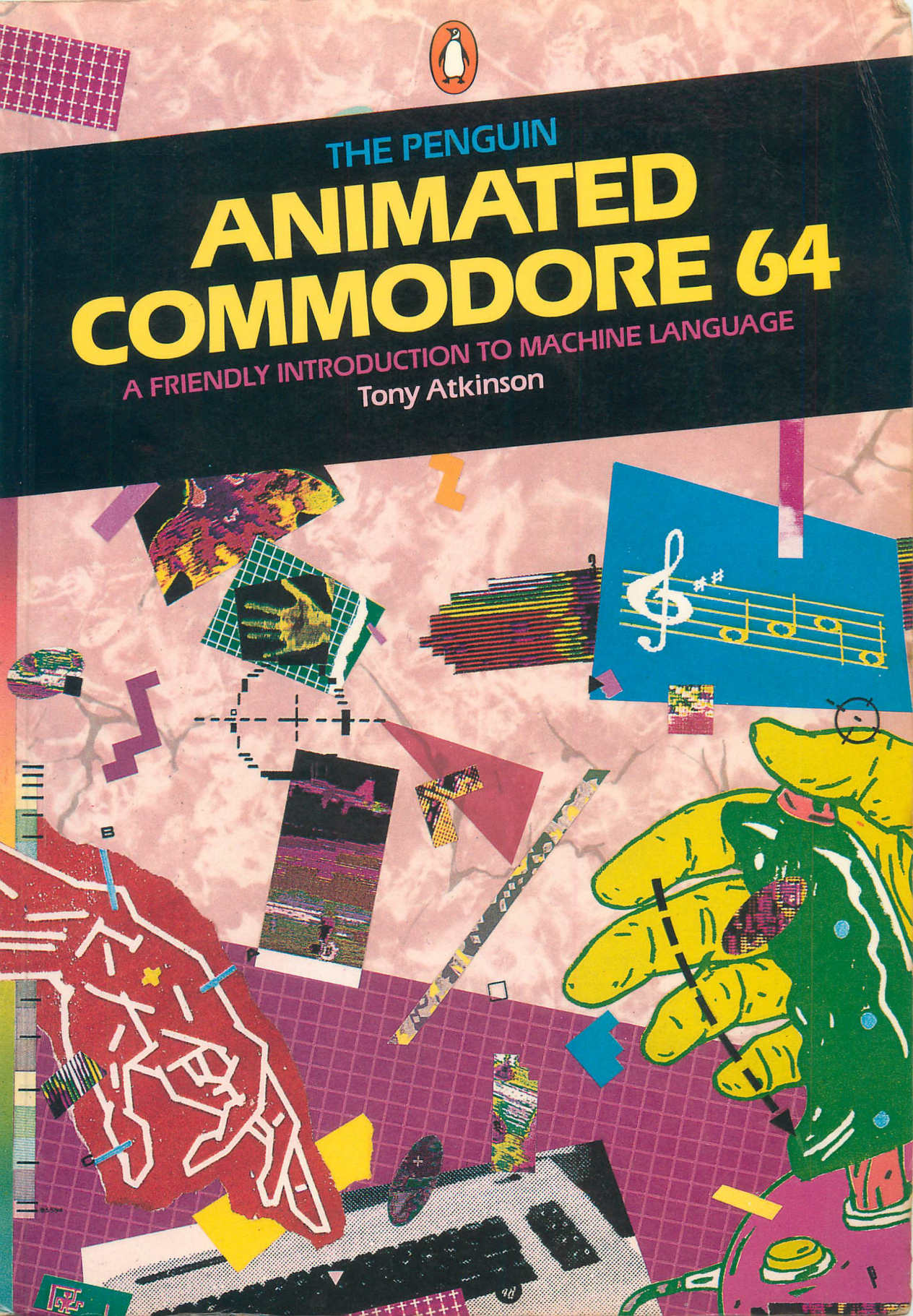

THE PENGUIN
ANIMATED
COMMODORE 64
A FRIENDLY INTRODUCTION TO MACHINE LANGUAGE
Tony Atkinson

Penguin Books

The Animated Commodore 64 –
A Friendly Introduction to Machine Language

Tony Atkinson was born in Adelaide in 1926 and has been associated with computers for many years, mainly in the fields of systems design and marketing. He has held managerial positions with the Ford Motor Company and Volkswagen (Australia), as well as being a management consultant with John P. Young. He received a Bachelor of Commerce degree from the University of Melbourne and has recently been involved in the retail selling of home computers.

The Animated Commodore 64 – A Friendly Introduction to Machine Language

Tony Atkinson

Penguin Books

Penguin Books Australia Ltd,
487 Maroondah Highway, P.O. Box 257
Ringwood, Victoria, 3134, Australia
Penguin Books Ltd,
Harmondsworth, Middlesex, England
Penguin Books,
40 West 23rd Street, New York, N.Y. 10010, U.S.A.
Penguin Books Canada Ltd,
2801 John Street, Markham, Ontario, Canada
Penguin Books (N.Z.) Ltd,
182–190 Wairau Road, Auckland 10, New Zealand

Produced for Penguin Books Australia Ltd
by Lucas Comber McGrath Pty Ltd, Glenhuntly, Victoria

First published by Penguin Books Australia, 1984

Typeset in Times Roman by H&M Typesetting and Graphics,
St Kilda, Victoria

Made and printed in Australia
by Globe Press Pty Ltd, Melbourne

CIP

Atkinson, Tony, 1926–.
The animated Commodore 64.

ISBN 0 14 007894 0.

1. Commodore 64 (Computer) – Programming.
2. Basic (Computer program language).
I. Title.

001.64'24

Contents

Appendices

Acknowledgements

The tables of screen codes that appear from time to time throughout the book have been reproduced from the **Commodore 64 Programmers Reference Guide** by kind permission of Commodore Business Machines Pty Ltd, Australia.

Introduction

WHY LEARN MACHINE LANGUAGE?

A program in machine language uses less of your computer's memory than a BASIC program that does the same job. More importantly, a machine language program can be executed at incredible speed. Games programs in machine language are therefore far more exciting than the same programs written in BASIC. The realistic animation and instant displays that you can produce on the screen via machine language are either difficult or impossible via BASIC. A machine language programmer can, for example, create laser beams that look and sound like laser beams.

So let's get straight to it. Switch on your C-64 and type in Program Number 1. It contains, in fact, two programs which do exactly the same thing in almost exactly the same way. One is written in BASIC and the other in machine language.

Program # 1

```
1 REM**BASIC PROGRAM**
5 PRINT CHR$(147)
10 N=1024
20 FOR M=1 TO 7: FOR I=N TO N+15
30 POKE I,102: POKE I+54272,0
40 NEXT I: N=N+40: NEXT M
45 REM**MACHINE LANGUAGE PROGRAM WITH**
        **BASIC LOADING PROGRAM**
50 FOR N=0 TO 26: READ D: POKE 828+N,D:
   NEXT
```

```
60 DATA 169,0,160,16,72,170,169,102,157,
    232,6,169,0,157,232,218,232
70 DATA 136,208,242,104,24,105,40,144,
    232,96
80 SYS 828
```

RUN the program. It will take a moment to READ the DATA lines and then you will see a block drawn in the top left-hand corner. As this finishes a similar block will appear instantly in the lower right-hand corner.

Well, that in itself isn't very exciting but think for a moment what this sort of action could do for a game! The left-hand block is drawn in BASIC, the other in machine language. What an enormous difference in speed there is. Furthermore, the BASIC program to draw the block used 100 bytes while the machine language routine took only 29. Now type NEW and the BASIC program is cleared. But type SYS 828 and immediately on will come the right-hand block. The 29 bytes are safely tucked away in the cassette file buffer and the whole BASIC RAM is free for other use. We will take a closer look at this machine language program in Chapter 4.

WHAT IS MACHINE LANGUAGE?

Machine language for the C-64 is the group of instructions which the designers of the 6510 microprocessor have built into the chip itself. These are the instructions represented by the little groups of 0's and 1's of binary numbers which are the only things that the 6510 can understand and act upon.

Fortunately the designers of the computer have arranged the machine so that you can communicate with it directly, via the keyboard and the display screen, in decimal numbers. The 6510, of course, gets this all in binary which is what it really understands.

Machine language is made up of decimal number codes. It therefore looks quite different from BASIC, which is really very like plain English.

Putting in the decimal codes, however, is only the last step. We can (and do) write out our machine language programs in plain language. Once you have become familiar with these instructions you may even find you can talk your way through a program more easily than you can with BASIC. Actually,

microprocessors usually have quite small lists of instructions which have very precise meanings.

All the machine language instructions in this book are set out in plain language. It is important to understand what each instruction means in full and to think of each of them in this way.

You'll also need to learn some mnemonics. Now don't panic! I know that trying to figure out mnemonics in some books and magazines, can be difficult for the beginner. But mnemonics are nothing more than abbreviations of the machine language instructions. They are simply aids to memory. For example the designers of the 6510 in their operating manual don't say "141 means STA". They say something like, "code 141 means store the contents of the accumulator in the address represented by the next two bytes". Then they recommend that the abbreviation (or mnemonic) for "Store contents of Accumulator in Memory" should be STA.

There are two reasons for learning mnemonics. Firstly they make the job of writing down your programs easier and quicker. But keep the priorities in mind. The job is to learn, understand and use the full machine language instructions in order to start writing your own machine language programs. Do not try to memorise a long list of mnemonics. Just let them sink in, along with their full meanings, as you progress.

Secondly, at a later stage you may wish to get into machine language in a serious way and start using *assembly language.* I hope you do. Assembly language is only a step up from the machine language programming in this book and is based directly on mnemonics. It does, however, require an *assembler program* in the computer to translate it for the computer. But remember, you can program very effectively in machine language without any thought of assembly language or assemblers. You cannot use an assembler, though, without a knowledge of the machine language instructions you will be dealing with here.

What about BASIC?

Machine language adds a whole new dimension to your programming ability and to the pleasure and satisfaction of writing good programs. But do not forget that your C-64 has a powerful and very quick BASIC language facility. Many great programs, in both the commercial and gaming fields, are written with a combination of BASIC and machine language routines.

There is little doubt that writing and debugging in a high level language such as BASIC is easier and quicker than doing the same thing in machine

language. On the other hand, we have already seen how machine language can do a variety of things that BASIC can't do. My recommendation is to get into machine language programming but keep up your study of BASIC. Use the best of both worlds to write those really great programs.

Finally, bear in mind that the ultimate test of a program is whether it works and achieves the result you want. Whether you are programming in a high level language or in machine language, it is a safe bet that there are several ways that your program could be written. Some may be more efficient and/or elegant, but do not worry too much about these things at this stage; they come with practice and experience. It really does not matter much if you take a few extra bytes or do something in a rather round-about way. Machine language is so fast and generally economical of memory, that the result will be satisfactory to you. Efficiency and/or speed of operation may be absolutely critical in the performance of a large commercial program or in the control of vital instrumentation; but we are hardly at that stage yet!

So off we go. Chapter 1 looks at some BASIC statements which will give you an idea of the differences and similarities between BASIC and machine language. It also looks at some BASIC statements which are extensively used with machine language programs.

Then in the next 9 chapters you can get your hands on the C-64 and work through some practical programs. If you have any doubts about the explanations in these chapters, or would like a little more detail, have a look at Appendices A, B, C, and D.

1

A Quick Look At Some BASIC

ASSIGNMENT OF VALUES TO VARIABLES

After PRINT, the next thing you learned in BASIC was probably LET and then a variety of other statements that let you assign data to variables.

LET

In C-64 BASIC you don't have to type in LET but the statement is implied. You will be quite familiar with lines like:

```
10 LET X = 0              10 X = 0
10 LET A$ = "QWERTY"      10 A$ = "QWERTY"
```

These are the statements in BASIC programs most frequently used to assign values to numeric or string variables. And in BASIC, that is all you have to say. The BASIC interpreter does the rest for you. It records the variable name, finds the right area in RAM memory to store the variable values and constantly keeps them all under control.

In machine language programming, assignment of values is just as important. In machine language, however, you have to tell the computer *everything* it has to do: where you want it to store things in memory and where it has to look in memory to find things. If you want a record of the variable names you're using, you have to do it yourself with pencil and paper. Nor are there any syntax or operating error statements to help you.

These things may make machine language programming a little more demanding. But for your trouble you gain the tremendous advantage of talking directly with the 6510 microprocessor, and you can use the greatly increased programming power this facility offers.

GET and INPUT

These are the important BASIC statements that enable you to enter values from the keyboard. GET is a sort of *one-shot* statement and if no key is pressed it simply returns 0 or the empty string (" "), then goes on with the program. INPUT, on the other hand, sits and waits until the operator enters something from the keyboard.

In machine language programs you can directly access the keyboard from the program but the GET and INPUT statements are sometimes still the easiest and most convenient ways of handling input.

READ and DATA

This very handy facility in BASIC uses DATA to store values inside a line in a program so that they can be assigned in sequence by the READ statement.

When you're writing programs in machine language, you'll continue to use these statements frequently. Remember that we are working here with the unexpanded C-64. With the unexpanded C-64 you can either POKE the machine language decimal codes into the RAM one at a time, or else let the READ/DATA do it for you. The latter is far more convenient.

LOOPS, BRANCHES AND SUBROUTINES

These are the abilities that make the computer such a powerful tool for decision making. Without them (and the few other things we've dealt with so far), there would not be much left in most BASIC programs! To do these things in BASIC you use the following statements:

- GOTO which causes the program to branch or jump to another line and start operations from there;
- ON. . . GOTO which evaluates the arithmetic expression following the ON and then jumps to the line numbers resulting from the evaluation;
- IF. . . THEN which allows the program to execute an instruction IF the condition is fulfilled;
- FOR . . . NEXT which instructs the program to loop through the same set of instructions a specified number of times;
- GOSUB which sends the program to a special subroutine — usually a routine which will be used a number of times during the performance of the main program.

And that is about it. By far the majority of machine language instructions you use will be concerned with storing values in memory addresses and doing the loops, branches and subroutines. The instructions available in machine language are not as automatic nor as helpful as the BASIC statements. Therefore you have to plan more of the work. But you are not limited by the rules of the BASIC interpreter.

BASIC STATEMENTS USED WITH MACHINE LANGUAGE PROGRAMS

Now we can have a look at those commonly-used BASIC statements which have some special importance to machine language programming. In essence these instructions provide the links between the BASIC and machine language parts of a program.

PEEK

This is the statement which allows the user to see what value is stored in any memory address of RAM or ROM. The simple statement PRINT PEEK (addresses) either as a direct command or in a BASIC program, puts on the screen the decimal number contained in the address. It allows us to get the results of machine language program operations and bring them back into the BASIC part.

POKE

This is the BASIC statement which allows you to POKE into any RAM address any decimal number you wish to use. To start off with you have to put your machine language codes into memory before you can use them. You use POKE to put them there. But later POKE may also be used to change the values in the machine language program, as for example, when you want to change the speed or level of difficulty, or the location of the action.

Keep in mind that PEEKs and POKEs cannot do any harm to your computer so you can practise these to your heart's content. POKEs into ROM memory are harmless although they have no effect.

SYS

This funny little word is the simplest, most commonly used and preferred method of transferring operation from BASIC to the machine language program. Remember that the C-64 is essentially a BASIC machine. When switched on it is all set up to receive instructions in BASIC. In a sense, machine language programs work as subroutines of a BASIC program. Even if a program does not have a single line of BASIC in it, it still has to be called into operation with the BASIC statement SYS. Similarly, like a BASIC subroutine, the machine language program must be ended with a RETURN instruction. Otherwise the computer may well stay with the machine language and you will not be able to stop it short of switching the computer off.

USR (X)

This is a rather specialised BASIC function used to transfer to the machine language program. It is very seldom used, and you will probably never need to concern yourself with it.

The function is used, mainly in mathematics programs, to transfer a parameter (X) from the BASIC program to the machine language program and return a value or result to BASIC. You have to store the starting address of the machine language program in memory addresses 785 and 786 and the computer looks at these addresses to find out where it has to start. This is a fairly complex function to use and really of value only to those who need its particular feature.

If you need to transfer values between BASIC and machine language programs, you can easily do so by reserving a few addresses of memory and using the very easy PEEK and POKE statements.

2

Get it on the Screen

BASIC v MACHINE LANGUAGE

With BASIC, the easiest way to put a graphic character on the screen is to use the POKE statement. If you key:

```
POKE 1065,83  RETURN
POKE 55337,2  RETURN
```

a red heart will appear on the top left-hand corner of your screen. Screen memory maps for characters and colors are given in Appendix E of this book. If you turn to Appendix E now you'll see that 1065 is given as the character code memory location for the position we've used near the top left of the screen. POKEing 83 into this location gets you a heart (see Appendix F for the screen codes for the characters themselves).

55337 is the color code memory location for the same screen position (see the color memory map), so into that location we POKEd 2 — the color code for red.

What's Different About A Machine Language Program?

A machine language program that did the same thing would be different in three ways:

1. It would be written as a subroutine of a BASIC program. In other words, it would have a BASIC program to operate it.

2. Instead of the data being POKEd directly into the memory locations, by the BASIC interpreter, they would be loaded into a temporary storage on the 6510 chip called a *register.*

3. The contents of the register would then be stored in the address location by *your* instructions.

An Equivalent Machine Language Program

Program # 2

```
10 N=828
20 READ D:IF D=-1 THEN 50
30 POKE N,D: N=N+1:GOTO 20
40 DATA 169,83.............LDA 83
41 DATA 141,41,4...........STA 1065
42 DATA 162,2..............LDX 2
43 DATA 142,41,216.........STX 55337
44 DATA 96.................RTS
45 DATA -1
50 PRINT CHR$(147): SYS 828
60 REM DON'T TYPE IN MNEMONICS
```

The actual machine language program is contained in the eleven numbers in the DATA lines.

Explanation of the Program

Line 10 tells the computer that you want to store this program starting at address 828. This is the first address in a memory block which runs from addresses 828 to 1019. It is set aside for use as the *cassette file buffer*. When you're not using cassette files, however, the block is vacant. It is therefore a convenient place to store machine language programs. Most of the programs in this book will be stored in this memory block.

Line 20 is the instruction to READ the DATA and to keep reading it until the "-1" marker.

Line 30 POKEs the DATA — and changes the address to the next address in the 828 to 1019 block.

Line 40 loads 83 (ie, the heart) into register A (169). The mnemonic is LDA 83.

Line 41 stores the contents from register A (141) into address 1065. Since the number 1065 is greater than 256 it needs to be written in two bytes — ie, 41,4. 41,4 means 1065 since 1065 = 41 + (4 × 256). The mnemonic is STA 1065.

Line 42 loads 2 (red) into register X (162).

Line 43 stores the contents from register X into address 55337. 55337 is written as 41,216 since 55337 = 41 + (216 × 256).

Line 44 takes you back to BASIC.

Line 45 serves as a marker which stops the READ process.

Line 50 clears the screen. Then the SYS statement calls up the machine language program stored at that address.

Why Would You Bother?

In this case it's clear that the machine language program is a complicated way of doing something that can be far more easily done in BASIC. It's even more complicated if you crunch lines 40 to 45:

```
40 DATA 169,83,141,41,4,162,2,142,41,216,96,-1
```

But this was only a simple example. Later you will see small machine language routines that will do things you can't do in BASIC simply because of their sheer speed of operation.

REGISTERS

When you use machine code you have to tell the 6510 these things:

- what to do with the instructions
- what the instructions are.

The first thing that happens to your data is that it goes into a *register*. Once it's loaded into a register, it stays there until you load other instructions to change it.

There are several registers in the 6510 chip. The main one — called register A or the *accumulator* — is the only register that can hold numbers for arithmetic operations and the results of these operations.

The previous programs used both register A and register X. Register X is less useful than register A since there are more instructions available for the latter. Register A is the general purpose register. The programs we've just looked at, for example, could have been easily written without using register X at all.

What you put into a register stays there until you replace it with something else. This means that when you have loaded data into a register you can store it in any number of different addresses without having to load it again. (In fact, that's where a machine language program really comes into its own for the longer, more complex programs.)

A register can only hold one byte of 8 bits at any one time. This means that the largest number it can hold is 255 — ie, 11111111 in binary arithmetic.

For a fuller explanation of bits, bytes and binary numbers, see Appendix A.

A LARGER PROGRAM

Program Number 3 gives you a white screen and a red heart, red diamond, black spade and black club, located near the corners.

Program # 3

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 169,1,141,32,208,141,33,208,169,
    65,141,0,4,169,83,141,35,4,169,90
42 DATA 141,32,7,169,88,141,67,7,162,0,
    142,0,216,142,67,217,162,2
44 DATA 142,35,216,142,32,219,96,-1
50 PRINTCHR$(147):SYS828
```

Explanation of the Program

Lines 10 to 30 POKE the machine language routine into RAM memory.

Line 50 clears the screen. Then the SYS statement calls up the machine language program stored at address 828 and your C-64 starts operating the routine straight away. When it's finished, the computer returns to BASIC and awaits your next command.

Now let's look at the actual machine language routine represented by the decimal numbers in these DATA lines:

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 1	169,1	LDA 1
(b)	Store contents of accumulator in address 53280	141,32,208	STA 53280
(c)	Store contents of accumulator in address 53281	141,33,208	STA 53281
(d)	Load accumulator with 65	169,65	LDA 65
(e)	Store contents of accumulator in address 1024	141,0,4	STA 1024
(f)	Load accumulator with 83	169,83	LDA 83
(g)	Store contents of accumulator in address 1059	141,35,4	STA 1059
(h)	Load accumulator with 90	169,90	LDA 90
(i)	Store contents of accumulator in address 1824	141,32,7	STA 1824
(j)	Load accumulator with 88	169,88	LDA 88
(k)	Store contents of accumulator in address 1859	141,67,7	STA 1859
(l)	Load index register X with 0	162,0	LDX 0
(m)	Store contents of X register in address 55296	142,0,216	STX 55296
(n)	Store contents of X register in address 55619	142,67,217	STA 55619

(o)	Load index register X with 2	162,2	LDX 2
(p)	Store contents of X register in address 55331	142,35,216	STX 55331
(q)	Store contents of X register in address 56126	142,32,219	STX 56126
(r)	Return	96	RTS

Explanation of the Machine Language Program

When you RUN this program, the whole screen will turn white — screen and border. To change the colors of the screen and border you POKE one of the sixteen color codes shown in Appendix G into the following memory addresses:

- border color — POKE 53280
- screen color — POKE 53281

Remember that in machine language programming, you have to tell the 6510 just about everything it has to do for you.

Line (a) loads the accumulator with 1, the color code for white.

In line (b) the 6510 is told to store this number in address 53280 (the address for border color).

In line (c) the 6510 is told to store the same number in address 53281 (the address for screen color). Notice that you have told the 6510 to do exactly the same things you would tell the BASIC interpreter to do for you by typing POKE 53280,1 and POKE53281,1. The difference is that you have told this directly to the 6510 and it can carry out your instructions immediately. It does not have to wait while the BASIC interpreter works out what *it* has to tell the 6510.

In line (d) the accumulator is again loaded, this time with the number 65. A look at Appendix F will show you that code 65 represents the spade.

In line (e) this number in the accumulator is now stored in address 1024. A look at Appendix E will show you the memory addresses for the character codes in the screen display — ie, where the spade will appear.

Jump a few lines now down to line (l) where the 6510 has been told to load the index register X with the number 0. A look at Appendix G will show you that 0 is the color code for black for characters POKEd onto the screen. Number 0 is stored in address 55296. If you look at the Color Codes Memory Map in Appendix E you'll see that address 55296 is in the top left-hand corner and is in the same relative position as address 1024 in the Screen Character Codes Map where the spade (code 65) was put in lines (d) and (e).

Index register X is the second register inside the 6510 to have a look at. It is very similar to the A register or accumulator and holds 1 byte/8 bits (ie, numbers in the range 0 to 255). There are not as many instructions available for this register as there are for the accumulator which is the main general purpose register. *Index register Y* is exactly the same as the index register X. We've used the X register here just to introduce it to you.

In the same way as explained above, lines (f) and (g) put a heart in the top right-hand corner, lines (h) and (i) put a diamond in the bottom left-hand corner and lines (j) and (k) put a club in the bottom right-hand corner. Check out for yourself the screen codes and addresses in the various appendices at the back of the book. You'll need to know your way around them.

Note that lines (n), (p) and (q) store the various color codes according to the Color Codes Memory Map.

In line (o) the index register X is loaded with the number 2 which is the color code for red. This color code is then stored in the two color code memory addresses in the following two lines, (p) and (q).

Twice in this program the X register has been loaded with a number and then this number has been stored in *two* following addresses. We can do this because a number loaded into a register will stay in that register until another number is loaded into the register. So when you load a number into a register you can then store that number in as many addresses as you like.

ADDRESSING MODES

Under the heading "Decimal Codes" in the Machine Language Program are the actual numbers which the computer recognises as the machine language instructions. The first number in each code tells the 6510 two things:

1. What it has to do, (for example, load a register) and
2. What the following numbers in the instruction (if any) mean.

As mentioned before, in machine language programming you have to tell the 6510 what data to use, what address to get data from and what address in which to store data. Therefore, each instruction may have several different first numbers in its decimal code to identify what address information the 6510 has to look for in the rest of the decimal code instruction. These different sorts of address information are called *addressing modes.* In this program three modes of addressing are used. They are:

1. Immediate Addressing Mode

The instructions to Load accumulator. . . and Load X register. . . are in the *immediate addressing mode.* For example, code 169 means load the accumulator with the next immediate number. In line (a) this is the number 1.

2. Absolute or Direct Addressing Mode

All the instructions in this program to Store the contents of. . . are in the *absolute* or *direct addressing mode.* For example, code 141 means Store the contents of the accumulator in the address represented by the following two numbers. Similarly, code 142 says Store the contents of the X register in the address represented by the next two numbers.

3. Implied Addressing Mode (sometimes called Inherent)

The last instruction in line (r) telling the 6510 to return the program to BASIC is in the *implied addressing mode.* There are a number of these types of instructions which we will meet in later programs which mainly tell the 6510 to do things *inside itself* and we do not have to give it any other specific addressing information. They are all single number instructions.

Every machine language instruction must be ended with the RETURN instruction. The C-64 is essentially a BASIC machine and a machine language program is treated as a subroutine of the BASIC program. So just as a subroutine in BASIC must be finished with a RETURN statement, we have to end machine language programs with RETURN.

ADDRESSES: HOW TO HANDLE THEM

Except for the first 256 addresses (ie, 0 to 255) all addresses in the computer are more than 255 and therefore will not fit into one regular address byte which will only hold a maximum of 255. So all addresses must be held in two bytes. The rules are as follows:

1. All addresses must be held in two consecutive bytes (or addresses) with the Least Significant Byte (the smaller part of the address) in the first address and the Most Significant Byte (the larger part of the address) in the second or next higher, address.

2. The number represented in the Most Significant Byte is 256 times the actual value that is in that byte. Remember that this single byte of 8 bits can only hold numbers in the range 0 to 255, but we (and the 6510) treat this value as being 256 times its actual value.

3. An address must be shown in two bytes even if the first, or Least Significant Byte is 0.

Some Examples From This Program

line (b)	Most Significant Byte	208	
	multiply by	256	
	equals	53248	
	add Least Significant Byte	32	
	actual address	53280	(represented by 32,208)
line (d)	Most Significant Byte	4	
	multiply by	256	
	equals	1024	
	add Least Significant Byte	35	
	actual address	1059	(represented by 35,4)

Let's try it the other way around, the way you would work it out.

Line (f) actual address	53281
divide by	256
equals	208 (plus something)

Now 208 is the Most Significant Byte — ie, there are 208 complete 256's in the address number. Multiply 208 by 256 equals 53248 which is the value of the M.S.B.

Actual Address	53281
Subtract value of the M.S.B.	53248
Remainder is then the value of the L.S.B.	33

53281 is represented by 33,208

EXPERIMENTING

Now that you have something on the screen, you may care to conduct a few experiments of your own. If you're going to do so you should remember:

- to put your machine code program in the framework of a BASIC program like the one we've used and
- to use 96 and -1 as the last two pieces of DATA.

As well as loading different instructions to be stored at different addresses, you may care to try to store the same instruction at several different addresses.

3
Move it!

Now that you have some colored characters on the screen it's time to try moving them around. With machine language you can move things either instantly or slowly.

It's very useful, especially when you're writing games programs, to be able to draw lines instantly on the screen. Even if the rest of your game is written in BASIC and even if your frogs, spaceships and other symbols and characters overwrite the lines, it is still helpful to call up a machine language program that pops the line back on the screen.

The second thing we want to be able to do is to move laser beams, photon torpedoes and such across the screen. When using machine language we have a surprising problem. The problem is not how to make things move faster but how to slow things down so you can see them move! We'll look at ways of doing this in the second part of this chapter.

DRAWING LINES

So let's put a line across the screen. Type in Program Number 4. The machine language program is contained in the DATA lines and will again be POKEd into the cassette file buffer (828).

Program # 4

```
10 N=828
20 READ D:IF D=-1 THEN 50
30 POKEN, D:N=N+1:GOTO 20
40 DATA 162,40,169,120,157,143,5,169,7,
    157,143,217,202,208,243,96,-1
50 PRINT CHR$(147):SYS 828
```

The Machine Language Program

RUN this program and you will instantly see a line appear 16 rows down from the top of the screen. Notice that the machine language program contained in line 40 is only 16 bytes long and is stored in the cassette buffer. If you wanted to, you could type over this BASIC program with another BASIC program. You cannot SAVE the program on tape at this stage but we will talk about this later. The point is that you can call up and execute these 16 bytes of program simply by including SYS 828 in your BASIC program or by keying SYS 828 as a direct command. Every time you or your program give the SYS 828 instruction your line appears instantly.

	Instruction	Decimal Code	Mnemonic
(a)	Load X register with 40	162,40	LDX 40
(b)	Load accumulator with 120	169,120	LDA 120
(c)	Store contents of accumulator in address 1423 + X	157,143,5	STA 1423,X
(d)	Load accumulator with 7	169,7	LDA 7
(e)	Store contents of accumulator in address 55695	157,143,217	STA 55695
(f)	Decrement contents of X register by 1	202	DEX
(g)	Branch on condition that result is not 0	208,243	BNE
(h)	Return	96	RTS

Explanation of the Program

This program starts at address 1463, the address in the last column of the 10th row (see your Screen Character Codes Map) and puts in that address screen code 120, a narrow segment of a line. Then it loops back and by means of *address indexing* puts the same character in address 1462, 1461, and so on to 1424 — until the line is completed.

Now we'll look at the program line by line and in detail.

In line (a) the X register, in immediate addressing mode, is loaded with 40. This 40 will be used as a *loop counter* just as you use loop counters in BASIC. Notice in machine language, however, that loops are not the same as they are in BASIC — ie, FOR. . .NEXT. Rather, they are more like this:

```
10 LET N = (something)

20 ". . . BASIC LINE TO DO
   SOMETHING. . . "
30 IF N = 0 THEN GOTO . . .
40 LET N = N - 1
50 GOTO 20
```

OR

```
10 LET N = 0
20 (do something)

30 IF N = (something) THEN
   GOTO. . ."OUTSIDE LOOP"
40 LET N = N + 1
50 GOTO 20
```

The program model on the left-hand side is like the method used in this program.

Line (b) simply loads the accumulator with 120, the screen code for the line segment, in the immediate addressing mode.

Line (c) introduces a new addressing mode called *absolute indexed by X register.* There is a *base* address, 1423, to which the contents of the X register are added by the 6510 to get the *actual* address where the data, 120, will be stored. On the first cycle of the loop X will equal 40, so the *actual* address will be 1423 + 40 or 1463. Each time through the routine, as X decrements by 1 this actual address will become 1 less.

In line (d) we start the usual procedure for putting a color character on the screen. This time the accumulator is loaded with 7, the color code for yellow. This is the immediate addressing mode.

In line (e) this color code in the accumulator is stored in the appropriate color code memory location which again uses the absolute indexed by index register X addressing mode. This address has to move back along the row, so that on the first cycle the actual address will be 55695 + 40 or 55745. It will reduce by 1 each time through the loop as X is decremented by 1.

Line (f) is the instruction to decrement the value in the X register by 1. Eventually, as a result of this instruction, X will become 0.

Line (g) is the very important *conditional branch instruction.* Branch simply means go to some other place in the program and start operating from there. This particular branch instruction says to branch to the start of line (b) and continue to do this until the operation of the decrement instruction in line (f) produces a 0 result.

Line (h) is the essential *return* instruction required to get control back to BASIC.

CONDITIONAL BRANCH INSTRUCTION

There are two very important aspects of conditional branch instructions: *testing the condition* and *addressing mode.*

Testing the Condition

The 6510 has another register called the *status register.* This is a normal 1 byte/8 bit register but is used by the 6510 in an unusual way. Each individual bit can have only one of two conditions or states — a 0 or a 1. These flags are used to show whether or not particular conditions exist in the operation of the 6510 or the program. We are concerned here with the zero flag. If:

1. any arithmetic or logical operation produces a zero result, then the zero flag is set to 1;
2. a non-zero result is produced then the zero flag is reset to 0.

There are two conditional branch instructions which work with the zero flag. They are:

1. Branch on condition that the result of the arithmetic operation is 0, and
2. Branch on condition that the result of the arithmetic operation is *not* 0.

In this program, *decrement by 1* is an arithmetic instruction and the zero flag keeps a record of its operation. The particular instruction tells the 6510 to branch the program back until the result is 0 then knows to let the program pass on to the next instruction.

Addressing Mode

All the conditional branch instructions have a special addressing mode called relative. In this mode, instead of being given a particular address in memory, the 6510 is told to jump (or branch) the program over a certain number of bytes depending on the position of the branch instruction in the program.

Clearly when the program has to move back to an earlier line, it is taking off a number of bytes. To put it another way, the program is jumping a *negative number* of bytes. Unlike BASIC there is no minus sign in machine language so you have to use a special type of number representation for these negative numbers. These are called *2's complement representation.* For our purposes we'll calculate the 2's complement of the number of bytes to jump or branch backwards in the program by deducting the number of bytes from 256 and using the result in the machine language instruction.

In the program we're looking at, the branch is from the end of the instruction in line (g) back to the start of line (b). Count them and you will find that the program has to jump back 13 bytes. Subtracting 13 from 256 gives you the 243 shown as the second number of the instruction in line (g). Your little 6510 knows exactly what to do when it meets these 2's complement numbers. Notice that we did not jump back to the start of line (a) as this includes the loop counter in the X register. Branching to line (a) would therefore create an endless loop. The same sort of thing can happen in BASIC but when you're programming in machine language it's likely you'll have to switch the computer off to stop it! Remember that when you are in machine language you lose quite a lot of the protection that the BASIC system gives you against syntax and operating errors. Often RESTORE/STOP has no effect!

Later on you will meet some programs which have a jump or branch *forward.* In these cases you will use the regular positive numbers for the number of bytes to be jumped over. But more of this when we come to it. IMPORTANT NOTE: You can only jump 128 bytes backwards (that is 128 in 2's complement representation) and 127 bytes forward.

ABSOLUTE INDEXED BY INDEX REGISTER X ADDRESSING MODE

This is an important and frequently used mode of addressing. It means that you can have immediate access to up to 256 addresses while using only one address in a store instruction. Call that the *base* address. By

adding the value in the X register to the base address you get the *actual* address you want the 6510 to do something with.

If you use the absolute indexed by X addressing mode in an instruction in your program, the 6510 will automatically add the current value in the X register to the address you have given it in the instruction.

Absolute indexed by index register Y addressing mode is used in exactly the same way. There is a separate set of decimal codes for this addressing mode which causes the contents of the Y register to be added to the *base* address.

MOVE IT SLOWLY

Basically we will move things across the screen in exactly the same manner as we did in the first part of this chapter. That is, we'll use the conditional branch instruction but this time we will have to use more loops that do nothing more than waste time or delay the main operation so that we can see what is happening! Type in Program Number 5.

Program # 5

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 162,0,160,40,169,120,157,184,5,
    169,7,157,184,217,138,72,152,72
42 DATA 160,255,162,255,202,208,253,136,
    208,248,104,168,104,170
44 DATA 232,136,208,224,96,-1
50 PRINTCHR$(147):SYS828
```

When you RUN the program you will see the same line you had in the previous program but this time it will progress slowly across the screen from left to

right. (Note that this is the opposite direction to the previous program!)

The machine language program is now quite a bit longer and most of this is due to the timing loops. Since the 6510 registers are only 1 byte, the largest loop counter you can have in a register is 255. This slows the program down but it is still very fast and usually you have to nest at least two loops. This is how this program works; with one program loop inside another loop, both with counters set at 255. This is about 65,000 total cycles and it is still not all that slow. I will show you how to change it later on.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load X register with 0	162,0	LDX 0
(b)	Load Y register with 40	160,40	LDY 40
(c)	Load accumulator with 120	169,120	LDA 120
(d)	Store contents of accumulator in address 1464+X	157,184,5	STA 1464,X
(e)	Load accumulator with 7	169,7	LDA 7
(f)	Store contents of accumulator in address 55736+X	157,184,217	LDA 55736,X
	The Delay Loops		
(g)	Transfer contents of X register to accumulator	138	TXA
(h)	Push accumulator onto stack	72	PHA
(i)	Transfer contents of Y register to accumulator	152	TYA
(j)	Push accumulator onto stack	72	PHA
(k)	Load Y register with 255	160,255	LDY 255
(l)	Load X register with 255	162,255	LDX 255
(m)	Decrement contents of X register by 1	202	DEX
(n)	Branch on condition that result is not 0	208,253	BNE
(o)	Decrement contents of Y register by 1	136	DEY

(p)	Branch on condition that result is not 0	208,248	BNE
(q)	Pull accumulator off stack	104	PLA
(r)	Transfer contents of accumulator to Y register	168	TAY
(s)	Pull accumulator off stack	104	PLA
(t)	Transfer contents of accumulator to X register	170	TAX
(u)	Increment contents of X register by 1	232	INX
(v)	Decrement contents of Y register by 1	136	DEY
(w)	Branch on condition that result is not 0	208,224	BNE
(x)	Return	96	RTS

Explanation of the Machine Language Program

Lines (a) to (f) and (u) to (x) actually draw the line. But in this program the line starts at the left-hand side of the screen and grows to the right. This is opposite to the first program in this chapter and requires a change in the method of indexing the addresses in which the character and color codes will be stored.

In line (a) the X register is loaded with 0. The contents of this register will be used to index the addresses. In the first pass through the program, X will equal 0 and so the base address will be the actual address. In line (u) the *increment* instruction is used. Like decrement, this uses the implied addressing mode and each time the program passes this instruction, 1 is added to the contents of the X register. In this program the X register is not used as the loop counter.

Line (b) loads the Y register with 40. This time the Y register will be used as the loop counter but has nothing to do with address indexing.

In line (c) the accumulator is loaded with 120 which again is the code for the little line segment.

Line (d) stores the contents of the accumulator in address 1464 + X. From your screen character memory map you can see that 1464 is the first address on the 11th row. On the first time through since X = 0, the actual address will be

1464. Next time through, X will have been incremented and will go on across the line until the loop counter in Y register is used up with the decrement instruction in line (v).

As soon as Y becomes 0, the zero flag will fly (be set to 1) and the conditional branch instruction in line (w) will allow the program to pass on to the next instruction — the return to BASIC in line (x).

Now to the timing or delay loops themselves. But before we can do that, something new has to be added to your knowledge.

THE STACK

Another register inside the 6510 is called the *stack pointer.* A pointer is usually only an address. In other words, in this register the 6510 keeps an up-to-date record of the address of the last item on the stack.

So what is the *stack?* When it starts up, the C-64 reserves a block of memory addresses in RAM from address 256 to address 511. This is a block of 256 addresses in which the 6510 temporarily stores data during its operations. You can imagine that it will have numbers, like return addresses from subroutines and so forth, which it needs to use later. So it pushes these values onto the stack from time to time and keeps a record of where the last item on the stack is by keeping a record of the address in the stack pointer register.

In machine language programs we can also use the stack to store values which we want to use later in the program. The 6510 therefore has two instructions which allow us to push the contents of the *accumulator* onto the stack and pull these contents back into the *accumulator.*

IMPORTANT: The stack is essentially a last-in-first-out device. You must be sure that you observe this rule.

In the 6510 there are no direct instructions to put either the X register or the Y register on the stack. Fortunately there are some instructions that allow you to transfer the contents of the X (or Y) register into the accumulator and to transfer the contents of the accumulator into the X (or Y) register.

If you study the machine language program listing you will notice that at line (f) there are values in the X and Y registers which are essential to the drawing of the line part of the program. But you have to use these registers in the delay or timing loops, simply because there are no others. So in line (g) we transfer the contents of the X register to the accumulator. Then in line (h) this value, now in the accumulator, can be pushed onto the stack.

In lines (i) and (j) we do exactly the same thing to put the contents of the Y register onto the stack via the accumulator.

So these X and Y values essential to the line drawing program are safely stored away and both X and Y registers are freely available for any other use. Now the delay loops can start. Don't worry about the number of all these stack instructions; they are all single byte (implied addressing mode) instructions, very fast and quite economical of memory.

In line (k) the Y register is loaded with 255. This will be the loop counter for the *outside loop* and is the maximum you can put in this single byte register.

In line (l) the X register is loaded with 255. This will be the loop counter for the *inside loop*. The next two lines, (m) and (n), complete the inside loop.

Line (m) decrements the X register by 1 each time through.

Line (n) is a conditional branch instruction which branches the program back to the start of line (m) until the result of the decrement instruction causes X to equal 0. Notice the program jumps back over 3 bytes so the 2's complement representation of this is 256 - 3 = 253, the second number in the decimal code in line (n).

As soon as the conditional branch in (n) lets the program pass on, it comes to line (o) (now in the outside loop), where the contents of Y register are decremented by 1.

In line (p) is another conditional branch instruction which passes the program back to the start of line (l) where the counter (X register) is set up again for the inside loop, which loops up and down until the program passes on again to line (o). Then it goes back to the inside loop until such time as the contents of the Y register have been decremented to 0 and the program goes on to line (q).

This is an important part to look at. At this stage the line-drawing values of X and Y have to be recovered so the program can get on with drawing the line again.

In line (q) the last value on the stack is pulled off and placed in the accumulator. This value is then transferred from the accumulator to the Y register, in line (r).

Lines (s) and (t) do the same thing to get the original value back into the X register. NB: the contents of the X register were put on the stack first before the contents of the Y register. Therefore, you have to take the Y register off *before* the X register.

At this point the program is all set to go on with drawing the line.

In line (u) the X register is incremented by 1. This is now ready to index the address to the next position in which to put the line segment (code 120).

In line (w) the loop counter for drawing the line (in the Y register) is decremented by 1.

In line (w) there is another conditional branch instruction which will take the program right back to the start of line (c) to begin all over again. This will continue until Y has been decremented to 0. Then the conditional branch instruction will allow the program to pass on and return to BASIC in line (x). Be careful not to include line (a) or (b) in this loop otherwise the values of X and Y will simply be returned to their original values and you'll have a continuous loop again!

Count the number of bytes to be jumped back in this loop from the end of the instruction in line (w) to the start of the instruction in line (c). It should be 32 bytes — which in 2's complement representation is 256 −32 = 224 or the value of the second byte in line (w).

CHANGING THE LENGTH OF DELAYS

The loop counters for the two delays are the 255's in lines (k) and (l). The program has been POKEd into memory from address 828 onwards. So you can count down from 162 = 828 in line (a) and see these counters are in addresses 847 and 849. POKE a 1 into 847 (outside loop) and only the inside loop effectively delays the program. Try various values. Careful though, do not RUN the program because it will only put the old value in the DATA line back in memory. Use the direct command, SYS 828.

But What About A Photon Torpedo?

Well, maybe not a photon torpedo, but often you may want not a line to go across the screen but rather some particular object.

This is no real problem with the last machine language program and only involves a sort of draw and undraw operation added to the program. In other words, draw a space over the last object put on the screen!

LIST the BASIC program you (hopefully) still have in your C-64 and add the following line:

```
43 DATA169,32,157,184,5
```

Then change the 224 in line 44 to 219. RUN this program and you should see a single little 120 character move across the screen.

So what have we done? Between the lines (t) and (u) in the original machine language program the following two machine language instructions have been added:

Load accumulator with 32	169,32	LDA 32
Store contents of accumulator in address 1464 + X	157,184,5	STA 1464,X

The accumulator is loaded with 32 which is the screen code for a space. What happens is that the line segment (code 120), is first put into a screen address. Then after the delay loops, a space is stored in the same address and the program then immediately goes on to the branch instruction and back to the start to put 120 into the next address. Note that you have also changed the 2's complement number in line (w) to 219. Because of the additional machine language instructions the program has to jump over an additional 5 bytes.

4

Drawing a Block on the Screen

Why bother to draw a block on the screen? Many games are played on solid blocks considerably smaller than screen size. The rest of the screen can then be used for results and other information.

You do not need to draw a block of solid color. Instead, you could draw a block of spaces (code 32) that will instantly *clear* a block out of your existing display.

Program Number 6 does this. It's essentially the same as the program in the introduction, except that it doesn't contain the BASIC block.

Program # 6

```
10 N=828
20 READ D:IF D=-1THEN 50
30 POKE N,D:N=N+1:GOTO 20
40 DATA 169,1,160,10,72,170
42 DATA 169,160,157,90,4,169,0,157,90,
   216
44 DATA 232,136,208,242,104,24,105,40,
   144,232,96,-1
50 PRINTCHR$(147):SYS828
```

When you RUN this a black block will appear on the screen. Now add Program Number 7 to Program Number 6.

Program # 7

```
100 A=1114:POKE650,128
110 GETA$:IFA$=""THEN110
115 SYS828
120 IFA$=CHR$(133)THENB=1
130 IFA$=CHR$(134)THENB=-1
140 IFA$=CHR$(135)THENB=40
150 IFA$=CHR$(136)THENB=-40
160 A=A+B:IFPEEK(A)=32THENA=A-B
170 POKEA,209:GOTO110
```

RUN the program, then using the function keys try moving the little character around the block.

Explanation of the Program

In line 100 the first address in the top left-hand corner of the block has been assigned to variable A. Poke 650,128 makes all keys repeat. Normally 650 has a value of 0 and only the cursor keys repeat.

Line 110 is the regular GET statement which looks for input from the keyboard.

Line 115 calls the machine language program to draw the block on the screen with SYS 828.

The next four lines check the keyboard input against the Chr$ codes of the function keys. If one of these keys is pushed then a value will be assigned to B. (Do you see how B equal to 1 or -1 moves along a row while B equal to 40 or -40 moves up and down the screen?) B is added to A to give the new address for the moving character. But before moving it, the program PEEKs at the new address to see if it contains 32 (the code for space) to make sure it is not outside the block. If it is, then B is subtracted to make sure the little character cannot get off the block.

In line 170 the little character (code 209) is POKEd onto the screen and the program returns to GET in line 110 for the next key input.

If you have not pressed a key, the program waits at line 110 and the little character remains displayed in the same position on the screen. As soon as you do press one of the function keys, the program calls SYS 828 and prints the block over the previous one and the character disappears. The program then goes on to calculate the new address and POKE the character into the new position.

This idea can be used in many games where you chase things around a playing field. All you have to do is *overwrite* the old position with a new playing field!

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 1	169,1	LDA 1
(b)	Load Y register with 10	160,10	LDY 10
(c)	Push accumulator onto stack	72	PHA
(d)	Transfer accumulator to X register	170	TAX
(e)	Load accumulator with 160	169,160	LDA 160
(f)	Store contents of accumulator in address 1114 + X	157,90,4	STA 1114,X
(g)	Load the accumulator with 0	169,0	LDA 0
(h)	Store contents of accumulator in address 55386 + X	157,90,216	STA 55386,X
(i)	Increment contents of X register by 1	232	INX
(j)	Decrement contents of Y register by 1	136	DEY
(k)	Branch on condition that the result is *not* 0	208,242	BNE
(l)	Pull accumulator off stack	104	PLA
(m)	Clear carry flag	24	CLC
(n)	Add 40 to contents of accumulator	105,40	ADC 40
(o)	Branch on condition that carry flag is cleared	144,232	BCC
(p)	Return to BASIC	96	RTS

Explanation of the Machine Language Program

This program starts at our chosen address on the screen and draws the line of 10 characters (code 160) across the screen. Then it returns to the address immediately below the first starting address and draws another line of 10 characters. So using X to index the addresses, the program records the starting value of X on each row so it can go back, add 40 to the value of X and then do another row.

In line (a) the accumulator is loaded with 1. This will be the first value of X. We do it in the first line as the program does not want to come back to this again.

In line (b) the Y register is loaded with 10 to become the loop counter for doing the rows. Y can be changed freely to a maximum of 40 to change the width of the block.

Now in line (c) the contents of the accumulator are pushed onto the stack. This is the value of X at the start of the first row and is now safely tucked away on the stack. The value is still in the accumulator; it will not change until something else is loaded in.

So in line (d) the 1 in the accumulator is transferred to the X register to be used as the address index while the accumulator is free to be used again.

In line (e) the accumulator is loaded with 160, the code for a reverse space.

In line (f) this 160 in the accumulator is stored in address 1114 + X. X at this point is 0 so the actual address is also 1114.

In line (g) the accumulator is loaded with 0, the color code for black.

In line (h) the color code for black is stored in address 55386 + X or 55387 which is the color code address corresponding with the character code adress 1114 in line (f).

In line (i) the contents of the X register is incremented by 1 which now makes the index 1.

In line (j) the Y register containing the loop counter is decremented by 1.

In line (k) the conditional branch instruction will continue to branch the program back 14 bytes (see the 2's complement representation in the 242 in that decimal code) to the start of line (e) and this repeats until there are 10 reverse spaces (code 160) in the first row. When this is done and Y equals 0 the conditional branch instruction will allow the program to pass to the next instruction.

Remember that the stack holds the starting value of the X register. In line (l) this value is pulled off the stack and put in the accumulator and 40 can be added to this X value. This new value of X is added to the first address, 1114 to make 1154, the first position on the next line immediately below the starting address.

The Addition Instruction

There is only one *addition* instruction available in the instruction set for the 6510. It is called *addition with carry* and it indicates that the addition is related to the condition of the carry flag. When the addition instruction is executed, 1 is added to the sum if the carry flag is set to 1. It is good practice, therefore, to invoke the *clear carry flag* instruction immediately before you use the addition instruction. There is a lot more information about both the carry flag and addition in appendices A and B.

So in line (m) the clear carry flag instruction is put into the program so you can safely go ahead with the addition.

In line (n) 40 is added to the contents of the accumulator. You can only add a number to (or subtract a number from) a number held in the accumulator. The result of the addition or subtraction is then put back into the accumulator by the 6510.

In line (o) a conditional branch instruction is used and is tested by the carry flag. Notice that until now, we've been using the zero flag to test conditional branch instructions. We use the carry flag this time because at each cycle of the program, 40 is added to the accumulator till it reaches 240, the starting position for the last line in the block. On the next cycle of the program, this value goes to 280 — a number which can't be held in the accumulator. When

280 is reached, the carry flag is set to 1 and the difference is returned to the accumulator.

Line (n) is implemented as soon as the carry flag is set to 1: the program returns to BASIC.

A Larger Block?

With the above machine language program you can change the value of Y quite freely up to 40 to make a block spanning the whole screen.

But because X increases by 40 each loop (in order to come down one row) and the maximum you can put in the X register is 255, you're limited to processing 7 rows at a time (7 × 40 = 280). Often this will be enough for your graphic requirements but if it does present a problem, there's an easy way to overcome it. Simply edit lines 42 and 44 of your existing program so that now they read:

```
42 DATA 169,160,157,90,4,157,114,5,169,
    0,157,90,216,157,114,217
44 DATA 232,136,208,236,104,24,105,40,
    144,226,96,-1
```

When you RUN the program now you'll see that the block is double its previous depth. We have added the following machine code instructions.

Between line (f) and (g) Store contents of accumulator in address 1394+X	157,114,5	STA 1394,X
Between lines (h) and (i) Store contents of accumulator in address 55666+X	157,114,217	STA 55666,X

The changes made in BASIC line 44 account for the extra bytes to be jumped over by the conditional branch instructions. In effect, we've simply drawn two blocks simultaneously. There are ways of doing the whole screen in one run, however, and we'll deal with them later in the book.

5

Reversing the Screen

Why Reverse?

Reversing the screen is a commonly used method of simulating explosions and can provide quite spectacular displays. You can also reverse a section of text after a brief delay to draw special attention to, for example, games instructions or warnings.

A Sample Program

With this program only the top part of the screen will be reversed. However by changing the addresses and/or the size of the counter you can reverse any number of rows. Type in Program Number 8.

Program # 8

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 162,240,189,255,3,24,105,128,
   157,255,3,169,2
```

```
42 DATA 157,255,215,202,208,239,96,-1
50 POKE53281,0:PRINTCHR$(147)
60 SYS828:FORN=1TO500:NEXT:GOTO60
```

When you RUN this program, the screen will turn completely black. Then the top half will flash red and then return to black.

The machine language program contained in line 40 of this program does the reversing of the screen and prints the reversed characters in red. Notice, though, that the machine language program is simply a subroutine of the BASIC program. It is the BASIC program which actually controls what is displayed on the screen. It is worth remembering this point. It may be very satisfying to write a large machine language program of several thousands of bytes but it is often your ability to incorporate some small machine language routines within a good BASIC program that will produce a really great looking game.

The Machine Language Program

Instruction	Decimal Code	Mnemonic
(a) Load X register with 240	162,240	LDX 240
(b) Load accumulator with contents of address 1023 + X	189,255,3	LDA 1023,X
(c) Clear carry flag	24	CLC
(d) Add 128 to the contents of the accumulator	105,128	ADC 128
(e) Store contents of accumulator in address 1023 + X	157,255,3	STA 1023,X
(f) Load accumulator with 2	169,2	LDA 2
(g) Store contents of accumulator in address 55295 + X	157,255,215	STA 55295,X
(h) Decrement X register by 1	202	DEX
(i) Branch on condition that the result is *not* 0	208,239	BNE
(j) Return to BASIC	96	RTS

Explanation of the Machine Language Program

To get the reverse of the characters listed in Appendix F you add 128 to their code numbers. The blank part of the screen consists of addresses containing 32, the code for a space. To prove this, clear the screen and then type PRINT PEEK and any one of the screen addresses between 1024 and 2023. You will get 32 on the screen. Now type in as a direct command:

POKE 1044, 160: POKE 55316,0: POKE 1084,32: POKE 55356,6: POKE 1124,160: POKE 55396,0.

Type RETURN and you will get a black space near the middle of the screen, a blank space immediately underneath it and another black space under that again. The screen code 160 is the reverse of the normal space 32.

If the top part of the screen is blank, all addresses in that part of the screen hold code 32. When it is reversed and all addresses hold code 160, this part of the screen turns blue. The same thing also happens to any other screen character which is on the screen at the time.

To create reverse images we simply take the contents of each address out, add 128 to it and then put it back into the same address.

To do this we need a loop to go through all the addresses involved. So we use the absolute indexed by X register addressing mode.

In line (a) the X register is loaded with 240. This will serve both to index the addresses and as the loop counter.

In line (b) the accumulator is loaded with the contents of address 1023 + X. The first time through when X = 240, the actual address will be 1023 + 240 (or 1063). Notice that this is the last address on the 6th row of the screen character codes memory map in Appendix E.

In line (c) the addition part is started. Remember that addtion is started by establishing the carry flag in its right condition. So in this line we write the clear carry flag instruction.

In line (d) the addition instruction will add 128 to the contents of the accumulator and will then put the result back in the accumulator replacing the number it previously held.

In line (e) the contents of accumulator (now the reverse character code) is put back in the same address.

In line (f) the accumulator is loaded with the color code — in this case red. You could change this to any other color you liked.

In line (g) this color code is stored in the color code address corresponding to the character code address. This address is also indexed by the X register and will keep up with additions to the character code address.

In line (h) the contents of the X register are decremented by 1. This decreases X both as the address index and as the loop counter.

In line (i) is the conditional branch instruction which will continue to branch the program back to the start of line (b) as long as the result of decrementing X is not 0. Note that the program jumps back over 17 bytes and that this is represented by the 2's complement number 239 in the decimal code in line (i).

When X becomes 0 the conditional branch instruction will simply allow the program to pass on to line (j) where it will return to BASIC.

Note however, that X does not become 0 on the last run through until it reaches line (h) and is decremented. So in this pass at line (b) when X = 1, the address to have its contents loaded into the accumulator will be 1023 + 1 (or 1024) which is the first address on the screen.

REVERSING THE WHOLE SCREEN

This is an important program because it introduces two new tools for you to work with. They are the

- *zero page indirect indexed addressing mode* and your first
- *logical operator.*

This addressing mode is a little more complex than the others we've dealt with so far but if you take it step by step you should find it easy enough to understand and to use. Its key feature is that it lets you set up loops to any length you want and therefore to take full advantage of the large amount of RAM you have available on the C-64. So before we move on to the machine language program itself, we should pause for a moment to see how it works. To do that, we'll break it down into its three parts: *zero page, indirect* and *indexed.*

Zero Page

We've seen how addresses have to be held in two bytes — the L.S.B. in the first byte and the M.S.B. in the next higher byte. In the first 256 addresses (from 0 to 255) the M.S.B. will be 0 while the L.S.B. will vary between 0 and 255. This group of 256 addresses is called *zero page* because the M.S.B. is zero.

The 6510 has a special way of handling zero page addresses. For these addresses it does not require you to specify the M.S.B. in the instructions. When the 6510 sees one of the zero page instructions it simply assumes that the M.S.B. is 0 and goes to the specific address in the range 0 to 255 that you have put in the L.S.B.

Zero page instructions are very fast and economical of memory. Unfortuntely, in common with most computers using this type of microprocessor, Commodore uses most of zero page for the C-64's own system variables. There is very little of this area available for our use. There are, however, four addresses not so used which are set aside for the programmer. They are the addresses 251 to 254.

Indirect

In all the addressing we have done so far, the address shown in the instruction has been the actual address or the base address to be indexed by the X or Y registers. These have been addresses in which data have been stored.

In indirect mode, however, the address you put in the instruction does not contain such data. Rather, it contains *another address* in which the data you want the 6510 to work with, is stored. So, for example, when we put the L.S.B. of this address containing an address in 251 and its M.S.B. in address 252, we need only tell the 6510 to look at 251 on zero page. It will automatically look at both 251 and 252, to get the actual address it has to go to in order to get the data you want it to work with.

Now, from what we've learned so far, we know we can change the contents of any memory address in a variety of ways. We have instructions that let us decrement, increment, add and subtract. It follows, then, that we can access as many variations on the addresses in a zero page address as we wish — simply by changing the contents of that zero page address. In other words, loops to the limit of memory!

Indexed

This is much the same sort of indexing as we have already used in several programs. With this addressing mode, however, only the Y register may be used as

the index register. When the 6510 reads the base address in, say 251 and 252, it adds on the contents of the Y register to establish the actual address with which you want it to work.

You can access up to 256 addresses in this way, just as we did with absolute indexed addressing. But now, if you want to, you can change the base address held in zero page and do another lot of indexing with the Y register.

Steps in Zero Page Indirect Indexed Addressing

Putting it all together, we can see that there are three steps involved in this addressing mode:

1. We have to load the zero page addresses (two adjacent addresses like 251 and 252) with the L.S.B. and M.S.B. of the actual or base address containing the data we want on the 6510 to work with.

2. The 6510 gets the actual or base address when it reads the zero page instruction.

3. The 6510 adds the address it finds in zero page to the contents of the Y register and calls that the actual address. Naturally, then, if we want the two addresses to be the same, Y must equal 0.

Program to Reverse the Whole Screen

Program # 9

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 169,0,133,251,169,4,133,252,160,
   0,177,251,73,128,145,251,165,251
42 DATA 24,105,1,133,251,165,252,105,0
44 DATA 133,252,201,7,208,233,165,251,
   201,232,208,227,96,-1
50 SYS828
```

When you RUN this program all the characters on the screen will reverse. Then when you type SYS828 as a direct command, the screen will return to normal. There are no color codes in the machine language program so only the characters that have been printed on the screen, (ie, the program itself) will be affected by the reversal. Color codes will be handled in the programs in the next chapter.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 0	169,0	LDA 0
(b)	Store contents of accumulator in address 251	133,251	STA 251
(c)	Load accumulator with 4	169,4	LDA 4
(d)	Store contents of accumulator in address 252	133,252	STA 252
(e)	Load Y register with 0	160,0	LDY 0
(f)	Load accumulator with contents of address 251	177,251	LDA (251),Y
(g)	Exclusive OR contents of accumulator with 128	73,128	EOR 128
(h)	Store contents of accumulator in address 251	145,251	STA (251),Y
(i)	Load accumulator with contents of address 251	165,251	LDA 251
(j)	Clear carry flag	24	CLC
(k)	Add 1 to the contents of accumulator	105,1	ADC 1
(l)	Store contents of accumulator in address 251	133,251	STA 251
(m)	Load accumulator with contents of address 252	165,252	LDA 252
(n)	Add 0 to the contents of accumulator	105,0	ADC 0
(o)	Store contents of accumulator in address 252	133,252	STA 252
(p)	Compare contents of accumulator with 7	201,7	CMP 7

(q)	Branch on condition that result is *not* 0	208,233	BNE
(r)	Load accumulator with contents of address 251	165,251	LDA 251
(s)	Compare contents of accumulator with 232	201,232	CMP 232
(t)	Branch on condition that result is *not* 0	208,227	BNE
(u)	Return to BASIC	96	RTS

Explanation of the Machine Language Program

The program for reversing the whole screen falls neatly into three separate parts:

- Lines (a) to (e). These lines load the zero page addresses with the first actual address — ie, 1024, the first address on the screen where the program will start. Then they set the Y register to zero. The Y register won't be used here for indexing.
- Lines (f) to (h). These are the lines that carry out the actual reversing action. You could, in fact, change these lines to move very large blocks of memory for a variety of other purposes.
- Lines (i) to (u). These lines are occupied with changing the screen address and testing for the conditional branch instruction.

Line (a) loads the X register with 0, the L.S.B. of address 1024.

Line (b) stores this value in address 251.

Line (c) loads the X register with 4, the M.S.B. of 1024.

Line (d) stores this value in address 252.

Line (e) loads the Y register with 0.

Line (f) loads the accumulator with the contents of the address held in address 251. Notice that this instruction tells the 6510 to find the actual address (1024) *indirectly* this time through the program. That is, the 6510 must go to 251 and look at 251/252 to find the actual address before it can store it in the

accumulator. In lines (a) to (g) the instruction was to deal directly with the contents of 251 and 252.

Line (g) Exclusive Or's the contents of the accumulator with 128. This instruction adds 128 to the contents of the accumulator to produce the reverse character code.

Line (h) loads this reverse character code back into address 1024. It is similar to the indirect instruction in line (f).

Line (i) loads the accumulator with the contents of address 251. This is 0, the L.S.B. of address 1024 on this first cycle.

Line (j) starts an addition with the instruction to clear carry flag.

Line (k) adds 1 to the contents of the accumulator. Notice that the L.S.B. held in 251 is now 1 — which means that the actual address has moved to 1025, after line (l) has stored the accumulator in address 251.

Line (m) loads the accumulator with the contents of 252, the M.S.B. of the actual address, here still 4.

Line (n) adds 0 to the contents of the accumulator. The reason for this is that the addition instruction adds in the 1 if the carry flag is set to 1. As explained in the previous chapter, when a value to go into a byte exceeds 255, the carryover puts a 1 in the carry flag. Consequently, as 1 is added to the contents of address 251, its value will eventually reach 256 at which point 1 will go to the carry flag and address 251 will go back to 0 and start to increase again as more 1's are added to it.

Line (o) puts an end to the loop that increases 1024 by 1 at each cycle of the program. It does this when the last screen address is reached by comparing the actual address held in 251/252 with the end of screen address, 2031. For 2031 the L.S.B. is 231 and the M.S.B. is 7. At line (n) the result of the addition is returned to the accumulator and it is still in the accumulator after the store instruction in line (o); this is the M.S.B.

Line (p) compares the value in the accumulator with 7 and if it has not yet reached 7, the conditional branch instruction in line (q) returns the program to line (f) to do another reversal.

As soon as the M.S.B. reaches 7, the conditional branch allows the program to pass on to line (r) where the L.S.B. in address 251 is loaded into the accumulator.

Line (s) compares the contents of the accumulator with 232. This is one more than the L.S.B. of the final address as the program should not finish until the final address has been reversed. If the L.S.B. has not reached 232, the program will branch back to line (f) for another character reversal. When the L.S.B. reaches 232 the program will pass on to line (t) and return to BASIC.

Now to the fun bit, the *Exclusive OR* logical operator. (This is one of three logical operators covered in Appendix A.)

EXCLUSIVE OR

Logical operators work with the bits in binary numbers. Each bit in one number operates with the bit in the same position in the number held in the accumulator. The result of this operation is return to the accumulator.

With respect to any two bits, the *Exclusive OR* operation says that if bit A is 1 *OR* if bit B is 1 then the result of that bit position is 1. If both A and B are 1 or if both A and B are 0, then the result is 0.

For example

Space code 32 decimal	binary representation	00100000
Logically Exclusive OR'd with 128	binary representation	10000000
Result decimal 160	binary representation	10100000

Again:

Reverse space code 160	binary representation	10100000
Logically Exclusive OR'd with 128	binary representation	10000000
Result decimal 32	binary representation	00100000

Try this yourself with other screen character code numbers. Notice how this explains how SYS 828 reverses the screen to start with but also returns it to normal when it's called again.

6

Scrolling the Screen

In this chapter you will scroll the screen — right and left, up and down. With scrolls you can move instructions, adventure information, city skylines and backgrounds sideways or up and down on the screen. You may wish to scroll only a few lines or as much as the whole screen. Often, however, it is the moving of only part of the screen which makes a games presentation look so good. With the ideas in the chapter you'll be able to write programs to do most of these things.

LEFT TO RIGHT SCROLL — Whole Screen

Program # 10

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 169,230,133,251,169,7,133,252,
    169,230,133,253,169,219,133,254
42 DATA 162,0,169,32,157,0,4,157,200,
    4,157,144,5,157,88,6,157,32,7
```

```
44 DATA 138,24,105,40,170,201,200,208,
    230,160,0,169,14,145,253,177,251
46 DATA 200,145,251,165,253,56,233,1,
    133,253,165,254,233,0,133,254,165
48 DATA 251,56,233,1,133,251,165,252,
    233,0,133,252
49 DATA 201,3,208,215,165,251,201,255,
    208,209,96,-1
50 POKE53270,PEEK(53270)AND247
55 FORN=1TO24:PRINT"QWERTYUIOPASRUNJHGFD
QWERTYQWERTY":NEXT
60 SYS828:FORN=1TO50:NEXT:GOTO60
```

When you RUN this program, the characters you enter at line 55 will fill the screen and then scroll off the screen from left to right. But something else will also happen — something you'll have to be sharp-eyed to recognise. *The screen display area will become smaller.*

The machine language program makes use of a nice graphic facility available on the C-64 — that is, the facility instantly to reduce the screen display area from 40 rows wide to 38 rows wide. The two outside rows still exist but are not displayed on the screen.

It's an extremely useful facility to have available when you're scrolling characters into the left or right hand sides of the screen because it avoids the nasty flashes and jumps that accompany the introduction of characters into the first and last screen columns.

This masking of columns 0 and 39 is achieved with the following BASIC statement:

POKE 53270, PEEK(53270)AND 247

Generally speaking, the best place for this line is in the BASIC program itself — before you call the machine language program. This is especially so if your program starts with a scrolling screen. You can, however, incorporate it within the machine language program if you want to. Here are the instructions in machine code.

Load accumulator with contents of of address 53270	PEEK (53270)

Logically AND accumulator with 247	AND 247
Store contents of accumulator in address 53270	POKE 53270

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 230	169,230	LDA 230
(b)	Store contents of accumulator in address 251	133,251	STA 251
(c)	Load accumulator with 7	169,7	LDA 7
(d)	Store contents of accumulator in address 252	133,252	STA 252
(e)	Load accumulator with 230	169,230	LDA 230
(f)	Store contents of accumulator in address 253	133,253	STA 253
(g)	Load accumulator with 219	169,219	LDA 219
(h)	Store contents of accumulator in address 254	133,254	STA 254
(i)	Load X register with 0	162,0	LDX 0
(j)	Load accumulator with 32	169,32	LDA 32
(k)	Store contents of accumulator in address 1024 + X	157,0,4	STA 1024,X
(l)	Store contents of accumulator in address 1224 + X	157,200,4	STA 1224,X
(m)	Store contents of accumulator in address 1242 + X	157,144,5	STA 1424,X
(n)	Store contents of accumulator in address 1624 + X	157,88,6	STA 1624,X
(o)	Store contents of accumulator in address 1824 + X	157,32,7	STA 1824,X
(p)	Transfer contents of X register to accumulator	138	TXA
(q)	Clear carry flag	24	CLC
(r)	Add 40 to contents of accumulator	105,40	ADC 40

(s)	Transfer contents of accumulator to X register	170	TAX
(t)	Compare contents of accumulator with 200	201,200	CMP
(u)	Branch on condition that result is *not* 0	208,230	BNE
(v)	Load Y register with 0	160,0	LDY 0
(w)	Load accumulator with 14	169,14	LDA 14
(x)	Store contents of accumulator in address contained in address 251 indexed by Y register	145,253	STA (251),Y
(y)	Load accumulator with contents of address contained in address 251 indexed by Y register	177,251	LDA (251),Y
(z)	Increment contents of Y register by 1	200	INY
(aa)	Store contents of address contained in address 251 indexed by Y register	145,251	STA (251),Y
(bb)	Load the accumulator with contents of address 253	165,253	LDA 253
(cc)	Set carry flag	56	SEC
(dd)	Subtract 1 from contents of accumulator	233,1	SBC 1
(ee)	Store contents of accumulator in address 253	133,253	STA 253
(ff)	Load accumulator with contents address 254	165,254	LDA 254
(gg)	Subtract 0 from contents of accumulator	233,0	SBC 0
(hh)	Store contents of accumulator in address 254	133,254	STA 254
(ii)	Load accumulator with contents of address 251	165,251	LDA 251
(jj)	Set carry flag	56	SEC
(kk)	Subtract 1 from contents of accumulator	233,1	SBC 1
(ll)	Store contents of accumulator in address 251	133,251	STA 251

(mm)	Load accumulator with contents of address 252	165,252	LDA 252
(nn)	Subtract 0 from contents of accumulator	233,0	SBC 0
(oo)	Store contents of accumulator in address 252	133,252	STA 252
(pp)	Compare contents of accumulator with 3	201,3	CMP 3
(qq)	Branch on condition that result is *not* 0	208,215	BNE
(rr)	Load accumulator with contents of address 251	165,251	LDA 251
(ss)	Compare contents of accumulator with 255	201,255	CMP 255
(tt)	Branch on condition that result is *not* 0	208,209	BNE
(uu)	Return to BASIC	96	RTS

Explanation of the Machine Language Program

Figure 6.1 at the end of the chapter illustrates how the machine language program achieves the scrolling effect.

The program starts first at the bottom right-hand corner of the screen taking out whatever character is in the *second last screen address* and putting that character into the last screen address. Then the program progresses up the screen, loading a character into the accumulator from the lower address and storing it back into the next higher address. The final step occurs when the content of the first screen address is loaded into the next higher address on the top row. At that point you can see that the whole screen display has been moved over one address to the right.

Since we have to move 999 bytes (you don't have to move the bottom, last address on the screen), we use the same method as we did in the previous chapter — zero page indirect indexed addressing mode.

Notice how the diagram shows the data moving from the end of one row to the start of the next row. If you have a full screen of material or even some rows that are full, they will repeat from the left-hand side of the screen as it all scrolls across.

In lines (a) to (h) the zero page addresses are filled up with the actual starting addresses for the program. The first screen address will be 2022 (L.S.B. = 230

and M.S.B. = 7). This program also deals with color and the corresponding address is 55834 (L.S.B. = 230 and M.S.B. = 219). Notice how these values are first loaded into the accumulator and then stored in the four zero page addresses, 251 to 254.

Lines (i) to (u) load code 32 (the code for space) into each of the addresses in the screen's first column — the one on the far left.

Line (i) loads the X register with 0. The X register serves both as the loop counter and the index for the various screen addresses.

Line (j) loads the accumulator with 32, the code for space.

Lines (k) to (o) store the contents of the accumulator (code 32) in the starting address of each of the 5 blocks of 5 rows into which we've grouped the 25 screen rows. The grouping is necessary, of course, because the actual addreses increase by 40 with each new row and we're using the absolute indexed by X register addressing mode in this part of the program. So the starting address for each of the 5 blocks of rows is indexed by the X register.

Line (p) commences the routine to increase by 40 at each cycle of the program. You can only do arithmetic operations with the contents of the accumultor, so line (p) transfers the contents of the X register to the accumulator.

Line (q) clears the carry flag in readiness for the addition instruction.

Line (r) adds 40 to the contents of the accumulator.

Line (s) transfers the contents of the accumulator to the X register where it will be used again to index the screen addresses.

Line (t) compares the value held in the accumulator with 240.

Line (u) passes the program on to the next instruction when the requirements for the conditional branch are met — ie, when the five passes have placed a code 32 in each of the addresses down the left hand side of the screen. In other words, when X = 200 (5x40).

Lines (v) to (z) do the actual work of moving the contents of the addresses.

Line (v) loads the Y register with 0. At this stage the Y register has no effect on the actual address through indexing.

Line (w) loads the accumulator with 14, the code for light blue, the regular print color. This color code is then stored in the screen memory address for color.

Line (x) loads the accumulator with the contents of the screen address obtained indirectly from zero page address 251. As Y is 0 at this point, it does not 're-index' this address.

Line (y) increments the Y register by 1.

Line (z) stores the contents of the accumulator in the next higher address. It does this because the base address held in 251/252 is now being indexed by Y = 1 (from line (y)).

In lines (aa) to (gg) and (hh) to (nn) the program changes the addresses held in zero page addresses 253/254 and 251/252. The first group covers the color code addresses and the second, the screen addresses. This is essentially the same routine as we described in the previous chapter except that the addresses are being lowered. Therefore instead of adding 1 we subtract 1 from each address on each pass of the program.

Subtraction

There is only one instruction for subtraction in the 6510 set and like addition, it is *subtraction with carry*. This means that when your try to subtract from a particular byte address a number larger than it currently holds, you don't actually get a negative number. The 6510 handles it the way we do regular subtraction — that is, it 'borrows' 1. It borrows the 1 from the carry flag, restoring the carry flag to zero (if it happened to be set at 1 when the subtraction took place).

If the carry flag is 0 when the subtraction takes place, however, the instruction to subtract takes 1 from the result in addition to whatever else has been subtracted.

In the same way as for addition, you should put the carry flag in the condition required to give the answer you want, before you do the actual subtraction. Appendix A covers this aspect in detail.

It's important also to note that as the L.S.B. decreases below 0, the M.S.B. decreases by 1 — starting the L.S.B. off at 255 again.

Lines (cc) and (jj) set the carry flag to 1 before the subtraction takes place. Note, however, that the set carry flag instruction is *not* used with the second subtraction in each group — lines (gg) and (nn). This is because when the first subtraction has cleared the carry flag and you want the 1 from the first subtraction to be subtracted from the second address. There's not much point otherwise in having an instruction to subtract 0!

Lines (pp) to (uu) are similar to lines (p) to (u) in the program we used to reverse the whole screen in the previous chapter. In other words, the program is allowed to branch or loop until the address in addreses 251/252 reaches the address where we want the program to stop. This is one less than the first screen address. The program handles the first screen address (1024) but passes on at the next address (1023) where L.S.B. = 255 and M.S.B. = 3.

RIGHT TO LEFT SCROLL — Part Screen

The method we use for this scroll is quite different from the method of the previous one and the scroll itself handles only part of the screen. Actually, you can handle the whole screen with this part method simply by adding further parts to the program and selecting the correct addresses. On the other hand, you can use it to scroll only one row if that suits your display. Type in Program Number 11.

Program # 11

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 162,0,189,65,5,157,64,5,169,14,
    157,64,217,232,224,240,208,240
42 DATA 162,0,189,49,6,157,48,6,169,14,
    157,48,218,232,224,240,208,240
44 DATA 162,0,169,32,157,103,5,157,87,
    6,138,24,105,40,170,201,240
```

```
46 DATA 208,239,96,-1
50 FORN=1TO24:PRINT"QWERTYUIOPASDFGHJKLZ
QWERTYXCVBNMQWERTY":NEXT
60 SYS828:GOTO60
```

When you RUN this program the characters you entered in line 50 will fill the screen. Then a section just below centre screen will scroll away from right to left. I haven't used the 38 column mode here but you can add it to the BASIC program if you want to.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load X register with 0	162,0	LDX 0
(b)	Load accumulator with contents of address 1345 + X	189,65,5	LDA 1345,X
(c)	Store contents of accumulator in address 1344 + X	157,64,5	STA 1344,X
(d)	Load accumulator with 14	169,14	LDA 14
(e)	Store contents of accumulator in address 55616 + X	157,64,217	STA 55616,X
(f)	Increment contents of X register by 1	232	INX
(g)	Compare contents of X register with 240	224,240	CPX 240
(h)	Branch on condition that result is *not* 0	208,240	BNE
(i)	Load X register with 0	162,0	LDX 0
(j)	Load accumulator with contents of address 1585 + X	189,49,6	LDA 1585,X
(k)	Store contents of accumulator in address 1584 + X	157,48,6	STA 1584,X
(l)	Load accumulator with 14	169,14	LDA 14
(m)	Store contents of accumulator in address 55856 + X	157,48,218	STA 55816,X

(n)	Increment contents of X register by 1	232	INX
(o)	Compare contents of X register with 240	224,240	CPX 240
(p)	Branch on condition that result is *not* 0	208,240	BNE
(q)	Load X register with 0	162,0	LDX 0
(r)	Load accumulator with 32	169,32	LDA 32
(s)	Store contents of accumulator in address 1383 + X	157,103,5	STA 1383,X
(t)	Store contents of accumulator in address 1623 + X	157,87,6	STA 1623,X
(u)	Transfer contents of X register to accumulator	138	TXA
(v)	Clear carry flag	24	CLC
(w)	Add 40 to contents of accumulator	105,40	ADC 40
(x)	Transfer contents of accumulator to X register	170	TAX
(y)	Compare contents of accumulator with 240	201,240	CMP 240
(z)	Branch on condition that result is *not* 0	208,239	BNE
(aa)	Return	96	RTS

Explanation of the Machine Language Program

This program falls neatly into three separate parts. Lines (a) to (h) scroll the first six rows, starting at row 13, just below the centre of the screen. Lines (i) to (p) scroll the next six rows. Lines (q) to (aa) put code 32 (space) into the column on the far right of the screen. The absolute indexed by X register is used throughout and this explains why we've grouped the rows to be scrolled into two blocks of six. 6 × 40 is the maximum we can fit within the limit of 255 that can be held in a register.

Line (a) loads the X register with 0. This is the loop counter for this section and will also index the addresses.

Line (b) loads the accumulator with the contents of address 1345 which will be progressively indexed by the X register. Notice that these two lines move the contents back one address.

Line (d) loads the accumulator with 14 which is the color code for light blue, the regular print color.

Line (e) stores the color code 14 in the corresponding color code address — 55616.

Line (f) increments the X register by 1 at each cycle of the program.

Line (g) compares the value held in the X register with 240.

Line (h) holds the conditional branch instruction that allows the program to pass on when the X register reaches 240.

Lines (i) to (p) repeat the tasks of lines (a) to (h), simply changing the addresses in order to scroll the next six rows of the screen. Another way of achieving the same effect would be to use the method we used to draw the double-sized block in chapter 4. In most cases that will also be a perfectly satisfactory way to go about it — but with scrolls I prefer to do it in two separate sections like this.

Line (q) loads the X register with 0. This will be the loop counter for this last section and will also index the addresses.

Line (r) loads the accumulator with 32, the code for space.

Lines (s) and (t) store this code in the two addresses, 1383 and 1623, both indexed by X.

Line (u) transfers the contents of the X register to the accumulator in preparation for the addition that's to take place in line (w). Because going down a line means increasing the addresses by 40, we're going to have to add 40 to the X register. We can only add and subtract in the accumulator, though, with the result being returned to the accumulator.

Line (v) clears the carry flag in preparation for the addition.

Line (w) adds 40 to the accumulator.

Line (x) transfers the contents of the accumulator back to the X register.

Line (y) compares the contents of the X register with 240 in preparation for the conditional branch instruction in the following line.

Line (z) allows the program to pass on to its conclusion when the result of the comparison between the contents of the X register and 240 returns a zero.

Line (aa) returns the program to BASIC.

UPWARD SCROLLING — Whole Screen

This scroll moves the whole screen upwards, from bottom to top. We will use the zero page indirect indexed by Y addressing mode again but introduce a different method to change the addresses contained in the zero page addresses. Type in Program Number 12.

Program # 12

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 169,0,133,251,169,4,133,252,
    169,0,133,253,169,216,133,254
42 DATA 160,40,177,251,160,0,145,251,
    169,14,145,253,230,253,208,2,230
44 DATA 254,230,251,208,2,230,252,165,
    252,201,7,208,226
46 DATA 165,251,201,191,208,220,96,-1
50 FORN=1TO24:PRINT"QWERTYUIOPASDFGHJKLZ
XCVBNMQWERTY":NEXT
60 SYS828:FORN=1TO100:NEXT:GOTO60
```

When you RUN this program the characters you entered in line 50 will fill the screen. Then the screen will scroll upwards and clear to the top.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 0	169,0	LDA 0
(b)	Store contents of accumulator in address 251	133,251	STA 251
(c)	Load accumulator with 4	169,4	LDA 4
(d)	Store contents of accumulator in address 252	133,252	STA 252
(e)	Load accumulator with 0	169,0	LDA 0
(f)	Store contents of accumulator in address 253	133,253	STA 253
(g)	Load accumulator with 216	169,216	LDA 216
(h)	Store contents of accumulator in address 254	133,254	STA 54
(i)	Load Y register with 40	160,40	LDY 40
(j)	Load accumulator with contents of address held in address 251	177,251	LDA (251),Y
(k)	Load Y register with 0	160,0	LDY 40
(l)	Store accumulator with contents of address held in address 251	145,251	STA (251),Y
(m)	Load accumulator with 14	169,14	LDA 14
(n)	Store accumulator with contents of address held in address 253	145,253	STA (253),Y
(o)	Increment contents of address 253 by 1	230,253	INC 253
(p)	Branch on condition that result is *not* 0	208,2	BNE
(q)	Increment contents of address 254 by 1	230,254	INC 254
(r)	Increment contents of address 251	230,251	INC 251
(s)	Branch on condition that result result is not 0	208,2	BNE
(t)	Increment contents of address 252	230,252	INC 252

(u)	Load accumulator with contents of address 252	165,252	LDA 252
(v)	Compare contents of accumulator with 7	201,7	CMP 7
(w)	Branch on condition that result is *not* 0	208,226	BNE
(x)	Load accumulator with contents of address 251	165,251	LDA 251
(y)	Compare contents of accumulator with 191	201,191	CMP 191
(z)	Branch on condition that result is *not* 0	208,220	BNE
(aa)	Return	96	RTS

Explanation of the Machine Language Program

Lines (a) to (h) load the starting addresses of the screen and color codes into the appropriate zero page addresses. It's the same procedure we used to initiate our earlier programs.

Lines (i) to (n) do the actual work of scrolling. Figure 6.3 (at the end of the chapter) illustrates how the machine language program achieves the scrolling effect. It starts out by taking the contents of the first address on the second row and placing it in the first address in the first row. These two addresses, of course, are 40 places apart. The first address, 1024, is stored in zero page addresses 251/252 and the separation of the two addresses is achieved by indexing with the Y register.

Line (i) loads the Y register with 40.

Line (j) loads the accumulator with the contents of the address held in addresses 251/252 *indexed* by Y, ie, 1064.

Line (k) loads the Y register with 0.

Line (l) stores the contents of the accumulator in the address *held in* addresses 251/252 — now indexed by Y = 0. That is, 1024.

Line (m) loads the accumulator with 14, the color code for light blue.

Line (n) stores this in the address *held in* addresses 253/254. Note that Y is still 0.

Line (o) increments the contents of address 253 by 1 — ie, moves the screen address held in 253, one place on. If this increment instruction results in a zero in the address, the zero flag will be set to 1 and the conditional branch instruction in line (p) will allow the program to pass on to line (q) and to increment the contents of 254 by one. It's worth pausing a moment to reflect on what's actually happening here. The color code address is held with its L.S.B. in 253 and its M.S.B. in 254. While the L.S.B. in 253 is not zero, the conditional branch will take the program over line (q) to (r), a jump of two bytes, which starts the same increment of the address in 251/252. When the L.S.B. in 253 reaches its maximum of 255, however, the next increment of 1 will return it to zero and require its M.S.B. to increase by 1. That happens in line (q).

Lines (r) to (t) repeat the procedure carried out in lines (o) to (q) — but for the screen code address held in 251/252.

Lines (u) to (z) contain the instructions the program needs to check whether it should stop.

Line (u) loads the accumulator with the contents of address 252. Address 252 contains the M.S.B. of the screen address which, at the end of the program, will be 1983 (M.S.B. 7).

Lines (v) and (w) create the loop that sends the program back to its beginning until the contents of 252 is equal to 7.

Lines (x), (y) and (z) repeat the procedure for the L.S.B. of 1983, *plus one* (ie, 192). The plus one is required because we want the program to jump back and process this last address.

Line (aa) returns the program to BASIC.

DOWNWARD SCROLL — Part Screen

This program scrolls downwards a section of the screen commencing at row 13. Type in Program Number 13.

Program # 13

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 162,240,189,7,6,157,47,6,202,
    208,247,162,240,189,23,5,157,63,5
42 DATA 202,208,247,162,40,169,32,157,
    23,5,202,208,248,96,-1
50 FORN=1TO12:PRINT"QWERTYUIOPASDFGHJKLZ
XCVBNMQWERTY"
55 PRINT"12345678901234567890123456789O"
:NEXT
60 SYS828:FORN=1TO100:NEXT:GOTO60
```

When you RUN this program the screen will fill with the characters you entered in lines 50 and 60. Then a section of the screen will scroll downwards leaving it clear.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load the X register with 240	162,240	LDX 240
(b)	Load accumulator with contents of address 1543 + X	189,7,6	LDA 1543,X
(c)	Store contents of accumulator in address 1583 + X	157,47,6	STA 1583,X
(d)	Decrement X register by 1	202	DEX
(e)	Branch on condition that result is *not* 0	208,247	BNE
(f)	Load the X register with 240	162,240	LDX 240

(g)	Load accumulator with contents of address 1303 + X	189,23,5	LDA 1303,X
(h)	Store contents of accumulator in address 1343 + X	157,63,5	STA 1343,X
(i)	Decrement X register by 1	202	DEX
(j)	Branch on condition that result is *not* 0	208,247	BNE
(k)	Load X register with 40	162,40	LDX 40
(l)	Load accumulator with 32	169,32	LDA 32
(m)	Store contents of accumulator in address 1303 + X	157,23,5	STA 1303,X
(n)	Decrement X register by 1	202	DEX
(o)	Branch on condition that result is *not* 0	208,248	BNE
(p)	Return to BASIC	96	RTS

Explanation of the Machine Language Program

This program falls neatly into two separate parts and illustrates again how the technique keeps the sequence of addresses in the right order. Of course, you can use this same procedure to scroll anything from a single row to a full screen, simply by changing the addresses and the number of 'repeats' in the program.

Line (a) loads the X register with 240. It will serve both as loop counter and address index.

Lines (b) and (c) load the contents of one address into the accumulator and then store it back in another address 40 addresses further on — ie, in the same position on the next row.

Line (d) decrements the contents of the X register by 1 to move both addresses progressively up the screen.

Line (e) allows the program to pass onto the next instruction when the result of this decrement is zero.

Lines (f) to (j) repeat the procedure for the next block of addresses.

Line (k) loads the X register with 40, setting it up as the loop counter and address index for the individual addresses across the rows.

Line (l) loads the accumulator with 32, the code for space.

Line (m) stores the contents of the accumulator in address 1303 + X — ie, places a space in each of the addresses in the first row of the area to be scrolled.

Line (n) decrements the X register by 1 and when X = 0, the conditional branch instruction in line (o) allows the program to pass on to the Return to BASIC instruction in line (p).

Note that each of these scroll programs moves only one row or column of material at each pass through the program.

Suggestions on Scrolls

For the most part, you'll probably only ever want to scroll a small part of the screen at any one time. If that's the case and you need to include a row of spaces (code 32), it's easier and more economical of memory to store the 32s in specific addresses rather than to use a loop. Your machine language program would include lines like these:

(a) Load the accumulator with 32	LDA 32
(b) Store contents of accumulator in (address)	STA (address)
(c) Store contents of accumulator in (address + 1)	STA (address + 1)
(d) Store contents of accumulator in (address + 2)	STA (address + 2)

And so on

Here is an idea to think about. You now have all the knowledge you need and separately, you have done all the things required. How about writing your own program to scroll a block on the screen? Say, a little window, 10 by 10. Going upwards, you move the second row into the first, the third into the second and so on. You know how to retain the starting address. You can add 40 to X to come back to the starting address. It works quite easily — try it! You will really impress your friends.

IMPORTANT

Up until now we've been looking at a variety of machine language programs each of which has been designed for a specific purpose. It's important to understand, though, that these programs (with only a little modification) can do other work. Learning how the methods work and getting practice with the instructions is important but neither of those things will give you much satisfaction unless you can use the various techniques of machine language programming to design programs of your own to do the things you want to make happen.

So let's look *behind* the obvious purpose of the programs in this chapter. On the surface, we've been concerned with scrolling areas of the screen. In reality, though, we've actually been working with a large block of *data* (a full screen holds a thousand bytes). And that data can be anything we like to make it.

Suppose, for example, that you're designing a game and at various times throughout the game you want to recall a diagram or a map or a maze to the screen. Those graphics, of course, are data and if they fill the screen, they are a large block of data. The techniques we've developed in the course of this chapter don't need much modification to handle the new application.

Notice the similarities between Program Number 14 and the programs we've already used in this chapter.

> NB This program introduces the use of a large block of RAM (4K) freely available to you for machine languge programs or data storage. It occurs from 49152 to 53247 and is outside the BASIC system.

Program # 14

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 169,0,133,251,169,192,133,252,
    169,0,133,253,169,4,133,254
```

```
42 DATA 160,0,177,251,145,253,230,253,
    208,2,230,254,230,251,208,2
44 DATA 230,252,165,252,201,195,208,232,
    165,251,201,231,208,226,169,0
46 DATA 133,251,169,216,133,252,169,3,
    145,251,230,251,208,2,230,252
48 DATA 165,252,201,219,208,240,165,251,
    201,231,208,234,96,-1
50 FORN=1TO24:PRINT"QWERTYUIOPASDFGHJKLZ
XCVBNMQWERTY":NEXT
55 FORN=0TO999:POKE49152+N,PEEK(1024+N):
NEXT
```

When you RUN this program there will be a short delay while the machine language codes are POKEd into the cassette buffer at 828. Then the screen will fill with the characters you entered in line 50. There will then be a relatively long delay while the 1000 bytes of the screen are moved from the screen addresses to RAM starting at 49152. This movement of data is being done in BASIC in line 55. This delay is a good illustration of the difference in speed between BASIC and machine language.

Now clear the screen with CLR/HOME and SHIFT and type in SYS828 and RETURN. The screen will immediately fill with the characters from line 50 but this time in cyan. The machine language routine contained in the DATA lines has transferred all the characters back from the block starting at 49152 and in addition has loaded all the screen color code addresses from 55296 with 3, the color code for cyan!

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 0	169,0	LDA 0
(b)	Store accumulator in address 251	133,251	STA 251
(c)	Load accumulator with 192	169,192	LDA 192
(d)	Store accumulator in address 252	133,252	STA 252
(e)	Load accumulator with 0	169,0	LDA 0
(f)	Store accumulator in address 253	133,253	STA 253
(g)	Load accumulator with 4	169,4	LDA 4
(h)	Store accumulator in address 254	133,254	STA 254
(i)	Load Y register with 0	160,0	LDY 0
(j)	Load accumulator with contents of address contained in address 251	177,251	LDA (251),Y
(k)	Store accumulator in address contained in address 253	145,253	STA (253),Y
(l)	Increment contents of address 253	230,253	INC 253
(m)	Branch on condition that result is *not* 0	208,2	BNE
(n)	Increment contents of address 254	230,254	INC 254
(o)	Increment contents of address 251	230,251	INC 251
(p)	Branch on condition that result is *not* 0	208,2	BNE
(q)	Increment contents of address 252	230,252	INC 252
(r)	Load accumulator with contents of address 252	165,252	LDA 252
(s)	Compare accumulator with 195	201,195	CMP 195
(t)	Branch on condition that result is *not* 0	208,232	BNE
(u)	Load accumulator with contents of address 251	165,251	LDA 251
(v)	Compare accumulator with 231	201,231	CMP 231
(w)	Branch on condition that result is *not* 0	208,226	BNE

(x)	Load accumulator with 0	169,0	LDA 0
(y)	Store accumulator in address 251	133,251	STA 251
(z)	Load accumulator with 216	169,216	LDA 216
(aa)	Load accumulator with 252	133,252	STA 252
(ab)	Load accumulator with 3	169,3	LDA 3
(ac)	Store accumulator in address contained in address 251	145,251	STA (251),Y
(ad)	Increment contents of address 251	230,251	INC 251
(ae)	Branch on condition that result is *not* 0	208,2	BNE
(af)	Increment contents of address 252	230,252	INC 252
(ag)	Load accumulator with contents of address 252	165,252	LDA 252
(ah)	Compare accumulator with 219	201,219	CMP 219
(ai)	Branch on condition that result is *not* 0	208,240	BNE
(aj)	Load accumulator with contents of address 251	165,251	LDA 251
(ak)	Compare accumulator with 231	201,231	CMP 231
(al)	Branch on condition that result is *not* 0	208,234	BNE
(am)	Return	96	RTS

Explanation of the Machine Language Program

This program uses the same system we employed in the upward scroll of the whole screen. It shifts a block of memory by moving the first address and then letting a loop take care of the rest. In fact, that's all there is to moving any block of memory.

The program itself falls neatly into five separate parts and uses the zero page indirect indexed by Y addressing mode.

Lines (a) to (h) set up the addresses to be handled in addresses 251/254. These addresses are the RAM memory starting at 49152 and the screen locations starting at 1024.

Lines (i) to (k) handle the main action. In line (i) the Y register is loaded with 0 so it won't re-index the addresses. Then in line (j) the accumulator is loaded with the contents of the address obtained indirectly from address 251/252 —

ie, the RAM memory. Line (k) then stores the contents of the accumulator in the address obtained indirectly from address 253/254 — ie, the screen character location, starting at 1024.

Lines (l) to (n) increment the address of the RAM memory at each cycle of the program.

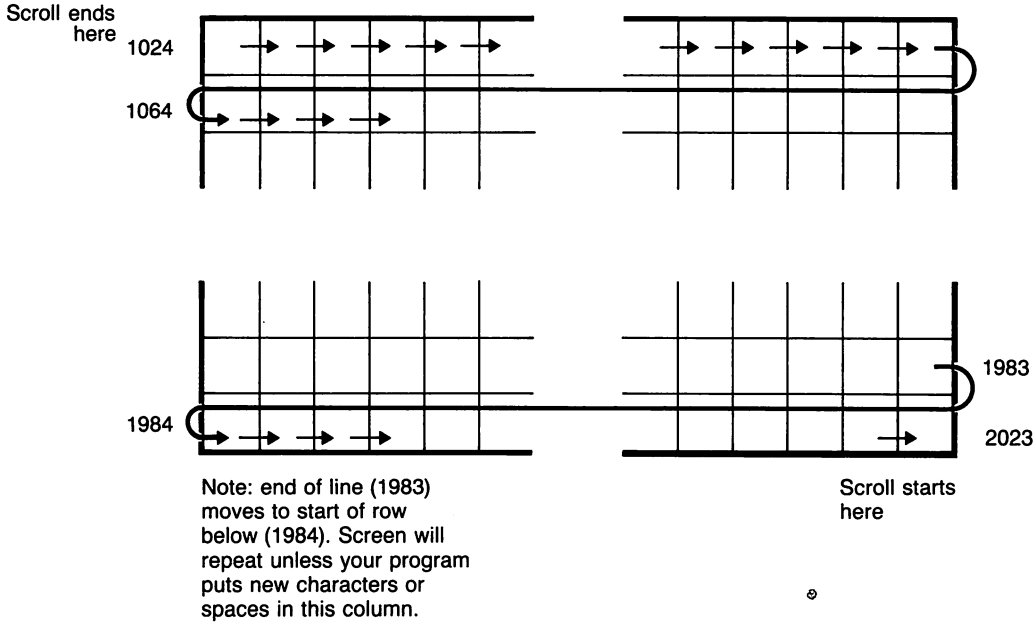
Lines (o) to (w) increment the screen address at each cycle of the program, keep count of the number of loops and stop the program when all the data has been transferred.

Lines (x) to (am) create a similar sort of loop counting program to load color code 3 into the screen addresses for color starting at 55296. This section also counts the number of loops and stops the program when all the screen color addresses are filled with color code 3. Notice that it also uses the zero page addresses 251/252 to store the color code address 55296. There's no problem about this because after line (w), the program is finished with the orginal addresses that were stored in the zero page addresses.

In the 4K of RAM available from 49152, you can store at least four full screens of data. That means that if you wanted to, you could transfer the first screen as we've done here and then POKE in BASIC (or LOAD in machine language) a whole set of new addresses in the LDA lines of the program and then store them away too.

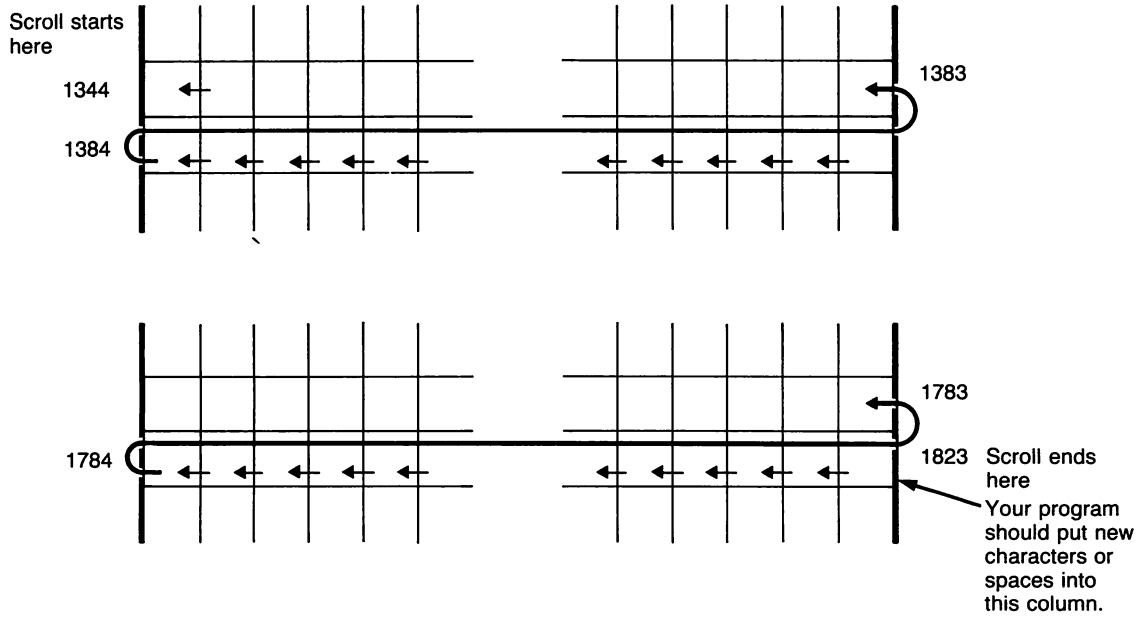
(Figure 6.1)

Left to Right Scroll — Whole Screen



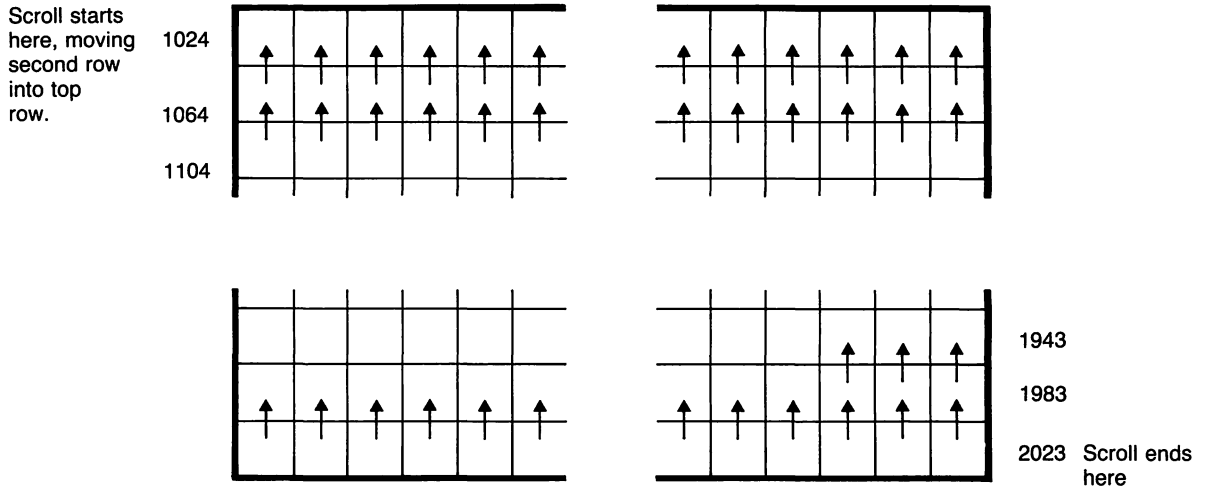
(Figure 6.2)

Right to Left Scroll — Part Screen



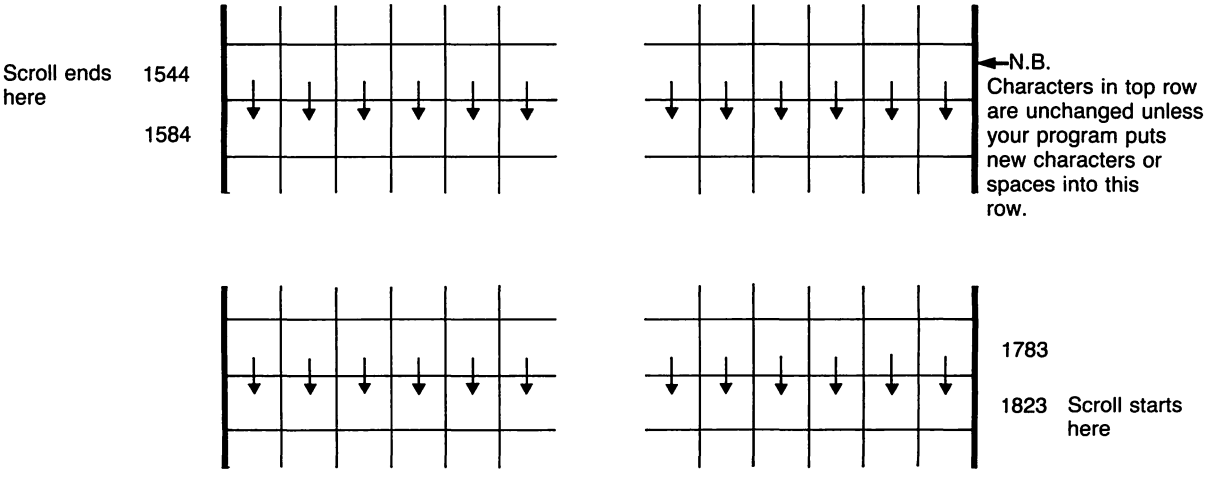
(Figure 6.3)

Upward Scroll — Whole Screen



(Figure 6.4)

Downward Scroll — Part Screen



7

Animating Your Own Characters

This chapter will show you how to produce some really good animation of characters of your own design.

You can make up some excellent games moving asterisks and those sorts of things around the screen. But it is much more fun to use the facilities of the C-64 to design your own characters and move *them* around the screen. Furthermore there is no reason why your characters should be limited to the size of a normal C-64 character.

You can design front and rear end characters for spaceships, cars or battleships. In fact, you can use as many parts (normal characters) as you like to build up really good looking spaceships or whatever you want.

In this set of programs, I have used 6 character spaces to build up a nice sized little man. By using a further 5 character spaces I've made the man really look as though he is running on the screen.

But first let's see how we go about producing a character of our own. Bear in mind that they will be high resolution characters: since you make them up with individual dots on the screen you get all the detail of a high resolution figure. The only problem is that you can't *move* them dot by dot. You can only move them in the regular rows and columns. Nevertheless it does create a really fine display and looks very effective.

YOUR C-64's CHARACTER SET

Built into the C-64's ROM is the complete set of standard characters which can be printed out on the screen from the keyboard. Each of these characters is represented by a set of 8 consecutive numbers. While we might think of these numbers as decimal numbers they are, of course, (to the C-64) a set of binary numbers made up of groups of 0's and 1's.

Each of the characters on the screen is made up of 8 rows by 8 columns, and each of the binary numbers in the character represents one *row* of the character. When the C-64 sees a 1 in a number it puts a dot on the screen and when it sees a 0 it leaves the spot blank. Here is a picture of the character C built up in this way:

BINARY		DECIMAL
00111100	=	60
01100XX0	=	102
01100000	=	96
01100000	=	96
01100000	=	96
01101110	=	110
00111100	=	60
00000000	=	0

128	64	32	16	8	4	2	1
		X	X	X	X		
	X	X			X	X	
	X	X					
	X	X					
	X	X					
	X	X		X	X	X	
		X	X	X	X		

As we said before, this character set is built into the ROM of the C-64 and so there is no way that it can be changed. But in the VIC II chip, which controls all the graphics, there is a register which tells it where to find the various character data. So if we put our own set of characters in RAM it is possible to tell the VIC II where to find it and put it on the screen instead of the set in ROM.

Our own characters must be related, however, to some of the standard C-64 characters. The standard characters and their codes are shown in Appendix F. The codes go from 0 to 127. We cannot use a different set of codes for our own special characters because there aren't enough numbers. Even the numbers 128 to 256 are taken up because adding 128 to the codes 0 to 127 produces the reverse of the ordinary characters. Therefore we have to substitute our own characters for some of the standard characters so that we can use the standard screen code numbers for them.

Once you've done this, though, you can't switch between the different sets of characters during a program. If you try to, all the characters already on

the screen will simply switch over to the new characters that the screen codes now represent. What we usually do is to move the first 64 characters of *set 1* of the character codes (Appendix F) into the top of our RAM memory. At that point we can decide which ones we don't need and replace them with our own special characters.

The codes in Set 1 represent a total of 512 bytes of memory; ie, 64 characters by 8 bytes each. We've said we're going to hold them at the top of the RAM area which runs from 2048 to 40959. It creates a problem, though, because this is where the C-64 puts BASIC programs and where it starts to store strings from 40959 downwards. The latter is especially a problem. Anything put up here would likely be overwritten — with disastrous results. *Therefore you have to protect this top part of RAM by lowering the top address of RAM.* Try this. Type:

```
PRINT PEEK (55) + 256 * PEEK (56)
```

You will get a reply on the screen of 40960 which C-64 recognises as the byte above the last byte in RAM it can use for BASIC. Addresses 55 and 56 are addresses which hold the systems variable which is this *ramtop* address. Now try this. Type:

```
PRINT PEEK (55), PEEK (56)
```

and you will get the reply 0 and 160. You will recognise these as the Least Significant Byte and Most Significant Byte of the address 40960. Now try this. Type:

```
POKE 52,28: POKE 56,48: CLR (and RETURN)
```

You have now POKEd 48 into the Most Significant Byte. So now when you type:

```
PRINT PEEK (55) + 256 * PEEK (56)
```

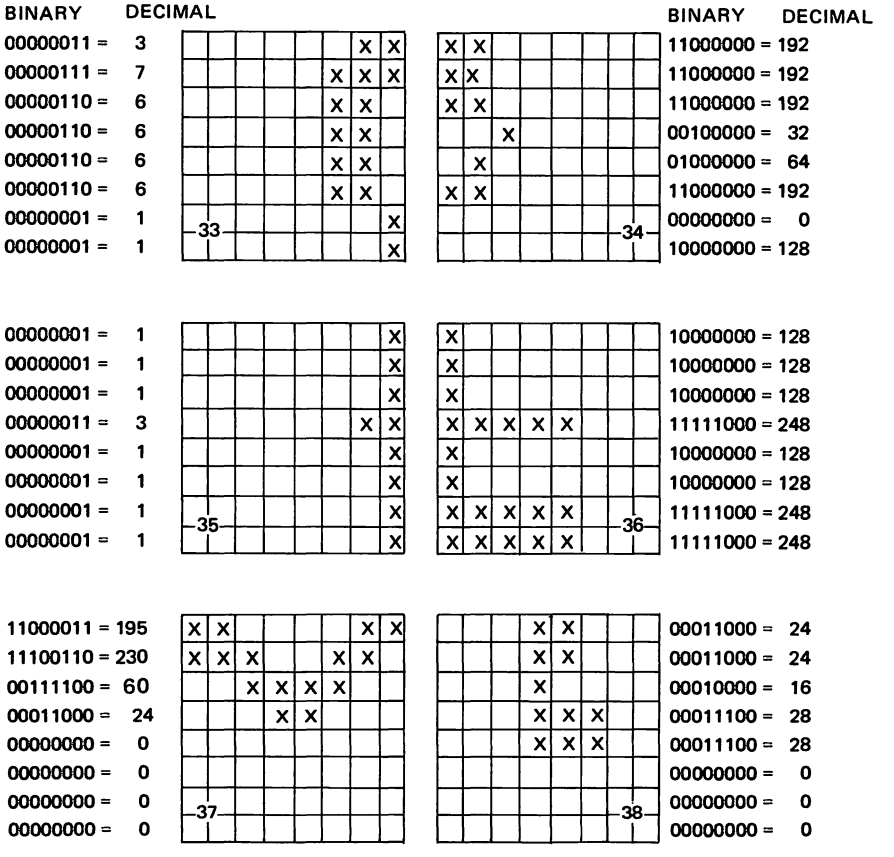
you will get an answer of 12288 which the C-64 recognises as the new ramtop. And anything between 12288 and 40960 is protected from statements like NEW, CLR and LOAD. Incidentally, POKEing 52 with 48 does the same thing with a system variable which tells the C-64 where to start storing strings and this address is always the same as ramtop. Now let's draw a man.

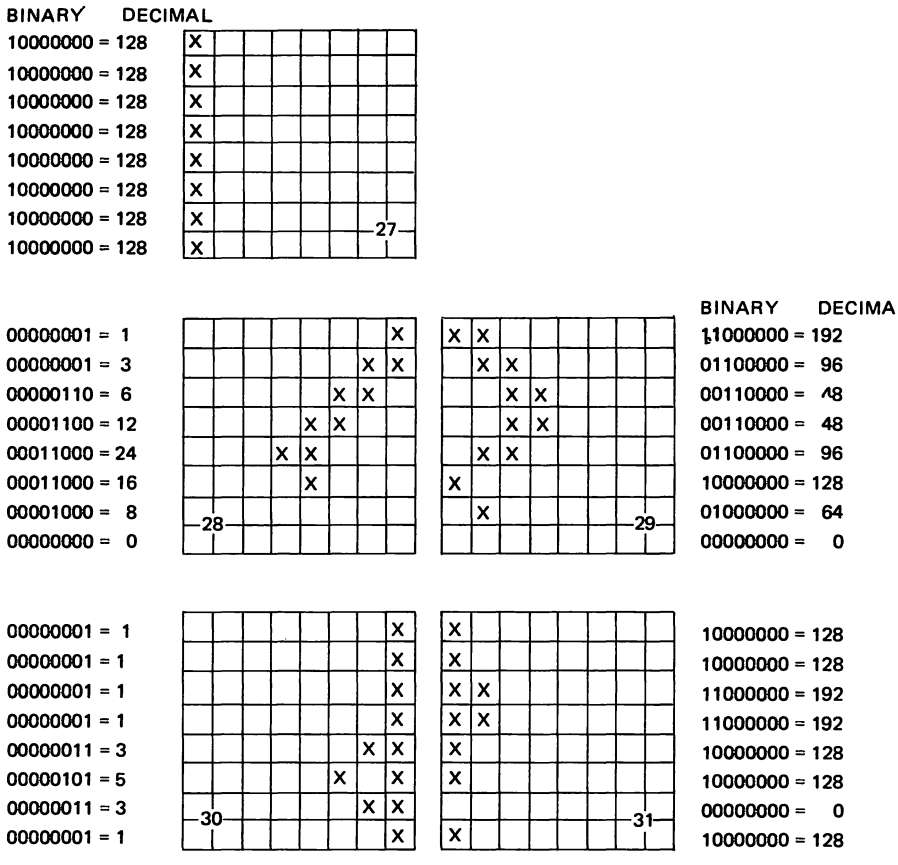
MAKING CHARACTERS

The following diagrams show the designs of the characters we're going to use. They also show which standard character keys we're going to use to call up our own characters.

We're actually going to replace characters 27 to 31 and 33 to 38. If you look at the characters represented by these numbers in the table of screen codes (Appendix F) you'll see that we aren't going to replace anything particularly useful.

Notice that we're not going to replace character 32 (the space) because it's likely that we'll need it.





If you can't follow the conversion from binary numbers to decimal numbers, have a look at Appendix A.

The next step is to transfer the 64 characters in the standard ROM set into the RAM space you just reserved and protected by POKEing 56 with 48. The VIC II gets its standard character set from ROM *images,* not directly from the ROM addresses starting at 53248 where the character set is actually located. This is controlled by the I/O port of the 6510 and while this is operative, the character set is only available to the VIC II. So the first thing we have to do is switch off the I/O port. This makes the ROM at 53248 available to us. We do it with the following BASIC statement (*but don't do it yet!*):

```
POKE 1,PEEK(1)AND251
```

NB. While the I/O port is switched off, no interruptions can be allowed to occur. You turn off the keyboard and any other possible sources of interruption with this BASIC statement (*don't do it yet!*):

```
POKE 56334,PEEK(56334)AND254
```

After you've done these two things, you can move the character information from ROM to the new area above 12288 and then reverse those two BASIC statements with these (*don't do it yet!*):

```
POKE56334,PEEK(56334)OR1
POKE 1,PEEK(1)OR4
```

We've delayed our entry of these commands because they have to be written inside a program. If you use either of the first two statements as direct commands, the machine will go dead as far as you are concerned because you will have no control over the keyboard or the screen. You will have to switch off the computer and start again. Here is the BASIC program that you can type in now. Line 30 transfers the character set from ROM to the RAM addresses starting at 12288.

Program # 15

```
10 POKE56334,PEEK(56334)AND254
20 POKE1,PEEK(1)AND251
30 FORI=0TO511:POKE12288+I,PEEK(53248+I)
:NEXT
40 POKE1,PEEK(1)OR4
50 POKE56334,PEEK(56334)OR1:END
```

Your new characters have to occupy the same places as the standard characters they are replacing. Note that each character address consists of 8 bytes. The character @ (code 0) starts at address 12288 and ends with 12295. You work out the address for any character by multiplying the character code by 8 and

adding the result to the starting address, 12288. Character N, for example, has the code 14. Its address is therefore 12288 + (8x14) = 12400.

Putting our Characters in RAM

I have shown on the diagram which character each of my special little segments will replace. You can now type in Program Number 16 to POKE these characters into RAM. Note that I am not replacing SPACE code 32 (it will be used later) but I have included it in the DATA statement so that the program can run straight through one block of memory.

Program # 16

```
10 FORN=12504TO12599:READ D:POKEN,D:NEXT

20 DATA 128,128,128,128,128,128,128,128
21 DATA 1,3,6,12,24,16,8,0
22 DATA 192,96,48,48,96,128,64,0
23 DATA 1,1,1,1,3,5,3,1
24 DATA 128,128,192,192,128,128,0,128
25 DATA 0,0,0,0,0,0,0,0
26 DATA 3,7,6,6,6,6,1,1
27 DATA 192,192,192,32,64,192,0,128
28 DATA 1,1,1,3,1,1,1,1
29 DATA 128,128,128,248,128,128,248,248
30 DATA 195,230,60,24,0,0,0,0
31 DATA 24,24,16,28,28,0,0,0
```

When you RUN this the new set of characters will be POKEd into RAM memory but you will not see anything else happen because the VIC II is still looking at the ROM image for its characters. The last step consists in

arranging it so that the VIC II will look for the character set in RAM starting at address 12288. Enter as a direct command, this BASIC statement:

```
POKE 53272, (PEEK(53272)AND 240)OR12
```

Now when you type any regular letters or numbers they will appear as normal on the screen because they are in the character set moved into RAM. But try typing in the character codes 27 to 38 (you can see what they are from Appendix F) and you will see the segments of my little man appear on the screen.

That is about all there is to programming your own special characters. Whether you use them in a BASIC program or machine language program, they can be great fun to create. They can also make your game display look really first class.

Putting the Man Together

At this stage you should still have the previous program in the computer with the top of RAM set at 12288 and the new character set in memory from that address upward. So type NEW and RETURN to clear out that program and type in Program Number 17.

Program # 17

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 169,33,141,244,5,169,34,141,245,
    5,32,105,3,32,129,3,32,148,3
42 DATA 32,129,3,169,14,141,244,217,141,
     245,217,141,28,218,141,29,218
43 DATA 141,68,218,141,69,218,76,60,3,
    169,35,141,28,6,169,36,141,29,6
44 DATA 169,37,141,68,6,169,38,141,69,
    6,32,162,3,96,169,27,141,29,6
```

```
46 DATA 169,28,141,68,6,169,29,141,69,6,
    32,162,3,96,169,30,141,68,6
48 DATA 169,31,141,69,6,32,162,3,96,162,
    255,160,255,136,208,253,202
49 DATA 208,248,96,-1
50 POKE53272,(PEEK(53272)AND240)OR12
60 PRINTCHR$(147):SYS828
```

When you RUN this program you should get a little man in centre screen, slowly running. This is set at the slowest speed to enable you to see clearly how the parts of the character change. The two 255's in line 48 are the delay loop counters, and set for maximum delay. Try decreasing the first of these (it is the outside loop counter) to speed up the action.

The machine language program is still being held in the cassette file buffer starting at address 828, but this program is approaching the limit of this area of memory.

GETTING THE PROGRAM ON TAPE

You can store the programs set out in previous chapters on tape just as you can with any BASIC program. In this program, however, you have a very different situation.

Firstly, the BASIC program used to POKE the new character into the protected part of top RAM from 12288 up has been overwritten by the last BASIC program you typed in. So if you SAVEd what is here now in basic you would have no program to POKE in the character set and it would be lost when the computer was switched off. Also, being stored *above* what the C-64 now believes is the top of RAM, the character set is outside the area of RAM which the BASIC system will handle with the regular LOAD statement. In any case we do not really want all that BASIC just to LOAD a machine language program. Neither do we want our special character set cluttering up the valuable RAM area that may be needed to carry the rest of the game program.

So here is a neat way of using a sort of modified file system to overcome the problem.

When the little man is correctly running on the spot, press STOP and RESTORE to halt the computer. Set up your tape recorder ready to SAVE the program and type in Program Number 18.

Program # 18

```
10 A$="RUNMAN":REM YOU CAN CHANGE NAME
20 L=LEN(A$):POKE183,L:FORI=1 TO L:POKE
   679+I,ASC(MID$(A$,I)):NEXT
30 POKE187,168:POKE188,2
40 POKE193,0:POKE194,48
50 POKE174,0:POKE175,60
60 POKE185,1:POKE186,1
70 SYS62957
```

When you RUN this program you will get the usual PRESS PLAY AND RECORD ON TAPE report on the screen. So do this and press RETURN. When READY appears, do not touch the tape recorder (it will be stopped). Type in SAVE, press RETURN and the BASIC program will be recorded in the normal manner.

Incidentally, you need to SAVE this last BASIC program to load the machine language program. This is because the cassette file buffer is cleared in this SAVE routine.

Now switch off the computer (to clear it out) switch it on again and rewind the tape. To LOAD the program back into the computer:

- POKE 56, 48 : NEW (and RETURN). This sets ramtop again at 12288.
- LOAD in the normal way and the screen will report LOADING RUN MAN
- When READY appears, leave the tape recorder alone (it will already be stopped).
- Type in LOAD (and RETURN) and the BASIC program will be LOADed in the usual way. When READY appears you are ready to RUN the program again.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 33	169,33	LDA 33
(b)	Store contents of accumulator in address 1524	141,244,5	STA 1524
(c)	Load accumulator with 34	169,34	LDA 34
(d)	Store contents of accumulator in address 1525	141,245,5	STA 1525
(e)	Jump to subroutine at address 873	32,105,3	JSR 873
(f)	Jump to subroutine at address 897	32,129,3	JSR 897
(g)	Jump to subroutine at address 916	32,148,3	JSR 916
(h)	Jump to subroutine at address 897	32,129,3	JSR 897
(i)	Load accumulator with 14	169,14	LDA 14
(j)	Store contents of accumulator in address 55796	141,244,217	STA 55796
(k)	Store contents of accumulator in address 55797	141,245,217	STA 55797
(l)	Store contents of accumulator in address 55836	141,28,218	STA 55836
(m)	Store contents of accumulator in address 55837	141,29,218	STA 55837
(n)	Store contents of accumulator in address 55876	141,68,218	STA 55876
(o)	Store contents of accumulator in address 55877	141,69,218	STA55877
(p)	Jump to address 828	76,60,3	JMP 828
	First subroutine		
(q)	Load accumulator with 35	169,35	LDA 35
(r)	Store contents of accumulator in address 1564	141,28,6	STA 1564
(s)	Load accumulator with 36	169,36	LDA 36
(t)	Store contents of accumulator in address 1565	141,29,6	STA 1565

(u)	Load accumulator with 37	169,37	LDA 37
(v)	Store contents of accumulator in address 1604	141,68,6	STA 1604
(w)	Load accumulator with 38	169,38	LDA 38
(x)	Store contents of accumulator in address 1605	141,69,6	STA 1605
(y)	Jump to subroutine at address 930 (delay loop)	32,162,3	JSR 930
(z)	Return	96	RTS
	Second subroutine		
(aa)	Load accumulator with 27	169,27	LDA 27
(bb)	Store contents of accumulator in address 1565	141,29,6	STA 1565
(cc)	Load accumulator with 28	169,28	LDA 28
(dd)	Store contents of accumulator in address 1604	141,68,6	STA 1604
(ee)	Load accumulator with 29	169,29	LDA 29
(ff)	Store contents of accumulator in address 1605	141,69,6	STA 1605
(gg)	Jump to subroutine at address 930	32,162,3	JSR 930
(hh)	Return	96	RTS
	Third subroutine		
(ii)	Load accumulator with 30	169,30	LDA 30
(jj)	Store contents of accumulator in address 1604	141,68,6	STA 1604
(kk)	Load accumulator with 31	169,31	LDA 31
(ll)	Store contents of accumulator in address 1605	141,69,6	STA 1605
(mm)	Jump to subroutine at address 930	32,162,3	JSR 930
(nn)	Return	96	RTS
	Fourth subroutine (delay loops)		
(oo)	Load X register with 255	162,255	LDX 255
(pp)	Load Y register with 255	160,255	LDY 255
(qq)	Decrement contents of Y register by 1	136	DEY

(rr)	Branch on condition that result is *not* 0	208,253	BNE
(ss)	Decrement contents of X register by 1	202	DEX
(tt)	Branch on condition that result is *not* 0	208,248	BNE
(uu)	Return	96	RTS

Once the new character set has been designed and put into RAM memory, it is very simple to bring it all together and animate the little man. Nearly all of the machine language program is concerned with loading the accumulator with the new special characters that are the sections of the little man's body and storing them in the six screen addresses.

The man's running involves continuous action so there will be certain parts of the program which will be repeated over and over again. Just as we do in BASIC, we put them into subroutines. Similarly, as a delay routine must be introduced after each change of leg position, there's a subroutine for it.

Explanation of the Machine Language Program

Lines (a) to (p) make up the main body of the program. In lines (a) to (d) those parts of the figure that do not change, the head and back, are stored in their respective positions on the screen.

In lines (e) to (g) the program makes four jumps to the subroutines in a regular sequence. These are the subroutines which put in the different leg and arm positions. Each of these (in lines (y), (gg) and (mm)) call the delay loop subroutine at address 930.

Lines (i) to (o) do the color work. Remember when you POKE a character code onto the screen you also have to POKE a color code into the corresponding address. These lines do that by storing the color code in the six screen color code addresses being used for the figure.

In line (p) a new *jump* instruction has been used. It takes the program back to address 828 so that the man will keep running forever. The program can,

however, be stopped with STOP and RESTORE. If you wanted to limit the number of times the object moved, you would use either the X or Y register as a loop counter together with a conditional branch instruction instead of the plain jump instruction used here. But remember that if you use either the X or Y register you will have to add to the delay loop subroutine the instructions to store them on the stack at the start of the subroutine and to return the original values from the stack at the end of the subroutine.

Jump to subroutine has only one addressing mode and that is *absolute/direct*. That is, the instruction code 32 is followed by the address (Least Significant Byte first then the Most Significant Byte) to which the program is to go. The subroutine must end with the Return instruction (code 96).

Jump is a plain jump instruction which means go to the specified address and start to operate from there. There is no Return instruction nor any other condition. The jump instruction is used in this program in the absolute/direct addressing mode. It has another addressing mode which we will meet in the next program.

Running Across the Screen

The same special character set will be used in Program Number 19 so if you still have Program Number 17 in the computer LIST it (or LOAD it if you've SAVEd it on tape) and then type in Program Number 19.

Program # 19

```
10 N=12600
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
39 DATA 162,0,232,169,33,157,234,5,169,
   34,157,235,5,169,35,157,18,6,169,14
```

```
40 DATA 157,234,217,157,235,217,157,18,
   218,157,19,218,157,58,218,157,59
41 DATA 218,108,97,49,99,49,169,36,157,
   19,6,169,37,157,58,6,169,38
42 DATA 157,59,6,32,215,49,169
43 DATA 128,141,97,49,32,227,49,76,58,
   49,169,27,157,19,6,169,28,157,58
44 DATA 6,169,29,157,59,6,32,215,49,169,
   157,141,97,49,32,227,49,76,58,49
45 DATA 169,27,157,19,6,169,30,157,58,
   6,169,31,157,59,6,32,215,49
46 DATA 169,186,141,97,49,32,227,49,76,
   58,49,169,27,157,19,6,169,28
47 DATA 157,58,6,169,29,157,59,6,32,215,
   49,169,99,141,97,49,32,227,49,76
48 DATA 58,49,169,32,157,233,5,157,57,6,
   157,17,6,96,138,72,152,72,162,255
49 DATA 160,255,136,208,253,202,208,248,
   104,168,104,170,96,-1
50 POKE53272,(PEEK(53272)AND240)OR12
60 PRINTCHR$(147):SYS12600
```

When you RUN the program the little man will run in from the left-hand side about half way down the screen. After crossing the screen he will come in from the left again one line down. he will continue to do this until the bottom line when he runs out of X's. He is moved across the screen by the *absolute indexed by X register* addressing mode. X starts at 0 and the run is finished when X = 255. However, there is a wrap-around effect here, and X returns to 0. Consequently he starts all over again.

NB. This machine language program is held above the 'artificial' top of RAM, following the new characters. It is a little too long to store in the cassette file buffer. However I wanted to show how all of the material could be held in this position and how, with the LOADing method given earlier in the chapter, you can SAVE this on tape. There is no need to tape the BASIC loading program. In fact, the rest of RAM is now completely free for use with another BASIC program.

The Machine Language Program

	Instructions	Decimal Code	Mnemonic
(a)	Load X register with 0	162,0	LDX 0
(b)	Increment X register by 1	232	INX
(c)	Load accumulator with 33	169,33	LDA 33
(d)	Store contents of accumulator in address 1514	157,234,5	STA 1514
(e)	Load accumulator with 34	169,34	LDA 34
(f)	Store contents of accumulator in address 1515	157,235,5	STA 1515
(g)	Load accumulator with 35	169,35	LDA 35
(h)	Store contents of accumulator in address 1554	157,18,6	STA 1554
(i)	Load accumulator with 14	169,14	LDA 14
(j)	Store contents of accumulator in address 55786	157,234,217	STA 55786
(k)	Store contents of accumulator in address 55787	157,235,217	STA 55787
(l)	Store contents of accumulator in address 55826	157,18,218	STA 55826
(m)	Store contents of accumulator in address 55827	157,19,218	STA 55827
(n)	Store contents of accumulator in address 55866	157,58,218	STA 55866
(o)	Store contents of accumulator in address 55867	157,59,218	STA 55867
(p)	Jump to address 12641 Indirect address 12641 12642	108,97,49 99 49	JMP 12641
	First leg routine		
(q)	Load accumulator with 36	169,36	LDA 36
(r)	Store contents of accumulator in address 1555 + X	157,19,6	STA 1555,X
(s)	Load accumulator with 37	169,37	LDA 37
(t)	Store contents of accumulator in address 1594 + X	157,58,6	STA 1594,X
(u)	Load accumulator with 38	169,38	LDA 38
(v)	Store contents of accumulator in address 1595 + X	157,59,6	STA 1595,X

(w)	Jump to subroutine address 12759	32,215,49	JSR
(x)	Load accumulator with 128	169,128	LDA 128
(y)	Store contents of accumulator in address 12641	141,97,49	STA 12641
(z)	Jump to subroutine address 12771	32,227,49	JSR
(aa)	Jump to address 12602	76,58,49	JMP
	Second leg routine		
(ab)	Load accumulator with 27	169,27	LDA 27
(ac)	Store contents of accumulator in address 1555 + X	157,19,6	STA 1555,X
(ad)	Load accumulator with 28	169,28	LDA 28
(ae)	Store contents of accumulator in address 1594 + X	157,58,6	STA 1594,X
(af)	Load accumulator with 29	169,29	LDA 29
(ag)	Store contents of accumulator in address 1595 + X	157,59,6	STA 1595,X
(ah)	Jump to subroutine address 12759	32,215,49	JSR
(ai)	Load accumulator with 157	169,157	LDA 157
(aj)	Store contents of accumulator in address 12641	141,97,49	STA 12641
(ak)	Jump to subroutine address 12771	32,227,49	JSR
(al)	Jump to address 12602	76,58,49	JMP
	Third leg routine		
(am)	Load accumulator with 27	169,27	LDA 27
(an)	Store contents of accumulator in address 1555 + X	157,19,6	STA 1555,X
(ao)	Load accumulator with 30	169,30	LDA 30
(ap)	Store contents of accumulator in address 1594 + X	157,58,6	STA 1594,X
(aq)	Load accumulator with 31	169,31	LDA 31
(ar)	Store contents of accumulator in address 1595 + X	157,59,6	STA 1595,X
(as)	Jump to subroutine address 12759	32,215,49	JSR
(at)	Load accumulator with 186	169,186	LDA 186

(au)	Store contents of accumulator in address 12641	141,97,49	STA 12641
(av)	Jump to subroutine address 12602	32,227,49	JSR
(aw)	Jump to address 12602	76,58,49	JMP
	Fourth leg routine		
(ax)	Load accumulator with 27	169,27	LDA 27
(ay)	Store contents of accumulator in address 1555 + X	157,19,6	STA 1555,X
(az)	Load accumulator with 28	169,28	LDA 28
(ba)	Store contents of accumulator in address 1594 + X	157,58,6	STA 1594,X
(bb)	Load accumulator with 29	169,29	LDA 29
(bc)	Store contents of accumulator in address 1595 + X	157,59,6	STA 1595,X
(bd)	Jump to subroutine address 12759	32,215,49	JSR
(be)	Load accumulator with 99	169,99	LDA 99
(bf)	Store contents of accumulator in address 12641	141,97,49	STA 12641
(bg)	Jump to subroutine address 12771	32,227,49	JSR
(bh)	Jump to address 12602	76,58,49	JMP
	Routine to Clear image		
(bi)	Load accumulator with 32	169,32	LDA 32
(bj)	Store contents of accumulator in address 1513 + X	157,233,5	STA 1513,X
(bk)	Store contents of accumulator in address 1593 + X	157,57,6	STA 1593,X
(bl)	Store contents of accumulator in address 1553 + X	157,17,6	STA 1553,X
(bm)	Return	96	RTS
	Delay Routine		
(bn)	Transfer contents of X register to accumulator	138	TXA
(bo)	Push contents of accumulator onto stack	72	PHA
(bp)	Transfer contents of Y register to accumulator	152	TYA

(bq)	Push contents of accumulator onto stack	72	PHA
(br)	Load X register with 255	162,255	LDX 255
(bs)	Load Y register with 255	160,255	LDY 255
(bt)	Decrement Y register by 1	136	DEY
(bu)	Branch on condition that result is *not* 0	208,253	BNE
(bv)	Decrement contents of X register by 1	202	DEX
(bw)	Branch on condition that result is *not* 0	208,248	BNE
(bx)	Pull last value on the stack and put in accumulator	104	PLA
(by)	Transfer contents of accumulator to Y register	168	TAY
(bz)	Pull last value now on stack and put in accumulator	104	PLA
(ca)	Transfer contents of accumulator to X register	170	TAX
(cb)	Return from subroutine	96	RTS

Explanation of the Machine Language Program

In the previous program where the man was running on the spot, the head and back needed to be drawn only once. The legs were then drawn by running through a series of subroutines.

In this program, however, the head and back have to be drawn in each column across the screen. Furthermore, each movement is associated with a different set of leg movements. Using ordinary subroutines seemed difficult, if not impossible, so a different method has been used. This method gives you an opportunity to have a look at a new addressing mode.

Line (a) sets up the X register for address indexing.

Line (b) increments the contents of the X register by one.

Lines (c) to (h) put the two halves of the head and the back on the screen.

Lines (i) to (p) handle the color requirements. It doesn't matter which pair of legs is being used. The address in use is having the color code put into it.

Line (p) is our new instruction, jump. It uses the *indirect* addressing mode. It selects the leg routine it will jump to each time the head and back are drawn.

The Leg Routines. Notice that the table defines four separate leg routines. In the first few lines of each of the routines the various parts of the legs and front are loaded into the accumulator and then stored in the appropriate screen addresses. Note that all these store instructions use the *absolute indexed by X register* addressing mode. As each of these routines is called by the jump instruction in line (p) it will have the same X register value as the first or head and back, routine, so the little character segments will be stored in the right addresses to line up with the head and back.

In lines (w) and (z) in the first leg routine, there are simple jumps to two subroutines which, respectively, clear the unwanted back bytes as the little man moves across the screen and provide the time delay.
Lines (x) and (y) put the address of the next leg routine to be used into addresses 12641 and 12642 so that the jump instruction knows where to take the program each time.

The four leg routines are substantially the same. The only things to change are the characters used and the addresses in which these characters are to be stored. But check each one out and make sure you understand precisely what each routine actually does.

The Clear Routine and the Delay Routine

These are true subroutines. They are used by each of the four leg routines. The first is the clear routine. The little man uses two columns. Consequently, as he moves forward, the characters in the first column are constantly overwritten by the new characters moving forward. However, it is necessary to clear out the column just vacated by storing in each address a space (code 32). This is done in line (bi) by loading the accumulator with 32 and then storing this in the following three addresses.

The delay subroutine is now familiar: two nested loops which allow you to delay the processing of the operation for as long as you like depending on how fast you want the character's movement to be. Note that the X register is doing some pretty important work in the overall program. Therefore it is necessary to hold the current value in X register on the stack while the delay loops do their thing and to pull this value off the stack when you want to get on with the main program. The Y register need not have been used here. It was left in to remind you to store all registers being used if you want to use them temporarily for another purpose.

Jump Instruction with Indirect Addressing Mode

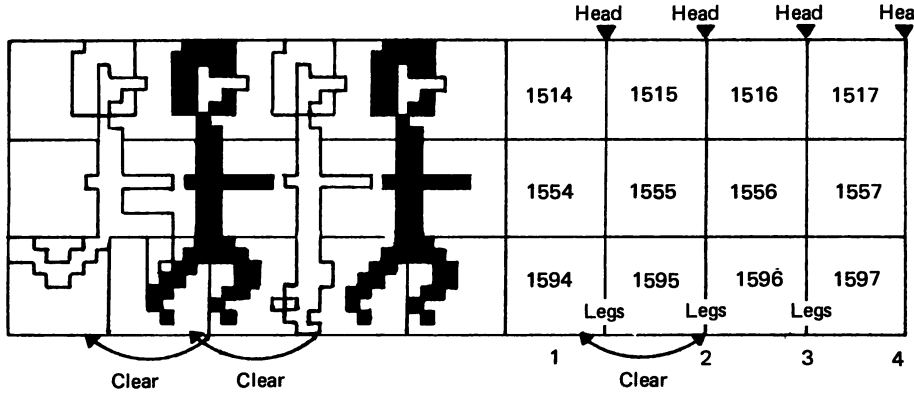
In previous instructions involving actual addresses (ie, not the implied and relative modes) there has been an actual address (or almost an actual address) in the instruction code varied only by the indexing of the X or Y register.

With *indirect addressing* the instruction code includes an address which itself holds the actual address where the 6510 will carry out the instruction. We have used zero page addressing as an indirect mode when y = 0 but this jump instruction is the only instruction in the 6510 that has a true indirect addressing mode.

In this program I have chosen two addresses in RAM (12641 and 12642) to hold the regularly changing addresses of the leg routines. Notice that, as usual, the leg routine address is held in two bytes. The first (12641) holds the Least Significant Byte, while the second (12642), holds the Most Significant Byte. The jump instruction only tells the 6510 to look at the first address but it automatically takes the address from both addresses. Fortunately all these addresses have the Most Significant Byte of 49 so that it is only necessary to change the first part of the leg routine address in

You might well ask why I did not use the ordinary jump instruction (code 76) in line (p) and then simply change the second byte in that line. There was no special reason! It was simply to give you the opportunity to see the *indirect addressing* mode in operation.

The following diagram may help you see how the various movements were worked. You should bear in mind that you have to work out every last movement in precise detail so that you can tell the 6510 exactly, step by step, what it has to do.



8
Working With the Kernal

So far we have been using the actual machine language instructions of the 6510. We have been telling it directly what to do, step by painful step.

Had we been working in BASIC a lot of the work would have been done for us by the BASIC interpreter and lots of other systems routines. These programs in the ROM instruct the 6510 in the same machine language that we have been using. If we could gain access to them we could include them as subroutines in our own machine language programs.

There are some very real advantages in using this tactic. Firstly the programs are integral parts of the C-64 system and are therefore foolproof. Secondly, since the programs are stored in ROM we can save valuable RAM space by using them.

As it happens, there is a section of the C-64 ROM called the *kernal.* The kernal contains a list of addresses which can be used to access many of these systems routines. Details of many of the routines, together with their addresses, are set out in the *C-64 Programmer's Reference Guide.*

While all these routines are useful, most of them are beyond the scope of this book. There are, however, three particularly handy routines that we want to make use of: CHROUT, PLOT and GETIN.

CHROUT (Output a character)

Call address: 65490

This routine puts any character in the accumulator onto the screen.

PLOT (Set cursor position)

Call address: 65520

When the carry flag is cleared this routine sets the cursor location to the coordinates specified by the values in the X and Y registers.

GETIN (Get a character from the keyboard buffer)

Call address: 65508
This routine removes a character from the keyboard buffer and puts its Chr$ code into the accumulator. If there is no character it will put 0 in the accumulator.

WORKING WITH KERNAL ROUTINES

Using routines from the kernal is quite easy. Doing so can save you a lot of work.

Putting Words on the Screen: CHROUT

In many of your programs you will want to put on the screen some text (perhaps like instructions or headings). The CHROUT routine is really just like the BASIC PRINT and does the same things. Try this. Type in Program Number 20.

Program # 20

```
10 N=828
20 READD:IFD=-1THEN35
30 POKEN,D:N=N+1:GOTO20
35 FOR N=951 TO 900 STEP-1:READ D:POKE
    N,D:NEXT
40 DATA 162,52,189,131,3,32,210,255,202,
    208,247,96,-1
41 DATA 147,17,17,159
42 DATA 75,69,82,78,65,76,32,82,79,85,
    84,73,78,69,83,32,65,82,69,32
```

```
44 DATA 86,69,82,89,32,85,83,69,70,85,76
,       32,70,79,82,32,77,47,76,32,80,82
46 DATA 79,71,82,65,77,83
50 SYS828
```

When you RUN this program a short message should appear on your screen.

The actual machine language program which puts this text on the screen is the 12 bytes contained in line 40. The rest of the DATA lines hold text characters and various cursor controls from the keyboard.

Note that when you want to put something on the screen in machine language you have to store it in the computer's memory first. It's the same in BASIC. When you type a line like PRINT "something or other", what you put in to be PRINTed is actually stored in memory.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load X register with 52	162,52	LDX 52
(b)	Load accumulator with contents of address 899 + X	189,131,3	LDA 899,X
(c)	Call kernal routine at address 65490	32,210,255	JSR 65490
(d)	Decrement X register by 1	202	DEX
(e)	Branch on condition that result is *not* 0	208,247	BNE
(f)	Return	96	RTS

This kernal routine is like the BASIC PRINT operation. However it uses a different set of character codes from the one we have been using so far. In our previous programs we have stored character codes directly into screen memory addresses. This method is the same as using the BASIC POKE. It uses the screen codes listed in Appendix F.

The kernal routine, however, uses the Chr$ codes shown in Appendix H. The great advantage of this is that the kernal routine responds to all these codes, including RVS OFF, CURSOR movements, COLOR, and so on.

If your text is built into your program as Chr$ codes, calling CHROUT will put it on the screen with the same effect as a BASIC PRINT statement. Lines 42 and 44 contain Chr$ codes which you can decipher by looking at Appendix H.

Explanation of the Machine Language Program

To use this kernal routine you load a character code into the accumulator and then call CHROUT with the jump instruction. The routine will execute the character code instruction, which may be to print a character, move the cursor, or clear the screen or something else.

So first you have to store the appropriate characters in memory. Then you put the kernal routine in a loop and let it look at each character in turn by loading the accumulator.

In line (a) the X register is loaded with 52 because there are 52 codes to be read. X will be used to index the addresses and as the loop counter. It is very easy and economical to use X for both these jobs and to use the decrement X instruction in line (d). However, it also means that the program reads the characters in memory from the highest memory address down. This explains line 35 in the BASIC program, where N = 951 TO 900 STEP – 1. There is no special significance in all this except to emphasise the point that you must make sure you get all these things in the correct order.

In line (b) the program loads these various characters from memory into the accumulator so that in line (c) the kernal routine can do whatever is needed, like print the character, move cursor down, change color, put reverse on and so on.

Using the Plot Routine from the Kernal

In many sorts of programs you need to draw lines on the screen, perhaps to construct mazes or other sorts of playing areas.

The PLOT routine used together with CHROUT can be extremely useful and quick for this sort of thing.

For simple jobs, of course, you will probably find it easier to use the normal keyboard controls. But for complex jobs, PLOT really comes into its own. Try this. Type in Program Number 21.

Program # 21

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 162,20,160,0,24,32,240,255,169,
    42,32,210,255,200,200
41 DATA 202,208,242
43 DATA 162,0,160,0,24,32,240,255,169,
    42,32,210,255,200,200
44 DATA 232,224,20,208,240
46 DATA 162,0,160,20,24,32,240,255,169,
    42,32,210,255,232,224,22
47 DATA 208,242
48 DATA 162,10,160,1,24,32,240,255,169,
    28,32,210,255,169,42,32,210,255,200
49 DATA 192,39,208,237,169,5,32,210,255,
    96,-1
50 PRINTCHR$(147):SYS828
```

When you RUN Program Number 21 you should get a sort of flag display, with two blue diagonals, a blue vertical and a red horizontal. The READY and flashing cursor return will be in white.

My flag design may not be a work of art but it is a good example of what you can achieve in only 85 bytes. The variety of patterns — lines, diagrams, mazes, etc — that you can produce using PLOT is limited only by your imagination.

The Machine Language Program

	Instruction	Decimal code	Mnemonic
(a)	Load X register with 20	162,20	LDX 20
(b)	Load Y register with 0	160,0	LDY 0
(c)	Clear carry flag	24	CLC
(d)	Call PLOT routine at 65520	32,240,255	JSR 65520
(e)	Load accumulator with 42	169,42	LDA 42
(f)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(g)	Increment Y register by 1	200	INY
(h)	Increment Y register by 1	200	INY
(i)	Decrement X register by 1	202	DEX
(j)	Branch on condition that result is *not* 0	208,242	BNE
(k)	Load X register with 0	162,0	LDX 0
(l)	Load Y register with 0	160,0	LDY 0
(m)	Clear carry flag	24	CLC
(n)	Call PLOT routine at 65520	32,240,255	JSR 65520
(o)	Load accumulator with 42	169,42	LDA 42
(p)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(q)	Increment Y register by 1	200	INY
(r)	Increment Y register by 1	200	INY
(s)	Increment X register by 1	232	INX
(t)	Compare X register with 20	224,20	CPX 20
(u)	Branch on condition that result is *not* 0	208,240	BNE
(v)	Load X register with 0	162,0	LDX 0
(w)	Load Y register with 20	160,20	LDY 20
(x)	Clear carry flag	24	CLC
(y)	Call PLOT routine at 65520	32,240,255	JSR 65520
(z)	Load accumulator with 42	169,42	LDA 42
(aa)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(bb)	Increment X register by 1	232	INX
(cc)	Compare X register with 22	224,22	CPX 22
(dd)	Branch on condition that result is *not* 0	208,242	BNE
(ee)	Load X register with 10	162,10	LDX 10
(ff)	Load Y register with 1	160,1	LDY 1

(gg)	Clear carry flag	24	CLC
(hh)	Call PLOT routine at 65520	32,240,255	JSR 65520
(ii)	Load accumulator with 28	169,28	LDA 28
(jj)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(kk)	Load accumulator with 42	169,42	LDA 42
(ll)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(mm)	Increment Y register by 1	200	INY
(nn)	Compare Y register with 39	192,39	CPY 39
(oo)	Branch on condition that result is *not* 0	208,237	BNE
(pp)	Load accumulator with 5	169,5	LDA 5
(qq)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(rr)	Return	96	RTS

Explanation of the Machine Language Program

This program contains four routines, each of which draws one of the lines. To make it easier to read, we've left a space after lines (j), (u), (dd) and (oo) — the last line of each routine.

In each of these routines the PLOT routine sets the position of the cursor and CHROUT then does the job of putting an asterisk (code 42) at that point.

First Routine

In lines (a) and (b) the X and Y registers are loaded with the starting coordinates for the first line to be put on the screen. In each of these line-drawing routines either or both of the X and Y registers will be incremented or decremented to change the cursor position for the next point to be printed. Consequently one of these registers can be used as the loop counter which enables the points in the particular line to be drawn. This has to be arranged for whatever line you want to draw.

In line (c) the clear carry flag instruction is used. As mentioned at the start of this chapter, this is a condition necessary for the PLOT routine to work correctly.

In line (d) the jump instruction is used to send the program to the PLOT routine which now sets the print position according to the current values of the X and Y registers.

In line (e) the accumulator is loaded with code 42, which is the code for an asterisk.

In line (f), the CHROUT routine is called to put the asterisk on the screen in the position determined by the PLOT routine.

Lines (g) and (h) increment the Y register by 1 *twice,* while (i) decrements X by 1. As the screen is about twice as wide as it is high, one coordinate has to increase twice as fast as the other coordinate. Notice that the X register is also being used as the loop counter. When X becomes 0 the conditional branch instruction in line (j) allows the program to pass onto the next instruction.

Second Routine

The second routine — lines (k) to (u) — is much like the first. The main difference is that *both* the coordinates, X and Y, are being incremented and we must therefore use the compare instruction for the conditional branch instruction to work with.

Third Routine

The third routine — lines (v) to (dd) — is very like the second. Once again the compare instruction is used for the conditional branch instruction.

Fourth Routine

The special feature of the last routine — lines (ee) to (oo) — is that we are going to use another color.

In line (ii) the accumulator is loaded with 28. This is the Chr$ code for red.

In line (jj) the CHROUT routine is called to act upon the contents of the accumulator. This means that everything will now be printed in red.

So when line (kk) loads the accumulator with 42 (the asterisk), and line (ll) calls CHROUT, the asterisk and consequently the line, will be in red.

Lines (pp) to (rr) finish off the program. In line (pp) the accumulator is loaded with 5 (the Chr$ code for the white), and the CHROUT routine is called again

in line (qq). When the program returns to BASIC in line (rr), READY and the flashing cursor will reappear in white.

If you had used BASIC to draw these lines on the screen they would look exactly the same. But a BASIC program would use much more memory, and the screen pattern would not appear instantly as this one does.

It isn't smart to write everything in machine language. Doing so could cost you months of heavy calculation and debugging. Sometimes the job can be done much more easily in BASIC. But a program like this can make a BASIC game a classic.

Keyboard Control with the Kernal: GETIN

GETIN is the last of the three kernal routines used in this book. It allows you to access the keyboard from within a machine language program. Try this. Type in Program Number 22.

Program # 22

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 32,228,255,201,0,240,249
42 DATA 32,210,255,76,60,3,-1
50 PRINT CHR$(147):SYS828
60 REM**TYPE BOTH DATA LINES LIKE THIS-
   DATA LINE 42 WILL BE REPLACED LATER
```

When you RUN Program Number 22 the screen will appear to stop. This is because the flashing cursor has disappeared. Actually the computer is now under the control of the machine language program contained in the two DATA lines. Everything works except the cursor.

Try CLR/HOME and SHIFT. The screen will clear. Then you can use any of the keys to type characters on the screen in the usual way. Even the cursor control keys work, although there is no cursor to show where the print position is on the screen. The program is in a continuous loop which allows it to take in information from the keyboard. RESTORE and RUN/STOP will return the machine to normal.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Call GETIN routine at 65508	32,228,255	JSR 65508
(b)	Compare accumulator with 0	201,0	CMP 0
(c)	Branch on condition that result is 0	240,249	BEQ
(d)	Call CHROUT routine at 65490	32,210,255	JSR 65490
(e)	Jump to address 828	76,60,3	JMP 828

Explanation of the Machine Language Program

In line (a) the kernal routine is called. If there is a key code in the keyboard buffer — ie, if a key has been pressed — the GETIN routine will place the Chr$ code of this key in the accumulator. If no key has been pressed it will place 0 in the accumulator.

In line (b) this is tested by comparing the accumulator with 0. If no key has been pressed then the conditional branch instruction in line (c) will branch the program back to the start of line (a) to have another look for a key being pressed.

If a key has been pressed then the condition for the branch in line (c) will not be met and the program will pass on the key's Chr$ code loaded in the accumulator.

Line (d) calls the CHROUT routine. If you hit a character key, CHROUT will put the character on the screen. If you hit a non-character key (eg, a cursor control key), CHROUT will take the appropriate action.

Now edit Program Number 22 so it reads as for Program Number 23.

Program # 23

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 32,228,255,201,0,240,249
42 DATA 201,133,240,7,201,134,240,7,76,
    64,3,141,132,3,96,141,133,3,96,-1
50 POKE900,0:POKE901,0
60 PRINTCHR$(147):SYS828
70 PRINT PEEK(900),"900 F1"
80 PRINT PEEK(901), "901 F3"
```

When you RUN this program the screen will go blank again. Now press any key other than f1 or f3 (the special function keys) and nothing will happen.

Press the f1 key and the screen will display:

```
133     900     F1
  0     901     F3
```

RUN again and press f3. The screen will display:

```
  0     900     F1
134     901     F3
```

The program returns to BASIC each time. 133 is the Chr$ code for f1 and 134 is the Chr$ code for f3. In this program these are the only *active* keys on the keyboard.

The Machine Language Program

The program is identical with the previous one down to and including line (d).

	Instruction	Decimal Code	Mnemonic
(e)	Compare accumulator with 133	201,133	CMP 133
(f)	Branch on condition that result *is* 0	240,7	BEQ
(g)	Compare accumulator with 134	201,134	CMP 134

(h)	Branch on condition that result *is* 0	240,7	BEQ
(i)	Jump to address 828	76,64,3	JMP 828
(j)	Store contents of accumulator in address 900	141,132,3	STA 900
(k)	Return to BASIC	96	RTS
(l)	Store contents of accumulator in address 901	141,133,3	STA 901
(m)	Return to BASIC	96	RTS

Explanation of the Machine Language Program:

From lines (a) to (d) the Chr$ code of the key pressed is loaded in the accumulator. Of course, if no key has been pressed, line (d) branches the program back to have another look for a key press.

If a key is pressed, line (e) compares its Chr$ code with 133. This is the Chr$ code of f1 function key. If f1 had been pressed then the comparison in line (e) would produce a 0 result.

The conditional branch instruction in line (f) branches if the result is 0, and will now jump the program forward 7 bytes. This takes it to the start of line (j) where the contents of the accumulator are stored in address 900.

The program returns to BASIC in line (k). Now the BASIC program simply prints the result on the screen.

If the key pressed had been f3 (with the Chr$ code 134) then lines (e) and (f) would pass the program through to line (g), where the comparison would give a 0 result. The branch instruction would then jump the program forward to the start of line (l). There it would put the accumulator in address 901, return to BASIC and print the result on the screen. If neither of the keys had been pressed then the conditional branch instructions in lines (f) and (h) would simply pass the program on to the next instruction in sequence and finally to line (i).

In line (i) a plain jump instruction takes the program back to the start of line (a) to await the next key press.

The program from line (e) onwards illustrates how you can get control by pressing certain special keys. In this case the result has been the printing out of results on the screen with the BASIC program.

This means that in a standard sort of game program, for example, you could jump to a routine to forward a torpedo or move a character to the left or right, just by having the user press a special key.

Usually you would set this up as a subroutine. In the main program (which could be a loop moving something around the screen) you would insert a jump to subroutine instruction. When the program read this instruction it would jump to the subroutine to check if a key had been pressed. If it had it would then do whatever else you had told it to do, contingent on that key having been pressed. If the key had not been pressed, however, it would immediately return to the main program and continue on its merry way. But make sure you end the subroutine with a Return instruction!

This is one of the features of machine language programming. In the normal course of screen display work, the machine language program works so quickly that it can go off and check these things and even pop other material on the screen. The result can look like continuous and/or simultaneous action.

9

Sprites or Movable Object Blocks

Through the Video Interface Chip (commonly called the VIC II), your C-64 provides you with an excellent system of special graphic symbols called sprites.

Sometimes you will see sprites referred to as movable object blocks — a term which is probably more descriptive of their actual function. They are a special type of high-resolution programmable object block, which you can make into almost any shape you want it to be. By relatively simple programming, you can change their size, shape and color and move them anywhere you want to on the screen. For gaming (and indeed all sorts of other graphic displays) sprites provide you with some of the most useful graphic facilities available.

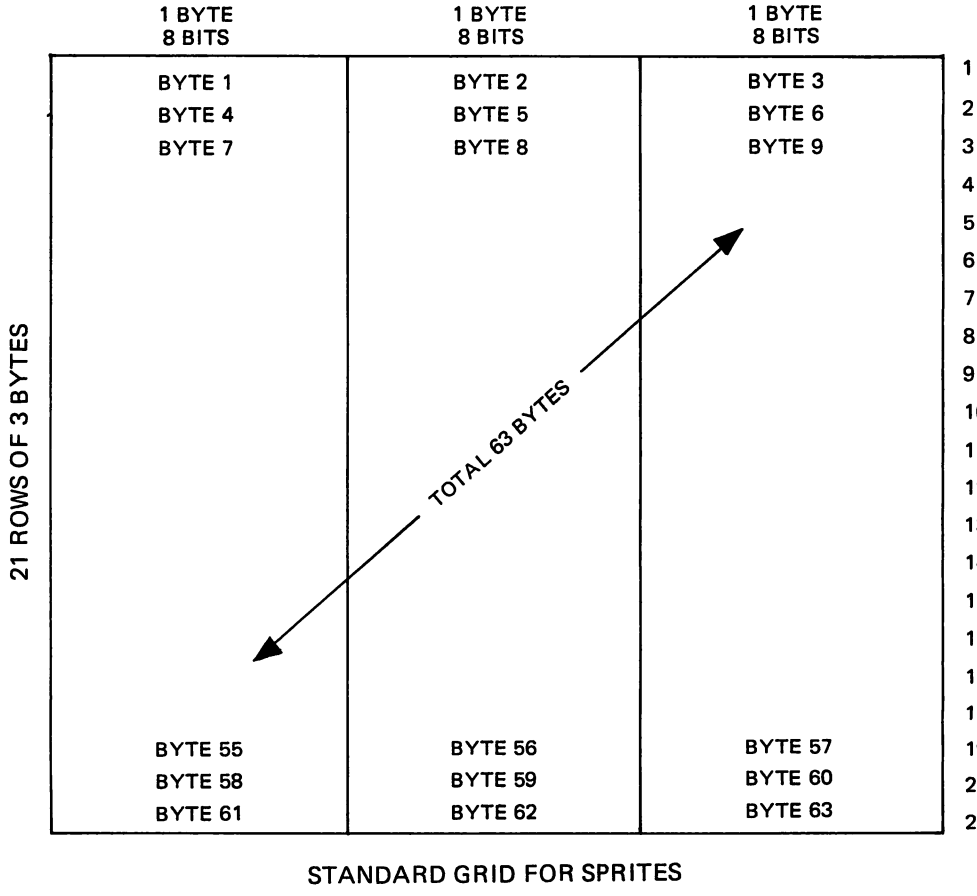
All the sprite operations can be programmed in BASIC. Subject to available RAM, which is not often a problem with your C-64, it is often easier and more economical of programming time to do some part of sprite work in BASIC. But for some things, particularly movement and the use of a number of sprites at the same time, BASIC is too slow. The results of sprites programmed fully in BASIC can be disappointing.

To produce the best results with sprites, it is usually essential for some part of your program to be written in machine language. There is nothing very special, however, about machine language as applied to sprites. Most of sprite operation is controlled by BASIC POKE and PEEK statements so it is really quite easy to convert this into machine language. But before we go into the details of machine language programming, we should have a look at sprite graphics in BASIC. We'll then have a better idea of what has to be done to program sprites in machine language.

DEFINING YOUR SPRITE

A sprite is a programmable object so your first job will be to design the shape you want it to be. The method for doing this is exactly the same as the method we employed in Chapter 7 to define the shape of the little man.

Unlike the programmable objects in Chapter 7, however, all sprites are designed on a standard size grid of 21 rows of 3 bytes per row, as shown in figure 9.1.



STANDARD GRID FOR SPRITES

As you can see, the grid is 24 bits wide — in other words, three bytes of 8 bits each. These are all binary bits which can hold either 1 (turned on) or 0 (turned off).

Using a sheet of graph paper you can create any shape you want to. Where you want a bit to show on the screen as part of your shape, you put a 1 into the box. If a bit is not to show, you use a 0. If it is to be a solid shape, you fill in all the boxes inside the outline with 1's.

When you have done this, you will have 63 bytes represented in binary form. That is to say, 21 rows of three bytes each where each byte consists of 8 bits which may be either 0 or 1. The order of the bytes is B1, B2 and B3 across the top row, then B4, B5 and B6 across the second row — and so on through the full 21 rows. That is the order you enter them in the DATA statement you use to put them into RAM storage.

Now, just as we did in Chapter 7, we convert each of these binary bytes to a decimal number equivalent. The result will be 63 decimal numbers which will provide all the data needed to define your own sprite design.

Actually, there are *64* bytes in each sprite. The last one is called the *place holder.* The machine takes care of this last one and all you need to do is remember that there are 64 bytes in each sprite. Anyway, 64 is a much more convenient number to work with on computers — it fits neatly into all our common numbers like 256, 1024 and so on.

In its normal operation, your C-64 will provide you with 8 sprites numbered from sprite 0 to sprite 7.

Note carefully the numbering of these sprites. A lot of sprite contol is done with single bits in a one byte register or address and the 8 sprite numbers correspond with the 8 bit numbers in a byte. If you keep in mind the numbering of the sprites you'll find it easier to see how a particular sprite relates to a particular bit and the decimal number equivalent of that bit position. If you have any doubts about this, have another look at Appendix A.

NB: All sprites are built up on the standard size grid and you can make sprites from a full grid down to 1 dot if you like. All the bits on the grid which contain 0 will be transparent (ie, act like a window) to any other display on the screen behind the sprite.

SPRITE POINTERS

You have now designed your sprite and have 63 decimal numbers ready to be stored somewhere in RAM. The question, of course, is where in RAM will you store them?

Before we can actually address that problem, however, we've got to make a slight detour. It concerns how the VIC II will find that RAM address once

you've set it up. It's a roundabout way of doing it but the reason for it will become clear as we progress.

Within RAM there is a block of memory addresses from 1024 to 2047 called the *screen memory area.* 1024 to 2023 hold the 1000 bytes representing the 1000 locations displayed on your TV monitor. 2040 to 2047 hold 8 bytes used for the *sprite pointers.* The pointer for sprite 0 is 2040 and the pointer for sprite 7 is 2047. So to get the sprite pointer for any sprite you just add the number of the sprite to 2040.

Each of sprite pointers is a byte of 8 bits so can hold any number up to 255. So the number you put into the sprite pointer can (theoretically) be any number from 0 to 255. Remember now that each of the sprites consists of 64 bytes, so the number that goes into the sprite pointer will have to be multiplied by 64 to give you the actual RAM address of the sprite. Let's look at a couple of examples. We'll take extreme cases to illustrate the effect of a particular number in the sprite pointer on the position of the sprite's 64 bytes of data held in RAM.

First, let's say that we place a zero in the sprite pointer. Zero times 64 will yield zero so the 64 bytes of our sprite data will commence at address zero. Now let's say that we place 255 (the maximum) in our sprite pointer. 255 times 64 yields 16320 so the 64 bytes will start at address 16320.

NB. Throughout this book we assume that your C-64 is operating with BANK 0. This is the memory arrangement when first you switch on your computer. For more about BANKs, see Appendix C.

That's all pretty straightforward — but there are limitations on the starting addresses you can actually use for this sprite data in RAM. In other words, limitations on the numbers between 0 and 255 that you put into the sprite pointer. To start with, under the BANK 0 arrangement, the VIC II can only work with RAM from address 0 to address 16383 — the first 16K of memory. So, for example, the address arrived at by multiplying 255 by 64 — 16320 — only provides space enough for one set of sprite data. And the address arrived at by multiplying zero by 64 — 0 — is on zero page where there is really no room to hold sprite data!

A further limitation is apparent for addresses 1024 to 2048. They are the addresses for the screen display area. So the numbers 16 to 32 should also be avoided. And from 4096, of course, you run into the area of RAM where

(under BANK 0) the VIC II gets its normal character set data from the ROM image.

Where to Store Sprite Data

We've said where we can't store sprite data, so now we should look at the problem positively. Let's try the number 13. 64 times 13 yields 832. Do you recognise it? It's almost at the start of that very useful area, the cassette file buffer. It's also a useful area for storing sprite data. There are, however, the usual things to remember about using it. You'll have to keep the sprite data and its loading program in your BASIC program so that it will be loaded into the buffer each time you LOAD and RUN the program.

Let's also try the number 192. 64 times 192 yields 12288. Recognise that one? It's the area we used to store the character data for RUNMAN in Chapter 7. It's also an area where we can very satisfactorily store a lot of sprite data. Of course, just as we did with RUNMAN , we'll have to lower ramtop to 12288 to protect the sprite data from our BASIC program. There's also another point worth noting carefully. If you're running special programmable characters (in addition to the sprites) you'll have to include these first (ie, from 12288 on) and follow them with the sprite data. The sprite data can then run up to 16383 before it gets out of sight of the VIC II. Running out of memory shouldn't worry you, though, because that's quite a lot of special graphics space and you still have 12K of BASIC RAM available and a further 24K from 16K to 40K for other data and machine language programs.

TURN ON THE SPRITE

Before a sprite will appear on the screen it must be turned on. This is done with the general BASIC statement:

```
POKE 53269,PEEK(53269)OR(2↑SN)
```

Address 53269 is a VIC II control register in which each of the 8 bits represents one sprite. Normally the bits hold a 0 and putting a 1 in a particular sprite's bit will turn that sprite on.

Recall the way sprites are numbered, from 0 to 7. Bit 0 in this register byte represents sprite 0 and the others are similarly numbered. The decimal value of each bit position represents 2 to the power of the bit number. This is what (2↑SN) means in the general statement. For example, sprite 2 corresponds with bit 2 so this value is (2↑2) or 4. Again, sprite 7 corresponds with bit 7 so

this value is (2 ↑ 7) or 128. Each of these numbers in binary means a 1 in one bit position and 0's in all the other positions. Therefore, logical OR by that binary number will ensure a 1 in that bit position — in other words, the sprite is turned on.

COLORING YOUR SPRITE

The VIC II has 8 individual color registers, one for each sprite. All you have to do is POKE one of the 16 standard color codes into the register corresponding with the sprite you're working with.

SPRITE COLOR REGISTERS

REGISTERS	SPRITE
53287	0
53288	1
53289	2
53290	3
53291	4
53292	5
53293	6
53294	7

POSITIONING YOUR SPRITE

Sprites are high-resolution graphics and their position on the screen is determined by individual dots and not the normal columns and rows. Two coordinates, X and Y, set the actual location of the sprite:

- X is the horizontal direction and Y is the vertical direction.
- X has 512 values and Y has 256 values.

Note this is in fact more dots or positions than there are on the high-resolution screen, but this allows sprites to move smoothly and progressively off and onto the screen. Collisions between sprites and with other graphic characters can be detected even when they're off the screen.

Note The position of a sprite is always measured from the dot or bit in the top left-hand corner of the sprite's 24 x 21 grid. This applies whether or not the bit is part of your design and displayed on the screen.

The following table shows the VIC II registers into which you have to put X and Y values to determine the position of sprites:

SPRITE POSITION REGISTERS

REGISTER	SPRITE NUMBER	POSITION COORDINATE
53248	0	X
53249	0	Y
53250	1	X
53251	1	Y
53252	2	X
53253	2	Y
53254	3	X
53255	3	Y
53256	4	X
53257	4	Y
53258	5	X
53259	5	Y
53260	6	X
53261	6	Y
53262	7	X
53263	7	Y
53264	MOST SIGNIFICANT BIT for all sprites	

Vertical Position (Y)

To set the vertical position of any sprite simply put a Y value in the VIC II register address corresponding to the sprite number and the Y position. Y can be any number in the range from 0 to 255. Later on, experiment with different values of Y to discover for yourself when the sprite starts to come onto the screen and when it goes right off. Y = 0 is off the top of the screen.

Horizontal Position (X)

Setting the horizontal position of your sprite is a little more complex. If the value that you want for X is in the range of 0 to 255, you handle it in the same way as you did the Y coordinate and simply put a value into the VIC II register from the table above. There are, however, 512 possible values for the X co-ordinate and you cannot store a value greater than 255 in a single 8 bit byte, which, of course, these registers are.

To overcome this, the VIC II has another register, 53264, which serves all 8 sprites. This register is handled as 8 separate bits and each sprite has one bit of the same bit number as the sprite number. Therefore, each sprite actually has 9 bits available to represent X values over 255. This 9th bit, bit position 8, is the Most Significant Bit. With 9 bits you can hold a maximum value of 511: the first 8 bits can hold 255 and the 9th bit holds 256.

X Values over 255

The most satisfactory way to handle this will be to work through an example. We'll take sprite 0. Remember that we're now dealing with a 9 bit binary number that can hold a maximum of 511 — the 9th bit having a value of 256 when it holds a 1.

So for sprite 0, for values up to 255, its X register is 53248. For values over 255, it has available to it bit 0 of the common register 53264. That is, bit 0 of register 53264 is sprite 0's 9th bit. Let's imagine what this 9 bit binary number looks like when X = 255.

0 1111 1111

The ninth bit is the one on the far left and because X = 255, that ninth bit is unused. It is resting at 0. This value of X places the top left-hand corner of the sprite grid as far to the right on the screen as it will go at this stage. The whole grid is still well in from the right-hand edge of the screen. Now when we move the sprite one more position to the right (X = 256) its 9 bit binary number will look like this:

1 0000 0000

Register 53248 now has a value of 0 and bit 0 of register 53264 has a 1 in it. A further movement of 1 position to the right (X = 257) will create:

1 0000 0001

where bit 0 of register 53264 continues to have a 1 in it and bit 0 of the X register (53248) gathers a 1. You can go on increasing 53248 until it reaches 255 again. By the time it does, however, your sprite will be well and truly off the screen. Actually, you only need about 85 increases to X to move the sprite off the right hand edge of the screen.

In summary, then, for values X *up to* 255, you simply POKE the value into the X register corresponding with the sprite you want to use. For values of X *greater than* 255, you

```
POKE 53264,PEEK(53265)OR(2↑SN)
```

where SN is the sprite number. This puts the sprite in the 256th position. Now to keep moving the sprite to the *right,* you continue POKEing more values into the X register.

Moving the Sprite to the Left

To move the sprite to the left (even if it's off the screen, to the right), you simply decrease the value of X.

```
POKE 53264,PEEK(53264)AND(255-(2↑SN))
```

We've said *simply,* but there's a little more to it than that. Actually the value of X decreases until X = 256 when there's still a 1 in the ninth bit and a zero in the X register. Then to decrease to 255, the 1 in the ninth bit has to be removed and the remaining 255 returned to the X register. Here's how it looks in binary numbers (still assuming there's only one sprite in operation — sprite 0). In binary, register 53264 will be 0000 0001. Its first bit (bit 0 for the 0 sprite) is set to 1.

Register 53264		0000 0001
Logically AND'd with 255	1111 1111	
minus 2↑SN	0000 0001	1111 1110
Result		0000 0000

Notice how the 1 in bit 0 of register 53264 has been replaced with a 0. From here on, in BASIC, you would put the X register (53248) into a loop with X = 255 TO 1 STEP–1. This would continue to move the sprite to the left and eventually right off the screen.

TURNING OFF THE SPRITE

When you have finished with a sprite it has to be turned off, otherwise it will continue to be displayed on the screen until you remove it with RUN/STOP and RESTORE.

Turning a sprite off is just the reverse of turning it on. *On and off* is controlled by register 53269 where each sprite is represented by one bit. When the bit for a particular sprite holds 0, the sprite is off and when the bit holds a 1, the sprite is on. This is consistent with turning off and on the bits in your own or programmable objects.

To turn off a sprite then, you have to replace (with a zero) the 1 you put in the bit to turn it on. This is done with the following BASIC statement:

```
POKE 53269,PEEK(53269)AND(255-(2↑SN))
```

This is the same logical operation we employed to get rid of the Most Significant Bit in the *movement left* routine above.

A BASIC PROGRAM

We have all the information we need now to put a sprite on the screen and to move it. We'll do it in BASIC first. Normally, you would load your sprite data in by using DATA statements just as you did with RUNMAN. You know how to do this, so to save you a bit of typing with this exercise I have designed a beautiful solid block character — ie, all 63 bytes will contain 255! Type in Program Number 24.

Program # 24

```
5 PRINT CHR$(147)
10 FOR N=0TO62:POKE 832+N,255: NEXT
20 POKE 2040,13:  REM POINTER
30 POKE 53287,7:  REM COLOR
40 POKE 53269,PEEK(53269)OR1:REM TURNON
50 POKE 53248,100:  REM X COORDINATE
60 POKE 53249,100:  REM Y COORDINATE
```

When you RUN this program a yellow rectangular sprite will appear on the screen. You do not need to turn off the sprite, of course, until the program has finished with it, but try this:

```
POKE53269,PEEK(53269)AND254
```

and the sprite will disappear. Now type:

```
POKE53269,PEEK(53269)OR1
```

and the sprite will come on again. Now add these lines to the BASIC program:

```
55 FORN=0TO20
60 POKE53249,(100+N):NEXT
```

When you RUN it now the sprite will move down the screen as Y varies in the loop started at line 55. Now type in as direct command:

```
POKE52,48:POKE56,48:CLR (RETURN)
```

This will lower the top of RAM to 12288. Now replace lines 10 and 20 of your BASIC program with these:

```
10 FORN=0TO62:POKE12288+N:NEXT
20 POKE2040,192
```

When you RUN it now you'll get the same display as you got the first time — but there is a difference. After lowering ramtop, line 10 has placed the sprite data in the 64 bytes starting from 12288. Then in line 20, the pointer address 2040 (for sprite 0) has been loaded with 192. The VIC II now knows that the sprite data starts at that address — $64 \times 192 = 12288$.

From a practical point of view, there seems little advantage in going beyond BASIC to load your sprite data into RAM. If you store this data above ramtop you can SAVE it to tape as you did with RUNMAN and then dispose of the BASIC loading program altogether. The BASIC READ. . .DATA statements are very effective in this type of application and easy to use.

Now that you've stored your sprite data, the rest of it is relatively easy. The remaining controls for sprite operations consist only of a variety of BASIC POKEs and PEEKs. These you can quite easily translate into machine language. Many machine language programmers, in fact, write first in BASIC, get the program working, and then convert it to machine code. Now enter Program Number 25.

Program # 25

```
1 POKE52,48:POKE56,48:CLR
2 PRINT CHR$(147)
5 FORN=0TO62:POKE12288+N,255:NEXT
10 N=13312
20 READD:IFD=-1THEN35
30 POKEN,D:N=N+1:GOTO20
35 FORN=0TO19 READA:POKE13500+N,A:NEXT:
   GOTO55
40 DATA 169,192,141,249,7,169,3,141,40,
   208,173,21,208,9,2,141,21,208
41 DATA 169,170,141,2,208,169,150,141,3,
   208,169,192,141,248,7,169,7,141,39
42 DATA 208,173,21,208,9,1,141,21,208,
   169,150,141,1,208,169,5,141,0,208
43 DATA 32,188,52,173,30,208,41,1,201,1,
   208,3,32,199,52,238,0,208,169
44 DATA 255,205,0,208,208,231,238,0,208,
   173,16,208,9,1,141,16,208,32,188,52
45 DATA 238,0,208,169,85,205,0,208,208,
   243,173,16,208,41,254
46 DATA 141,16,208,169,0,141,0,208,173,
   23,208,41,254,141
47 DATA 23,208,173,23,208,41,252,141,21,
208,96,162,5
48 DATA 160,155,136,208,253,202,208,248,
   96,169,1,141,23,208,96,-1
49 DATA 162,5,160,255,136,208,253,202,
   208,248,96,173,23
```

```
50 DATA 208,9,1,141,23,208,96
55 SYS13312:GOTO55
```

When you RUN this program a cyan block will appear and then a yellow block will enter from the left and move across the screen. When it hits the cyan block, it will expand in the Y direction and then move to the right, disappearing off the screen on that side.

There are several points of interest in this program. In the first place, the cyan block was created with the same routine we used in Program 24, except that we translated the BASIC program into machine language. Maybe that's not so startling, but the graphic effect which occurs when the yellow block passes over the cyan block certainly is! *It obscures the cyan block.* For games programmers and graphic artists of all kinds, that means three dimensional animated displays made easy.

In technical terms, it's what's known as *sprite priority.* In this program, the yellow block is sprite 0 and the cyan block is sprite 1. In terms of 3D displays, it means that the sprite with the lower number is the one at the *front* of the screen and the sprite with the higher number is further back *into* the screen. The sprite with the lower number will obscure the sprite with the higher number when they pass on the same track.

The next thing to notice is that the yellow sprite reaches a position where X = 255 about 4 or 5 centimetres from the right-hand edge of the screen. Then bit zero of 53264 (the ninth bit for sprite 0) is loaded with 1. Sprite 0 then moves to the right and off the screen as the value of X is increased.

The last thing to notice is what happens when the yellow sprite touches the cyan sprite. The yellow sprite doubles in size in the Y direction. Collisions between sprites and sprites, and sprites and other graphic characters, are signalled in the VIC II registers. Each time the yellow sprite moves one position to the right, the program branches off to check whether there has been a collision. You can do this in BASIC as well as machine language but it is the speed of machine language that makes this collision detection feature such an effective graphic display to use. The machine language program carries out the branching and the work needed to expand the sprite so quickly that the sprite appears to move smoothly across the screen without pause.

How Fast Can a Sprite Move?

Line 49 of the BASIC program contains the machine language codes for the timing loops that control the speed of the yellow sprite. The numbers 10 and 255 near the start of line 49 are the loop counters. Try changing the 10 to 1 to

experiment with the speed of the sprite. This effectively leaves only the inside loop to control the speed. Now reduce the 255 to 1 also. This gives you the maximum speed you can get — but it's probably altogether too fast for anything you're ever likely to need!

Combining Sprite Graphics with Regular Graphics

Try this. Add the following line to your BASIC program:

```
50 FOR N=1 TO 24:PRINT"QWERTYUIOPASDFGHJKLZXCVBNM
QWERTYUIOP":NEXT
```

and change line 35 so that the last number in it is 50. Now when you RUN the program, that set of ordinary characters will be displayed on the screen. The interesting thing happens when the sprites go into their act. It illustrates how you can use sprites with regular graphic characters. You could, for example, create sprite spaceships and then shoot at them with proper little torpedoes of your own design!

When you type RUN for this ammended program, you'll also notice an appreciable delay before the sprites start to work. This delay is caused by the time it takes to load all the data into RAM before lines 50 and 55 can be actioned.

The sprite data and the machine language program itself have been loaded above ramtop. After you've RUN the BASIC program once, stop it with the RUN/STOP key. You'll have to hold the key down until the yellow sprite has disappeared. (Why do you think this is necessary?) Now type in as a direct command:

```
SYS 13312
```

This calls the machine language program directly and the sprites will appear on the screen immediately. Now you can go further and NEW out the whole BASIC program. The SYS 13312 call will still bring the machine language program into operation and you can SAVE the program as we did earlier. If now you type in lines 50 and 55 of the BASIC program and RUN, you'll get immediate action again. It demonstrates how you can call a machine language program from a BASIC program.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 192	169,192	LDA 192
(b)	Store accumulator in address 2041	141,249,7	STA 2041
(c)	Load accumulator with 3	169,3	LDA 3
(d)	Store accumulator in address 53288	141,40,208	STA 53288
(e)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(f)	Logically OR accumulator with 2	9,2	ORA 2
(g)	Store accumulator in address 53269	141,21,208	STA 53269
(h)	Load accumulator with 170	169,170	LDA 170
(i)	Store accumulator in address 53250	141,2,208	STA 53250
(j)	Load accumulator with 150	169,150	LDA 150
(k)	Store accumulator in address 53251	141,3,208	STA 53251
(l)	Load accumulator with 192	169,192	LDA 192
(m)	Store accumulator in address 2040	141, 248,7	STA 2040
(n)	Load accumulator with 7	169,7	LDA 7
(o)	Store accumulator in address 53287	141,39,208	STA 53287
(p)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(q)	Logically OR accumulator with 1	9,1	ORA 1
(r)	Store accumulator in address 53269	141,21,208	STA 53269
(s)	Load accumulator with 150	169,150	LDA 150
(t)	Store accumulator in address 53249	141,1,208	STA 53249
(u)	Load accumulator with 5	169,5	LDA 5
(v)	Store accumulator in address 53248	141,0,208	STA 53248
(w)	Jump to subroutine at 13500	32,188,52	JSR 13500
(x)	Load accumulator with contents of address 53278	173,30,208	LDA 53278
(y)	Logically AND accumulator with 1	41,1	AND 1
(z)	Compare accumulator with 1	201,1	CMP 1
(aa)	Branch on condition that result is not 0	208,3	BNE
(ab)	Jump to subroutine at 13511	32,199,52	JSR 13511
(ac)	Increment content of address 53248 by 1	238,0,208	INC 53248
(ad)	Load accumulator with 255	169,255	LDA 255

(ae)	Compare accumulator with contents of address 53248	205,0,208	CMP 53248
(af)	Branch on condition that result is not 0	208,231	BNE
(ag)	Increment contents of addres 53248 by 1	238,0,208	INC 53248
(ah)	Load accumulator with contents of address 53264	173,16,208	LDA 53264
(ai)	Logically OR accumulator with 1	9,1	ORA 1
(aj)	Store accumulator in address 53264	141,16,208	STA 53264
(ak)	Jump to subroutine at 13500	32,188,52	JSR 13500
(al)	Increment contents of address 53248 by 1	238,0,208	INC 53248
(am)	Load accumulator with 85	169,85	LDA 85
(an)	Compare accumulator with contents of address 53248	205,0,208	CMP 53248
(ao)	Branch on condition that result is *not* 0	208,243	BNE
(ap)	Load accumulator with contents of address 53264	173,16,208	LDA 53264
(aq)	Logically AND accumulator with 254	41,254	AND 254
(ar)	Store accumulator in address 53264	141,16,208	STA 53264
(as)	Load accumulator with 0	169,0	LDA 0
(at)	Store accumulator in address 53248	141,0,208	STA 53248
(au)	Load accumulator with contents of address 53271	173,23,208	LDA 53271
(av)	Logically AND accumulator with 254	41,254	AND 254
(aw)	Store accumulator in address 53271	141,23,208	STA 53271
(ax)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(ay)	Logically AND accumulator with 252	41,252	AND 252
(az)	Store accumulator in address 53269	141,21,208	STA 53269
(ba)	Return	96	RTS
(bb)	Load X register with 5	162,5	LDX 5
(bc)	Load Y register with 255	160,255	LDY 255
(bd)	Decrement Y register by 1	136	DEY
(be)	Branch on condition that result is *not* 0	208,253	BNE
(bf)	Decrement X register by 1	202	DEX

(bg)	Branch on condition that result is *not* 0	208,248	BNE
(bh)	Return	96	RTS
(bi)	Load accumulator with contents of address 53271	173,23,208	LDA 53271
(bj)	Logically OR accumulator with 1	9,1	ORA 1
(bk)	Store accumulator in address 53271	141,23,208	STA 53271
(bl)	Return	96	RTS

Explanation of the Machine Language Program

The machine language instructions in lines 40 to 55 of the BASIC program are no more than the PEEKs and POKEs of BASIC sprite operation translated into machine code. It's easy to do and gives you full access to the speed that machine language offers.

A Note About Sprite Pointers

Before we start the line by line explanation of this program, we should pause to look at an important point concerning our use of sprite pointers. Notice that in both lines (a) and (l) we've loaded the accumulator with 192. Line (a) is the pointer for sprite 1 and line (l) is the pointer for sprite 0. What we're saying, then, is that the same set of sprite *data* will be used for both sprite 1 and sprite 0. $192 \times 64 = 12288$ so the VIC II will go to address 12288 for the 64 bytes of sprite data for both sprite 1 and sprite 0.

It's important for us to distinguish between *sprite pointers* and *sprite data.* The VIC II doesn't care that a particular set of 64 bytes of data is referenced to more than one sprite. The VIC II simply multiplies the value you've put into the pointer address by 64, goes there and picks up the data, whatever it is. It means, in fact, that you can have the same set of sprite data for all the sprites if you want to. This would be a convenient way, for example, to create a fleet of identical spaceships.

On the other hand, you could set up any number of different sets of sprite data (provided they could be accommodated within the 16K of memory in Bank 0) and then, during the program, change any sprite's shape to one of them, just by changing the value in the sprite pointer address. If you're using BASIC, you'll POKE. If you're using machine language, you'll LDA and STA in the pointer address.

Lines (a) to (k) set up sprite 1 and put it on the screen .

Lines (a) and (b) put the value 192 into the sprite pointer address.

Lines (c) and (d) load the color code for cyan and store it in the color register for sprite 1.

Lines (e) to (g) turn on sprite 1. The BASIC commands POKE53289,PEEK (53289)OR(2↑SN) become Load the accumulator with the contents of address 53289, logically OR with 2 then store it back in address 53289.

Lines (h) and (i) store a value of 170 in address 53250 to set the X coordinate for sprite 1.

Lines (j) and (k) then put the Y coordinate value for the sprite in the appropriate VIC II register, 53251. At this point, sprite 1 is set up and ready to pop onto the screen when RUN.

Lines (l) to (v) set up sprite 0 and put it in its starting position, just off the screen.

Lines (l) and (m) put the value 192 into the pointer address for sprite 0.

Lines (n) and (o) load the color code for yellow and store it in the color register for sprite 0.

Lines (p) to (r) turn on sprite 0 in the same way as lines (e) (g) turned on sprite 1.

Lines (s) to (v) establish the X and Y coordinates for sprite 0 — its starting position just off the screen on the left-hand side. You can verify the VIC II registers used here by checking them against the table of position registers given above.

Line (w) jumps the program to a subroutine at 13500. This subroutine is the timing or delay loop routine which slows the movement of the sprite. Up to the point when X = 255, the loop which moves the sprite will jump back to this line to include the delay. We chose 13500 as the address for this subroutine only because it was obviously far enough above the machine language program not to overlap it. If memory space was tight, though, we'd count from 13312 to determine the most economic place to start it. We wouldn't leave any unused bytes between the machine language program and the subroutine.

Collision Detection

Lines (x) to (aa) carry out the routine that detects collisions between the two sprites. The VIC II uses two registers to determine whether there has been a collision between two sprites or between a sprite and some other graphic character. These are single byte registers and each sprite is represented by one bit of the same number, ie, sprite 0 uses bit 0.

Sprite to Sprite Collisions

In BASIC, the statement is:

IF PEEK(53278) AND X= X THEN *action*

where X represents the decimal value of the bit position for each particular sprite: 0 = 1 through to sprite 7 = 128. When a collision takes place, the VIC II puts a 1 into the appropriate bit and the address then has the equivalent decimal value. This value then remains in the address until the address is looked at.

Sprite to Other Character Collisions

In BASIC, the statement is:

IF PEEK(53279) ND X= X THEN *action*

and the explanation is the same as for sprite to sprite collisions.

Line (x) loads the contents of address 53278 into the accumulator.

Line (y) logically ANDs the accumulator with 1. This is sprite 0 whose bit is bit position 0. If there is a 1 in this bit it will have a decimal value of 1. This, of course, does not affect address 53278 itself because we are now dealing only with what is in the accumulator.

Line (z) compares the contents of the accumulator with 1.

Line (aa) contains the branch instruction that becomes operative when the result of the comparison is not 0 — ie, when the accumulator does not hold a 1. The branch instruction jumps the program three bytes forward to line (ac). If the accumulator does hold a 1, however, the result of the comparison will be 0 and the program will simply move on to the next instruction — line (ab).

Line (ac) becomes operative if no collision has been detected. It moves the sprite one position to the right by incrementing the contents of register 53248 (the X position register) by 1.

Line (ad) loads the accumulator with 255 —.the largest value possible for X before you have to use the special operation to continue the movement to the right.

Line (ae) compares the contents of the accumulator (255) with the contents of address 53248 (the X position register).

Line (af) branches the program back to line (w) until X = 255. When X = 255, the result of the comparison will be 0 and the program will be allowed to move on to line (ag).

Line (ag) increments the contents of address 53248 (the X value) by 1.

Line (ah) loads the accumulator with the contents of address 53264, the Most Significant Bit register.

Line (ai) logically ORs the accumulator with 1.

Line (aj) returns the result of the operation in line (ai) to address 53264. This means that address 53264 now holds a 1 in bit 0 and that X has reached 256.

Line (ag): A Special Note. You could omit line (ag) from the program and the sprite would still continue to move to the right. Lines (ah) to (aj) actually do the work of moving the sprite past the 255 'barrier' by putting a 1 into the 9th bit (address 53264). At this point, putting a 1 into address 53248 — the operation in line (ag) — simply takes the X position register from 255 back to zero. If you did omit line (ag), though, you'd get a nasty little jump as the sprite passed the X = 255 position.

Lines (ak) to (ao) create a loop that increments the value in the X position register (53248) until X = 85. X = 85 takes the sprite neatly off the screen. This sequence effectively completes the movement program, leaving only a few loose ends to be tidied up before we get to the subroutines.

Lines (ap) to (ar) deal with the problem of the sprite's screen re-entry after a complete cycle of the program has been made. At line (ao), the Most Significant Bit register (53264) holds a 1 in bit 0. Consequently, when the program goes back and starts again (as we've told it to do in line 55 of the BASIC pro-

gram), it simply places the yellow sprite in position 256, 4cm or so in from the right-hand edge of the screen, and lets it run from there to the right hand edge. It does this, of course, because X has a value greater than 255. The problem is solved by replacing the 1 in 53264 with 0.

Line (ap) loads the accumulator with the contents of address 53264.

Line (aq) logically AND's the contents of the accumulator with 254.

Line (ar) returns the result of line (aq) to address 53264.

Lines (as) to (az) turn off the sprites and return the expanded sprite to its regular size. The lines are not much more than a translation of the corresponding BASIC statements. There is, however, one point worth noting. Line (ay) logically AND's the accumulator with 252. In binary, 252 is 1111 1100 and this causes both bits 0 and 1 to be returned to 0, thereby turning both sprites off.

The Subroutines

Lines (bb) to (bh) create the delay routine. It's the same set of two nested loops that we've used on a number of earlier occasions. Notice that no attempt has been made to store the X or Y registers on the stack because neither of them has been used elsewhere in the program.

Lines (bi) to (bl) create the subroutine that expands the sprite when it collides with the other sprite.

Expanding and Contracting Sprites

You can expand sprites vertically and horizontally — that is, in the Y direction and the X direction. You can also shrink them back to their original size.

The VIC II has two registers for this purpose: one for the Y values and one for the X values. These are single byte registers and each sprite is served by a single bit of the same number as the sprite it serves. In BASIC, the operations are carried out like this:

To expand vertically

```
POKE 53271,PEEK (53271)OR(2 ↑ SN)
```

To contract or return to original size:

POKE 53271,PEEK(53271)AND(255-(2 ↑ SN))

To expand horizontally

POKE 53277,PEEK(53277)OR(2 ↑ SN))

To contract or return to original size:

POKE 53277,PEEK(53277)AND(255-(2 ↑ SN))

You can only expand a sprite in one direction at a time but if you use both expansion statements, one after the other, machine language will perform the sequence so fast that the expansion in both directions will appear to take place simultaneously.

Line (bi) loads the accumulator with the contents of address 53271.

Line (bj) logically OR's the contents of the accumulator (the contents of address 53271) with 1.

Line (bk) returns the result of the calculation in line (bj) to address 53271. The process in these three lines has been no more than a simple translation of the BASIC procedure.

Line (bl) returns the program to line (ab).

Returning the Sprite to its Regular Size

Lines (au), (av) and (aw) contain the machine language version of the BASIC statement that contracts the sprite to its regular size. If this step is not taken, the sprite will return from the left side still at double size. When you put values into addresses, they stay there until you enter instructions that change them or remove them!

10
Sound

The Commodore 64 is an extremely powerful sound unit. The chip that controls the sound (the 6581 Sound Interface Device — SID for short), is essentially a three voice electronic music synthesiser and sound effects generator. With it you can control pitch, volume, tonal color, wave form and envelope generation. It includes programmable filters, voice synchronisation and ring modulators.

The SID chip does for sound what the VIC II does for graphics. It works through 29 registers from address 54272 to 55295 and similar to graphics, the access is through POKEs and PEEKs to these registers.

For the average user, there are probably two main areas of interest in the C-64 as a sound device: the creation of music and the generation of sound effects for games. Speed isn't generally of any great importance in the creation of music, so the need for machine language may not be very great. In fact, it's a whole lot easier just to use BASIC. There may be times, however, when you want to include a music routine in a program that's otherwise written completely in machine language. In these cases it will be neater and better to write the music in machine code too. Fortunately, it's not hard. You just have to convert the PEEKs and POKEs to machine language.

It is in the area of sound effects for games, however, that the control of sound through machine language comes into its own.

In this chapter we'll look at machine language programs for both music and sound effects. The first program creates a little single octave piano that you can play on and the second, a game called *UFO Shooting Gallery* to demonstrate the use of sound effects. Now go ahead and type in Program Number 26.

Program # 26

```
10 N=828
20 READD:IFD=-1THEN100
30 POKEN,D:N=N+1:GOTO20
40 DATA 32,228,255,201,0,240,249,201,57,
    176,245,201,49,144,241
42 DATA 170,189,103,3,141,1,212,189,111,
    3,141,0,212,189,119,3,141,33,208
43 DATA 169,17,141,4,212 169,15,141,24,
    212,169,9,141,5,212
44 DATA 138,72,162,155,160,255,136,208,
    253,202,208,248,104,170
46 DATA 169,0,141,4,212,141,5,212,76,60,
    3,-1
100 FORN=920TO943
200 READA:POKEN,A:NEXT
300 DATA 17,19,21,22,25,28,32,34,37,63,
     154,227,177,214,94,175
350 DATA 1,3,4,5,7,13,10,9
400 PRINTCHR$(147):SYS828
```

When you RUN the program, the screen will go blank and you will be able to use keys 1 through 8 as if they were keys of a piano. No other keys are active. When you want to stop, press RUN/STOP and RESTORE.

There are two good reasons for choosing keys 1 to 8 in an exercise like this. The first is that the character codes for keys 1 to 8 are in sequence from 49 to 57, making programming easier. None of the other keys are arranged in code sequence on the keyboard. The second reason for choosing 1 to 8 is that it makes sense for playing a musical scale! The point to note is that you should think about the keys you choose for game control. Choosing the right keys can make your programming task a lot easier.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Jump to subroutine at 65508 kernal routine GETIN	32,228,255	JSR 65508
(b)	Compare accumulator with 0	201,0	CMP 0
(c)	Branch if result is 0	240,249	BEQ
(d)	Compare accumulator with 57	201,57	CMP 57
(e)	Branch if carry flag set	176,245	BCS
(f)	Compare accumulator with 49	201,49	CMP 49
(g)	Branch if carry flag cleared	144,241	BCC
(h)	Transfer accumulator to X register	170	TAX
(i)	Load accumulator with contents of address 871 + X	189,103,3	LDA 871,X
(j)	Store accumulator in address 54273	141,1,212	STA 54273
(k)	Load accumulator with contents of address 879 + X	189,111,3	LDA 879,X
(l)	Store accumulator in address 54272	141,0,212	STA 54272
(m)	Load accumulator with contents of address 886 + X	189,119,3	LDA 887,X
(n)	Store accumulator in address 53281	141,33,208	STA 53281
(o)	Load accumulator with 17	169,17	LDA 17
(p)	Store accumulator in address 54276	141,4,212	STA 54276
(q)	Load accumulator with 15	169,15	LDA 15
(r)	Store accumulator in address 54296	141,24,212	STA 54296
(s)	Load accumulator with 9	169,9	LDA 9
(t)	Store accumulator in address 54277	141,5,212	STA 54277
(u)	Transfer X register to accumulator	138	TXA
(v)	Push accumulator onto stack	72	PHA
(w)	Load X register with 155	162,155	LDX 155
(x)	Load Y register with 255	160,255	LDY 255
(y)	Decrement Y register by 1	136	DEY
(z)	Branch on condition that result is *not* 0	208,253	BNE
(aa)	Decrement X register by 1	202	DEX
(ab)	Branch on condition that result is *not* 0	208,248	BNE
(ac)	Pull accumulator off stack	104	PLA

(ad)	Transfer accumulator to X register	170	TAX
(ae)	Load accumulator with 0	169,0	LDA 0
(af)	Store accumulator in address 54276	141,4,212	STA 54276
(ag)	Store accumulator in address 54277	141,5,212	STA 54277
(ah)	Jump to address 828	76,60,3	JMP 828

Explanation of the Machine Language Program

Before we commence our line by line discussion of the program, let's consider the range of things the program has to do. It has to:

- read the keyboard. The kernal routine, GETIN, is probably the easiest way to do this.
- eliminate all the keys other than keys 1 to 8.
- set the volume, waveform and attack/decay rate.
- set values for high frequency and low frequency for each note.
- turn off the note once it's been played.

Lines (a) to (c) contain the GETIN routine. This routine puts the character code of the key being pressed into the accumulator. If no key is being pressed, the accumulator holds a 0. The compare instruction in line (b) checks for a 0 (no key press) and the branch instruction returns the program to line (a) until a key is pressed.

Lines (d) to (g) prevent any keys outside the range code 49 to code 57 from operating the program.

Line (d) compares the value held in the accumulator with 57.

Line (e) sets the carry flag to 1 if the value in the accumulator is greater than 57. Any code larger than 57 will therefore cause the branch instruction to return the program to the start.

Line (f) compares the value held in the accumulator with 49.

Line (g) clears the carry flag (or resets it to 0) if the value in the accumulator is less than 49. Any code smaller than 49 will therefore cause the branch instruction to return the program to the start.

Lines (h) to (l) set the high and low frequency values for each note. Line 300 of the BASIC program contains the data. The first eight numbers are the high frequency values and the second eight numbers are the low frequency values. The third set of eight numbers (in line 400) are standard color codes. I included them just to add a little interest to the screen as a tune was being played!

Line (h) transfers the value held in the accumulator (at the end of line (g)) to the X register. The value for X is therefore the code of the key (49 through 57) that has been pushed.

The values for the high frequency part of the notes are stored in addresses 920 to 927 and the values for the low frequency part of the notes are stored in addresses 928 to 935. The high frequency values will be stored in register 54273 and the low frequency values will be stored in register 54272 (when the appropriate key is pressed). The values for the color codes are stored in addresses 936 to 943. These have to be stored progressively in register 53281, the register for screen color. To get these values out of the table between 920 and 943 and into the appropriate registers, we load the accumulator with the contents of the address *indexed by the X register.*

Line (i) loads the accumulator with the contents of address 871 + X. We start with 871 because it's the number that will return 920 when the lowest possible value for X is added to it. The lowest possible value for X is 49, which occurs when key 1 is pressed. Taking key 1 is an example, the accumulator is loaded with 871 + 49 = 920.

Line (j) (following the example for key press 1) stores the value held in address 920 in register 54273. If key 2 had been pressed, the value for X would have been 50 and the next value in the table (871 + 50), 921, would have gone into register 54273.

Lines (k) and (l) carry out the same function for the low frequency values. Notice that the base address is 879 — the number that will return 928 when 49 (the lowest possible value for X) is added to it.

Lines (m) and (n) carry out the same function for the color codes, storing their values in register 53281. The base address is 887, the number that will return 936 when 49 is added to it.

Lines (o) to (t) are straightforward translations of BASIC POKEs.

Lines (o) and (p) set the waveform.

Lines (q) and (r) set the volume.

Lines (s) and (t) set the attack/decay rate.

Lines (u) to (ad) contain the nested delay loops we've used on a number of earlier occasions. Notice that the X register has been transferred to the stack in line (v).

Lines (ae) to (ag) turn off the waveform and attack/decay settings.

Line (ae) loads the accumulator with 0.

Lines (af) and (ag) store this 0 in registers 54276 and 54277 — the registers for waveform and attack/decay.

Line (ah) contains the jump instruction that returns the program to the start, ready for the next note to be played.

UFO Shooting Gallery

Type in Program Number 27. I suggest you SAVE it to tape or disk as soon as you've finished keying it in and before you RUN it. It's a long program and a minor error in transcription could mean the loss of the whole thing.

Program # 27

```
10 N=12500
20 READD:IFD=-1THEN52
30 POKEN,D:N=N+1:GOTO20
31 DATA 169,193,141,249,7,173,21,208,9,
    2,141,21,208,169,8,141
```

```
32 DATA 40,208,169,100,141,2,208,160,
    255,140,3,208,173,31,208,41,2,201
33 DATA2,240,16,152,72,32,113,49,32,188,
    52,104,168,136,208,231
34 DATA 76,212,48,173,21,208,41,253,141,
    21,208,169,192,141,248,7,173,21
35 DATA 208,9,1,141,21,208,169,7,141,39,
    208,173,29,208,9,1,141,29
36 DATA208,173,23,208,9,1,141,23,208
38 DATA 169,90,141,0,208,141,1,208
39 DATA 162,15,142,24,212,169,129,141,4,
    212,169,15,141
43 DATA 5,212,169,40,141,1,212,169,200,
    141,0,212,32,188,52,32,188,52
44 DATA 202,208,224,169,0,141,4,212,141,
    5,212,173,21,208,41,254,141,21
45 DATA 208,76,212,48,96,32,228,255,201,
    0,240,248,32,154
47 DATA 49,162,40,169,120,157,143,5,169,
    1,157,143,217,32,218,52,202
48 DATA 208,240,32,218,52,162,40,169,32,
    157,143,5,202,208,248,162,15
49 DATA 142,24,212,169,33,141,4,212,169,
    128,141,5,212,169,100,141,1
50 DATA 212,169,223,141,0,212,32,218
51 DATA 52,202,208,227,169,0,141,4,212,
    141,5,212,96,-1
52 FORN=0TO18:READD:POKE13500+N,D:NEXT
53 DATA 138,72,152,72,162,5,160,200,136,
    208,253,202
54 DATA 208,248,104,168,104,170,96
```

```
55 FORN=0TO9:READD:POKE13530+N,D:NEXT
57 DATA 138,72,162,255,202,208,253,104,
   170,96
60 FORN=0TO62:READD:POKE12288+N,D:NEXT
61 DATA 0,30,96,0,255,254,1,255,255,3
62 DATA 255,255,3,255,254,15,255,254
64 DATA 31,255,254
65 DATA 63,255,254,127,255,252,255,255
66 DATA 255,255,255,255,255,255,255
67 DATA 255,255,255,255,255,254,255,255
68 DATA 254,31,255,252,63,255,248,63
69 DATA 255,0,7,255,0,7,248,0,3,208,0
70 FORN=0TO62:READD:POKE12352+N,D:NEXT
71 DATA 0,0,0,0,126,0,0,255,192,1,255
72 DATA 224,3,255,224,3,255,248,7,255
73 DATA 248,127,255,255,255,255,255,170
74 DATA 170,170,255,255,255,255,255,255
75 DATA 63,255,248,0,0,0,0,0,0,0,0,0,0
76 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0
80 POKE53280,0:POKE53281,0:PRINTCHR$(147
):SYS12500
```

When you RUN the program, the screen will turn black and an orange flying saucer will appear at the bottom of the screen. It will then move upwards till it eventually disappears off the top of the screen. Your mission (should you accept it) is to destroy the alien craft with a laser beam. Any key will fire the laser but I suggest you use f7. It's best to use the special function keys whenever you can in games because it saves wear and tear on the regular keys.

The first two POKEs in line 80 of the BASIC program are the ones that turn the screen color black. They put the color code for black (0) into the two registers that control the color for screen border and background respectively.

The program loads in from address 12500. This is above ramtop which has been set to 12288 by line 5 of the BASIC program. 12500 was chosen because it was far enough away not to run the risk of overlapping the sprite

data. The data for sprite 0 is loaded from 12288 upwards and the data for sprite 1 from 12352 upwards (lines 60 and 70 of the BASIC program).

The program uses two delay routines. The first is loaded at line 13500 (line 53 of the BASIC program) and the second at line 13530 (line 55 of the BASIC program). Memory is not at a premium in this program so we can set these locations from the very beginning. If you wanted to, though, you could crunch up the routines at a later stage by counting out the positions for the most economical starting addresses for the various routines. Remember, however, that the location of the sprite data is fixed by what you put into the pointer address.

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load accumulator with 193	169,193	LDA 193
(b)	Store accumulator in address 2041	141,249,7	STA 2041
(c)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(d)	Logically OR accumulator with 2	9,2	ORA 2
(e)	Store accumulator in address 53269	141,21,208	STA 53269
(f)	Load accumulator with 8	169,8	LDA 8
(g)	Store accumulator in address 53288	141,40,208	STA 53288
(h)	Load accumulator with 100	169,100	LDA 100
(i)	Store accumulator in address 53250	141,2,208	STA 53250
(j)	Load Y register with 255	160,255	LDY 255
(k)	Store Y register in address 53251	140,3,208	STY 53251
(l)	Load accumulator with contents of address 53279	173,31,208	LDA 53279
(m)	Logically AND accumulator with 2	41,2	AND 2
(n)	Compare accumulator with 2	201,2	CMP 2
(o)	Branch on condition that result is 0	240,16	BEQ
(p)	Transfer Y register to accumulator	152	TYA
(q)	Push accumulator onto stack	72	PHA
(r)	Jump to subroutine at 12650	32,113,49	JSR 12650
(s)	Jump to subroutine at 13500	32,188,52	JSR 13500
(t)	Pull accumulator off stack	104	PLA
(u)	Transfer accumulator to Y register	168	TAY
(v)	Decrement Y register by 1	136	DEY

(w)	Branch on condition that result is *not* 0	208,231	BNE
(x)	Jump to address 12500	76,212,48	JMP 12500
(y)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(z)	Logically AND accumulator with 253	41,253	AND 253
(aa)	Store accumulator in address 53269	141,21,208	STA 53269
(ab)	Load accumulator with 192	169,192	LDA 192
(ac)	Store accumulator in address 2040	141,248,7	STA 2040
(ad)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(ae)	Logically OR accumulator with 1	9,1	ORA 1
(af)	Store accumulator in address 53269	141,21,208	STA 53269
(ag)	Load accumulator with 7	169,7	LDA 7
(ah)	Store accumulator in address 53287	141,39,208	STA 53287
(ai)	Load accumulator with contents of address 53277	173,29,208	LDA 53277
(aj)	Logically OR accumulator with 1	9,1	ORA 1
(ak)	Store accumulator in address 53277	141,29,208	STA 53277
(al)	Load accumulator with contents of address 53271	173,23,208	LDA 53271
(am)	Logically OR accumulator with 1	9,1	ORA 1
(an)	Store accumulator in address 53271	141,23,208	STA 53271
(ao)	Load accumulator with 90	169,90	LDA 90
(ap)	Store accumulator in address 53248	141,0,208	STA 53248
(aq)	Store accumulator in address 53249	141,1,208	STA 53249
(ar)	Load X register with 15	162,15	LDX 15
(as)	Store X register in address 54296	142,24,212	STX 54296
(at)	Load accumulator with 129	169,129	LDA 129
(au)	Store accumulator in address 54276	141,4,212	STA 54276
(av)	Load accumulator with 15	169,15	LDA 15
(aw)	Store accumulator in address 54277	141,5,212	STA 54277
(ax)	Load accumulator with 40	169,40	LDA 40
(ay)	Store accumulator in address 54273	141,1,212	STA 54273
(az)	Load accumulator with 200	169,200	LDA 200
(ba)	Store accumulator in address 54272	141,0,212	STA 54272
(bb)	Jump to subroutine at 13500	32,188,52	JSR 13500
(bc)	Decrement X register by 1	202	DEX

(bd)	Branch on condition that result is *not* 0	208,224	BNE
(be)	Load accumulator with 0	169,0	LDA 0
(bf)	Store accumulator in address 54276	141,4,212	STA 54276
(bg)	Store accumulator in address 54277	141,5,212	STA 54277
(bh)	Load accumulator with contents of address 53269	173,21,208	LDA 53269
(bi)	Logically AND accumulator with 254	41,254	AND 254
(bj)	Store accumulator in address 53269	141,21,208	STA 53269
(bk)	Jump to address 12500	76,212,48	JMP 12500
(bl)	Return	96	RTS
(bm)	Jump to subroutine at 65508 kernal routine, GETIN	32,228,255	JSR 65508
(bn)	Compare accumulator with 0	201,0	CMP 0
(bo)	Branch on condition that result is 0	240,248	BEQ
(bp)	Jump to subroutine at 12698	32,154,49	JSR 12698
(bq)	Load X register with 40	162,40	LDX 40
(br)	Load accumulator with 120	169,120	LDA 120
(bs)	Store accumulator in address 1423 + X	157,143,5	STA 1423,X
(bt)	Load accumulator with 1	169,1	LDA 1
(bu)	Store accumulator in address 55695 + X	157,143,217	STA 55695,X
(bv)	Jump to subroutine at 13500	32,218,52	JSR 13530
(bw)	Decrement X register by 1	202	DEX
(bx)	Branch on condition that result is *not* 0	208,240	BNE
(by)	Jump to subroutine at 13530	32,218,52	JSR 13530
(bz)	Load X register with 40	162,40	LDX 40
(ca)	Load accumulator with 32	169,32	LDA 32
(cb)	Store accumulator in address 1423 + X	157,143,5	STA 1423,X
(cc)	Decrement X register by 1	202	DEX
(cd)	Branch on condition that result is *not* 0	208,248	BNE
(ce)	Load X register with 15	162,15	LDX 15
(cf)	Store X register in address 54296	142,24,212	STX 54296
(cg)	Load accumulator with 33	169,33	LDA 33

(ch)	Store accumulator in address 54276	141,4,212	STA 54276
(ci)	Load accumulator with 128	169,128	LDA 128
(cj)	Store accumulator in address 54277	141,5,212	STA 54277
(ck)	Load accumulator with 100	169,100	LDA 100
(cl)	Store accumulator in address 54273	141,1,212	STA 54273
(cm)	Load accumulator with 223	169,223	LDA 223
(cn)	Store accumulator in address 54272	141,0,212	STA 54272
(co)	Jump to subroutine at 13530	32,218,52	JSR 13530
(cp)	Decrement X register by 1	202	DEX
(cq)	Branch on condition that result is *not* 0	208,227	BNE
(cr)	Load accumulator with 0	169,0	LDA 0
(cs)	Store accumulator in address 54276	141,4,212	STA 54276
(ct)	Store accumulator in address 54277	141,5,212	STA 54277
(cu)	Return	96	RTS

Long Delay subroutine at 13500

(cv)	Transfer X register to accumulator	138	TXA
(cw)	Push accumulator onto stack	72	PHA
(cx)	Transfer Y register to accumulator	152	TYA
(cy)	Push accumulator onto stack	72	PHA
(cz)	Load X register with 5	162,5	LDX 5
(da)	Load Y register with 200	160,200	LDX 200
(db)	Decrement Y register by 1	136	DEY
(dc)	Branch on condition that result is *not* 0	208,253	BNE
(de)	Decrement X register by 1	202	DEX
(df)	Branch on condition that result is *not* 0	208,248	BNE
(dg)	Pull accumulator off stack	104	PLA
(dh)	Transfer accumulator to Y register	168	TAY
(di)	Pull accumulator off stack	104	PLA
(dj)	Transfer accumulator to X register	170	TAX
(dk)	Return	96	RTS

Short Delay subroutine at 13530

(dl)	Transfer X register to accumulator	138	TXA
(dm)	Push accumulator onto stack	72	PHA
(dn)	Load X register with 255	162,255	LDX 255
(do)	Decrement X register by 1	202	DEX
(dp)	Branch on condition that result is *not* 0	208,253	BNE
(dq)	Pull accumulator off stack	104	PLA
(dr)	Transfer accumulator to X register	170	TAX
(ds)	Return	96	RTS

Explanation of the Machine Language Program

Compared with the other programs in this book, *UFO Shooting Gallery* would appear to be very long. It's deceptive, though, because it's really only a collection of a number of short routines. They are all tied together but they can usually be written and tested individually and even SAVEd as separate modules.

Lines (a) to (aa) set up sprite 1, the flying saucer. It was designed on the standard 24 × 21 sprite grid. At the end of this chapter there's a short BASIC program that converts binary numbers to their decimal equivalents. If you're having trouble with those conversions, you might find it useful.

Line (a) loads the accumulator with 193. Recall from the earlier discussion about sprites that this number is multiplied by 64 to give the starting address for the *sprite data.*

Line (b) loads the sprite pointer register (2041) with this address pointer for the sprite data (193 × 64 = 12352).

Lines (c), (d) and (e) turn on the sprite.

Lines (f) and (g) color the sprite orange.

Lines (h) and (f) set the X coordinate. This remains fixed throughout the program because the sprite is only moving up the screen.

Line (j) commences the Y coordinate at 255. A simple loop will decrement it to 0 but you'll need to glance down to line (v) to see where the decrement instruction and the following conditional branch complete the loop.

Lines (l) to (o) contain the routine for collision detection. They are virtually a straight translation of the BASIC command, IF PEEK (53279) AND X=X THEN. . . if the condition for the branch instruction in line (o) is met (ie, the result is 0) the program jumps forward 16 bytes to the explosion routine. If there has been no collision, however, the program simply passes on to the instruction in line (p) where the Y register is transferred to the accumulator.

Lines (p) and (q) transfer the Y register to the accumulator and from there to the stack. Lines (t), (u), (v) and (w) recover the Y register from the stack and continue with the loop. This is an important 'detour' for you to understand. Unless you know exactly what you're doing, it's advisable when you have an important value in a register to save it on the stack while your program does something else. In this program, for example, the GETIN routine affects all registers and Y is used elsewhere in the program. If you don't put the Y value on the stack, the spaceship jumps up and down quite alarmingly when you fire the laser. It costs you 4 bytes to put Y on the stack and pull it off again but you know the loop values will be right whatever happens elsewhere in the program.

Line (r) jumps the program to the subroutine at address 12650. This subroutine starts within the GETIN routine to read the keyboard and initiates the firing of the laser. If no key has been pressed, it allows the program to come straight back and the spaceship to keep going.

Line (s) jumps the program back to the long delay subroutine that keeps the spaceship moving at a reasonable speed.

Line (x) jumps the program back to the start and causes another spaceship to start moving up the screen. It becomes operative, of course, if no collisions have been detected.

Lines (y), (z) and (aa) turn off the spaceship sprite. They become operative when a collision has been detected and the conditional branch instruction in line (o) sends the program to line (y).

Lines (ab) to (aq) handle the representation of sprite 0, the explosion graphic.

Lines (ab) and (ac) load the sprite pointer register (2040) with the address pointer for the sprite data ($192 \times 64 = 12288$). Figure 10.2 shows how I designed my explosion on the sprite grid.

Lines (ad) to (ah) turn on the sprite and color it yellow.

Lines (ao) to (aq) set both the X and Y coordinates at 90.

Lines (ai) to (an) expand the sprite to double height and double width. They are a translation of the BASIC procedures.

Line (ar) and (ba) handle the sound effects.

Lines (ar) and (as) set the volume.

Lines (at) and (au) set the waveform.

Lines (av) and (aw) set the attack/decay rate.

Lines (ax) to (ba) set the high and low frequency values for the sound.

Lines (av), (bc) and (bd) create a loop that causes the volume of the sound to die away gradually. The loop also includes a jump to the long delay subroutine that controls the time the explosion will stay on the screen and the duration of the sound.

Lines (be) to (bg) turn off the sound.

Lines (bh) to (bj) turn off the sprite.

Line (bk) jumps the program back to the start for the presentation of another spaceship.

Line (bl) returns the program to line (r), the spaceship movement routine. It is linked with the conditional branch instruction in line (bo).

Line (bm) jumps the program to the GETIN routine, the keyboard scanner.

Lines (bn) and (bo) check for a key press. If no key has been pressed, the branch instruction in line (bo) takes the program back to line (bl) — the Return instruction.

Line (bp) jumps the program to the laser sound subroutine that starts at address 12698. It does this, of course, only if a key has been pressed and it has been allowed to pass the branch instruction in line (bo).

Line (bq) becomes operative after the sound routine. It commences the routine that draws the laser beam across the screen. Line (bq) itself sets up the X register as the loop counter and index for the screen and color addresses.

Line (br) loads the code for our little line segment.

Line (bt) colors the line segment white (code 1).

Lines (bs) and (bu) handle the addressing mode. We're using the absolute indexed by X register addressing mode where the first screen address is 1423 and the corresponding color address is 55695.

Lines (bv) to (bx) complete the loop that puts the laser beam across the screen.

Line (by) jumps the program to the delay routine at 13530 and this holds the beam on the screen for a brief period.

Lines (bz) to (cd) create a loop similar to the one which drew the laser beam. The only difference is that we're using spaces instead of line segments. The spaces erase the segments that create the beam.

Lines (ce) to (ct) handle the sound effects for the laser beam.

Line (ce) loads the X register with 15 for volume. From this point down to line (co), the program simply sets up the various values for the sound. You should experiment with these values to discover for yourself the variations you can get.

Line (co) jumps the program to the delay subroutine at 13530. This loop determines the duration of the laser sound.

Lines (cp) and (cq) constitute part of the loop. They decrement the value of X causing the volume to be decreased at each cycle of the loop. Notice also how this branches back to line (cf) where the progressively smaller values of X are stored in the volume register.

Lines (cr) to (ct) turn off the sound and return the program to line (bp).

Lines (cv) to (dk) contain the routine for the long delay.

Lines (dl) to (ds) contain the routine for the short delay.

We've used both of these delay loops in a number of earlier programs. Check them out for yourself and try to explain how they work.

A Program for Binary to Decimal Conversion

Designing your own sprites and defining them in decimal numbers involves a fair amount of detailed calculation. Why not let your computer do it for you? When you've designed your sprite on its 24 x 21 grid, all you have to do is key the binary numbers from the grid into Program Number 28. It will give you the decimal equivalents. Be sure to list your bytes correctly on the grid before you start.

Program # 28

```
1 PRINTCHR$(147)
2 PRINT"ENTER 8-BIT BINARY NUMBER UNDER
  THE DOTS"
3 PRINT"  ........"
4 INPUTA$
5 IF LEN(A$)<>8THEN PRINT"ENTER 8 BITS
  PLEASE":GOTO3
6 DV=0:C=0
7 FORN=8TO1STEP-1:C=C+1
8 DV=DV+VAL(MID$(A$,C,1))*2↑(N-1)
9 NEXTN
10 PRINT SPC(11) CHR$(145)
   "BINARY EQUALS";DV;"DECIMAL":GOTO3
```

Fig. 10.1 SPRITE # 1 SPACESHIP

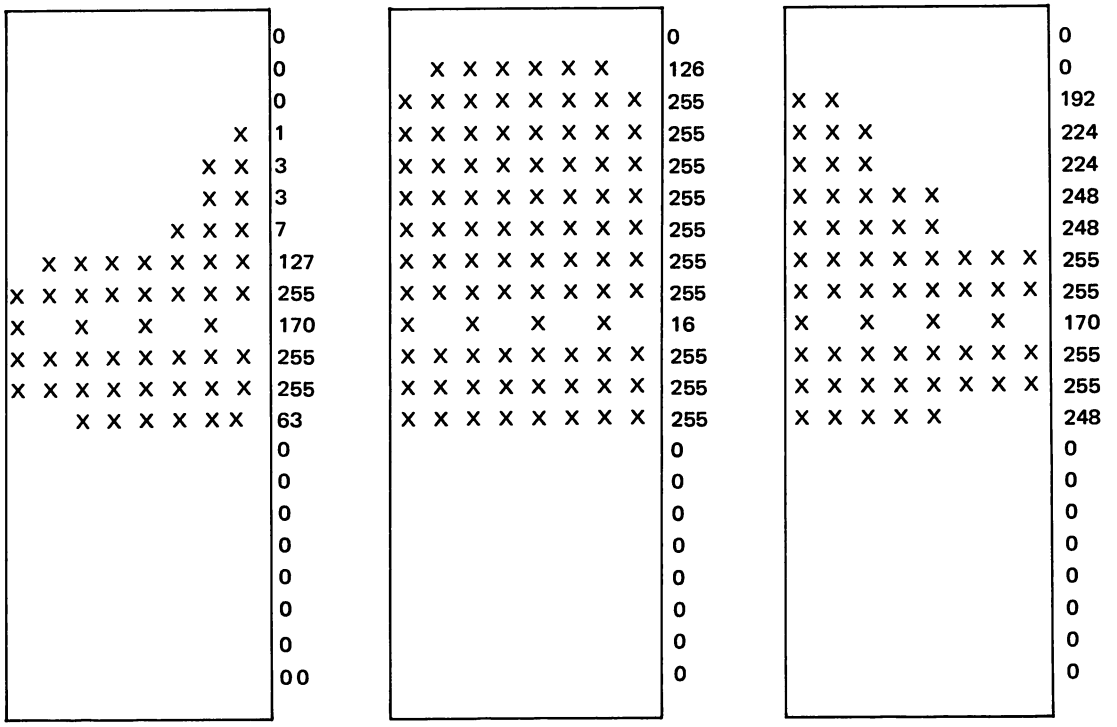
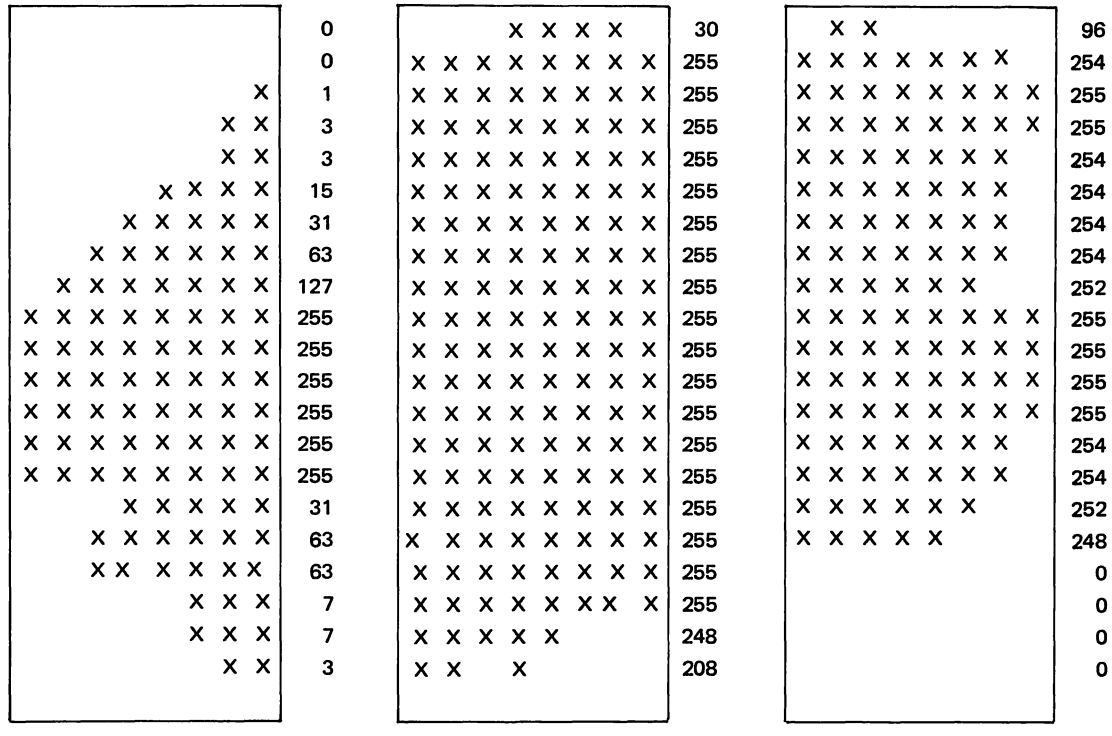


Fig. 10.2 SPRITE # 0 EXPLOSION CLOUD



11

About Your Own Programs Storing & Debugging

WHERE DO YOU STORE YOUR OWN MACHINE LANGUAGE PROGRAMS?

This is an important question unless you can save your programs all your work will be lost when you turn off your computer. Furthermore, however you handle the location of your programs in memory, you've still got to think about conserving memory. There are four ways you can store your machine language programs:

- In the cassette file buffer
- As a REM statement in a BASIC program
- Above a lowered *ramtop*
- In RAM from address 49152 to address 53247.

Each has advantages and disadvantages, so let us look at each in detail.

Cassette File Buffer

This is an area set up by the C-64 at addresses 828 to 1019 inclusive, a total of 192 bytes.

If you are not using files then the area is available to hold your machine language programs. For machine language routines, 192 bytes is a very useful amount of memory. Most of the programs in this book use this area of RAM.

The buffer is not affected by the BASIC statement NEW (but it does disappear if you use your tape recorder). To use the space effectively you should therefore include your machine language codes within a BASIC program as DATA statements along with the instructions needed to POKE the codes into the buffer. To this extent you do use a little extra memory.

Nevertheless, for a small BASIC program (and certainly for your developmental work) it is an excellent storage place.

REM Statement in a BASIC Program

This method avoids the problems associated with storage in the cassette file buffer by letting you store your machine language programs as a line of a BASIC program. You can then use the line in any other BASIC program you write and, of course, store it on tape like any other line from a BASIC program.

You do it by writing a four line BASIC program that sets up a *dummy* REM line and then transfers your machine language program from its DATA line in the program to the REM line. Try this. Type Program Number 29. The machine language program is the one we used earlier to draw a line.

Program # 29

```
10 REM.................
20 FORN=2054TO2069:READD:POKEN,D:NEXT
30 DATA 162,40,169,120,157,143,5,169,7,
    157,143,217,202,208,243,96
50 PRINT CHR$(147):SYS 2054
```

Notice that we typed 16 fullstops after the REM statement and that the FOR . . . TO instruction in line 20 also accommodates a count of 16. That's because there are 16 bytes in the machine language program in the DATA line and each will replace one of those fullstops. It's not essential, by the way, that you use fullstops. Any character will do.

When you RUN the program, the only screen display you'll get will be the word READY. When now you LIST the program however, you'll find that the fullstops have been replaced by a series of peculiar looking words and characters. These words and characters are the keywords in C-64 BASIC that represent the codes you POKEd into the REM line with line 30 of the BASIC program. Don't worry if there's not always a keyword to correspond with a particular number: the instruction code is still stored in the memory address.

Notice in line 20 that we used 2054 as the starting address for our machine language program. There was a good reason for doing so. Were you to type and RUN the following program:

```
10 REM . . . . . (any number of fullstops)
20 FOR N=2048 TO 2060
30 PRINT PEEK (N),N
40 NEXT
```

you would get the following display:

0	2048
23	2049
8	2050
10	2051
0	2052
143	2053
46	2054
46	2055
46	2056

and so on to the number of fullstops following REM.

Don't do it yet though because you'll replace the previous program and we want to continue to use it for a while! Try it for yourself at the end of this section.

The number 46 is the Chr$ code for a fullstop. Notice that these fullstops start at address 2054. That is, the first character after REM occupies address 2054. 2048, of course, is the first address in RAM for BASIC programs.

Recalling the Machine Language Program

To recall the machine language program, first clear the screen with CLR/HOME and SHIFT. Now type SYS 2054 (the address from which the machine language program starts) and the program will run — ie, the line will be drawn immediately.

Now we're ready to see the most useful application of the exercise. LIST the BASIC program and delete lines 20, 30 and 40 from it so that only line 10 remains: ie, it's a one line program consisting only of line 10. Now type SYS 2054. The machine language program runs! The line is drawn. The machine language program has been stored in memory in the REM line.

And it follows, of course, that if we want to we can use this line 10 in any other BASIC program we like to key in. We simply number the lines of the new program to follow on from 10 and call the machine language program from within the BASIC program as required.

It also follows that we can SAVE the machine language program contained in line 10 along with any other lines of BASIC we've added to it. To your C-64, it's just like any other BASIC REM line.

The method is an excellent one for handling your machine language programs. It uses minimum memory and tape space. There are, however, some limitations to its use.

1. Some of the Chr$ codes can cause problems. When you RUN the BASIC program and then LIST it, it prints in white from part of the way down line 10. This is caused by the 5 — the Chr$ code for white. You can overcome this sort of problem, however, by enclosing the fullstops in the REM line within quotation marks. If you do this, the program will LIST normally but you do have to remember that the addition of the quotation marks moves the starting and ending addresses up by one. In this program, 2054 becomes 2055, 2069 becomes 2070 and SYS 2054 becomes SYS 2055. And the inclusion of Chr$ code 0 (zero) in your machine language program causes a similar sort of problem. Its effect when you LIST the program, however, is to cause *nothing* to be printed after its occurrence in the REM line. The problem, though, is more apparent than real. If you PEEK at the memory location into which you POKEd the machine code, you'll find that it's all there and that it works with the usual SYS call. If code zero is the only problem, you don't need to worry about using quotes.

2. Like any other line in a BASIC program, the REM line will accommodate only 80 characters. That will be enough for many but not all machine language programs you'll want to use. You can get around the limitation by creating a second REM line — provided that you allow for the link, the line number and the REM statement itself. It's easy enough to do. A little PEEKing will soon locate the addresses for you.

3. The number of fullstops following REM must not be fewer than the number of machine codes in the DATA line. If you haven't provided enough fullstops, the codes will be POKEd in over the top of the next line number and the material in that line. It's better to provide a few more spaces than not enough! In any case, if you find you need more address spaces in the REM line, you can add them in the usual way that you'd edit any other line.

Storage Above Ramtop

This is probably the best area in which to hold your machine language programs — especially the longer ones. The area above ramtop can be protected from BASIC so that nothing short of switching off the computer will affect it. It is also reasonably easy to SAVE on tape and to LOAD from this area.

In the C-64, the area of RAM set aside for your BASIC programs extends from 2048 to 40959. The pointer (or address) of the limit of memory is stored in addresses 55 and 56. When you type PRINT PEEK (55) + 256 *PEEK (56) and RETURN you will receive a reply of 40960. This is actually one more than the number of addresses available to you.

Storage of your BASIC programs (and most other information) starts from the bottom of this area (2048) and goes upwards. Strings, however, are stored from 40959 *downwards*. So anything (like a machine language program) put at the top of this RAM must be protected from strings — and vice versa.

Your procedure then is to put into addresses 55 and 56 some other address which the system of BASIC in the C-64 will accept as the upper limit for BASIC programs. The area of RAM from the *new upper limit* of RAM to 40959 is thus completely removed from the BASIC system and you are virtually free to do what you want with it. Notice also that the C-64 reads the new Ramtop as *one above* the address to which it can actually go (just as it did with 40960). Your protected area thus includes this ramtop address. It will often be the starting address for your machine language program.

If you PEEK (55) and (56) separately you will find that they contain 0 and 160 respectively. This is the normal 6510 representation of Least Significant Byte and Most Significant Byte; Least Significant in the first address and Most Significant in the next higher address. And from what we've seen earlier, we know that 0 and 160 in that order represent 40960. Therefore, by POKEing other numbers into these two addresses we can place this ramtop address wherever we want it. POKEing into these addresses (followed by NEW or CLR or loading in as new tape program) will set the new ramtop.

In Chapter 7 we POKEd the Most Significant Byte (56) with 48 and lowered ramtop to 12288. This was a special case as we were bringing up C-64 characters to be changed to a new character design.

If your program is not involved with a special character set, you are free to set ramtop anywhere you want to. Calculate the Least Significant Byte and Most Significant Byte of an address and POKE them into addresses 55 and 56. You need only as many addresses as there are bytes in your program; anything more is wasting memory.

Saving to Tape

As we've said before, this area above ramtop is outside the normal BASIC system and material on it cannot be SAVEd in the usual way. In Chapter 7, however, we used a little routine that avoided the problem by breaking the program into two parts — treating the machine language data in much the same way as a file, and the BASIC program (held below ramtop) as a regular BASIC program.

When you use the cassette recorder, you don't have to use a file name. If you don't want to name the file, you need only use the following procedure. We'll assume we have ramtop lowered and protected — at, say, 12288. Our machine language program is sitting safely above ramtop and the BASIC program is located in normal RAM.

1. Set up the cassette drive ready to SAVE.

2. Type the following lines as direct commands, each line followed by RETURN.

POKE 193,0: POKE 194,48	(L.S.B. and M.S.B. of lowest address of this protect area to be saved)
POKE 174,0: POKE 175,160	(Same for upper address of this area)
POKE 185,1: POKE 186,1	(185 sets secondary address, 186 is device number, 1 for cassette)
SYS 62957	Address in ROM for a SAVE routine

Note: These POKEs save all of RAM from 12288 to 40959. The operation takes a long time so you should POKE addresses 174 and 175 with an upper address just sufficient to cover the program you want to SAVE.

3. When the display reads PRESS RECORD AND PLAY ON TAPE, press the RECORD and PLAY keys on the cassette drive at the same time.

4. When the cassette drive stops and READY is displayed on the screen, the part of the program above ramtop will have been SAVEd. DO NOT TOUCH THE CASSETTE DRIVE YET.

5. Type SAVE and RETURN. Now the BASIC part of the program will be SAVEd in the regular way.

6. To LOAD the program on this cassette back into the computer you must first set ramtop to the correct address: so *note the address of ramtop* on the cassette label as soon as you've finished SAVEing.

Loading from Tape

1. Set up the cassette drive ready for LOADing.

2. Type as a direct command: POKE 56,48:LOAD RETURN (using the POKEs relevant to the ramtop you are using)

3. When the screen display reads PRESS PLAY ON TAPE, start the cassette drive.

4. When the recorder stops and the screen display reads READY, the *first part* of the program will have been loaded into its previous home above ramtop.

5. Type LOAD and RETURN. Now the BASIC program will be LOADed in the regular way.

This is a very effective procedure but you have to remember to protect and lower ramtop before LOADing (step 2) and then LOAD twice if you have two parts to your program.

Storage in RAM from 49152 to 53247

This is a 4K block of RAM freely available for the storage of your machine language programs. The block is outside the BASIC system but you can POKE to these addresses and PEEK at them from a regular BASIC program. You can also SAVE to tape any programs held in the block by the method we've described earlier.

DEBUGGING

Prevention is Better than Cure

1. Be clear about the objective of your program — ie, exactly what it is that you want to achieve. *Write it down.*

2. Break down the objective into the major parts or 'modules' you'll need to achieve the objective. Then break down each of the modules into the individual steps required to achieve the objective of the module. *Write them down.*

3. Write things out fully. Flowcharts and fancy ways of abbreviating documentation are fine for experienced programmers but it's safer to use a few extra words and diagrams in the planning stage if you're a beginner. Don't worry if your documentation doesn't look quite like the neat little listings you see in magazines. These sorts of listings are often a challenge to the understanding of experienced programmers!

4. Don't rely on your memory for such things as codes and addresses. One wrong number will cause your program to crash.

Checking for Bugs

1. Loop counters and index registers: have they been set up correctly?

2. Loops and branches: do they go to the correct addresses? Are the relative numbers and two's complements correct?

3. Loops and branches: do they include the counter?

4. Index registers: are loops returning them to their starting value on each pass?

5. L.S.B. and M.S.B. are they in the correct order? Have they been calculated correctly?

6. Addresses: have you distinguished correctly between the code for the address itself and the code for its contents?

7. The carry flag and the zero flag: check their *conditions.*

8. Transfer instructions: check their *directions.*

9. The stack rule: last in will be first out.

10. Chr$ codes and screen codes: have you used the appropriate code?

11. The screen memory maps: check that character addresses are correctly matched with color addresses.

12. Missing screen printout? Make sure your print color is different from your screen color.
13. Indexes: have their contents been saved on the stack?
14. Subroutines: are they putting the correct material into the registers?
15. Machine instruction codes: are they correct?
16. Addressing modes: are you using the right one?

SUGGESTIONS FOR THE FUTURE

If you have operated all of the programs in this book and studied carefully how they work you should have a good working knowledge of machine language programming. If you want to write long involved programs, do it! If you write them in small modules you can test, debug and tape them module by module. It is not a difficult or tedious job to produce effective programs of considerable length.

As you progress, build up a library of your routines. Collect routines and ideas from magazines. Analyse them to make sure you know exactly how they work and modify them if you need to. But keep records of them all so that you know exactly what they are and how you can incorporate them in your own programming.

If you want to get really serious about programming your C-64 (either in BASIC or machine language), I strongly recommend you get a copy of Commodore's *Programmer's Reference Guide* for the C-64. It contains a wealth of information about the machine.

If your ambitions lean towards writing large machine language programs, I think you should consider acquiring Commodore's excellent C-64 machine language monitor cartridge. It contains an assembler as well as a number of other goodies for the machine language programmer. With the knowledge you now have from the study of this book you will have no trouble using the assembler properly. It is a great help if you do a lot of programming.

But finally, do not neglect your study of C-64 BASIC. It is a good and powerful high level language. Use the best of both worlds!

Good luck and have fun!

Appendix A
Some Simple Arithmetic

Most of the information on arithmetic needed for machine language programming has been included progressively along with the programs. However, it is important material to understand so it is all brought together in this appendix with some further material.

Before starting with the arithmetic it is as well to have a quick look at the 6510 itself. Not in great detail, but it may help you to understand what's behind the arithmetic.

THE 6510

The 6510 microprocessor is the heart of the C-64. Some of the other chips contribute a great deal to the operation of the computer but the 6510 is the Central Processing Unit. In broad terms the 6510 consists of the following major parts:

- The control unit which knows all the machine language instructions, interprets the program instructions and generally keeps its finger on everything.
- The registers, which are really a bunch of memories. The 6510 uses some of them for its own purposes. Others are available for the programmer to hold his data in while the 6510 does something to or with them.
- The arithmetic logic unit, which is not terribly bright but is extremely quick and accurate. It can add and subtract, do some logic operations and move some bits around.

The 6510 communicates through two main bunches of wires (each called a bus) and some special control wires with which we are not concerned. The first bus is the *data* bus, which consists of 8 wires, and which passes all the data information to, from and around the 6510.

The other bus is the *address* bus, along which go signals to tell the computer where to get information from (addresses in memory) and where to store information. This bus consists of 16 wires and can thus handle 16 pieces of information.

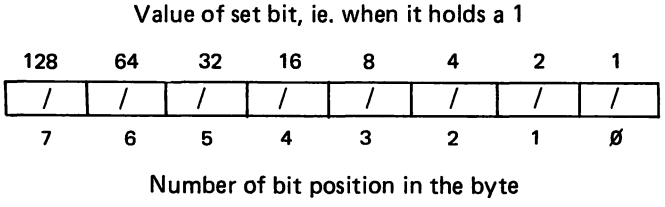
The 6510 can reasonably be thought of as a massive (though physically tiny) bunch of flip-flops or switches. Like most switches, they can have two states: off or on — or as we think of them, 0 or 1. Because the data information bus has 8 wires, the 6510 does all its work in bunches of eight bits, each of which can be on or off — ie, be 0 or 1. One unit (byte) of information handled by the 6510 is thus made up of eight of these *bits*.

BITS AND BYTES

Just as each letter in a word can be one of 26 and each digit in a telephone number can be one of ten, so each bit in a 6510 byte can be one of two possibilities — 0 or 1.

Any whole number you can think of (and you normally think of them in the decimal system) can also be expressed as a binary number — ie, a number in a system of counting that uses only 0s and 1s.

The following diagram represents a byte of 8 bits. We'll use the convention which numbers the bits from 0 on the right hand side to 7 on the left hand side.



So that we can see how the binary byte works and how it can represent decimal values, let us imagine that we have a byte handy and can start feeding unit impulses into its right-hand end.

Let us assume that each bit starts with a value of 0. In this condition it follows that the byte also has a total value of 0.

Now in comes the first unit pulse — call it a 1. Bit 0 will now hold a 1. The other bits still hold 0 so the whole byte has a value of 1.

When the next pulse comes in it can't put a second 1 in bit 0 since the capacity of bit 0 (or any other bit) is limited to a single 1. It therefore puts a 1 in bit 1 and restores bit 0 to 0. This gives the byte a total value of 2. The 1 in bit 1 is worth twice the 1 in bit 0.

Since bit 0 now holds 0 again it can be changed to 1 by a third pulse. With 1 in each of bits 0 and 1, the byte now has a value of 3.

A fourth pulse can't put an extra 1 into bits 0 or 1 since both contain 1 already. It therefore puts a 1 in bit 2 and restores bits 1 and 0 to 0. The 1 in bit 2 is worth twice the 1 in bit 1 — ie, 4.

This process can be repeated right across the byte. These principles emerge:

1. A 1 in any of bits 1 to 7 represents twice the value of a 1 in the bit position on its immediate right.
2. A bit position which holds 0 adds nothing to the value of the byte.
3. A byte with a 0 in each bit position has a value of 0 — the minimum value of a byte.
4. A byte with a 1 in each bit position has a value of 255 — the maximum value of a byte.

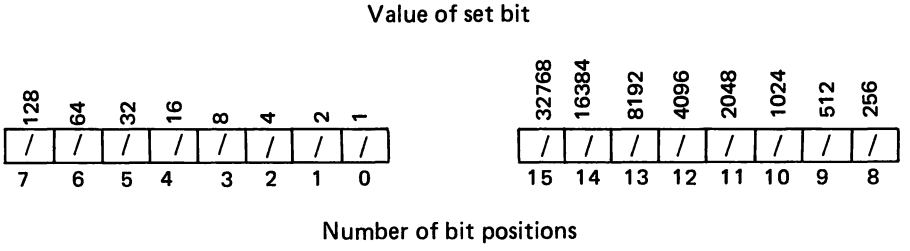
The decimal values of 1 in the different positions can be seen from the following table:

Bit #	Value of 1 in this bit
7	2^7 or 128
6	2^6 or 64
5	2^5 or 32
4	2^4 or 16
3	2^3 or 8
2	2^2 or 4
1	2^1 or 2
0	2^0 or 1

From all of this you can see how the 6510 handles numbers in the range 0 to 255. Let's now see how it handles larger numbers.

Obviously, if the maximum value of a byte is 255, you will need more than one byte to handle numbers from 255 onwards. This is quite simple. If you want to use a second byte the computer will lend you one to use together with the first and as an extension of it.

With two bytes on the job we can now think in terms of a diagram like this:



The byte on the left hand side works exactly like the one we have just looked at. If it is filled with 1s (and therefore has a value of 255), a further pulse will put a 1 into the second byte and restore all bits in the first byte to 0. After a further pulse we have a 1 in bits 0 and 8 for a value of 257.

By extending this you can see that a 1 in the 15th bit position has a value of 32768 and that the two bytes can hold a value in the range 0 to 65535.

It is important to note here that the effective value of a 1 in a given bit position in the right hand byte is 256 times the value of a 1 in the same position in the left hand byte.

This principle is also used in BASIC. You may be familiar with the instruction

```
PRINT PEEK(55) + 256 * PEEK (56)
```

This instruction looks at both addresses, multiplies the value in address 56 and then adds it to the value in address 55. The result (40960) is then held as a system variable by the C-64 in addresses 55 and 56. It is the first address above the top of RAM available for BASIC programs. If you PRINT PEEK (55), PEEK (56) — ie, look at the addresses separately — you will get the reply 0 and 160.

The 6510 knows how to do this in machine language. But you will have to remember to handle addresses and large numbers in two adjacent bytes.

LEAST SIGNIFICANT BYTE/MOST SIGNIFICANT BYTE

We've just seen that addresses and large numbers must be stored in two bytes. But equally important is to store these two parts of the address in the correct order.

If you won a prize of $1,000,056 you'd have no trouble distinguishing the most significant part of the prize from the least significant part. In the previous discussion about using two bytes we saw that the byte on the right-hand side of the diagram was 256 times the value of the other byte. So the byte on the right-hand side must hold the largest or most significant part of the address and the one on the left, the smaller or least significant part of the address.

The byte which holds the major part of the address is called the *Most Significant Byte* (sometimes the high byte) and the smaller part of the address, the *Least Significant Byte* (sometimes the low byte). These terms are used frequently and it is as well to be sure you are clear about their meanings.

It is critical that they be stored in the following order: Least Significant Byte in the first address and Most Significant Byte in the next higher address (larger number address). The 6510 *must* have address bytes in this order.

Calculation of Address Bytes

Let us assume you want to store an address of 43579 in memory. In one sense you could say that 43 thousand is the most significant part of this address. Unfortunately, however, if you put 43 in one byte, you can't put the balance of 579 in the other byte.

So this is how we do it. Firstly, we divide 43579 by 256. This gives us the number of sets of 256 contained in 43579 and something over. Next we multiply the number of whole sets of 256 by 256. The answer is called the Most Significant Byte of the address. The remainder (a number which must be less than 256) is the Least Significant Byte.

Our example, in figures, would look like this:

$$\begin{aligned} 43579 \div 256 &= 170 \text{ (plus something)} \\ 170 \times 256 &= 43{,}250 \\ 43579 - 43520 &= 59 \end{aligned}$$

59 is the Least Significant Byte and is stored in the first address. 170 is the Most Significant Byte and is stored in the next higher address. 43579 will therefore be represented as 59,170.

2'S COMPLEMENT REPRESENTATION

As you have seen from some of the programs in this book, you have to use 2's complement representation to specify backward jumps in conditional branch instructions.

That's because the 6510 does not do subtraction in the usual way. Rather, it performs this operation by adding 2's complement numbers.

Consider the byte of 8 bits which we've seen can hold a number in the range 0 to 255. From the byte diagram we see that the first bits (0 to 6) can hold numbers in the range 0 to 127. This leaves the eighth bit (bit 7) free to be used to signify whether the number is negative or positive. If bit 7 contains a 1 then the number is said to be negative. If bit 7 contains 0, the number is positive. These numbers are called *signed numbers* and cover the range from 127 through 0 to -128.

Unfortunately these signed numbers do not follow all of the normal rules of binary addition and so it is necessary to go another step: the two's complement.

The 2's complement representation does work correctly with binary addition and is now probably the most commonly used system in microcomputers.

For positive numbers the 2's complement is the same positive number. For negative numbers, however, we must make some calculations. First take the binary representation of the same positive number. Complement this binary. We *complement* by changing all the 0's to 1's and all the 1's to 0's. When we add 1 to the result we have the 2's complement of the negative number.

Example: 2's Complement of – 3

decimal + 3 is represented in binary by	00000011
Complement this (called 1's complement)	11111100
Add 1 to this figure	1
2's complement of -3 (decimal 253)	11111100

For our purposes, however, it is easier to calculate the 2's complement in decimals. Simply subtract the number of bytes from 256 and the result is the 2's complement we need for the instruction code. For example, to branch 3 bytes, 256 - 3 = 253, the 2's complement of -3.

LOGICAL OPERATORS

We used a logical operator in one of the programs as an efficient way of changing a number during the running of the program. Logical operators can

be very useful for simple bit manipulation. From time to time you may see them in programs published in magazines. It is worth getting to know something about them.

In 6510 machine language instructions, there are three logical operators: AND, OR and EXCLUSIVE OR. These operators work with the individual bits of the number, so for convenience of explanation we'll consider two such bits — A and B.

AND

If A and B are *both* 1 then the result of the operation is 1. All other combinations produce a 0 result.

OR

If *either* A or B is 1 then the result is 1. This includes the case where both A and B are 1.

EXCLUSIVE OR

If A or B is 1 then the result is 1. But if A and B are both 1 then the result is 0.

This can be illustrated by the following truth tables:

A	AND	B	Result
1		1	1
1		Ø	Ø
Ø		1	Ø
Ø		Ø	Ø

A	OR	B	Result
1		1	1
1		Ø	1
Ø		1	1
Ø		Ø	Ø

A	EXCLUSIVE OR	B	Result
1		1	Ø
1		Ø	1
Ø		1	1
Ø		Ø	Ø

Logical Operators in Action

Logical operators test the relationship between the bits of two separate bytes. They operate on the individual bits in the same bit positions in the two bytes. Their operation does not affect bits adjacent to each other.

The two bytes we're concerned with here are the number held in the accumulator and the number we've specified in our machine language program. When the logical operation between the two numbers has been completed, the result is returned to the accumulator.

Masking is a term sometimes used in connection with logical operators. In the following examples we'll see how logical operators can be used to mask certain parts of binary numbers held in the accumulator.

The procedure involves establishing the binary representation of the number in the accumulator and converting it according to the rules of the truth tables, to the decimal notation you need for your machine language program.

The following examples assume that the first number is in the accumulator and that the result of the operation will be returned to the accumulator.

Logical AND

Example 1			
In the accumulator	102	binary equivalent	01100110
Logically AND'd with	15	binary equivalent	00001111
Return to accumulator	6	binary equivalent	00000110
Example 2			
In the accumulator	102	binary equivalent	01100110
Logically AND'd with	240	binary equivalent	11110000
Return to accumulator	96	binary equivalent	01100000
Example 3			
In the accumulator	102	binary equivalent	01100110
Logically AND'd with	102	binary equivalent	01100110
Return to accumulator	102	binary equivalent	01100110

Note that:

- a 0 in the second number causes a 0 in the result (ie, a 0 will *mask* any 1's in the accumulator);
- a 1 in the second number allows the corresponding bit in the accumulator to fall through to the result — ie, the bottom line.

- a number AND'd by itself will return itself.

Logical OR

Example 1

In the accumulator	102	binary equivalent	01100110
Logically OR'd with	15	binary equivalent	00001111
Return to accumulator	111	binary equivalent	01101111

Example 2

In the accumulator	102	binary equivalent	01100110
Logically OR'd with	240	binary equivalent	11110000
Return to accumulator	246	binary equivalent	11110110

Example 3

In the accumulator	102	binary equivalent	01100110
Logically OR'd with	102	binary equivalent	01100110
Return to accumulator	102	binary equivalent	01100110

Note that:

- a 0 in the second number allows the corresponding bit in the accumulator to fall through unchanged;
- a 1 in the second number causes a 1 in the result (ie, 1's in the accumulator are allowed to come through unchanged but 0's are changed to 1's);
- a number OR'd by itself returns itself.

Logical EXCLUSIVE OR

Example 1

In the accumulator	102	binary equivalent	01100110
Logically EXCLUSIVE OR'd with	15	binary equivalent	00001111
Return to accumulator	105	binary equivalent	01101001

Example 2

In the accumulator	102	binary equivalent	01100110
Logically EXCLUSIVE OR'd with	240	binary equivalent	11110000
Return to accumulator	150	binary equivalent	10010110

Example 3

In the accumultor	102	binary equivalent	01100110
Logically EXCLUSIVE OR'd with	102	binary equivalent	01100110
Return to accumulator	102	binary equivalent	00000000

Example 4

In the accumulator	102	binary equivalent	01100110

Logically EXCLUSIVE OR'd with	255	binary equivalent	11111111
Return to accumulator	153	binary equivalent	10011001

Note that:

- a 0 in the second number allows the corresponding bit in the accumulator to fall through unchanged;
- a 1 in the second number returns the *complement* of the corresponding bit in the accumulator;
- a number EXCLUSIVE OR'd by itself returns 0;
- a number EXCLUSIVE OR'd with 255 returns the *complement* of the number in the accumulator.

ADDITION and SUBTRACTION

As we've seen previously, when we're communicating directly with the 6510, we can add and subtract numbers only if they are held in the accumulator. Here now are several other programs dealing with 8-bit and 16-bit addition and subtraction that you might find useful in the future. These programs are also good examples of the function of the carry flag.

To operate these programs you just use the BASIC loading program that occurs throughout the book and follow the suggestions given under each program. The complete listing is shown in Program Number 30.

Program # 30.

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 24,169,100,105,50,141,132,3,
    96,-1
50 SYS828:PRINT"900",PEEK(900)
```

8-Bit Addition

	Instruction	Decimal Code	Mnemonic
(a)	Clear Carry Flag	24	CLC
(b)	Load accumulator with 100	169,100	LDA 100
(c)	Add 50 to contents of accumulator	105,50	ADC 50
(d)	Store contents of accumulator in address 900	141,132,3	STA 900
(e)	Return	96	RTS

As well as substituting the above machine language program for the DATA lines in the BASIC program you should add a new line 50:

```
50 SYS 828: PRINT"900",PEEK(900)
```

When you RUN the program the screen will display: 900 150.

Explanation of the Machine Language Program

Line (a) clears the carry flag. In 8-bit addition this is a necessary precaution. The 6510 allows only one instruction for addition and one instruction for subtraction. Both of them are *with carry*. This means that the condition of the carry flag is brought into the operation. In the addition function, if the carry flag is set to 1, 1 will be added to the result of the addition *whether or not* it was caused by the addition. So before you start 8-bit addition, you clear carry flag just to make sure it's in the correct condition — ie, 0.

Line (b) loads the accumulator with 100. As we said before, addition and subtraction can only be carried out with numbers held in the accumulator.

Line (c) is the addition instruction to add 50 to the number in the accumulator. This is the end of the addition operation as such and the result of the addition is now stored in the accumulator.

Line (d) simply stores the accumulator in address 900 — you have to do something with the result of your addition!

Line 50 of the BASIC program recalls the machine language program with SYS 828, PEEKs address 900 and PRINTs the result on the screen.

Now let's see what happens when the result of an addition is greater than 255, the maximum that can be held in the accumulator. Substitute 250 for the 50 in line (c) of the machine language program. The result of the addition will now be obviously greater than 255. When you RUN the program now, you'll see that the screen shows 94 as the result. The correct answer, of course, is 350. So what has happened? The number greater than 255 has set the carry flag to 1 and regarding the carry flag as the ninth bit, the computer gives it the value of the carry over — ie, 256. In other words, 350 less 256 in the carry flag leaves 94 in the accumulator.

16-Bit Addition

	Instruction	Decimal Code	Mnemonic
(a)	Clear carry flag	24	CLC
(b)	Load accumulator with contents of address 900	173,132,3	LDA 900
(c)	Add contents of address 902 to accumulator	109,134,3	ADC 902
(d)	Store contents of accumulator in address 904	141,136,3	STA 904
(e)	Load accumulator with contents of address 901	173,133,3	LDA 901
(f)	Add contents of address 903 to accumulator	109,135,3	ADC 903
(g)	Store contents of accumulator in address 905	141,137,3	STA 905
(h)	Return	96	RTS

Add to your BASIC loading program the following lines:

```
50 POKE 900,156                          (L.S.B. of 5532)
60 POKE 901, 21                          (M.S.B. of 5532)
70 POKE 902,133                          (L.S.B. of 6789)
80 POKE 903,26                           (M.S.B. of 6789)
90 SYS 828
100 PRINT PEEK (904) + 256 * PEEK (905)
```

The program adds 6789 to 5532 and the POKE lines put the LSB and MSB of these numbers into addresses 900 to 903. The result is stored back in 904 and 905 and the number read with line 100. Note that all the numbers are stored in

two addresses, Least Significant Byte in the first address followed by the Most Significant Byte in the next higher address.

When you RUN the program you should get the answer 12321 on the screen. The complete listing is shown in program Number 31.

Program # 31

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 24,173,132,3,109,134,3,141,136,
    3,173,133,3
42 DATA 109,135,3,141,137,3,96,-1
50 POKE900,156   :REM L.S.B. OF 5532
60 POKE901,21    :REM M.S.B. OF 5532
70 POKE902,133   :REM L.S.B. OF 6789
80 POKE903,26    :REM M.S.B. OF 6789
90 SYS828
100 PRINTPEEK(904)+256*PEEK(905)
```

Explanation of the Machine Language Program

As the 6510 can only handle one byte of 8 bits at a time, the method of 16-bit addition is to add the two Least Significant Bytes together. then, together with any carry-over generated by the first addition, the two Most Significant Bytes are added together.

Line (a) clears the carry flag.

Line (b) loads the accumulator with L.S.B. of 5532.

Line (c) adds the L.S.B. of 6789 to the L.S.B. of 5532 (already in the accumulator).

Line (d) stores the result of the addition in address 904. Note that the addition generates a carry over. Notice also that we did *not* clear carry flag with the second addition. If the first addition generates a carry over it will have to be added to the sum of the M.S.B.'s.

Lines (e) and (f) execute the addition of the M.S.B.'s.

Line (g) stores the result of the addition in address 905.

8-Bit Subtraction

	Decimal Instruction	Code	Mnemonic
(a)	Set carry flag	56	SCF
(b)	Load accumulator with 100	169,100	LDA 100
(c)	Subtract 50 from contents of accumulator	233,50	SBC 50
(d)	Store contents of accumulator in address 900	141,132,3	STA 900
(e)	Return	96	RTS

Add the same lines to your BASIC loading program as for 8-bit addition. When you RUN the program you should get the display:

900 50

Now change the 50 in line (c) of the machine language program to 150. When you RUN the program you should get the result 206 instead of the correct answer of – 50. This shows that there has been a 'borrow' from the carry flag — ie, 256 from the carry flag less – 50 leaves 206 in the accumulator (note the 2's complement representation of – 50). The complete listing is shown in Program Number 32.

Program # 32

```
10 N=828
20 READD:IFD=-1THEN50
```

```
30 POKEN,D:N=N+1:GOTO20
40 DATA 56,169,100,233,50,141,132,3,
    96,-1
50 SYS828:PRINT"900",PEEK(900)
```

Explanation of the Machine Language Program

Line (a) sets the carry flag to 1. The carry flag reset to 0 would give a wrong result. As a general practice the set carry flag instruction is used at the start of all 8-bit subtraction operations.

From here on the program is substantially the same as the 8-bit addition program.

16-Bit Subtraction

	Instruction	Decimal Code	Mnemonic
(a)	Set carry flag	56	SCF
(b)	Load accumulator with contents of address 900	173,132,3	LDA 900
(c)	Subtract contents of address 902 from accumulator	237,134,3	SBC 902
(d)	Store contents of accumulator in address 904	141,136,3	STA 904
(e)	Load accumulator with contents of address 901	173,133,3	LDA 901
(f)	Subtract contents of address 903 from accumulator	237,135,3	SBC 903
(g)	Store contents of accumulator in address 905	141,137,3	STA 905
(h)	Return	96	RTS

Add to your BASIC loading program the following lines:

```
50 POKE 900,56          (L.S.B. of 6712)
60 POKE 901,26          (M.S.B. of 6712)
70 POKE 902, 133        (L.S.B. of 5509)
80 POKE 903,21          (M.S.B. of 5509)
```

```
90 PRINT "reverse heart": SYS 828
100 PRINT PEEK (904) + 256 * PEEK (905)
```

When you RUN this program you should get the correct answer, 1203 on the screen. The complete listing is shown in Program Number 33.

Program # 33

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 56,173,132,3,237,134,3,141,136,
    3,173,133,3,237,135,3
42 DATA 141,137,3,96,-1
50 POKE900,56:     REM L.S.B. OF 6712
60 POKE901,26:     REM M.S.B. OF 6712
70 POKE902,133:    REM L.S.B. OF 5509
80 POKE903,21:     REM M.S.B. OF 5509
90 SYS828
100 PRINTPEEK (904)+256*PEEK(905)
```

Explanation of the Machine Language Program

This program is essentially similar to the 16-bit addition and is also handled in two parts. The L.S.B.'s are subtracted and the result stored. Then the M.S.B.'s are subtracted and any borrow from the first subtraction is accounted for.

Note that the L.S.B. of 5509 is larger than that of the 6712. Consequently the first subtraction will generate a borrow and 1 will be subtracted from the result of the subtraction of the M.S.B.'s.

The set carry flag instruction is used again before the first subtraction. It is not used preceding the second subtraction. When a borrow has been caused by the L.S.B. subtraction it has to be accounted for in the subtraction of the M.S.B.'s.

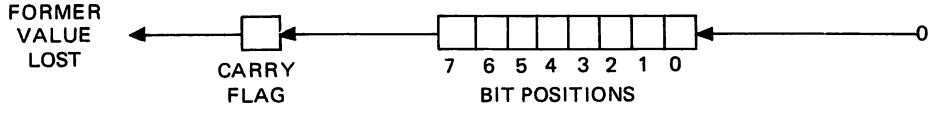
SHIFTS AND ROTATES

Shifts and rotates are among the less commonly used machine language instructions but they do have some important applications and from time to time you may run accross them in published programs. It's a good idea, therefore, to know something of the way they operate.

There are four shift and rotate instructions in the 6510 set. Like the logical operators, they work with the bits in a byte. Unlike the logical operators, however, they work with the values stored either in the accumulator or in a memory address. They therefore have several addressing modes.

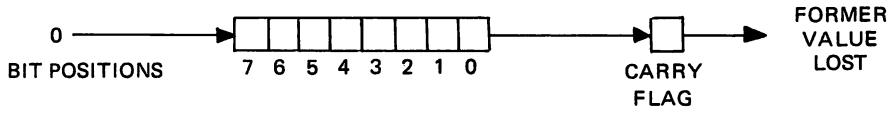
In the descriptions that follow we must be aware that there is a value in the byte made up of 0's and 1's and that the carry flag will also hold a value (either 0 or 1) when we start.

Arithmetic Shift Left



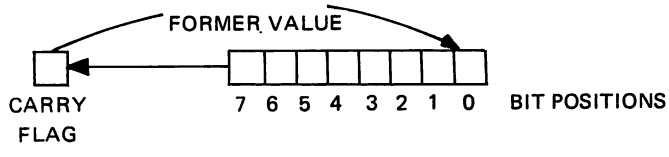
This instruction shifts the value in each bit, one bit position to the left. As the value in bit 0 moves to the left, a 0 then enters that bit and the value in bit position 7 moves into the carry flag. Whatever value was previously held in the carry flag is lost.

Logical Shift Right



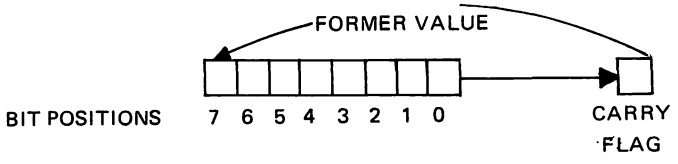
This instruction shifts the values in each bit, one bit position to the right. A 0 comes into bit position 7 to replace the bit value moving to the right. The value in bit 0 moves into the carry flag replacing the previous value which is lost.

Rotate One Bit Left



This instruction moves the values in each bit, one bit position to the left. The previous value in the carry flag is not lost. Instead, the value in bit 7 moves into the carry flag and the old value in the carry flag moves (rotates around the byte) into bit position 0.

Rotate One Bit Right



This instruction moves the values in each bit, one bit position to the right. The previous value in the carry flag is not lost. Instead, the value in bit 0 moves into the carry flag and the old value in the carry flag moves (rotates around the byte) into bit position 7.

Uses for Shift and Rotate Instructions

There are two fairly common uses for shift and rotate instructions. The uses are for multiplying or dividing the value of the byte by two.

Earlier in this appendix we saw how the value of any bit position was double the value of the bit position on its immediate right and conversely, half the value of the bit position on its immediate left. It follows then, if we move all the values in the bit positions one place to the left, we double the value of the byte. And if we move all the values to the right, we halve the total value of the byte.

This sort of instruction can be quite useful but there are some limitations to its use. The maximum value any byte can hold is 255 so when the value created by shifts to the left reaches a number greater than 255, the carry flag will be called to bear. 256, for example, will set the carry flag to 1 and reset the byte itself to zero.

There is a different sort of limitation relating to shifts to the right. Even numbers halve exactly, so don't create any problem. Odd numbers, however, are halved but are than rounded down to the next whole number. The carry flag reflects this but the value in the byte itself will be less than exactly half.

An interesting thing happens with rotate instructions after the ninth rotation: the byte contains the same value it began with.

Appendix B
Registers & Memory

Much of machine language programming involves loading registers and moving data around in the memory of the computer.

MEMORY

We know that our computer has two sorts of memory:

- RAM — the part where our BASIC programs are stored and which we can add to or change at will and
- ROM — the part where the BASIC interpreter and the C-64's control programs are permanently located.

In BASIC programming we don't have much to do (in a direct way) with either RAM or ROM. The computer does the work of arranging and organising memory for us.

In machine language programming, however, we have to tell the 6510 exactly how to handle memory — where to store data and where to get it from when we want to do something with it. The following is an abridged map of the C-64's memory.

Addresses

0 to 1019	RAM reserved by the C-64 to hold a variety of systems variables and buffers used in the general control and operation of the computer. But note the function of the following pieces in this block of memory.
178 to 179	Holds the pointer or address of the cassette file buffer. Can be used as indirect jump for a routine held in the cassette buffer.
251 to 254	Free zero page space. Used for a number of programs in this book.
828 to 1019	This is the tape I/0 buffer where you can store your programs when you're not using the cassette recorder.
1024 to 2023	RAM Screen Memory Area.
2040 to 2047	RAM Sprite Data Pointers.
2048 to 40959	RAM memory where your BASIC programs are stored. Machine language programs are frequently stored here but it is necessary to lower ramtop to protect them.
49152 to 53247	RAM-4096 bytes of free RAM available for storing machine language programs or other data.
55296 to 56295	RAM Color Code Memory Area. You should make yourself very familiar with this area and the Screen Memory Area above. With machine language you have to allocate the correct addresses you want the 6510 to use.
40960 to 49151	ROM 8K BASIC.
57344 to 65535	ROM 8K KERNAL.

In addition to RAM and ROM memory, in machine language programming we are vary much involved with some additional memory located inside the 6510 chip. We have met these progressively in the programs so we can now put them all together.

THE REGISTERS

A – Accumulator	1 byte
X – Index register	1 byte
Y – Index register	1 byte
Stack pointer	1 byte
Status register	1 byte
Program counter	2 bytes

Note that most of these registers are one byte 8 bits wide, and so can hold numbers in the range 0 to 255. The *program counter,* however, is 2 bytes wide and can hold numbers up to 65535.

A Register

This register is generally called the *accumulator.* It is the main general purpose register of the 6510 in which you will do most of your work. There are more instructions available for this register than the others. Additions, subtractions and logical operations can only be done with numbers held in the accumulator and the results are returned to the accumulator.

X and Y Index Registers

These registers are essentially the same. As you have seen from the programs, they are extensively used to index addresses — hence their name. They can, however, be used for many other purposes of a more general nature and there are a variety of very useful instructions available to use with them. *Increment, decrement* and *comparison* instructions are available for both registers but they cannot be used for other arithmetic or logical operations.

These are the three registers that you will be using most frequently in your machine language programming and you should make yourself familiar with the range of instructions available for each.

The Stack Pointer

This is a special register in which the 6510 keeps a record of the address of the last item on the stack. For our purposes the *stack* itself is the important part. The C-64 reserves in RAM a block of 256 bytes located from address 256 to 511. On this stack the 6510 temporarily stores information it wants to save during its operations and then recover at a later time.

The stack is also available to you as a temporary storage place for data you want to use again later in a program. As you have seen in the programs you quite often need to use a register several times during a program but need to keep certain data apart so that it can be put back into a register later on.

Unfortunately, the facility is only open to the contents of the accumulator. You can only push the contents of the accumulator onto the stack and pull this value off again. Fortunately, however, with the use of the Transfer instructions it is quite simple to push the contents of the X and Y registers onto the stack via the accumulator!

It is important to understand that the stack is a last-in-first-out device. It puts data in at the top of the block and grows downwards. The stack pointer simply checks the last item that has been put on the stack — it does not know where any previous items put on the stack are located. Therefore you must arrange to pull items off the stack in the correct order, that is, in reverse to the order they were put on the stack.

To illustrate

Push the accumulator onto stack
Push X register onto stack (via accumulator)
Push Y register onto stack (via accumulator)

. . . . . .

Pull Y register off stack (via accumulator)
Pull X register off stack (via accumulator)
Pull accumulator off stack

NOTE — When the accumulator is pushed onto the stack the value remains in the accumulator until another value is loaded into it.

STATUS REGISTER

This register looks like a normal 1 byte/8 bit register but is used by the 6510 in a special way as a group of 8 individual bits. These bits are used as *flags* to

show the status or result of various operations taking place in the computer.

Of the 8 bits in this byte, 7 are used to provide the status flags and the other bit is unused. As separate bits, they can therefore have only two states — ie, either reset to 0 or set to 1. Each flag can therefore show that a particular condition either exists or that it does not exist.

It is worth noting that resetting or setting a flag may be regarded as a positive action. In other words, if a particular instruction or operation is said to set the flag, then regardless of the previous state of the flag (it may already be set), you can view it as a positive action of the 6510 to set the flag as a result of the condition which now exists.

Of the 7 flags available, the only flags used in this book are the *zero flag* and the *carry flag*. The other flags are mainly concerned with some different forms of arithmetic not covered here. These other flags are not so commonly used. The zero flag and the carry flag are both frequently used flags and are very important in programming with conditional branch instructions.

The Carry Flag

This is bit 0 of the status register and is affected by any arithmetic operation.

We know that one byte of 8 bits can hold a maximum value of 255. Therefore when two numbers are added together to produce a value greater than 255, there will obviously be a carry over. The carry flag, as the name implies, holds the carry over. It can be thought of as the 9th bit of the byte.

This is what happens. If a result exceeds 255 then the carry flag is set to 1. That '1' is therefore equal to 256. If the result does not exceed 255 then the carry flag is reset to 0.

In the case of subtraction, when the result would be negative, the carry flag provides the extra bit from which a 'borrow' can be made.

When the result requires a borrow, the carry flag is reset to 0. When no borrow is required, the carry flag is set to 1.

So the rules are:

- If a carry results from addition, 1 goes into the carry flag.
- If a borrow results from subtraction 1 comes out of carry flag.

It is worth noting that a number of other microprocessors set the carry flag to 1 if a borrow is required. If you have had experience with or read about another type of microprocessor, check for this possible difference. It can be the source of a program bug. (The Bubble Sort program at the end of this appendix gives some good examples and information on the operation of the carry flag.)

The Zero Flag

This flag is bit 1 of the status register. Its rules for use are:

- Any arithmetic or logical operation which produces a 0 result will set the zero flag to 1.
- Any arithmetic or logical operation which produces a non-zero result will reset the zero flag to 0.

Caution. Do not confuse a 0 result with a 0 in the zero flag bit. It is said that when the result is 0 then the zero flag flies! That is, that a 1 comes up in the zero flag bit.

The zero flag is the one used to test for the condition of the two conditional branch instructions:

	Mnemonic
Branch if the result is zero	BEQ
Branch if the result is not zero	BNE

The 6510 works on the basis of the status flag — not on the result of an instruction like *decrement X* or *compare X with.*

In this case it's probably easier to work quite literally with the branch instructions and not worry too much about the zero flag. By all means get to know what the zero flag does so you can follow the programs in other books, but in your own programming just work with the machine language instructions. If you want the program to branch when X has been decremented to 0, then use BEQ. If you want a program to keep branching while the result of a compare or decrement instruction is not 0, use BNE.

THE BUBBLE SORT

Program Number 34 is quite different from the graphics programs elsewhere in this book. It has two special features:

- It provides some important information on the operation of the carry flag;
- It uses its own flag to control the program. This can be a most useful facility to be able to use in your own work.

Program # 34

```
10 N=828
20 READD:IFD=-1THEN50
30 POKEN,D:N=N+1:GOTO20
40 DATA 160,0,162,6,202,189,131,3,221,
    132,3,176,13,160,1,72,189,132,3
42 DATA 157,131,3,104,157,132,3,202,208,
    232,192,1,240,223,96,-1
50 POKE 900,25
60 POKE 901,56
70 POKE 902,12
80 POKE 903,40
90 POKE 904,72
100 POKE 905,100
110 SYS 828
120 FORN=900 TO 905
130 PRINT N,PEEK(N):NEXT
```

The Machine Language Program

	Instruction	Decimal Code	Mnemonic
(a)	Load Y register with 0	160,0	LDY 0
(b)	Load X register with 6	162,6	LDX 6
(c)	Decrement X register by 1	202	DEX
(d)	Load accumulator with contents of address 899 + X	189,131,3	LDA 899,X
(e)	Compare contents of accumulator with contents of address 900 + X	221,132,3	CMP 900,X

(f)	Branch on condition that carry flag is set to 1	176,13	BCS
(g)	Load Y register with 1	160,1	LDY 1
(h)	Push accumulator onto stack	72	PHA
(i)	Load accumulator with contents of address 900 + X	189,132,3	LDA 900,X
(j)	Store contents of accumulator in address 899 + X	157,131,3	STA 899,X
(k)	Pull accumulator off stack	104	PLA
(l)	Store contents of accumulator in address 900 + X	157,132,3	STA 900,X
(m)	Decrement X register by 1	202	DEX
(n)	Branch on condition that result is *not* 0	208,232	BNE
(o)	Compare contents of Y register with 1	192,1	CPY 1
(p)	Branch on condition that result *is* 0	240,223	BEQ
(q)	Return	96	RTS

When you RUN the program you should get the following display:

900	100
901	72
902	56
903	40
904	25
905	12

Notice that the program has sorted the numbers in descending order and then loaded them back into the same block of memory addresses into which they had previously been randomly POKEd.

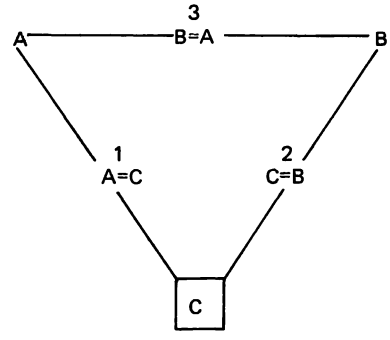
The program works by starting at one end of the list and comparing the first two values. If they are out of order then their positions are reversed. The next pair are then compared and the reversal made if needed. Pair by pair this is done through the list. Then the whole procedure is repeated. This continues until no reversal is required in one complete pass. The list has then been sorted. There are these points to note:

1. A flag will be needed to record when a reversal has been made in a pass through the list.

2. The number of *pairs* in a list is one less than the number of *items* in a list. The number of loops must be the same as the number of pairs. If the program loops by the number of items, one comparison will be with an item outside the list, whatever that is.

3. Two loops are needed: one to compare the pairs through the list and one to return the program to the start if there has been a reversal. The number of pairs and the exchange flags can be used to test these conditional branches.

4. The program needs a routine to reverse the items that are out of order. It's an exercise that has to be done in both BASIC and machine language when an exchange of assignments is required. The procedure requires that you set up a temporary address or temporary variable to hold one of the numbers so that its place can be vacated for the other one to take over. The stack will do nicely as a temporary address. Diagramatically it looks like this:



Temporary
Address

Explanation of the Machine Language Program

Line (a) loads the Y register with 0. This is the flag to be used for exchanges of the position of items. If Y = 0 then no exchange has been made. If Y = 1 then an exchange has been made.

Line (b) loads the X register with the number of items in the list.

Line (c) decrements the X register by 1. This is now the number of pairs and can be used as counter for the compare loop.

Line (d) loads the accumulator with the contents of address 899 + X. On the first cycle, X = 5 so the actual address is 904 (899 + 5) which at this stage contains 72.

Line (e) compares the contents of the accumulator with the contents of address 900 + X, (now 905) which contains 100. The compare instruction acts like a subtraction, taking 100 from 72 in the accumulator. That will result in a borrow so the carry flag will be cleared, ie, reset to 0.

Line (f) will not allow the branch because the condition of the carry flag has not been met and consequently the program passes on to the next instruction. The items are clearly out of order and must be reversed.

Line (g) loads the Y register with 1. The flag flies to show an exchange is to be made.

Line (l) commences the exchange:

- 72 in accumulator goes into temporary storage on the stack;
- 100 in address 905 is loaded into the accumulator replacing the 72;
- 100 in the accumulator is now stored in address 904 replacing the 72;
- 72 is loaded into the accumulator from the stack and finally
- 72 from accumulator is stored in address 905. The exchange is complete in line (l).

Line (m) decrements the contents of the X register by 1 and if X is not 0 (line (n)) the program branches back to the start of line (c) and starts another comparison.

Note that if the comparison in line (e) had shown that no exchange was necessary the carry flag would have been set to 1 and the program would have branched to the start of line (m), jumping over the part of the program that handles the exchange. Also observe that the exchange flag, Y would still be 0.

Now let's suppose that after the decrement in line (m) the X register is 0 (ie, all 5 passes have been made):

Line (p) compares the Y flag with 1. If no exchanges have been made in the 5 passes then Y will be 0 and the sort will have been completed. But if Y = 1 it means an exchange has been made during those 5 passes and the program will branch right back to the start and compare the 5 pairs again.

CARRY FLAG OPERATION

The same program can be used to illustrate the operation of the carry flag.

Change the 176 in line (f) of the machine language program to 144. 144 is the decimal code for the instruction *branch on condition that the carry flag is cleared* — ie, is reset to 0. This has an effect opposite to the previous branch instruction (code 176). When you RUN the program now the list of addresses and items will show on the screen in *ascending* order — ie, 100 is now at the bottom.

But what happens when the list contains two numbers which are the same? Let's see. Change the 12 in line 70 of the BASIC program to 40 so that there are two 40's in the list. When you RUN the program the screen will stop with RUN on the bottom. READY and the flashing cursor will not appear. The program is in an endless loop and cannot go on to complete the Return instruction.

The way out of it is to press RUN/STOP and RETURN. READY and the cursor will appear at the top of the screen. Type in as a direct command GOTO 120 RETURN and the screen will display the list of addresses with the items properly sorted.

Now go back and change the 144 in line (f) of the machine language program to 176 — ie, the original branch instruction — and leave the two 40's in the program. When you RUN the program now there will be no hitch. You should get the normal display of addresses with the items all nicely sorted, READY, flashing cursor and all!

From these examples we can distil some important principles concerning the operation of the carry flag in the 6510.

Let's assume we have a value in the accumulator (call it A) and a value in a memory address (call it M) and that the carry flag is to be used as the condition for a branch. When we compare M with A (in other words, subtract M from A) then:

(a) If M is greater than A, the carry flag will be cleared — ie, reset to 0. The branch if carry flag *cleared* instruction will work and the program will branch. The carry flag is not cleared to 0 if M is equal to A.

(b) If A is equal to or greater than M, the carry flag will be set to 1. The branch if carry flag *set* instruction will work and the program will branch.

In this program, when the branch instruction is changed to code 144 (branch if carry flag cleared) it actually branches when M is greater than A. If M equals A, the carry flag is not cleared and on each pass through the list the two identical items are in fact exchanged. The exchange flag, Y, therefore remains at 1 and the program never goes on to the Return instruction.

An Interesting Exercise

If you'd like to see just how fast machine language is, try this:

1. From the Bubble Sort program, delete all BASIC lines from 50 on and replace with the following:

```
50FORN=0TO99:POKE900+N,INT(RND(0)*100):NEXT
55FORN=0TO99:PRINTPEEK(900+N):NEXT
60SYS828
70FORN=0TO99:PRINTPEEK(900+N):NEXT
```

2. Change 6 in DATA line 40 to 100, ie, LDX 100.

3. RUN. 100 random numbers will scroll up the screen. There will be a moment's hesitation and then 100 sorted numbers will scroll up the screen.

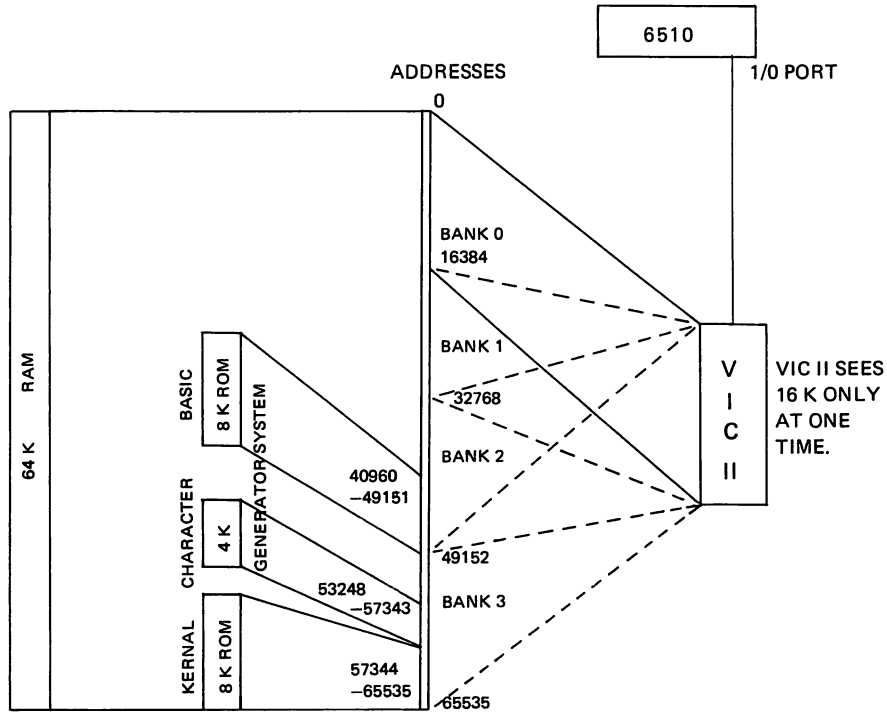
Consider what has happened during that moment's hesitation between lines 60 and 70 — between the time when line 60 calls SYS 828 and line 70 commences to print out the sorted numbers!

Appendix C:

A Little Memory Management

This is not the place for a detailed discussion of the hardware of which the Commodore 64 is built but there are several features concerning its management of memory that have bearing on machine language programming and these are worth looking at.

In common with most 8 bit microprocessors, the C-64 handles a total of 64K of memory addresses. In the diagram that follows, you'll see these listed just to the right of the big box. They run from 0 at the top to 6535 at the bottom.



Looking at that big box itself, however, we see that the C-64 in fact has available 84K of memory addresses. The 84K consists of 64K of RAM and 20K of ROM. The 20K of ROM consists of:

- 8K of kernal routines
- 4K of character data and
- 8K of BASIC interpreter.

These segments of ROM (Read Only Memory) have memory addresses assigned to them and if you look closely, you'll see that these addresses contain both ROM and RAM — ie, the ROM and RAM overlap each other.

Now notice on the right-hand side of the diagram, the *Video Interface Chip* (known, for short, as the VIC II). This chip controls all of the graphics and video interfacing of the computer. In it, there are 47 registers (like ordinary addresses), starting at address 53248. It is through these registers that you control the VIC II and tell it what to do with sprites and other graphics. Outside of these registers, the VIC II has to get information from ROM and RAM for character information and general working. It can certainly gather data from these other parts of memory but there is a limitation. At any one time, the VIC II can only cope with a *bank* of 16K. Now in the diagram, see how the total 64K of memory is divided into 4 banks (numbered 0 to 3) each of which contains 16K of addresses.

When first you switch on your C-64, the VIC II is set to look at the first 16K of RAM — ie, from address 0 to address 16383 or bank 0. This is called the *default* condition. In other words, it's where the machine operates in the absence of any other instructions. It explains why (in the normal course of events) you have to keep sprite data and other programmable character data shown in this area of 0 to 16383. If it's somewhere else, the VIC II won't be able to see it and won't be able to use it to put displays on the screen.

There is, however, an apparent anomally that needs explaining. The information for the normal character set is situated in the character ROM located at addresses 53248 to 57343. We must ask how it is that this data is available to the VIC II under the conditions of bank 0. The answer is that the VIC II never actually gets this information from these addresses. The machine system arranges for a ROM *image* of the character information to appear starting at address 4096. This, of course, raises another question because 4096 is squarely in the area of RAM reserved for our BASIC programs. The answer to this question lies in the function of the I/O port of the 6510. It regulates the availability of 4096 for ROM images and BASIC programs, ensuring that the area is available when your BASIC program is required.

Concerning this function of the I/O port, we saw in Chapter 7 how we had to switch it off in order to PEEK the characters in the character ROM at 53248.

In any event, the concept of the VIC II working in banks of 16K is an essential one to grasp when we're concerned with graphics. Far from limiting the memory available to us, it opens up many possibilities for the use of the C'64's total memory. Consider, for example, the result of telling the C-64 to operate with bank 2. In the first place, the ROM image (which the VIC II treats as the normal character set) will appear from 36864 to 40959. More importantly, though, as far as your programming in machine code is concerned, you will have the opportunity to store your special character data and your sprite data at addresses much higher in memory.

In Chapter 9 we said that the formula for locating sprite data was 64 times the value in the sprite pointer address. This true, of course, under the conditions imposed by bank 0 — ie, that the result of the multiplication be inside the range 0 to 16383. So bank 0 imposed a limit of 255 on the number we could put into a sprite pointer address.

Strictly speaking, however, the formula for locating sprite data is 64 times the value in the sprite pointer *plus* 16384 times the bank number. In other words, each time you move to a higher bank, you add 16K to the addresses held in the previous bank.

What all this means, of course, is the availability of many large chunks of memory — should you need them and be prepared to switch from bank to bank to make use of them. At this point, though, it has to be said that it will probably be some time before you will need to use these memory management facilities. You're unlikely to be writing programs with so much data that you'll need anything beyond the first 16K of memory! If that's making a hasty judgement about your skill or ambition, however, the information you'll need to use the various banks is to be found in the *Commodore 64 Programmer's Reference Guide.* It's largely a matter of using the PEEKs and POKEs with which we've become familiar.

The Location of Character Memory

Your Commodore 64 has two sets of regular characters. The two sets are illustrated in Appendix F. Each of the sets (with its corresponding set of reverse characters) contains 256 characters and each of the characters consists of 8 bytes of data. This means, of course, that each set of 256 characters occupies 2K of memory.

We've already seen how the VIC II can only access 16K of memory at any one time but there is another factor concerning this access of which we must also be aware. In its normal operation the VIC II can also only access one complete character set (of 2K) in the character ROM at any one time. Consequently, the machine system divides the 16K character ROM into 8 blocks of 2K each.

This means that when you transfer a character set (or part of a character set) into RAM in order to change the standard characters to your own characters, you must arrange it so that your character set starts at the beginning of one of these 8 blocks. When first you switch on your C-64, of course, the VIC II will look for the standard character set (as the ROM image) at address 4096. You can, however, tell the VIC II to get character data from any one of the 8 blocks. You do this with the following BASIC statement.

```
POKE 53272, (PEEK(53272)AND240)OR A
```

It works like this. In the first place, we need to understand that the location of character memory is determined by the value held in bits 1, 2 and 3 of register 53272. These bits, when set to 1, have a decimal value of 2, 4 and 8 respectively — a possible total of 14. In the BASIC statement, the total value of the three bits is represented by A and, of course, it can vary between 0 and 14. The other five bits in register 53272 (bits 4 to 7) control the location of screen memory so they must not be touched while you are playing with the location of character memory. The AND240 in the BASIC statement provides this protection. It masks off these bits and allows them to drop through unchanged.

The following table sets out the addresses in character memory for each of the possible values of A. You cannot use the odd values for A because the character memory location must start at the beginning of each 2K block of RAM. Set 1 of the ROM image, for example, starts at 4096 and set 2 at 6144.

VALUE OF A	LOCATION OF CHARACTER MEMORY
0	0
2	2048
4	4096
6	6144
8	8192
10	10240
12	12288
14	14336

In the programs in this book (when we've been using programmable characters and sprites) we've put the data in RAM from address 12288 up. You can try putting your character data at these other locations when you need the space but avoid starting at 0. You will find yourself right amongst the systems variables and there may be dire consequences!

Appendix D

6510 Machine Language Instructions

We've said a number of times now that it is our responsibility to tell the 6510 where to find data and where to store it. The 6510 instructions help us to do just that.

In each of the decimal codes representing the machine language instruction, the first number (or byte) tells the 6510 two things:

1. The action to be taken or what is has to do (eg. Load the accumulator)
2. What the following byte or bytes mean in terms of an address.

For example:

Instruction	Decimal Code
Load the accumulator with (a number)	169, (a number)
Load the accumulator with contents of address xx xx	173,xx,xx
Load the accumulator with contents of address xx xx indexed by the contents of X register	189,xx,xx
Load the accumulator with contents of zero page address xx	165,xx

In other words, for each machine language instruction there may be one or several codes representing different *addressing modes.*

ADDRESSING MODES

Implied

This is also known as *inherent addressing*. Several instructions only do things inside the 6510 itself — for example set carry flag and transfer contents of X register to accumulator. In these cases addressing is *implied* in the single byte instruction. There is no problem in selecting the right addressing mode with these instructions because there is only one decimal code for each of them.

Immediate

This is a two byte instruction where the first byte is the actual instruction telling the 6510 to do something with the value contained in the second byte. In other words, the second byte contains the data or value you want the 6510 to do something with. For example:

Instruction	Decimal Code
Load the accumulator with 100	169,100

The first byte 169, tells the 6510 to Load the accumulator with the value in the second byte — in this case 100.

In a program the value 100 would actually be stored in the address immediately following the address which holds the instruction 169.

Absolute or Direct (Non-Zero Page)

(Zero page addressing will be discussed shortly. In the meantime, *non-zero* page covers all addresses from 256 upwards.)

This is a three byte instruction. The first byte is the instruction to the 6510 to do something with the *contents* of the memory address represented by the second and third bytes.

NB — the address must be stored in the usual 6510 manner: Least Significant part in the first address byte and Most Signifiant part in the second address byte.

Absolute Indexed with X (or Y) Register (Non-Zero Page)

Indexed addressing gives fast economical access to a block of up to 256 memory addresses. It is a three byte instruction. As usual, the first byte is the action instruction.

The second and third bytes (in the usual Low byte/High byte manner) hold an address called the *base address*. When the 6510 sees this instruction it adds to the base address the *contents* of the X register (or the Y register depending on which instruction code you have chosen to use) and the result is the *actual address* on whose contents it will work.

When X = 1, the *actual* address is the *base* address plus 1 — and so on to the limit where X = 255 and the actual address is the base address plus 255.

Relative

This is a two byte instruction and is only used with conditional branch insructions. It is not directly concerned with memory addresses as such but rather with the *relative byte position* of the conditional branch instruction in the program — hence the name.

The conditional branch instruction is similar to the IF. . .THEN. . . GOTO instruction in BASIC. BASIC, however, has line numbers and the program may be told to GOTO a line number whether it occurs earlier or later in the program listing. In machine language programs there are no line numbers so the 6510 has to be told the number of bytes to branch or jump over — backwards or forwards — in the program.

The 6510 keeps track of its position in the operation of the program with the *program counter* register. As soon as the 6510 reaches a program instruction, it immediately adds to the program counter the number of bytes in that instruction.

If the conditional branch instruction tells the 6510 to go back to an earlier instruction in the program it has to be told to subtract a certain number of bytes from the program counter — ie, a negative number. Machine language, however, doesn't have a minus sign so you cannot say for example, branch −14 bytes. This is where the 2's complement representation of negative numbers (discussed in Appendix A) is used. In this example, therefore, the instruction is branch 242 (ie, the 2's complement of −14).

As we saw in Appendix A, the system of 2's complements only covers negative numbers up to −128 and positive numbers up to 127. This limits the size of branches but the limitation is not one with which we need to be overly concerned: 127 represents a sizeable branch!

How many bytes to branch?

To branch backwards you count the number of bytes from the end of the branch instruction (including the two bytes in the instruction itself) to the *start* of the instruction to which the branch is being made.

To branch forwards you count the number of bytes in all the instructions from the end of the branch instruction (excluding any bytes in the branch instruction) to the *start* of the instruction to which the branch is to make.

It is *critical* that you count the correct number of bytes for the branch and use the correct 2's complement when necessary. The 6510 is entirely dependent on you for this and will do or try to do, exactly what you have told it. If you cause it to branch to the wrong instruction or to the middle of an instruction, it will go there and start to execute what it believes is the start of the correct instruction with some peculiar results; usually a crash.

Indirect

The *jump to another location* instruction is the only one in the 6510 set which has a true *indirect* addressing mode.

This is a three byte instruction where the second and third bytes contain an address. On receiving this instruction the 6510 will read from this address the Least Significant Byte of another address and check the address immediately following the address in the instruction to get the Most Significant Byte of the new address. The parts of the actual address have to be put into the instruction address and the following address, either directly by you or during execution of the program.

Indirect addressing is often used in systems where several users want access to blocks of data which are changing and moving in memory. Another address, fixed and known to all, holds the address of the data. Therefore one only needs to look at the first address to see where the material is at any time.

This instruction is not often used, as *jump* (with the absolute addressing mode) meets most of our needs.

ZERO PAGE ADDRESSING

Zero page addressing is a special feature of the 6510. As a general rule all addresses must be specified in two bytes with the Least Significant Byte first and the Most Significant Byte in the next higher address.

As discussed in Appendix A, the Most Significant Byte goes up in steps of 256. For the address 0 the Least Significant Byte is 0 and the Most Significant Byte is also 0. For all addresses from 0 to 255, the Most Significant Byte remains 0. At address 256, the Least Significant Byte becomes 0 and the Most Significant Byte increases to 1.

It is fairly common practice to refer to the Most Significant Byte as a *page* of memory. The Least Significant Byte is then said to contain the items on the page. Zero page therefore refers to those memory addresses from 0 to 255 inclusive, where the Most Significant Byte is zero.

The 6510 has been designed to allow a special ease of use with zero page addressing: only the Least Significant Byte has to be specified. The 6510 then assumes that the Most Significant Byte is 0. Zero page addressing is therefore very fast and economical of memory.

However, in common with other computers using the 6502 or 6510, the C-64 designers have used this first part of RAM memory to hold a variety of systems variables essential to the total operation of the machine. There is very little room available in zero page for the user.

ZERO PAGE ADDRESSING MODES

There are four zero page addressing modes:

- Absolute/direct
- Absolute indexed with X (or Y) register
- Pre-indexed indirect
- Post-indexed indirect

The first two modes are exactly the same as *absolute/direct* and *absolute indexed with X (or Y) register* as explained for the non-zero page addressing modes. Each has its own instruction code but the zero page address requires only one byte.

Example:

Suppose you use the instruction 'Load accumulator from memory' using *absolute/direct* addressing mode.

	Decimal Code
If address is on zero page (say, address 1)	165,1
If address is non-zero page (say, address 7680)	173,0,30

Notice that the zero page instruction has only one address byte (the Least Significant Byte) but that the non-zero page address is specified in two bytes, even if the Least Significant Byte is 0 as it is in this example.

The other zero page addressing modes are quite different and rather more complex.

Pre-Indexed Indirect (with X register)

This is a two byte instruction where the second byte contains a number (called an offset) which the 6510 will add to the contents of the X register. The result

of this addition then specifies a zero page address which contains the Least Significant Byte of the actual address. The Most Significant Byte of this actual address will be picked up by the 6510 from the next zero page address.

This is a wrap-around addition in that any carry over from the addition of the offset and X register will be discarded. Consequently, the first address from which the actual address is obtained will always stay within the zero page.

As there is so little available memory left on zero page it's unlikely that this particular addressing mode will be of much use in any of your machine language programs. It is sometimes referred to as the *indexed indirect* mode, because you use the index (X register) to find the first address and then find the actual address indirectly from that.

Post-Indexed Indirect (with Y register)

This is also a two byte instruction where the second byte specifies the Least Significant Byte on zero page in which is stored the Least Significant Byte of a *base* address. The 6510 will pick up the Most Significant Byte of this *base* address from the next higher address on zero page.

To this *base* address the 6510 will add the contents of the Y register to arrive at the *actual* address on which it will work.

This mode is virtually a combination of the indirect addressing mode available with the jump instruction and the absolute indexed by Y register. Actually if the Y register is loaded with 0 then this mode is really an indirect addressing mode.

Of the four zero page addressing modes, this is the one which offers the most in terms of machine language programming on the C-64. It is also called *indirect indexed* because an address is first found indirectly from the zero page address and then it is indexed with the Y register.

Example:

Suppose zero page address 251 contains the value 255 and that zero page address 252 contains the value 29. The Y register contains the value 1. The instruction to be executed might be:

Instruction	**Decimal Code**
Store contents of accumulator in address (252),Y	145,251

Firstly the 6510 will go to zero page address 251 for the Least Significant Byte of the base address, or 255. Then it will find the Most Significant Byte, or 29, from the next address 252. These low and high bytes represent the address 7679 to which the contents of the Y register, or 1, will be added to produce an actual address of 7680.

INSTRUCTIONS	AVAILABLE ADDRESSING MODE											MNEMONIC
	Immediate	Absolute/Direct	Absolute Indexed by X	Absolute Indexed by Y	Zero Page Direct	Zero Page Indexed by X	Zero Page Indexed by Y	Pre-Indexed Indirect	Post-Indexed Indirect	Relative	Implied	
Add with Carry	x	x	x	x	x	x		x	x			ADC
AND — Logical Operator	x	x	x	x	x	x		x	x			AND
Accumulator or Memory, Shift Left		x	x		x	x					x	ASL
Branch if Carry Cleared										x		BCC
Branch if Carry Set										x	x	BCS
Branch if Equal to Zero										x		BEQ
Branch if Not Equal to Zero										x		BNE
Clear Carry Flag											x	CCF
Compare Accumulator with Memory	x	x	x	x	x	x		x	x			CMP
Compare X Register with Memory	x	x			x							CPX
Compare Y Register with Memory	x	x			x							CPY
Decrement Memory by 1		x	x		x	x						DEX
Decrement X Register by 1											x	DEX
Decrement Y Register by 1											x	DEY
Exclusive OR — Logical Operator	x	x	x	x	x	x		x	x			EOR
Increment Memory by 1		x	x		x	x						INC
Increment X Register by 1											x	INX
Increment Y Register by 1											x	INY
Jump to New Location		x										JMP
Jump to Subroutine		x										JSR
Load Accumulator	x	x	x	x	x	x		x	x			LDA
Load X Register	x	x		x	x		x					LDX
Load Y Register	x	x	x		x	x						LDY
Logically Shift Right		x	x		x	x					x	LSR
No Operation											x	NOP
OR — Logical Operator	x	x	x	x	x	x		x	x			ORA
Push Accumulator on Stack											x	PHA
Pull Accumulator off Stack											x	PLA
Rotate Left		x	x		x	x					x	ROL
Rotate Right		x	x		x	x					x	ROR
Subtract with Carry	x	x	x	x	x	x		x	x			SBC
Set Carry Flag											x	SEC
Store Accumulator		x	x	x	x	x		x	x			STA
Store X Register												STX
Store Y Register		x			x		x					STY
Transfer Accumulator to X Register											x	TAX
Transfer Accumulator to Y Register											x	TAY
Transfer X Register to Accumulator											x	TXA
Transfer Y Register to Accumulator											x	TYA

6510 MACHINE LANGUAGE INSTRUCTIONS

NOTE:

dd = data in one byte
aa = L.S.B. of Zero Page address
aa, aa = address in two bytes, L.S.B. first

ADD WITH CARRY, CONTENTS OF MEMORY TO ACCUMULATOR ADC

Addressing Mode	Instruction
Immediate	105, dd
Absolute/Direct	109, aa, aa
Absolute Indexed by X register	125, aa, aa
Absolute Indexed by Y register	121, aa, aa
Zero Page Direct	101, aa
Zero Page Indexed by X register	117, aa
Pre-Indexed Indirect	97, dd
Post-Indexed Indirect	113, aa

AND LOGICAL OPERATION OF MEMORY WITH ACCUMULATOR AND

Addressing Mode	Instruction
Immediate	41, dd
Absolute/Direct	45, aa, aa
Absolute Indexed by X register	61, aa, aa
Absolute Indexed by Y register	57, aa, aa
Zero Page Direct	37, aa
Zero Page Indexed by X register	53, aa
Pre-Indexed Indirect	33, dd
Post-Indexed Indirect	49, aa

(ACCUMULATOR OR MEMORY) SHIFT ONE BIT LEFT ASL

Addressing Mode	Instruction
Accumulator (see note)	10
Absolute/Direct	14, aa, aa
Absolute Indexed by X register	30, aa, aa
Zero Page Direct	6, aa
Zero Page Indexed by X register	22, aa

NOTE – Accumulator addressing only applies to shifts and rotate instructions – 1 byte similar to implied addressing mode.

BRANCH IF CARRY CLEAR — BCC

Addressing Mode	Instruction
Relative	144, dd

BRANCH IF CARRY SET — BCS

Addressing Mode	Instruction
Relative	176, dd

BRANCH IF EQUAL TO ZERO — BEQ

Addressing Mode	Instruction
Relative	240, dd

BRANCH IF NOT EQUAL TO ZERO — BNE

Addressing Mode	Instruction
Relative	208, dd

CLEAR CARRY FLAG — CLC

Addressing Mode	Instruction
Implied	24

COMPARE ACCUMULATOR WITH CONTENTS OF MEMORY — CMP

Addressing Mode	Instruction
Immediate	201, dd
Absolute/Direct	205, aa, aa
Absolute Indexed by X register	221, aa, aa
Absolute Indexed by Y register	217, aa, aa
Zero Page Direct	197, aa
Zero Page Indexed by X register	213, aa
Pre-Indexed Indirect	193, dd
Post-Indexed Indirect	209, aa

COMPARE INDEX REGISTER X WITH MEMORY — CPX

Addressing Mode	Instruction
Immediate	224, dd
Absolute/Direct	236, aa, aa
Zero Page Direct	228, aa

COMPARE INDEX REGISTER Y WITH MEMORY — CPY

Addressing Mode	Instruction
Immediate	192, dd
Absolute/Direct	204, aa, aa
Zero Page Direct	196, aa

DECREMENT MEMORY BY 1 — DEC

Addressing Mode	Instruction
Absolute/Direct	206, aa, aa
Absolute Indexed by X register	222, aa, aa
Zero Page Direct	198, aa
Zero Page Indexed by X register	214, aa

DECREMENT INDEX REGISTER X BY 1 — DEX

Addressing Mode	Instruction
Implied	202

DECREMENT INDEX REGISTER Y BY 1 — DEY

Addressing Mode	Instruction
Implied	136

EXCLUSIVE OR LOGICAL OPERATION
OF ACCUMULATOR WITH MEMORY EOR

Addressing Mode	Instruction
Immediate	73, dd
Absolute/Direct	77, aa, aa
Absolute Indexed by X register	93, aa, aa
Absolute Indexed by Y register	89, aa, aa
Zero Page Direct	69, aa
Zero Page Indexed by X register	85, aa
Pre-Indexed Indirect	65, dd
Post-Indexed Indirect	81, aa

INCREMENT CONTENTS OF MEMORY BY 1 INC

Addressing Mode	Instruction
Absolute/Direct	238, aa, aa
Absolute Indexed by X register	254, aa, aa
Zero Page Direct	230, aa
Zero Page Indexed by X register	246, aa

INCREMENT CONTENTS OF X REGISTER BY 1 INX

Addressing Mode	Instruction
Implied	232

INCREMENT CONTENTS OF Y REGISTER BY 1 INY

Addressing Mode	Instruction
Implied	200

JUMP JMP

Addressing Mode	Instruction
Absolute/Direct	76, aa, aa
Indirect	108, aa, aa

JUMP TO SUBROUTINE JSR

Addressing Mode	Instruction
Absolute/Direct	32, aa, aa

LOAD ACCUMULATOR FROM MEMORY LDA

Addressing Mode	Instruction
Immediate	169, dd
Absolute/Direct	173, aa, aa
Absolute Indexed by X register	189, aa, aa
Absolute Indexed by Y register	185, aa, aa
Zero Page Direct	165, aa
Zero Page Indexed by X register	181, aa
Pre-Indexed Indirect	161, dd
Post-Indexed Indirect	177, aa

LOAD INDEX REGISTER X FROM MEMORY LDX

Addressing Mode	Instruction
Immediate	162, dd
Absolute/Direct	174, aa, aa
Absolute Indexed by Y register	190, aa, aa
Zero Page Direct	166, aa
Zero Page Indexed by Y register	182, aa

LOAD INDEX REGISTER Y FROM MEMORY LDY

Addressing Mode	Instruction
Immediate	160, dd
Absolute/Direct	172, aa, aa
Absolute Indexed by X register	188, aa, aa
Zero Page Direct	164, aa
Zero Page Indexed by X register	180, aa

LOGICALLY SHIFT RIGHT OF ACCUMULATOR OR MEMORY LSR

Addressing Mode	Instruction
Accumulator	74
Absolute/Direct	78, aa, aa
Absolute Indexed by X register	94, aa, aa
Zero Page Direct	70, aa
Zero Page Indexed by X register	86, aa

NO OPERATION NOP

Addressing Mode	Instruction
Implied	234

OR LOGICAL OPERATION OF ACCUMULATOR WITH MEMORY ORA

Addressing Mode	Instruction
Immediate	9, d
Absolute/Direct	13, aa, aa
Absolute Indexed by X register	29, aa, aa
Absolute Indexed by Y register	25, aa, aa
Zero Page Direct	5, aa
Zero Page Indexed by X register	21, aa
Pre-Indexed Indirect	1, dd
Post-Indexed Indirect	17, aa

PUSH CONTENTS OF ACCUMULATOR ONTO STACK PHA

Addressing Mode	Instruction
Implied	72

PULL OFF STACK AND PLACE IN ACCUMULATOR PLA

Addressing Mode	Instruction
Implied	104

RETURN FROM SUBROUTINE RTS

Addressing Mode	Instruction
Implied	96

ROTATE ACCUMULATOR OR MEMORY, LEFT THROUGH CARRY ROL

Addressing Mode	Instruction
Accumulator	42
Absolute/Direct	46, aa, aa
Absolute Indexed by X register	62, aa, aa
Zero Page Direct	38, aa
Zero Page Indexed by X register	54, aa

ROTATE ACCUMULATOR OR MEMORY, RIGHT THROUGH MEMORY ROR

Addressing Mode	Instruction
Accumulator	106
Absolute/Direct	110, aa, aa
Absolute Indexed by X register	126, aa, aa
Zero Page Direct	102, aa
Zero Page Indexed by X register	118, aa

SET CARRY FLAG SEC

Addressing Mode	Instruction
Implied	56

STORE CONTENTS OF ACCUMULATOR IN MEMORY STA

Addressing Mode	Instruction
Absolute/Direct	141, aa, aa
Absolute Direct Indexed by X register	157, aa, aa
Absolute Direct Indexed by Y register	153, aa, aa
Zero Page Direct	133, aa
Zero-Page Indexed by X register	149, aa
Pre-Indexed Indirect	129, dd
Post-Indexed Indirect	145, aa

STORE CONTENTS OF X REGISTER IN MEMORY STX

Addressing Mode	Instruction
Absolute/Direct	142, aa, aa
Zero Page Direct	134, aa
Zero Page Indexed by Y register	150, aa

STORE CONTENTS OF Y REGISTER IN MEMORY STY

Addressing Mode	Instruction
Absolute/Direct	140, aa, aa
Zero Page Direct	132, aa
Zero Page Indexed by X register	148, aa

SUBTRACT MEMORY FROM ACCUMULATOR WITH BORROW SBC

Addressing Mode	Instruction
Immediate	233, dd
Absolute/Direct	237, aa, aa
Absolute Indexed by X register	253, aa, aa
Absolute Indexed by Y register	249, aa, aa
Zero Page Direct	229, aa
Zero-Page Indexed by X register	245, aa
Pre-Indexed Indirect	
Post-Indexed Indirect	241, aa

TRANSFER CONTENTS OF
ACCUMULATOR TO X REGISTER TAX

Addressing Mode	Instruction
Implied	170

TRANSFER CONTENTS OF
ACCUMULATOR TO Y REGISTER

Addressing Mode	Instruction
Implied	168

TRANSFER CONTENTS OF
X REGISTER TO ACCUMULATOR TXA

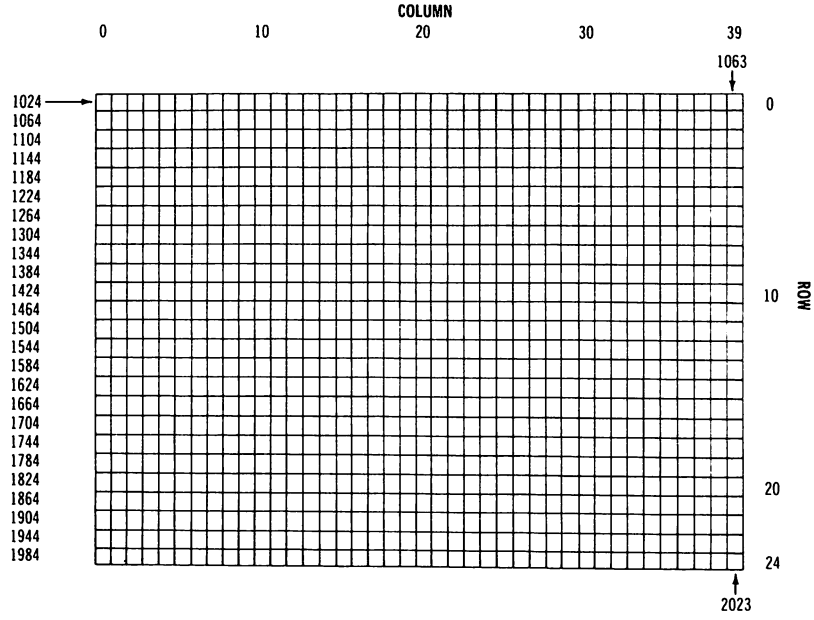
Addressing Mode	Instruction
Implied	138

TRANSFER CONTENTS OF
Y REGISTER TO ACCUMULATOR TYA

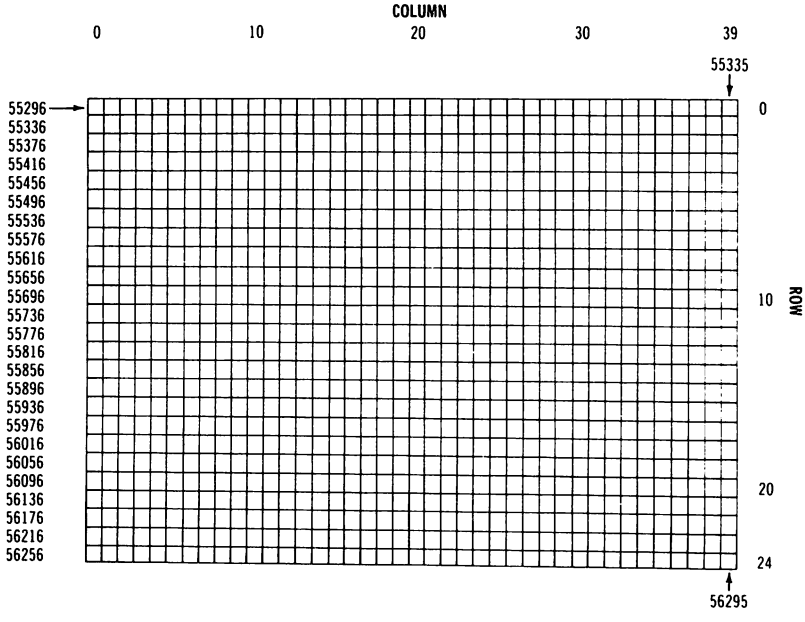
Addressing Mode	Instruction
Implied	152

APPENDIX E

SCREEN MEMORY MAP



COLOR MEMORY MAP



APPENDIX F
Screen Display Codes

To display any of the characters in Set 1, store the given code according to the addresses set out in the screen character code memory map in Appendix D.

For characters in Set 2, first store 23 in address 53272.

To return to characters in Set 1, store 21 in address 53272.

SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	R	r	18	$		36
A	a	1	S	s	19	%		37
B	b	2	T	t	20	&		38
C	c	3	U	u	21	'		39
D	d	4	V	v	22	(		40
E	e	5	W	w	23	)		41
F	f	6	X	x	24	*		42
G	g	7	Y	y	25	+		43
H	h	8	Z	z	26	,		44
I	i	9	[		27	–		45
J	j	10	£		28	.		46
K	k	11	]		29	/		47
L	l	12	↑		30	0		48
M	m	13	←		31	1		49
N	n	14	SPACE		32	2		50
O	o	15	!		33	3		51
P	p	16	"		34	4		52
Q	q	17	#		35	5		53

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
6		54		O	79			104
7		55		P	80			105
8		56		Q	81			106
9		57		R	82			107
:		58		S	83			108
;		59		T	84			109
<		60		U	85			110
=		61		V	86			111
>		62		W	87			112
?		63		X	88			113
		64		Y	89			114
	A	65		Z	90			115
	B	66			91			116
	C	67			92			117
	D	68			93			118
	E	69			94			119
	F	70			95			120
	G	71	SPACE		96			121
	H	72			97			122
	I	73			98			123
	J	74			99			124
	K	75			100			125
	L	76			101			126
	M	77			102			127
	N	78			103			

Codes from 128-255 are reversed images of codes 0-127.

APPENDIX G
Screen Color Codes

To assign color to any character, store the given code according to the addresses set out in the screen color codes memory map in Appendix E.

Color	Code
Black	0
White	1
Red	2
Cyan	3
Purple	4
Green	5
Blue	6
Yellow	7
Orange	8
Brown	9
Light Red	10
Grey 1	11
Grey 2	12
Light Green	13
Light Blue	14
Grey 3	15

Screen & Border Colors

To set screen color — store color code in 53281.

To set border color — store color code in 53280

APPENDIX H
Chr$ Codes

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0		22	,	44	B	66
	1		23	–	45	C	67
	2		24	.	46	D	68
	3		25	/	47	E	69
	4		26	0	48	F	70
WHT	5		27	1	49	G	71
	6	RED	28	2	50	H	72
	7	CRSR →	29	3	51	I	73
DISABLES SHIFT C=	8	GRN	30	4	52	J	74
ENABLES SHIFT C=	9	BLU	31	5	53	K	75
	10	SPACE	32	6	54	L	76
	11	!	33	7	55	M	77
	12	"	34	8	56	N	78
RETURN	13	#	35	9	57	O	79
SWITCH TO LOWER CASE	14	$	36	:	58	P	80
	15	%	37	;	59	Q	81
	16	&	38	<	60	R	82
CRSR ↓	17	'	39	=	61	S	83
RVS ON	18	(	40	>	62	T	84
CLR HOME	19	)	41	?	63	U	85
INST DEL	20	*	42	@	64	V	86
	21	+	43	A	65	W	87

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
X	88		114	f8	140		166
Y	89		115	SHIFT RETURN	141		167
Z	90		116	SWITCH TO UPPER CASE	142		168
[	91		117		143		169
£	92		118	BLK	144		170
]	93		119	CASE	145		171
↑	94		120	RVS OFF	146		172
←	95		121	CLR HOME	147		173
	96		122	INST DEL	148		174
	97		123		149		175
	98		124		150		176
	99		125		151		177
	100		126		152		178
	101		127		153		179
	102		128		154		180
	103		129		155		181
	104		130	PUR	156		182
	105		131	CASE	157		183
	106		132	YEL	158		184
	107	f1	133	CYN	159		185
	108	f3	134	SPACE	160		186
	109	f5	135		161		187
	110	f7	136		162		188
	111	f2	137		163		189
	112	f4	138		164		190
	113	f6	139		165		191

CODES	**192-223**	**SAME AS**	**96-127**
CODES	**224-254**	**SAME AS**	**160-190**
CODE	**255**	**SAME AS**	**126**

Index

– the ideal book for the Commodore 64 owner who has learnt BASIC and now wants to develop further.

Machine language enables you to program the chip itself, and thereby unleash the full capability of your Commodore 64 in a whole range of sensational colour and sound effects. Even the beginner can discover the ease and power of machine language by following this clear, friendly introduction. For those with more experience, this book gives you all sorts of subroutines that you can easily use in your own games.

Whatever your skill level, if you want to write faster, more exciting programs, then this is the book for you.

Cover illustration by Chris Payne

AUST. $14.95
(recommended)

Games,
Crafts and Hobbies
ISBN 0 14
00 7894 0