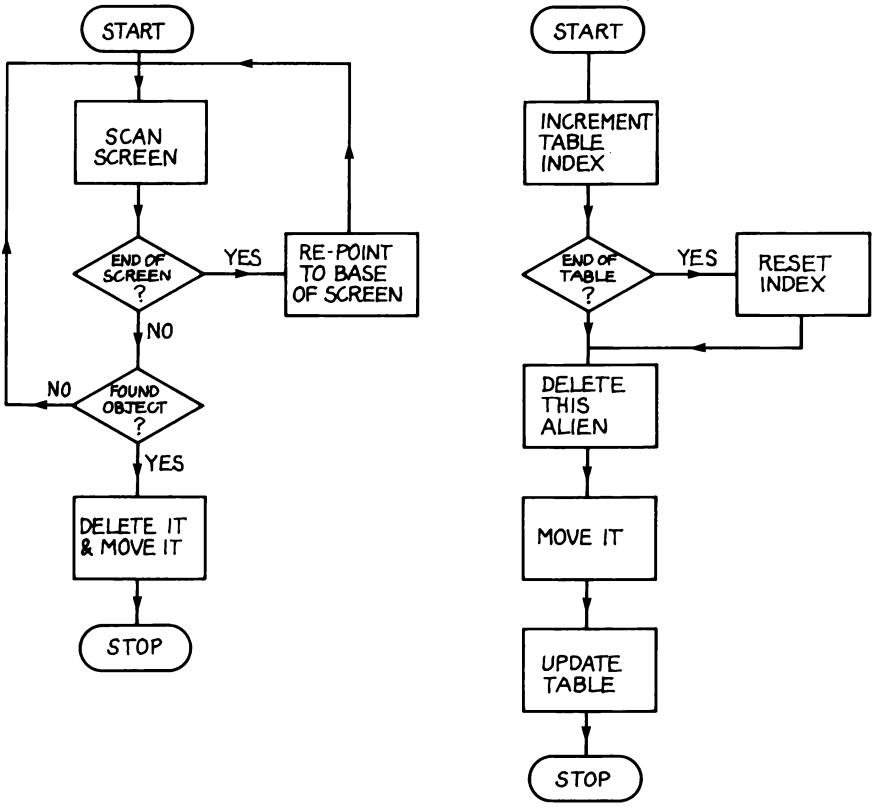
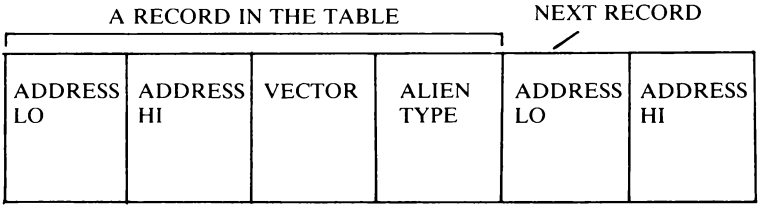


Figure 10.10: A Typical Alien Table.



more attributes and thus more realism. If you are using sound, then this is a good place to call the sound routine so you can synchronise it with (or against) the motion. If you want to link it into the interrupt, then see the section on the use of the interrupt, at the beginning of this chapter.

Array routines

The following routine reads a table of addresses and plots characters (as many bytes as you want) which start at these addresses. The base of the table is stored in locations 168–169. These are not altered by the routine. The subroutine PLOT plots the character at the address found in locations 253–254. For simple shapes, you may as well write your own routine but the alternative sprite system may be of use when manipulating larger characters. The compare-Y instruction takes as its operand twice the number of characters to be plotted (maximum of 127). This is because each character is represented by a two-byte entry.

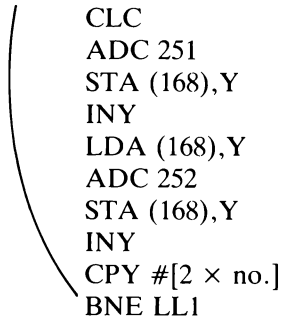
```
:LL0    LDY #0
:LL1    LDA (168),Y
        STA 253
        INY
        LDA (168),Y
        INY
        STA 254
        TYA          } Preserves the value of Y
        PHA          }
        JSR PLOT
        PLA          } Picks it up again
        TAY          }
        CPY #[2 × number characters]
        BNE LL1
        RTS
```

If you are simply plotting one character (eg a '*'), then the PLOT routine should look something like this:

```
LDA #[Character Code]
LDY #0
STA (253),Y
```

The routine is fast and effective. Deleting an entire fleet is achieved by plotting it in code 32s (the sprite routine does this for you). Moving any particular character is done by adding a vector to an address in the table. The following routine adds the same vector to every character in the table. As before, the address of the base of the table is held in 168–169 (and remains unchanged), and the compare-Y takes twice the number of characters as its operand.

```
:LL0    LDY #0
:LL1    LDA (168),Y
```



```
        CLC
        ADC 251
        STA (168),Y
        INY
        LDA (168),Y
        ADC 252
        STA (168),Y
        INY
        CPY #[2 × no.]
        BNE LL1
        RTS
```

The vector is contained in 251 and 252. To effect a move backwards, put very large numbers in the vector (eg 65535 will subtract 1 from each address). When a ship is hit and you want to destroy it, then it's not enough just to delete it from the screen — you must also delete it from the table. The following routine searches the table for a match with the address contained in 251 and 252. The value of the Y register which points to the match is returned in the Y register and the routine also stores this somewhere for future reference.

```
:LL0    LDY #0
:LL2    LDA (168),Y
        INY
        CMP 251
        BNE LL1
        LDA (168),Y
        INY
        CMP 252
        BNE LL2         Matches?
        DEY             Yes, so put Y to the start of it
        DEY
        STY 253         Store it at 253 for safety
        RTS
:LL1    INY
        CLC
        BCC LL2
```

Once we know which parts of the table to delete, it's a simple matter to delete them. Taking the X register to contain twice the number of elements in the table (what we keep on comparing Y to) and the Y-register the index to the table as produced by the search routine, then the following routine deletes that element by overwriting the entry for deletion by the final entry in the table and then removing the end entry by decrementing the end pointer.

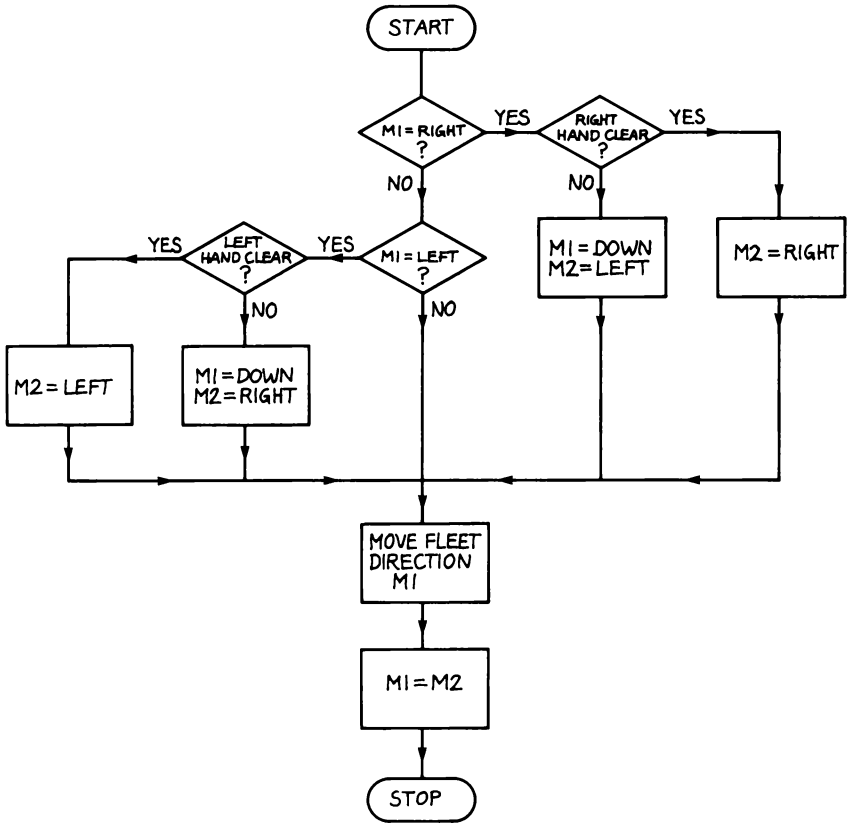
```
:LL0    DEX
        DEX
        STY 253
        TXA
        TAY
        LDA (168),Y
        LDY 253
        STA (168),Y
        INX
        INC 253
        TXA
        TAY
        LDA (168),Y
        LDY 253
        STA (168),Y
        RTS
```

A loop could have been introduced but in the interests of simplicity it's clearer to do it this way. These table manipulation routines are designed to work together: they use things such as the end pointer a lot, passing the same piece of data around among themselves. Just because I have used the immediate mode in the CPY instructions doesn't mean they will only work like that. If the size of the table is going to change from time to time, this is a very poor approach anyway. The delete routine could be accommodated inside the search routine itself, thus eliminating the need for the JSR call, but don't forget to initialise the X register first. These routines are 'modules' with which to build your movement routines: on their own they will work, but linked together they may need a little service routine to synchronise parameter passing.

The Shuffle-march

Many games have the fleets of aliens approaching by a sort of 'shuffle-march': they move left and right until the edge is reached and then jump down a line and move in the opposite direction. The logic for actually moving like this isn't so simple. **Figure 10.11** shows how to work out the movement vector for the fleet: you call it once per fleet update and not once per ship. Two variables M1 and M2 are used. M1 contains the first movement and M2 the second. If you follow the chart, you will see how moving down and reversing direction is fitted into the sequence of events. LEFT and RIGHT hand clear simply means checking down that edge of the screen to see if the invaders are on the edge. This checking should be done in machine code, although the fleets could be maintained in BASIC with string variables. Best of all, keep it all in machine code but keep it simple.

Figure 10.11: Vector Determination for 'Invader Shuffle-march'.



RANDOM NUMBERS

The problem of getting random numbers in machine code has caused many a would-be programmer to give up. The key problem is how to make a machine like a computer, which follows a series of distinct instructions, produce an unpredictable sequence of numbers? In BASIC, of course, there is the RND(X) function. In normal machine code there is nothing, but on the 64 there is a hardware random number generator! Think for a moment about 'white noise'. This is simply a series of rapidly changing frequencies. It's more than rapidly changing — it's random. The makers of the SID set one of the registers to the output of voice 3. Thus if it's in white noise mode then, by PEEKing the location, we can get our random numbers. The frequency at which the voice is operating determines the rate at which the numbers are being generated. The output frequency is given by:

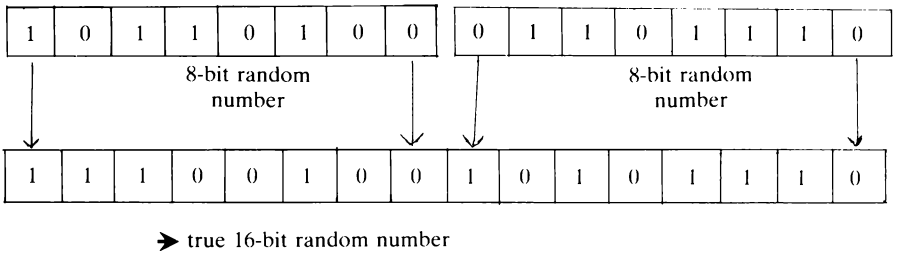
f = (16-bit frequency setting) × 0.0596 Hz

This means a maximum of 3906 updates a second. To utilise the generator then, the following steps are taken:

1) Switch on voice 3 for noise.
2) Disable output of voice 3 (set bit 3 of 54290).
3) Set frequency at 54286 and 54287.
4) Use PEEK(54299) for random numbers.

This gives us random numbers in the range 0–255. How do we construct a random address? Simply put two random numbers back to back. By taking one bit at a time and shifting in, we can construct numbers as large as we like (**Figure 10.12**). Don't, however, try adding two random bytes together to obtain a random number in the range 0–510: you can stick them together, but not add them.

Figure 10.12: The Correct Way to Add.

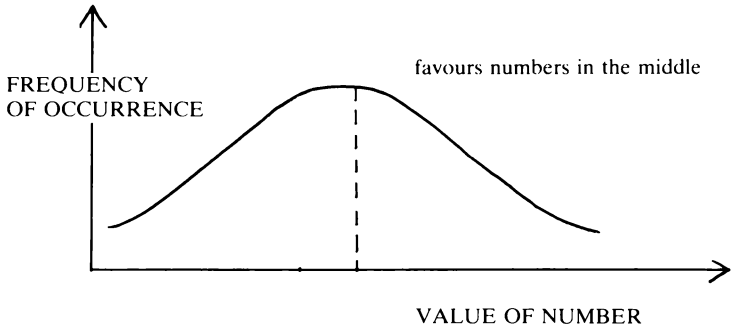


The problem with adding is that it biasses the number towards the centre of the range. Try tossing two dice and record how many times each total comes up. You'll find that there are many more sevens than both twelves and twos put together. The result is a bell-shape curve like that in **Figure 10.13**. Of course, this biassed distribution might be what you want. The more components you add together the heavier the bias. The way you make random numbers is largely up to you. Used constructively they can help to spice up a game with an element of the unknown. See Fleet Movements and Non-linear Movements, both in this chapter, for details on how to use them.

NON-LINEAR MOTION

Some games call for the aliens to move with more intelligence than just in the standard single direction. This may entail adding a random component to the motion, forcing the alien to follow a predefined path or even to 'home in' on a particular object. Undoubtedly this makes games much more fun but, as always, to achieve this you must be

Figure 10.13: The Result of Conventional Adding: Bias.



prepared to work a little harder. To help illustrate this technique, some of the examples will be written in BASIC so that you can follow them and type them in more efficiently.

The first case to consider is the one where an extra component is added to the object's vector. Suppose that the object is a bomb falling down the screen. Its vector is therefore +40. To liven things up a bit, we might consider adding one of the numbers 1, 0 or −1 to the address as well as the vector. The 0 obviously leaves the bomb unmoved while the 1 and −1 swing it to the right and left. There is nothing to stop us from adding any number from the full vector range, but this would tend to delay the bomb's progress towards the bottom of the screen. The following two programs will help explain this process.

Random Bomb

```
10  REM RANDOM BOMB
15  POKE 53281,1 : PRINT "[CLS]" : POKE 53281,0
20  S = 1044
30  DEF FNR(X) = INT(X*RND(1)+1)
40  POKE S,42
50  S = S+42-FNR(3)
60  IF S>2023 THEN RUN
70  GOTO 40
```

Random Motion

```
10  REM RANDOM MOTION
20  POKE 53281,1 : PRINT "[CLS]" : POKE 53281,0
30  S = 1368
```

```
40  DEF FNR(X) = INT(X*RND(1)+1)
50  POKE S,46
60  S = S+2-FNR(3)+(2-FNR(3))*40
70  IF S<1024 OR S>2023 THEN RUN
80  POKE S,160
90  GOTO 50
```

Of course, true random motion is achieved by adding only a number from the full vector range to the address. The process of randomly extracting a number from a table in machine code is described below.

The first stage is to obtain a random number. This is done by PEEKing 54299 (see the Random Number section earlier in this chapter). Each entry in the table consists of two bytes (so we can represent negatives as large numbers) and, as there are eight directions there, we want the random number to take on one of the following values: 0, 2, 4, 6, 8, 10, 12, 14. To get this range we simply AND our random number with 14. A quick TAX moves the value into the X-register, and we can look up the entry using the absolute X addressing mode.

```
LDA 54299
AND #14
TAX
```

LDA TABLE,X gives the first entry and INX, LDA TABLE,X supplies the second. This vector is simply added into the old address.

Non-random motion

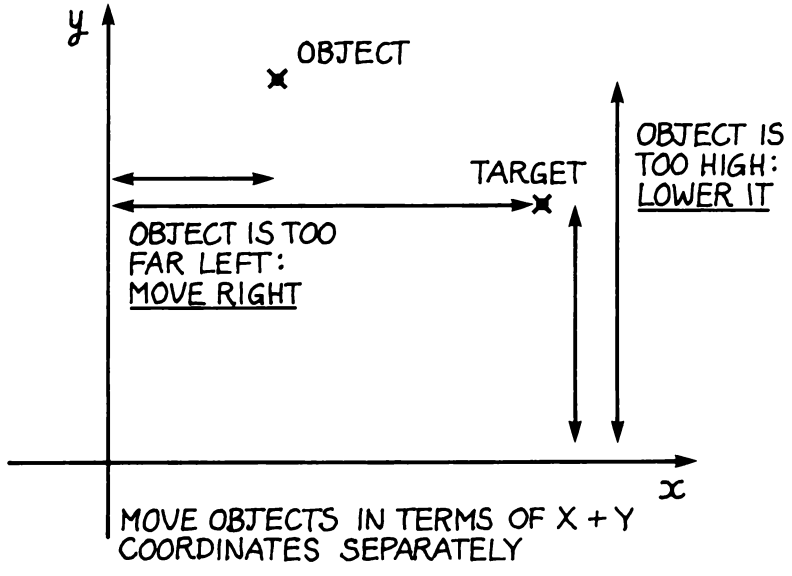
The natural progression, from a random motion, is to one that has been laid out. What I mean is that you may want an alien ship to take a spiral path down to the bottom of the screen and then return after the foray. Instead of trying to establish an algorithm that will compute the next position, the easiest way is to add in successive numbers from a table. The structure is easy: you simply have a pointer keeping your position in the table and to move the object on you simply increment the pointer and look up the corresponding vector in the table. A null vector denotes the end of the table and might cause the pointer to be reset to another position in the table. This technique is illustrated in the next program, where the vector table is contained in DATA statements.

Dance

```
10  REM **DANCE
20  DATA 41,41,41,41,41,41,41,41,41,41,41,41,41,41,1,1,1,1,1
30  DATA -39,-39,-39,-39,-41,-41,-41,-41,-41
40  DATA -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,41,39,39,39,39
50  DATA 1,1,1,1,1,1,1,1,1,0
60  DIM P(60)
70  L=1
80  READ P(L) : IF P(L)<>0 THEN L=L+1: GOTO 60
90  POKE 53281,1 : PRINT "[CLS]" : POKE 53281,0
100 S=1024 : L=1
110 POKE S,42 : L=L+1 : POKE S+P(L),42 : POKE S,46
120 IF P(L)=0 THEN L=10
130 IF S>2000 THEN END
140 S=S+P(L) : GOTO 110
```

The first stage is to read this table into an array and initialise the objects position. The table is then worked through until the null entry is reached, at which stage the pointer is reset to a new position inside the table. The program continues like this for ever, with the result that the object flutters down from the top left corner and proceeds to perform a figure-of-eight dance. This method is easily adapted to machine code use.

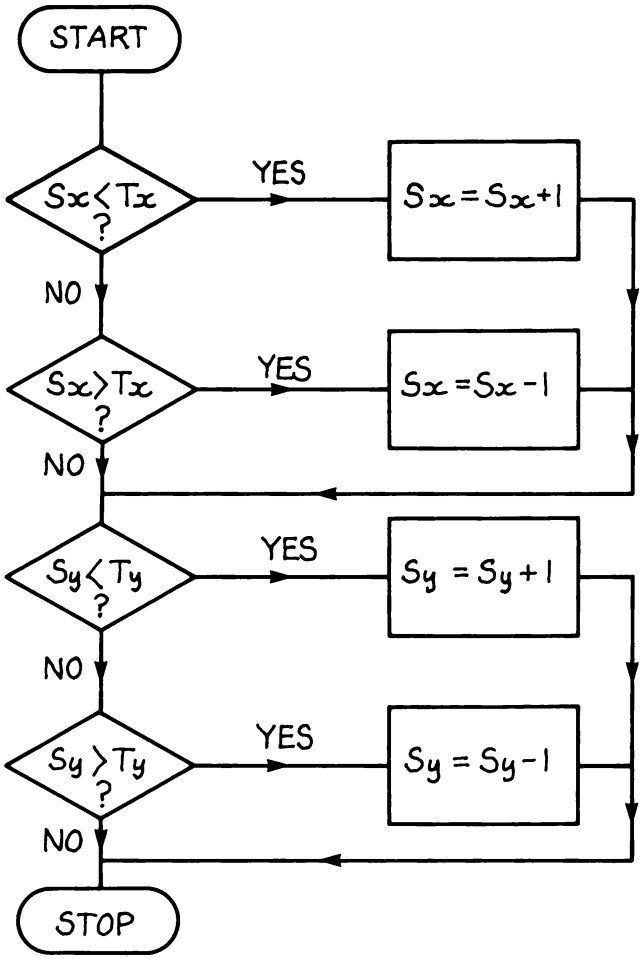
Figure 10.4: Homing Theory.



Homing Motion

The natural progression from pre-defined motion is homing motion. This is where an object homes in on another object. The classic *Zombie Island* uses the principle to good effect. The mechanics of the situation are also remarkably simple. The first stage is to break up the addresses into a pair of X,Y coordinates like those used on a graph. For this reason, sprites are the easiest to use as they are already expressed in this fashion. Now look at **Figure 10.14**, which shows two points A and B. B is the point towards which A is moving. To update A's position apply the following rules:

Figure 10.15: Home (Sx, Sy) on to (Tx, Ty).



If $X_b > X_a$ then increment X_a
If $X_b < X_a$ then decrement X_a
If $Y_b > Y_a$ then increment Y_a
If $Y_b < Y_a$ then decrement Y_a

If you study **Figure 10.15**, the reasons for these rules should become apparent. This BASIC program updates sprites 1 to 7 towards the location of sprite 0.

```
10    REM **SPRITE HOMING ROUTINE
20    HX=PEEK(53248) : HY=PEEK(53249)
30    FOR A=53250 TO 53262 STEP 2
40    X=PEEK(A)+SGN(HX-PEEK(A))
50    Y=PEEK(A+1)+SGN(HY-PEEK(A+1))
60    POKEA,X : POKEA+1,Y
70    NEXT : GOTO 30

1000  REM **SET UP SPRITES FOR DEMONSTRATION
1010  FOR A=53248 TO A+15 : POKE A,FNR(255) : NEXT
1020  POKE 53269,255
1030  FOR A=2040 TO 2047 : POKE A,0 : NEXT
      ----FNR(X)=INT(X*RND(1)+1)----
```

This program drives sprites 1–7 towards sprite 0. It's very slow, but the program clearly shows how the algorithm works. The logic is in lines 40 and 50 which use the available SGN(X) function. This returns either a −1, 0 or 1 depending on whether the number is less than 0, 0 itself or greater than 1.

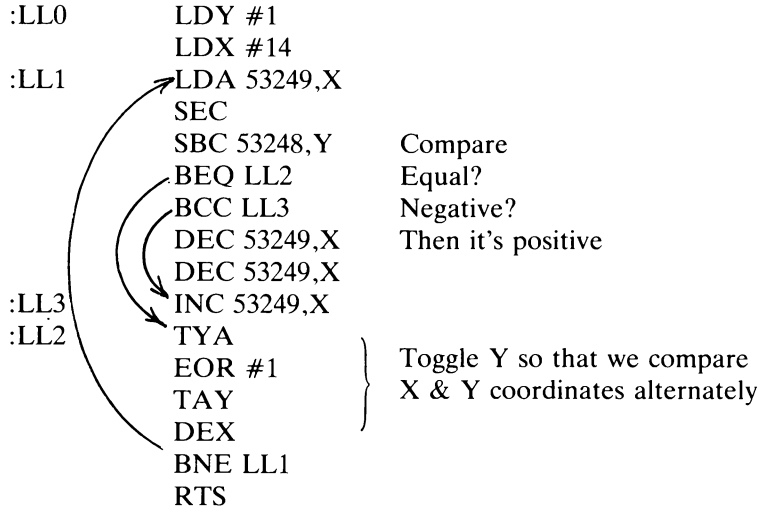
Run the 1000 subroutine first, to set up some sprites for a chase.

The machine code program which follows does the same thing, but much faster. A clever way of using this routine would be to attach it to the interrupt – see the section on the interrupt earlier in this chapter. You can achieve the same effect using normal characters, but it's far more tedious. Eight sprites should be enough; try to avoid using up all your sprites with other things.

Using non-linear methods will liven up your game, but don't overdo it. The methods outlined here are not the only ones, they are just the simplest, so don't hesitate to write your own custom-designed routines. For example, to add some detail to the sprite routine you may consider altering it, so that when a sprite arrives on the 'target' it is automatically disabled. This sprite routine almost constitutes a game in itself. The player controls sprite 0 with a joystick and has to avoid being struck by any of the other sprites.

The logic is a little complicated but if you run it through by hand, you should see what's happening. The Y register alternates from a 1 to a 0,

so that the SBC 53248,Y instruction alternates between comparing X and Y coordinates. The effect of two DEC instructions followed by an INC is simply to produce one DEC instruction. If you follow the arrows then you should see what's happening.



This runs in under 5/1000 second and so is capable of producing a very fast display indeed. Hooking it on to the interrupt can be quite fun, as well allowing you to write an 'Escape and Evade' game in BASIC. For details of how to remove colliding sprites see the section on sprites earlier in this chapter.

THE COLOR MEMORY

The color memory has a lot to say for itself. For each character on the screen, there is a corresponding color byte which dictates the colour of that character. The paper colour set at 53281 controls the colour of the background. This means that clever things can be done. One use is the 'sudden appearance' idea. This entails setting the paper colour to the colour in color memory. If this is done, then something can be printed on the screen and not be visible — yet it's there. By simply changing the paper colour the 'hidden' screen jumps out all at once.

By setting different sections of the color memory to different values, we can obtain a colourful screen. The old *Space Invader* machines have horizontal bands of colour created through strips of green and red plastic taped over the screens. This effect can be simulated very easily and may actually serve some purpose in a game (eg telling a player he is in a dangerous zone — RED). If the color memory is set up at the start

of each game, then you can forget all about it. On the other hand if you are POKEing in the colour as you move the aliens about, the program becomes complicated where it could be simple.

If some areas only of the screen are the same colour as the paper, then they are in effect 'invisible' patches. What's inside is unknown territory — perhaps as the player explores, the contents are revealed. Doing this saves having another section of memory which contains what should be on the screen. Once again, doing something this way makes certain tasks a lot easier.

The border colour pointer (53280) is another valuable location. Strangely enough it actually has an effect on the paper colour. As far as I can tell, the brighter the border the duller the paper. This gives us 16 shades for each paper colour! Personally I use the border to act as an information source to the player or merely a guide to what level the player is playing at. If you change the paper and border colours for each attack wave, then this helps to keep the game fresh and exciting.

APPENDIX A
Utility Programs

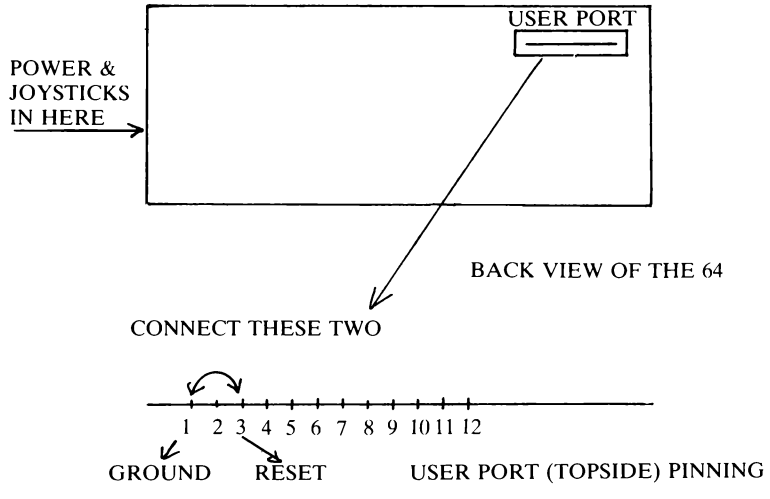
The appendices contain a large amount of information that didn't seem to fit in elsewhere in the book. There are utility programs, routines and details of the machine, all of which should be of help to you.

CRASH RECOVERY

It will happen sooner or later — your program will just freeze and the keyboard will go dead. (It's a wonderful feeling when five hours work is lost . . .) To start with, you should make a point of SAVEing every half-hour at least.

But even when you do have a crash and the RUN/STOP + RESTORE combination fails to respond, all is not lost (if you want to disable the RUN/STOP + RESTORE then just change the vector 792–793). On the back of the machine you will find the ports as shown in **Figure A.1**. Take a pair of conducting tweezers and short the reset pin, as shown, for half a second. Let go. The screen will clear and the Commodore title will flash up. Now type the following sequence of numbers into, say, the tape buffer (anywhere will do) and call them with the appropriate SYS.

Figure A.1: Grounding the Reset Pin.



169,1,141,2,8,32,51,165,24,165,34,105,2,133,45,165,35,105,0,
133,46,76,94,166

Stand back and pray. Your program is now reincarnated.

Of course the clever ones amongst us will load in this routine somewhere safe at the start of every day. Then on crashing, all that's needed is a quick SYS call. The code disassembles:

```
LDA #1
STA 2050    Reincarnate program
JSR 42291
LDA 34      Utility pointer
CLC
ADC #2
STA 45
LDA 35
ADC #0
STA 46      Pick up variables
JMP 42590   Finish
```

If you do keep the recovery routine resident all the time, then find somewhere other than the tape buffer as this is cleared on RESET. Try in the spare 4K RAM zone.

SITING THE PROGRAM AND MEMORY MANAGEMENT

If you're running a BASIC program along with machine code support, UDG and sprite data and large screen layouts, then the problem arises of where to site the routines and, of course, where you can store your machine code variables, parameters and tables. Don't worry, this is a 64K machine — if you can fill up the memory with sensible code and still be looking for more space then you are either crazy or you need a larger machine.

Traditionally the abode of machine code routines on Commodore machines has been the tape buffer. We still have this and it runs from 828 to 1019, but there is unused memory on either side of it extending the range to 820 to 1023 — 204 bytes. This will house most collections of 'assisting' routines, but if you are really pushed then there is 4K available from 49152 to 53247. Sprites site very well in the tape buffer (they won't go in many other places) so you may be forced to move up to the 4K zone. This won't affect the speed of execution at all.

The *Programmer's Reference Guide* has a habit of using the middle of

the BASIC workspace for such things as UDG tables. This seems crazy, and it is, but you can get round the problem by moving the workspace itself. This is, of course, the key to housing the really big programs. Locations 43 and 44 contain the start of the workspace. PEEK them and you'll get the address 2049. 55 and 56 contain the highest address used by BASIC — a value of 40960. Try pulling the top down to within 8K of the base. You now have ample space for a BASIC program and over 30,000 bytes to play with for machine code purposes! Two important points: first, POKE the top down before you create any variables and then CLR to inform the 64; second, only POKE the base up *outside* a program and then call a CLR (if you do this inside, then you're going to lose the bottom of your program).

This should have solved all routine-siting problems. The availability of page-zero memory, however, is another matter. Page-zero is very special and on the 64 there is very little spare. Of course, if you aren't using BASIC then it's all yours (except 0 and 1). Locations 2, 251, 252, 253 and 254 are all free for the user, whatever. This is sufficient, but if it's really necessary some of the other locations are so rarely used (especially tape registers) that you should get away with using them. Another useful little hole is 679–767. But be very careful with page-zero operations — they can cost you your program.

ENTERING MACHINE CODE FROM BASIC

This may seem trivial but the chances are that you know perhaps one or two of the possible ways to enter machine code. There are five main ways to do this. Each has its advantages and disadvantages, and so it's worth considering them instead of blindly using an SYS.

The SYS call is the standard entry. You simply use SYS (X) and call the machine code subroutine at X as if it were a GOSUB command. This method is quick and easy, but passing data to and from the routine can get messy on the BASIC side.

This problem is overcome by use of the little-used USR(X) command. This is treated just like a function such as SIN(X) or COS(X). The value inside the brackets is evaluated and placed in the 'floating-point accumulator #1'. The associated machine code routine is then called. The address is pre-arranged by placing it in the USR vector at 785 and 786. On completion of the routine the floating-point accumulator has its value returned as the value of USR(X). Thus the routine is called like this:

```
A=USR(1000)
```

For details of the floating-point accumulator see Appendix C (BASIC Text and Variable Storage). It consists of five bytes and can thus be used to send quite a lot of data to and from machine code routines. By altering the USR vector, a number of different routines may be called throughout the program. If you are converting a program from an earlier Commodore machine, take great care to change the vector pointers from 1 and 2 to 785 and 786.

The interrupt provides another link into machine code. You simply pick up the vector from 788 and 789 and re-point to your routine, taking great care to chain the end of your routine on to the start of the interrupt service routine at 59953. If you don't do this and just stop with an RTI, then you're going to lose out on all the housekeeping routines. Entry by the IRQ interrupt needs a little more foresight in programming but is definitely worth the effort. Don't forget that, if your routine takes any longer than a sixtieth of a second to execute, then the clock (TI$) will go all cock-eyed (slow). By pressing one of the CIAs into use, you can even dictate your own interrupt signal schedule. There's no doubt about it — the interrupt is the chief way to harness the 64.

An underhand way of sneaking into machine code is to intercept the BASIC interpreter while it's happily scanning a line to see what it says. The routine known as CHRGET supplies successive characters from a line with each call. The routine is thoughtfully present in zero-page RAM. This means that we can drive a 'wedge' in and divert the latter half to a routine of our choice. By watching for a control character, the possibility of adding a new command to BASIC is opened up. Have a look at the CHRGET routine:

```
INCZ122
BNE ↑ 2
INCZ123
LDA£------
CMP #58
BCS ↑ 10
CMP#32
BEQ ↑ 239
SEC
SBC#48
SEC
SBC#208
RTS
```

The trick is to replace the first SEC instruction with a JMP£820 (or

wherever). Then at 820 we can do the following:

```
CMP #0
BEQ ↑ [User routine]
SEC
SBC#48
etc etc
```

The only real problem is how to insert the JMP instruction without causing an error (since it can't all be done at once). The answer is to use a separate piece of code to drive the wedge in. Like the interrupt, this method is handy but fiddly.

By altering the vector to, say a LIST or SAVE routine, using that command will cause a jump to the new vector. You may wish to modify the effect of the old command or totally change its sense (fun). See the table for some vector addresses you might like to play with. This is the last of the five chief ways of entering machine code. There are more but it's really not worthwhile listing them here.

USER-DEFINABLE GRAPHICS

This section is devoted to the subject of UDGs and contains a little editor. UDGs are simply graphics characters that the user has designed. When you switch on, the 64 has a set of 256 different preset characters. While these are remarkably versatile, it's often better to define your own. All that must be done is to draw out the shape of your new character, lay a square grid over it (8 × 8) and decide which squares should be filled. Then you simply add across each row considering it as an 8-bit binary number to give eight numbers, and the definition is complete.

The problem is that the standard character set is stored in ROM so we can't directly alter it. Instead we must copy it elsewhere, change the appropriate matrices and tell the machine to look at the new set instead of the old. Further problems arise as the ROM is not normally visible to BASIC, so the ROM must be banked in: to do this safely, we must POKE out the interrupt so that it doesn't get upset when it finds that someone is looking in the memory. The procedure is as follows:

1) Stop interrupt.
2) Bank in character ROM.
3) Copy the character definitions.
4) Bank the ROM out.
5) Restart interrupt.

To stop the interrupt use POKE 56334,PEEK(56334)AND 251; to restart use POKE 56334,PEEK(56334)OR 4.

To bank the ROM in and out use POKE 1,PEEK(1)AND 254, POKE 1,PEEK(1)OR 254.

The character ROM starts at 53248 (it resides *under* the VIC-II chip). If you copy the first 512 bytes then all the alphanumeric and special symbols will be copied without copying the entire character set. In BASIC this can take time and so a COPY routine is quite handy in this sort of situation.

To re-point the address of the character definitions use:

```
POKE 12288,PEEK(12288)AND 240+X
```

Where X is an integer from 0 to 15 pointing to the start of the character matrix — X × 1024. This may mean you have to move BASIC around a bit to make space. For further details of UDGs consult the *Programmer's Reference Guide.*

A UDG Editor

The program listed here allows you to design your own UDGs with great ease. When RUN, there is a short delay as the alphanumerics are copied and the character matrix moved. Then the screen clears to reveal an 8 ×8 square on which to create your own graphics and a mini-menu outlining the purpose of the function keys. On the square is a cursor which may be moved around using the cursor keys. To fill a square, simply press the space-bar; to clear a square use the left-arrow sign. The function keys aid your design in the following ways:

F1: This works out the matrix for the figure on the design square and prints the number opposite the corresponding row.

F2: This option requests a character for input. Simply type a character (eg 'A') and press RETURN. The character matrix is then interrogated and the matrix copied out on to the design square.

F5: This is the same as F2 except that it overlays the requested symbol.

F7: This evaluates the symbol in the design square and assigns it to the POKE code that is requested.

Using these functions together you can look at the character for an 'A' or 'Q' and see how Commodore designed them, you can mix two graphics characters, alter them yourself using the cursor, and then create your own personalised character set. Using UDGs in your games will make a lot of difference to the final result. It's even possible to append a UDG section to the end of any BASIC program to spice it up a bit without the program noticing. There's no excuse for copying

somebody else's — all told there are 2 × 2 × 2 × 2 (etc 64 times) different symbols possible on an 8 × 8 square. That's about 1.8×10^{19}!

Editor

```
10 REM UDG EDITOR
20 POKE56334,PEEK(56334)AND254:POKE1,PEE
   K(1)AND251
30 FORA=0TO511:POKE12288+A,PEEK(53248+A)
   :NEXT
40 POKE1,PEEK(1)OR4:POKE56334,PEEK(56334
   )OR1
50 POKE53272,(PEEK(53272)AND240)+12
60 POKE53281,0:PRINT"[CLS]":POKE53281,24
   6
70 FORA=1TO8:PRINT" ........":NEXT
80 PRINT"[CD][CD]F1 - BREAK DOWN CHARACT
   ER"
90 PRINT"F3 - DRAW CHARACTER"
100 PRINT"F5 - OVERLAY CHARACTER"
105 PRINT"F7 - ASSIGN CHARACTER"
110 P=1066:U=46
120 GETA$:IFA$=""THEN120
130 V=0:IFA$="[CR]"THENV=1
140 IFA$="[CL]"THENV=-1
150 IFA$="[CU]"THENV=-40
160 IFA$="[CD]"THENV=40
170 IFPEEK(P+V)=46 OR PEEK(P+V)=160 OR V
   =0THEN190
180 V=0
190 IF V<>0 THEN POKEP,U:P=P+V:U=PEEK(P)
   :POKEP,81
200 IF A$=" "THENPOKEP,160:U=160
205 IFA$="_"THENPOKEP,46:U=46
210 IFA$="[F3]"THENGOSUB2000
220 IFA$="[F5]"THENGOSUB2000
230 IFA$="[F7]"THENGOSUB3000
240 IFA$="[F1]"THENGOSUB4000
999 GOTO120
1000 PRINT"[HOM][CD][CD][CD][CD][CD][CD]
   [CD][CD][CD][CD][CD][CD][CD][CD][CD][
   CD][CD][CD][CD][CD][CD][CD][CD]
                                    [CU]"
1010 RETURN
```

```
2000 REM OVERLAY
2010 GOSUB1000
2020 INPUT"CHARACTER";C$
2030 N=PEEK(1955)
2040 FORA=0TO7:D=PEEK(12288+N*8+A)
2050 FORJ=0TO7:S=1065+A*40+J:IFA$="[F3]"
   THENPOKES,46
2060 IF 2^(7-J)=<D THEN D=D-2^(7-J):POKE
   S,160
2070 NEXTJ,A:U=PEEK(P):RETURN
3000 REM ASSIGN
3010 GOSUB1000:INPUT"CHARACTER CODE";C:I
   FC<0ORC>255THEN3010
3020 FORA=0TO7:T=0:FORJ=0TO7
3025 IFPEEK(1065+A*40+J)=160THENT=T+2^(7
   -J)
3030 NEXT:POKE12288+C*8+A,T:NEXT
3040 POKE1063,C:RETURN
4000 REM VALUES
4010 PRINT"[HOM]"
4020 FORA=0TO7:T=0
4030 FORJ=0TO7:IFPEEK(1065+A*40+J)=160TH
   ENT=T+2^(7-J)
4040 NEXT:PRINT"[CR][CR][CR][CR][CR][CR]
   [CR][CR][CR][CR][CR][CR][CR]        [C
   L][CL][CL][CL][CL][CL][CL]"T
4050 NEXT:RETURN
```

SPRITES

This short section is aimed at the superb sprite graphics facility available on the 64. However, to use it well a lot of work must go into designing the characters. Sprites are very similar to UDGs in that each dot is controllable. They are much larger, being 24 dots across (three bytes) and 21 dots down. Their definitions are stored as the chain of numbers they represent if each set of eight bits is read (left to right) as a binary value. There are eight sprites and the manual describes their use and abuse in fuller detail, but I present here a flexible Sprite Editor.

The program produces a large square and cursor arrangement on which a sprite can be created. The cursor is moved by the cursor keys and the repeat facility can be used to good effect on the large square. Each of the eight function keys serves a purpose as well as the space-bar and left-arrow keys, which respectively plot and erase squares.

F1: Calculates the corresponding matrix numbers and prints them in the appropriate position on the screen.
F2: Re-RUNs the program — effectively clearing the square for another design.
F3: This reflects the left half on to the right half — very handy for producing symmetrical sprites.
F4: Exits the program.
F5: 'PEN DOWN' — This causes a permanent trail to be left after the cursor.
F6: DUMP? YES — Sprite matrix calculated and entered into DATA statements — just press RETURN over them.
NO — Stores sprite symbol on to tape.
F7: 'PEN UP' — This stops the permanent trail effect of F5.
F8: This loads DATA up from tape.

Sprite Editor

```
10 REM SPRITE EDITOR
11 REM
12 REM **PAUL ROPER
15 DIM S(64)
20 PRINT"[CLS][CR][CR][CR][CR][CR][CR][C
   R][CR][CR][CR][CR][CR][CR][CR][CR][CR
   ][CR][CR][CR][CR][CR][CR][CR][CR][CR]
    SPRITE-EDITOR"
30 FORA=1TO21:PRINT".....................
   ....":NEXT
40 P=1024+160+10
50 UC=46
60 GETK$
61 IFK$="[F2]" THEN RUN
62 IFK$="[F1]" THEN 1000
63 IFK$="[F6]" THEN 3000
64 IFK$="[F8]" THEN 4000
65 IFK$="[F4]" THEN 5000
70 IFK$="[CR]"THENIFPEEK(P+1)<>32THENPOK
   EP,UC:P=P+1:UC=PEEK(P)
72 IFK$="[CL]"THENIFPEEK(P-1)<>32THENPOK
   EP,UC:P=P-1:UC=PEEK(P)
73 IFK$="[F3]"THEN2000
74 IFK$="[CU]"THENIFPEEK(P-40)<>32THENPO
   KEP,UC:P=P-40:UC=PEEK(P)
75 IFK$="[F7]"THENPD=0
```

```
76 IFK$="[CD]"THENIFPEEK(P+40)<>32THENPO
   KEP,UC:P=P+40:UC=PEEK(P)
77 IFK$="[F5]"THENPD=1
78 IFK$=" "THENUC=160
79 IFK$="_"THENUC=46
81 IFPDTHENUC=160
90 IFUC=46THENPOKEP,81
100 IFUC=160THENPOKEP,102
110 GOTO60
1000 REM ADDUP SPRITE
1005 PRINT"[HOM]"
1010 FORQ=1064 TO 1864 STEP40
1015 PRINT"[CR][CR][CR][CR][CR][CR][CR][
   CR][CR][CR][CR][CR][CR][CR][CR][CR][C
   R][CR][CR][CR][CR][CR][CR][CR]";
1020 FORA=Q TO Q+16 STEP8
1030 V=128:T=0
1040 FORB=A TO A+7:IFPEEK(B)=160 OR PEEK
   (B)=102THENT=T+V
1050 V=V/2:NEXT
1060 PRINTT;:NEXT:PRINT
1070 NEXT
1080 GOTO60
2000 REM REFLEFT LEFT TO RIGHT
2010 FORQ=1064TO1864 STEP40
2020 FORA=0TO11
2030 K=Q+A:IFPEEK(K)=102 OR PEEK(K)=160
   THENPOKEQ+23-A,160
2040 NEXTA,Q
2050 GOTO60
3000 REM SAVE SPRITE
3005 N=1
3006 POKEP,UC
3008 X=1
3010 FORQ=1064TO1864STEP40:FORA=0TO16STE
   P8
3020 V=128:T=0:FORB=0TO7
3030 IFPEEK(Q+A+B)=160THENT=T+V
3040 V=V/2:NEXT:S(X)=T:X=X+1:NEXTA,Q
3045 INPUT"[CLS]DUMP VALUES";D$:IFLEFT$(
   D$,1)="Y"THEN9000
3050 INPUT"[CLS]NAME OF SPRITE SYMBOL";N
   $
3060 OPEN1,1,1,N$
```

```
3070 FORA=1TO63:PRINT#1,S(A):NEXT:CLOSE1
   :RUN
4000 REM LOAD SPRITE
4005 N=1
4010 INPUT"[CLS]NAME OF SPRITE SYMBOL";N
   $
4020 OPEN1,1,0,N$
4030 PRINT"[CLS][CR][CR][CR][CR][CR][CR]
   [CR][CR][CR][CR][CR][CR][CR][CR][CR][
   CR][CR][CR][CR][CR][CR][CR][CR][CR][C
   R]SPRITE EDITOR"
4040 FORA=1TO21:PRINT".................
   ......":NEXT
4050 FORQ=1064TO1864STEP40:FORA=0TO16STE
   P8
4060 INPUT#1,D:V=128
4070 FORW=0TO7:IFD>=VTHEND=D-V:POKEQ+A+W
   ,160
4080 V=V/2:NEXT:NEXT:NEXT:CLOSE1:GOTO60
5000 PRINT"[CLS]PROGRAM TERMINATED.":END
9000 INPUT"LINE START";LS:PRINT"[CLS]";
9010 FORA=1TO63STEP9
9020 L$="":L$=STR$(LS)+"DATA":LS=LS+10
9030 FORB=ATOA+8:L$=L$+STR$(S(B)):IFB<A+
   8THENL$=L$+","
9040 NEXT:PRINTL$:NEXT
9050 END

READY.
```

APPENDIX B
Mini-assembler

Whether or not you already own an assembler, you should find 'Mini-assembler' a useful addition to your utility set. It is written in BASIC but is very compact and has some handy features, including a fast disassembler and an ingenious method of editing. The first part of this appendix describes the program and its use. Then there are a number of 'tool' programs, which run on the interrupt and are called by the function keys. The appendix is concluded with a machine code loader which loads a mini-toolkit, carrying all of the previous features, on to the interrupt.

Mini-assembler makes great use of DATA statements: the list of mnemonics, their opcodes, and the program for assembly are all stored using them. This means that editing is simple — you use the flexible screen editor as normal. You can have one instruction per line or separate instructions via commas. Your program is written in the area of lines 20–999. This will give you plenty of space. If you wish to POKE down the top of BASIC then this should be done before RUNning. Normally BASIC assemblers are slow and laborious but, by using the editor in this fashion, all the assembling is done at once and the program takes short-cuts through the mnemonic table from time to time with the more common commands.

Program description

When RUN, the title routine at line 3000 is executed. This moves straight on to the mnemonic loader at line 9000. With the arrays dimensioned and data read in, control returns to line 3000 where the user is asked for ASSEMBLY or DISASSEMBLY. Disassembly takes place in line 2000–2999: assembly in lines 1000–1999. The data pointer is first RESTOREd, and then each mnemonic is read through in turn. String-slicing routines extract details such as mode, three-letter mnemonic and operand. If the mnemonic is commonly used then short-cuts are initiated. Any problems are dumped to the screen beside the offending code and, once the end-of-program flag comes up, assembly ends. To economise on program efficiency, the addressing modes are represented by a single-character suffix straight after the three-letter mnemonic. Consult **Table B.1** for details of this. There is no

provision for the use of labels, but relative branches are still calculated for you. Be careful to type in the DATA statements exactly — it will take some time.

Table B.1: Prefix Table for Mini-assembler.

ADDRESSING MODE	PREFIX
Immediate	'#'
Absolute	'£' — Listed as ' \ ' in listing
Absolute X	'X'
Absolute Y	'Y'
Relative	' ↑ '
Accumulator	'A'
Implied	3-letter mnemonic only
Zero X	'1'
Zero Y	'2'
Zero-page	'Z'
Indirect	'$'
Indexed indirect	'!'
Indirect indexed	'?'

The Mini-assembler command is structured as follows:

[3-letter mnemonic] + [prefix] + [operand]

For implied instructions, only the 3-letter mnemonic is required.

In use

To make use of the disassembler, run the program and answer 'yes'. The prompt now requires a start address to commence disassembly. Once this is done, the space-bar will produce one instruction per depression, or you can hold the space-bar down. To leave the program, use the RUN/STOP key. GOTO 2000 will restore you to the disassembler.

The assembler is used by first typing in the code into DATA statements with line numbers <1000 and then running the program and requesting the assembler. LOADing and SAVEing is easily done: just save the whole thing along with your code. It might be a good idea to keep a permanent copy full of blank DATA statements. Each instruction is represented by the standard three-letter mnemonic, a mode suffix and then an operand. Look at the following commands:

LDA#100	Load the accumulator immediately with 100
STAX1024	Store the accumulator at 1024+X

BCC ↑ 851	Branch on carry clear to 851
LDA!251	Load the accumulator indexed indirect at 251
RTS	Return from subroutine.

When RUN, the code is assembled by default to location 49152 onwards. This can be changed by use of the directive which simply re-points the address of the next instruction to be assembled. The directive is used like this:

*820

This starts assembly at location 820. The * command may be used several times so that you can assemble more than one routine at a time. The assembly listing shows the action of these commands by a reverse field message.

Mini-assembler is a little strange, but after a short while you will find it easy enough. The following routines are all printed in the 'Mini' format. Each routine performs some useful task based around the function keys. These are widely-publicised attributes of the 64, but nobody ever seems to use them for their intended purpose. I hope these routines show you how to utilise them for yourself.

Mini-assembler

```
10 REM ASSEMBLY CODE FOLLOWS
15 DATA*820
20 DATA LDX#0,STAX1024,INX,BNE^822,RTS
30 DATA ASLA
1000 REM ASSEMBLER
1010 GOTO3000
1020 RESTORE:P=49152
1025 PRINT"ASSEMBLY COMMENCES."
1030 READ C$:IF C$="*^*"THENPRINT"ASSEMB
   LY COMPLETE.":END
1035 PRINT:PRINTP,C$"    ";:S=0
1036 IFLEFT$(C$,1)="*"THENP=VAL(RIGHT$(C
   $,LEN(C$)-1)):PRINT"[RVS]P=";P;:GOTO1
   030
1040 M$=MID$(C$+CHR$(0),4,1)
1045 I$=LEFT$(C$,4):B$=LEFT$(C$,3)
1065 IFM$=CHR$(0)THEN1081
1070 A$=RIGHT$(C$,LEN(C$)-4)
1071 IFM$<>"^"THEN1080
1072 V=VAL(A$)-P-2:IFV<0THENV=256+V
```

```
1073 IFV<00RV>255THENPRINT"ERROR:":PRINT
   "[RVS]"C$" OUT OF RANGE":V=0
1074 L=V:GOTO1090
1080 W=VAL(A$):H=INT(W/256):L=W-H*256
1081 IFB$="CLC"THENG=24:GOTO1110
1082 IFB$="SEC"THENG=56:GOTO1110
1083 IFB$="JSR"THENS=32:GOTO1090
1084 IFB$="RTS"THENG=96:GOTO1110
1085 IFLEFT$(B$,2)="ST"THENS=129:GOTO109
   0
1086 IFLEFT$(B$,2)="LD"THENS=160
1090 FORX=STO255:IFM$(X)=I$THENG=X:X=255
   :NEXT:GOTO1110
1095 NEXT:PRINT"ERROR:[RVS]"I$" UNKNOWN"
   :END
1110 POKEP,G:P=P+1:IFNB(G)=0THEN1030
1120 IFNB(G)=1THENPOKEP,L:P=P+1:GOTO1030
1130 POKEP,L:POKEP+1,H:P=P+2:GOTO1030
2000 REM DISSASSEMBLE
2010 INPUT"START ADDRESS";P
2020 M=PEEK(P):M$=M$(M):IFM$="0"THENM$="
   [RVS]GARBAGE CODE":PRINTP,M$:P=P+1:GO
   TO2080
2030 IFNB(M)=0THENPRINTP,M$,A:P=P+1:GOTO
   2080
2040 IFNB(M)=1THENPRINTP,M$,PEEK(P+1):P=
   P+2:GOTO2080
2050 PRINTP,M$,PEEK(P+1)+PEEK(P+2)*256:P
   =P+3
2080 IFPEEK(197)<>60THEN2080
2090 GOTO2020
3000 PRINT"[CLS]*** MINI-ASSEMBLER ***[C
   D][CD][CD]"
3010 PRINT"LOADING DATA":GOSUB9000
3020 INPUT"[CD][CD]DISSASSEMBLE";Q$
3030 IFLEFT$(Q$,1)<>"Y"THEN1020
3035 PRINT
3040 GOTO2000
9000 DATA*^*,BRK,0,ORA!,1,0,0,0,0,0,0,OR
   AZ,1,ASLZ,1,0,0,PHP,0,ORA#,1,ASLA,1
9010 DATA0,0,0,0,ORA\,2,ASL\,2,0,0,BPL^,
   1,ORA?,1,0,0,0,0,0,0,ORA1,1,ASL1,1
9020 DATA0,0,CLC,0,ORAY,2,0,0,0,0,0,0,OR
   AX,2,ASLX,2,0,0
```

```
9030 DATAJSR↖,2,AND!,1,0,0,0,0,BITZ,1,AN
   DZ,1,ROLZ,1,0,0,PLP,0,AND#,1,ROLA,0
9040 DATA0,0,BIT↖,2,AND↖,2,ROL↖,2,0,0,BM
   I^,1,AND2,1,0,0,0,0,0,0,AND1,1
9050 DATAROL1,1,0,0,SEC,0,ANDY,2,0,0,0,0
   ,0,0,ANDX,2,ROLX,2,0,0
9060 DATARTI,0,EOR!,1,0,0,0,0,0,0,EORZ,1
   ,LSRZ,1,0,0,PHA,0,EOR#,1,LSRA,0,0,0
9070 DATAJMP↖,2,EOR↖,2,LSR↖,2,0,0,BVC^,1
   ,EOR?,1,0,0,0,0,0,0,EOR1,1,LSR1,0
9080 DATA0,0,CLI,0,EORY,2,0,0,0,0,0,0,EO
   RX,2,LSRX,2,0,0
9090 DATARTS,0,ADC!,1,0,0,0,0,0,0,ADCZ,1
   ,RORZ,0,0,0,PLA,0,ADC#,1,RORA,0,0,0
9100 DATAJMP$,2,ADC↖,2,ROR↖,2,0,0,BVS^,1
   ,ADC?,1,0,0,0,0,0,0,ADCZ,1,RORZ,1,0,0
9110 DATASEI,0,ADCY,2,0,0,0,0,0,0,ADCX,2
   ,RORX,2,0,0
9120 DATA0,0,STA!,1,0,0,0,0,STYZ,1,STAZ,
   1,STXZ,1,0,0,DEY,0,0,0,TXA,0,0,0
9130 DATASTY↖,2,STA↖,2,STX↖,2,0,0,BCC^,1
   ,STA?,1,0,0,0,0,STY1,1,STA1,1,STX1,1
9140 DATA0,0,TYA,0,STAY,2,TXS,0,0,0,0,0,
   STAX,2,0,0,0,0
9150 DATALDY#,1,LDA!,1,LDX#,1,0,0,LDYZ,1
   ,LDAZ,1,LDX,1,0,0,TAY,0,LDA#,1,TAX,0
9160 DATA0,0,LDY↖,2,LDA↖,2,LDX↖,2,0,0,BC
   S^,1,LDA?,1,0,0,0,0,LDY1,1
9170 DATALDA1,1,LDX2,1,0,0,CLV,0,LDAY,2,
   TSX,0,0,0,LDYX,2,LDAX,2,LDXY,2,0,0
9176 DATACPY#,1,CMP!,1,0,0,0,0,CPYZ,1,CM
   PZ,1,DECZ,1,0,0,INY,0,CMP#,1,DEX,0,0,
   0
9177 DATACPY↖,2,CMP↖,2,DEC↖,2,0,0,BNE^,1
   ,CMP?,1,0,0,0,0,0,0,CMP1,1,DEC1,1
9178 DATA0,0,CLD,0,CMPY,2,0,0,0,0,0,0,0,CM
   PX,2,DECX,2,0,0
9184 DATACPX#,1,SBC!,1,0,0,0,0,CPXZ,1,SB
   CZ,1,INCZ,1,0,0,INX,0,SBC#,1,NOP,0
9185 DATA0,0,CPX↖,2,SBC↖,2,INC↖,2,0,0,BE
   Q^,1,SBC?,1,0,0,0,0,0,0
9190 DATASBC1,1,INC1,1,0,0,SED,0,SBCY,2,
   0,0,0,0,0,0,0,SBCX,2,INCX,2,0,0
9200 RESTORE
```

```
9210 READA$:IFA$<>"*^*"THEN9210
9220 DIM M$(255),NB(255),Q$(10,20)
9230 FORA=0TO255:READM$(A),NB(A):NEXT
9240 RETURN
```

User-definable Function Keys

This routine sets up the key F1 to type a maximum of 10 characters by a single depression. The 'phrase' is stored at 820 onwards and is terminated by a 13 which effects a carriage return. Using this routine, you can set up the key to clear the screen and LIST or RUN the program. It's hooked on to the interrupt so that, once installed, you can forget it and NEW the loader.

```
*49152
LDA£197
CMP#4
BNE ↑ 49174
LDX#0
LDAX820
STAX631
INX
CMP#13
BNE ↑ 49161
STXZ198
JMP£59953
```

Suppose we want to set up the key to print 'LIST' followed by a return. Use the following routine:

```
T$="LIST"+CHR$(13)
FOR A=1 TO LEN(T$) : POKE 819+A,ASC(MID$(T$,A,1)) :
NEXT A
```

Don't forget to re-point the interrupt vector:

```
POKE788,0 : POKE789,192
```

Dynamic Halt

This routine allows you to stop the entire machine whatever it's doing (apart from during tape operations). When F3 is depressed, the machine enters a 'dynamic halt' until the space-bar is depressed. This is particularly useful when inspecting or debugging a listing and the CTRL

key is no use. The internal clock, however, does stop during the halt.

```
LDAZ197
CMP#5
BNE ↑ 49167
JSR£65439     Scan keyboard
LDAZ197
CMP#60        Space pressed?
BNE ↑ 49158
JMP£59953
```

Once again (as with all these routines), the interrupt connection must be made.

Quotes-mode Alleviator

You know the feeling? You're editing a line with a PRINT staement in, and somehow you've got into quotes mode and the cursor controls seem to be working against you. This routine restores sanity at the touch of F5 and plucks you out of quotes mode:

```
LDAZ197
CMP#6
BNE ↑ 49162
LDA#0
STAZ212       Exit quotes mode
JMP£59953
```

Sound and Sprite Mute

Playing around with sound and sprites often leaves you with a screenful of sprites and a headache-inducing wail from the television. One quick answer is the RUN/STOP + RESTORE combination. This, however, clears the screen of useful information and resets the interrupt (disabling these routines). This routine endows F7 with the mute command — it's just touch and go!

```
LDAZ197
CMP#3
BNE ↑ 49166
LDA#0
STA£54296     Mute sound
STA£53269     Mute sprites
JMP£59953
```

Mini-toolkit

All these routines start at 49152. To mix them, you'll have to re-point the branches and chain them together — a good exercise to test your proficiency. The following loader program simply reads in a machine code program that combines all four of these routines and re-points the interrupt. Once this is done, you may NEW the program and carry on programming as normal, except you now have a Mini-toolkit on board.

```
1     REM **MINI-TOOLKIT
2     REM ENABLE : POKE 788,0 : POKE 789,192
3     REM DISABLE : POKE 788,49 : POKE 789,234
10    P=49152
20    READ D : POKE P,D : P=P+1 : GOTO 20
6000  DATA 165,197,201,4,240,15,201,5,240,29,201,6,240,37,201,3,240
6010  DATA 43,76,49,234,162,0,189,52,3,157,119,2,232,201,13,208,245
6020  DATA 134,198,76,49,234,32,159,255,165,197,201,60,208,247,76,49
6030  DATA 234,169,0,133,212,32,228,255,76,49,234,169,0,141,24,212
6040  DATA 141,21,208,76,49,234
```

Type 'RUN'.
Wait for '?OUT OF DATA ERROR'.
Enable as above.
NEW program.

F1: Type string at 820.
F3: Enter dynamic halt.
Space-bar continues.
F5: Escape from quotes mode.
F7: Mute sound and remove sprites.

The toolkit is totally relocatable — all branches are relative. Incidentally, if you have a joystick in PORT 1 then the FIRE button becomes another space-bar.

APPENDIX C
Advanced Machine Characteristics

THE KEYBOARD AND ITS BUFFER

The section on interaction (Chapter 9) briefly skips over the keyboard and its operation. However, an in-depth understanding of its operation allows certain tricks in programming to be used. In some routines in this book, I utilise the buffer to achieve certain things. In simple terms, the buffer is simply a list of keyboard depressions that haven't yet been printed on the screen.

Every sixtieth of a second, our friend the interrupt runs round and looks at the keyboard. If a new key has been pressed, this is recorded in the buffer and the buffer pointer incremented. If the machine's cursor is flashing, then another routine empties the buffer. Thus while the cursor isn't flashing the buffer fills up — to a maximum of 10 characters. If you list a program and jab at the keyboard, after the listing you will find a list of the keys you hit.

The buffer runs from 631–640. The buffer pointer is 198. As we know where it is, there's nothing to stop us seeding the buffer with characters. The characters are stored in ASCII format and this is exactly the technique used for defining function keys. A command such as LIST is entered into the buffer along with a RETURN (code 13). If you find the keyboard buffer is causing problems then its size can be reduced by POKEing in a new size to location 649. Don't exceed 10 as otherwise the buffer will stray into dangerous territory (the operating system variables).

The following routine POKEs M$ into the buffer.

```
FOR A=1 TO LEN(M$) : POKE 630+A,ASC(MID$(M$,A,1)) :
NEXT
POKE 198,LEN(M$)
```

Clever use of the buffer allows programs to alter themselves by coming across an END or STOP statement and then executing 'auto-returns' over a new line, and a CONT or GOTO command which picks up execution again. This is rather straying from the purpose of this book however.

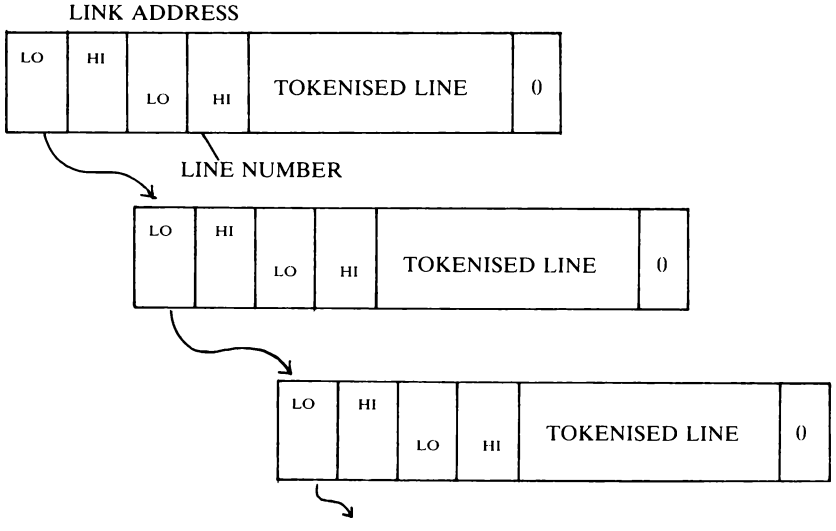
BASIC TEXT AND VARIABLE STORAGE

Although strictly not of machine code relevance, this section does have indirect connections and, anyway, it's not in the *Programmer's Reference Guide.* Of particular importance is the description of the floating-point variable as this is the format in which data is passed via the USR(X) function. The way in which programs and variables are stored on the 64 should allow you to write your own routines such as a renumber and trace routine.

Programs start at location 2049. Each line is stored as a record in a file and is held in ascending order of line number. The format of each record is as follows.

The first two bytes contain what is known as the 'link address'. These two bytes constitute the address of the start of the next record. This makes it easy to chase through the program without having to search every character. The next two bytes contain the line number. This is why there is a limit placed on the magnitude of line numbers. After this the record is simply a string of bytes terminated by a zero byte. The next byte is the lo-byte of the next link address. Keywords such as GOTO, GOSUB and PRINT are represented by 'tokens'. These are one-byte values unique to that keyword and thus represent a large saving on memory. **Figure C.1** explains this structure.

Figure C.1: BASIC Text Storage.



Locations 45 and 46 hold the start of the BASIC variables. There are many types: real, integer, real array, integer array, string, string array. For each type, there is a particular representation. Real variables are stored in seven bytes. The first two hold the ASCII values of the name.

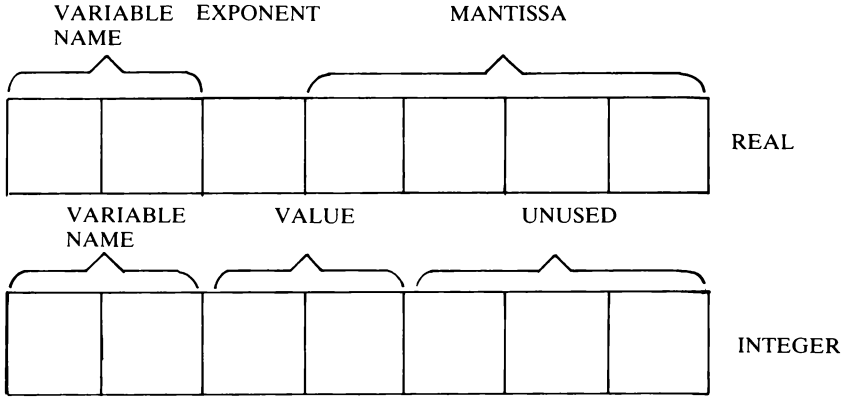
Thus, as far as storage is concerned, a variable takes up just as much space as two characters as it would as one. The next five bytes constitute the value. Scientific notation is used. The method is that every number can be represented in the following manner:

$\pm$ a $\times$ 10^x where $0 < a < 1$ and x is an integer.

The only difference on the 64 is that 10 isn't used. As it's a computer, 2 is used. The 'x' or 'exponent' is stored in the first byte. To allow the use of negative numbers it is stored in 'excess 129' notation. This means that 129 is added to its real value. Thus the actual range of x is -129 to 126. Don't think that a negative exponent means a negative number — it simply means very small (10^{-3} = .001).

The 'mantissa' (the 'a' part) is stored in the next four bytes. It is stored in Binary Coded Decimal form. This means that each byte is split up into a pair of 4-bit nybbles, each representing a number. This means that 8-bit precision is attained. The first bit of the first mantissa byte contains a sign bit.

Figure C.2: Real and Integer Variable Storage.



For the sake of completeness, I'll describe the integer variable. Once again a seven-byte representation is used. The first two hold the name, distinguished from 'real' by having ASCII codes + 128. The next two bytes hold the lo-hi values, followed by three dummy zeros (you only economise on integer arrays). **Figure C.2** should graphically present these methods of variable storage.

THE CIAs

In the beginning there was the VIA — the Versatile Interface Adaptor.

This was (and still is) described as a remarkably flexible peripheral chip for both I/O and timing.

The 64 contains not one but two CIAs — Complex Interface Adaptor. These are extremely powerful chips, only nobody seems to use them much. One of the CIAs is in charge of the IRQ interrupts. Each of the CIAs contains a full am/pm clock, two connectable 16-bit timers, an alarm feature and a full handshaking capability. For the purposes of this book I will just discuss the timers and their control for IRQ scheduling. The *Programmer's Reference Guide* describes them briefly in 'jargonese' but for a full appreciation the data sheets from the manufacturers are required.

The two timers are known as A and B. For each timer there is a 16-bit latch from which they count down. Thus when the timer reaches zero the value in the latch is reloaded ready for the next countdown. An IRQ is also generated when either reaches zero. Each timer can operate in either a one-shot mode or continuously. In the one-shot mode, the timer stops on reaching zero and then latches in again. It then awaits a start signal. In the continuous mode the value is repeatedly relatched ad infinitum. The timers count 1 for every cycle of the main CPU clock (1,000,000 Hz). On switching on, timer A is initialised to interrupt every sixtieth of a second.

Both timers are controlled by a timer control register. The lower five bits perform the following functions:

Bit 0: Start/stop. This bit starts and stop the timer. If set, then the timer is running.
Bit 1: Not relevant here.
Bit 2: Not relevant here.
Bit 3: SET = one-shot. CLEAR = continuous.
Bit 4: If SET this immediately relatches the start value, whatever the timer is doing.

So that you can use the timers and not interrupts for timing, an interrupt mask is provided. This simply refuses to let an IRQ originate from that source. The timers may be individually masked. On switching on, timer A is capable of interrupting while timer B is refused. The masking register is 56333. Bit 7 determines whether to clear or set a mask. If bit 7 is clear, then any ones written to 56333 will remove the corresponding source's ability to generate an interrupt. The converse is true for a zero. Timer A is bit 0 and B is bit 1. Timer B can be made to count, not clock pulses, but underflows from timer A. This means that if timer A has an interrupt mask on, the two timers work together to create a maximum of a 32-bit timer. The count method is dictated by the state of bits 5 and 6 in 56335: 00 means 'count clock pulses'; 10 means 'count timer A underflows'.

To set a latch value, simply POKE it into the lo-hi timer location. PEEKing these supplies the present count. The following table supplies all the locations we are going to need for CIA#1:

56324	Timer A lo-byte.
56325	Timer A hi-byte.
56326	Timer B lo-byte.
56327	Timer B hi-byte.
56333	IRQ control register.
56334	Timer A control register.
56335	Timer B control register.

Let's watch the timers in action. The flashing cursor is flashed via the interrupt. Try the following POKE and watch the cursor:

```
POKE 56325,20
```

Try the repeat facility on the cursor keys — notice anything? The cursor is flashing three times as fast! If you want your interrupt routines to work faster, then this is the way to do it. Don't forget that TI$ is also changing three times as fast. Let's bring in timer B:

```
POKE 56335,1          Switch on.
POKE 56327,64         Set latch.
POKE 56333,127+1+2    Remove IRQ mask.
```

The result of this is that timer A is causing interrupts every hundred and eightieth of a second, while B is working a third as quickly. Play around; it's the only way to learn. Try not to disable all IRQ lines as this causes a loss of keyboard interaction (and other things).

The second CIA has all these facilities, but it's not connected to the IRQ line. Both CIAs can send and receive these pulse trains through the ports with great ease. Here I have merely touched on the use of a CIA to generate a complex pulse train. A whole book could be devoted to their use — don't stop here. If the IRQ is the key to the 64, the CIA #1 is the key to the IRQ.

SID AND VIC MEMORY MAPS

Despite being informative and concise, the *Programmer's Reference Guide* tends to demand a lot of page-turning when writing programs. This is due to the complexity of the SID and VIC-II chips — it takes a long time to learn all their memory locations off by heart. Furthermore the format is not easy to scan through quickly. Here I present some

Figure C.3: SID Reference Chart.

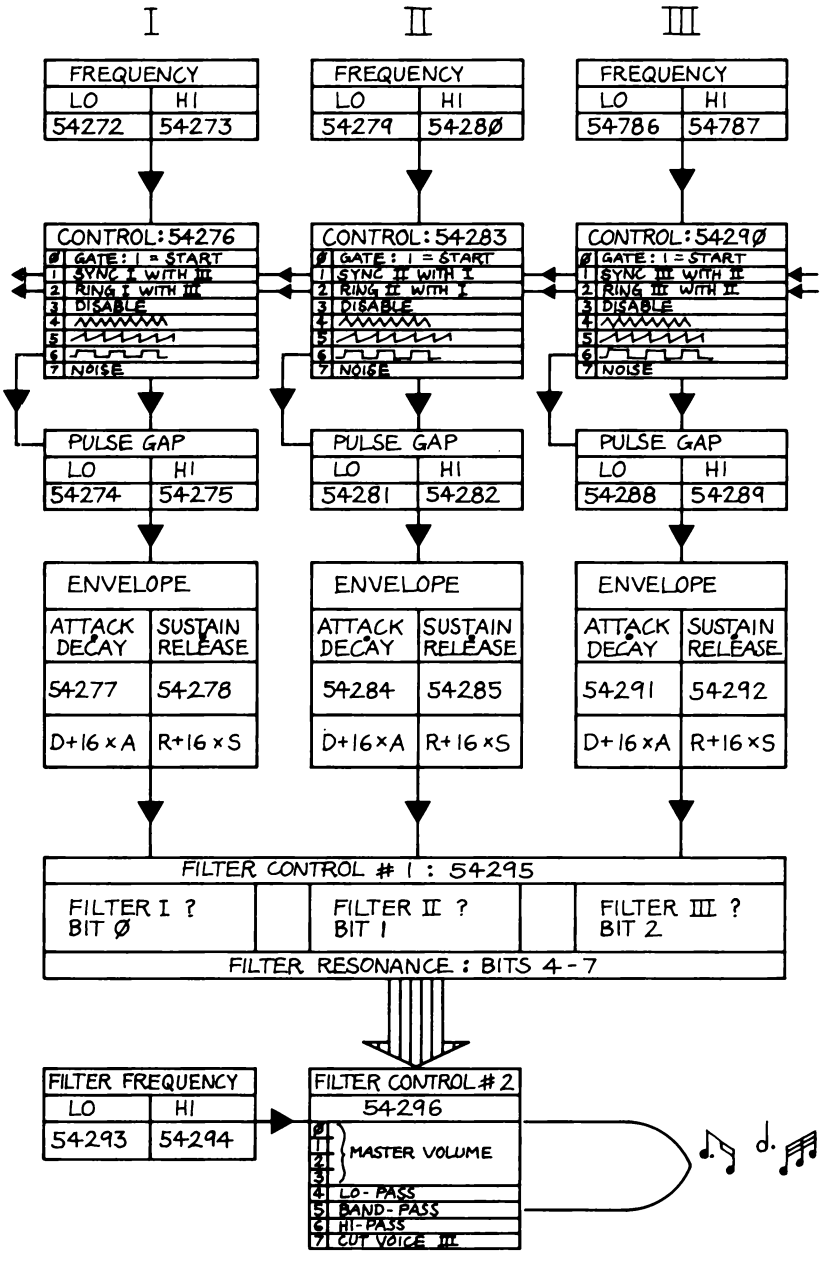


Table C.1: Sprite Reference Chart.

SPRITE	CTRL VALUE	X	Y	COLOUR	M.POINTER
0	1	53248	53249	53287	2040
1	2	53250	53251	53288	2041
2	4	53252	53253	53289	2042
3	8	53254	53255	53290	2043
4	16	53256	53257	53291	2044
5	32	53258	53259	53292	2045
6	64	53259	53260	53293	2046
7	128	53260	53262	53294	2047

Use ConTRoL value to set bits in the following sprite control registers:

53264 Most significant bit of sprite X coordinate.
53271 Expand sprite vertically.
53277 Expand sprite horizontally.
53278 Sprite + sprite collision log.
53279 Sprite + background collision log.

53280 Colour of the border.
53281 Colour of the screen (paper).

0	Black	1	White
2	Red	3	Cyan
4	Purple	5	Green
6	Blue	7	Yellow
8	Orange	9	Brown
10	Light red	11	Gray #1
12	Gray #2	13	Light green
14	Light blue	15	Gray #3

charts which I use. They show clearly what each location does and how to use them. The SID diagram in particular (**Figure C.3**) helps show the complex arrangement of voices and filters, the structure of which is difficult to perceive otherwise.

NB. These charts assume you already understand the operation of the SID and VIC-II chips. They make no attempt to teach their function and use.

APPENDIX D
Laserbike

Here is a game that uses a few of the ideas in this book. You have to ride a bike around a cylindrical world of which only a small portion is ever visible. Everywhere you go, you leave an uncrossable blazing trail — to cross it means death.

The objective of the game is to collect all the keys ('$' signs) required on that level and exit through the Time-Gate ('#' sign). To keep you alert, there are bombs littered around the playing area — if one goes off then it's curtains. The Navascan directs you to these so that you can defuse them. You have three bikes, and if you stop moving they may overheat. Extra points may be gained by picking up bonus fruit ('*') on the way. To move the bike use ',' for left; '.' for right; 'Q' for up; 'A' for down.

As it stands, the game is interesting but not brilliant: it's simply designed to show you the sort of things possible with machine code backup. Two machine code routines are used: the window routine which projects the window on to the screen; and a fill routine which clears out the playing area after each play. You might like to spice things up a bit, but 'Laserbike' was only written in one afternoon to test out the window routine. Improvements might be to hook a UDG switch on to the interrupt, so that the trail really blazes, and you could improve on the standard Commodore Graphics used.

```
0 REM *****LASER BIKE*****
10 GOSUB9000
12 GOSUB9500
15 DEF FNR(X)=INT(X*RND(1)+1)
30 GOSUB1000:REM GAME
31 PRINT"[HOM][CD]"
35 M$="GAME OVER":GOSUB10000
40 PRINT"[CD][CD]YOU SCORED "SC
50 IFSC>HSTHENPRINT"[RVS][CR][CR][CR][CR
   ][CR] A NEW HIGH SCORE!!":HS=SC
55 FORT=1TO3000:NEXT
60 GOTO30
```

```
1000 REM GAME
1010 LV=0:SC=0:LB=3:TP=0:CT=1653
1011 S$="[HOM][CD][CD][CD][CD][CD][CD][C
   D][CD][CD][CD][CD][CD][CD][CD][CD][CD
   ][CD][CD][CD][CD][CR][CR][CR][CR][CR]
   [CR][CR][CR][CR][CR][CR][CR][CR][CR][
   CR][CR]"
1012 S1$=S$+"[CD][CD]":B$=S$+"[RVS]----
   ":B1$=S1$+"[RVS]----"
1013 L$=S$+"[RVS]LEFT":R$=S$+"[RVS]RGH
   T"
1014 U$=S1$+"[RVS]-UP-":D$=S1$+"[RVS]D
   OWN"
1015 CC=0:LV=LV+1:GOSUB6000:GOSUB8000
1020 GOSUB5000:REM STATUS
1025 NK=3+INT(LV/5)
1030 FORA=1TOLV*10:POKE49266+FNR(59)+FNR
   (63)*60,42:NEXT
1032 FORA=1TOLV*50:POKE49266+FNR(59)+FNR
   (63)*60,46:NEXT
1033 BL=49266+FNR(59)+FNR(63)*60:IFPEEK(
   BL)<>32THEN1033
1034 POKEBL,30:BD=255
1040 FORA=1TO(NK*2+1):L=49266+FNR(59)+FN
   R(63)*60:IFPEEK(L)<>32THENA=A-1:NEXT
1050 POKEL,36:NEXT:POKEL,35
1060 REM NOW PLAY GRID
1070 PL=52000:D=1
1080 K=PEEK(197):C(0)=C(D):D=0:IFK=47THE
   ND=2
1085 POKEPL+305,160
1090 IFK=44THEND=1
1100 IFK=62THEND=3
1110 IFK=10THEND=4
1115 IFD=0THENTP=TP+0.2:IFINT(TP-.05)<IN
   T(TP)THENCT=CT+1:GOSUB5000
1120 PL=PL+V(D):H=INT(PL/256):L=PL-H*256
1121 Q=PEEK(PL+305):IFD=0THENQ=1
1122 IFQ=32THEN1130
1123 IFQ=42THENSC=SC+15*LV:GOSUB5000:POK
   E54276,0:POKE54276,33
1124 IFQ=36THENCC=CC+1:SC=SC+100:GOSUB50
   00:POKE54276,0:POKE54276,17
1125 IFQ=35ANDNK=CCTHENGOSUB9600:GOTO101
```

```
   5
1126 IFQ=35THENRETURN
1127 IFQ=30THENGOSUB5900
1128 IFQ=46THENGOSUB5500
1129 IFQ=160THENGOSUB5500
1130 POKE253,L:POKE254,H
1135 POKEPL+305,C(D)
1140 SYS49152
1145 D1=(PL+305)-BL:V1=INT(D1/60):H1=D1-
   V1*60:IFH1<0THENH1=H1+60
1146 IFH1=0THENPRINTB$:GOTO1149
1147 IFH1<30THENPRINTL$:GOTO1149
1148 PRINTR$
1149 IFV1=0 THENPRINTB1$:GOTO1160
1150 IFV1<0 THENPRINTD$:GOTO1160
1151 PRINTU$
1160 BD=BD-1:IFBD=0THENGOSUB5500
1999 GOTO1080
5000 PRINT"[HOM][CD][CD][CD][CD][CD][CR]
   [CR][CR][CR][CR][CR]"LV
5005 POKECT,81:IFCT=1656THENCT=1653:TP=0
   :FORA=CTTOCT+3:POKEA,173:NEXT:GOTO550
   0
5010 PRINT"[CD][CR][CR][CR][CR][CR][CR][
   CR][CR]"NK
5020 PRINT"[CD][CD][CR][CR][CR][CR][CR][
   CR][CR][CR]"CC
5030 FORA=1157 TO (1155+2*LB)STEP2:POKEA
   ,107:NEXT:POKEA,32
5035 PRINT"[HOM]"TAB(10)HS:PRINT"[HOM]"T
   AB(31)SC
5040 RETURN
5500 REM BIKE DEATH
5505 POKE54276,0:POKE54276,129
5510 FORT=1TO3000:NEXT
5520 LB=LB-1:TP=0:PL=PL-V(D):POKEPL+305,
   32:D=0
5530 IFLBTHENGOTO5000
5540 GOTO31
5900 BL=49266+FNR(59)+FNR(63)*60:IFPEEK(
   BL)<>32THEN5900
5910 POKEBL,30:BD=255:RETURN
6000 REM LASER-BIKE CONSOLE
6010 PRINT"[CLS]HI-SCORE:
```

```
    SCORE:       "
6020 PRINT"[RVS]/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/
   \/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\"
6025 PRINT"
    "
6026 PRINT"
    "
6030 PRINT"[WHT]LEVEL:0      [RVS]XXXXXXXXXXX
   XXXXXX
6040 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6050 PRINT"NO.KEYS:          [RVS]XX[OFF]
       [RVS]XX
6060 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6070 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6080 PRINT"NO.HELD:          [RVS]XX[OFF]
       [RVS]XX
6090 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6100 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6110 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6120 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6130 PRINT"                  [RVS]XX[OFF]
       [RVS]XX
6140 PRINT"                  [RVS]XXXXXX[OFF]▼
   [RVS]XXXXXX
6150 PRINT"                      [RVS]NAVASCAN
6160 PRINT"                       [RVS]       [OF
   F]
6170 PRINT"                       [RVS]       [OF
   F]
6180 PRINT"                       [RVS]       [OF
   F]  <  <
6190 PRINT"                       [RVS]       [OF
   F]
6200 PRINT"                       [RVS]       [OF
   F]
6210 PRINT"[CD][YLW]  [RVS]***LASER-BIKE
   ***  (C)1983 PAUL ROPER[HOM]"
```

```
6220 PRINT"[HOM][CD][CD][CD][CD][CD][CD]
   [CD][CD][CD][CD][CD][CD]"
6230 PRINTTAB(25)"     [RVS]◤  ◥
6240 PRINTTAB(25) "  [RVS]◤┌TEMP┐◥"
6250 PRINTTAB(25)" [RVS]LO├----┼HI"
6260 PRINTTAB(25)"    [RVS]└────┘"
6999 RETURN
8000 REM SETUP
8010 SYS820
8020 FORA=49205TO49264:POKEA,160:NEXT
8040 FORA=53105TO53164:POKEA,160:NEXT
8050 RETURN
8500 FORA=1TOLV*10:POKE49266+FNR(58)+FNR
   (63)*60,42:NEXT
8510 FORA=1TO4:L=49266+FNR(58)+FNR(63)*6
   0
8520 IFPEEK(L)<>32THENA=A-1:NEXT
8530 POKEL,36:NEXT
8540 POKEL,35:RETURN
9000 REM M/C FOR SCANNER
9010 DATA169,253,133,251
9020 DATA169,4,133,252
9030 DATA162,0,160,0
9040 DATA177,253,145,251
9050 DATA200,192,10,208
9060 DATA247,232,24,165
9070 DATA253,105,60,133
9080 DATA253,165,254,105
9090 DATA0,133,254,24
9100 DATA165,251,105,40
9110 DATA133,251,165,252
9120 DATA105,0,133,252
9130 DATA224,10,208,214,96,*
9140 RESTORE:P=49152
9150 READD$:IFD$="*"THEN9200
9160 POKEP,VAL(D$):P=P+1:GOTO9150
9200 DATA 107,1,115,-1,113,-60,114,60
9210 FORA=1TO4:READC(A),V(A):NEXT
9220 P=820
9230 READD:IFD=-1THENRETURN
9240 POKEP,D:P=P+1:GOTO9230
9250 DATA169,53,133,253
9260 DATA169,192,133,254
9270 DATA160,0,169,32
9280 DATA145,253,230,253
```

```
9290 DATA208,2,230,254
9300 DATA165,253,201,172
9310 DATA208,240,165,254
9320 DATA201,207,208,234,96,-1
9500 REM SOUND WORKSHOP
9510 FORS=54272TO54295:POKES,0:NEXT:POKE
   54296,15
9520 POKE54277,12:POKE54278,0
9530 POKE54273,10:RETURN
9600 REM SCREEN START
9610 FORA=30 TO100STEP25:POKE54276,0:POK
   E54276,129
9620 FORB=A TO A+35:POKE54273,B:POKE5328
   0,FNR(16)-1:NEXT:NEXT
9630 POKE53280,254:RETURN
10000 PRINTTAB(INT((40-LEN(M$))/2));
10010 FORA=1TOLEN(M$):PRINTMID$(M$,A,1);
   :FORT=1TO150:NEXT:NEXT
10020 PRINT"[CD][CD]":RETURN
11000 RETURN
```

Index of Routines

Routine	*Description*	*Location*
Alternative Sprite System	Character-based sprite system	Chapter 9, Alternative Sprite System
Array Add	Adds vector throughout array/table	Chapter 10, Fleet Movements
Array Delete	Deletes element in array	Chapter 10, Fleet Movements
Attribute Flasher	Automatically flashes any screen lines	Chapter 9, Inverting and Explosions
Array Search	Searches array/table	Chapter 10, Fleet Movements
Block Fill	Straightforward fill routine	Chapter 9, Filling Memory
Bomb Update	Scans screen and moves bombs down a character	Chapter 9, Updating Bombs
CHRGET Wedge	A special gateway into machine code	Appendix A, Entering Machine Code
Crash Recovery	System reset *without* program loss	Appendix A, Crash Recovery
Delay	Three different delay routines	Chapter 9, Delays
Dynamic Halt	Freezes and restarts computer on request	Appendix B, Dynamic Halt
***Defender* Landscape Window**	Projects landscapes	Chapter 10, Large-scale Games

Function Keys Assign	Assigns a text string to F1	Appendix B, Function Keys
Interrupt Tunes	Plays tunes on the interrupt	Chapter 10, Sound
Invert	Two routines to flash the screen	Chapter 9, Inverting and Explosions
Line Blank	Scrolling accessory routine	Chapter 9, Scrolling
Memory Copy	Copies memory from one area to another	Chapter 9, Copying Memory
Mini-assembler		Appendix B, Mini-assembler
Mini-toolkit	Four handy function key assignments	Appendix B, Mini-assembler
Rectangle Fill	Fills in a rectangle in a different code to main area	Chapter 9, Filling Memory
Quotes-mode Alleviator	Sorts out the irksome cursor controls!	Appendix B, Quotes-mode Alleviator
Screen Scanning	Simply scans the screen	Chapter 6
Scroll	Three scrolling routines	Chapter 9, Scrolling
Sound and Sprite Mute	Cuts sound and sprites without data loss	Appendix B, Sound and Sprite Mute
Spiral Screen Fill	Fills screen from inside out as if drawing a spiral	Chapter 6
Sprite Collisions	Removing sprite collisions	Chapter 10, Sprites
Sprite Homer	Homes sprites 1–7 on to 0	Chapter 10, Non-linear Motion

Sprite on Joystick	Moves sprite 0 in accordance with joystick	Chapter 9, Interaction
Sprite Vectoring	Automatically moves sprites 0–7	Chapter 10, Sprites
String Printer	Prints out a given string	Chapter 9, Text and String Printing
Tune	Plays a tune stored in a table	Chapter 10, Sound
2D Window Projector	Projects 2D window on to screen	Chapter 10, Large-scale Games

Other titles from Sunshine

SPECTRUM BOOKS

Artificial Intelligence on the Spectrum Computer
Keith & Steven Brain — ISBN 0 946408 37 8 — **£6.95**

Spectrum Adventures
Tony Bridge & Roy Carnell — ISBN 0 946408 07 6 — **£5.95**

Machine Code Sprites and Graphics for the ZX Spectrum
John Durst — ISBN 0 946408 51 3 — **£6.95**

ZX Spectrum Astronomy
Maurice Gavin — ISBN 0 946408 24 6 — **£6.95**

Spectrum Machine Code Applications
David Laine — ISBN 0 946408 17 3 — **£6.95**

The Working Spectrum
David Lawrence — ISBN 0 946408 00 9 — **£5.95**

Inside Your Spectrum
Jeff Naylor & Diane Rogers — ISBN 0 946408 35 1 — **£6.95**

Master your ZX Microdrive
Andrew Pennell — ISBN 0 946408 19 X — **£6.95**

COMMODORE 64 BOOKS

Graphic Art for the Commodore 64
Boris Allan — ISBN 0 946408 15 7 — **£5.95**

DIY Robotics and Sensors on the Commodore Computer
John Billingsley — ISBN 0 946408 30 0 — **£6.95**

Artificial Intelligence on the Commodore 64
Keith & Steven Brain — ISBN 0 946408 29 7 — **£6.95**

Machine Code Sound and Graphics for the Commodore 64
Mark England & David Lawrence — ISBN 0 946408 28 9 — **£6.95**

Commodore 64 Adventures
Mike Grace — ISBN 0 946408 11 4 — **£5.95**

Business Applications for the Commodore 64
James Hall ISBN 0 946408 12 2 **£5.95**

Mathematics on the Commodore 64
Czes Kosniowski ISBN 0 946408 14 9 **£5.95**

Advanced Programming Techniques on the Commore 64
David Lawrence ISBN 0 946408 23 8 **£5.95**

The Working Commodore 64
David Lawrence ISBN 0 946408 02 5 **£5.95**

Commodore 64 Disk Companion
David Lawrence & Mark England ISBN 0 946408 49 1 **£7.95**

Commodore 64 Machine Code Master
David Lawrence & Mark England ISBN 0 946408 05 X **£6.95**

Programming for Education on the Commodore 64
John Scriven & Patrick Hall ISBN 0 946408 27 0 **£5.95**

Writing Strategy Games on your Commodore 64
John White ISBN 0 946408 54 8 **£6.95**

ELECTRON BOOKS

Graphic Art for the Electron Computer
Boris Allan ISBN 0 946408 20 3 **£5.95**

Programming for Education on the Electron Computer
John Scriven & Patrick Hall ISBN 0 946408 21 1 **£5.95**

BBC COMPUTER BOOKS

Functional Forth for the BBC Computer
Boris Allan ISBN 0 946408 04 1 **£5.95**

Graphic Art for the BBC Computer
Boris Allan ISBN 0 946408 08 4 **£5.95**

DIY Robotics and Sensors for the BBC Computer
John Billingsley ISBN 0 946408 13 0 **£6.95**

Artificial Intelligence on the BBC and Electron Computers
Keith & Steven Brain ISBN 0 946408 36 X **£6.95**

Essential Maths on the BBC and Electron Computers
Czes Kosniowski ISBN 0 946408 34 3 **£5.95**

Programming for Education on the BBC Computer
John Scriven & Patrick Hall ISBN 0 946408 10 6 **£5.95**

Making Music on the BBC Computer
Ian Waugh ISBN 0 946408 26 2 **£5.95**

DRAGON BOOKS

Advanced Sound & Graphics for the Dragon
Keith & Steven Brain ISBN 0 946408 06 8 **£5.95**

Artificial Intelligence on the Dragon Computer
Keith & Steven Brain ISBN 0 946408 33 5 **£6.95**

Dragon 32 Games Master
Keith & Steven Brain ISBN 0 946408 03 3 **£5.95**

The Working Dragon
David Lawrence ISBN 0 946408 01 7 **£5.95**

The Dragon Trainer
Brian Lloyd ISBN 0 946408 09 2 **£5.95**

ATARI BOOKS

Atari Adventures
Tony Bridge ISBN 0 946408 18 1 **£5.95**

Writing Strategy Games on your Atari Computer
John White ISBN 0 946408 22 X **£5.95**

SINCLAIR QL BOOKS

Introduction to Simulation Techniques on the Sinclair QL
John Cochrane ISBN 0 946408 45 9 **£6.95**

GENERAL

Home Applications on your Micro
Mike Grace ISBN 0 946408 50 5 **£6.95**

Sunshine also publishes

POPULAR COMPUTING WEEKLY

The first weekly magazine for home computer users. Each copy contains Top 10 charts of the best selling software and books and up-to-the-minute details of the latest games. Other features in the magazine include regular hardware and software reviews, programming hints, computer swap, adventure corner and pages of listing for the Spectrum, Dragon, BBC, VIC 20 and ZX 81 and other popular micros. Only 40p a week, a year's subscription costs £19.95 (£9.98 for six months) in the UK and £37.40 (£18.70 for six months) overseas.

DRAGON USER

The monthly magazine for all users of Dragon microcomputers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news related to the Dragon. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

MICRO ADVENTURER

The monthly magazine for everyone interested in Adventure games, war gaming and simulation/role-playing games. Includes reviews of all the latest software, lists of all the software available and programming advice. A year's subscription (12 issues) costs £10 in the UK and £16 overseas.

COMMODORE HORIZONS

The monthly magazine for all users of Commodore computers. Each issue contains reviews of software and peripherals, programming advice for beginners and advanced users, program listings, a technical advisory service and all the latest news. A year's subscription costs £10 in the UK and £16 overseas.

For further information contact:
Sunshine
12–13 Little Newport Street
London WC2R 3LD
01-437 4343
Telex: 296275

>Here's our way of saying
Thanks
for buying this Sunshine book...

>Sunshine Special Offer



Subscribe to Commodore Horizons, Britain's brightest Commodore magazine, and get an extra three months' issues absolutely free!

For just £10 you get a year's subscription plus the next 3 months without charge – a saving of £2.50. Complete and send the order card now, to reserve your special subscription, along with your cheque or postal order, or charge to your Access card.

See over for two other free Sunshine Offers

>See over for two other free Sunshine Offers ➡

>Sunshine Special Offers Claim Card

Yes, I'd like to take advantage of your Sunshine Special Offers.
Please make sure I get the following:

☐ A year's subscription to Commodore Horizons, plus 3 months free issues, for just £10.

☐ A year's subscription to Popular Computing Weekly, plus 12 issues free, for just £19.95.

☐ A copy of your free colour Sunshine Books brochure.

Payment methods:

☐ I enclose a cheque/postal order payable to Sunshine Books for £_____

☐ Charge my Access card No._____

Signed_____

Name_____

Address_____

_____

All offers end on 31st December 1984.

>Sunshine Special Offer

2

Popular Computing Weekly is the best-selling weekly newsmagazine for every computer whiz. And here's your opportunity to get 12 issues, worth £4.20, free of charge.

Use the special offer order card to claim your special Popular Computing Weekly subscription – 64 issues for the price of a year's copies – at a price of just £19.95. Send today, and look forward to 15 months of great weekly reading.

>Sunshine Special Offer

3

If you liked this Sunshine Book, take a look at the rest of our amazing range. Send today for our free colour brochure detailing all the Sunshine Books you can buy today, and all the new books on the way.

And once you're on our special mailing list, we'll keep you updated on what's new for you and your computer from Sunshine. Just tick the box on the special order card.

Send to:

Sunshine Books,
12/13 Little Newport Street,
London WC2R 3LD.

The magic of the power of the Commodore 64 lies in the use of machine code. This book will show you how to harness the enormous potential of the C64, opening up new vistas of programming possibilities.

In this book Paul Roper shows that machine code programming on the Commodore 64 can be simple to learn. He starts by showing how to use machine code from within BASIC programs and then explores the essential techniques required for programming the 6502 processor.

The approach to games programming is tackled in three ways. He introduces techniques of games design, writes and develops some complex machine code games and then presents a collection of machine code subroutines for you to use in your own games designs.

Topics such as scrolling, window projecting and sprite vectoring are typical of the type of ideas explored and explained. There is also a host of useful and essential machine code subroutines developed in this book, including an assembler and an editor.

Paul Roper has been interested in computers ever since the emergence of the first hobbyist machines. He is now living in Hampshire as a part-time freelance programmer and consultant.

GB £ NET +006.95

ISBN 0-946408-47-5

00695

9 780946 408474

SUNSHINE

ISBN 0 946408 47 5

£6.95net



This was brought to you

from the archives of

http://retro-commodore.eu