C64

Melbourne House

SUPERCHARGE

YOUR COMMODORE 64

BARRY THOMAS

Ready Made
Machine Language Routines

Super Charge Your Commodore 64

SUPERCHARGE YOUR COMMODORE 64

Ready-Made Machine Language Routines

Barry Thomas

Melbourne House Publishers

IN THE UNITED KINGDOM —
Melbourne House (Publishers) Ltd
Melbourne House
Church Yard
Tring Hertfordshire HP23 5LU

IN THE UNITED STATES OF AMERICA —
Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

IN AUSTRALIA —
Melbourne House (Australia) Pty Ltd
70 Park Street
South Melbourne Victoria 3205

Cataloguing in Publication

Thomas, Barry.
Super Charge Your Commodore 64.
ISBN 0 86161 174 8.
1. Commodore 64 (Computer) — Programming. I. Title.
001.64'2

EDITION 7 6 5 4 3 2 1
PRINTING F E D C B A 9 8 7 6 5 4 3 2 1
YEAR 90 89 88 87 86 85 84

Contents

Introduction

The Commodore 64 is an extremely powerful machine but, unfortunately, the version of the programming language BASIC built into the memory of the machine is rather lacking in tools which can exploit the full capabilities of the computer. This deficiency is remedied by the short programs contained in this book.

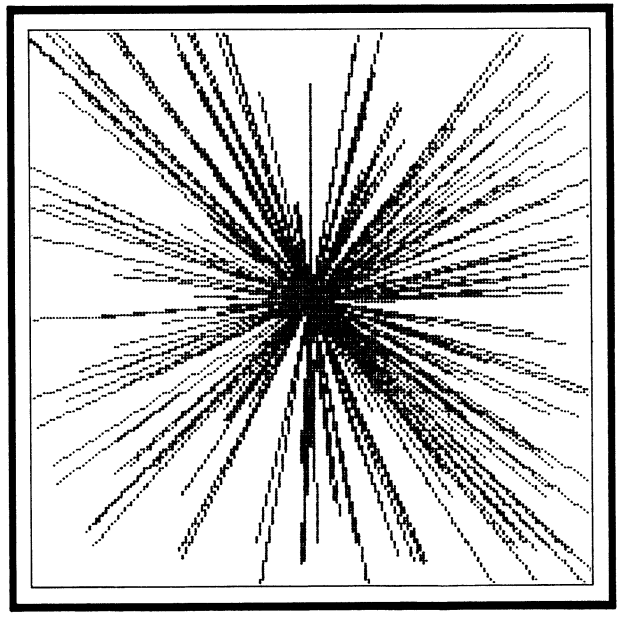
Included are routines to handle high-resolution graphics, sounds and sprites, and a collection of useful utility routines — including a program to renumber your BASIC programs and a bubble-sort routine which takes up only 31 bytes!

Because they are written in machine language, all of these routines run at staggering speeds compared to that of BASIC and occupy only a fraction of the memory used by a BASIC program written to do the same job. The speed and efficiency of machine language is a wonder to behold and, by building these routines into your own programs, your Commodore 64 will take on a new lease of life.

There are whole chapters on how the computer stores and deals with the binary numbers which make up machine language, and a brief look at the architecture of the microprocessor which makes the Commodore 64 tick. A whole chapter is devoted to the subjects of loading, saving and using the machine language programs contained in this book.

In the appendices are a collection of useful charts and memory references, along with some suggestions for further reading which will hopefully lead you to write your own machine language routines!

I owe grateful thanks to several people who have helped this book see the light of day: Fred Milgrom, for the original commission; Ian Logan and all at Melbourne House; Charles Wotton, for immeasurable enthusiasm and encouragement along the way; and Sharon, to whom goes the greatest thank-you of all for support and understanding above and beyond the call of duty.



1 Computer Numbers

COUNTING FOR HUMANS: DECIMAL

Most of us are used to counting with the decimal system of numbers, or numbers to the base 10. This arrangement probably came about simply because early humans found it convenient to use their fingers as counters. Thus, the decimal system employs ten digits, 0 to 9.

We can use each of the ten digits to represent decimal numbers on paper to equal 9 or less. After we have run through the digits 0 to 9, we need a way to show numbers that are greater than 9. To do this, we place a second digit to the left of the first, which indicates the number of times that we have used the full set of ten digits.

Using this method of representing numbers, the decimal number 24 shows that we have used the full sequence of digits twice (2 tens) and that we also have added the number 4 (4 units). We can break down the number 24 like this:

$$24 = (2 \text{ tens}) + (4 \text{ units})$$
$$= (2 \times 10) + (4 \times 1)$$

The number 324 is decomposed like this:

$$324 = (3 \text{ hundreds}) + (2 \text{ tens}) + (4 \text{ units})$$
$$= (3 \times 100) + (2 \times 10) + (4 \times 1)$$

So, in the decimal system, each digit of a number has ten times the magnitude of the next digit to the right.

Take that just a couple of steps further with the decimal number 1324. The number 1324 is decomposed like this:

$$1324 = (1 \text{ thousand}) + (3 \text{ hundreds}) + (2 \text{ tens}) + (4 \text{ units})$$
$$= (1 \times 1000) + (3 \times 100) + (2 \times 10) + (4 \times 1)$$
$$= (1 \times 10^3) + (3 \times 10^2) + (2 \times 10^1) + (4 \times 10^0)$$

In this last example, we have decomposed 1324 into powers of 10, where each digit is 10 times the digit to the right; figure 1.1 sums up.

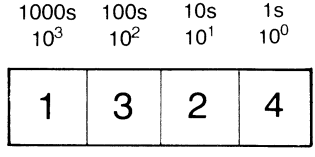


Figure 1.1 Decimal place value

Computers, however, have a different way of looking at things.

COUNTING FOR COMPUTERS: BINARY

Owing to the fact that the electricity which flows through the circuits within the computer can be only either off or on, the computer is limited to using two different digits only to represent these two possible states. That is, the digit 0 represents current off and the digit 1 represents current on.

This is known as the binary system, which represents numbers in base 2. (Base 10 uses ten digits but base 2 uses two digits only.) Whereas in decimal numbers each digit has ten times the magnitude of the digit to its right, in binary numbers each digit has only two times the magnitude of the digit to its right. Starting on the right, in binary numbers the first digit represents the number of ones, the next digit represents the number of twos, the next the fours, the next the eights and so on.

The decimal number 37 is represented in binary as 0010 0101 which, decomposed, looks like this (see figure 1.2):

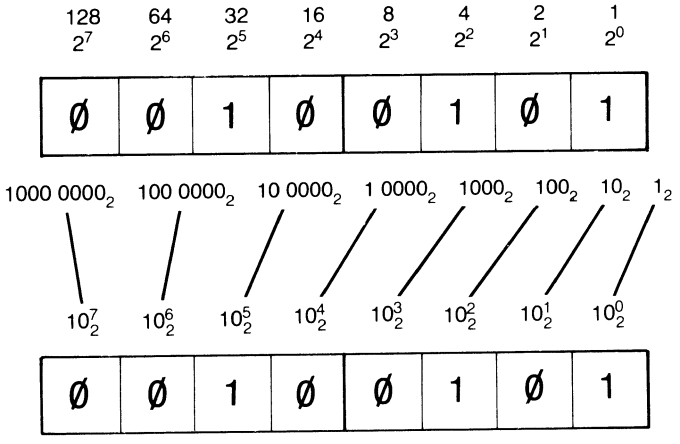


Figure 1.2 Binary place value

$$ØØ1Ø\ Ø1Ø1_2 = (Ø \times 128) + (Ø \times 64) + (1 \times 32) + (Ø \times 16) + (Ø \times 8) + (1 \times 4) + (Ø \times 2) + (1 \times 1)$$
$$= 37_{1Ø}$$

The digits in ØØ1Ø Ø1Ø1 are called *bits*, from **BI**nary digi**TS.** Most home computers handle these bits in groups of 8, known in computer jargon as a *byte*, or an eight-bit word. Each byte may hold a number in the range ØØØØ ØØØØ$_2$ to 1111 1111$_2$ (Ø to $255_{1Ø}$ in decimal), as shown in figure 1.3.

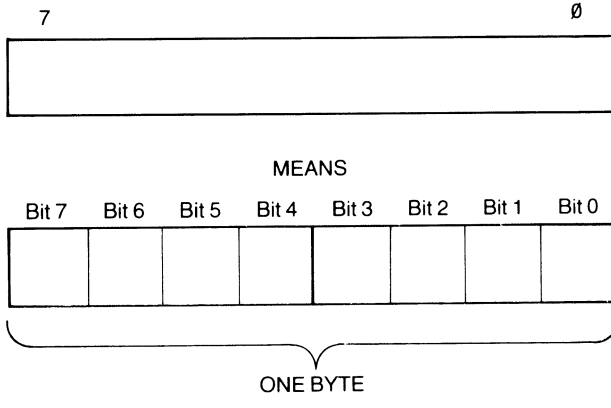


Figure 1.3 Bytes and bits

Binary numbers can also be taken four bits at a time to make a *nybble* or 16 bits at a time to make a 16-bit word or, for short, a *word*. These ways of describing binary numbers will be used often throughout this book, so take a good mental snapshot of figure 1.4.

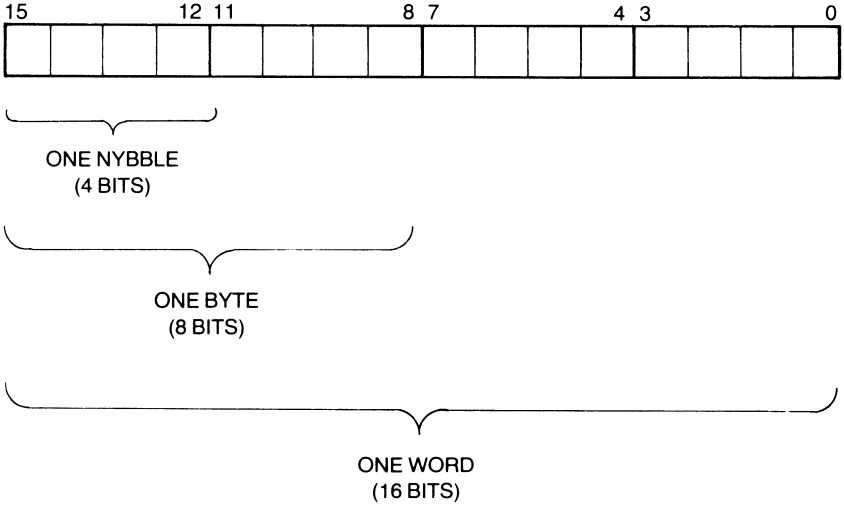


Figure 1.4 Nybbles, bytes and words

Two more terms which crop up later in this book and sometimes cause confusion are *most significant bit* and *least significant bit*. In the decimal number 1324, which we saw earlier, 1 has the largest value and is the *most significant digit* and 4 has the smallest value and is the *least significant digit* (see figure 1.5).

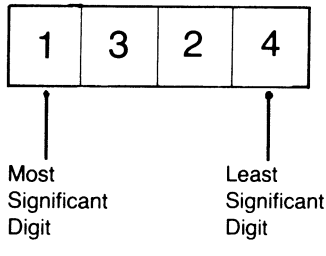


Figure 1.5 Significance of decimal digits

Similarly, in a byte, bit 7 is the *most significant bit* (or MSB) and bit Ø is the *least significant bit* (or LSB). See figure 1.6.

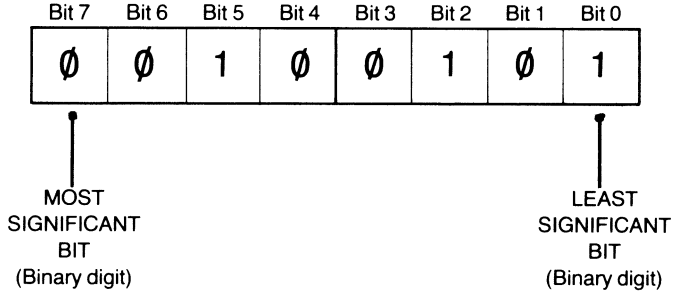
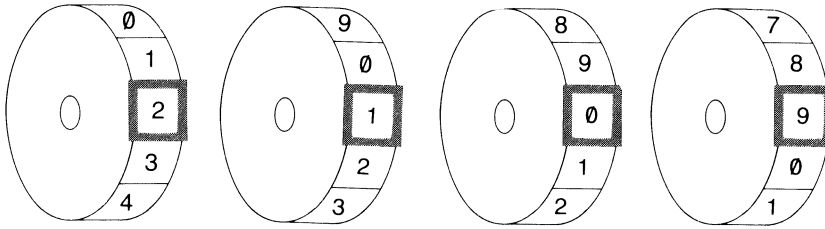


Figure 1.6 Significance of bits

Well! so far, so good. The binary system of counting now holds no mystery for us and any binary number in the range $Ø_{1Ø}$ to $255_{1Ø}$ is within our grasp.

The only drawback so far is that within one byte we can only represent *positive* numbers. Just imagine how difficult life would be without negative numbers, particularly as one of the features of machine language is that negative numbers are rather useful. It would be quite helpful if there was a method of indicating positive *and* negative numbers without using any more than just the eight bits in one byte.Of course, there is such a method, called *signed binary,* and this is how it operates.

Imagine a single dial from inside a car's odometer or out of the tape counter on a cassette recorder, as in figure 1.7. Assume the dial has the digits Ø to 9 around the perimeter and the dial is set to show 2 as in figure 1.7(a). When the 2 is showing to the front, then the reading is 2, of course. But, if we roll the dial *backwards* one digit, i.e. subtract 1, the reading changes to 1 (figure 1.7(b)). Another digit backwards and the dial reads Ø (figure 1.7(c)).



1.7(a) 1.7(b) 1.7(c) 1.7(d)

Figure 1.7 Negative numbers on speedometers (1)

This is simple so far but, if we subtract 1 once more, we *should* show −1 in the window. In fact, if we roll the dial back another digit, the reading is 9 (figure 1.7(d)). Suppose we take that 9 to represent −1, since it is one digit back from Ø. We can continue in this fashion, subtract the next digit and say that 8 represents −2, and so on. In a base 10 system we would call this method of representing *negative* numbers the *tens complement*.

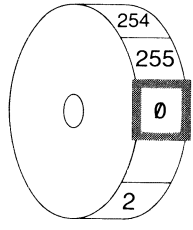


Figure 1.8 Negative numbers on speedometers (2)

Now, instead of a dial with ten digits on it (Ø to 9), we have a dial with 256 numbers on it, from Ø to 255 (see figure 1.8). It is clear that as we successively subtract 1 and pass backwards through zero then, by analogy with our previous example, −1 is represented by the number 255. Rewritten in binary notation the number 255 becomes the byte 1111 1111. Suppose we allow $1111\ 1111_2$ to represent −1; then by subtracting another digit $1111\ 111Ø_2$ must represent −2; and so on (see figure 1.9).

BINARY	DECIMAL
0111 1111	+127
0111 1110	+126
*	*
*	*
*	*
0000 0011	+3
0000 0010	+2
0000 0001	+1
0000 0000	0
1111 1111	−1
1111 1110	−2
1111 1101	−3
*	*
*	*
*	*
1000 0010	−126
1000 0001	−127
1000 0000	−128

Figure 1.9 Eight-bit signed numbers

Purists call this way of representing negative binary numbers the *twos complement* and the method used to generate signed binary numbers is called *complementing*. If you want to know more, then I recommend to you a book called *Programming the 6502* by Rodnay Zaks (see Appendix G).

In signed binary, zero lies in the middle of the range of numbers that our byte can represent, i.e. the range −128 to +127. As can be seen in figure 1.9, all of the negative numbers have a 1 as their MSB (most significant bit) and all of the positive numbers have a 0 as their MSB. This conveniently allows us to determine the sign of a number represented in signed binary by examining just this bit, the MSB, which is also known as the *sign bit*.

MORE COUNTING FOR HUMANS: HEXADECIMAL

So far, we have learnt about the binary system which is ideal for the on/off circuitry of the computer. However, if it takes eight bits to represent the number −128, just imagine how many bits would be required to represent a billion!

The virtue of the decimal system is that you can represent a large number such as 1000 000 in just seven digits. What is needed is a number system that, like decimal, can represent large numbers in as few digits as possible — yet bears some relationship to the binary system which so suits computers.

The system which fulfills these conditions is the *hexadecimal system*, to the base 16. With this system, instead of using only two different digits to build up our numbers as in binary notation, or 10 digits as in decimal, we use 16 digits! Starting with 0 we run through the digits 0 to 9 in the usual way but then, instead of carrying a 1 into the next column, we continue in the same column and work through the letters A to F and *then* carry a 1 into the next column, and repeat the process (see figure 1.10).

NYBBLE VALUES	HEXADECIMAL DIGIT	DECIMAL NUMBER
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15
1 0000	10	16

Figure 1.10 Number base conversions

By using this system we use only *two* digits to represent the full range of numbers that can be held in one byte. Why was base 16 chosen and not base 12 or base 19? If you look at figure 1.10 you will see that the digits 0 to F can represent all the values which can be held in a nybble. This means that a binary number can be quickly converted to a

hexadecimal number by breaking it into nybbles and replacing each nybble with a hexadecimal digit or *hex digit* (see figure 1.11).

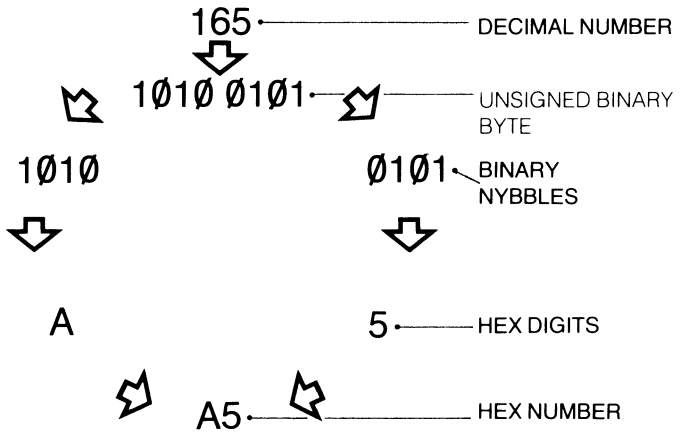


Figure 1.11 Decimal to binary to hexadecimal

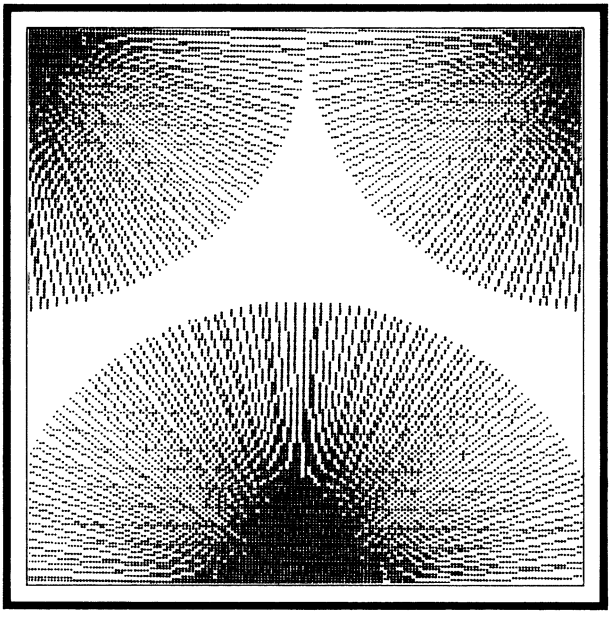
In a hex number, the lefthand digit represents the high nybble, or most significant nybble, while the righthand digit represents the low or least significant nybble.

NUMBER CONVENTIONS

The convention for showing to which base a number is written is to write decimal numbers as they are usually, and to precede hexadecimal numbers with a **$** sign and binary numbers with a % sign. For comparison:

DECIMAL	255
HEXADECIMAL	$FF
BINARY	%1111 1111

Whether a number is unsigned or signed is usually evident from the context. 'Hexadecimal numbers' is often abbreviated to *hex numbers*.



2 Machine Language Explained

WHAT IS MACHINE LANGUAGE?

At the heart of all home computers is one of the now famous *chips*. This is rather a vague term, so from now on I use the more correct term central processing unit, or CPU for short.

The Commodore 64 is no exception and is built around a CPU known as the 651Ø. A look at the 651Ø reveals it to be a small block of plastic with a row of pins attached to each side — not terribly impressive! But if we were to take the plastic casing away, we would see that the pins lead into a small square about the size of your little fingernail. This square is a fantastically complex electronic circuit etched into a tiny backing wafer of pure silicon, hence the popular name *silicon chip*.

This circuit is made up of many thousands of microscopic switches or gates which, being either open or closed at any one time, will either conduct or not conduct an electric current.

Fortunately, as programmers, we do not have to understand the technology behind these intricate devices. All we have to know is that when certain combinations of 'current' and 'no-current' (our binary 1s and Øs) are applied to the pins on one side of the CPU, the circuitry within will operate according to the set rules of logic to give us other patterns of 1s and Øs in reply. Not all patterns will cause the CPU to respond. To be effective, these patterns of 1s and Øs must be part of the recognised *instruction set* of that CPU. The set of recognised bit patterns are the *machine language* for that CPU.

WHAT IS ASSEMBLY LANGUAGE?

Suppose we had to add two numbers, say 1Ø and 41. On older computers with front-panel switches, we could enter the following instructions in binary to perform this task:

1Ø1Ø 1ØØ1	ØØØØ 1Ø1Ø	Load the accumulator with 10_{10}
Ø11Ø 1ØØ1	ØØ1Ø 1ØØ1	Add 41_{10} with carry
1ØØØ Ø1Ø1	1111 1Ø11	Store sum in memory location 251_{10}
ØØ11 11ØØ		Return to main program

What a job! First we would have to convert 1Ø and 41 to binary form and then toggle the front panel switches to enter all the bits without making a single mistake!

Perhaps we could use a hex keypad to form the two numbers. We would have to enter the following hex numbers:

A9	ØA	Load the accumulator with $1Ø_{10}$
69	29	Add 41_{10} with carry
85	FB	Store sum in memory location 251_{10}
6Ø		Return to main program

This version is an improvement in that we don't have to enter as many digits. But it is still necessary to convert 1Ø and 41 to hex and the program hex numbers are hard to understand. We could write a loader program in BASIC so that we could poke the program into memory and then cause it to run. We wouldn't need to convert the decimal numbers but, unfortunately, we would have to convert the hex instruction numbers to decimals instead!

By far the easiest method of getting a machine language into memory is by the use of an *assembler* program. These programs are available commercially on tape or disc, or as plug-in cartridges in ROM or *read-*

only memory. (See Appendix F for one such assembler program.) Their job is to act as a go-between binary instruction values and the programmer by expressing instructions in an English-like language. For example, a short program to add two numbers can by typed in like this:

```
LDA #10     ! Load the accumulator with 10₁₀
ADC #41     ! Add 41₁₀ with carry
STA 251     ! Store the sum in memory location 251₁₀
RTS         ! Return
```

The assembler program looks at each statement 'word' and checks to see whether that instruction will need an item of data to work on or an address to refer to. When this has been decided, it places the correct *binary* instruction values in the appropriate places in memory ready to be used. (Although the English-like statements were called words, the correct term is *mnemonic*, meaning *reminder*. The mnemonic LDA reminds you that the instruction is **L**oa**D** the **A**ccumulator.)

Most assemblers allow the use of address labels to make the process of writing long programs even easier. Comments to make the program easier to understand and debug also help us to remember exactly what each part of our program does if we return to it weeks or months later.

To return to our original question 'What is assembly language?' the answer is this:

It is the collection of mnemonics understood by an assembler program which converts the mnemonics into the correct binary values and assembles them, i.e. places them at the locations in memory specified by the programmer. Mnemonics are far easier to remember than hex values, and it is surprising the speed with which one can write, assemble, de-bug and use even a lengthy program once the features of the assembler are familiar.

THE 6510 MICROPROCESSOR

Many of the home computers on the market today are built around the 6502 CPU or microprocessor. It is used in the Apple II series, the ORIC 1 and ORIC ATMOS, the Acorn ELECTRON, BBC Models A and B, and the Commodore VIC 20. The Commodore 64 uses the 6510 CPU which is a direct descendant of the 6502, so that all programs written for the 6502 will run perfectly well on the 6510. The only differences between the two CPUs are in the layout of some of the connections between the internal circuitry and the outside world. These differences need not worry us as our interest is in the programming side of the 6510.

The 651Ø is an *eight-bit microprocessor*. All of the instructions we give to the microprocessor and all of the numbers or data with which it is to work must be presented to it in *eight-bit format*, i.e. one byte at a time. All of the results of any data processing occurring inside the microprocessor are returned to us in the same eight-bit format.

We now take a brief look inside the 651Ø to see the tools it has for us to be able to manipulate data so that we may build up a working program in machine language.

INSIDE THE 651Ø

Within there are storage areas called registers which have the ability to hold a number temporarily while we perform arithmetic or some other process upon it, that the instruction set allows. The 651Ø has four eight-bit registers and one 16-bit register (see figure 2.1), and a special-purpose register called the *status register*.

FIG. 2.1

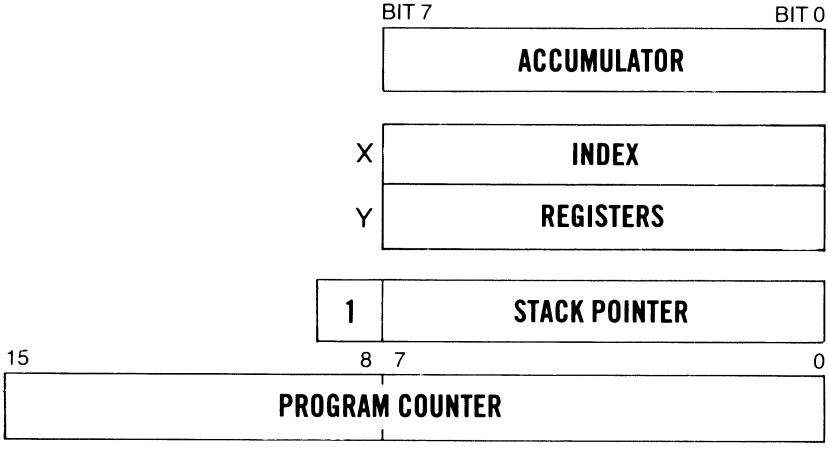


Figure 2.1 Registers in the 651Ø

The status register of the 651Ø, unlike the other registers, does not act as a singe byte. In the status register, each bit has its own individual and distinct use and meaning. We will cover the use of status register later in this chapter.

Accumulator

The accumulator is a single byte in size and is used to hold an unsigned binary number in the range Ø to 255, or, as we saw in chapter 1, to hold the signed binary values in the range −128 to +127. While a number is

resident in the accumulator we can add other numbers to it, subtract from it, compare it to a number in memory, transfer it to another register or perform any one of a whole range of actions upon it.

Index Registers or X and Y Registers

These two registers are also a byte in size and are able to contain unsigned and signed numbers in the ranges shown above. The X and Y registers may also be effectively used as counters since with just one instruction we can *increment* their contents (add 1) or *decrement* their contents (subtract 1).

Like the accumulator, these registers may be loaded with a value directly from within the program, or from memory, and store their contents to an address anywhere in the 64 K byte memory range of the Commodore 64. However, the most valuable use of the index registers is as a *variable offset* from a *base address* so that a whole block of memory can be accessed a byte at a time very quickly and easily by using the X or Y register. (More of this in chapters 4-7.)

Stack Pointer or SP

In computer terms, a stack is an area of memory which the CPU uses to temporarily store numbers as it goes about its work of handling our programs.

The Commodore 64 uses the locations 256_{10} to 511_{10} for its stack, so be careful with these addresses as the machine will not be very happy if you start altering numbers that it has stored there! The stack pointer is used by the CPU to remember the exact position of the next available byte in stack memory. The stack pointer register is not directly accessible to the programmer, so we can ignore it as the CPU will take care of the stack and its pointer without any help from us.

Program Counter or PC

The program counter is different to the other registers in the 6510 by being two bytes in size, while the others are only one byte. The program counter contains the address of the next instruction or item of data to be operated on. This register always contains a full 16-bit address. This register gives us the ability to access any address in the range 0 to 65535_{10} (or 0000 to $FFFF_{16}$), the full amount of memory available on the Commodore 64.

Memory addresses are usually given in hexadecimal format because they break down into pages of 256 bytes or blocks of 1024 bytes (1Kbyte). In hexadecimal, 256_{10} is 0100_{16} and 1Kbyte is 0400_{16}.

Status Register

The status register is best regarded as a set of eight individual bits, each of which has its own use and meaning, depending upon whether it is set to 1 or Ø. Of all of the registers in the 651Ø, the status or P register can at first be the most forbidding, but if we look at each bit in turn and examine its meaning then you will see just how little there is to be afraid of (see figure 2.2).

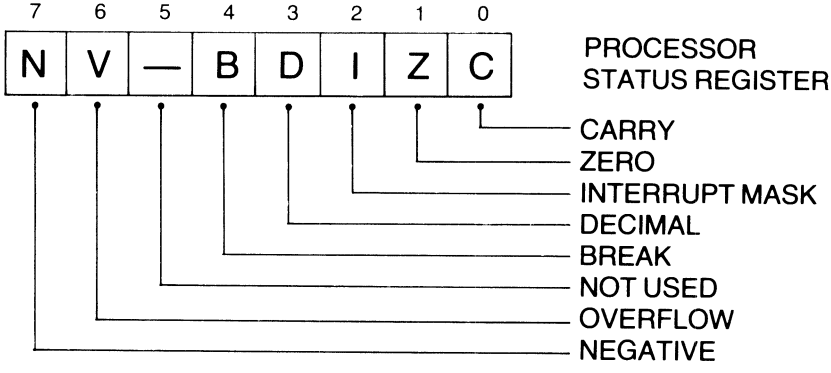
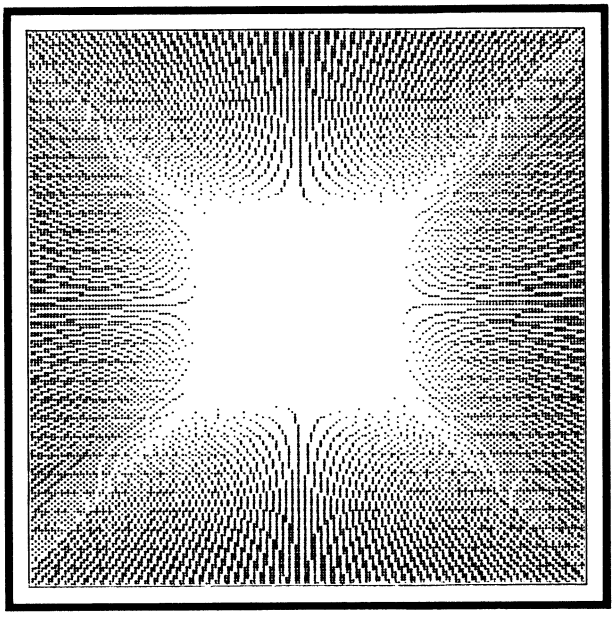


Figure 2.2 Status bits in the status register

- **Bit Ø CARRY** The carry bit is used by the processor for two different jobs:
  To show the presence of a carry or borrow after an arithmetic operation;
  To store the bit displaced by a shift or rotate operation.
  The carry bit may be directly set or cleared by the programmer.
- **Bit 1 ZERO** The zero bit is set to 1 whenever any arithmetic or comparison operation results in a value of Ø (zero).
- **Bit 2 INTERRUPT** The interrupt mask bit may be set by the programmer or, in certain circumstances, by the CPU itself. Its effect is to prevent any further interrupts from occurring.
- **Bit 3 DECIMAL** When the D bit is set the processor operates in binary-coded decimal mode. (For an explanation of binary-coded decimal (BCD), see *Programming the 65Ø2* by Rodnay Zaks (Sybex) or any good microprocessor reference.)
  When the D bit is reset to Ø the processor returns to binary operation.

- **Bit 4 BREAK** The break bit is set only after the BRK (break) instruction is encountered by the CPU. This instruction is primarily used when writing and debugging programs.
- **Bit 5 UNUSED**
- **Bit 6 OVERFLOW** The V bit is set to 1 when an overflow has occurred from bit 6 into bit 7 (the sign bit) after an arithmetic operation on signed binary numbers which, if allowed to pass unregarded, could lead to an error later in the program.
- **Bit 7 NEGATIVE** The N bit is set to 1 when any arithmetic or comparison operation results in a negative number. This bit is normally identical to the sign bit of the accumulator.

We have now studied the internal organs of the 651Ø and the ways in which it handles numbers, so let's look a little more closely at the process of writing a simple program.



3 Using Machine Language

USING AND READING FLOWCHARTS

First of all, let us just define exactly what a flowchart is. In the broadest sense of the term, a flowchart is a diagram which shows a process as a sequence of events or actions. There is a beginning and an end, and, we hope, a flow of logic in between!

A flowchart can be drawn to explain any process or group of events that follows a logical sequence. All of the programs in chapters 4 to 7 have their own flowcharts. Some of the charts are self-explanatory but, since others are a little more complex, a few words here on how they are built up are quite useful.

Compiling a flowchart for every program that we write is quite a good habit to acquire. It makes the actual writing of the program much faster if we have a clear idea on paper of the major events within a program and their order of occurrence. We often overestimate our ability to write a

successful, bug-free program off the top of our heads and, as a result, we then spend many fruitless hours following a broken chain of logic through a non-running program.
Let's look at a simple flowchart for a delay loop — a quite useful item in the world of machine language as, without them, things tend to happen too fast for the eye to see! (see figure 3.1).

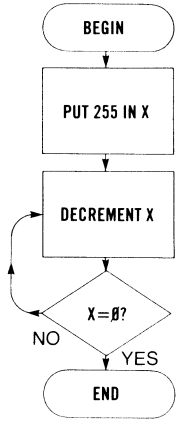


Figure 3.1 Flowchart example

Three fundamental types of box are used to build a flowchart (see figure 3.2):

- The terminal box, to begin and end the flowchart,
- The instruction or process box and its variant, the subroutine box, and
- The decision or branch box.

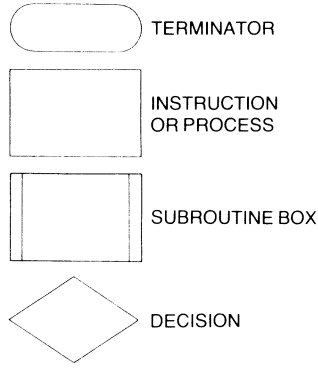


Figure 3.2 Basic flowchart symbols

The use of the terminal box is obvious. The instruction or process box is used to contain any command to the CPU. (The subroutine box indicates that a section of the program is described in another expanded flowchart.)

The most important of these boxes is the decision or branch box because it allows a program to branch back upon itself to repeat an instruction or sequence of instructions until a condition is met. The decisions which can be taken by a machine language program must be of the type that offers only two possible alternatives: either a condition is met or it is not met. If it is not met, then the program branches back to the beginning. If it is met, then the program continues with the next instruction.

As we can see from our delay loop example (figure 3.1), the decision box has two possible exits for the flow of the diagram to follow: either the number in memory has not yet reached zero (so the program flow branches back and decrements the number again) or the number *has* reached zero (so the program flow continues into the next box which, in this example, is the terminator).

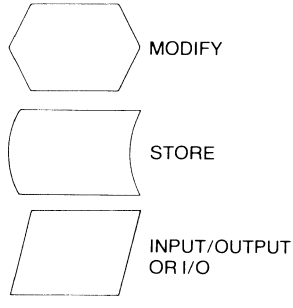


Figure 3.3 Additional flowchart symbols

There are other boxes from which to construct our flow diagrams, i.e. the *modify box*, the *store box* and the *input/output box* (see figure 3.3). Two additional boxes, the on-page and the off-page connectors, are used to organise a flowchart which has grown too large for one page (see figure 3.4). For most diagrams, the first three boxes are usually sufficient.

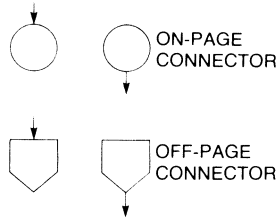


Figure 3.4 Flowchart connectors

Just to be sure that you can now do it for yourself, make up your own flowchart for some everyday process — getting a drink of water, say?

LOADING AND SAVING MACHINE LANGUAGE PROGRAMS

All of the routines in chapters 4 to 7 have a BASIC loader program included. Using a BASIC loader has the advantage that we can save, load and verify the machine language program in the normal manner. However, drawbacks of BASIC loaders are that they take up a lot of memory and must be run each time you want to use the machine language routines that they produce.

A better method of getting a machine language routine into memory is to load it off tape. You still have to use a loader at least once to get the routine into memory. Once there, the following five-line BASIC program can be used to save the routine — of any length — to tape from memory. The routine may be kept on tape or disc, and loaded whenever it is needed.

```
10  REM  --SAVER--
20  POKE 43,(SL):POKE 44,(SH)
30  POKE 45,(EL):POKE 46,(EH)
40  SAVE "ROUTINE"
50  POKE 43,1 : POKE 44,8 : POKE 45,3 : POKE 46,8
```

Let's take a closer look at this program to see exactly how it works.

```
20  POKE 43,(SL) : POKE 44,(SH)
```

To find SL and SH (Start Low and Start High) for a particular routine we must take the start address and subtract 2. To find the high and low bytes, apply the formulae:

```
SH = INT (ADDRESS) / 256
SL = ADDRESS – (256 * SH)
```

Don't forget to subtract 2 from the true start address of the routine before applying the formula. We have to do this as the operating system of the Commodore 64 replaces the first two bytes from the start address with the low and high bytes of the end address of the routine, and a SYS statement to the start of the program would almost certainly mean a crash.

The addresses 43 and 44 that we are pokeing these values into are the locations where the Commodore 64 stores the start address of the BASIC program currently in memory. The Commodore 64 refers to these addresses when it saves a program to tape or disc, so all we are doing is getting the Commodore 64 to save the section of memory which contains our machine language program instead of the area containing a BASIC program.

However, during the save operation, the Commodore 64 also refers to addresses 45 and 46, which usually contain the low and high bytes respectively of the end address of the BASIC program. So we must also alter these values to the low and high bytes of the end address of our program.

```
30  POKE 45,(EL) : POKE 46,(EH)
```

To find the low and high bytes of the end address of our routine, we can apply the same formula as before except that we don't have to subtract anything from the true end address, we just see the formula and put the values straight in.

```
40  SAVE "routine"
```

Now the usual message to PRESS RECORD AND PLAY will appear as the Commodore 64 saves the area of memory we have specified with our start and end addresses.

When the READY message appears, all that remains to do is to set the addresses of the start and end of the BASIC program to the correct values.

```
50  POKE 43,1 : POKE 44,8 : POKE 45,3 : POKE 46,8
```

This last line accomplishes this.

Now that we have saved our routine onto tape, the next thing to be able to do is to load it back again! This time, however, we cannot do it as part of a pre-existing program, as the LOAD command cannot be followed by any other program lines. The loading operation must be done directly from the keyboard.

The first thing we must do is alter the contents of locations 43 and 44 to the start address minus 2 as before. Then we can load our routine in the usual way:

```
LOAD "routine", 1
```

When the READY message appears, we must reset the values in locations 43 to 46 as before. Type in:

```
POKE 43,1 : POKE 44,8 : POKE 45,3 : POKE 46,8
```

All that we have to do now is type NEW and we are ready to use the routine whenever we need it.

USING MACHINE LANGUAGE PROGRAMS FROM BASIC

Once our programs are in the high area of memory of the Commodore 64 we need a method of using them as and when we need them.

BASIC gives us two distinct methods of accessing our machine language routines, either of which can be used in direct mode straight from the keyboard or from within a BASIC program. They are the SYS and USR commands.

The SYS Command

The syntax of this command is

```
SYS address
```

where *address* is in the range Ø to 65535.

This command causes the machine to place the current contents of the program counter (PC) register onto the stack. Then the *address* is placed in the PC register and execution continues from that address. So the value of *address* must be the first byte of our machine language program. If an RTS (**R**e**T**urn from **S**ubroutine) instruction is found by the CPU while executing the machine language program, the original value of the PC register is retrieved from the stack and execution continues from where it left off.

When using this method of accessing a routine, we must leave any necessary data at an address in memory where the routine can expect to find it using, say, the POKE statement. Similarly, if the routine has a result to be passed back to us, it must be left at an address that we know of in advance so that we can examine that result (by using PEEK, say) and continue on our way. The SYS method is known — not surprisingly — as the 'mailbox' method.

The USR Command

The syntax for this command is

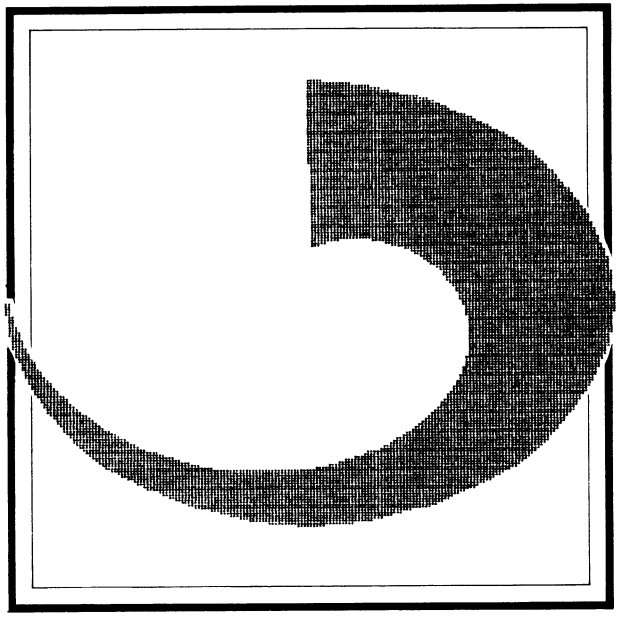
```
X = USR (B)
```

which translates into English as:
'Pass the value B into the machine language program at the address stored in the two locations 785 and 786 decimal, and let the result given

by that routine be called X'. Well, did you follow all that? Read it through again, it's not as complicated as it seems.

The advantage of USR is that the same routine can be used to work on various items of data. The disadvantage is that only one item of data can be passed to the routine at a time. Also, for each different routine used, the start address must be poked into those two locations 785 and 786 before the routine is accessed. If several routines are to be used, then the values in 785 and 786 must be altered before the next routine is called, which can be time consuming at a point when speed is very important — in the middle of an action game.

Let's look at a selection of useful machine language routines which are complete and ready to use, either directly or in your BASIC programs.



4

Sound Ideas

THE SOUND INTERFACE DEVICE

Built into the Commodore 64 is an extremely powerful sound synthesiser. This part of the computer is sometimes known affectionately as the SID chip, short for **S**ound **I**nterface **D**evice. This synthesiser is so powerful that, to fully explore its capabilities, we would need a whole book. However, it is possible to produce some very impressive sounds without actually digging too deeply into the workings of 'SID', to add a professional feel to your games programs.

The SID chip may be regarded as a block of 29 memory locations from 54272 to 54300.

Of these, the first 25 are directly concerned with sound production the first 21 addresses consist of three identical blocks for VOICE 1 VOICE 2 and VOICE 3.

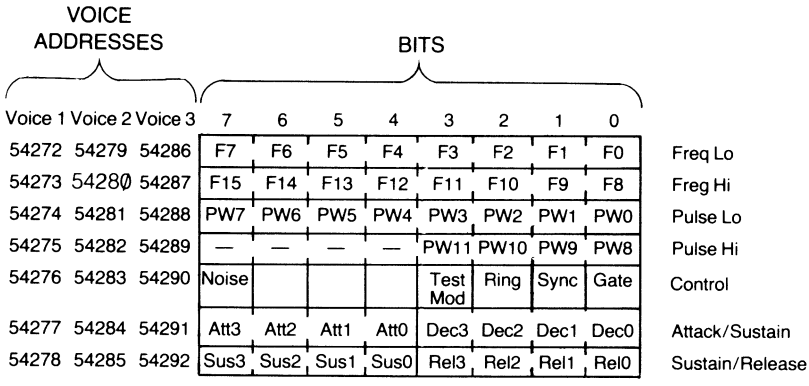


Figure 4.1 Sound interface device registers

VOICE 1 occupies the addresses 54272 to 54278, and each location has its own particular function (see figure 4.1). The block of addresses of VOICE 2 and VOICE 3 are identical to that of VOICE 1, with starting addresses of 54279 and 54286 respectively.

All of the locations of all three voices are *write only* memory, i.e. we can poke new values into them from BASIC or from within a machine language program but we may not peek into them. If you do, the result is garbage.

Here are the various parts of just one voice out of the three that the SID chip can command.

Frequency: Locations 54272 and 54273

These two addresses are used to contain the low and high bytes of a number in the range Ø to 65535 which, when multiplied by Ø.Ø596, gives us the frequency of the note in hertz (cycles per second). Thus a high number poked into 54273 will give a high note and a low number will give a low note.

- To alter the note being produced by a large amount we must alter the value in 54273 (FREQ HI).
- To alter the note by a small amount, we must alter the value in 54272 (FREQ LO).

Square Wave Mark/Space Ratio: Locations 54274 and 54275

The contents of these two addresses together make up a 12-bit number which corresponds to the width of the pulse when the square-wave

output is selected (bits 4 to 7 of 54275 are not used.) When we change the values at these addresses, we alter the ratio between the mark and the space of the overall square-wave (figure 4.2).

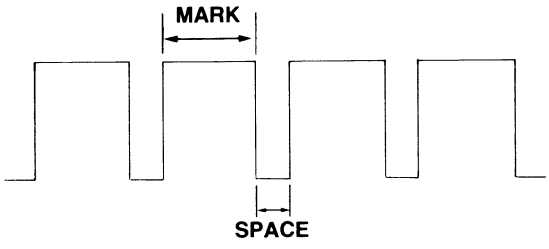


Figure 4.2 Square wave (1): terminology

Timbre: Location 54276

All of the bits of this register have their own particular use, so we must look at them individually.

- **Bit Ø GATE** To understand the function of this bit, we need to know a little about sound envelopes. A sound envelope is made up of four parts (see figure 4.3):
  attack — the start and build-up of the sound,
  decay — the fall-away from maximum volume,
  sustain — a plateau of 'sustained' volume and
  release — the fall-off to zero volume.

  A sound envelope is sometimes known as an ADSR envelope (**A**ttack, **D**ecay, **S**ustain, **R**elease). See figure 4.3.

  By setting bit Ø to 1, we are starting the attack, decay, sustain part of the envelope and, when we reset bit Ø to contain Ø, the release part of the envelope occurs.

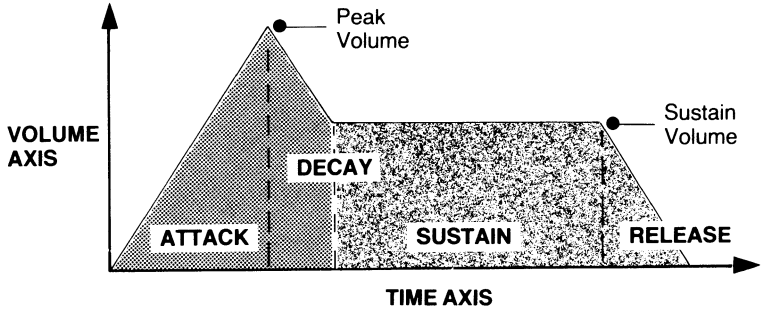


Figure 4.3 The attack/decay/sustain/release envelope

- **Bit 1 SYNC** By setting this bit to 1 we can synchronise the output of VOICE 1 with that of VOICE 3.
- **Bit 2 RING MOD** By setting bit 2 to 1, it is possible to partly combine VOICE 1 with VOICE 3 when the triangle waveform is selected.
- **Bit 3 TEST** The use of this bit is generally for test purposes or to cause the output of VOICE 1 to be synchronised with an external event.
- **Bit 4 TRIANGLE WAVEFORM** When bit 4 is set to 1, the triangle waveform is selected for VOICE 1. This enables us to imitate the sound of instruments from a flute to xylophone (see figure 4.4).

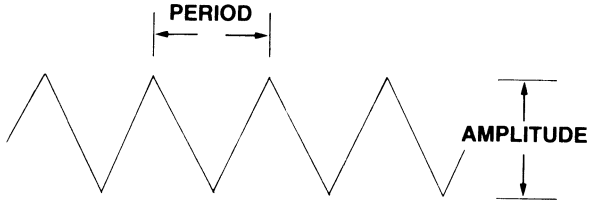


Figure 4.4 Triangle wave form

- **Bit 5 SAWTOOTH WAVEFORM** By setting bit 5 to 1 we are able to create sounds rather like those from brass instruments, using a sawtooth waveform (see figure 4.5).

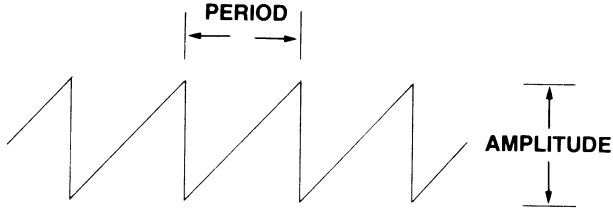


Figure 4.5 Sawtooth wave form

- **Bit 6 SQUARE WAVEFORM** When bit 6 is set, the waveform produced is a square wave, which allows us to produce the sound of, among others, a piano (see figure 4.6).

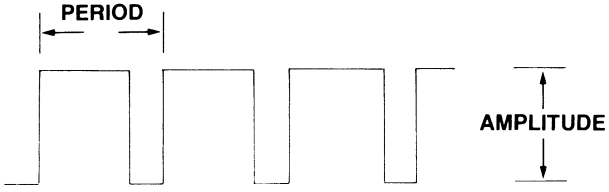


Figure 4.6 Square wave form (2)

- **Bit 7 WHITE NOISE** When bit 7 is set, the waveform produced is that of white noise, the sound equivalent of white light, i.e. a mixture of all wavelengths. This is useful for producing a variety of non-musical sound effects.

Attack and Decay: Location 54277

The high nybble of this address is used to contain values from Ø to 15 for the attack part of the envelope, the low nybble contains values for the decay part of the envelope.

Sustain and Release: Location 54278

The high nybble of this location is used to contain the value for the sustain part of the envelope, and the low nybble the value for the release part. Both values must be in the range Ø to 15.

Ouput Filters: Locations 54293 to 54295

These three locations are directly related to the process of filtering the output from the SID chip and are beyond the scope of this book.

Volume: Location 54256

Bits 4 to 7 of this address are also concerned with filtering but the low nybble, bits Ø to 3, are the place where we must put our volume setting. The volume setting has a possible range of Ø to 15. A volume other than Ø must be selected to allow any sound to be heard.

USING THE SID

For more detailed information on the capabilities of the SID chip, refer to the *Commodore 64 Programmer's Reference Guide* (see Appendix G, 'Further Reading').

SOUND ROUTINES

Here follow some routines. Each has a BASIC loader included, along

with the flowchart and assembly listing. Notes on the use of each routine are given, along with any mailbox addresses used.

I have located all of the routines in a 4 Kbyte section of memory which lies between BASIC and the VIC II chip. However, if you wish to locate them somewhere else, then the addresses in the BASIC loader must be altered, as will the start address to SYS the routine when it is in position. (The block transfer routine in chapter 7 may be employed to shift the contents of an area of memory containing a machine language programme — or anything else. *Use with care*!)

GUNSHOT

START	49155
END	49185
LENGTH	31 BYTES
REGISTERS USED	A, X

As the flowchart for this routine shows, it is extremely simple in its working, since it does not contain even one loop! To produce the sound, certain values are put into the registers of the SID chip for volume, pitch and so on, and the sound is started. Other values are placed in the SID chip and the sound is ended. What could be more simple than that?

The sound produced is a short burst of white noise, with a short attack time followed by a rather longer decay. Almost as soon as the sound starts it is ended by resetting bit 0 in location 54276, the control register for voice 1. The volume is set to maximum (15) and the pitch to 30720 by placing 120 in location 54273, the FREQ HI register for voice 1.

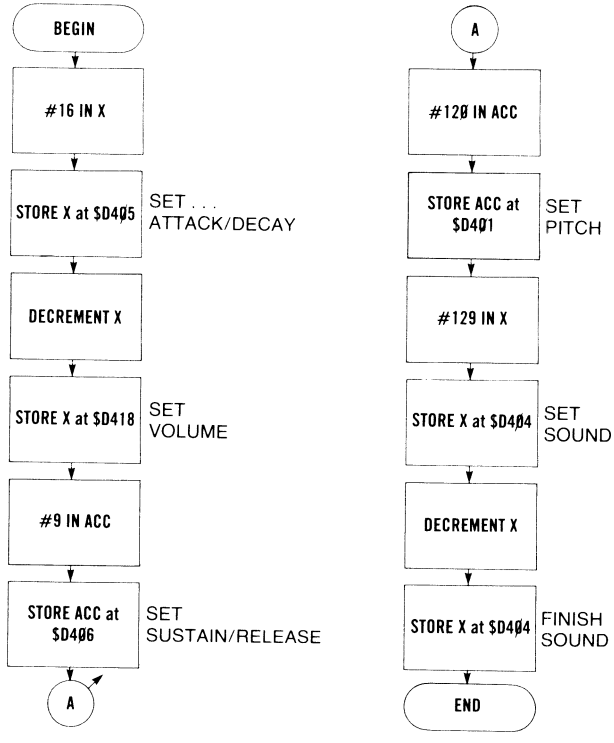
No preparation for this routine is needed, just SYS the start address and duck the flying lead!

Gunshot Assembly

```
 0  C003  A210      LDX  #$10
 1  C005  8E05D4    STX  $D405
 2  C008  CA        DEX
 3  C009  8E18D4    STX  $D418
 4  C00C  A909      LDA  #$09
 5  C00E  8D06D4    STA  $D406
 6  C011  A978      LDA  #$78
 7  C013  8D01D4    STA  $D401
 8  C016  A281      LDX  #$81
 9  C018  8E04D4    STX  $D404
10  C01B  CA        DEX
11  C01C  8E04D4    STX  $D404
12  C01F  60        RTS
```

Gunshot Loader

```
90 REM-----GUNSHOT LOADER-----
100 FOR X=49155 TO 49183
110 READ A:POKE X,A
120 NEXT
125 REM-----GUNSHOT DATA-----
130 DATA 162,16,142,5,212,202,142,24,212
140 DATA 169,9,141,6,212,169,120,141
150 DATA 1,212,162,129,142,4,212,202,142
160 DATA 4,212,96
```



4.7(a) **4.7(b)**

Figure 4.7 'Gunshot' flowchart

ECHO . . . ECHO . . . ECHO . . .

START	49190
END	49228
LENGTH	39 BYTES
REGISTERS USED	A, X, Y

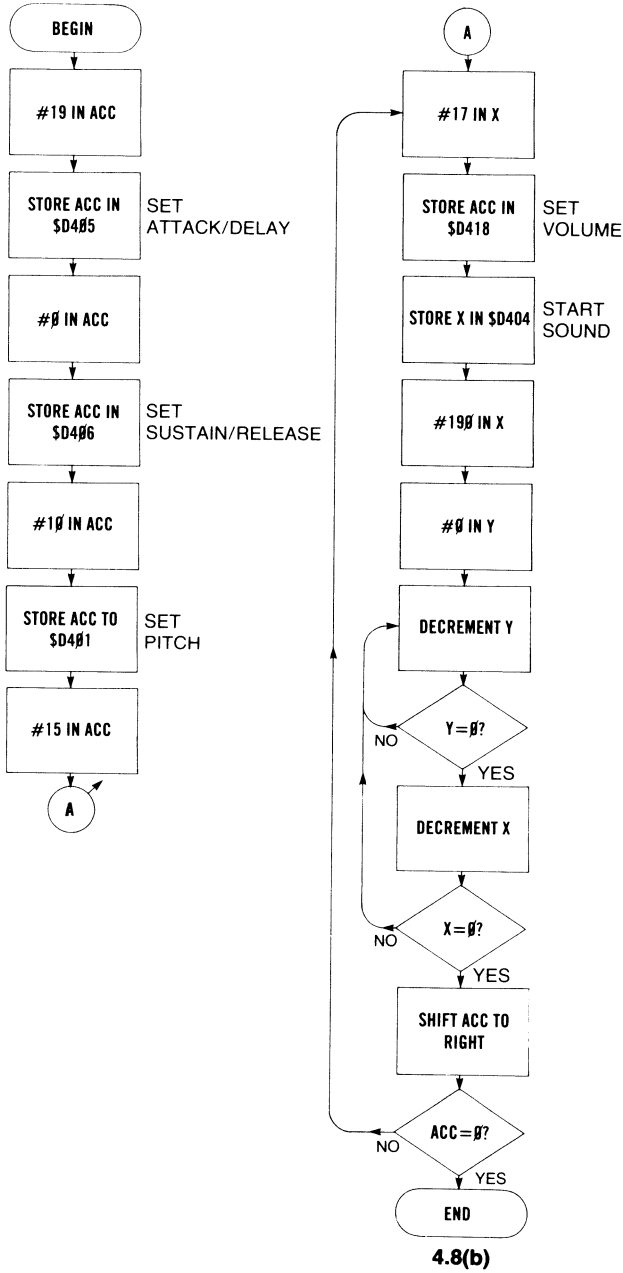
A look at the flow diagram for this routine shows us the action of the program may be divided roughly into two parts:

- First, the sound chip is set up with pitch, attack, decay, sustain and release values.
- Second, the program takes the value for maximum volume (15), and places it into the volume register before starting the ADSR envelope.

A short delay is then set up, and the volume halved.

Provided that the volume has not reached zero, the process is repeated and, as the contents of the volume register are replaced with the right-shifted contents of the accumulator, the successive volume settings for the sound produced are 15, 7, 3 and 1. The volume then reaches zero and the routine ends.

No preparation is needed, just SYS the start address and away you go. However, if the echo sound is to be synchronised with some on-screen action, remember to make the event occur on the screen first, as the echo sound takes a second or so to run. Otherwise, the sound will echo away and *then* the action that made the sound will be seen — not all that realistic!



4.8(b)

Figure 4.8 'Echo . . Echo . . Echo . . ' flowchart

Echo Assembly

```
 0  C026  A913      LDA  #$13
 1  C028  8D05D4    STA  $D405
 2  C02B  A900      LDA  #$00
 3  C02D  8D06D4    STA  $D406
 4  C030  A90A      LDA  #$0A
 5  C032  8D01D4    STA  $D401
 6  C035  A90F      LDA  #$0F
 7  C037  A211      LDX  #$11
 8  C039  8D18D4    STA  $D418
 9  C03C  8E04D4    STX  $D404
10  C03F  A2BE      LDX  #$BE
11  C041  A000      LDY  #$00
12  C043  88        DEY
13  C044  D0FD      BNE  $C043
14  C046  CA        DEX
15  C047  D0FA      BNE  $C043
16  C049  4A        LSR  A
17  C04A  D0EB      BNE  $C037
18  C04C  60        RTS
```

Echo Loader

```
100 REM -----ECHO LOADER-----
110 FOR X=49190 TO 49228
120 READ A:POKE X,A
130 NEXT
140 REM -----ECHO DATA-----
150 DATA 169,19,141,5,212,169,0,141,6
160 DATA 212,169,10,141,1,212,169,15,162
170 DATA 17,141,24,212,142,4,212,162,190
180 DATA 160,0,136,208,253,202,208,250
190 DATA 74,208,235,96
```

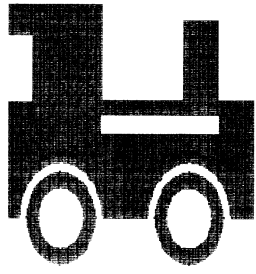
LOCOMOTIVE

START	49230
END	49270
LENGTH	41 BYTES
REGISTERS USED	A, X, Y

The days of the old steam locomotives can be effectively brought back to life with a SYS to this routine.

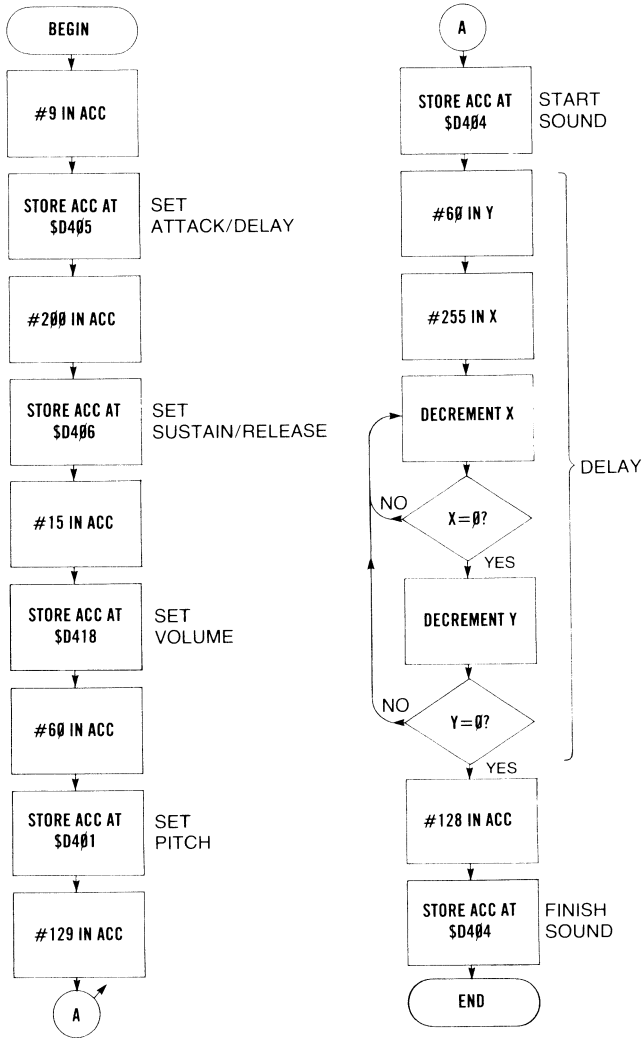
Each call to the routine produces one *wooossh* of steam leaving the locomotive's cylinders. To reproduce the sound of a fast-moving train, a quite short interval between calls is sufficient. For the sound of a slow-moving train, a longer interval between calls is required.

The program uses the white noise waveform of VOICE 1 at its maximum volume, with the pitch set at 15360 by putting 60 into the FREQ HI register (54273) of VOICE 1.



Locomotive Assembly

```
 0  C04E  A909      LDA  #$09
 1  C050  8D05D4    STA  $D405
 2  C053  A9C8      LDA  #$C8
 3  C055  8D06D4    STA  $D406
 4  C058  A90F      LDA  #$0F
 5  C05A  8D18D4    STA  $D418
 6  C05D  A93C      LDA  #$3C
 7  C05F  8D01D4    STA  $D401
 8  C062  A981      LDA  #$81
 9  C064  8D04D4    STA  $D404
10  C067  A03C      LDY  #$3C
11  C069  A2FF      LDX  #$FF
12  C06B  CA        DEX
13  C06C  D0FD      BNE  $C06B
14  C06E  88        DEY
15  C06F  D0FA      BNE  $C06B
16  C071  A980      LDA  #$80
17  C073  8D04D4    STA  $D404
18  C076  60        RTS
```



4.9(a) **4.9(b)**

Figure 4.9 'Locomotive' flowchart

Locomotive Loader

```
100 REM -----STEAM TRAIN LOADER-----
110 FOR X=49230 TO 49270
120 READ A:POKE X,A
130 NEXT
140 REM -----STEAM TRAIN DATA-------
150 DATA 169,9,141,5,212,169,200,141,6,212,169
160 DATA 15,141,24,212,169,60,141,1,212,169,129
170 DATA 141,4,212,160,60,162,255,202,208,253,136
180 DATA 208,250,169,128,141,4,212,96
```

Train Departure Demonstration

```
5 REM -----STEAM TRAIN DEPARTURE-----
10 I=1000
20 SYS 49230
30 FOR T=0 TO I:NEXT
40 IF I>75 THEN I=I-25
50 GOTO 20
```

When the loader program has been run to put the routine into memory, check that it works by typing SYS 49230. If all is well, one *wooossh* should be produced; if not, check over your loader for typing errors. When the routine is operational, NEW the loader and type in the train departure program to get some idea of the effects that can be achieved by altering the length of the delay loop between calls to the routine.

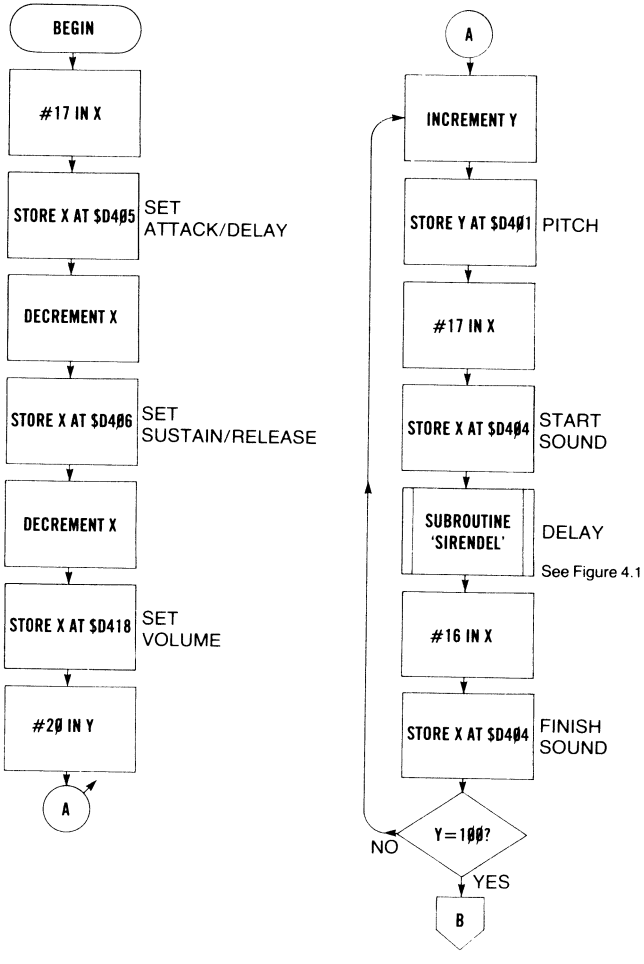
POLICE SIREN

START	49275
END	49352
LENGTH	78 BYTES
REGISTERS USED	A, X, Y

The old two-tone police siren of two clear-cut tones, has been superseded by a far more ear-wrenching sound, which rises slowly to a peak and slides back down again.

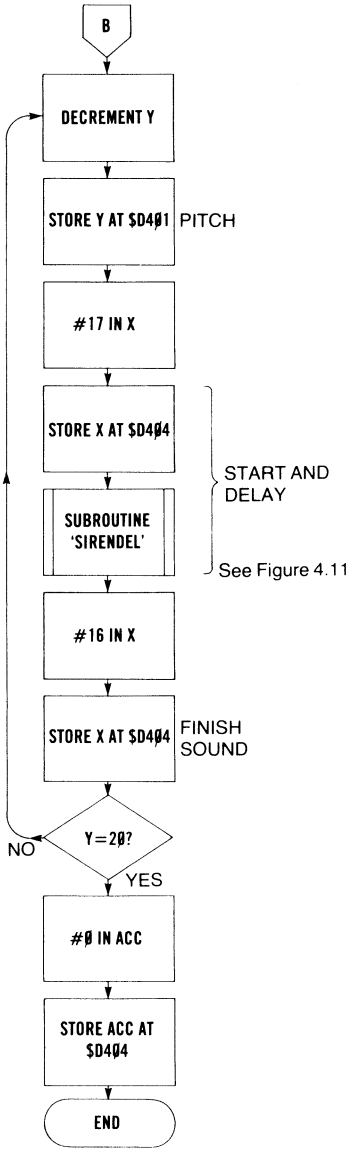
It is this sliding sound which grabs the attention so forcibly. The Commodore 64 can produce this type of sound by making a very short sound of a particular frequency, then another of a little higher frequency and so on up the scale. When the slide reaches a predetermined point the routine returns.

This effect happens just once when we SYS this routine, so we can effect screen action in between cycles of the siren itself. This routine needs no setting up; just run the BASIC loader and switch on your blue flashing light!



4.10(a) **4.10(b)**

Figure 4.10 'Police Siren' flowchart



4.10(c)

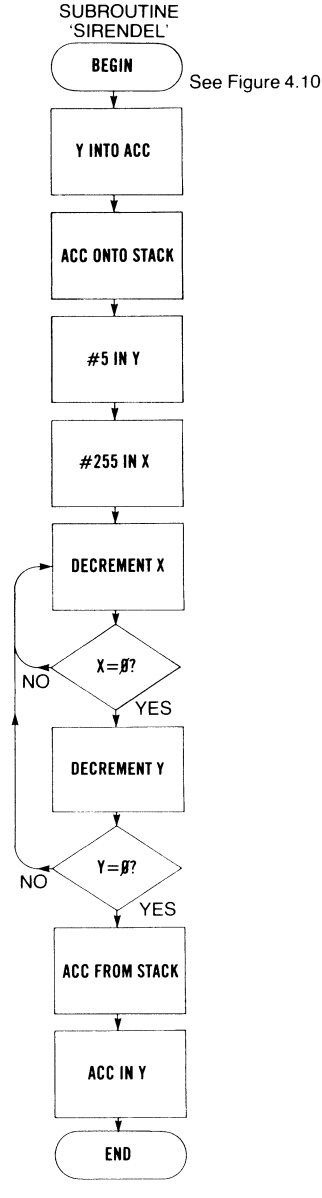


Figure 4.11
'Police Siren' delay subroutine flowchart

Siren Assembly

```
 0  C07B  A211      LDX  #$11
 1  C07D  8E05D4    STX  $D405
 2  C080  CA        DEX
 3  C081  8E06D4    STX  $D406
 4  C084  CA        DEX
 5  C085  8E18D4    STX  $D418
 6  C088  A014      LDY  #$14
 7  C08A  C8        INY
 8  C08B  8C01D4    STY  $D401
 9  C08E  A211      LDX  #$11
10  C090  8E04D4    STX  $D404
11  C093  20BAC0    JSR  $C0BA
12  C096  A210      LDX  #$10
13  C098  8E04D4    STX  $D404
14  C09B  C064      CPY  #$64
15  C09D  D0EB      BNE  $C08A
16  C09F  88        DEY
17  C0A0  8C01D4    STY  $D401
18  C0A3  A211      LDX  #$11
19  C0A5  8E04D4    STX  $D404
20  C0A8  20BAC0    JSR  $C0BA
21  C0AB  A210      LDX  #$10
22  C0AD  8E04D4    STX  $D404
23  C0B0  C014      CPY  #$14
24  C0B2  D0EB      BNE  $C09F
25  C0B4  A900      LDA  #$00
26  C0B6  8D04D4    STA  $D404
27  C0B9  60        RTS
28  C0BA  98        TYA
29  C0BB  48        PHA
30  C0BC  A005      LDY  #$05
31  C0BE  A2FF      LDX  #$FF
32  C0C0  CA        DEX
33  C0C1  D0FD      BNE  $C0C0
34  C0C3  88        DEY
35  C0C4  D0FA      BNE  $C0C0
36  C0C6  68        PLA
37  C0C7  A8        TAY
38  C0C8  60        RTS
```

Siren Loader

```
100 REM -----SIREN LOADER-----
110 FOR X=49275 TO 49352
120 READ A:POKE X,A
130 NEXT
140 REM -----SIREN DATA-----
150 DATA 162,17,142,5,212,202,142,6,212,202
160 DATA142,24,212,160,20,200,140,1,212,162
170 DATA 17,142,4,212,32,186,192,162,16,142
180 DATA 4,212,192,100,208,235,136,140,1,212
190 DATA 162,17,142,4,212,32,186,192,162,16
200 DATA 142,4,212,192,20,208,235,169,0,141
210 DATA 4,212,96,152,72,160,5,162,255,202
220 DATA 208,253,136,208,250,104,168,96
```

BOMB FALL

START	49355
END	49412
LENGTH	58 BYTES
REGISTERS USED	A, X, Y

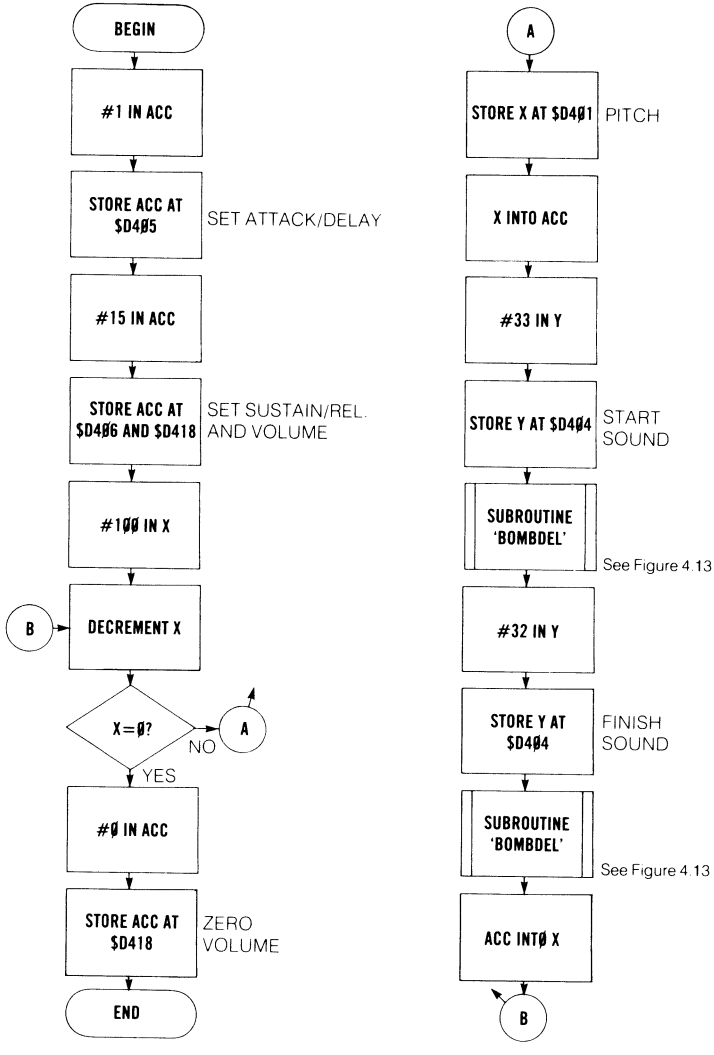
The 'shoot-em-up' arcade games have by their very nature involved the use of a dazzling variety of guns, bombs and other assorted armaments. To give these death-bringers a touch of realism, added sound effects are a must. This routine and the next one, 'Explosion', add realism to such events as a bomb dropping on an unsuspecting alien and subsequently exploding.

This routine is rather like the 'Siren' routine in that a sound of a certain frequency is produced, followed by another slightly lower in pitch and so on down the scale. The only differences are the time taken to descend through the pitch range, and that the SIREN routine uses the triangle waveform of VOICE 1 while the 'Bomb Fall' routine uses the harsher sound of the sawtooth waveform.

No preparation is necessary for 'Bomb Fall'; just SYS 49355 and then SYS 49475 for 'Explosion' to generate the resultant explosion noises!

Bomb Fall Assembly

```
 0  C0CB  A901      LDA  #$01
 1  C0CD  8D05D4    STA  $D405
 2  C0D0  A90F      LDA  #$0F
 3  C0D2  8D18D4    STA  $D418
 4  C0D5  8D06D4    STA  $D406
 5  C0D8  A264      LDX  #$64
 6  C0DA  CA        DEX
 7  C0DB  F022      BEQ  $C0FF
 8  C0DD  8E01D4    STX  $D401
 9  C0E0  8A        TXA
10  C0E1  A021      LDY  #$21
11  C0E3  8C04D4    STY  $D404
12  C0E6  20F4C0    JSR  $C0F4
13  C0E9  A020      LDY  #$20
14  C0EB  8C04D4    STY  $D404
15  C0EE  20F4C0    JSR  $C0F4
16  C0F1  AA        TAX
17  C0F2  D0E6      BNE  $C0DA
18  C0F4  A004      LDY  #$04
19  C0F6  A2FF      LDX  #$FF
20  C0F8  CA        DEX
21  C0F9  D0FD      BNE  $C0F8
22  C0FB  88        DEY
23  C0FC  D0FA      BNE  $C0F8
24  C0FE  60        RTS
25  C0FF  A900      LDA  #$00
26  C101  8D18D4    STA  $D418
27  C104  60        RTS
```



4.12(a) **4.12(b)**

Figure 4.12 'Bomb Fall' flowchart

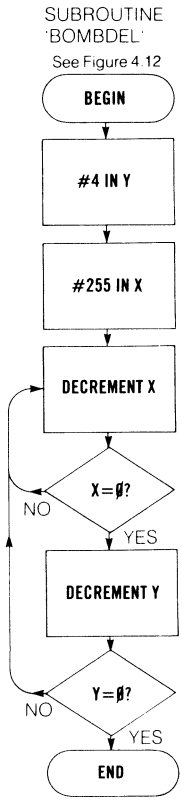


Figure 4.13 'Bomb Fall' delay subroutine flowchart

Bomb Fall Loader

```
100 REM -----BOMB-FALL LOADER---
110 FOR X=49355 TO 49412
120 READ A:POKE X,A
130 NEXT
140 REM -----BOMB-FALL DATA-----
150 DATA 169,1,141,5,212,169,15,141,24,212,141,6,212,162
160 DATA 100,202,240,34,142,1,212,138,160,33,140,4,212
170 DATA 32,244,192,160,32,140,4,212,32,244,192,170,208
180 DATA 230,160,4,162,255,202,208,253,136,208,250,96,169
190 DATA 0,141,24,212,96
```

EXPLOSION

START	49475
END	49523
LENGTH	49 BYTES
REGISTERS USED	A, X, Y

As the bomb falls to the ground the previous routine, 'Bomb Fall', will give rather an effective falling sound and, when it is followed directly by this routine, the sound is finished off by a full volume explosion!

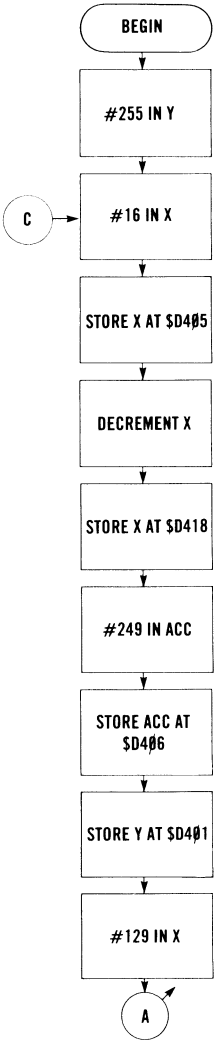
To operate, just SYS 49475 and hear the bomb's devastation occur!

Explosion Assembly

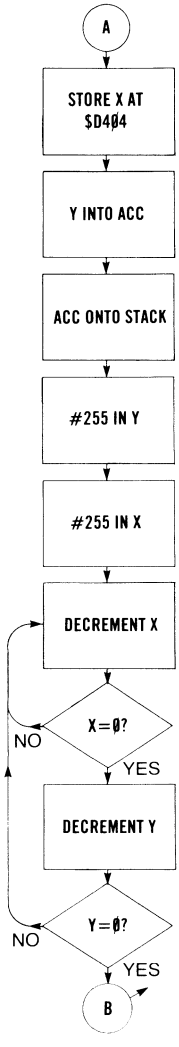
```
 0  C143  A0FF      LDY  #$FF
 1  C145  A210      LDX  #$10
 2  C147  8E05D4    STX  $D405
 3  C14A  CA        DEX
 4  C14B  8E18D4    STX  $D418
 5  C14E  A9F9      LDA  #$F9
 6  C150  8D06D4    STA  $D406
 7  C153  8C01D4    STY  $D401
 8  C156  A281      LDX  #$81
 9  C158  8E04D4    STX  $D404
10  C15B  98        TYA
11  C15C  48        PHA
12  C15D  A0FF      LDY  #$FF
13  C15F  A2FF      LDX  #$FF
14  C161  CA        DEX
15  C162  D0FD      BNE  $C161
16  C164  88        DEY
17  C165  D0FA      BNE  $C161
18  C167  68        PLA
19  C168  A8        TAY
20  C169  A280      LDX  #$80
21  C16B  8E04D4    STX  $D404
22  C16E  88        DEY
23  C16F  88        DEY
24  C170  88        DEY
25  C171  10D2      BPL  $C145
26  C173  60        RTS
```

Explosion Loader

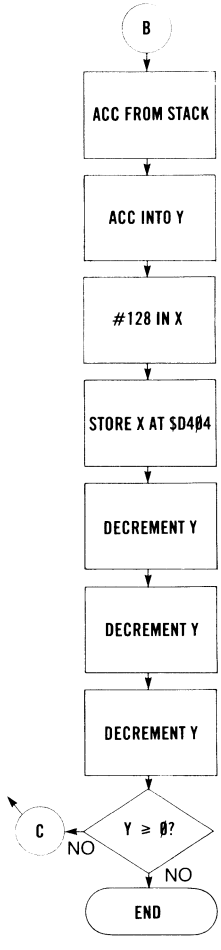
```
10 REM --------EXPLOSION LOADER--------
20 FOR X=49475 TO 49523
30 READ A:POKE X,A
40 NEXT
50 REM ---------EXPLOSION DATA---------
60 DATA 160,255,162,16,142,5,212,202,142
70 DATA 24,212,169,249,141,6,212,140,1
80 DATA 212,162,129,142,4,212,152,72,160
90 DATA 255,162,255,202,208,253,136,208
100 DATA 250,104,168,162,128,142,4,212
110 DATA 136,136,136,16,210,96
```



4.14(a)



4.14(b)



4.14(c)

Figure 4.14 'Explosion' flowchart

PIANO KEYS

START	49430
END	49470
LENGTH	41 BYTES
REGISTERS USED	A, X, Y

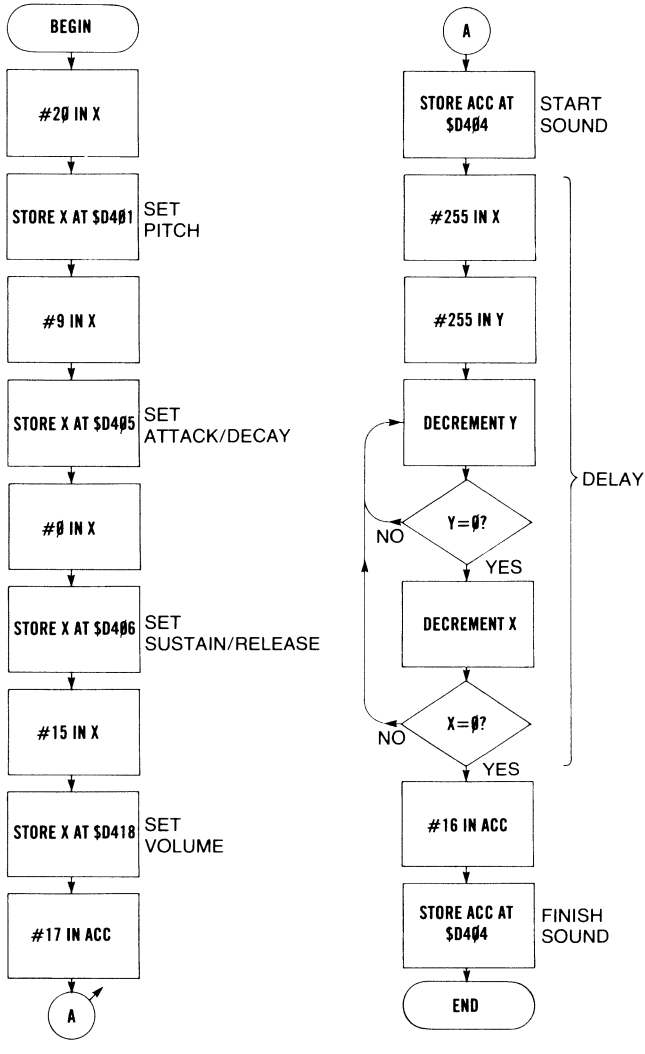
An electric piano is reproduced here rather than an acoustic piano, by using the triangle waveform on VOICE 1. A minimum attack rate (0) is combined with a medium decay rate (9). A delay is introduced before the sustain/release part of the envelope is triggered, lasting about 0.25 s. The sustain and release are both 0 to give a quick cut-off, like the damping action of releasing the piano key.

The note produced is fractionally below an F# note, given by storing 20 in FREQ HI of VOICE 1 at location 54273. By altering this value at byte 49431 in the routine we can make a sound with a different pitch. Any value between 0 and 255 may be put at this address, giving us a range of seven octaves.

But bear in mind that, because of the way each note in an octave doubles in the next octave, the scale may *seem* to contain too many notes at the high end of the scale, with large gaps at the low end. When you have loaded the routine into memory, try this short jingle — a suitable sound effect to use when an alien bites the dust!

Piano Assembly

```
 0  C116  A214      LDX  #$14
 1  C118  8E01D4    STX  $D401
 2  C11B  A209      LDX  #$09
 3  C11D  8E05D4    STX  $D405
 4  C120  A200      LDX  #$00
 5  C122  8E06D4    STX  $D406
 6  C125  A20F      LDX  #$0F
 7  C127  8E18D4    STX  $D418
 8  C12A  A911      LDA  #$11
 9  C12C  8D04D4    STA  $D404
10  C12F  A2FF      LDX  #$FF
11  C131  A0FF      LDY  #$FF
12  C133  88        DEY
13  C134  D0FD      BNE  $C133
14  C136  CA        DEX
15  C137  D0FA      BNE  $C133
16  C139  A910      LDA  #$10
17  C13B  8D04D4    STA  $D404
18  C13E  60        RTS
```



4.15(a) **4.15(b)**

Figure 4.15 'Piano Keys' flowchart

Piano Demonstration

```
100 REM -----SAMPLE JINGLE USING PIANO KEYS-----
110 FOR N=1 TO 11
120 READ A:READ B
130 POKE 49431,A
140 SYS 49430
150 FOR T=1 TO B:NEXT T
160 NEXT N
170 END
180 DATA 25,125,25,125,25,0,25,200,30,75
190 DATA 28,0,28,75,25,0,25,75,24,0,25,0
```

Piano Loader

```
10 REM -----PIANO LOADER-----
20 FOR X=49430 TO 49470
30 READ A:POKE X,A
40 NEXT
50 REM -----PIANO  DATA------
60 DATA 162,20,142,1,212,162,9,142,5,212,162,0,142,6,212,162
70 DATA 15,142,24,212,169,17,141,4,212,162,255,160,255,136
80 DATA 208,253,202,208,250,169,16,141,4,212,96
```

NOISES OFF, OFF NOISES

START	49415
END	49425
LENGTH	11 BYTES
REGISTERS USED	A, Y

When any of the sound routines has been used, particularly during the development and debugging stage of a program's life, we may interrupt the process of making a sound with the Commodore 64 and, in doing so, cause a hum, buzz or other unwanted noise to be produced continuously.

One way of cutting this sound off is to press the <RUN/STOP> and <RESTORE> keys simultaneously. This method will unfortunately also switch off any sprites currently in use, close any open output channels (i.e. to a printer) and clear any variables. To avoid this contretemps and the resultant waste of time restoring the sprites and so on, this routine will cut off all unwanted sounds from the SID. Sound cut-off is accomplished by placed a Ø in all of the registers of the sound chip, to effectively switch off all three voices.

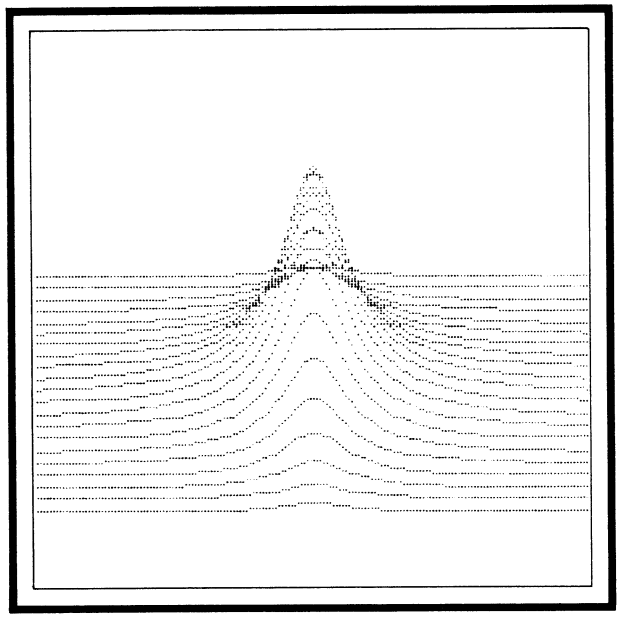
To use the routine, just SYS 49415.

Noises Off Assembly

```
Ø  C1Ø7  A919      LDY  #$19
1  C1Ø9  A9ØØ      LDA  #$ØØ
2  C1ØB  99FFD3    STA  $D3FF,Y
3  C1ØE  88        DEY
4  C1ØF  DØFA      BNE  $C1ØB
5  C111  6Ø        RTS
```

Noises Off Loader

```
10 REM -----NOISES OFF LOADER-----
20 FOR X=49415 TO 49425
30 READ A:POKE X,A
40 NEXT
50 REM ------NOISES OFF DATA------
60 DATA 160,25,169,0,153,255,211,136,208,250,96
```



5 Bit Mapping

HIGH-RESOLUTION FACILITIES OF THE VIDEO CHIP

The Commodore 64 has an extremely powerful video chip known as the VIC II which is in sole command of all graphic displays. The VIC II is not only powerful but versatile too, and allows the programmer to produce high-resolution graphics and mobile *sprites;* sprites are described in chapter 6.

For the moment, we will examine the ability of the VIC II to produce pictures which are fixed on the screen (unless of course the programmer writes a routine to move them around!). Like the SID chip examined in the previous chapter, the VIC II can be seen as a collection of registers, each of which has its own specialised use and all of which are available to the programmer to produce the impressive graphic displays for which the Commodore 64 is known.

The addresses which access VIC II begin at 53248 and end at 53294. The ones which concern us in this chapter are these:

- Location 53265, high-resolution mode usage,
- Location 53270, multicolour mode usage,
- Location 53272, screen memory positioning and character set selection,
- Locations 55296 to 56295, foreground colour selection, and
- Locations 53281 to 53284, background colour selection.

Location 53265

This address in the area of memory assigned to the VIC II chip can be seen as eight separate bits, each of which has its own particular use. (Note that bits 7, 2 and 0 are unused.)

- **Bit 6 EXTENDED COLOUR** When set to 1, this bit engages extended colour mode. This means that not only can the foreground colour of any character be set, but the background colour too, independently of any other screen location. For example, you could display a green Y on a yellow background.

  The foreground and background colours can be set to any of 16 colours by placing a number corresponding to a colour from the usual range into one of the colour memory locations: locations 55296 to 56295 for foreground colours, and locations 53281 to 53284 for background colour.

- **Bit 5 BIT-MAP MODE** When set to 1, this bit causes the screen to take on the standard bit-map configuration of 320 by 200 dots, each of which may be individually turned on or off. To disengage bit-map mode, bit 5 is re-set to 0.

- **Bit 4 DISPLAY ENABLE** As we normally wish to see the screen display, this bit will be set to 1 but, if we do not need the display, by re-setting this bit to 0, we blank out the display to the current border colour. The information on the screen is still there, it is just not visible.

  This action also has the advantage of speeding up the CPU slightly, so screen blanking could be useful during a protracted session of number crunching or a lengthy sort routine.

- **Bits 3 and 1 SCROLLING** Screen scrolling is made possible in either a horizontal or vertical direction by the VIC II chip under the control of a machine language routine. For more information on this aspect of the VIC II, refer to the *Commodore 64 Programmer's Reference Guide*.

Location 53270

Bits 7, 6, 2 and 0 are unused.

- **Bits 4 and 5 MULTICOLOUR BIT MAP** These bits are used in conjunction with bit 5 at location 53265 mentioned above to engage multicolour bit-map mode. This mode is similar to the ordinary bit-map mode, except the resolution is halved and the range of colours possible in any 8 × 8 square is now doubled. The screen is only 160 pixels wide by 200 high but up to four colours may be used in any character space.

  Multicolour bit-map mode is selected by setting bit 5 of location 53265 and bit 4 of location 53270 to 1. This process is reversed by resetting these same two bits to 0.

- **Bits 3 and 1 SCROLLING** These two bits, like the corresponding bits of location 53265, are used in the process of scrolling the screen display.

Location 53272

This address is best regarded as two nybbles, which have two distinct uses. The low nybble is concerned with the position in memory of the bit-map screen — if bit-map mode is selected. If not, the low nybble is concerned with which of the Commodore 64's character sets is currently in use.

The high nybble is a pointer to the position in memory where the screen display starts. The value in the high nybble (0 to 15) is multiplied by 1024 to give the address of the first screen location.

Locations 53281 to 53284

These registers store the background colour information for various modes. (See location 53265 preceding.)

Locations 55296 to 56319

These addresses represent the beginning and end of colour memory, which is used to store the foreground colours of characters printed to the screen. (See location 53265 preceding.)

THE HIGH RESOLUTION SCREEN

Like most home micro computers the Commodore 64 uses *bit-mapping* to produce high-resolution graphic displays. The impressive graphics possible by this method are demonstrated by some of the graphic adventure and arcade games on the market.

However, as the BASIC of the Commodore 64 has no graphics

commands at all, even to clear the bit map screen from BASIC takes over 28 seconds! Waiting that length of time during an exciting game of 'save-the-universe' could lead to terminal boredom! Machine language is a must for dealing rapidly with the large amounts of data and memory involved when manipulating the graphic bit map.

Perhaps the most effective way of using the graphic capabilities of the Commodore 64 is to use bit mapping to construct background scenery but animate the action with sprites. For the moment we shall concentrate on bit mapping and leave graphic sprites to the next chapter.

The bit map screen can be thought of as a grid 32Ø points across by 2ØØ points down (see figure 5.1).

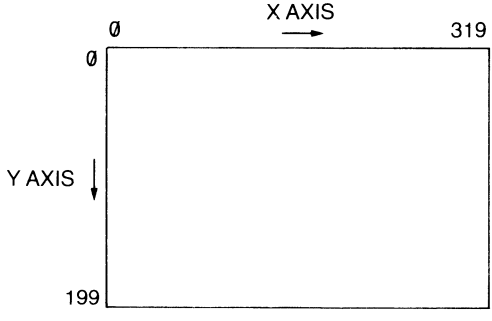


Figure 5.1 Bit-map screen co-ordinates

The screen can be accessed as 8ØØØ consecutive bytes of memory. Ø is in the top left corner of the screen and covers the first eight points on the grid with its eight bits. Byte 1 is directly below the first and so on until byte 7 (see figure 5.2). Byte 8 is to the right of byte Ø, below that comes byte 9 and on down to byte 15, when we are back to the top again with byte 16.

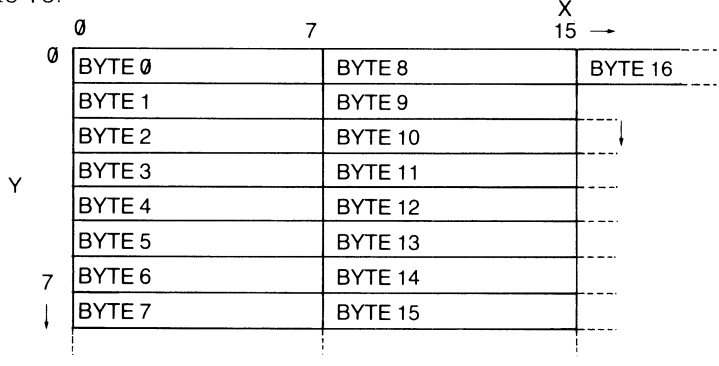


Figure 5.2 Mapping of screen memory to screen

This screen layout may seem peculiar but it makes things much simpler for the machine to display the text characters, since one character is built directly from eight sucessive bytes of data.

The routines in this chapter set up the bit map screen to reside at addresses 8192 to 16191. The 8000 bytes from 24000 to 32000 have been allocated as storage for a second bit map screen which may be recalled into the visible bit map area at any time, used to contain an overlay of new data to place on top of the original graphic or, if your program takes up a large amount of memory, used as normal memory.

Even if a ROM cartridge is fitted to your Commodore 64 — an extension to BASIC or an assembler — this spare screen area will not interfere with the memory set aside for external ROM in any way.

CHANGING COLOURS ON THE BIT-MAP SCREEN

When we want to change the colour of any part of the bit-map screen, we must bear in mind that there is a direct relationship between the ordinary text screen and the bit-map screen. This is proven by the multicoloured ghosts of text which shine through into the bit-map screen from the text area.

However, when we look at these ghosts from the bit-map side, we cannot tell which characters they are because they all just appear as a block of colour. That fact can be quite useful since, if we knew which character gave us which colour, then we would be able to change the background and foreground colours of any 8 x 8 block on the bit-map screen by simply placing a suitable character behind that block in the corresponding position on the text screen.

Before we go any further, let's decide what we mean by the terms *foreground* and *background*. For any particular character-sized block, the foreground colour is that colour which appears in any pixel whose corresponding bit in memory is a 1 and the background colour is the colour of any pixel whose bit is Ø.

If we look at just one character position on the text screen, at address 1524 say, i.e. right in the middle of the screen, this is how we determine the possible colours for that block.

The rules governing bit-map colour at this and every other position are these:

> The number in the high nybble of that byte refers to the foreground colour at that position on the bit-map grid and the number in the low nybble refers to the background colour.

The colours available for foreground and background are generated by depositing the following values in the high and low nybbles:

BLACK	Ø	ORANGE	8
WHITE	1	BROWN	9
RED	2	PINK	1Ø
CYAN	3	GREY (1)	11
PURPLE	4	GREY (2)	12
GREEN	5	PALE GREEN	13
BLUE	6	PALE BLUE	14
YELLOW	7	GREY (3)	15

To have a background colour of green with a foreground colour of pink (what a combination!), we put 1Ø into the high nybble and 5 into the low nybble, for a total value of 165, because 1Ø*16 + 5*1 = 165 (refer to the discussion of binary unsigned numbers in Chapter 1). If we poke this

number into address 1524 whilst we are in bit-map mode, we should see the pattern at one block in the middle of the screen change its colours to those selected.

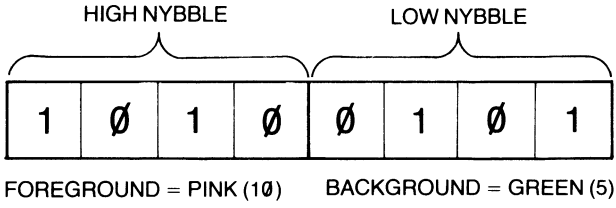


Figure 5.3 Handling of foreground and background colours

By changing the combination of foreground and background colours, we can make the screen *appear* blank while still retaining its data. This is managed by setting the *same* colour for foreground and background over the whole screen. If we put the value 2 into both the low and high nybbles of every screen location, i.e. a total value of 34 in each byte, we then have a completely red screen, its actual contents being hidden. Another way to clear the bit-map screen is by using a ready-written routine (see the routine 'Bit-Maps Clear').

A completely red screen is accomplished by running this line of BASIC which first turns on the bit-map mode and then floods . . . er, *dribbles* the screen with colour:

```
POKE 53265,32: FOR X = 1Ø24 TO 2Ø23 : POKE X,34 : NEXT
```

But, of course, we almost have time for lunch while we wait for BASIC to come up with the goods.

The routine 'New Bit-Map Colours' will do the same job for us in a fraction of a second but, meanwhile, we can make the data on the bit-map screen visible again by giving different colours to the foreground and background, like this:

```
FOR X = 1Ø24 TO 2Ø23 : POKE X,81 : NEXT
```

which will return the pattern to view but in green on white.

BIT-MAPPING ON

START	49525
END	49541
LENGTH	17 BYTES
REGISTERS USED	A

Before we can use bit-map mode, we have to be able to see the section of memory which will make up our screen display. This is done with the routine below, which is only 17 bytes long!

It moves the start of the screen display to location 8192 and then enters bit-map mode by setting bit 5 of location 53265 in the VIC II chip to 1.

Figure 5.4 Typical screen garbage

When the BASIC loader has been entered, saved and run, the screen should look rather like figure 5.4. This display is just the garbage which fills all of the unused memory on power-up. To be able to clear this garbage in under the 28 seconds taken by BASIC, turn to the next routine, 'Bit-Map Clear'.

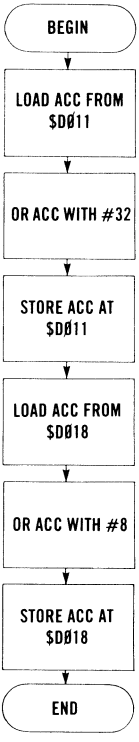


Figure 5.5 'Bit-Mapping On' flowchart

Bit-Map On Assembly

```
Ø  C175  AD11DØ     LDA  $DØ11
1  C178  Ø92Ø       ORA  #$2Ø
2  C17A  8D11DØ     STA  $DØ11
3  C17D  AD18DØ     LDA  $DØ18
4  C18Ø  Ø9Ø8       ORA  #$Ø8
5  C182  8D18DØ     STA  $DØ18
6  C185  6Ø         RTS
```

Bit-Map On Loader

```
1Ø REM -----BIT-MAP ON LOADER-----
2Ø FOR X=49525 TO 49541
3Ø READ A:POKE X,A
4Ø NEXT
5Ø REM ------BIT-MAP ON DATA------
6Ø DATA 173,17,208,9,32,141,17,208
7Ø DATA 173,24,208,9,8,141,24,208,96
```

BIT MAP CLEAR

START	49600
END	49629
LENGTH	30 BYTES
REGISTERS USED	A, X, Y

Before you try out this routine, switch on bit-map mode with 'Bit-Map On' and then type in this piece of BASIC, but press the <RUN/STOP> key before the screen is fully clear.

```
FOR X = 8192 TO 16191 : POKE X,0 : NEXT
```

Dreadfully slow isn't it? Now load this routine, save and run it, and SYS to the start address. See how long it takes to do the same job! My calculations return the result of 0.064206 of a second! That's an increase in speed by a factor of about 450. If that isn't fast enough for you then you're reading the wrong book!

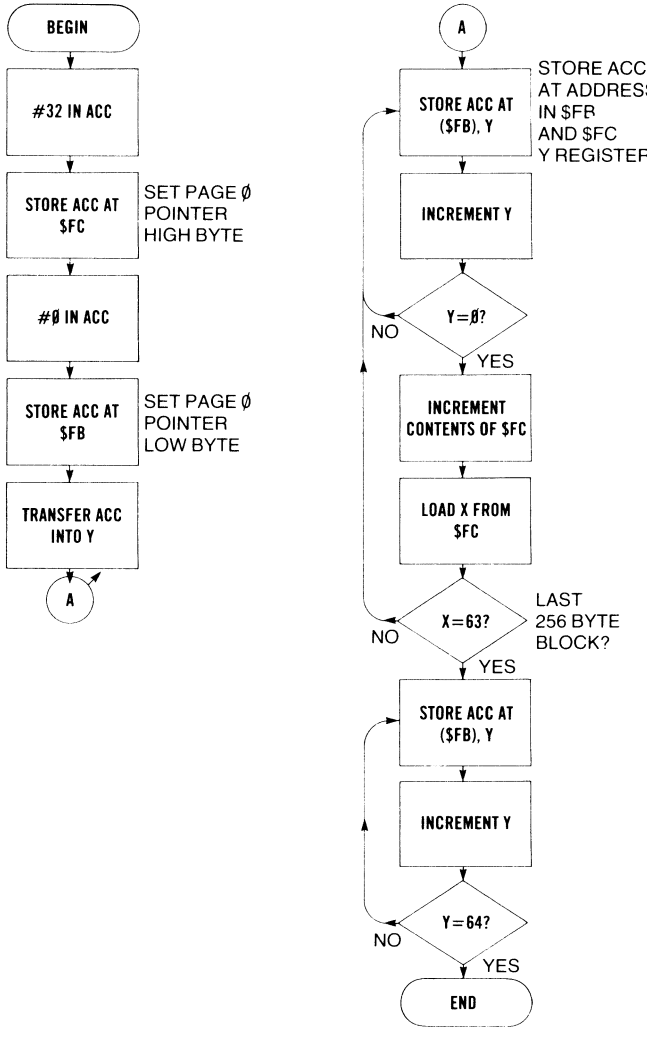
The routine works in the same way as the line of BASIC above, by placing the value 0 in all 8000 locations of the bit-map screen. The routine 'New Bit-Map Colours' later on in this chapter is used to actually change the colours on the screen.

Bit-Map Clear Assembly

```
 0  C1C0  A920      LDA  #$20
 1  C1C2  85FC      STA  $FC
 2  C1C4  A900      LDA  #$00
 3  C1C6  85FB      STA  $FB
 4  C1C8  A8        TAY
 5  C1C9  91FB      STA  ($FB),Y
 6  C1CB  C8        INY
 7  C1CC  D0FB      BNE  $C1C9
 8  C1CE  E6FC      INC  $FC
 9  C1D0  A6FC      LDX  $FC
10  C1D2  E03F      CPX  #$3F
11  C1D4  D0F3      BNE  $C1C9
12  C1D6  91FB      STA  ($FB),Y
13  C1D8  C8        INY
14  C1D9  C040      CPY  #$40
15  C1DB  D0F9      BNE  $C1D6
16  C1DD  60        RTS
```

Bit-Map Clear Loader

```
10 REM -----BIT-MAP SCREEN-CLEAR LOADER-----
20 FOR X=49600 TO 49629
30 READ A:POKE X,A
40 NEXT
50 REM ------BIT-MAP SCREEN-CLEAR DATA------
60 DATA 169,32,133,252,169,0,133,251,168,145
70 DATA 251,200,208,251,230,252,166,252,224,63
80 DATA 208,243,145,251,200,192,64,208,249,96
```



5.6(a) **5.6(b)**

Figure 5.6 'Bit-Map Clear' flowchart

BIT-MAP OFF

START	49545
END	49558
LENGTH	14 BYTES
REGISTERS USED	A

When the time comes to switch off the bit-map mode and return to the text screen, the easiest way from the keyboard is to press the <RUN/STOP> and <RESTORE> keys together but, when this routine is in the appropriate place in memory, it may be accessed from within a program by using the SYS statement.

The routine works in exactly the opposite way to the 'Bit Map On' routine earlier in the chapter, by restoring the screen display to its original place and resetting bit 5 in location 53265 to a Ø.

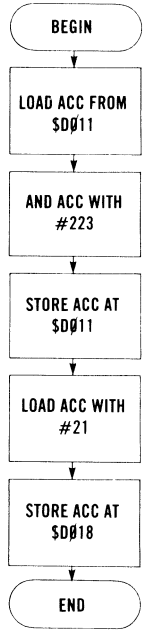


Figure 5.7 'Bit-Map Off' flowchart

Bit-Map Off Assembly

```
0  C189  AD11D0     LDA  $D011
1  C18C  29DF       AND  #$DF
2  C18E  8D11D0     STA  $D011
3  C191  A915       LDA  #$15
4  C193  8D18D0     STA  $D018
5  C196  60         RTS
```

Bit-Map Off Assembly

```
10 REM -----BIT-MAP OFF LOADER-----
20 FOR X=49545 TO 49558
30 READ A:POKE X,A
40 NEXT
50 REM ------BIT-MAP OFF DATA------
60 DATA 173,17,208,41,223,141,17,208
70 DATA 169,21,141,24,208,96
```

BIT MAP SCREEN SAVER

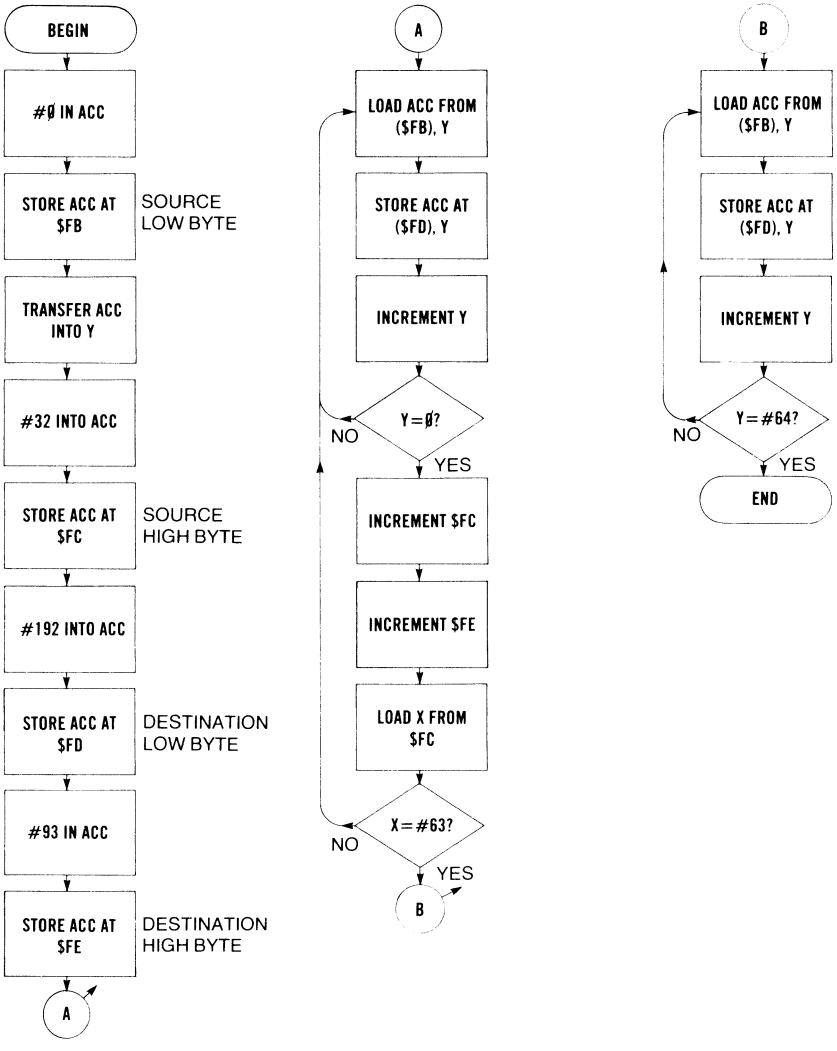
START	49725
END	49768
LENGTH	44 BYTES
REGISTERS USED	A, X, Y

The next two routines are very similar in that they are both used to shift a large block of information from one location in memory to another. In the case of this routine, the section of memory in question is the bit-map screen, which is being moved up to occupy the locations 24000 to 32000, a temporary store for the graphics it contains.

Any previous contents of those addresses will be overwritten, so take care to ensure there is nothing there that is valuable. The transfer is really just a *copying* procedure, as the contents of the source addresses are not altered.

When this routine is used, protect that area of the memory map used as a store since, if your BASIC program is large, it will overrun these locations and corrupt your graphic data. This is done by altering the contents of locations 55 and 56. These two addresses contain the low and high bytes respectively of the highest address available to BASIC: to stop the locations above 24000 being overwritten by BASIC, poke 191 and 93 into addresses 55 and 56.

If you wish to simply clear the addresses in the temporary store area, SYS the 'Bit Map Clear' routine and then use 'Bit-Map Screen Saver' to save the blank screen to the temporary store.



5.8(a) **5.8(b)** **5.8(c)**

Figure 5.8 'Bit-Map Screen Saver' flowchart

Bit-Map Screen Saver Assembly

```
 0  C23D  A900      LDA  #$00
 1  C23F  85FB      STA  $FB
 2  C241  A8        TAY
 3  C242  A920      LDA  #$20
 4  C244  85FC      STA  $FC
 5  C246  A9C0      LDA  #$C0
 6  C248  85FD      STA  $FD
 7  C24A  A95D      LDA  #$5D
 8  C24C  85FE      STA  $FE
 9  C24E  B1FB      LDA  ($FB),Y
10  C250  91FD      STA  ($FD),Y
11  C252  C8        INY
12  C253  D0F9      BNE  $C24E
13  C255  E6FC      INC  $FC
14  C257  E6FE      INC  $FE
15  C259  A6FC      LDX  $FC
16  C25B  E03F      CPX  #$3F
17  C25D  D0EF      BNE  $C24E
18  C25F  B1FB      LDA  ($FB),Y
19  C261  91FD      STA  ($FD),Y
20  C263  C8        INY
21  C264  C040      CPY  #$40
22  C266  D0F7      BNE  $C25F
23  C268  60        RTS
```

Bit-Map Screen Saver Loader

```
10 REM -----BIT-MAP SCREEN SAVER LOADER-----
20 FOR X=49725 TO 49768
30 READ A:POKE X,A
40 NEXT
50 REM ------BIT-MAP SCREEN SAVER DATA------
60 DATA 169,0,133,251,168,169,32,133,252
70 DATA 169,192,133,253,169,93,133,254
80 DATA 177,251,145,253,200,208,249,230
90 DATA 252,230,254,166,252,224,63,208
100 DATA 239,177,251,145,253,200
110 DATA 192,64,208,247,96
```

NEW BIT-MAP SCREEN

START	49680
END	49723
LENGTH	44 BYTES
REGISTERS USED	A, X, Y

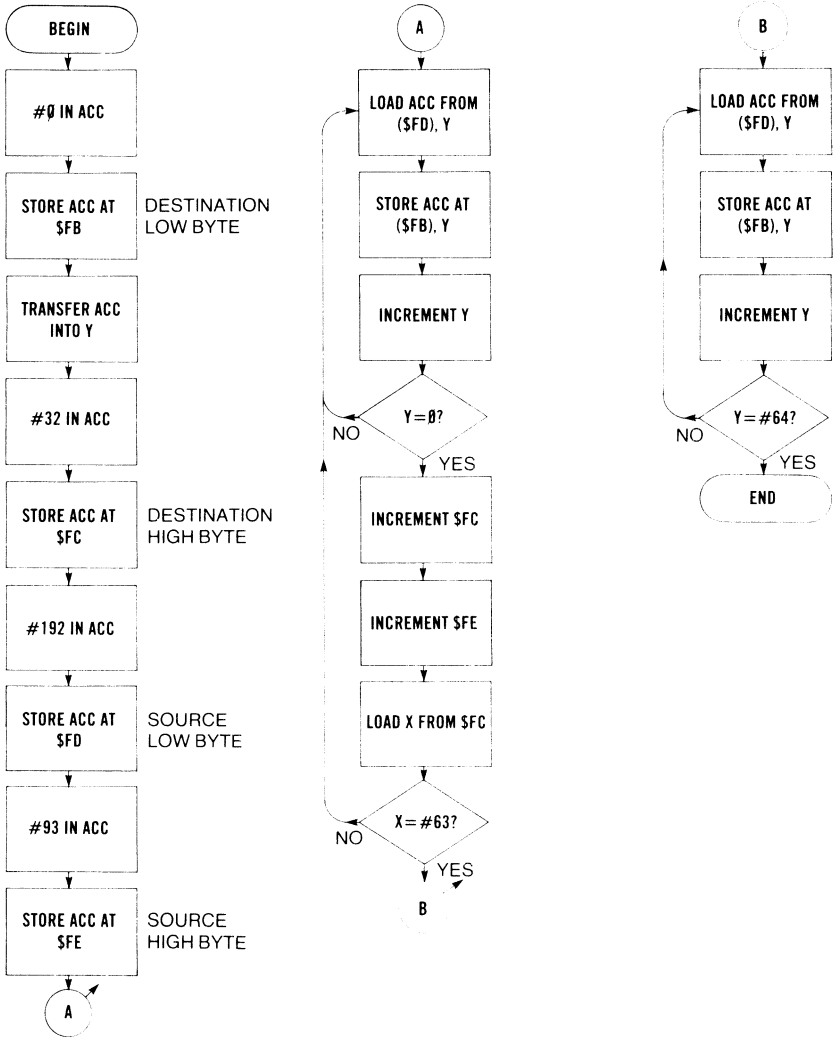
As you can see, the flowchart for this routine is practically identical to that for 'Bit-Map Screen Save' but I have included it for reference anyway.

The only differences in the routine itself are that the source addresses and destination addresses have been transposed so that, to save a bit-map screen, the source for the transfer of data is 8192+ and the destination is 24000+. To reverse the process and retrieve a screen that has previously been saved, the source address will be 24000+ and the destination 8192+.

As with 'Bit-Map Screen Save', the action of calling this program will not alter the contents of the source, only the destination, where all previous contents will be overwritten.

New Bit-Map Screen Assembly

```
 0  C210  A900      LDA  #$00
 1  C212  85FB      STA  $FB
 2  C214  A8        TAY
 3  C215  A920      LDA  #$20
 4  C217  85FC      STA  $FC
 5  C219  A9C0      LDA  #$C0
 6  C21B  85FD      STA  $FD
 7  C21D  A95D      LDA  #$5D
 8  C21F  85FE      STA  $FE
 9  C221  B1FD      LDA  ($FD),Y
10  C223  91FB      STA  ($FB),Y
11  C225  C8        INY
12  C226  D0F9      BNE  $C221
13  C228  E6FC      INC  $FC
14  C22A  E6FE      INC  $FE
15  C22C  A6FC      LDX  $FC
16  C22E  E03F      CPX  #$3F
17  C230  D0EF      BNE  $C221
18  C232  B1FD      LDA  ($FD),Y
19  C234  91FB      STA  ($FB),Y
20  C236  C8        INY
21  C237  C040      CPY  #$40
22  C239  D0F7      BNE  $C232
23  C23B  60        RTS
```



5.9(a) **5.9(b)** **5.9(c)**

Figure 5.9 'New Bit-Map Screen' flowchart

New Bit-Map Screen Loader

```
10 REM -----NEW BIT-MAP SCREEN LOADER-------
20 FOR X=49680 TO 49723
30 READ A:POKE X,A
40 NEXT
50 REM ------NEW BIT-MAP SCREEN DATA--------
60 DATA 169,0,133,251,168,169,32,133,252
70 DATA 169,192,133,253,169,93,133,254
80 DATA 177,253,145,251,200,208,249,230
90 DATA 252,230,254,166,252,224,63,208
100 DATA 239,177,253,145,251,200,192,64
110 DATA 208,247,96
```

NEW BIT-MAP COLOURS

START	49890
END	49922
LENGTH	33 BYTES
REGISTERS USED	A, X, Y

Rather than wait around for BASIC to change the background and foreground colours for us, we can use this routine to do it in a fraction of a second.

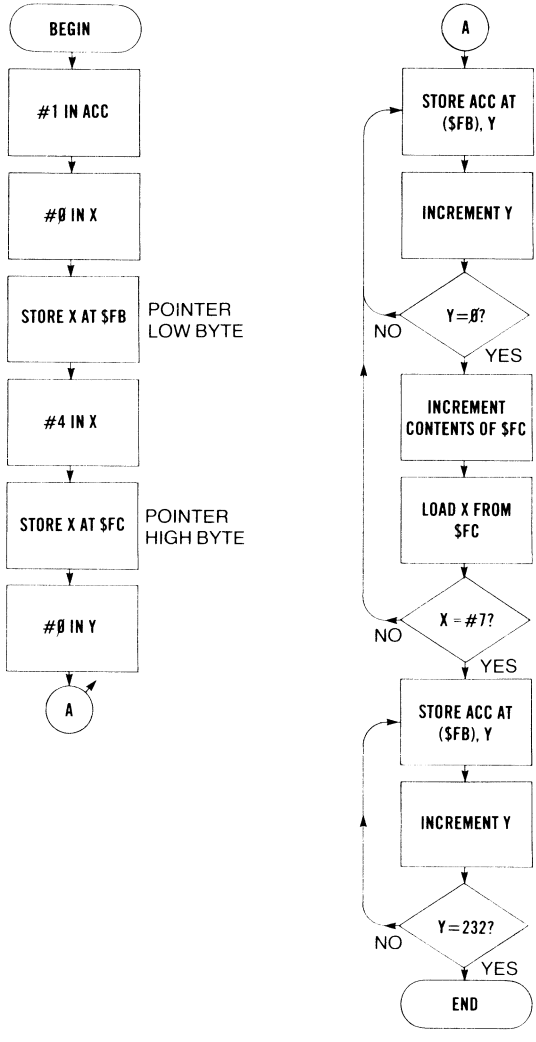
When the loader has been entered, saved and run, we can just SYS the start address and the result will be a foreground colour of black on a white background. For combinations of other colours, the values of low and high nybbles can be found for the required colours by referring to the colour chart above, and the total value poked into the second byte of this routine at address 49891.

Of course, this must be done before SYS is used to call the routine. Any value placed in that location will remain there until altered by the program or directly from the keyboard, e.g.

POKE 49891,34 (*red on red*)

New Bit-Map Colours Assembly

```
 0  C2E2  A901      LDA  #$01
 1  C2E4  A200      LDX  #$00
 2  C2E6  86FB      STX  $FB
 3  C2E8  A204      LDX  #$04
 4  C2EA  86FC      STX  $FC
 5  C2EC  A000      LDY  #$00
 6  C2EE  91FB      STA  ($FB),Y
 7  C2F0  C8        INY
 8  C2F1  D0FB      BNE  $C2EE
 9  C2F3  E6FC      INC  $FC
10  C2F5  A6FC      LDX  $FC
11  C2F7  E007      CPX  #$07
12  C2F9  D0F3      BNE  $C2EE
13  C2FB  91FB      STA  ($FB),Y
14  C2FD  C8        INY
15  C2FE  C0E8      CPY  #$E8
16  C300  D0F9      BNE  $C2FB
17  C302  60        RTS
```



5.1Ø(a) **5.1Ø(b)**

Figure 5.1Ø 'New Bit-Map Colours' flowchart

New Bit-Map Colours Loader

```
10 REM ----- NEW-COLOUR LOADER -----
20 FOR X=49890 TO 49922
30 READ A:POKE X,A
40 NEXT
50 REM ----- NEW-COLOUR DATA --------
60 DATA 169,1,162,0,134,251,162,4,134
70 DATA 252,160,0,145,251,200,208,251,230
80 DATA 252,166,252,224,7,208,243,145,251
90 DATA 200,192,232,208,249,96
```

BIT-MAP INVERTER

START	49560
END	49597
LENGTH	38 BYTES
REGISTERS USED	A, X, Y

It could be useful as part of a game — especially an end-of-game screen explosion — to invert the screen data, i.e. turn the display into its 'negative', where all of the ON bits (1) are turned OFF (0), and all of the OFF bits (0) are turned ON (1).

Screen inversion is quite simple to do in either BASIC or machine language. To invert all bits in a byte, simply subtract its value from 255. For example, starting with a value of 15 in our byte, 15 subtracted from 255 results in 240. Let's confirm this by looking at the bit patterns.

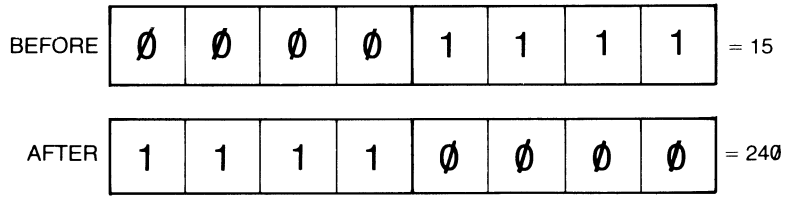
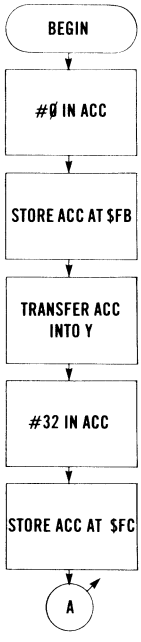
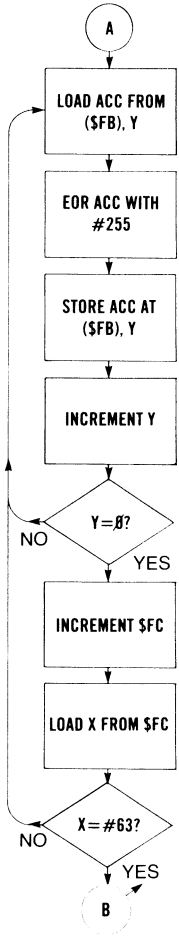
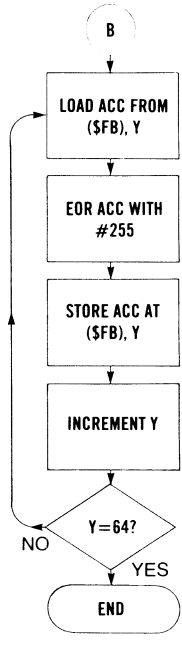


Figure 5.11 Inverting bit patterns

This routine inverts all 8000 bytes on the bit-map screen and is fast enough to use it several times a second as part of a high-speed screen action. To invert a byte in this way is done through a process known as **E**clusive **OR**ing the byte with 255. For more on this, refer to a good book on programming the 6502 or 6510 (see Appendix G).







5.12(a) **5.12(b)** **5.12(c)**

Figure 5.12 'Bit-Map Inverter' flowchart

Bit-Map Inverter Assembly

```
 0  C198  A900      LDA  #$00
 1  C19A  85FB      STA  $FB
 2  C19C  A8        TAY
 3  C19D  A920      LDA  #$20
 4  C19F  85FC      STA  $FC
 5  C1A1  B1FB      LDA  ($FB),Y
 6  C1A3  49FF      EOR  #$FF
 7  C1A5  91FB      STA  ($FB),Y
 8  C1A7  C8        INY
 9  C1A8  D0F7      BNE  $C1A1
10  C1AA  E6FC      INC  $FC
11  C1AC  A6FC      LDX  $FC
12  C1AE  E03F      CPX  #$3F
13  C1B0  D0EF      BNE  $C1A1
14  C1B2  B1FB      LDA  ($FB),Y
15  C1B4  49FF      EOR  #$FF
16  C1B6  91FB      STA  ($FB),Y
17  C1B8  C8        INY
18  C1B9  C040      CPY  #$40
19  C1BB  D0F5      BNE  $C1B2
20  C1BD  60        RTS
```

Bit-Map Inverter Loader

```
10 REM -----BIT-MAP INVERTER LOADER-----
20 FOR X=49560 TO 49597
30 READ A:POKE X,A
40 NEXT
50 REM ------BIT-MAP INVERTER DATA------
60 DATA 169,0,133,251,168,169,32,133,252,177
70 DATA 251,73,255,145,251,200,208,247,230,252
80 DATA 166,252,224,63,208,239,177,251,73,255
90 DATA 145,251,200,192,64,208,245,96
```

BIT-MAP OVERLAY

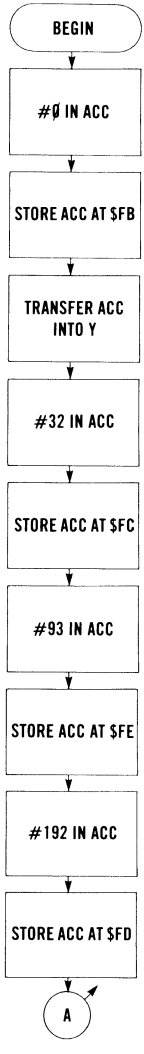
START	49630
END	49677
LENGTH	48 BYTES
REGISTERS USED	A, X, Y

When you have one picture on the visible bit map screen and you have another in the temporary storage area at 24000+ in memory, the two screens can be combined by using this routine to overlay the second onto the first. (This routine could be used to build up a map or a maze in stages.) The combined screen can then be saved to the temporary store by using the 'Bit Map Screen Save' routine earlier in this chapter.

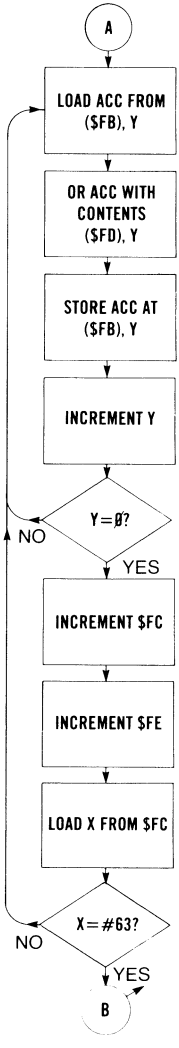
When using the overlay routine, the data at the source addresses is not corrupted but the data at the destination, is; i.e. the bit map screen has another set of data laid over the top of it, so that both screens are visible at the same time. After using this routine it is impossible to separate the two screens, so be sure you are ready to have them combined!

Bit-Map Overlay Assembly

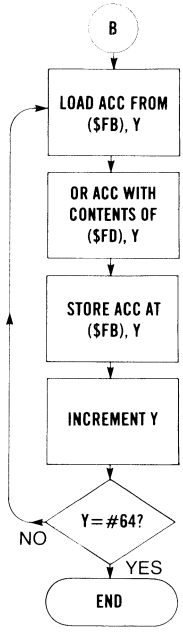
```
 0  C1DE  A900      LDA  #$00
 1  C1E0  85FB      STA  $FB
 2  C1E2  A8        TAY
 3  C1E3  A920      LDA  #$20
 4  C1E5  85FC      STA  $FC
 5  C1E7  A95D      LDA  #$5D
 6  C1E9  85FE      STA  $FE
 7  C1EB  A9C0      LDA  #$C0
 8  C1ED  85FD      STA  $FD
 9  C1EF  B1FB      LDA  ($FB),Y
10  C1F1  11FD      ORA  ($FD),Y
11  C1F3  91FB      STA  ($FB),Y
12  C1F5  C8        INY
13  C1F6  D0F7      BNE  $C1EF
14  C1F8  E6FC      INC  $FC
15  C1FA  E6FE      INC  $FE
16  C1FC  A6FC      LDX  $FC
17  C1FE  E03F      CPX  #$3F
18  C200  D0ED      BNE  $C1EF
19  C202  B1FB      LDA  ($FB),Y
20  C204  11FD      ORA  ($FD),Y
21  C206  91FB      STA  ($FB),Y
22  C208  C8        INY
23  C209  C040      CPY  #$40
24  C20B  D0F5      BNE  $C202
25  C20D  60        RTS
```



5.13(a)



5.13(b)

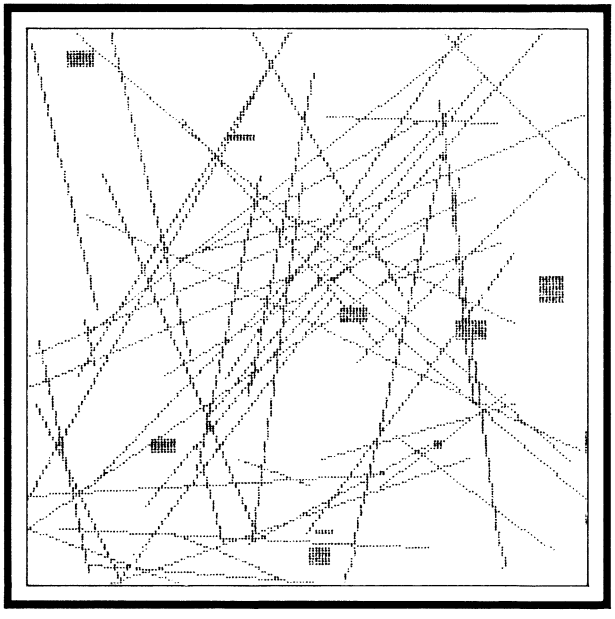


5.13(c)

Figure 5.13 'Bit-Map Overlay' flowchart

Bit-Map Overlay Loader

```
10 REM -----BIT-MAP SCREEN OVERLAY LOADER-----
20 FOR X=49630 TO 49677
30 READ A:POKE X,A
40 NEXT
50 REM -----BIT-MAP SCREEN OVERLAY DATA-------
60 DATA 169,0,133,251,168,169,32,133,252,169
70 DATA 93,133,254,169,192,133,253,177,251
80 DATA 17,253,145,251,200,208,247,230,252
90 DATA 230,254,166,252,224,63,208,237,177
100 DATA 251,17,253,145,251,200,192,64
110 DATA 208,245,96
```



6 Sprite Might

SPRITE CONTROL BY THE VIDEO CHIP

The VIC II chip, as examined at the start of chapter 5, is also responsible for the control of the Commodore 64's sprite facilities.

Up to 8 sprites, numbered Ø to 7, may be controlled at any one time. There are quite a number of registers within the VIC II chip concerned with sprites, but as they are rather dotted around the VIC II's memory we will look at them function by function, rather than in chronological order. These locations are:

- Location 53269, sprite enabling,
- Locations 53248 to 53264, sprite positioning,
- Locations 53277 and 53271, sprite expansion,
- Locations 53287 to 53294, sprite colour,
- Location 53276, sprite multicolour,

- Locations 53278 and 53279, sprite collisions, and
- Location 53275, sprite priority.

Location 53269 Sprite Enable

This single location contains eight 'switches', one for each of the eight sprites. To 'switch on' or 'enable' sprite Ø, bit Ø of location 53269 must be set to 1, and so on. The process must be reversed to disable the sprites.

Locations 53248 to 53264 Sprite Position

Each sprite may be positioned anywhere on the screen by placing the X and Y co-ordinates of its position into the appropriate registers in this block. These are the registers and the sprites they are concerned with:

SPRITE	LOCATION	CO-ORDINATES
Sprite Ø	53248	X
	53249	Y
Sprite 1	5325Ø	X
	53251	Y
Sprite 2	53252	X
	53253	Y
Sprite 3	53254	X
	53255	Y
Sprite 4	53256	X
	53257	Y
Sprite 5	53258	X
	53259	Y
Sprite 6	5326Ø	X
	53261	Y
Sprite 7	53262	X
	53263	Y

This leaves us with a problem: the screen is 32Ø pixels wide but the register to contain the X position of any sprite is only 8 bits wide, which limits us to the range Ø to 255.

Here is the answer. Each bit in location 53264 is allocated to one of the sprites: bit Ø is for sprite Ø, bit 1 for sprite 1, and so on. Each of these bits may be regarded as the ninth bit of the location which holds the X position of any of the sprites. To make any sprite travel across the

righthand side of the screen, to continue the action after the X position reaches 255, the appropriate bit of location 53264 must be set to 1. The contents of the X position register for that sprite must stay the same.

Location 53277 and 53271 Sprite Expansion

These two locations can be used to make any of the eight sprites expand in either the X or Y direction, or both at the same time. As above, in either of these registers bit Ø refers to sprite Ø, and so on to bit 7 and sprite 7.

To make any sprite expand widthways, i.e. in the X direction, we place a 1 in the bit corresponding to that sprite in location 53277. Similarly, to make any sprite expand vertically, we place a 1 in the appropriate bit in location 53271.

The expanded sprite has its top lefthand corner in the original position as governed by the values in the sprite position registers, and expands away from that point to the right or downwards, for X and Y expansion respectively.

Locations 53287 to 53294 Sprite Colour

These eight addresses may each contain a number in the range Ø to 15 which gives each sprite its colour. If we put the value Ø at address 53294 then, when enabled, the foreground of sprite 7 will be black. These registers are laid out in an obvious pattern, with address 53287 governing the colour for sprite Ø, 53288 for sprite 1 and so on for all eight sprites up to location 53294.

Location 53276 Sprite Multicolour

By re-setting any bit within this byte to Ø, we can elect to have a particular sprite displayed in up to four colours at once. (As usual, the bit number is the same as the sprite number.) As with the multicolour bit-map mode, there is a price to pay: the horizontal resolution of the sprite is halved. Any sprite chosen to appear in glorious technicolour will be only 12 pairs of bits wide and the two bits in each pair must be the same colour.

The colours in which we wish to show our sprite must go to the appropriate locations and the numbers to go into the sprite data area to select that colour are shown alongside in the following table:

BIT PAIRS	COLOUR
ØØ	Screen colour
Ø1	Colour in location 53285
1Ø	Colour in colour register for that sprite
11	Colour in location 53286

Locations 53278 and 53279 Sprite Collisions

Yes, the trusty VIC II can even detect a pile-up between sprite and sprite, and sprite and background!

If any bit in location 53278 is set to **1**, the sprite corresponding to that bit number has been in collision with another sprite, whose bit number will also be set to **1**. If more than two bits are set at the same time, then more than two sprites have collided at the same spot!

PEEKing to this address to determine whether any sprites have actually crashed clears all bits within it, so it is wise to store the result of such a PEEK until you have finished with the collision information obtained.

Location 53279 works in a similar way to that of 53278, except that it detects collisions between sprites and the background.

Location 53275 Sprite Priority

A sprite of a higher number than another will appear to pass in front of it. Strictly, this situation is a collision (see locations 53278-9 above) but, if no action is taken to generate an explosion, say, the higher-rating priority sprite will overlap the other momentarily and then continue on its way.

Normally, all sprites have greater priority than all background data but, by setting any bit at this address to **1**, the sprite of the same number then has a *lower* priority than the background and appears to travel or lie behind the background, rather than in front of it.

SPRITE REVERSER

START	49770
END	49836
LENGTH	67 BYTES
REGISTERS USED	A, X, Y

Many different types of arcade games involve a figure rushing to and fro on the screen, under control of either the player or the program itself.

By using this routine, when the car, plane, boat, train or whatever reaches the side of the screen or playing area, it turns around ready for the trip back. So instead of the train, say, facing right, it will face left after this routine is called by the SYS statement. The position, colour and size of the sprite are unaffected.

To use the routine, the address of the first byte of the sprite must first by poked into locations 253 and 254, in the familiar low byte, high byte fashion. All that remains is to:

```
SYS 49770
```

Once the address of the start of the sprite has been poked into locations 253 and 254, the routine may be called repeatedly by SYS, until another sprite needs to be reversed, when the start address of that sprite must be poked into locations 253 and 254 in preparation for the routine.

If a sprite has been reversed using this routine and then another routine from this book is used to do a different job, the same two page 0 locations (253 and 254) will probably have been used in the process. To reverse the same sprite again, you will need to poke the start address of the sprite data into those locations before you SYS the routine. If you forget, you could reverse part of the operating system perhaps or BASIC, either of which will almost certainly lead to a crash!

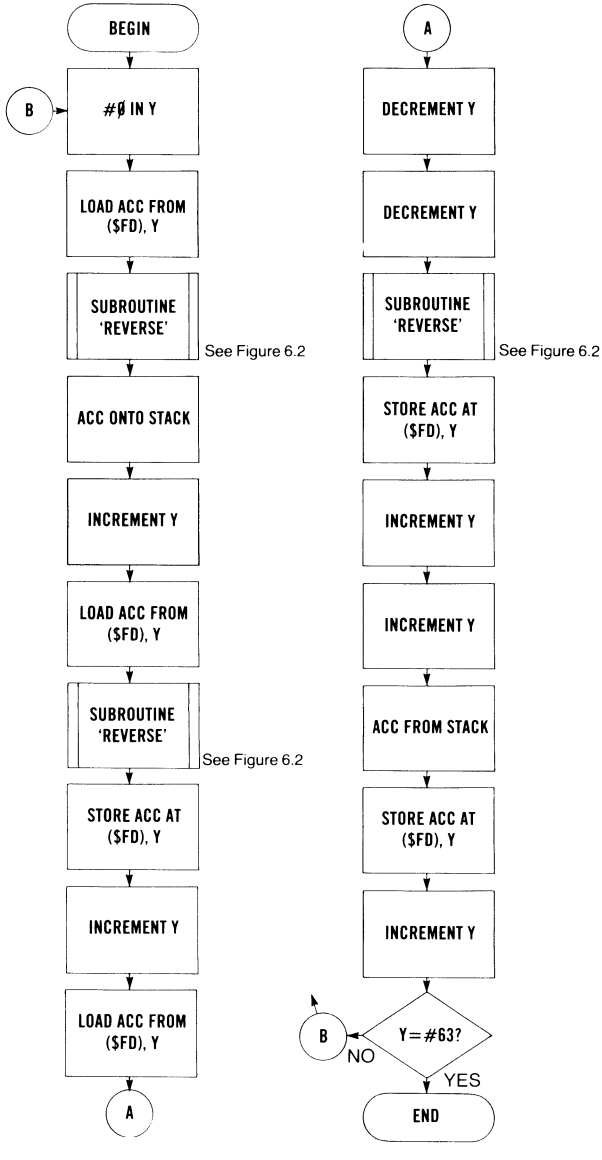
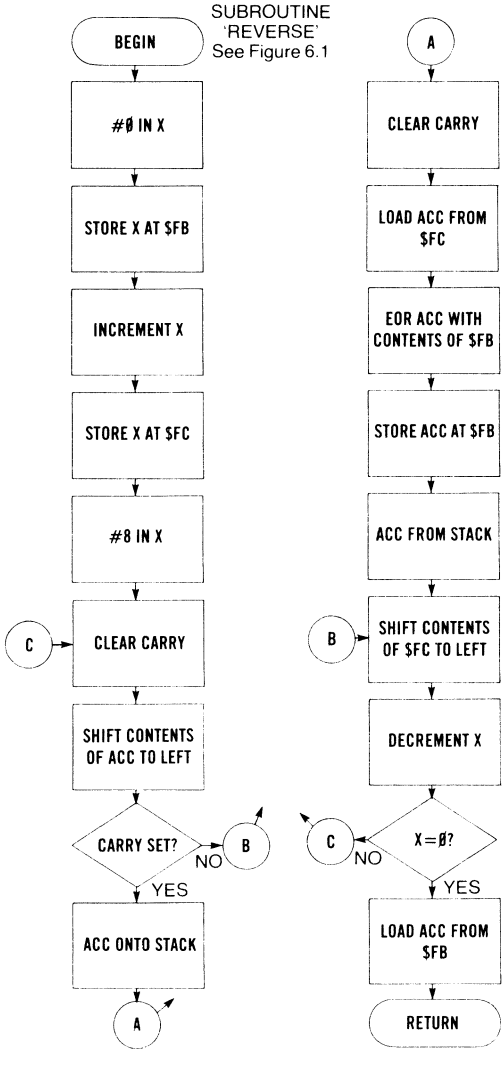


Figure 6.1 'Sprite Reverser' flowchart



6.2(a) **6.2(b)**

Figure 6.2 'Sprite Reverser' reversing subroutine flowchart

Sprite Reverser Assembly

```
 0  C26A  A000      LDY  #$00
 1  C26C  B1FD      LDA  ($FD),Y
 2  C26E  208FC2    JSR  $C28F
 3  C271  48        PHA
 4  C272  C8        INY
 5  C273  B1FD      LDA  ($FD),Y
 6  C275  208FC2    JSR  $C28F
 7  C278  91FD      STA  ($FD),Y
 8  C27A  C8        INY
 9  C27B  B1FD      LDA  ($FD),Y
10  C27D  88        DEY
11  C27E  88        DEY
12  C27F  208FC2    JSR  $C28F
13  C282  91FD      STA  ($FD),Y
14  C284  C8        INY
15  C285  C8        INY
16  C286  68        PLA
17  C287  91FD      STA  ($FD),Y
18  C289  C8        INY
19  C28A  C03F      CPY  #$3F
20  C28C  D0DE      BNE  $C26C
21  C28E  60        RTS
22  C28F  A200      LDX  #$00
23  C291  86FB      STX  $FB
24  C293  E8        INX
25  C294  86FC      STX  $FC
26  C296  A208      LDX  #$08
27  C298  18        CLC
28  C299  0A        ASL  A
29  C29A  9009      BCC  $C2A5
30  C29C  48        PHA
31  C29D  18        CLC
32  C29E  A5FC      LDA  $FC
33  C2A0  45FB      EOR  $FB
34  C2A2  85FB      STA  $FB
35  C2A4  68        PLA
36  C2A5  06FC      ASL  $FC
37  C2A7  CA        DEX
38  C2A8  D0EE      BNE  $C298
39  C2AA  A5FB      LDA  $FB
40  C2AC  60        RTS
```

Sprite Reverser Loader

```
10 REM --------REVERSER LOADER--------
20 FOR X=49770 TO 49836
30 READ A:POKE X,A
40 NEXT
50 REM ---------REVERSER DATA---------
60 DATA 160,0,177,253,32,143,194,72,200,177
70 DATA 253,32,143,194,145,253,200,177,253,136
80 DATA 136,32,143,194,145,253,200,200,104
90 DATA 145,253,200,192,63,208,222,96,162,0
100 DATA 134,251,232,134,252,162,8,24,10,144
110 DATA 9,72,24,165,252,69,251,133,251,104,6
120 DATA 252,202,208,238,165,251,96
```

UPSIDE-DOWN SPRITES

START	49840
END	49891
LENGTH	52 BYTES
REGISTERS USED	A, X, Y

By using the previous routine 'Sprite Reverser', we may turn a sprite from facing right to facing left or vice-versa. With this routine we can turn the sprite upside down, again without having any affect on its colour, size or position. Just as with the 'Sprite Reverser' routine, the address of the first byte of the sprite data must be poked into page 0 locations 253 and 254 before the routine is used. The call statement is:

SYS 49840

Again, if we wish to use the routine repeatedly, we must not have previously altered the address poked into 253 and 254 or we risk a serious crash.

Upside Down Assembly

```
 0  C2B0  A21D      LDX  #$1D
 1  C2B2  86FB      STX  $FB
 2  C2B4  A023      LDY  #$23
 3  C2B6  84FC      STY  $FC
 4  C2B8  A903      LDA  #$03
 5  C2BA  8502      STA  $02
 6  C2BC  A4FB      LDY  $FB
 7  C2BE  B1FD      LDA  ($FD),Y
 8  C2C0  48        PHA
 9  C2C1  A4FC      LDY  $FC
10  C2C3  B1FD      LDA  ($FD),Y
11  C2C5  A4FB      LDY  $FB
12  C2C7  91FD      STA  ($FD),Y
13  C2C9  68        PLA
14  C2CA  A4FC      LDY  $FC
15  C2CC  91FD      STA  ($FD),Y
16  C2CE  C6FB      DEC  $FB
17  C2D0  C6FC      DEC  $FC
18  C2D2  C602      DEC  $02
19  C2D4  D0E6      BNE  $C2BC
20  C2D6  A5FB      LDA  $FB
21  C2D8  3009      BMI  $C2E3
22  C2DA  A5FC      LDA  $FC
23  C2DC  18        CLC
24  C2DD  6906      ADC  #$06
25  C2DF  85FC      STA  $FC
26  C2E1  D0D5      BNE  $C2B8
27  C2E3  60        RTS
```

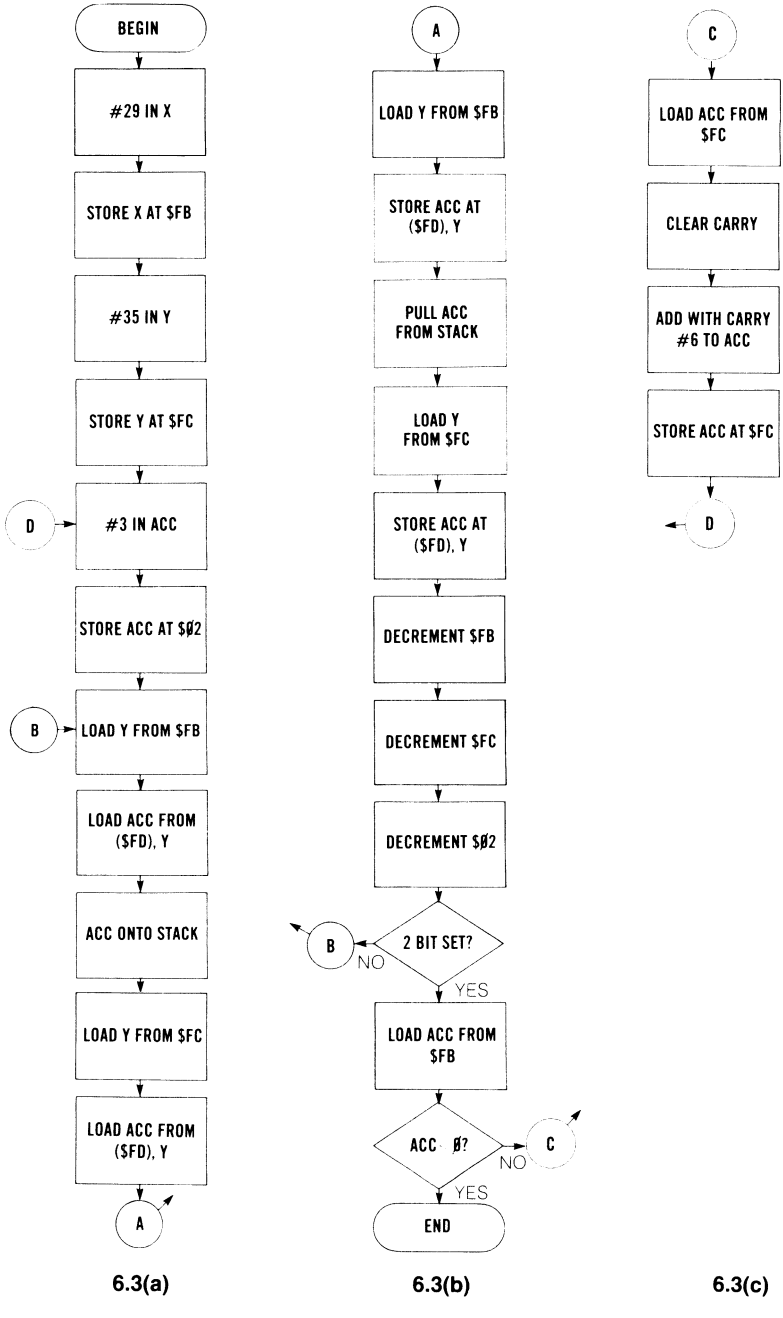


Figure 6.3 'Upside-Down Sprites' flowchart

Upside Down Loader

```
10 REM-------UPSIDEDOWN LOADER----------
20 FOR X=49840 TO 49891
30 READ A:POKE X,A
40 NEXT
50 REM--------UPSIDEDOWN DATA-----------
60 DATA 162,29,134,251,160,35,132,252,169
70 DATA 3,133,2,164,251,177,253,72,164,252
80 DATA 177,253,164,251,145,253,104,164,252
90 DATA 145,253,198,251,198,252,198,2,208
100 DATA 230,165,251,48,9,165,252,24,105
110 DATA 6,133,252,208,213,96
```

SPRITE TOGGLER

START	5Ø215
END	5Ø257
LENGTH	43 BYTES
REGISTERS USED	A, X, Y

A toggle switch is like an old-fashioned pull-cord switch: pull for ON, pull for OFF. The same action will switch the light on or off depending on which state the light was in when the switch was pulled. This routine works in the same way.

If we use the routine on sprite 4, say, when that sprite is switched off, then it will switch the sprite on, and vice-versa.

The call statement for using this routine is a little different to most of the routines we have used so far. After the SYS address, we must specify a sprite number, like this:

```
SYS 5Ø215,4          (to toggle sprite 4 ON or OFF)
```

The comma and the sprite number *must* be used. If they are left off, an error message is produced.

The routine uses three subroutines which are built into the ROM of the Commodore 64, which:

- Check for the comma,
- Evaluate the following expresson and
- Convert the result to a two-byte integer.

The upshot of all this is that not only can we specify a sprite by typing

```
SYS 5Ø215,4
```

but, for example, we may also type in

```
SYS 5Ø215,SN
```

where SN stands for **S**prite **N**umber. (Note that a selection may be made from eight sprites only, so that SN may take only the values Ø to 7.)

This method of transferring information to the routine is extremely useful. We will see it used in later routines to very good effect.

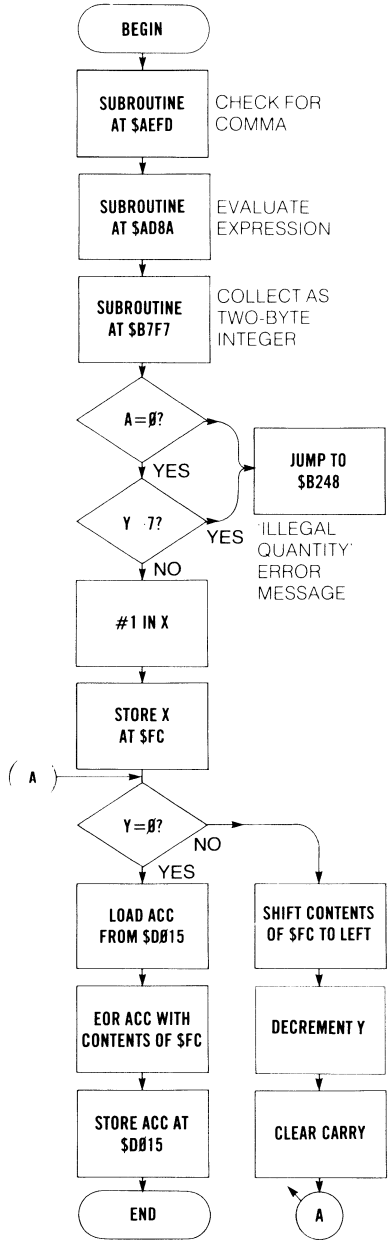


Figure 6.4 'Sprite Toggler' flowchart

Sprite Toggler Assembly

```
 0  C427  20FDAE    JSR  $AEFD
 1  C42A  208AAD    JSR  $AD8A
 2  C42D  20F7B7    JSR  $B7F7
 3  C430  C900      CMP  #$00
 4  C432  D01B      BNE  $C44F
 5  C434  C007      CPY  #$07
 6  C436  B017      BCS  $C44F
 7  C438  A201      LDX  #$01
 8  C43A  86FC      STX  $FC
 9  C43C  C000      CPY  #$00
10  C43E  F006      BEQ  $C446
11  C440  06FC      ASL  $FC
12  C442  88        DEY
13  C443  18        CLC
14  C444  90F6      BCC  $C43C
15  C446  AD15D0    LDA  $D015
16  C449  45FC      EOR  $FC
17  C44B  8D15D0    STA  $D015
18  C44E  60        RTS
19  C44F  4C48B2    JMP  $B248
```

Sprite Toggler Loader

```
10 REM------SPRITE TOGGLER LOADER------
20 FOR X=50215 TO 50257
30 READ A:POKE X,A
40 NEXT
50 REM-------SPRITE TOGGLER DATA-------
60 DATA 32,253,174,32,138,173,32,247,183
70 DATA 201,0,208,27,192,7,176,23,162,1
80 DATA 134,252,192,0,240,6,6,252,136,24
90 DATA 144,246,173,21,208,69,252,141,21
100 DATA 208,96,76,72,178
```

SPRITE COLOUR SET

START	50265
END	50310
LENGTH	46 BYTES
REGISTERS USED	A, X, Y

The technique of following the SYS address of the routine by other information is taken a step further here, because to change the colour of a sprite by using this routine we must supply both a sprite number and a colour number.

The call statement is

```
SYS 50265,SN,COL
```

where SN is the **S**prite **N**umber and COL is its **COL**our. Again, SN and COL may be real numbers or variables, but SN must be in the range 0 to 7, and COL must be in the range 0 to 15.

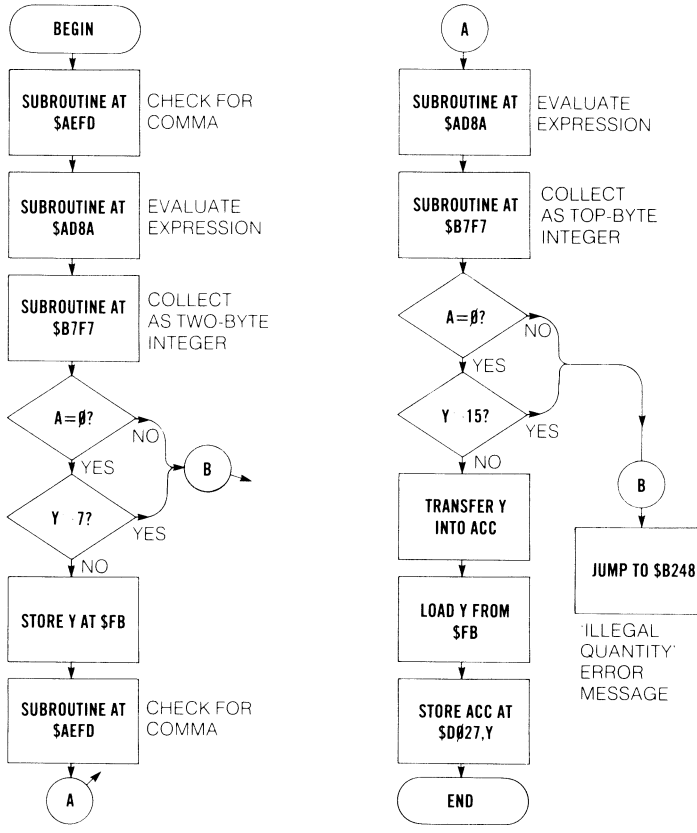
The colours available are shown in figure 6.5. Say we choose to change the colour of SPRITE 1 to black (0) by the statement

```
SYS 50265,1,0
```

SPRITE 1 need not be switched on at the time. That is, we may toggle a sprite OFF with the 'Sprite Toggler' routine, change its colour while it is invisible and cause it to reappear in a different position, in a different colour.

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	PINK
3	CYAN	11	GREY 1
4	PURPLE	12	GREY 2
5	GREEN	13	PALE GREEN
6	BLUE	14	PALE BLUE
7	YELLOW	15	GREY 3

Figure 6.5 Sprite colour codes



6.6(a)

6.6(b)

Figure 6.6 'Sprite Colour Set' flowchart

Sprite Colour Set Assembly

```
 0  C459  20FDAE    JSR  $AEFD
 1  C45C  208AAD    JSR  $AD8A
 2  C45F  20F7B7    JSR  $B7F7
 3  C462  C900      CMP  #$00
 4  C464  D01E      BNE  $C484
 5  C466  C008      CPY  #$08
 6  C468  B01A      BCS  $C484
 7  C46A  84FB      STY  $FB
 8  C46C  20FDAE    JSR  $AEFD
 9  C46F  208AAD    JSR  $AD8A
10  C472  20F7B7    JSR  $B7F7
11  C475  C900      CMP  #$00
12  C477  D00B      BNE  $C484
13  C479  C010      CPY  #$10
14  C47B  B007      BCS  $C484
15  C47D  98        TYA
16  C47E  A4FB      LDY  $FB
17  C480  9927D0    STA  $D027,Y
18  C483  60        RTS
19  C484  4C48B2    JMP  $B248
```

Sprite Colour Set Loader

```
10 REM-----SPRITE COLOUR SETTER LOADER-----
20 FOR X=50265 TO 50310
30 READ A:POKE X,A
40 NEXT
50 REM------SPRITE COLOUR SETTER DATA------
60 DATA 32,253,174,32,138,173,32,247,183
70 DATA 201,0,208,30,192,8,176,26,132,251
80 DATA 32,253,174,32,138,173,32,247,183
90 DATA 201,0,208,11,192,16,176,7,152,164
100 DATA 251,153,39,208,96,76,72,178
```

NEW SPRITE

START	5Ø315
END	5Ø38Ø
LENGTH	66 BYTES
REGISTERS USED	A, X, Y

One of the limitations of the VIC II video chip in the Commodore 64 is that it can only address 16 Kbytes of memory at a time. Unless we start banking memory in and out to enable the VIC II to see a different 16 Kbyte area, all of the data for our eight sprites must be located below address 16384.

There are areas below that address where sprite data is commonly stored, but only one of these may be considered safe, i.e. location 7Ø4 to 766, sprite area 11. These addresses are otherwise unused by the machine. The cassette buffer at locations 828 to 1Ø19 may be used to hold sprite data but, if the LOAD or SAVE commands are used, your sprite is overwritten with garbage. All addresses over 2Ø48 are likely to contain your BASIC program, so we are still faced with the problem of where to store sprite data so that it is safe but available.

There are two ways around the problem:

1 The first solution is to move up the pointer to the bottom of free memory by an amount large enough to make room for eight sprites, i.e. 64 x 8 = 512 bytes. Let's be greedy and give ourselves 1 Kbyte. To do this we use the statement:

```
POKE 44,PEEK(44)+4
```

Location 44 is the high byte of the system pointer to the start of BASIC text, so we are just pointing to a place 1Ø24 bytes higher up in memory.

2 This second solution is used when your particular application requires the use of large numbers of different sprites. The statement

```
SYS 5Ø315,ADD,SN
```

wipes the 63 bytes from address ADD into the block of memory you have allocated to sprite SN. If SPRITE 1 takes its data from block 11, that 63 bytes will be filled with the data in the 63 bytes from the address following the SYS command.

Ideally, we can use a combination of both methods, so that we gain 1 Kbyte of memory, say, in which to house our current sprite data but we are also able to store almost any number of sprites in any free area of memory we may create. When we use this routine, the address following

the SYS 50315 can be any address within the full range of the 6510 CPU (0 to 65535). Of course, the sprite number will be in the range 0 to 7.

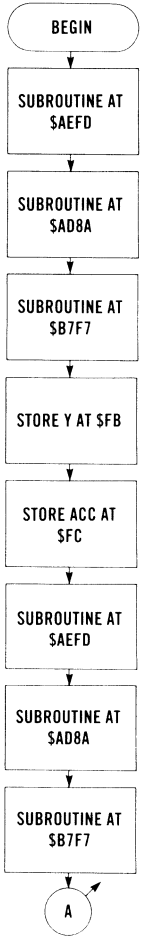
Warning Make sure that the sprite number you choose to receive the data from high memory has first been allocated space in the usual way: e.g. for SPRITE 1,

POKE 2041,11

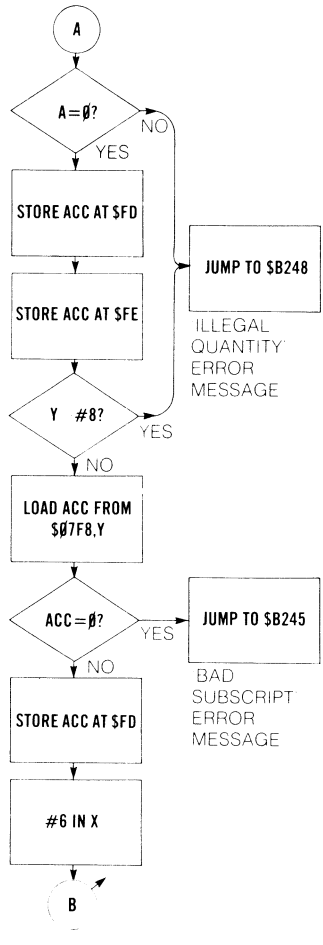
if SPRITE 1 is to take its data from sprite block 11. If you do not do this then the result could be a resounding crash!

New Sprite Assembly

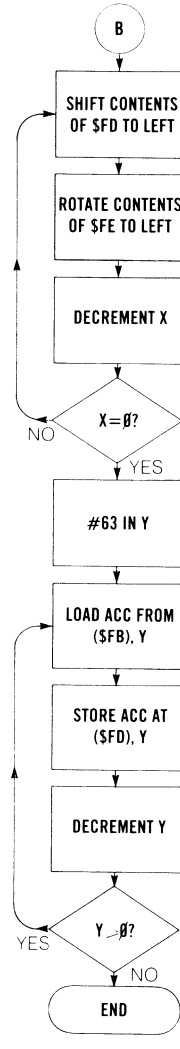
```
 0  C48B  20FDAE    JSR  $AEFD
 1  C48E  208AAD    JSR  $AD8A
 2  C491  20F7B7    JSR  $B7F7
 3  C494  84FB      STY  $FB
 4  C496  85FC      STA  $FC
 5  C498  20FDAE    JSR  $AEFD
 6  C49B  208AAD    JSR  $AD8A
 7  C49E  20F7B7    JSR  $B7F7
 8  C4A1  C900      CMP  #$00
 9  C4A3  D022      BNE  $C4C7
10  C4A5  85FD      STA  $FD
11  C4A7  85FE      STA  $FE
12  C4A9  C008      CPY  #$08
13  C4AB  B01A      BCS  $C4C7
14  C4AD  B9F807    LDA  $07F8,Y
15  C4B0  F018      BEQ  $C4CA
16  C4B2  85FD      STA  $FD
17  C4B4  A206      LDX  #$06
18  C4B6  06FD      ASL  $FD
19  C4B8  26FE      ROL  $FE
20  C4BA  CA        DEX
21  C4BB  D0F9      BNE  $C4B6
22  C4BD  A03F      LDY  #$3F
23  C4BF  B1FB      LDA  ($FB),Y
24  C4C1  91FD      STA  ($FD),Y
25  C4C3  88        DEY
26  C4C4  10F9      BPL  $C4BF
27  C4C6  60        RTS
28  C4C7  4C48B2    JMP  $B248
29  C4CA  4C45B2    JMP  $B245
```



6.7(a)



6.7(b)

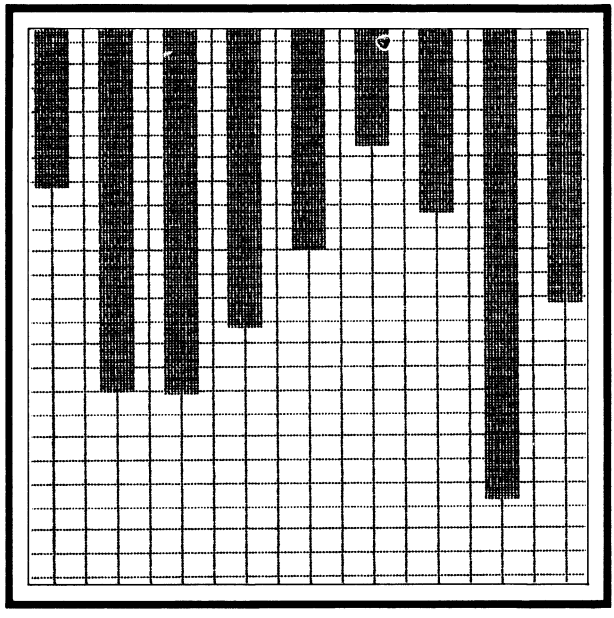


6.7(c)

Figure 6.7 'New Sprite' flowchart

New Sprite Loader

```
10 REM--------NEW SPRITE LOADER---------
20 FOR X=50315 TO 50380
30 READ A:POKE X,A
40 NEXT
50 REM---------NEW SPRITE DATA----------
60 DATA 32,253,174,32,138,173,32,247,183
70 DATA 132,251,133,252,32,253,174,32,138
80 DATA 173,32,247,183,201,0,208,34,133
90 DATA 253,133,254,192,8,176,26,185,248
100 DATA 7,240,24,133,253,162,6,6,253,38
110 DATA 254,202,208,249,160,63,177,251
120 DATA 145,253,136,16,249,96,76,72,178
130 DATA 76,69,178
```



7 Miscellaneous Utilities

This chapter contains a selection of machine language routines which you may find useful, either to supplement your own BASIC programs or as components of your machine language programs.

I hope the examples in this book have shown you that machine language subroutines, and indeed, whole programs, can be most useful to the user of the home micro. BASIC is a fine tool for many programming tasks but, when faced with the speed and efficiency that we have seen over the past few chapters, it must step aside for a faster worker from time to time.

When you start writing your own machine language programs, please don't take these examples of mine as the ultimate; a program is never complete or perfect. Someone will always come along and write a program to do the same job in less time using less memory! Just use my subroutines as springboards to move on to bigger and better programs of your own.

PRESS ANY KEY TO CONTINUE

START	49925
END	49930
LENGTH	6 BYTES
REGISTERS USED	A, X, Y

No flow diagram has been included with this routine because it is simplicity itself and by this time you should as a matter of course be making up charts far more complex than the one for this routine.

There are only three parts to the routine, but the first of these is a jump to a subroutine in the KERNAL of the Commodore 64. This subroutine takes a character from the keyboard queue and places it in the accumulator. If there is nothing in the queue, then the value in the accumulator will be zero, and the next instruction checks for this possibility.

If the accumulator contains zero, the routine branches back to take another look at the queue and continues this process until a character is found. For the sake of six bytes of memory, we may have any number of 'Press Any Key' pauses we like.

'Press Any Key To Continue . . .' Assembly

```
0  C305  20E4FF    JSR  $FFE4
1  C308  F0FB      BEQ  $C305
2  C30A  60        RTS
```

'Press Any Key To Continue . . .' Loader

```
10 REM ----- PRESS-ANY... LOADER-----
20 FOR X=49925 TO 49930
30 READ A:POKE X,A
40 NEXT
50 REM ----- PRESS-ANY... DATA ------
60 DATA 32,228,255,240,251,96
```

STOP KEY DETECTOR

START	49935
END	4994Ø
LENGTH	6 BYTES
REGISTERS USED	A, X

This routine is very similar to 'Press Any Key To Continue . . .' and again there is no flow diagram. This time, instead of checking for just any key, the routine will only continue if the <STOP> key is pressed. Otherwise, its working is identical to the 'Press Any Key To Continue . . .' routine.

Note If this routine is called from a BASIC program it will cause the program to exit to direct mode when the <STOP> key is pressed.

Stop Key Detect Assembly

```
Ø  C3ØF  2ØE1FF    JSR  $FFE1
1  C312  DØFB      BNE  $C3ØF
2  C314  6Ø        RTS
```

Stop Key Detect Loader

```
1Ø REM -----DETECT-STOP-KEY LOADER-----
2Ø FOR X=49935 TO 49940
3Ø READ A:POKE X,A
4Ø NEXT
5Ø REM ------DETECT-STOP-KEY DATA------
6Ø DATA 32,225,255,2Ø8,251,96
```

DECIMAL TO BINARY CONVERSION

START	49945
END	49974
LENGTH	30 BYTES
REGISTERS USED	A, X, Y

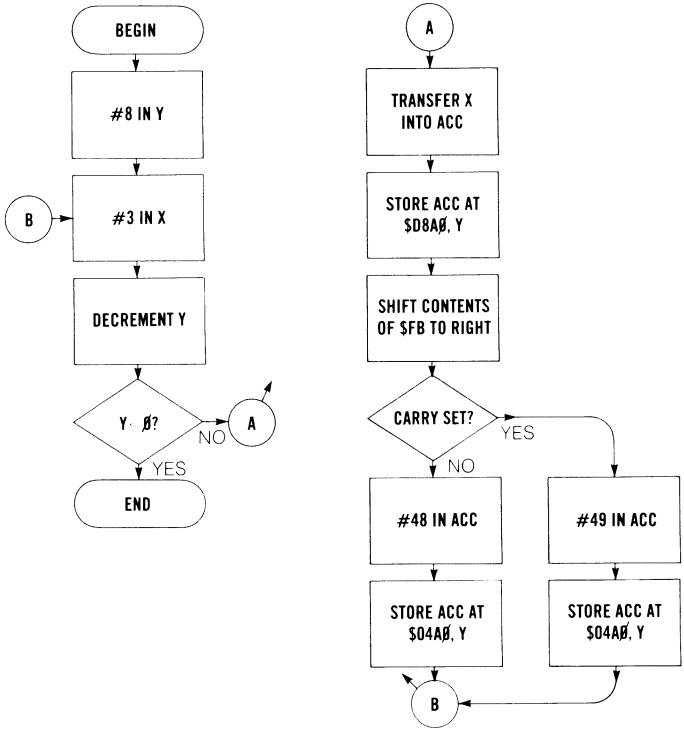
This routine will take any number in the range 0 to 255 and turn it into its binary equivalent, which is displayed on the fourth line of the screen as a series of 0s and 1s. The routine takes the number to be converted from location 251, so we need to poke the number we wish to convert into that address prior to using the SYS statement.

When building up graphic displays in bit map mode, this routine can be quite useful for checking the bit patterns before entry. The routine works by shifting the number to the right in its own location, testing the carry bit to see if the bit falling out of the right end is a 0 or a 1, and displaying the appropriate character on the screen.

The number converted is destroyed in the process so, to use the routine twice on the same number, the number must be poked into address 251 once before each SYS.

Decimal to Binary Conversion Assembly

```
 0  C319  A008      LDY  #$08
 1  C31B  A203      LDX  #$03
 2  C31D  88        DEY
 3  C31E  3016      BMI  $C336
 4  C320  8A        TXA
 5  C321  99A0D8    STA  $D8A0,Y
 6  C324  46FB      LSR  $FB
 7  C326  B007      BCS  $C32F
 8  C328  A930      LDA  #$30
 9  C32A  99A004    STA  $04A0,Y
10  C32D  D0EC      BNE  $C31B
11  C32F  A931      LDA  #$31
12  C331  99A004    STA  $04A0,Y
13  C334  D0E5      BNE  $C31B
14  C336  60        RTS
```



7.1(a) 7.1(b)

Figure 7.1 'Decimal to Binary Conversion' flowchart

Decimal to Binary Conversion Loader

```
10 REM -----DEC TO BIN LOADER-----
20 FOR X=49945 TO 49974
30 READ A:POKE X,A
40 NEXT
50 REM -----DEC TO BIN DATA ------
60 DATA 160,8,162,3,136,48,22,138,153
70 DATA 160,216,70,251,176,7,169,48,153
80 DATA 160,4,208,236,169,49,153,160
90 DATA 4,208,229,96
```

MEMORY BLOCK MOVER

START	49975
END	49988
LENGTH	14 BYTES
REGISTERS USED	A, Y

When it is necessary to shift the contents of a block of memory to another location or, to be more accurate, *copy* a block of memory to another location, this routine will do the trick.

The address of the first byte of the source area must be poked into addresses 251 and 252, in the usual low byte, high byte fashion. Similarly, the address of the first byte of the destination area must be poked into addresses 253 and 254 in the same way, before the routine is called by SYS. The routine checks for a terminator character decided before the move, which must be poked into the sixth byte of the routine at location 49980.

Let's see how the routine works using an example. We wish to transfer a block of bytes from a particular address to somewhere else. The block of bytes is terminated by a full stop. The routine will take the first byte from the source area, check that it is not the terminator character and place it at the first byte of the destination area. The second byte is treated in the same way and so on for all bytes in the block or until either the terminator is found or 255 bytes have been copied.

To start, make sure your source and destination addresses are safely poked into correct places and SYS the start address.

Memory Block Move Assembly

```
0  C337  A000      LDY  #$00
1  C339  B1FB      LDA  ($FB),Y
2  C33B  C9FF      CMP  #$FF
3  C33D  F005      BEQ  $C344
4  C33F  91FD      STA  ($FD),Y
5  C341  C8        INY
6  C342  D0F5      BNE  $C339
7  C344  60        RTS
```

Memory Block Move Loader

```
10 REM --------BLOCK MOVE LOADER-----
20 FOR X=49975 TO 49988
30 READ A:POKEX,A
40 NEXT
50 REM --------BLOCK MOVE DATA-------
60 DATA 160,0,177,251,201,255,240,5,145,253
70 DATA 200,208,245,96
```

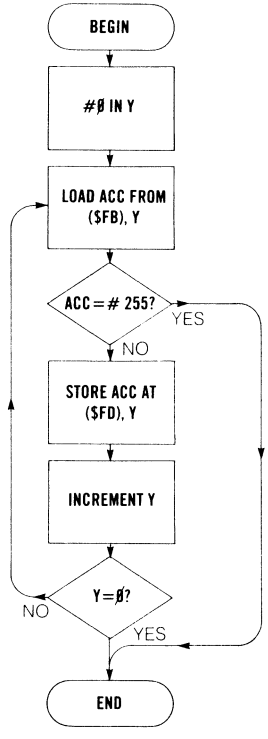


Figure 7.2 'Memory Block Move' flowchart

LINE RENUMBERER

START	49990
END	50080
LENGTH	91 BYTES
REGISTERS USED	A, X, Y

One of the many BASIC commands which does not exist on the Commodore 64 is RENUMBER. When we need to insert another line between 80 and 81, there is only one thing to do: manually resequence the line numbers to leave enough room between existing line numbers.

On a program of any length or complexity, this process can be quite a chore, so this routine can save the day to an extent. The routine will renumber all the line numbers of a BASIC program which is resident in memory but it will *not* affect GOTO or GOSUB commands. As long as we bear this limitation in mind the routine will be immensely useful, especially as we are able to set the value of the first line number and the increment value between successive lines.

The statement for using the routine is as follows:

```
SYS 49990,FL,I
```

where FL is the number of the **F**irst **L**ine and I is the **I**ncrement.

After the usual SYS command with the address of the start of the routine comes a comma, which is checked for by a KERNAL routine. If the comma is absent, a 'Syntax Error' message is produced.

Next comes FL, a number in the range 0 to 63999, the highest BASIC line number the Commodore 64 will allow. This number is the first line number of the *new* sequence. Since the Commodore will not allow line numbers greater than 63999, be sure that the new sequence neither starts above this figure nor strays over it in the course of the renumbering process.

Another comma is then expected, followed by `I`, a number in the range 1 to 255. If a number outside this range is typed in, the routine jumps to another KERNAL routine to produce an 'Illegal Quantity' error message. This number will be the increment between lines.

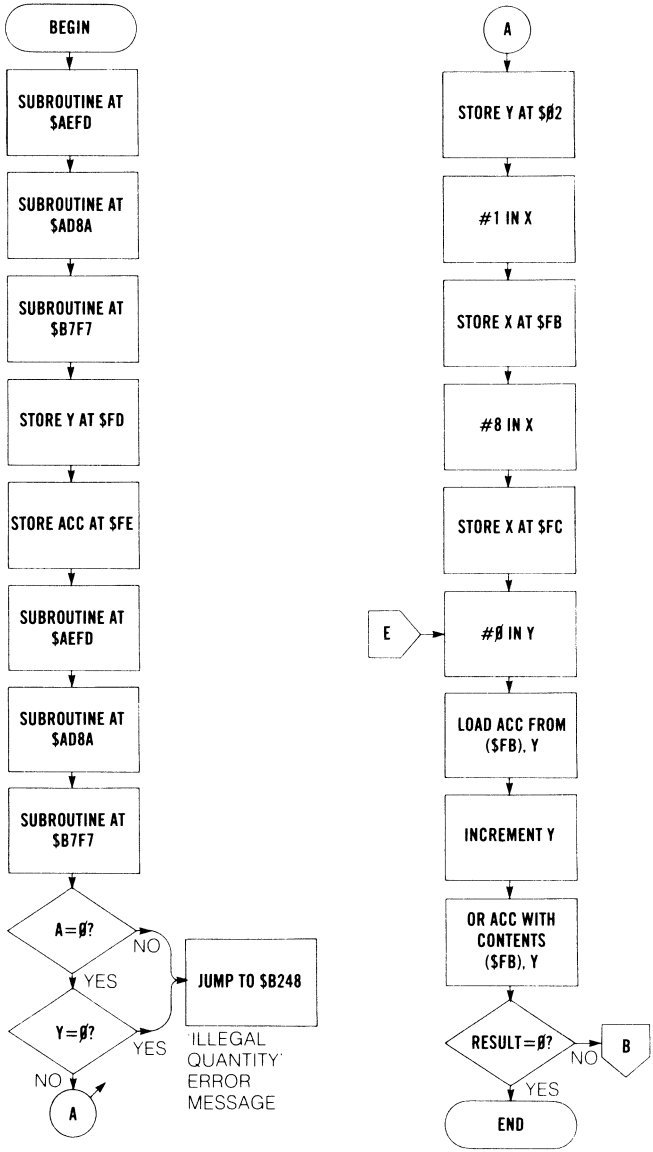
We may also use a variable in place of either or both of the two numbers following the SYS, so if we type in:

```
FL = 1000 : I = 5 : SYS 49990,FL,I
```

the routine will evaluate the variables FL and I and insert them in the correct places to allow the program to be renumbered as if values had been used in the SYS statement.

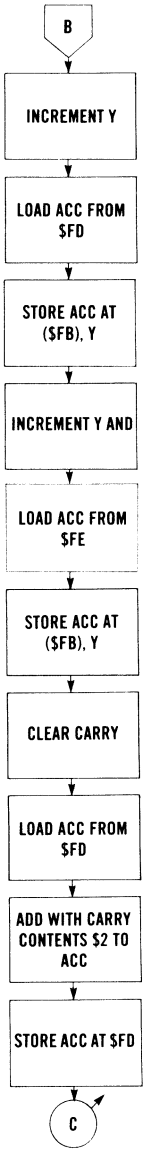
Line Renumberer Assembly

```
 0  C346  20FDAE    JSR  $AEFD
 1  C349  208AAD    JSR  $AD8A
 2  C34C  20F7B7    JSR  $B7F7
 3  C34F  84FD      STY  $FD
 4  C351  85FE      STA  $FE
 5  C353  20FDAE    JSR  $AEFD
 6  C356  208AAD    JSR  $AD8A
 7  C359  20F7B7    JSR  $B7F7
 8  C35C  C900      CMP  #$00
 9  C35E  D03D      BNE  $C39D
10  C360  C000      CPY  #$00
11  C362  F039      BEQ  $C39D
12  C364  8402      STY  $02
13  C366  A201      LDX  #$01
14  C368  86FB      STX  $FB
15  C36A  A208      LDX  #$08
16  C36C  86FC      STX  $FC
17  C36E  A000      LDY  #$00
18  C370  B1FB      LDA  ($FB),Y
19  C372  C8        INY
20  C373  11FB      ORA  ($FB),Y
21  C375  F029      BEQ  $C3A0
22  C377  C8        INY
23  C378  A5FD      LDA  $FD
24  C37A  91FB      STA  ($FB),Y
25  C37C  C8        INY
26  C37D  A5FE      LDA  $FE
27  C37F  91FB      STA  ($FB),Y
28  C381  18        CLC
29  C382  A5FD      LDA  $FD
30  C384  6502      ADC  $02
31  C386  85FD      STA  $FD
32  C388  9002      BCC  $C38C
33  C38A  E6FE      INC  $FE
34  C38C  88        DEY
35  C38D  88        DEY
36  C38E  B1FB      LDA  ($FB),Y
37  C390  48        PHA
38  C391  88        DEY
39  C392  B1FB      LDA  ($FB),Y
40  C394  85FB      STA  $FB
41  C396  68        PLA
42  C397  85FC      STA  $FC
43  C399  A000      LDY  #$00
44  C39B  F0D1      BEQ  $C36E
45  C39D  4C48B2    JMP  $B248
46  C3A0  60        RTS
```

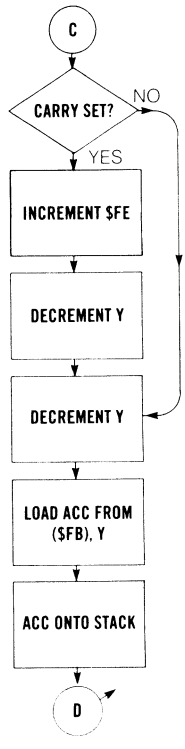


7.3(a)

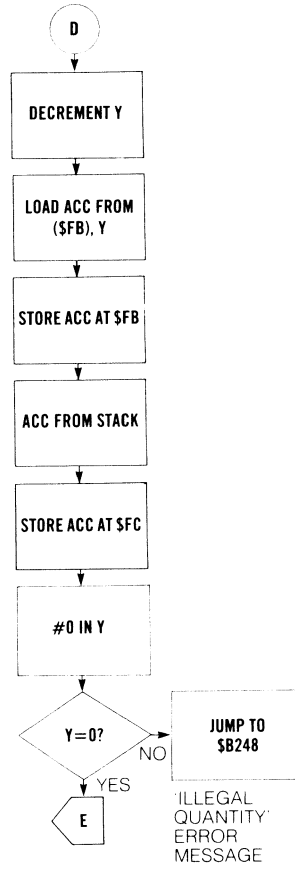
Figure 7.3 Line Renumberer' flowchart


B
INCREMENT Y
LOAD ACC FROM $FD
STORE ACC AT ($FB), Y
INCREMENT Y AND
LOAD ACC FROM $FE
STORE ACC AT ($FB), Y
CLEAR CARRY
LOAD ACC FROM $FD
ADD WITH CARRY CONTENTS $2 TO ACC
STORE ACC AT $FD
C

7.3(c)


C
CARRY SET?
NO
YES
INCREMENT $FE
DECREMENT Y
DECREMENT Y
LOAD ACC FROM ($FB), Y
ACC ONTO STACK
D

7.3(d)


D
DECREMENT Y
LOAD ACC FROM ($FB), Y
STORE ACC AT $FB
ACC FROM STACK
STORE ACC AT $FC
#0 IN Y
Y=0?
NO
JUMP TO $B248
YES
E
'ILLEGAL QUANTITY' ERROR MESSAGE

7.3(e)

Line Renumberer Loader

```
10 REM-------RENUMBER LOADER-----------
20 FOR X=49990 TO 50080
30 READ A:POKE X,A
40 NEXT
50 REM---------RENUMBER DATA------------
60 DATA 32,253,174,32,138,173,32,247,183,132,253,133
70 DATA 254,32,253,174,32,138,173,32,247,183,201,0,208
80 DATA 61,192,0,240,57,132,2,162,1,134,251,162,8,134
90 DATA 252,160,0,177,251,200,17,251,240,41,200,165
100 DATA 253,145,251,200,165,254,145,251,24,165,253
110 DATA 101,2,133,253,144,2,230,254,136,136,177,251
120 DATA 72,136,177,251,133,251,104,133,252,160
130 DATA 0,240,209,76,72,178,96
```

ASCII SET DISPLAY

START	50085
END	50100
LENGTH	16 BYTES
REGISTERS USED	A, X, Y

This routine may be used when you have redefined all or some of the characters and wish to see the results of your efforts.

The top quarter of the screen needs to be clear before using the routine as that is where the complete ASCII set is displayed when you SYS the start address.

As this one is so simple, I have not included a flow diagram, so here is your chance to construct one for yourself. Really, the flow diagram should exist *before* the program, but as an exercise I'll allow you to work backwards and build your flowchart from my completed program. See how you get on!

ASCII Set Display Assembly

```
0  C3A5  A200      LDX  #$00
1  C3A7  A005      LDY  #$05
2  C3A9  98        TYA
3  C3AA  9D24D8    STA  $D824,X
4  C3AD  8A        TXA
5  C3AE  9D2804    STA  $0428,X
6  C3B1  E8        INX
7  C3B2  D0F5      BNE  $C3A9
8  C3B4  60        RTS
```

ASCII Set Display Loader

```
10 REM --------ASCII SET LOADER-------
20 FOR X=50085 TO 50100
30 READ A:POKE X,A
40 NEXT
50 REM --------ASCII SET DATA--------
60 DATA 162,0,160,5,152,157,36,216
70 DATA 138,157,40,4,232,208,245,96
```

CHARACTER BUBBLE SORT

START	5Ø1Ø5
END	5Ø135
LENGTH	31 BYTES
REGISTERS USED	A, X, Y

A sort program takes a series of unordered strings, numbers or characters, and rearranges them into either ascending or descending alphabetic or numeric order. This sort program is known as a 'bubble' sort, because the 'heavier' elements sink progressively to the bottom of the list and the 'lighter' elements rise to the top.

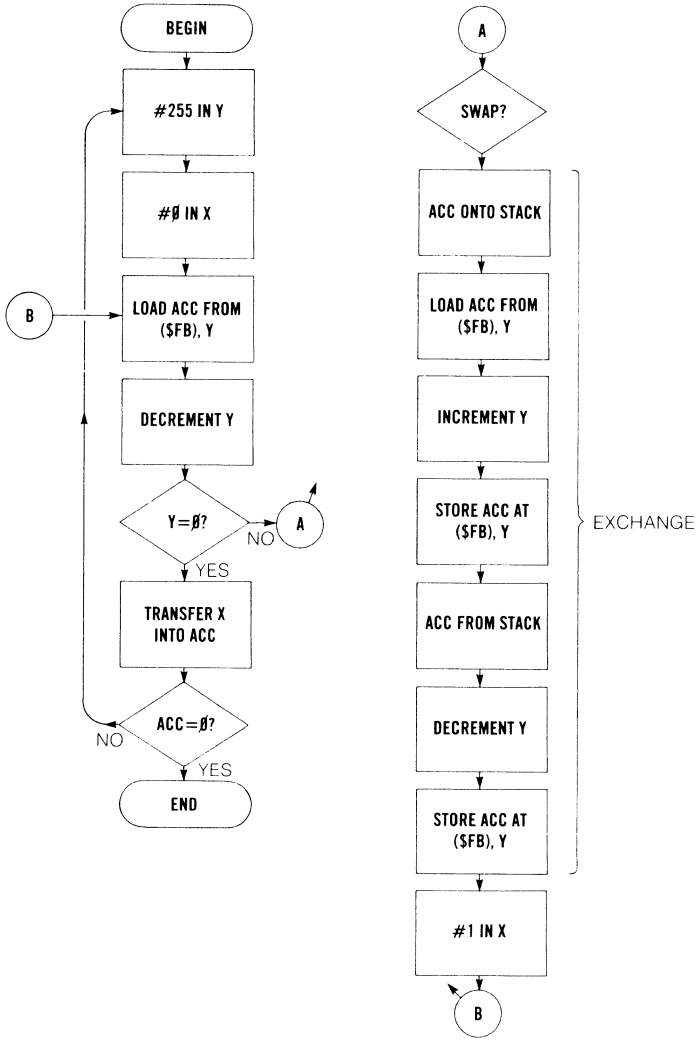
The bubble sort is one of the easiest sort algorithms to understand and implement and, although it is not the fastest as we will see, it has a fair turn of speed when executed in machine language.

To use the routine, we must first take the address of the byte before the first byte of the table of numbers or characters, and poke that address into locations 251 and 252 in conventional low byte, high byte fashion. The number of characters to sort is assumed by the routine to be 255. To sort fewer elements we must alter the second byte of the routine at location 5Ø1Ø6 to the number of elements in the table.

The program works by comparing the last pair of elements in the table and if they are unordered, it swaps them over. Then the counter is decremented and the previous pair are compared in the same way, and so on until every element has been compared with its neighbour. That process constitutes one pass of the program. If no exchanges have been made during that pass then all of the elements are in order and the process ends.

However, if any exchanges have been made, the routine will restart and run through the table again and again, comparing and making any necessary exchanges until it *has* made one full pass with no swaps being performed.

Load the routine using the BASIC loader and then run the tester program to see the routine in action.



7.4(a) **7.4(b)**

Figure 7.4 'Character Bubble Sort' flowchart

Character Bubble Sort Assembly

```
 0  C3B9  A0FF      LDY  #$FF
 1  C3BB  A200      LDX  #$00
 2  C3BD  B1FB      LDA  ($FB),Y
 3  C3BF  88        DEY
 4  C3C0  F012      BEQ  $C3D4
 5  C3C2  D1FB      CMP  ($FB),Y
 6  C3C4  B0F7      BCS  $C3BD
 7  C3C6  48        PHA
 8  C3C7  B1FB      LDA  ($FB),Y
 9  C3C9  C8        INY
10  C3CA  91FB      STA  ($FB),Y
11  C3CC  68        PLA
12  C3CD  88        DEY
13  C3CE  91FB      STA  ($FB),Y
14  C3D0  A201      LDX  #$01
15  C3D2  D0E9      BNE  $C3BD
16  C3D4  8A        TXA
17  C3D5  D0E2      BNE  $C3B9
18  C3D7  60        RTS
```

Character Bubble Sort Loader

```
10 REM -----CHARACTER SORT LOADER-----
20 FOR X=50105 TO 50135
30 READ A:POKE X,A
40 NEXT
50 REM ------CHARACER SORT DATA-------
60 DATA 160,255,162,0,177,251,136,240
70 DATA 18,209,251,176,247,72,177,251
80 DATA 200,145,251,104,136,145,251
90 DATA 162,1,208,233,138,208,226,96
```

Character Bubble Sort Tester

```
5 REM -----CHARACTER SORT TESTER----
10 POKE 251,255
20 POKE 252,3
30 PRINT"□";:REM ---CLEAR SCREEN CHARACTERS
40 FOR X=1 TO 255
50 PRINT CHR$((RND(1)*26)+65);
60 NEXT
70 FOR T=1 TO 1500:NEXT
80 SYS 50105
```

RANDOM NUMBER GENERATOR

START	5Ø14Ø
END	5Ø18Ø
LENGTH	41 BYTES
REGISTERS USED	A, X, Y

No home microcomputer can generate a sequence of truly random numbers, but all have some form of 'random' number function to produce a reasonable simulation. This routine works as well as most of these functions and may be used to produce a random number in the range Ø to X, where X is less than or equal to 255. The call statement to use the routine is as follows:

```
SYS 5Ø14Ø,X
```

where X may be a variable or an integer in the range above.

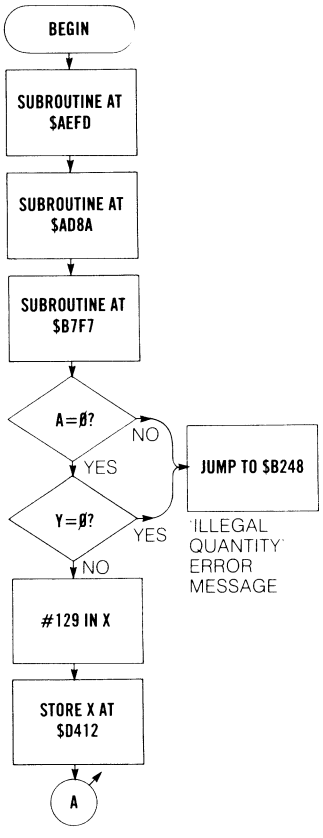
The routine works in a similar way to the 'Line Renumberer' routine earlier in this chapter in that it first looks for a comma after the SYS address and then evaluates the expression following. If a value less than 1 or greater than 255 is returned, then the '**I**llegal **Q**uantity' error message is produced.

If we type in

```
SYS 5Ø14Ø,1ØØ
```

the number produced will be in the range Ø to 1ØØ inclusive. (If the value Ø is not acceptable for your purposes, then you must add 1 to the number produced.) The routine leaves the random number in location 251, so a PEEK to this address will give us our number.

The program works by setting VOICE 3 of the SID chip (see chapter 4) to the white noise waveform and reading the pitch being produced at the instant the routine is called by taking the value in address 54299 and checking that it is less than or equal to the maximum value we have set. If this condition is not true then another number is taken until the number does fall within the range.



7.5(a) 7.5(b)

Figure 7.5 'Random Number Generator' flowchart

Random Number Generator Assembly

```
 0  C3DC  20FDAE    JSR  $AEFD
 1  C3DF  208AAD    JSR  $AD8A
 2  C3E2  20F7B7    JSR  $B7F7
 3  C3E5  C900      CMP  #$00
 4  C3E7  D019      BNE  $C402
 5  C3E9  C000      CPY  #$00
 6  C3EB  F015      BEQ  $C402
 7  C3ED  A281      LDX  #$81
 8  C3EF  8E12D4    STX  $D412
 9  C3F2  A2FF      LDX  #$FF
10  C3F4  8E0FD4    STX  $D40F
11  C3F7  AD1BD4    LDA  $D41B
12  C3FA  85FB      STA  $FB
13  C3FC  18        CLC
14  C3FD  C4FB      CPY  $FB
15  C3FF  90F6      BCC  $C3F7
16  C401  60        RTS
17  C402  4C48B2    JMP  $B248
```

Random Number Generator Loader

```
10 REM ------RANDOM NUMBER LOADER------
20 FOR X=50140 TO 50180
30 READ A:POKE X,A
40 NEXT
50 REM --------RANDOM NUMBER DATA---------
60 DATA 32,253,174,32,138,173,32,247,183
70 DATA 201,0,208,25,192,0,240,21,162,129
80 DATA 142,18,212,162,255,142,15,212,173
90 DATA 27,212,133,251,24,196,251,144,246
100 DATA 96,76,72,178
```

SIXTEEN-BIT SUM

START	50185
END	50210
LENGTH	26 BYTES
REGISTERS USED	A, Y

This routine will compute the sum of up to 255 numbers in the usual eight-bit range (0 to 255) and store the total as a two-byte number in low byte, high byte format in locations 251 and 252. The address of the start of the table of numbers must first be poked into locations 253 and 254, and the first byte of the table must contain the number of elements in the table.

When the routine has been used, the two numbers in 251 and 252 may be retrieved as a 16-bit number (0 to 65535) by typing:

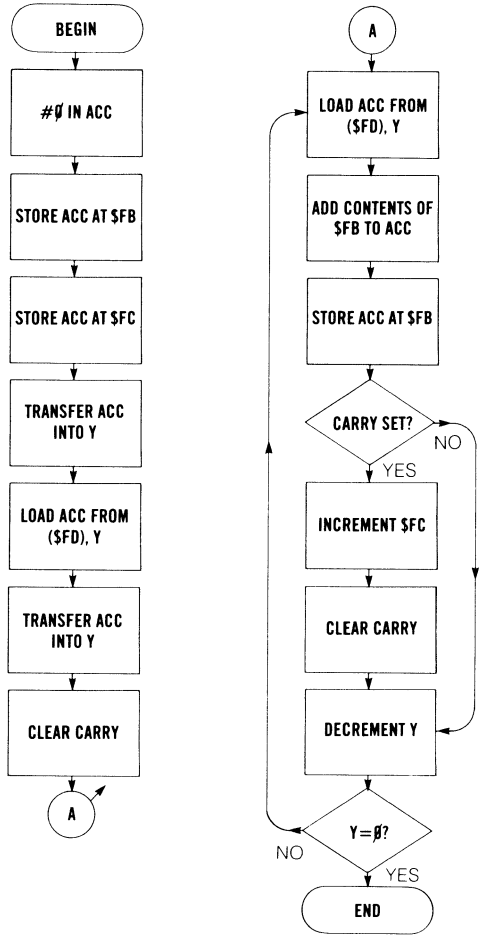
```
PRINT(PEEK(252)*256)+PEEK(251)
```

Sixteen-Bit Sum Assembly

```
 0  C409  A900      LDA  #$00
 1  C40B  85FB      STA  $FB
 2  C40D  85FC      STA  $FC
 3  C40F  A8        TAY
 4  C410  B1FD      LDA  ($FD),Y
 5  C412  A8        TAY
 6  C413  18        CLC
 7  C414  B1FD      LDA  ($FD),Y
 8  C416  65FB      ADC  $FB
 9  C418  85FB      STA  $FB
10  C41A  9003      BCC  $C41F
11  C41C  E6FC      INC  $FC
12  C41E  18        CLC
13  C41F  88        DEY
14  C420  D0F2      BNE  $C414
15  C422  60        RTS
```

Sixteen-Bit Sum Loader

```
10 REM--------16 BIT SUM LOADER---------
20 FOR X=50185 TO 50210
30 READ A:POKE X,A
40 NEXT
50 REM ----------16 BIT SUM DATA---------
60 DATA 169,0,133,251,133,252,168,177,253
70 DATA 168,24,177,253,101,251,133,251
80 DATA 144,3,230,252,24,136,208,242,96
```



7.6(a) **7.6(b)**

Figure 7.6 'Sixteen-Bit Sum' flowchart

*** APPENDICES ***

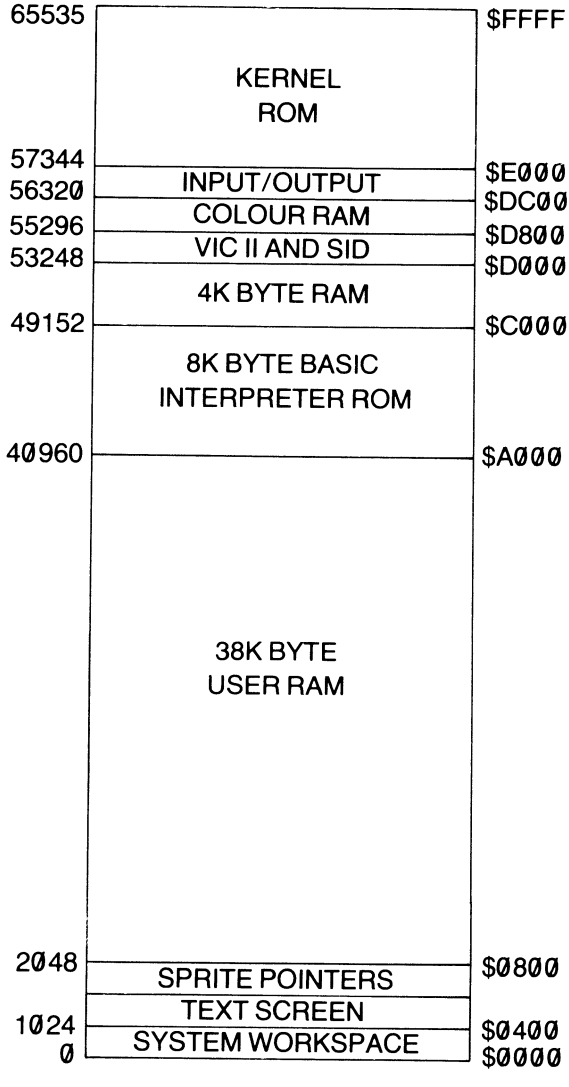
A Number-Base Conversions

0	00000000	0
1	00000001	1
2	00000010	2
3	00000011	3
4	00000100	4
5	00000101	5
6	00000110	6
7	00000111	7
8	00001000	8
9	00001001	9
10	00001010	A
11	00001011	B
12	00001100	C
13	00001101	D
14	00001110	E
15	00001111	F
16	00010000	10
17	00010001	11
18	00010010	12
19	00010011	13
20	00010100	14
21	00010101	15
22	00010110	16
23	00010111	17
24	00011000	18
25	00011001	19
26	00011010	1A
27	00011011	1B
28	00011100	1C
29	00011101	1D
30	00011110	1E
31	00011111	1F
32	00100000	20
33	00100001	21
34	00100010	22
35	00100011	23
36	00100100	24
37	00100101	25
38	00100110	26
39	00100111	27
40	00101000	28
41	00101001	29
42	00101010	2A
43	00101011	2B
44	00101100	2C
45	00101101	2D
46	00101110	2E
47	00101111	2F
48	00110000	30
49	00110001	31
50	00110010	32
51	00110011	33
52	00110100	34
53	00110101	35
54	00110110	36
55	00110111	37
56	00111000	38
57	00111001	39
58	00111010	3A
59	00111011	3B
60	00111100	3C
61	00111101	3D
62	00111110	3E
63	00111111	3F
64	01000000	40
65	01000001	41
66	01000010	42
67	01000011	43
68	01000100	44
69	01000101	45
70	01000110	46
71	01000111	47
72	01001000	48
73	01001001	49
74	01001010	4A
75	01001011	4B
76	01001100	4C
77	01001101	4D
78	01001110	4E
79	01001111	4F
80	01010000	50
81	01010001	51
82	01010010	52
83	01010011	53
84	01010100	54
85	01010101	55

86	01010110	56
87	01010111	57
88	01011000	58
89	01011001	59
90	01011010	5A
91	01011011	5B
92	01011100	5C
93	01011101	5D
94	01011110	5E
95	01011111	5F
96	01100000	60
97	01100001	61
98	01100010	62
99	01100011	63
100	01100100	64
101	01100101	65
102	01100110	66
103	01100111	67
104	01101000	68
105	01101001	69
106	01101010	6A
107	01101011	6B
108	01101100	6C
109	01101101	6D
110	01101110	6E
111	01101111	6F
112	01110000	70
113	01110001	71
114	01110010	72
115	01110011	73
116	01110100	74
117	01110101	75
118	01110110	76
119	01110111	77
120	01111000	78
121	01111001	79
122	01111010	7A
123	01111011	7B
124	01111100	7C
125	01111101	7D
126	01111110	7E
127	01111111	7F
128	10000000	80
129	10000001	81
130	10000010	82
131	10000011	83
132	10000100	84
133	10000101	85
134	10000110	86
135	10000111	87
136	10001000	88
137	10001001	89
138	10001010	8A
139	10001011	8B
140	10001100	8C
141	10001101	8D
142	10001110	8E
143	10001111	8F
144	10010000	90
145	10010001	91
146	10010010	92
147	10010011	93
148	10010100	94
149	10010101	95
150	10010110	96
151	10010111	97
152	10011000	98
153	10011001	99
154	10011010	9A
155	10011011	9B
156	10011100	9C
157	10011101	9D
158	10011110	9E
159	10011111	9F
160	10100000	A0
161	10100001	A1
162	10100010	A2
163	10100011	A3
164	10100100	A4
165	10100101	A5
166	10100110	A6
167	10100111	A7
168	10101000	A8
169	10101001	A9
170	10101010	AA
171	10101011	AB
172	10101100	AC
173	10101101	AD
174	10101110	AE
175	10101111	AF
176	10110000	B0
177	10110001	B1
178	10110010	B2
179	10110011	B3
180	10110100	B4
181	10110101	B5
182	10110110	B6
183	10110111	B7
184	10111000	B8
185	10111001	B9
186	10111010	BA
187	10111011	BB
188	10111100	BC
189	10111101	BD
190	10111110	BE
191	10111111	BF
192	11000000	C0
193	11000001	C1
194	11000010	C2
195	11000011	C3
196	11000100	C4
197	11000101	C5
198	11000110	C6
199	11000111	C7
200	11001000	C8
201	11001001	C9

202	11001010	CA
203	11001011	CB
204	11001100	CC
205	11001101	CD
206	11001110	CE
207	11001111	CF
208	11010000	D0
209	11010001	D1
210	11010010	D2
211	11010011	D3
212	11010100	D4
213	11010101	D5
214	11010110	D6
215	11010111	D7
216	11011000	D8
217	11011001	D9
218	11011010	DA
219	11011011	DB
220	11011100	DC
221	11011101	DD
222	11011110	DE
223	11011111	DF
224	11100000	E0
225	11100001	E1
226	11100010	E2
227	11100011	E3
228	11100100	E4
229	11100101	E5
230	11100110	E6
231	11100111	E7
232	11101000	E8
233	11101001	E9
234	11101010	EA
235	11101011	EB
236	11101100	EC
237	11101101	ED
238	11101110	EE
239	11101111	EF
240	11110000	F0
241	11110001	F1
242	11110010	F2
243	11110011	F3
244	11110100	F4
245	11110101	F5
246	11110110	F6
247	11110111	F7
248	11111000	F8
249	11111001	F9
250	11111010	FA
251	11111011	FB
252	11111100	FC
253	11111101	FD
254	11111110	FE
255	11111111	FF

B Memory Map


65535
$FFFF
KERNEL
ROM
57344
$E000
56320
INPUT/OUTPUT
$DC00
55296
COLOUR RAM
$D800
53248
VIC II AND SID
$D000
4K BYTE RAM
49152
$C000
8K BYTE BASIC
INTERPRETER ROM
40960
$A000
38K BYTE
USER RAM
2048
$0800
SPRITE POINTERS
TEXT SCREEN
1024
$0400
0
SYSTEM WORKSPACE
$0000

C Useful Addresses

The following addresses will be found useful when using the machine language programs in this book and when writing your own BASIC and machine language programs.

43-44	Pointer to start of BASIC program
45-46	Pointer to start of BASIC variables
47-48	Pointer to start of BASIC arrays
49-50	Pointer to end of arrays + 1
55-56	Pointer to highest address used by BASIC
160-162	Real time Jiffy clock
197	Current key pressed
198	Number of keys in keyboard buffer
251-254	Unused — for user applications
631-640	Keyboard buffer
649	POKE 649,0 disables the keyboard
	POKE 649,10 restores the keyboard to normal
650	POKE 650,128 gives all keys auto repeat

D Useful ROM Routines

When you begin writing your own machine language and BASIC programs these routines, which are all programmed into the ROM of the Commodore 64, should prove quite handy.

Some subroutines will return control to the place in the program whence they came but some will jump back into BASIC command mode after producing an error message. The best thing to do is to experiment and see how the routines might be useful to *you*. If you are working on a program where every byte counts, the ability to produce error messages, for example, without the loss of any potentially useful memory for PRINT statements could be a great bonus.

But try them all out anyway. It's quite fun to see the computer fooled into thinking certain things are going on that actually aren't!

42037	**$A435**	'Out of Memory' message
42100	**$A474**	'Ready' message
44791	**$AEF7**	Check for right bracket ')'
44794	**$AEFA**	Check for left bracket '('
44797	**$AEFD**	Check for comma ','
44808	**$AF08**	'Syntax Error' message
45637	**$B245**	'Bad Subscript' message
45640	**$B248**	'Illegal Quantity' message
47486	**$B97E**	'Overflow' message
59626	**$E8EA**	Scroll screen 1 line up
61107	**$EEB3**	1 ms delay
62895	**$F5AF**	'Searching' message
62930	**$F5D2**	'Loading/Verifying' message
63511	**$F817**	'Press Play . . .' message
63544	**$F838**	'Press Record . . .' message
65126	**$FE66**	Warm start
64738	**$FCE2**	Cold start

E 6510 Instruction Set

INTRODUCTION APPENDIX E

One way to speed up your familiarity with the world of machine language is to study routines and programs written by other people. The routines in this book all work and, thanks to the very nature of machine language, all display a dazzling turn of speed when compared with an equivalent program written in BASIC.

Please examine these routines, see how they have been put together, and how memory and the registers within the CPU have been juggled to achieve a result. But do not assume that these routines are definitive; almost every one could be made to run faster or by using less memory — possibly at the expense of legibility. If you have the urge to try to improve upon them as an exercise in programming, go ahead. Practice makes perfect!

This table has been included to enable you to examine each instruction in turn in any of the routines in this book, and to help you fathom out exactly how the routine works, and what is happening in the CPU and the memory during execution.

ADC	Add with copy
AND	Logical AND
ASL	Arithmetic shift left
BCC	Branch if carry clear
BCS	Branch if carry set
BEQ	Branch if equal to Ø
BIT	Test bit
BMI	Branch if minus
BNE	Branch if not equal
BPL	Branch if plus
BRK	Break
BVC	Branch if V bit clear
BVS	Branch if V bit set
CLC	Clear carry
CLD	Clear decimal
CLI	Clear interrupt mask
CLV	Clear overflow bit
CMP	Compare to memory
CPX	Compare to X
CPY	Compare to Y
DEC	Decrement memory
DEX	Decrement X
DEY	Decrement Y
EOR	Exclusive OR
INC	Increment memory
INX	Increment X
INY	Increment Y
JMP	Jump to address
JSR	Jump to subroutine
LDA	Load accumulator
LDX	Load X
LDY	Load Y
LSR	Logical shift right
NOP	No operation
ORA	Logical OR
PHA	Push A to stack
PHP	Push P status to stack
PLA	Pull A from stack
PLP	Pull P status from stack
ROL	Rotate left
ROR	Rotate right
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract with carry
SEC	Set carry
SED	Set decimal
SEI	Set interrupt mask
STA	Store accumulator
STX	Store X
STY	Store Y
TAX	Transfer A into X
TAY	Transfer A into Y
TSX	Transfer SP into X
TXA	Transfer X into A
TXS	Transfer X into SP
TYA	Transfer Y into A

F Assemblers

THE MIKRO ASSEMBLER

All of the routines in this book were written with the aid of an excellent piece of firmware, a cartridge distributed by Supersoft of Harrow which contains a three-pass assembler in ROM.

Very short machine language programs may be written and assembled by hand, using a reference chart to find the hex numbers for each instruction and a BASIC loader like those in this book to position the routine in memory (after the hex code has been converted to decimal for the loader!). However, any program containing more than five or six instructions would be developed far more easily and quickly with the use of such an assembler.

There are several assemblers on the market for the Commodore 64, some supplied on cassette or disc and some in cartridge form. The obvious advantage of the cartridge over the other media is that no loading time is required, just power up and away.

During the development of these programs, the MIKRO cartridge was permanently resident in the expansion slot at the rear of my Commodore 64, as it brings a number of handy extensions to the rather poor version of BASIC on the machine. These are: AUTO, for the automatic line numbering of long programs; DELETE, used to delete sections of BASIC programs; FORMAT, which lines up the machine language source program in regular columns on either the screen or printer, with line numbers on the left, followed by labels, instructions operands and comments. Another useful command is NUMBER which, when followed by any number in the range Ø to 65535, will print out that number in decimal, hexadecimal, binary and octal (base 8) formats.

One or two other commands are built in which enable a parallel printer to be attached to the Commodore 64 and used without further printer software.

All in all, apart from the obvious job of producing error-free object code from our source programs, the MIKRO cartridge is a valuable asset.

The assembler allows us to type in our machine language program just as if it were written in BASIC, with numbered lines which are stored internally by the machine in exactly the same way as BASIC programs are. When the program is complete, we just have to type in ASSEMBLE and the program goes to work putting the final instructions and their operands in their proper places in the area of memory specified by the first line of the program.

One of the chief complaints of the buying public against the producers of software for the home computer market is that often the documentation supplied with a program does not do justice to the product itself. Supersoft have produced an excellent aid to the machine language programmer in their MIKRO cartridge but the documentation is of a rather lower standard.

To complement the use of the cartridge you will need a good book on 65Ø2 and 651Ø assembly language and be prepared for a modicum of trial and error, as the instruction manual is not as good as I would like to see with such a fine piece of firmware. In this case, the quality and usefulness of the program outweigh the poor manual so, if the price tag is no problem, then buy with confidence. Happy programming!

The MIKRO Assembler is available in the United States and Canada from

Skyles Electric Works
231e S Whisman Road
Mountain View
CA 94041
U.S.A.

and in Great Britain and the rest of the world from

SUPERSOFT
Winchester House
Canning Road
Wealdstone
Harrow
HA3 7SJ
England

For more information on the MIKRO assembler than the supplied manual provides, refer to *Introducing Commodore 64 Machine Code* by Ian Sinclair (Granada) which contains a chapter on MIKRO.

THE ALPA MONITOR

A listing for a monitor/assembler called ALPA (**A**ssembly **L**anguage **P**rogramming **A**id) is given in *Commodore 64 Machine Language for the Absolute Beginner* (Melbourne House, 1984). While not as fast or memory-efficient as the MIKRO assembler, ALPA is certainly a lot cheaper and will enable you to experience machine language programming before buying a more advanced assembler program.

G Further Reading

I hope the examples of machine language programming in this book have shown you what can be achieved by directly harnessing the speed and power of the microprocesor at the heart of your Commodore 64. The next step is to start writing your own machine language routines. Study the examples in this book and improve upon them. No program is ever finished — there will always be room for improvement.

Commodore 64 Machine Language for the Absolute Beginner (Melbourne House)

Commodore 64 Exposed (Melbourne House)

Commodore 64 Programmer's Reference Guide (Commodore Business Machines)

This book has chapters on machine language programming and contains lists of user-callable routines in the ROM of the computer.

Programming the 6502 (Sybex)

This book covers not only the specific subjects of the 65Ø2 and 651Ø but also has chapters on number theory and the more common data structures.

Using the Commodore 64 (Duckworth)

Introducing Commodore 64 Machine Code (Granada)

Personal Computer World (VNU)

This magazine has a monthly column called 'Subset' devoted to machine language for most popular processors, including the 65Ø2 and 651Ø.

Write to Us

Melbourne House is always interested in receiving letters from its readers.

Publishing Ideas

If you have written a book or program that you think would be of interest to other computer users, we want to hear from you.

We are always interested in discussing new ideas for books with authors. If you think you have a good book idea, please send a detailed outline first. We prefer to work with authors as early as possible in the writing process.

BASIC programs are wanted for inclusion in our books, and machine language programs are wanted for our list of adventure and game software. Always send a tape or disk and, if possible, a code printout with your submission letter.

Fees and royalties are negotiated according to the quality and ingenuity of the submission, and are more than competitive with those of other publishing houses.

Send your book or program to the Melbourne House office closest to you — see the back of the title page for the address. Mark your letter to the attention of the Editorial Department to ensure an early review of your idea and a prompt reply.

Bugs and Problems

Every effort is made to ensure that our books are error-free. Occasionally, however, you may have difficulties — in such instances do not hesitate to write to Melbourne House. Send your letter to the Melbourne House office closest to you — see the back of the title page for the address.

So that we can process your query as quickly as possible, mark your letter to the attention of Customer Support. Quote the title of this book in your letter, together with the printing and edition numbers, and the year of publication. This information is on the back of the title page at the foot.

Describe your problem precisely, quoting the program title and the offending line numbers.

Index

T

U

V

W

X

Y

Z

Supercharge Your Commodore 64

Customer Registration Card

Please fill out this page (or a photocopy of it) and return it so that we may keep you informed of new books, software and special offers. Post to the appropriate address on the back.

Date .....19...

Name .....

Street & No. .....

City .....Postcode/Zipcode .....

Model of computer owned .....

Where did you learn of this book:

☐ FRIEND ☐ RETAIL SHOP

☐ MAGAZINE (give name) .....

☐ OTHER (specify) .....

Age? ☐ 10-15 ☐ 16-19 ☐ 20-24 ☐ 25 and over

How would you rate this book?

QUALITY: ☐ Excellent ☐ Good ☐ Poor

VALUE: ☐ Overpriced ☐ Good ☐ Underpriced

What other books and software would you like to see produced for your computer?

.....

.....

.....

EDITION 7 6 5 4 3 2 1

Melbourne House addresses

Put this Registration Card (or photocopy) in an envelope and post it to the appropriate address:

United Kingdom

Melbourne House (Publishers) Ltd
Castle Yard House
Castle Yard
Richmond, TW10 6TF

United States of America

Melbourne House Software Inc.
347 Reedwood Drive
Nashville TN 37217

Australia and New Zealand

Melbourne House (Australia) Pty Ltd
Level 2, 70 Park Street
South Melbourne, Victoria 3205

C64

Melbourne House

Give your Commodore 64 programs the power of machine language without actually having to learn machine language.

Now, without any additional effort, you can overcome the limitations of BASIC. This book will help you to develop programs of professional quality: not only will your programs look better by using more powerful graphic commands, run faster and be more spectacular — with realistic explosions and other great sound effect — but you will also be able to develop them in a fraction of the time you would expect to take.

The routines in this book allow you to enlarge sprite-definition memory, manipulate sprites, produce high-resolution, high-speed screens; manipulate screen colours; move memory blocks; renumber BASIC programs and much more. The large number of explanatory diagrams and flowcharts make the task even easier.

Whether you are a beginner Commodore 64 user or an experienced programmer, Supercharge Your Commodore 64 is a book you cannot be without.

Melbourne House Publishers

ISBN 0-86161-174-8

9 780861 611744