Commodore 64 Machine Code

Ian Stewart and Robin Jones

SHIVA's friendly micro series

Commodore 64 Machine Code

Ian Stewart
Mathematics Institute, University of Warwick

Robin Jones
Computer Unit, South Kent College of Technology

Shiva Publishing Limited

SHIVA PUBLISHING LIMITED
64 Welsh Row, Nantwich, Cheshire CW5 5ES, England

ISBN 1 85014 025 1

Typeset and printed by Devon Print Group, Exeter

Contents

Introduction

The Commodore 64 has become one of the most popular home computers in Europe and the USA. It is a versatile and interesting machine. The aim of this book is to show you how to enhance its abilities still further, by learning the rudiments of *Machine Code* programming. Want to fill the TV screen with a grid of symbols, in the twinkling of an eye? Move sprites around fast enough to play a reasonable game? Count how many times the REM character occurs in a program? Then it's Machine Code you'll need. It places many more demands on the programmer than BASIC does; but as a reward, it expands the range of tasks that your computer can do.

Most of the general principles in this book apply to any computer that uses a 6502 or 6510 microprocessor; but throughout we have borne the specific features of the Sixty-four in mind, and written the text on the assumption that you are sitting at a warm Commodore 64 keyboard as you read. The result is a gentle but thorough introduction to Machine Code and Assembly Language programming, assuming no prior experience other than a modest familiarity with BASIC.

We begin by discussing how numbers are represented in Machine Code (hexadecimal, signed and unsigned binary, positive and negative numbers) and how—and where—the code is stored in the memory. Next we take a look at the internal structure of the 6510 (and 6502) microprocessor, the Brain of your Sixty-four, from the programmer's point of view. It has a number of special memory areas, called registers, and we say what these do. A simple Machine Code program is then analysed in detail to show how it differs from BASIC.

Some of the difficulties in Machine Code programming can be avoided by making the computer do the work. We develop a BASIC program (LOADER) to help you write, edit, load, and run Machine Code, and to allow you to save programs to tape or disc, and load them back into memory. This program should itself be saved on tape or disc, ready for use in later chapters.

With our BASIC toolkit ready, we are able to introduce the main Machine Code instructions and some of the important techniques: arithmetic, branching, looping, flags, the stack, subroutines, logical operations. This is the 'theory' section and it covers essentially every 6510 instruction.

In the final chapters we develop Machine Code programs that exploit specific features of the Sixty-four: sprites, colour, keyboard control of moving graphics, low and high resolution graphic displays. The main emphasis is on simple Machine Code programs that can be *understood* and used as building blocks in more complicated programs. We want you to learn to *write* Machine Code, not just copy it!

A noteworthy feature is the program MINIASS, which 'borrows' the Commodore's BASIC editor and cunningly enlists its aid to edit Machine Code instead (saving us all a lot of trouble writing a decent editor). The Machine Code is then loaded automatically into memory from the BASIC program area, ready for execution.

To round off the discussion, we have provided a large number of appendices which will prove invaluable in writing Machine Code: tables of hex/decimal conversions, mnemonics, opcodes, addressing modes, sprite registers, flag behaviour, keyboard scan codes.

This book provides a comprehensible but thorough introduction to 6510 and 6502 Machine Code in general, and to the Commodore 64 in particular. Machine Code is challenging but rewarding. Try it!

The Rubáiyát of Programmer Khayyám

Awake! For Morning's fickle hand doth load
Updated software in the daylight mode.
    Return from sluggish subroutine of night:
DIM the array, but brilliant the code!

Myself when young did frequently frequent
The data-punching rooms, and heard great argument;
    But evermore it seemed I must emerge
By that same interface wherein I'd went.

Ah, but my computations, people say,
Process the text to clearer meaning? Nay,
    Though Man may seek the symbols to construe
The Greater Editor will have his way.

The User programs while the disk-drives whisk;
Taps the mad keyboard of a mind at risk.
    The work of years comes suddenly to naught
As random noise corrupts the floppy disk.

Some for the glories of this world, and some
Sigh for a pointer to the world to come.
    Ah, seize the output, let the record go,
Nor heed the rumble of magnetic drum!

A User-Manual 'neath a labelled tree,
A pint of beer, a ploughman's lunch—and Thee!
    What care I then for megabytes?
Thy tiniest bits yield megabytes for me.

The moving cursor writes, and having writ
Moves on: nor all your piety nor wit
    Shall lure it back to cancel half a line
Nor all your Tears wash out a word of it.

But wait! say ye: The console's cursor keys
Can Backspace, Rubout, Edit as we please?
    Not so! These merely tidy the display:
Still the grim input's in the memories.

Some peek the ROM of Time's predestined flight;
Some seek within Life's RAM new lines to write.
    In vain each strives t'assemble faultless code,
For still Death's Digits poke the final byte.

Machine Code can be used to do things that just aren't possible in BASIC. The aim of this chapter is:

1 To Whet Your Appetite

You wouldn't have bought this book, or be thumbing through it in the bookshop, unless you'd heard that the Commodore 64 can do remarkable things, quickly, in something called *Machine Code*. Now that's true; but the trouble with Machine Code is that, unlike BASIC, it doesn't do your thinking for you. You have to pay much more attention to finicky details, and keep an eye on exactly whereabouts in the machine your code sits. Machine Code is emphatically *not* 'user-friendly', and to begin with looks rather like Egyptian hieroglyphs, and has the charm and immediate comprehensibility of an Urdu telephone directory.

It's not really quite as bad as that, and with practice you'll soon get a feel for it; but you'll certainly need to put in quite an effort before you come to the real payoff. So, to convince you it will all be worthwhile, I'm going to show you a Machine Code routine that can change the colours or characters appearing on the screen in the twinkling of an eye. It's embedded in a BASIC program, and could be speeded up even more by converting the rest of the program to Machine Code too. It would be well-nigh impossible to persuade BASIC to do this job at a quarter of the speed (though I won't say it's totally impossible, because people are very ingenious).

Don't try to understand how all this works: that comes later. Just copy it out and RUN. You'll notice a few commands that you probably don't use very often, namely:

SYS

which tells the computer to carry out the Machine Code, and (perhaps more familiar!)

PEEK

POKE

which fiddle about with the memory.

Here we go:

```
10  DATA 0, 162, 0, 173, 0, 192, 157, 0, 4
20  DATA 232, 224, 0, 240, 3, 76, 6, 192, 96
30  FOR T = 0 TO 17
40  READ X
50  POKE 49152 + T, X
60  NEXT
100 K = 4
110 POKE 49152, INT(256 * RND(0) )
120 POKE 49160, K
130 SYS(49153)
140 K = K + 1
150 IF K = 8 THEN K = 216
160 IF K = 220 THEN K = 4
170 GOTO 110
```

Now, *check carefully* that you've copied that out exactly as listed—Machine Code plays nasty tricks if there's an error. Happy? OK, RUN it. Break the program when it becomes too stressful to the eyes!

As a variation, change line 110 to:

```
110 GET A$: IF A$ = "" THEN 110
115 POKE 49152, ASC(A$)
```

Now, when you RUN the program, try pressing different keys on the keyboard and watch the computer responding. It's very quick, isn't it?

WHAT'S HAPPENING?

The way this works is that the computer is being told to fiddle about with two areas of memory: the *screen memory* (which holds character data) and the *colour memory* (which holds colour data). See *Easy Programming**, Chapter 19. (Incidentally, in the first printing of that book, someone got Figures 19.1 and 19.2 interchanged by mistake. Sorry about that.) The actual Machine Code is contained in the DATA statements in lines 10 and 20; it is loaded into a suitable area of memory in lines 30–60. Line 130 tells the computer to run the Machine Code routine that starts in that memory area.

* *Easy Programming for the Commodore 64*, Stewart and Jones, Shiva Publishing.

Anyway, I hope that's given you some idea of what a very simple piece of Machine Code (only 18 bytes of memory and, as we'll see later, only 8 Machine Code commands) can do. Short Machine Code routines can greatly enhance the capabilities of a BASIC program.

What structure should numbers *take in a machine code program?*

2 Numbers in Machine Code

We normally think about numbers in terms of tens. If I write the number 3814 we all understand that to mean:

3 × 1ØØØ + 8 × 1ØØ + 1 × 1Ø + 4 × 1

and we can see that to get a 'place value' from the one on its right we simply multiply by ten. We say the number is in *base* ten.

Because we've been doing this for as long as we can remember, it's difficult to realize that there are other, perfectly sensible, ways of doing the same job. Early computer designers certainly didn't; they used base ten representations in their machines and hit some nasty snags. Mostly they were caused by the fact that electronic amplifiers don't behave the same way for all the signals you want to input to them. For instance, an amplifier that is supposed to output double its input signal may well do so for inputs of 1, 2, 3 and 4 units; but then it starts to 'flatten off' so that an input of 5 produces an output of only 9.6; 6 produces 10.8; and maybe you can hardly tell the difference between the outputs for inputs of 8 and 9.

Put a music tape in your cheapo cassette recorder and wind up the volume. Hear the distortion in the loud bits? It's the same effect.

Pioneer computer designers didn't hear any distortion; they just found that the machines couldn't distinguish between different digits at times, and that was hopeless for a computer. So they had to rethink their number representation to suit what the electronic gubbins would do best.

The simplest thing you can do with an electrical signal is to turn it on or off; so you can represent the digits Ø (off) and 1 (on) satisfactorily. Distortion no longer matters. It's clear whether a signal is present or not regardless of how mangled it is. But can we devise a number system which only uses Øs and 1s?

Yes. In a base ten number, the largest possible digit is 9. Add 1 to 9 and you get 1Ø—a *carry* has taken place. We can write any number using any other base we choose, and the largest possible digit will always be one less than the base. If the base is 2, the largest digit is 1, so a base 2 (or *binary*) number only contains Øs and 1s.

What about the place values? In the base ten case we got those by starting at 1 (on the right) and multiplying by 10 every time we moved left one place. For a binary number we still start at 1, but we multiply by 2 every time we move left.

So for instance the binary number 1101 can be converted to base 10 like this:

```
1   1   0   1
|   |   |   └──────────────► × 1 ──► 1
|   |   └──────────────────► × 2 ──► 0
|   └──────────────────────► × 4 ──► 4
└──────────────────────────► × 8 ──► 8
                                     —
                                = 13
```

Converting the other way is easy as well; take 25 for example. If we write down the binary place values:

32	16	8	4	2	1

and work from the left, it's clear that we need a 16, which leaves 9, and that's made up of an 8 and a 1, so 25 is:

0	1	1	0	0	1

HEXADECIMAL CODE

This is fine for relatively small values, but a bit messy for large ones. There are a number of quick conversion techniques; but I want to examine a procedure which makes use of *hexadecimal* code, because it will stand us in good stead later.

A number in hex (nobody ever says 'hexadecimal', except me, just now) is a number in base 16. So the place values are obtained by successive multiplications by 16. The first five are:

65536	4096	256	16	1

'Hang about!' everybody's saying. 'Those are nasty numbers, and anyway, in base 16 the largest digit has the value 15. Things are getting complicated.'

Bear with me. We handle the problem of digits greater than 9 by assigning the letters A–F to the values 10–15. So the number 2AD in hex converts to decimal like this:

```
2   A   D
|   |   └──► × 1   ───►  13      (D = 13)
|   └──────► × 16  ───► 160      (A = 10)
└──────────► × 256 ───► 512
                        ───
                      = 685
```

Now for the nice feature of hex. Because 16 is one of the binary place values (the fifth one) it turns out that each hex digit in a number can be replaced by the four binary digits which represent it. (By the way, 'binary digit' takes almost as long to say as 'hexadecimal' so it's normally abbreviated to *bit*.) Table 2.1 shows the conversions:

Table 2.1

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

A more extensive table is given in Appendix 1.

Now suppose we want to convert 9041 to hex. First we extract two 4096s, then some 256s and so on like this:

```
                9041
2 × 4096 =      8192 −
                ────
                 849
```

```
3 ×  256 =     768 –
              ----
                81
5 ×   16 =      80 –
              ----
                 1
1 ×    1 =       1 –
              ----
                 0
```

So the hex representation is 2351.

Now we just copy the digit codes from the table:

2	3	5	1
0010	0011	0101	0001

and that's the binary equivalent of 9041; just run the four blocks together to get 0010001101010001.

The hex-to-binary conversion is so easy that, more often than not, we leave numbers in hex even when, ultimately, we need them in binary. After all, it's easy to make an error in copying long strings of 0s and 1s.

CONVERSION BY COMPUTER

Here's a program to convert from decimal to hex. It successively divides the number by 16, looking at the remainder each time. So it works out the digits in the opposite order to the way I did it above.

```
 20  LET H$ = " "
 30  INPUT "DECIMAL NUMBER"; DN
 40  N = INT(DN/16)
 50  M = DN – 16 * N
 60  IF M > 9 THEN M = M + 7
 70  H$ = CHR$ (M + 48) + H$
 80  DN = N
 90  IF DN > 0 THEN 40
100  PRINT "HEX VALUE IS:"; H$
```

Experiment, converting various decimal numbers to hex. (They have to be positive whole numbers 0, 1, 2, ... etc.)

Here's the code to convert in the opposite direction (hex to decimal).

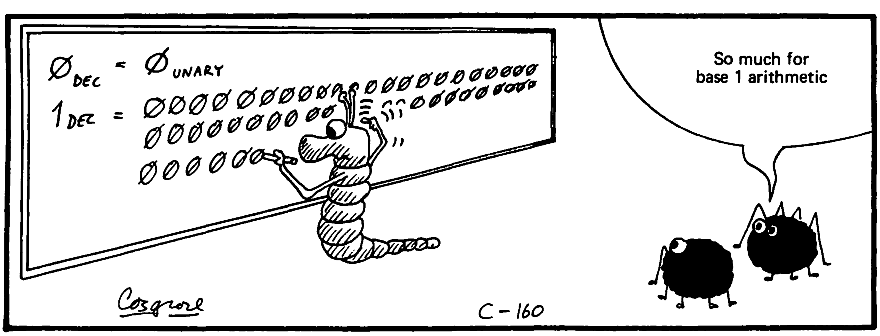
```
110  LET DN = 0
120  INPUT "HEX NUMBER"; H$
```

```
13Ø  FOR T = 1 TO LEN(H$)
14Ø  D$ = MID$(H$, T, 1)
15Ø  A = ASC(D$)
16Ø  A = A − 48
17Ø  IF A > 9 THEN A = A − 7
18Ø  DN = 16 * DN + A
19Ø  NEXT
2ØØ  PRINT "DECIMAL VALUE IS:"; DN
```

We could tie these routines together with a little menu:

```
 2  PRINT "DEC/HEX CONVERTOR"
 3  PRINT "1) DEC − > HEX"
 4  PRINT "2) HEX − > DEC"
 5  PRINT "3) END"
 6  PRINT "ENTER 1, 2, OR 3"
 7  INPUT SEL
 8  IF SEL = 1 THEN GOSUB 2Ø
 9  IF SEL = 2 THEN GOSUB 11Ø
1Ø  IF SEL = 3 THEN STOP
15  GOTO 6
```

and, of course, we'll need RETURNs at lines 1Ø5 and 21Ø.



To deal with negative numbers, the machine uses a clever trick.

3 Positive and Negative

Now that we've seen something about manipulating binary numbers let's return to looking at the way they are handled inside the machine. Usually, a number is held in a fixed number of bits, often 16 or 24 or 32, depending on the machine design. This number of bits is called the *word size* for the machine.

Let's examine what numbers could be held in a 4-bit word:

4-bit pattern	Decimal value
ØØØØ	Ø
ØØØ1	1
ØØ1Ø	2
ØØ11	3
Ø1ØØ	4
Ø1Ø1	5
Ø11Ø	6
Ø111	7
1ØØØ	8
1ØØ1	9
1Ø1Ø	1Ø
1Ø11	11
11ØØ	12
11Ø1	13
111Ø	14
1111	15

It's obvious why bigger word sizes are chosen in practice; a machine which can only represent the numbers Ø to 15 is unlikely to be adequate. But there are two other problems; the notation can't represent fractional values (7.14, for instance) and it can't represent negative numbers.

We'll ignore the fractions problem because most machine code routines only use integers, but the way in which negative numbers are dealt with is more pressing.

The technique is simple: if you've got the binary representation of a positive number and you want to create its negative equivalent you do two things:

1. Change all the Øs to 1s and all the 1s to Øs (this is rather picturesquely called 'flipping the bits').
2. Add 1 to the result.

For instance, suppose you want −3.

3 = ØØ11 in a 4-bit word	
Flipping the bits gives:	11ØØ
Now add 1:	+1
	11Ø1

So 11Ø1 represents −3. It's called the *2's complement* of ØØ11.

I'm not going to explain exactly why this works, but you can prove to yourself that it does in any particular case like this:

If we add 3 to −3 (or 5 to −5 or anything to minus itself) we should get zero. So:

	ØØ11	(= 3)
+	11Ø1	(= −3)
=	1ØØØØ	
	111	(Don't forget that 1 + 1 = Ø carry 1 in binary!)

So we *don't* get ØØØØ at all; but the junior 4 bits *are* zero, and if we're working in a 4-bit word the senior bit will just drop off the end. (For a convenient analogy, think about a car trip-meter with 3 digits; if it reads 999 and you drive an extra mile, it reads ØØØ and a '1' has 'dropped off' the left hand end).

In other words we should have seen it like this:

$$
\begin{array}{cc}
 & \boxed{\text{ØØ11}} \\
+ & \boxed{\text{11ØØ}} \\
\hline
\swarrow & \boxed{\text{ØØØØ}} \\
1 &
\end{array}
$$

This always works provided that the number of bits is fixed throughout. Don't forget to include leading zeros to make up the number of bits to this standard length, *before* taking the 2's complement.

Let's rewrite the 4-bit table of values, now including negatives:

Decimal	Binary	2's complement	Decimal
Ø	ØØØØ	ØØØØ	Ø
1	ØØØ1	1111	−1
2	ØØ1Ø	111Ø	−2
3	ØØ11	11Ø1	−3
4	Ø1ØØ	11ØØ	−4
5	Ø1Ø1	1Ø11	−5
6	Ø11Ø	1Ø1Ø	−6
7	Ø111	1ØØ1	−7
8	1ØØØ	1ØØØ	−8
9	1ØØ1	Ø111	−9
1Ø	1Ø1Ø	Ø11Ø	−1Ø
11	1Ø11	Ø1Ø1	−11
12	11ØØ	Ø1ØØ	−12
13	11Ø1	ØØ11	−13
14	111Ø	ØØ1Ø	−14
15	1111	ØØØ1	−15

Straight away we see that there's a problem; every bit-pattern occurs twice so that, for instance, 1ØØ1 could mean 9 or −7. So we'll have to restrict the range of values still further. I've drawn a dotted line around the region we actually choose to represent. If you look at the senior (leftmost) bit in each of the patterns you'll notice that it's 'Ø' if the number is positive and '1' if the number is negative. This is obviously a very convenient distinction.

So the range of numbers we can get into a 4-bit word is −8 to +7. For 5 bits it would be −16 to +15. For 6 bits it will be −32 to +31 and so on.

A 16 bit word (which is important so far as the Sixty-four is concerned) holds the range −32768 to +32767. A table of 2's complement notations for 8-bit words is given in Appendix 1.

To program in Machine Code, you must know exactly where information is stored in the computer, and in what form.

4 Memory Organization

As you no doubt know, the computer's memory comes in two types:

1. ROM (Read Only Memory) which contains permanent information that can be used but not changed by the programmer.
2. RAM (Random Access Memory) which can be modified at will.

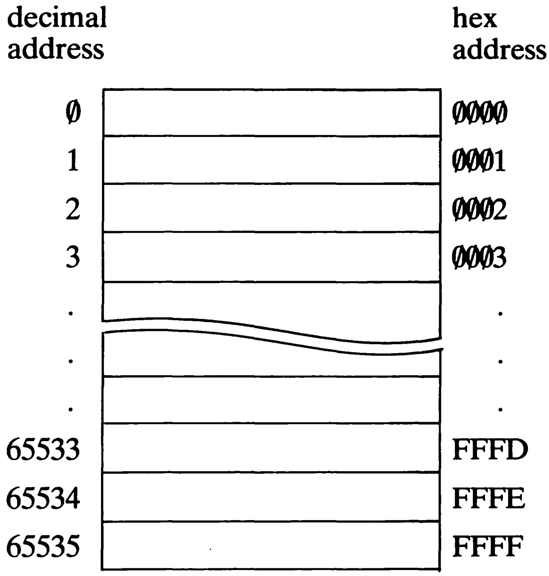
Both ROM and RAM are organized in a way which appears to the programmer to be a single long list of *memory locations*. Each location is able to store a single *byte* of information. A byte is a word made up of eight bits, such as 10011100: there are 256 possible bytes, whose decimal values range from 0 to 255. A byte can also be represented by a two-digit hexadecimal number, ranging from 00 to FF.

Associated with each memory location is its *address*, which acts as a reference number. On the Sixty-four, the possible addresses run from 0 to 65535 decimal. Each address can be written as a four-digit hexadecimal number, from 0000 to FFFF. That means you can represent an address with two bytes (16 bits) of information. Note that 65536 = 256 * 256. A *kilobyte* of memory is 1024 bytes; and 65536 is 64 kilobytes (64K) of memory—which is why the *Sixty-four* is called what it is.

(Actually, that's not quite true, because the Sixty-four has some additional memory areas used for special purposes. However, you can only get at 64K of it at any given time. Other banks of memory can be switched in or out as appropriate. See *Easy Programming*, Chapter 13, or the *Reference Guide**, page 260. I'll ignore this possibility to keep the story simple.)

* *Commodore 64 Programmer's Reference Guide*—available from your Commodore dealer.

So, without going into fine details, we can picture the memory like this:



On this scale, a complete diagram is about ¼ mile (400 metres) long!

PEEK AND POKE

From BASIC, you can gain direct access to a memory location by using the command:

PEEK

to see what's in it (which will work on ROM and RAM), and

POKE

to change its contents (RAM only). For the full low-down on these see *Easy Programming*, Chapter 13. A brief reminder will suffice here.

To find the contents of address AD you use

PEEK(AD)

with AD in *decimal*. For instance, try this program:

```
100  FOR AD = 900 TO 920
110  PRINT AD, PEEK(AD)
120  NEXT
```

If you RUN this, you'll end up with a list of the contents (in decimal) of the memory locations whose addresses run from 900 to 920 (decimal).

The command POKE is used in the form:

```
POKE AD, NUM
```

where AD is the address, NUM the number to be put into it (Ø–255, in decimal). For example, add this routine to the three program lines above:

```
1Ø  FOR AD = 9ØØ TO 91Ø
2Ø  POKE AD, 77
3Ø  NEXT
```

Run the lot. You'll find that the contents of addresses 9ØØ–91Ø have now become 77. (This is 4D in hex.)

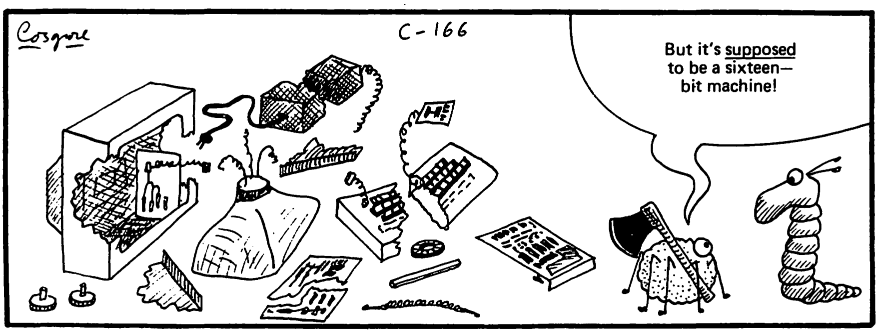
There are some areas of RAM in which POKE appears not to have the expected effect. This is due to the BASIC operating system, which uses some parts of the memory and clobbers your POKEs. The addresses 9ØØ–92Ø above are actually in an area known as the *Cassette Buffer*, which remains unclobbered provided you don't LOAD or SAVE programs. Try LOADing a program and then PEEKing addresses 9ØØ–91Ø. Are they still set to 77?

This is a problem that we must address (no pun intended or taken) later on, when we want to store Machine Code. It's not hard to find a safe place to put it; but it's important to do so.

Machine Code programs are very rigid as regards the way addresses are specified. Addresses are always four-digit hex numbers, such as

A1C7   FFFC   55DØ   ØØØ7

and leading zeros, as in the final example, are *included*.



PAGES

Each 256-byte section of memory is known as a *page*. This means there are 256 pages. The first two hex digits of an address give its *page number*. For instance, the addresses above are on pages:

A1   FF   55   ØØ

respectively. *Page zero* (ØØ) is special for Machine Code, and is treated in a rather different way from all other pages.

MEMORY MAP

You've got to be able to find your way around in the Sixty-four's memory, to be able to influence the way the beast behaves. With the computer in its standard configuration, the most important memory areas are as follows:

Decimal	Hex	Uses
0–827	0000 – 033B	Operating system
828–1019	033C – 03FB	Cassette buffer
1024–2023	0400 – 07FF	Screen memory
2040–2047	07F8 – 07FF	Sprite data pointers
2048–40959	0800 – 9FFF	BASIC area
40960–49151	A000 – BFFF	BASIC ROM *or* 8K RAM
49152–53247	C000 – CFFF	4K RAM
53248–54271	D000 – D3FF	VIC chip (sprites, video display)
54272–55295	D400 – D7FF	SID chip (sound)
55296–56319	D800 – DBFF	Colour memory
56320–57343	DC00 – DEFF	Input/output etc.
57344–65535	E000 – FFFF	KERNAL ROM *or* 8K RAM

BIT NUMBERING

There is a conventional way to number the bits in a byte:

7	6	5	4	3	2	1	0

So bit 0 contributes 1 to the value, bit 1 contributes 2, bit 2 contributes 4; and in general bit N contributes 2↑N. The more senior bits (those more to the left) have higher numbers and count more towards the value of the byte (just as do the digits in decimal).

Similarly, in a two-byte address, the two left-hand hex digits form the *senior* (or *high*) byte and the two right-hand digits form the *junior* (or *low*) byte. For instance:

```
C3FA
│ └──────── junior or low byte
└────────── senior or high byte
```

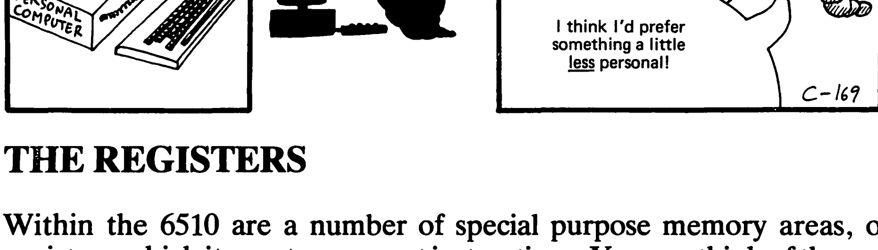
And now, let me introduce you to the Brains behind this organization:

5 The 6510 Microprocessor

At the heart (or brain) of your Sixty-four is a remarkable (though by today's standards a trifle outdated) piece of technology: the 6510 microprocessor chip. It's your computer's *Central Processing Unit* or CPU, and it contains all the circuitry needed to perform logic and arithmetic, and to control the way everything else works. It's a modified version of the famous MOS Technology 6502 chip; and as far as Machine Code programming goes, the two are identical. (I mention this because most of the available books are about the 6502: you can buy these, safe in the knowledge that anything in them will apply equally well to the 6510.)

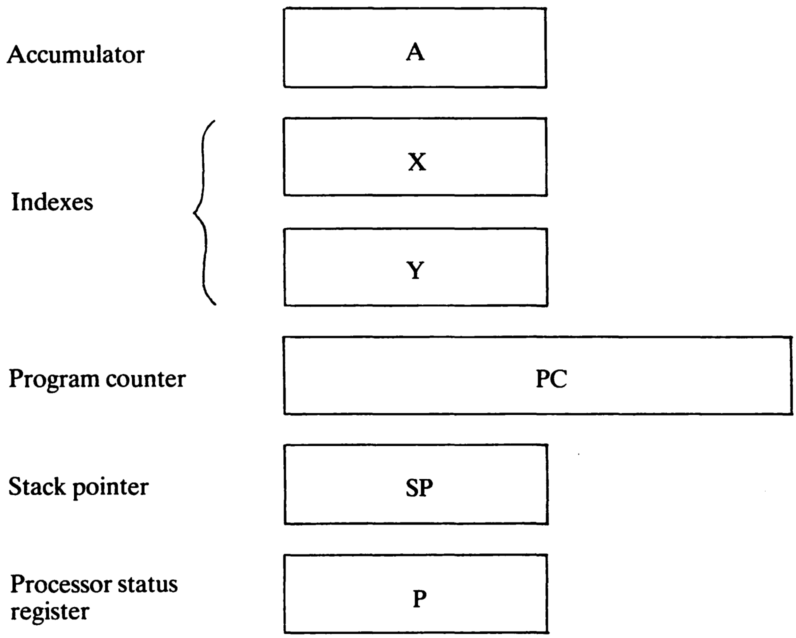
As microprocessors go, the 6510 is reasonably simple; but there are a number of minor complications and side issues which, frankly, I'd prefer not to discuss. A book full of ifs and buts and maybes makes for rocky reading. So I'll warn you right now that I'm not always going to tell you the *whole* truth. Rather than hedge about with confusing qualifying remarks where it really doesn't matter except to an expert, I'll slide over the odd fine point.

In particular, the exact physical layout of the 6510 doesn't matter to us: what we need to know is how to think about it when writing a program. So let's take a quick look at its major features.



THE REGISTERS

Within the 6510 are a number of special purpose memory areas, or *registers*, which it uses to carry out instructions. You can think of them as being arranged like this:



Each register holds one byte, except for the PC-register which holds *two* bytes. You'll see why in a minute. To get us oriented, here's a quick run-down of what they all do. I'll say more later, when we come to make use of them.

THE ACCUMULATOR

This is the basis of all arithmetical and logical operations. For example, to store a particular byte in memory (as in the BASIC POKE) you must:

1. Load it into the accumulator.
2. Store the contents of the accumulator in memory.

You'll find you spend a lot of your time shovelling stuff into the accumulator and hauling it out again. If you want to add (or subtract) two numbers, you must put one in the accumulator, then add (or subtract) the other, and then look in the accumulator to see what the result was.

Since the accumulator is only 8 bits wide, you can only do arithmetic on numbers up to 255. We'll see how to get round this later on, too.

After programming in BASIC, with its limitless range of variables, it takes a while to get used to the dreadful fact that *there is only one accumulator*. A good way to think about what you have to do is to imagine a pocket calculator with a single 8-digit display. Whenever you do a calculation, the result ends up in the display. Whatever was there before is lost—*unless* you take the precaution of memorizing it first. The accumulator is just like this.

THE INDEX REGISTERS

The 6510 has two *index registers*, X and Y. These store numbers that can be used to run through areas of memory one step at a time. They're useful for lists, tables, or anything that requires something to be done to a whole block of memory. You can also use them to cobble together the Machine Code version of a BASIC FOR/NEXT loop.

THE PROGRAM COUNTER

You only make use of this in an indirect way, and you don't normally need to worry about what it's doing. It tells the CPU which program instruction it should carry out next. This is important, because you can make the program jump to a different command by changing the value stored in the PC-register. This gives the analogue of BASIC's GOTO command. In actual fact the PC-register holds the *address* of the memory location containing the code for the next command. Since addresses are two-byte hex numbers, the PC-register also has to be two bytes long. That's why! For more information, see Chapter 11.

THE STACK POINTER

There's a special memory area in the Sixty-four used for temporary storage during calculations, known as the *stack* (see Chapter 15). The SP-register tells the CPU whereabouts the business end of the stack is. The stack is also crucial to the use of *subroutines* in Machine Code.

THE PROCESSOR STATUS REGISTER

This contains information that can be used to take decisions. Is a number positive? Negative? Zero? Did an arithmetical operation result in a carry digit? Every time a command is obeyed, the P-register is automatically updated. (See Chapter 10 on *flags*.)

That's the bare bones, but of course there's more to tell. (In computing there's *always* more to tell.) To get us used to Machine Code painlessly, we'll take a look at some simple but instructive examples first. Then we'll be ready to discuss how to make effective use of the 6510's registers and commands.

The best way to understand how to write a program in Machine Code is to see what happens when the computer works its way through a simple example of one.

6 A Machine Code Program

The aim of this chapter is to show you what form a Machine Code program takes when it's stored in memory; and what kinds of nifty footwork go on inside the CPU when it runs the program. I'm going to start with one of the simplest programs possible: an 8-bit addition routine. This will take two numbers between Ø and 255 (decimal) and add them up. In BASIC this would be pretty easy:

```
1Ø  INPUT M, N
2Ø  L = M + N
```

In Machine Code... well, we'll see!

In BASIC we rapidly get used to the idea that a particular byte of memory can have more than one meaning. For example it could be a number, or the ASCII code for a character, or instructions for controlling a Sprite. Its meaning depends not so much on where it is, as *what the computer intends to do with it*. And the possibilities for *that* were set up by whoever designed the circuits.

It's the same in Machine Code. The contents of a particular memory location may be treated as a positive number, or a signed number between −128 and 127, or an instruction code. If you write the program correctly, the computer will always know which meaning you intend. However, if you make a mistake, there is a definite chance that the computer will get confused. As a result, when a Machine Code program goes wrong, the effect can sometimes be rather bizarre.

THE PROGRAM

First, I'll show you what the program looks like when it's sitting in memory. I'll store it from location 49152 onwards, that is, CØØØ (hex) onwards. This once only, I'll give you the contents of memory in hex, binary and decimal. (Hex is what you'll have to learn to think in for Machine Code; binary is what's actually in the hardware; and decimal is what you'll see if you PEEK.)

Table 6.1

	Address	Contents of address		
		Hex	Binary	Decimal
data	C000	07	00000111	7
	C001	05	00000101	5
	C002	00	00000000	0
program	C003	18	00011000	24
	C004	AD	10101101	173
	C005	00	00000000	0
	C006	C0	11000000	192
	C007	6D	01101101	109
	C008	01	00000001	1
	C009	C0	11000000	192
	C00A	8D	10001101	141
	C00B	02	00000010	2
	C00C	C0	11000000	192
	C00D	60	01100000	96

I've done several things here to help *us* see what's going on. First, I've labelled two areas of memory as 'data' and 'program'. The program is going to use the 'data' area as storage for variables. I've also drawn horizontal lines to break the program into its individual commands: note that some are three bytes long, some only one byte. (Two-byte commands can also occur, but not in this program.)

When the computer first gets hold of the program, it does not 'know' any of this: all it has is a list of bytes. But, as it runs through the program, it can tell from the *context* whether a given byte is program, data, or whatever.

WHAT HAPPENS WHEN IT RUNS

Leaving aside, till the next chapter, the by no means trivial task of feeding these bytes into the correct addresses, let's see what the CPU does when it's told to run this program. The program itself starts at address C003; and the programmer kicks off by telling the computer to load this address into the Program Counter register.

The computer now 'knows' that there's an instruction coming, which it must decode. It uses the address held in the PC-register to look up the code, which is 18. Circuitry already wired into the chip tells it that this means 'Clear Carry flag'. This refers to the P-register, and is a small piece of spring-cleaning needed to make sure everything starts off neat and tidy, uncorrupted by traces of previous programs.

The computer also 'knows' that code 18 is a *1-byte code*: this means that in order to find the next command it must bump the PC up by 1. The PC now holds CØØ4. This address contains the code AD, which means 'Load the accumulator with the number stored in an address given by the next two bytes of program'. The next two bytes are ØØ and CØ. The computer puts these together in the order CØ, ØØ, to get address CØØØ. This is in the data area, and contains Ø7. The computer therefore puts the *number* Ø7 in the accumulator.

Now all this has used up *three* bytes of program: AD, ØØ, CØ. In other words, AD is a *3-byte code*. To get the next instruction, the PC must be bumped up by 3. So now the PC contains CØØ7 (the address of the first byte of program after the AD, ØØ, CØ sequence just carried out).

By now you'll be getting the idea. The computer now decodes whatever is in CØØ7. This is 6D, which means 'Add to the contents of the accumulator whatever the number is that's stored in the address specified by the next two bytes of program'. The next two bytes are Ø1, CØ; as before these refer to address CØØ1, containing the byte Ø5. So the CPU adds Ø5 to the Ø7 already in the accumulator, getting ØC. (Hex, remember? 5 + 7 = 12 in decimal, which is ØC in hex.) That was also a 3-byte code, so the PC goes up to CØØA.

That's the code 8D, which means 'store the contents of the accumulator in the address specified by the next two bytes of program'. The next two bytes, Ø2 and CØ, refer to address CØØ2. So the computer stores the number ØC in address CØØ2, another 3-byte code; so the PC goes up to CØØD.

Decoding CØØD, which contains 6Ø, the computer finds it now has to 'return to BASIC'. So it does, ending the execution of the Machine Code. That was a 1-byte code only, so the PC bumps up by one to read CØØE; but now we're back in BASIC and that promptly takes over control of all the registers.

WHAT IS IT?

What did it achieve? It took the contents of address CØØØ, added to that the contents of CØØ1, and stored the answer in CØØ2. It's an 8-bit adder. If we changed the contents of CØØØ to 1A (26 decimal) and CØØ1 to ØE (14 decimal) then CØØ2 would end up containing the sum, which is 28 (4Ø decimal). And so on.

In fact, it's a bit like the BASIC command L = M + N, where now we've chosen to use address CØØØ for the variable M, CØØ1 for N, and CØØ2 for L. Notice that *you* have to decide *where* to put these variables. Ordinarily, BASIC does this for you automatically. In Machine Code, you're on your own.

OPCODES

The code bytes that define a given operation within the CPU are called

Operation Codes or *opcodes*. The program above breaks up into opcodes like this:

Ø7	
Ø5	Data
ØØ	
18	Opcode for 'Clear Carry flag'
AD ØØ CØ	Opcode for 'Load accumulator from address CØØØ'
6D Ø1 CØ	Opcode for 'Add the contents of address CØØ1'
8D Ø2 CØ	Opcode for 'Store result in address CØØ2'
6Ø	Opcode for 'Return to BASIC'

The 651Ø has 56 different instructions, but most of these can be used in several distinct ways (called *addressing modes*; see Chapter 9). There are 151 different opcodes. We'll cover all of the important ones by the end of the book. Some require 1 byte, some 2 and some 3. However, we *won't* have to learn the codes by heart! They are all listed in Appendix 4.

The computer can be made to handle most of the routine work in Machine Code programming . . .

7 Loading and Running Machine Code

Running Machine Code isn't hard. Most of the problems come in loading it (and debugging it, which is a topic worthy of a separate book!). By writing suitable BASIC routines, a great deal of effort can be saved. The main aim of this chapter is to develop such routines. They could be made quite elaborate, but I'd like to keep the listings reasonably short so that we can concentrate on the main objective: the Machine Code itself.

WHERE TO STORE MACHINE CODE

In principle you could put your code anywhere in RAM—but in practice, as I said earlier, BASIC will clobber it if you put it in an area that BASIC happens to be using.

One attractive answer—and the one that I will standardize on in this book—is to use the 4K section of RAM between addresses CØØØ and CFFF (49152–53247 decimal). This area is not used by BASIC, and it's a safe place to put your code. (You may have noticed that your Sixty-four's much-heralded 64K of RAM miraculously becomes '38911 BASIC bytes free' when you switch the beast on: this 4K block of spare RAM is one of the reasons.)

Another place that people often use for very short Machine Code routines is the Cassette Buffer, Ø33C to Ø3FB (828–1Ø19 decimal). That's fine if you don't need to use the cassette; but it's not very good programming practice.

Another method—which you'll *have* to use if (heaven forfend!) you have more than 4K of Machine Code—is to change the pointers that determine the boundaries between memory areas. For instance, you can move the top end of the BASIC area down, leaving free space which BASIC can no longer get its grubby little hands on. See the Reference Guide for details. In this book I'll stick to the area CØØØ–CFFF. I'll refer to this as the *standard space*.

LOADING FROM A DATA STATEMENT

There's a very simple way to load Machine Code, which I used in Chapter 1. It has several disadvantages, which I'll discuss in a moment; but for short routines that you've already debugged and just want to have hanging around ready to use, it's sometimes the simplest and quickest solution. The idea is to incorporate the list of bytes to be loaded into a DATA statement, and then POKE them into place using a loop. For instance, we can load the 8-bit adder above like this:

```
10 DATA 7,5,0,24,173,0,192,109,1,192,141,2,192,96
20 FOR T = 0 TO 13
30 READ X
40 POKE 49152 + T,X
50 NEXT
```

The advantages are relatively obvious. The disadvantages include:

1. The need to convert codes from hex to decimal.
2. The occurrence of the Machine Code *twice*—once in its final 'loaded' locations, and again in the DATA statement. You're wasting memory.
3. If the DATA list is at all long, it's easy to make a mistake when keying it in.
4. It's hard to read a DATA list and see what it really means.

Now, some of these problems could be overcome if you used a DATA list of hex codes, thought of as strings of length 2, and added some program lines to convert these to decimal. You might like to think about that idea.

However, with a little extra effort, we can develop a BASIC program that will not only let us load Machine Code, but also list it out, change it, and indeed run it. To this we turn.

> **Warning:** make sure your cassette recorder is connected up (switch the computer OFF first) before you go any further. There's a fairly long typing job coming up, and you don't want to have to do it again! In fact, you may prefer to type out all of the program lines below *before* trying the program out; and SAVEing them to tape.

A HEX LOADER

I'm going to take the view that in Machine Code programming, any extra information that you can get cheaply is worth having. So the program will print out addresses and codes both in decimal and in hex. It will offer you the option of where to put the code. And it will let you precede the program area with a data area.

I'll give it to you piece by piece, to make it more comprehensible. First, there's a menu:

```
90   PRINT CHR$(147)
100  PRINT "HEX LOADER: OPTIONS"
110  PRINT "L:LOAD      P:PRINT      E:EDIT
     R:RUN      S:STOP"
120  GOSUB 1300
130  IF Q$ = "L" THEN GOSUB 200
140  IF Q$ = "P" THEN GOSUB 800
150  IF Q$ = "E" THEN GOSUB 1000
160  IF Q$ = "R" THEN GOSUB 1200
170  IF Q$ = "Q" THEN STOP
180  GOTO 110
```

To go with this we need a little input routine:

```
1300  GET Q$: IF Q$ = " " THEN 1300
1310  RETURN
```

The program has a lot of single-character inputs, and this method avoids you having to type RETURN all over the place.

Now comes the *load* option:

```
200  PRINT "LOAD DATA AND PROGRAM"
210  PRINT "BASE ADDRESS IN DECIMAL
     (DEFAULT 49152)"
220  INPUT BA
230  IF BA = 0 THEN BA = 49152
240  PRINT BA: PRINT
250  INPUT "NUMBER OF DATA BYTES"; D
260  AD = BA
270  PRINT: PRINT "TYPE CODE IN HEX"
280  PRINT "TYPE S TO STOP": PRINT
```

```
290 PRINT "HEXAD", "DECAD", "HEXCODE",
    "DECCODE"
300 IF AD = BA AND D > 0 THEN PRINT "*DATA*"
310 IF AD = BA + D THEN PRINT "*PROGRAM*"
320 GOSUB 500
330 PRINT HA$, AD,
340 GOSUB 1300: H$ = Q$
350 PRINT H$;
360 IF H$ = "S" THEN RETURN
370 GOSUB 1300: L$ = Q$
380 PRINT L$,
390 GOSUB 600
400 PRINT DC
410 POKE AD, DC
420 AD = AD + 1
430 GOTO 300
```

This involves a couple of hex/decimal code conversion routines, based on the ones given in Chapter 2. The first is:

```
500 HA$ = " ": AM = AD
510 FOR T = 1 TO 4
520 N = INT(AM/16)
530 M = AM - 16 * N
540 IF M > 9 THEN M = M + 7
550 HA$ = CHR$(48 + M) + HA$
560 AM = N
570 NEXT
580 RETURN
```

And here's the second:

```
600 H = ASC(H$): H = H - 48: IF H > 9 THEN H = H - 7
610 L = ASC(L$): L = L - 48: IF L > 9 THEN L = L - 7
620 DC = 16 * H + L
630 RETURN
```

There's more to come, but now we'll take a look at:

HOW TO USE THE LOADER

As an example, I'll take the Machine Code program from Chapter 6 again: the 8-bit adder. Recall that this had three data bytes, and was placed from address CØØØ onwards—the standard space. The complete hex code for it is

```
Ø7  Ø5  ØØ  18  AD  ØØ  CØ  6D  Ø1  CØ  8D  Ø2  CØ  6Ø
```

and we want LOADER to feed this into place.

RUN the LOADER program. The menu comes up: hit key L for the *load* option. The program asks you for the base address in decimal, and tells you the 'default' is our old favourite 49152 (the standard space in decimal). If you type Ø or RETURN the program will automatically assign this as the address at which the Machine Code will start. (If you want any other start address, you input that instead—in decimal.)

You are now asked for the number of data bytes: this is 3, so input that. The computer tells you to input the code in hex, and reminds you that an input of S will stop the loading sequence.

It then types four column headings, which are abbreviations for Hex Address, Decimal Address, Hex Code and Decimal Code. As a reminder it tells you that you are about to input

```
* DATA *
```

The start address comes up in both hex and decimal:

```
CØØØ    49152
```

Press in turn the keys Ø and 7 for the first two hex digits of the Machine Code. The screen now reads

```
CØØØ    49152    Ø7    7
CØØ1    49153
```

and you can type in the next two hex digits Ø5. Keep typing the Machine Code until you reach the 6Ø at the end. (You'll get a reminder when the * PROGRAM * area is reached.) The bottom of the screen now reads

```
CØØD    49165    6Ø    96
CØØE    49166
```

We've finished now, so type S. The program returns to the menu: another S will stop the program.

LOADER works the same way on all other routines. First you tell it the base address (or go for the default); then the number of data bytes (Ø if there are none); and then you type in the hex codes in order, two digits at a time, ending with an S when you've finished. The computer does the rest, and you get a printout on the screen as you go.

THE PRINT OPTION

Since the screen scrolls as you type codes in, you only see the last twenty or so at any given time. If you want to check the listing, you'll need to print it out in single screenfuls* until you reach the bit you want. So LOADER has a PRINT option to do just that:

```
800  PRINT CHR$(147); "PRINT A LISTING"
810  AD = BA
820  PRINT "HEXAD", "DECAD", "HEXCODE",
     "DECCODE"
830  FOR K = 0 TO 19
840  DC = PEEK(AD)
850  GOSUB 700
860  GOSUB 500
870  PRINT HA$, AD, HC$, DC
880  AD = AD + 1
890  NEXT
900  GOSUB 1300
910  IF Q$ = "S" THEN RETURN
920  GOTO 820
```

Again there's a code conversion:

```
700  H = INT(DC/16): L = DC - 16 * H
710  H = H + 48: IF H > 57 THEN H = H + 7
720  L = L + 48: IF L > 57 THEN L = L + 7
730  HC$ = CHR$(H) + CHR$(L)
740  RETURN
```

To use this, just press key P when the menu appears, and you'll get one screenful of listing. Hit key S to stop, and any other key to get the next screenful. (Note: if you use the P option after RUNning LOADER but before setting the base address BA, the computer will assume it is 0. One way round this snag is to add the line

```
805  IF BA = 0 THEN BA = 49152
```

getting the default option again.)

* Or is that 'screensful'?

RUNNING MACHINE CODE

That's easy. The BASIC command

SYS

does the job for you. To run the Machine Code routine starting at address AD, you use

SYS(AD)

So our 8-bit adder, whose *program* part started at address 49155 (C003 in hex), can be run by the command

SYS(49155)

In general, we add a RUN routine to LOADER:

```
1200 PRINT: PRINT "RUNNING"
1210 PRINT "PRESS A KEY: S TO ABORT"
1220 GOSUB 1300
1230 IF Q$ = "S" THEN RETURN
1240 SYS(BA + D)
1250 PRINT: PRINT "PROGRAM EXECUTED": PRINT
1260 RETURN
```

Add these lines to LOADER: you're all set! Now:

1. Use the L option to load in the 8-bit adder code (if you haven't already done so).
2. Use the P option to check that it's right.
3. Use the R option to RUN it.

The computer will wait for you to press a key (and you have the option to press S and avoid a run if you've suddenly remembered some awful mistake). Press something other than S.

Quick as a flash comes the message

PROGRAM EXECUTED

and the menu.

Fine, but where's the answer?

Well, recall that we stored the result of the addition in address C002, that is, 49154. You can check this very easily by using the P option to list out the program again. You should see this entry for C002:

```
C002    49154    0C    12
```

At the start, it was

```
C002    49154    00    0
```

We told it to add 7 and 5, and it's done just that. And the answer, 12, *has* been placed in address C002.

To see that this isn't just coincidence, you can modify the contents of C000 and C001 and then see if C002 still ends up with the sum. One way is to use the direct mode commands:

```
POKE 49152, 23          (say)
POKE 49253, 11          (say)
SYS(49155)
PRINT PEEK(49254)
```

You should now get the result 34, which is 23 + 11. Try repeating this with different numbers (say less than 100) in place of 23 and 11.

IMPORTANT WARNING

When you use SYS to run a Machine Code program, you *must* end it with an

RTS

instruction (opcode 60, **ReTurn** from **S**ubroutine) which in this case gets you back into BASIC. If you don't, the computer keeps churning merrily through memory, interpreting the garbage scattered therein as *bona fide* Machine Code—well, the silly beast knows no better—and *carrying it out*. The result is usually weird to say the least: it's not unusual for the computer to gobble up its own program and commit the electronic equivalent of hara-kiri.

In fact, it's not a bad idea to modify LOADER to tack on a final 60 to anything you give it, just in case you forgot. (A spare one does no harm.) Change line 36= to:

```
360  IF H$ = "S" THEN POKE AD, 96: RETURN
```

THE EDIT OPTION

To make testing easier—and to allow you to correct mistakes—we'll add an editing routine to LOADER. It's extremely rudimentary: it just lets you change the contents of an address, and repeat if you wish. For a fancy editor (the Sixty-four's own BASIC editor, in fact) see Chapter 21.

```
1000  PRINT: PRINT "EDIT": PRINT
1010  INPUT "DECIMAL ADDRESS"; AD
1020  PRINT "NEW CONTENTS HEX"
```

```
1030  GOSUB 1300: H$ = Q$: PRINT H$;
1040  GOSUB 1300: L$ = Q$: PRINT L$
1050  GOSUB 600
1060  POKE AD, DC
1070  PRINT "MORE?"
1080  GOSUB 1300
1090  IF Q$ = "S" THEN RETURN
1100  GOTO 1010
```

Suppose you've done this. RUN, press option E, and input

```
49152        17
```

When asked

```
MORE?
```

hit RETURN and then input

```
49153        0B
```

Again you're asked

```
MORE?
```

but this time you stop by hitting

```
S
```

and get the menu back. Type R to run, then P to list out the result. Look at address C002. It should contain 22(hex) and 34(decimal). Now 17 hex is 23 decimal, 0B hex is 11 decimal, and 23 + 11 = 34. So it worked! You can now edit in various other numbers, run, and print out the results.

SAVING MACHINE CODE

You can't save Machine Code to tape or disc as easily as you can with BASIC. You'll need to write your own routines for doing this. One good way is to use *files*—see the *Reference Guide* or *Easy Programming*, Chapter 34. I'll give you some routines that you can add to the LOADER program.

First you must extend the options:

```
115  PRINT "F:FILE      I:INPUT"
172  IF Q$ = "F" THEN GOSUB 1500
174  IF Q$ = "I" THEN GOSUB 1700
```

Then add a routine to save data to a file:

```
1500  PRINT "MAKE SURE YOU HAVE THE RIGHT
      TAPE"
1510  INPUT "NAME OF FILE"; F$
1520  OPEN 1, 1, 1, F$
1530  INPUT "BASE ADDRESS"; BA
1540  INPUT "LENGTH OF CODE"; LC
1550  FOR T = BA TO BA + LC - 1
1560  Y = PEEK(T)
1570  PRINT# 1, Y
1580  NEXT
1590  CLOSE 1
1600  RETURN
```

If you have a disc drive instead of a cassette recorder, change line 1520 to read

```
1520  OPEN 1, 8, 2, F$ + ", SEQ, W"
```

Now comes the input routine:

```
1700  INPUT "NAME OF FILE TO BE INPUT"; F$
1710  OPEN 1, 1, 0, F$
1720  INPUT "BASE ADDRESS"; BA
1730  INPUT "LENGTH OF CODE"; LC
1740  FOR T = BA TO BA + LC - 1
1750  INPUT# 1, X
1760  POKE T, X
1770  NEXT
1780  RETURN
```

Again, for a disc drive use

```
1710  OPEN 1, 8, 2, F$ + ", SEQ, R"
```

To use these, suppose you've loaded in 100 bytes of code starting at base address 49152. By pressing option 'F' you can save the code to tape. You'll need to have the cassette connected up, of course.

First you'll be asked for the name you want to give the file. Input the name, say

FRED

The tape whirrs as the header block for the file is added to it. It stops. You'll then be asked for the base address; input

49152

You can fancy up the program to use the same base address as the loading routine did, but for flexibility it's worth being able to change this. Next you're asked for the length of code (same remarks apply) and you input

100

now the tape whirrs again: for a longish program you'll notice it stopping and starting several times. (This is the result of the way the cassette buffer works.)

To load it back, use option 'I' and repeat the same steps.

BEFORE GOING FURTHER

Save the finished version of LOADER on to cassette, using

SAVE "LOADER"

because we're going to use it from now on to load all of our Machine Code programs.

MORE TESTS

You don't *have* to use LOADER to run the Machine Code. Plain SYS(49155) will do that. So you can test the whole thing much more quickly if you use a BASIC program:

```
2000  PRINT "8-BIT ADDER: TEST ROUTINE"
2010  INPUT "CONTENTS OF C000"; M
2020  INPUT "CONTENTS OF C001"; N
2030  POKE 49152, M: POKE 49153, N
2040  SYS(49155)
2050  PRINT "CONTENTS OF C002", PEEK(49154)
2060  GOTO 2010
```

Now start with GOTO 2000 and play around to your heart's content.

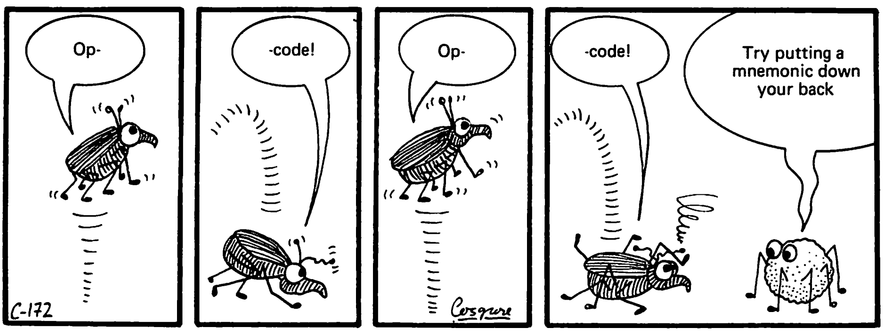
OVERFLOW

I said to use numbers less than 1ØØ. It pays to be suspicious of this sort of cop-out. What happens if we ask LOADER's 8-bit adder to add 2ØØ to 2ØØ? What answer do you expect?

What you *get* is 144. Has the machine gone crazy?

Not a bit of it. The problem, as I've emphasized all along, is that we have built an *8-bit* adder. Any carry digits that go into the ninth bit (256 onwards) are simply lost. Note that 144 + 256 = 4ØØ, the correct answer. Remember the car trip-meter in Chapter 3? The same is happening here.

This phenomenon is called *overflow*. It's something that the programmer has to take care of, if it matters. In fact, when I said above that the carry digit is 'simply lost', that wasn't quite true. There's a slot in the Processor Status Register that lets the computer check whether an overflow occurred. The programmer can use this to take adequate steps to keep the calculation on the right track. I mention it here only as a warning, yet again, that Machine Code leaves most of the thinking up to *you*.



When designing a Machine Code program, something a little more tractable than two-digit hex codes makes life a lot easier!

8 Assembly Language

If you're trying to write a Machine Code program, you've got enough to think about without having to remember all those hexadecimal opcodes. For instance, it's a lot easier to think 'Store the contents of the accumulator in memory' than it is to remember the opcode 8D. There is a systematic set of *mnemonics*, used by programmers to do this. The mnemonic for '**ST**ore the contents of the **A**ccumulator' is just:

STA

and that's a lot easier on the eye.

So the programmer generally works out his program in mnemonics, and only after he's happy does he (or a special program called an *assembler*) convert to hex opcodes. Programs written using mnemonics are said to be in *assembly language*.

Here's the 8-bit adder in assembly language. First let's set up the memory areas:

C000	Data: first number
C001	Data: second number
C002	Data: sum to be placed here
C003	Start of program

Now the program:

CLC		(**CL**ear **C**arry flag)
LDA	C000	(**L**oa**D** **A**ccumulator from C000)
ADC	C001	(**AD**d (with **C**arry) from C001)
STA	C003	(**ST**ore **A**ccumulator in C003)
RTS		(**R**e**T**urn from Machine Code **S**ubroutine)

Now that's a lot easier to follow—especially with a little practice!

What I'm going to do in this chapter is show you a series of simple examples—programs for doing arithmetical operations that we can check easily. I'll write them in mnemonics, explain what they're doing, and convert them to hex. Your job is then to use LOADER to get them into memory, run them, and check that they did the right thing. (Be careful about the data bytes.) I'll save any systematic run-through of the available mnemonics and their opcodes for later chapters.

SUBTRACTION

The mnemonic for '**SuB**tract' is

SBC

The C on the end serves to remind us that any **C**arry digits left over from previous arithmetical operations will be treated as 'borrows' for the purposes of subtraction. That's why the 'add' mnemonic is ADC, not ADD: it too has a carry digit included. To avoid having to worry about these borrows and carries, we adjust the carry before using ADC or SBC. The only potential pitfall is that, while we should use CLC (**CL**ear **C**arry) before an ADC, the correct thing to use before an SBC is the new instruction SEC (**SE**t **C**arry) with opcode 38. This is because the 6510's Carry flag is a bit strange (see Chapter 10).

The program will work in exactly the same way as before: we'll store the two numbers in CØØØ and CØØ1, and their difference in CØØ2. The CPU will have to:

Set the Carry flag
Load the accumulator with the contents of CØØØ
Subtract from that the contents of CØØ1
Store the result in CØØ2
Return to BASIC

So, in assembly language mnemonics, we have:

```
SEC
LDA   CØØØ
SBC   CØØ1
STA   CØØ2
RTS
```

Now we convert to hex, using Appendix 4:

Assembly	Hex
SEC LDA C000 SBC C001 STA C002 RTS	38 AD 00 C0 ED 01 C0 8D 02 C0 60

That's my bit done. Now comes yours: I want you to use this, together with LOADER, to work out 114 − 75 (decimal). See if you can do this on your own *before* reading on.

Here's the way I intended you to do it.

First, work out what 114 and 75 are in hex, using Appendix 1. They're 72 and 4B. Then use the L option on LOADER to load data and program into memory, with the standard base address (default value) and 3 data bytes. The code to load in is data + program, in the order:

$$\underbrace{\text{72 4B 00}}_{\text{data}} \qquad \underbrace{\text{38 AD 00 C0 ED 01 C0 8D 02 C0 60}}_{\text{program}}$$

terminating with S to get back to the menu. Use P to check this went in OK, and S to exit again; finally use R to run the Machine Code and P to find out what's in C002. If all's right with the world, it should be 39 decimal (27 hex).

TOTALLING A LIST

Using the same repertoire of commands, let's consider a similar problem: totalling up a list of five numbers, stored in C000–C004, and putting the result in C005. The program itself will start at C006. No sweat—here's the code in mnemonics, plus its translation into hex:

Assembly	Hex
CLC LDA C000 ADC C001 ADC C002 ADC C003 ADC C004 STA C005 RTS	18 AD 00 C0 6D 01 C0 6D 02 C0 6D 03 C0 6D 04 C0 8D 05 C0 60

This time there will be 6 bytes of data: 5 for the numbers and 1 for the total. Use LOADER to load the whole lot in, with your own choice of numbers to add up, but recall that any total over 255 will have some missed carry digits. I suggest you keep all your numbers below 50 decimal (32 hex) to avoid running into trouble.

A 16-BIT QUIRK

You may have spotted a pattern to the way the addresses are inserted into the opcodes for LDA, ADC, SBC and STA. For instance, when I wanted to store the accumulator in CØØ5, the opcode was like this:

```
STA  CØØ5          8D  Ø5  CØ
```

The second and third bytes of the opcode are the two address bytes—but *in the reverse order.*

This is an inviolable rule for the 6510. Whenever an opcode includes a two-byte address, those two bytes are in the *opposite* order to the way they occur in the address. That is:

Junior byte first, senior byte second.

It's no problem once you get used to it, but you do have to be careful.

ADDING WITH A CARRY

Now let's see how to deal with Carries, and write a 16-bit (2-byte) adder. The data area will look like this:

CØØØ		First number, junior byte
CØØ1		First number, senior byte
CØØ2		Second number, junior byte
CØØ3		Second number, senior byte
CØØ4		Sum, junior byte
CØØ5		Sum, senior byte
CØØ6		Program starts here

The main steps will be:

Clear Carry flag
Load accumulator with junior byte of first number

Add junior byte of second number
Store result in junior byte of sum
DO NOT CLEAR CARRY FLAG THIS TIME
Repeat process for senior bytes

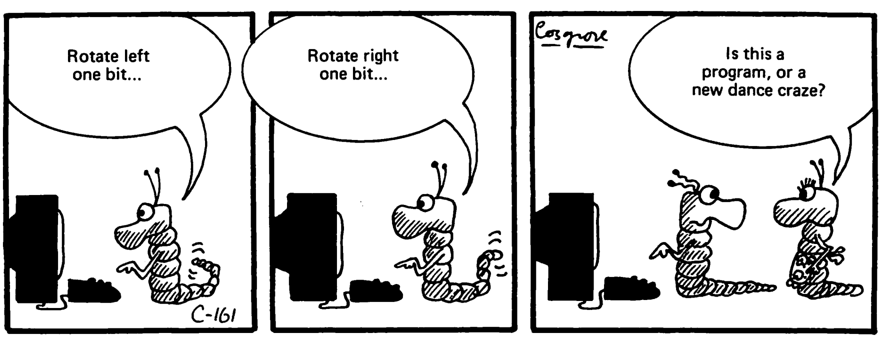
By failing to Clear the Carry, we ensure that any Carry digit resulting from the first addition is *included* in the second.

Here it is in assembly language and hex:

```
CLC          18
LDA C000     AD 00 C0
ADC C002     6D 02 C0
STA C004     8D 04 C0
LDA C001     AD 01 C0
ADC C003     6D 03 C0
STA C005     8D 05 C0
RTS          60
```

Load this with six data bytes, and test it. For instance, to add 30669 (decimal) to 17391 (decimal) we convert these to hex, getting 77CD and 43EF. So we need to put these bytes into data (and zeros in the remaining two data slots) like this:

Address	Contents
C000	CD
C001	77
C002	EF
C003	43
C004	00
C005	00



When we run the program, we get the result:

CØØ4	BC
CØØ5	BB

And BBBC (hex) is 48Ø6Ø (decimal), which is correct.

Try adding another CLC command in the program, after the first STA. Now you'll find we get BABC as the answer, which is 478Ø4. This is 256 too small—and the missing Carry digit is the culprit!

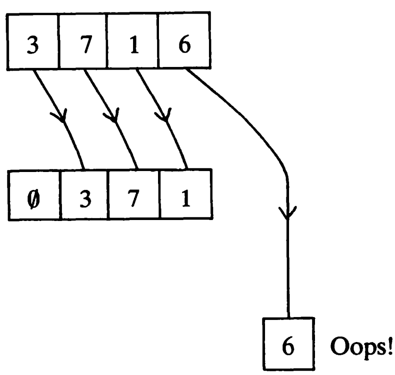
Even though we've taken care of *this* Carry, there's *yet another* Carry that will occur if the total goes over 65535 (FFFF hex), and the current program loses this. (You could think about enlarging the data area by one more byte at CØØ6, to store the final Carry—if any. HINT: if you ADC# ØØ to ØØ the result is the Carry digit.) We'll see just how the Carry works when we consider *flags* in Chapter 10.

HALVING

Things that use tens are usually easy in decimal; and things that use twos are correspondingly easy in binary or hex.

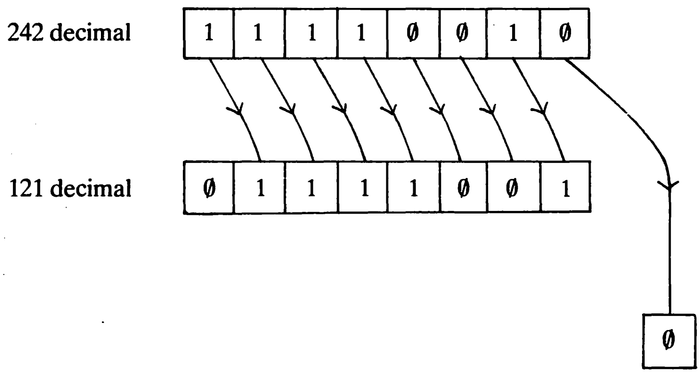
Think decimal for a moment—if you still can! How do you divide 371Ø (say) by 1Ø? Of course, you knock off the last digit, to get 371. This method also works pretty well on a number like 3716: exact division gives 371.6, and if you're prepared to omit everything after the decimal point (round *down*) you get 371, which again has just had the right-hand digit lopped off.

In other words, the number is *rotated* one place to the right, with the rightmost digit falling off the end, like this:



That Ø I've put on the front is harmless; it just keeps the slots tidy.

What decimal does with tens, binary does with twos. So in binary we can divide by two—that is, *halve* a number—by rotating its digits one place to the right. (If the original number is odd, the extra ½ on the end, which is binary .1, gets lost in the wash.) Let's just check that on the number 242 (decimal), which is 1111ØØ1Ø in binary. Here we go:

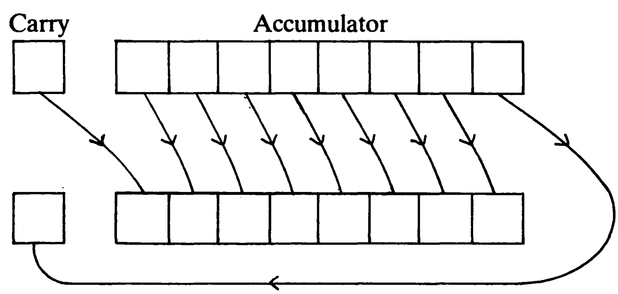


The result is Ø1111ØØ1, or 121 decimal: spot on!

There is a 6510 instruction 'Rotate accumulator right' whose mnemonic is

ROR (**RO**tate **R**ight)

with opcode 6A. If there is a Carry digit left over from a previous operation, this gets moved to the leftmost bit (and the one that I've said 'falls off' actually ends up in the Carry slot):



Sometimes you want this to happen; but if not, a nifty bit of CLC will soon sort it out.

As an example, let's store a number in CØØØ and put half of it (omitting a spare ½ if it's odd) into CØØ1:

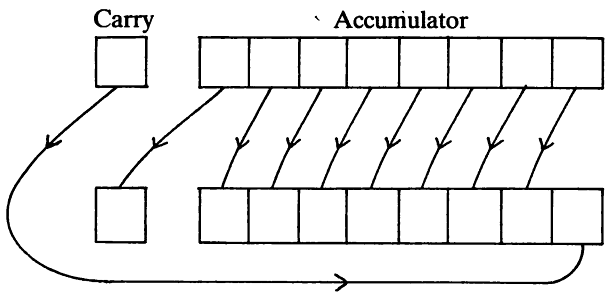
```
CLC          18
LDA CØØØ     AD ØØ CØ
ROR          6A
STA CØØ1     8D Ø1 CØ
RTS          6Ø
```

Load this with two data bytes, and test it.

DOUBLING

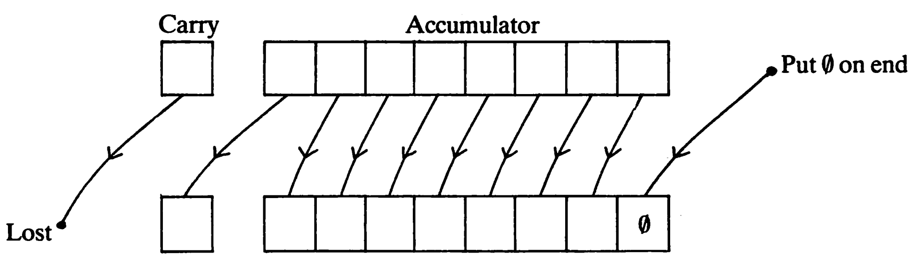
To double a number, we rotate it to the *left*. With overwhelming generosity, the designers of the 6510 have provided us with *two* different ways to do this. Only the effect on Carries varies. The first is:

ROL (**RO**tate Left, opcode 2A)



The other one is:

ASL (Arithmetic Shift Left, opcode ØA)

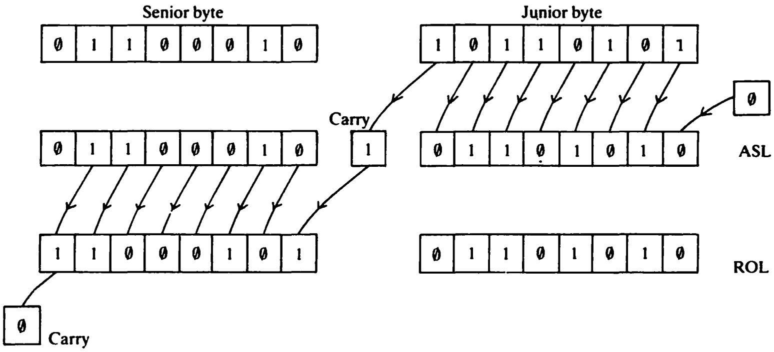


To double a 1-byte number (less than 128 to avoid Carry problems) held in CØØØ, and put the result in CØØ1, we do:

```
LDA CØØØ      AD ØØ CØ
ASL           ØA
STA CØØ1      8D Ø1 CØ
RTS           6Ø
```

(No need for a CLC this time—why?) Load this using two data bytes, and see that it does the job.

However, to double a 16-bit (2-byte) number we use ASL on the junior byte and ROL on the senior, because we *want* the first Carry to shift up:



In the usual fashion, I'll put the number in CØØØ–CØØ1 and store its double in CØØ2–CØØ3 (junior byte first, then senior). The code is:

```
LDA CØØØ        AD ØØ CØ
ASL             ØA
STA CØØ2        8D Ø2 CØ
LDA CØØ1        AD Ø1 CØ
ROL             2A
STA CØØ3        8D Ø3 CØ
RTS             6Ø
```

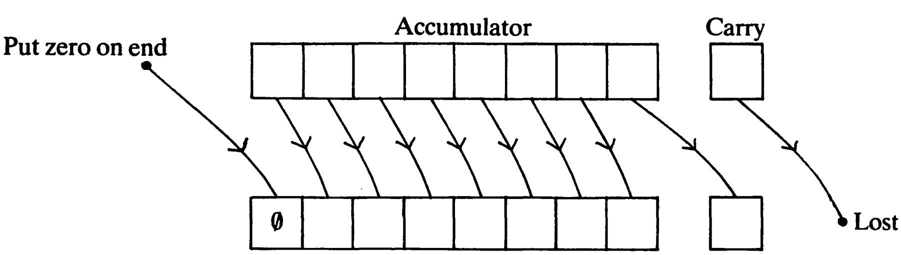
Load this using 4 data bytes, and test it in the usual way.

ANOTHER SHIFT COMMAND

There's one more command in this general order of ideas, which goes with ROR in the same way that ASL goes with ROL. It is:

LSR (Logical Shift Right)

and, like ROR, it does a right shift; but it puts a zero into bit 7. It thus halves an individual byte without having to Clear the Carry first.



Now look, I did say *I wasn't always going to tell the* whole *truth*...

9 Addressing Modes

The 6510 is a more versatile beast than I have hitherto led you to believe. Many of its instructions can be used in *several different ways*—called *addressing modes*—each with its own opcode. It depends on what distinctions you choose to make, just how you count them: I make it 12 different addressing modes altogether, though some people manage to get 13 by being more prepared to split hairs.

The easiest way to see what's going on is by examples. Let me take our old friends LDA and STA to begin with.

IMMEDIATE ADDRESSING

You use this to put a specific *number* into the accumulator (or to operate using a number). Thus, to load 7D (hex) into the accumulator, you use:

```
LDA #7D          A9 7D
```

This is a 2-byte opcode. The first byte, A9, tells the computer 'Load accumulator in immediate mode'. It now *knows* that the next byte, 7D, is *the number to be loaded*.

The # sign (often pronounced 'hash') in the mnemonic reminds the *programmer* that it is the number 7D, not an address of the form 7D, that's involved. The symbol # is used for 'number' in the USA in the same way that Europeans use 'No.' or 'n°'.

STA can't be used in immediate mode; and if you think about it, this should be pretty obvious. The only place you can store something is in an *address*.

ABSOLUTE (NON-ZERO PAGE) ADDRESSING

This is the LDA we've been using happily all along. It loads the accumulator with the contents of the address specified by the next two bytes of code (in the order junior: senior). Thus to load the accumulator *from* (that is, *with the contents of* the address) CØ51, we use

```
LDA  CØ51        AD  51  CØ
```

Similarly to store the contents of the accumulator in the address CØ51 we use

```
STA  CØ51          8D  51  CØ
```

So AD tells the computer 'LDA in absolute mode'; and 8D tells it 'STA in absolute mode'.

The third byte in the opcode is the senior byte of the address; and you'll recall from Chapter 4 that this is the *page number*. It should be non-zero in this mode, because there's a special way to address page zero—known, curiously enough, as . . .

ZERO-PAGE ADDRESSING

If you want to use absolute addressing on page ØØ (addresses ØØØØ–ØØFF hex, Ø–255 decimal) you may *omit* the ØØ senior byte. But, the opcode changes. For example, to LDA from address ØØB6 in page zero, you use:

```
LDA  B6     A5  B6
            |   |
            |   └──────► 2nd byte of address on page ØØ
            └──────────► opcode for zero-page absolute
                         addressing
```

And to STA from address ØØB6 you would use:

```
STA  B6     85  B6
```

Page zero is particularly useful when (as is *not* the case!) you start with a 'naked' 6510, because the omission of the superfluous ØØ byte saves RAM space. The people who wrote the Commodore 64's operating system know this—and the rotten pigs have hogged almost all of page zero! However, they *have* left us mere mortals a miserable *four* token bytes on page zero, at the addresses:

ØØFB
ØØFC
ØØFD
ØØFE

If you want to use page zero, and still have BASIC intact, you should shove everything into these.

IMPLIED AND ACCUMULATOR ADDRESSING

Some operations don't involve anything except the accumulator—and some don't even involve that! Examples of the first type are (one possible mode of) ROR and ROL. To rotate the accumulator to the right you use plain:

```
ROR        6A
```

with *no* extra bytes in the opcode for addresses or numbers.

An example of the second is:

```
RTS        60
```

which we've used to return to BASIC. (More generally, it lets us return from any subroutine to the main program. See Chapter 15.)

As far as this book is concerned, both of these modes with no extra bytes will be considered 'implied addressing'. That is, the 'addressing' mode without an address!

OTHER MODES

The remaining eight modes are somewhat more complicated. They are: *indirect* and *relative* addressing (which I'll describe in Chapter 11 on branching and jumps) and six *indexed* modes (Chapters 13 and 14 on indexing and indirection). Appendix 4 gives all the possible modes for each instruction, together with the corresponding opcodes. Note that many instructions use only one or two modes, and no instruction uses them all.

AN EXAMPLE

Here's a simple example using all four of the modes explained so far. It's a bit contrived, but it should clarify any remaining problems.

1. Think of a number, say 43 decimal, 2B hex; store it in 00FB on page zero.
2. Double it. (Note that it's still in the accumulator too: STA places a *copy* in the desired place, but leaves the original intact.)
3. Add 17 decimal, 11 hex.
4. Place the result in C000, *not* on page zero.

Addressing mode	Mnemonic	Opcode	Number of bytes
Implied	CLC	18	1
Immediate	LDA #2B	A9 2B	2
Zero page	STA FB	85 FB	2
Implied	ROL	2A	1
Immediate	ADC #11	69 11	2
Absolute (non-zero)	STA C000	8D 00 C0	3
Implied	RTS	60	1

(If you decide to test it out, remember to use one data byte CØØØ before the program area.)

Notice how the format of the mnemonics makes it clear which mode is involved:

Implied:	CLC	(no extras)
Zero page:	LDA FB	(one extra byte)
Immediate:	LDA #2B	(# plus one extra byte)
Absolute:	STA CØØØ	(two extra bytes CØ and ØØ)

The other eight modes still to come have their own formats too. Note that the format of the mnemonic is of interest only to the programmer: the computer only worries about the opcode. You can invent your own system of mnemonics if you wish. However, the mnemonics recommended by the manufacturers of the 6510 are an industry standard, so (a) you'll find it easier to read other people's code if you stick to the standard ones; (b) other people will find it easier to read yours; and (c) if you buy an assembler program it will almost certainly use the standard formats.

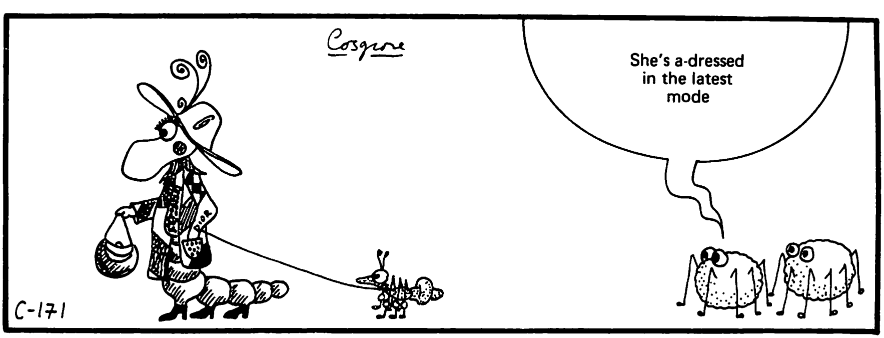
The mnemonics used in this book are non-standard in one respect. It is usual to add the symbol $ to the front of any hex number: that is, to write:

    $F7

instead of plain:

    F7

I've taken the point of view here that it is easier just to standardize on hex throughout (avoiding potential nasties confusing hex with decimal); anyway, I have enough trouble hanging on to my dollars without scattering them blithely about in program listings! However, you should note that on occasion the dollar signs are mandatory (for instance, in most commercial assembler programs). *You have been warned!*

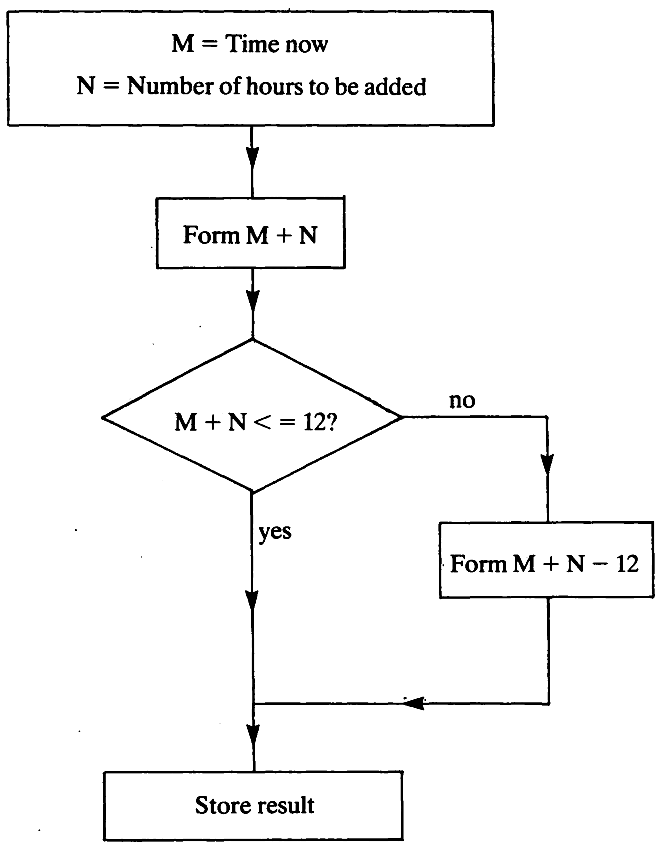


A flexible program must be able to behave in different ways under different conditions. The 6510 keeps a permanent record of the important conditions, and how they are affected by the most recent operation, by setting digits of the Processor Status Register.

10 Flags

An absolutely fundamental technique in computing, which goes back at least to ideas of Charles Babbage in about 1830, is to make the flow of calculation *branch* according to certain conditions. The IF/THEN command in BASIC performs this function.

For example, consider 'clock addition', with a 12-hour clock. Here 1 o'clock + 7 hours = 8 o'clock, as normal; but 9 o'clock + 7 hours = 4 o'clock, not 16 o'clock! You don't just add up the numbers. The flow of calculation goes like this:



Note that we use 12 o'clock, not Ø o'clock! In BASIC you'd do it this way:

```
1Ø  LET M = whatever
2Ø  LET N = whatever else
3Ø  S = M + N
4Ø  IF S < = 12 THEN 6Ø
5Ø  S = S − 12
6Ø  PRINT S
```

I've deliberately used a 'GOTO' approach here—albeit with a tacit use of GOTO—rather than a more 'structured' one, because it gives the clues for Machine Code, which is very far from being 'structured'!

THE PROCESSOR STATUS REGISTER

The crucial problem in branching is to decide whether a given number (here M + N − 12) is positive, zero or negative. That's where the P-register, whose pretentious name decorates this section, comes in. So I'm going to stop dodging the issue, and tell you what it does.

Each individual bit out of the eight bits in the P-register is used as a *flag*. That is, the digit is either:

	set	(to 1)
or	*reset*	(to Ø)

depending on whether some desirable condition does or does not hold at the time. Most operations change this pattern of flags: for a summary see Appendix 5.

Actually, there are only seven flags in the P-register, because one bit is 'reserved for future expansion', which is a delicate way of saying 'we couldn't decide what to do with it'. And some of these seven aren't of much interest to any but hardware buffs. So it's not too bad.

The flags are arranged like this:

N	V	not used ↓	B	D	I	Z	C

Taking them in a convenient (jumbled) order, I'll say what they do:

Z: Zero flag

This is set to 1 if the result of an arithmetical or logical operation is Ø; and reset to Ø if the result is non-zero. (A minor curiosity: if the *result* is zero then the flag *isn't* zero, and vice versa. Computing can drive you mad—mad, I tell you! The point is that a digit 1 in a flag means 'wave the flag' and says 'the desired event has occurred'; and here the desired event is zero.)

N: Sign flag

If the result of an arithmetical operation is negative, this is set to 1; if positive or zero, it is reset to Ø.

Well, that's what it *should* do. Unfortunately, that's not entirely true!

It *is* true if you are thinking of an 8-bit number as a 7-bit number together with a sign bit, as I explained in Chapter 3. That is, if your numbering system goes:

Ø, 1, 2, ..., 126, 127

but then switches to:

−128, −127, −126, ..., −3, −2, −1

instead of continuing from 128 up to 255.

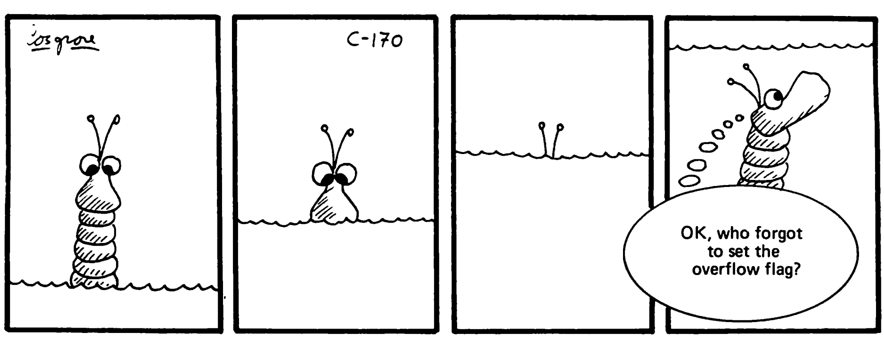
So what it *really* does is tell you what the *leftmost* bit of the result is. (Bytes from Ø to 127 start Øsomething; the rest start 1something.) I'll postpone further discussion until we have a need for it.

C: Carry flag

If addition or subtraction results in a Carry (or Borrow) digit, then the Carry flag signals this event in its own peculiar fashion:

1. On an ADC instruction, if the result goes over 255, so there is a Carry digit, then the Carry flag is set to 1. If not, it is reset to Ø.
2. On an SBC instruction, if the result goes below Ø, giving a Borrow digit, then the Carry flag is *reset to Ø*. If there is no Borrow then the Carry flag is *set to 1*.

You're beginning to understand why I didn't really want to say too much about the flags, right?



The main thing is to keep a clear head. The Carry flag tells you what happened, either way; but you have to remember how to interpret what it tells you, depending on whether you've added or subtracted. Here's a little table to help (it assumes you have M in the accumulator and add or subtract a byte N):

Operation	Result in accumulator	Status of Carry flag
ADC	M + N (omit Carry)	Ø if M + N < = 255 1 if M + N > = 256
SBC	M − N (omit Borrow)	1 if M − N > = Ø (N < = M) Ø if M − N < Ø (N > M)

This is on the understanding that M and N are thought of as unsigned 8-bit numbers between Ø and 255, as usual.

So between them, the C and Z flags will tell you whether M and N are equal, or M < N, or M > N, when M and N are between Ø and 255.

In fact, the C and Z flags (occasionally augmented by the N flag) are the only ones you're likely to want to use unless you get really serious. But, for completeness, here's a quick run-down of the other four.

V: Overflow flag

If you're doing arithmetic thinking of an 8-bit number as a signed 7-bit number (−128 to 127 again, right?) this is kind of like the Carry flag in ordinary 8-bit arithmetic. If the answer goes outside the range −128 to 127, the V flag is set to 1.

It can also be set from outside by suitable circuitry, and used for totally different purposes.

D: Decimal mode flag

There is another type of arithmetic called *binary coded decimal* (BCD). What this does is represent a decimal number *digit by digit* in hex. For instance:

34125476

is represented by the bytes:

34 12 54 76

But, of course, if you did arithmetic treating this as a valid hex number, you'd get into a terrible mess: for instance, the Carrying rules in BCD are quite different.

Nonetheless, BCD is sometimes used because of its direct relationship to decimal. Many pocket calculators use it, for example. If the D flag is set to 1, the 6510 will treat all arithmetic as if it were BCD arithmetic.

When you switch your computer on, the D flag is automatically reset to Ø. I suggest you leave it that way!

I: Interrupt mask flag

Interrupts are how external devices communicate with the CPU. If the I flag is set to 1, interrupts are disabled (can't happen); if reset to Ø, they are enabled (can happen).

B: Break status flag

This is set to 1 by the BRK (software break) instruction. At that point the CPU stops working and waits for outside help. It's very useful in the organization of the whole computer, but of little interest to us.

COMMANDS THAT AFFECT FLAGS DIRECTLY

There are some commands that let you Clear a flag to Ø or set it to 1, without doing anything else. They are:

CLC	18	**CL**ear **C** flag	SEC	38	**SE**t **C** flag
CLD	D8	**CL**ear **D** flag	SED	F8	**SE**t **D** flag
CLI	58	**CL**ear **I** flag	SEI	78	**SE**t **I** flag
CLV	B8	**CL**ear **V** flag			

All are 1-byte opcodes, implied addressing only.

Now that we know what the flags do, we can take a look at how to use them to control how a program branches: This introduces a new addressing mode, called relative addressing.

11 Branching and Jumps

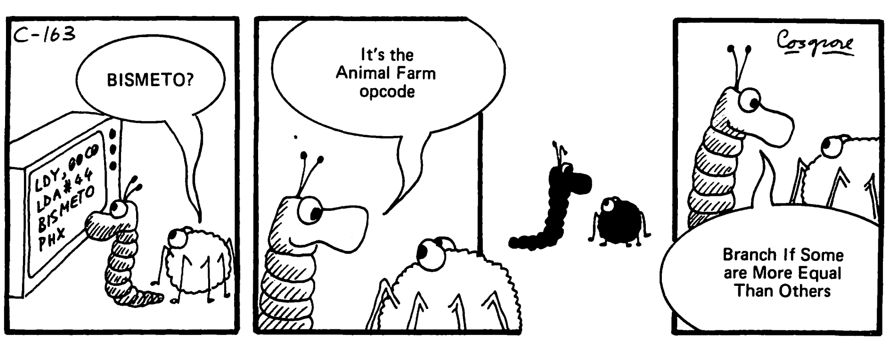
I mentioned earlier that there's a delightfully simple way to make the program jump from one instruction to another. The PC-register (Program Counter; not to be confused with the P-register, which holds the flags) holds the address of the next instruction to be obeyed. By altering that address, you can fool the PC into redirecting the entire flow of calculation to some totally different command.

You can't get at the PC-register directly; but you can produce the required changes by using a whole string of branching commands: BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS. Each of these tells the computer to look at one of the flags, see what it is, and depending on that, to bump the PC up or down by a suitable amount—thereby shifting control to the new instruction.

BEQ

They all work in the same way, so once you've understood one, the others are easy—except for the little matter of flag-handling. I'll start with BEQ because that's especially straightforward. The mnemonic stands for 'Branch if EQual' but what it really does is branch if the Zero flag is set.

It has a 2-byte opcode. The first byte is F0. The second byte is a *displacement*. It is treated as a signed binary number (seven bits plus sign



digit) ranging from −128 to 127. We've met this idea before, but this is the place where we have to face it head on. A positive displacement tells the PC-register 'move so many places ahead' and a negative displacement tells it 'move so many places back'. I'll explain this more carefully after we've seen an example.

Suppose we have two numbers stored in CØØØ and CØØ1, and we want to see whether they are equal. If they are, we'll put the number EE in address CØØ2. If not, we'll put DD there instead. Here's how we do it.

```
        SEC          38
        LDA CØØØ     AD ØØ CØ     Get first number M
        SBC CØØ1     ED Ø1 CØ     Subtract second number N
        BEQ skip     FØ Ø6        Branch to skip if M − N is zero
        LDA #DD      A9 DD      ┐
                                ├ Otherwise store DD
        STA CØØ2     8D Ø2 CØ   ┘
        RTS          6Ø           Back to BASIC
skip:   LDA #EE      A9 EE        If we're here, M − N was zero
        STA CØØ2     8D Ø2 CØ     Store EE instead of DD
        RTS          6Ø           Back to BASIC
```

First let's see how it works. The first few instructions we've seen before. When we get to the BEQ instruction, the computer looks at the *Zero flag*. If this is *set* to 1 (which means that the last operation that changed the flag resulted in a zero—namely the SBC operation) then the Program Counter should be increased by 6. (That's the 2nd byte Ø6 in the opcode—the displacement.)

Suppose we'd started with M and N (in CØØØ and CØØ1) equal: say both were 7B (hex). Then the result of the subtraction would indeed be zero (provided the Carry was cleared, as it was), so this branch would occur.

At the time the BEQ instruction is being thought about, the PC-register has its beady eye on the next instruction in sequence, which is LDA #DD, beginning with the opcode byte A9. Now it gets the message: 'move ahead 6 more bytes'. So it counts down the program from that A9, getting

```
(A9)   DD   8D   Ø2   CØ   6Ø   [A9]
  └─────────────────►──────────────┘
        1    2    3    4    5    6
```

which is the start of the LDA #EE instruction (at the line marked *skip*). So the computer now carries out *that* instruction next. That stores EE in address CØØ2 and then returns to BASIC.

On the other hand, suppose M and N were different, say M = 7B, N = C3. Then the Zero flag would not have been set by the SBC command;

so the BEQ would have told the computer *not* to branch. This would have left the PC pointing to the next command in the list, LDA #DD. Continuing from there, the computer would have stored DD in CØØ2 and then returned to BASIC. (Note that each part of the branch needs its own RTS. It's what the computer actually *carries out*, not what else is floating around in the program listing, that counts. If you missed out the RTS in the 'not equal to Zero' branch, the program would just carry on to *skip* and keep going—and you'd end up with EE in CØØ2, willy-nilly.)

RELATIVE BRANCHING

Now let me say more about the way to calculate the correct displacement byte in a branching command. I've underlined it above: it was Ø6. Why?

Consider how the program bytes go into memory (see Figure 11.1).

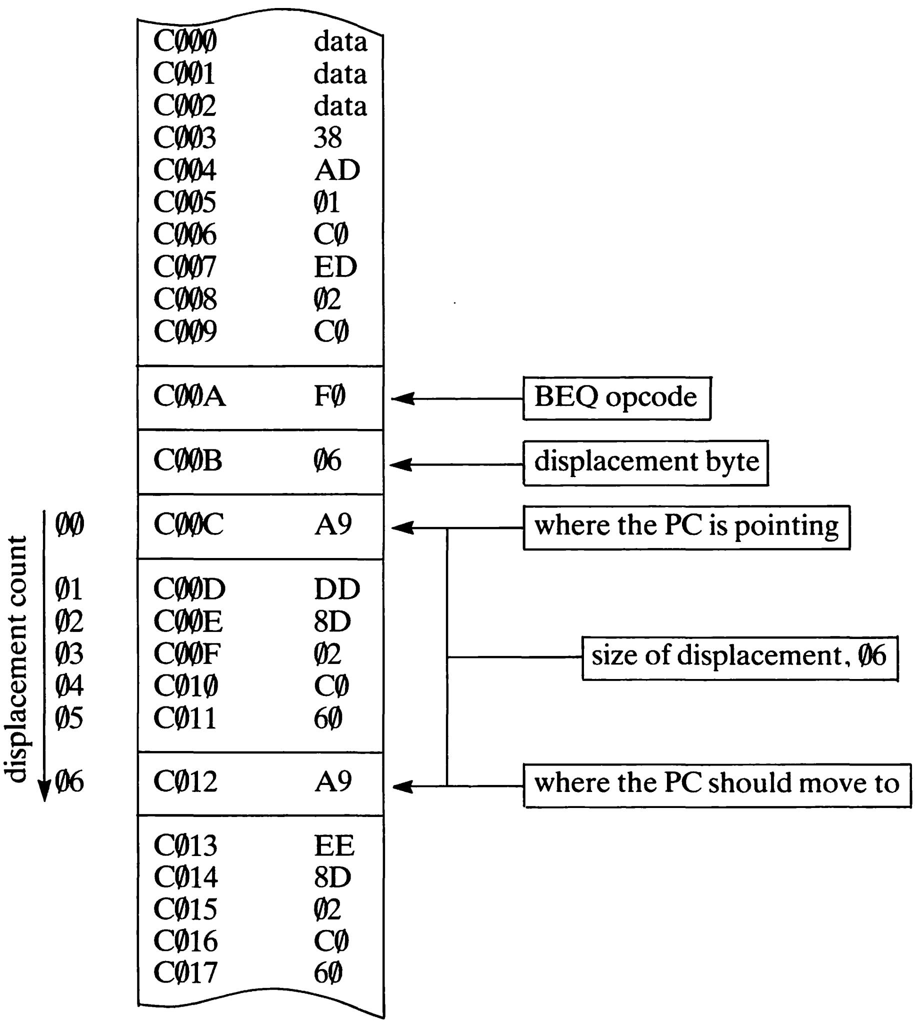


Figure 11.1

That's how it works for a *positive* displacement. For a *negative* one, you do the same thing, still starting from the place the PC would have gone to (the command immediately after the BEQ and the displacement byte), but now you count backwards:

$-1, -2, -3, \ldots$

until you get to the byte that contains the *start* of the opcode you want. However, you still have to take the resulting negative number and convert it into a 2's complement signed binary number, and thence to hex. Fortunately Appendix 1 does this for you.

For instance, to branch 37 bytes backwards, you look up −37 in the Appendix, and get DB. This would be the displacement byte. Figure 11.2 gives a general picture:

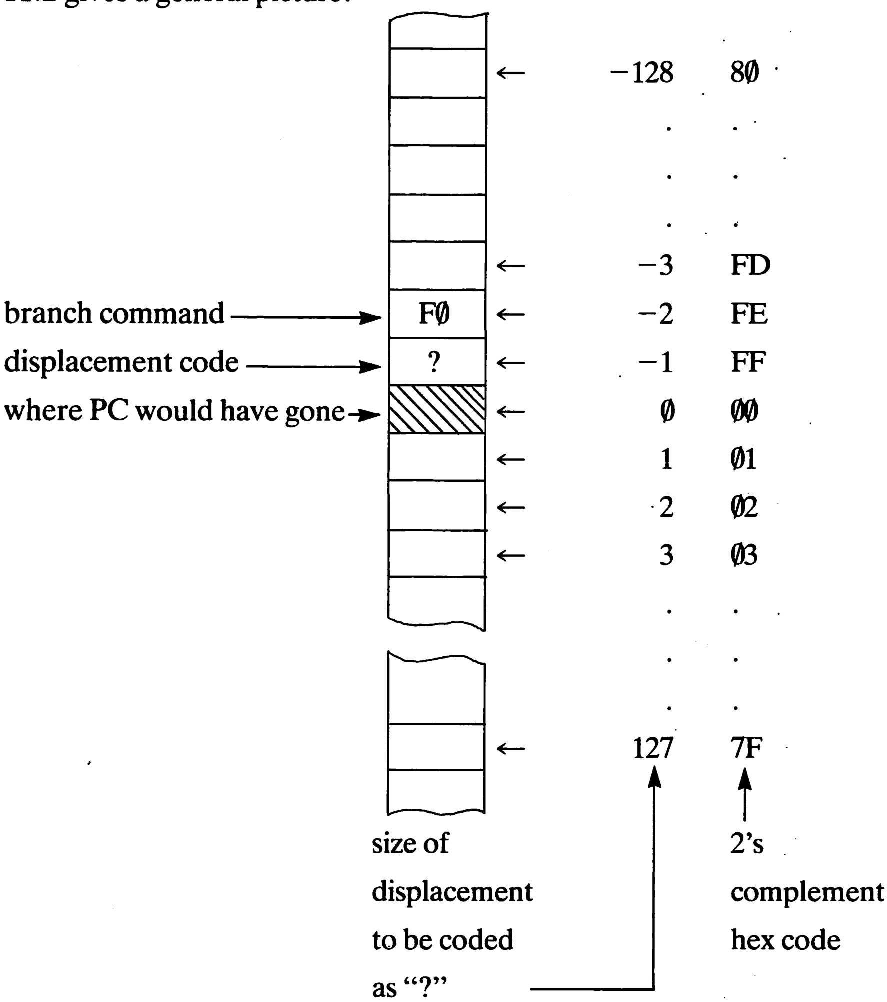


Figure 11.2

As a convention, I'll underline all relative jump displacements in Machine Code listings.

You're all sitting there thinking, 'But what do I do if I want to branch more than 127 bytes?' Basically, you have to do it in several short hops; or you can use the JMP command to be explained below. The answer in practice is that by the time you're writing programs that *need* such big displacements, you won't need me to help you figure it out anyway.

LABELS

When you're writing the program, you don't want to go through all this rigmarole. Instead, what you do is leave a blank where the displacement should go (I prefer an underline since that reminds you a byte is missing) until you've written all the parts of the program. Then go back to that blank, count bytes using the opcodes, and fill it in. (Note that it is the number of bytes, *not* the number of instructions, that goes into the displacement. Since opcodes have different lengths, the simplest solution is to count the actual bytes.)

To identify the instructions that you want to branch to, you use *labels*: short, snappy names like *skip*, *loop5*, and so on. Label the instruction down the left, as shown, and refer to it in the mnemonic at the place where the displacement would go. For instance:

```
        BEQ  skip
        .
        .
        .
skip:   LDA#EE
```

A good assembler will let you use the labels, and compute the relative displacements automatically.

OTHER KINDS OF BRANCH

The other branching commands work in exactly the same way, using relative addressing (the displacement byte). The only difference is which flags they look at and how they react. Here's the full list, with opcodes in brackets:

BCC	(9Ø)	**Branch if Carry Clear:**	if the C flag is Ø
BCS	(BØ)	**Branch if Carry Set:**	if the C flag is 1
BEQ	(FØ)	**Branch if EQual** (to zero):	if the Z flag is 1
BNE	(DØ)	**Branch if Not Equal** (to zero):	if the Z flag is Ø
BMI	(3Ø)	**Branch if MInus:**	if the N flag is 1
BPL	(1Ø)	**Branch if PLus:**	if the N flag is Ø
BVC	(5Ø)	**Branch if oVerflow Clear:**	if the V flag is Ø
BVS	(7Ø)	**Branch if oVerflow Set:**	if the V flag is 1

Each has a 2-byte opcode: the byte shown, plus the displacement. Since they can *only* be used in one addressing mode (relative) no special format is required in their mnemonics.

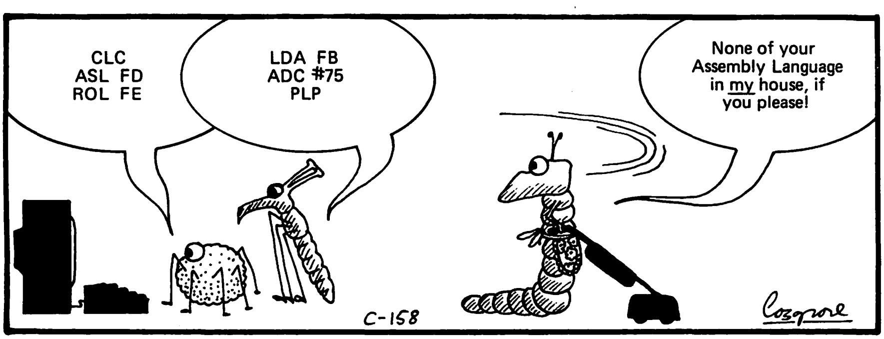
TESTING THE SIGN

As one example of the use of branching, I'll write down a program that will let you test what I said earlier about the N and C flags. The basic idea will be a problem that one often encounters in a program: given two 1-byte unsigned numbers (Ø–255), say M and N, decide whether M > N or not.

I'll put the two numbers M and N in CØØØ and CØØ1 as usual. In CØØ2 I'll put a little flag of my own: Ø if M − N is negative, 1 if M − N is positive. You'll see why in a moment.

Here's the code:

```
       SEC        38
       LDA CØØØ   AD ØØ CØ
       SBC CØØ1   ED Ø1 CØ
       BCC neg    9Ø Ø6         Relative jump displacement
       LDA #1     A9 Ø1
       STA CØØ2   8D Ø2 CØ
       RTS        6Ø
neg:   LDA #Ø     A9 ØØ
       STA CØØ2   8D Ø2 CØ
       RTS        6Ø
```



Load this in, with 3 data bytes; but this time *don't* run it. Break the LOADER program using the S option; and write a BASIC routine that will make it easy to test lots of possibilities:

```
5000 INPUT "M, N"; M, N
5010 PRINT M; "-"; N; "  IS  ";
5020 POKE 49152, M
5030 POKE 49153, N
5040 SYS(49155)
5050 X = PEEK(49154)
5060 IF X = 1 THEN PRINT "POSITIVE"
5070 IF X = 0 THEN PRINT "NEGATIVE"
5080 GOTO 5000
```

This loops indefinitely, and lets you test any pair M, N you like. However, they must be between 0 and 255.

Start with GOTO 5000 and see what happens. Does it make sense? It should do. Note that the machine treats M − M, a zero answer, as being *negative*. So the Carry flag is reset to 0 on a zero answer.

Now let's see what would happen if you did the 'obvious' and used not BCC, but BMI (**B**ranch if **MI**nus). Change the BCC line to:

```
BMI   neg      30  06
```

and use the E option on LOADER to do it (good practice!). If you now try the BASIC routine at 5000, all will appear well when you use numbers like 100 − 68, which the computer does consider to be positive. But try doing 130 − 1. The machine steadfastly insists that this is *negative*. What it's done is treat 130 as a signed number, namely −126. Then −126 − 1 = − 127 which indeed is negative.

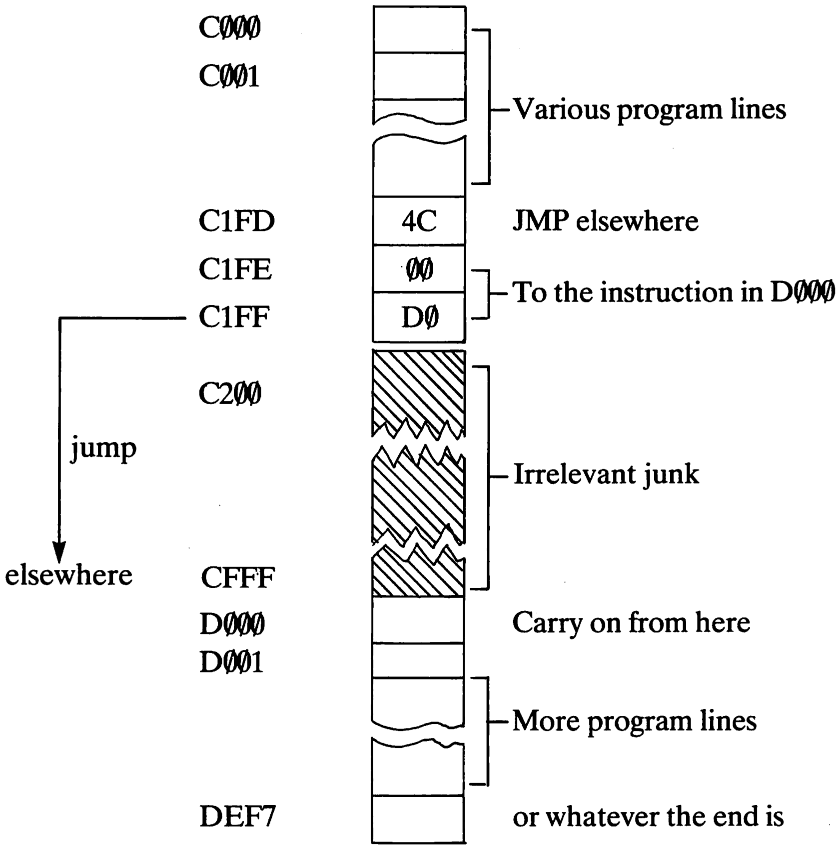
Moral: if you're thinking 0–255, use BCC and BCS, not BMI and BPL.

JUMPS

There's a different way to change the PC-register (and hence move to another instruction), the **J**u**MP** command

JMP   (opcode 4C)

It is normally used in Absolute mode, that is, followed by a 2-byte address. This address is the address of the instruction you want carried out next: it is simply shoved into the PC. For instance, here's a program that uses JMP to hop over an irrelevant area of memory:



In the mnemonic you usually use a label:

JMP elsewhere

but you can also just put the address:

JMP D000

There's one other mode for the JMP instruction—*indirect* mode. See Chapter 14.

Using branch instructions, you can set up the Machine Code equivalent of a BASIC FOR/NEXT loop. As usual, you have to think it through carefully yourself.

12 Looping

Suppose you want the program to carry out a given task several times, and then stop. In BASIC you'd use a FOR/NEXT loop; in Machine Code you have to build one for yourself. If you've ever tried writing a GOTO version of a FOR/NEXT loop (so that you can jump out of the loop without leaving extraneous junk lying around in the machine) you'll have got the idea already. Here are two ways to loop in BASIC:

```
10  FOR K = 1 TO 7
20  PRINT "HELLO"
30  NEXT K
```

```
10  K = 1
20  PRINT "HELLO"
30  K = K + 1
40  IF K < = 7 THEN 20
```

In the second version we use K as a *loop counter*. Each time round the loop, K is *incremented* by 1, until the test at line 40 finally fails, in which case we exit the loop. And this is exactly what you have to do in Machine Code.

A good example, with our present state of knowledge, is a program that will multiply two 8-bit numbers, giving a 16-bit result, by using repeated addition—for example:

$$17 \times 6 = \underbrace{17 + 17 + 17 + 17 + 17 + 17}_{\text{6 times}} = 102 \qquad \text{(decimal)}$$

This is *not* an efficient way to do the job, but it should adequately illustrate how to set up a loop. There are some slick tricks to improve on what we'll end up with, but for now I'd prefer to be simple-minded.

The two numbers to be multiplied will be stored in C000 and C001. The second should be non-zero. The answer will go into C002–C003 as a 2-byte number (junior: senior). I'll need C004 to act as a loop counter; and C002–C003 can store the total as it builds up, as well as the answer.

So that's 5 data bytes. Here's the program:

	LDA #ØØ	A9 ØØ	
	STA CØØ2	8D Ø2 CØ	Set running total to zero
	STA CØØ3	8D Ø3 CØ	
	STA CØØ4	8D Ø4 CØ	Set counter to zero
	INC CØØ1	EE Ø1 CØ	Minor adjustment for correct number of loops
loop:	CLC	18	
	LDA CØØ2	AD Ø2 CØ	
	ADC CØØØ	6D ØØ CØ	
	STA CØØ2	8D Ø2 CØ	Junior byte of running total
	LDA CØØ3	AD Ø3 CØ	
	ADC #ØØ	69 ØØ	Senior byte may have Carry to be added in
	STA CØØ3	8D Ø3 CØ	
	INC CØØ4	EE Ø4 CØ	Increment counter (add 1 to it)
	CLC	18	
	LDA CØØ1	AD Ø1 CØ	Put second number in accumulator
	SBC CØØ4	ED Ø4 CØ	See if counter equals it
	BNE loop	DØ E2	Branch back if not
	RTS	6Ø	Otherwise exit loop

There is a backwards relative branch of −3Ø from BNE loop, underlined. (In signed arithmetic, −3Ø is E2 hex. See Appendix 1.)

INCREMENT AND DECREMENT

You'll have noticed a new instruction:

INC (INCrement)

This can be used to increase the contents of a memory location by 1. Its opcodes are:

EE (Non-zero page absolute)

E6 (Zero-page)

plus two indexed versions (see Chapter 13). There is a corresponding command:

DEC (**DEC**rement)

which subtracts 1, with opcodes:

CE (Non-zero page absolute)

C6 (Zero-page)

Both commands 'wrap around' ignoring any Carries (except that they set suitable flags), so that:

255 + 1 = Ø   Ø − 1 = 255

as far as these operations are concerned.

COMPARE

In the above example we decided when to end the loop by subtracting the counter from the number of loops required and using the result to control a branch. That's not strictly necessary. Some bright spark noticed that what you often want to do is to take a decision based on *what the flags would have been* if you'd carried out the subtraction—*without* actually doing it. So he invented the **CoMP**are instruction:

CMP (opcodes C9, CD, C5 respectively for immediate, absolute, and zero-page addressing)

What this does is to set the flags exactly as if you'd used SBC, but leave the contents of the accumulator intact. This is a very smart idea: you only need the flags to take the decision, and you'd often prefer not to muck up the accumulator.

For example, in the routine above, we can replace the SBC CØØ4 instruction by:

CMP CØØ4   CD Ø4 CØ

It doesn't actually shorten the code in this particular case, but we'll see examples later where it most certainly does. In any case, it's a more civilized approach to the whole game; and in conjunction with indexing (Chapter 13) leads to much more efficient programs.

THE INDEX REGISTERS

Speaking of indexing . . . I haven't yet given you any uses for the index registers. If they're not being used for more esoteric purposes (Chapter 13 again) they're perfect for use as loop counters. They're already there inside the 6510 chip, so there's no need to fiddle about with LDA and STA and suchlike; and there's a whole host of instructions that affect them directly, so that you can bypass the accumulator a lot of the time.

As I mentioned, there are two index registers, X and Y. Each is a 1-byte register. The relevant instructions come in X, Y pairs too. You can look up the opcodes and the addressing modes that are available in Appendix 4; I'll quickly run through the instructions (which are very similar to the ones we've seen before, but using the X or Y registers in place of the accumulator):

LDX, LDY:	**LoaD** a byte into **X** or **Y**
INX, INY:	**IN**crement **X** or **Y**
DEX, DEY:	**DE**crement **X** or **Y**
STX, STY:	**ST**ore contents of **X** or **Y** in memory
CPX, CPY:	**C**om**P**are **X** or **Y** register with a selected byte. This can be thought of as follows: subtract the selected byte from what's in the X-register (or Y-register), set the flags accordingly, and then forget what the result of the subtraction was

The increment, decrement and compare commands are what make these registers into excellent loop counters. Let's rewrite the above program using the X-register as loop counter.

This time we only need four data bytes: CØØØ and CØØ1 for the two numbers M and N to be multiplied; and CØØ2–CØØ3 to store the junior: senior bytes of the answer. It goes like this:

```
       LDA #ØØ     A9 ØØ       Initialize
       STA CØØ2    8D Ø2 CØ
       STA CØØ3    8D Ø3 CØ
       LDX CØØ1    AE Ø1 CØ    Put N into index
loop:  CLC         18
       LDA CØØ2    AD Ø2 CØ
       ADC CØØØ    6D ØØ CØ
       STA CØØ2    8D Ø2 CØ
       LDA CØØ3    AD Ø3 CØ
       ADC #ØØ     69 ØØ
       STA CØØ3    8D Ø3 CØ
```

Very similar so far. But now the denouement comes much more abruptly:

DEX	CA	Decrement index
CPX #ØØ	EØ ØØ	Compare with zero
BNE loop	DØ E9	Branch if not equal
RTS	6Ø	

The E9 in the BNE is the displacement: it's actually −23.

To move systematically through a block of memory, or run through a sequence of addresses in turn, here's the very thing you want:

13 Indexing

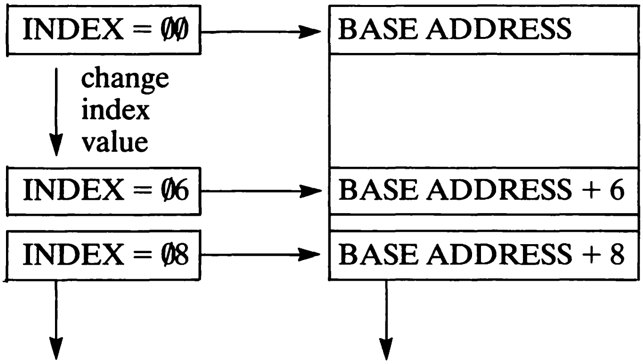
Suppose you want to copy a section of memory into some other section. For example, you may wish to move a Machine Code program from our standard C000 starting-point to some other place, or put a pre-prepared display on the screen while saving the old one. As a specific example, suppose you want to copy the contents of page CE on to page CF. Then you've got to start at CE00, load it into the accumulator, store that into CF00, then repeat on CE01 and CF01, and so on.

This is delightfully repetitive and, therefore, just what a computer ought to be able to do backwards, before breakfast, while standing on its head and whistling, 'The Foggy Foggy Dew'. Unfortunately, this is not the case, with our current repertoire of instructions. We as yet have no way to modify the address part of an opcode.

Well, that's not quite true. We *could* simply store a new byte in memory at the right place in the program—'rewriting' the address byte in the opcode while the program is running. This is called a *self-modifying program*. Those went out with the Ark; and quite right too, because it's hard to debug a program that won't sit still. (I recall the same problem with my Aunt Matilda's poodle.) If running the program changes the listing, then whatever caused the bug may be long gone. The Wonderful Self-Erasing Bug is an absolute pig to deal with, and even Sherlock Holmes would shudder at the very thought.

INDEXED ADDRESSING

In *indexed addressing* you don't actually change the address bytes in the opcode; but you do change their meaning. The idea is to use them to specify a *base address*, where a block of codes starts; and to use an *index* to say how much further the CPU should move in order to reach the address we really want. Again, it's a 1-byte displacement; but this time in the range 0–255 rather than the −128 to 127 for branching. And it goes not in the opcode, but in one of the index registers X or Y. This gives you exactly one page worth of freedom in what you can address (although you can cross page boundaries without trouble—other than a slight slowing down that's only of interest to an electronic engineer). Here's the basic picture:



There are four variants (which for the purposes of this book I'm considering as distinct addressing modes, otherwise Appendix 3 would get all muddled): X-register or Y-register in combination with zero or non-zero page base address. The opcodes are in Appendix 4. In fact a base address on page zero isn't much use to us: it would only allow us access to addresses on pages Ø and 1 which, as I've said already, have been snaffled by the gentlemen who designed the operating system. (Page 1 is, of all things, the *stack* to which the SP-register points.) So the only modes that we really care about are the two non-zero page modes: one for X and one for Y. At this level there's no essential difference between X and Y: mostly I'll use X.

TRANSFERRING A PAGE OF MEMORY

For starters, here's a program that solves the problem we opened with—transfer page CE to CF:

```
       LDX #ØØ       A2 ØØ
loop:  LDA CEØØ, X   BD ØØ CE
       STA CFØØ, X   9D ØØ CF
       INX           E8
       CPX #ØØ       EØ ØØ
       BNE loop      DØ F5          (−11 displacement)
       RTS           6Ø
```

Note the format of the mnemonics using indexed addressing:

LDA (two base address bytes), X

to say we're using the X-register as index. (for zero-page, you use the same thing but omitting the ØØ page number byte from the base address.)

The way the index works is this. The command:

```
LDA  CEØØ, X
```

tells the computer 'add the contents of X to CEØØ and load the accumulator from that address'. So as X runs through Ø, 1, 2, ... the accumulator gets loaded with successive bytes from page CE. Similarly:

```
STA  CFØØ, X
```

tells it 'add the contents of X to CFØØ and store the accumulator at *that* address'.

All indexing works in this way: add the index to the base address to get the actual address that will be used.

One other point to note: the use of CPX #ØØ to test for the end of the loop. The point is that we increment X before testing; so on the first run, X holds 1 at the time it gets tested; then 2, 3, ..., 255. On the final (256th) run through the loop, the INX bumps it up from 255 to 256, but that wraps around to give Ø (the Carry bit falls off) so the loop ends at that stage. The way 1-byte addition wraps around from 255 to Ø is really quite useful sometimes—just don't forget it, or you'll be amazed at what some of your programs will do!

To test this routine, feed it in using LOADER; then take option S. Type in the following BASIC program:

```
5ØØØ  FOR T = Ø TO 255
5Ø1Ø  POKE 52736 + T, T: POKE 52992 + T, 119
5Ø2Ø  NEXT
```

This fills page CE with Ø, 1, 2, ..., 255 in order; and CF with 119 (77 hex). (See below for a Machine Code way to do this.) Then we run the Machine Code:

```
5Ø3Ø  SYS(49152)
```

and then take another look at page CF:

```
5Ø4Ø  FOR T = Ø TO 255
5Ø5Ø  PRINT PEEK(52992 + T),
5Ø6Ø  NEXT
```

Run this using GOTO 5ØØØ. For future reference, notice how long it takes for BASIC to carry out lines 5ØØØ–5Ø2Ø. It's *slow*. You'll find that page CF has switched to Ø, 1, 2, ..., 255, just as I claimed.

Note that, as I remarked obscurely above, you don't have to start indexing at the top of a page. The base address can be any (2-byte) address, so the block you're working with can straddle the boundary between consecutive pages.

FILLING A PAGE WITH DATA

I used BASIC to POKE things to pages CE and CF above (and remarked on how slow it is: about 5–10 seconds in fact). That's because I thought you'd trust BASIC more than my Machine Code. When testing something, it's wise only to use stuff that you *know* is OK. But Machine Code can set up the data in pages CE and CF quickly and easily.

To fill page CF with 77s (hex) we do this:

```
       LDX #ØØ       A2 ØØ
       LDA #77       A9 77
loop:  STA CFØØ, X   9D ØØ CF
       INX           E8
       CPX #ØØ       EØ ØØ
       BNE loop      DØ F8
       RTS           6Ø
```

To fill CE with Ø, 1, 2, ..., 255 we use a variant:

```
       LDX #ØØ       A2 ØØ
loop:  TXA           8A
       STA CEØØ, X   9D ØØ CE
       INX           E8
       CPX #ØØ       EØ ØØ
       BNE loop      DØ F7
       RTS           6Ø
```

Note the new instruction:

TXA	(opcode 8A)	**Transfer X-register to Accumulator**

which copies X into A (but leaves X intact). There are some similar instructions involving the X, Y, and A registers:

TAX	(opcode AA)	**Transfer Accumulator to X-register**
TYA	(opcode 98)	**Transfer Y-register to Accumulator**
TAY	(opcode A8)	**Transfer Accumulator to Y-register**

You *can't* do the obvious and replace:

```
       TXA
       STA CEØØ, X
```

by

STX CEØØ, X

because there ain't no such animal. You can't use X-indexed mode to play games with X itself. (The prospect of the X-register eating its own tail is such a frightening one that the 6510 refuses to contemplate it.)

SEVERAL PAGES

If you want to transfer (or set up) more than 256 bytes of data, you have three alternatives:

1. Cheat. If it's only two or three pages, repeat the program two or three times in a row with different page numbers.
2. Write a self-modifying program in which the program byte that holds the page number is changed by an STA instruction, and loop it. Now I *don't* recommend this, but there may be occasions when it's worth doing, even if it is considered bad style.

Suppose you want to fill pages C4 to CF with the byte 77. (This is typical—though not with byte 77—of various 'initialization' routines, and we'll encounter a similar problem in Chapter 20.) That's twelve pages of memory. Here's a self-modifying routine to do it, using the program above to set up each page, and another loop controlled by the Y-register to deal with the twelve pages. The program starts at CØØØ, the standard space:

```
       LDY #C4         AØ C4
       LDX #ØØ         A2 ØØ
       LDA #77         A9 77
loop:  STA C4ØØ, X     9D ØØ C4      ←pagebyte
       INX             E8
       CPX #ØØ         EØ ØØ
       BNE loop        DØ F8
       INY             C8
       STY pagebyte    8C Ø8 CØ          (see below)
       CPY #DØ         CØ DØ
       BNE loop        DØ FØ
       RTS             6Ø
```

By counting from CØØØ we see that the page byte we want to modify is stored in address CØØ8. Hence the double-underlined part of the op-code. The BNE displacements are −8 and −16 respectively.

If you've got this far you'll find it easy enough to run this and test it (using a suitable BASIC program to look at the memory area between pages C4 and CF, namely 5Ø176–53247 in decimal). So I'll leave that as an exercise for you.

To drive home the point about self-modifying programs, however, try changing the first line of the routine to the erroneous:

```
        LDY #BF        AØ BF
```

and then pretend you don't know this mistake has occurred.

Now run it. It doesn't work, of course. So you want to list out your program ready for a debugging session. What has happened to the offending byte?

Well, first, you may have some trouble getting the program to break when it gets stuck. And then you'll find that the bug has overwritten its *own program area* with 77s!

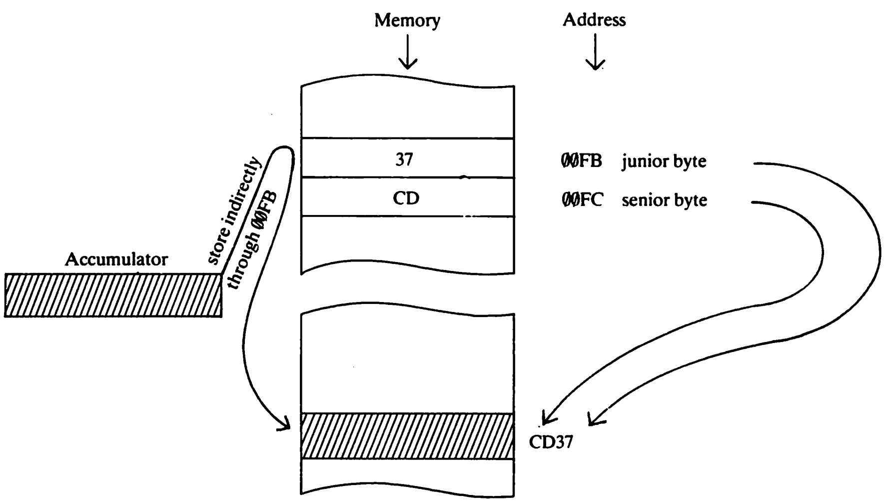
3. There's a third (and much better) way, using 'Indexed Indirect Addressing'. That's what the next chapter is about.

The final group of addressing modes lets you leave a message at one address saying 'don't use this address, use the one 'I'm going to tell you now'. It's the treasure-hunt principle applied to addressing.

14 Indirection

In a treasure-hunt, you have to follow a series of clues. Each clue tells you where to find the next clue. Indirect addressing is much the same—but the process is only carried out once. You specify an address in the opcode. This is *not* the address to be used in the final operation; it is *the place where that address may be found.*

For example, suppose addresses ØØFB and ØØFC contain the bytes 37 and CD respectively. The command 'store the accumulator indirectly through ØØFB' would have the same effect as 'store the accumulator in address CD37'. The address in the opcode, ØØFB, contains the junior byte of the actual address; and the next byte, ØØFC, contains the senior byte of the actual address. Like this:

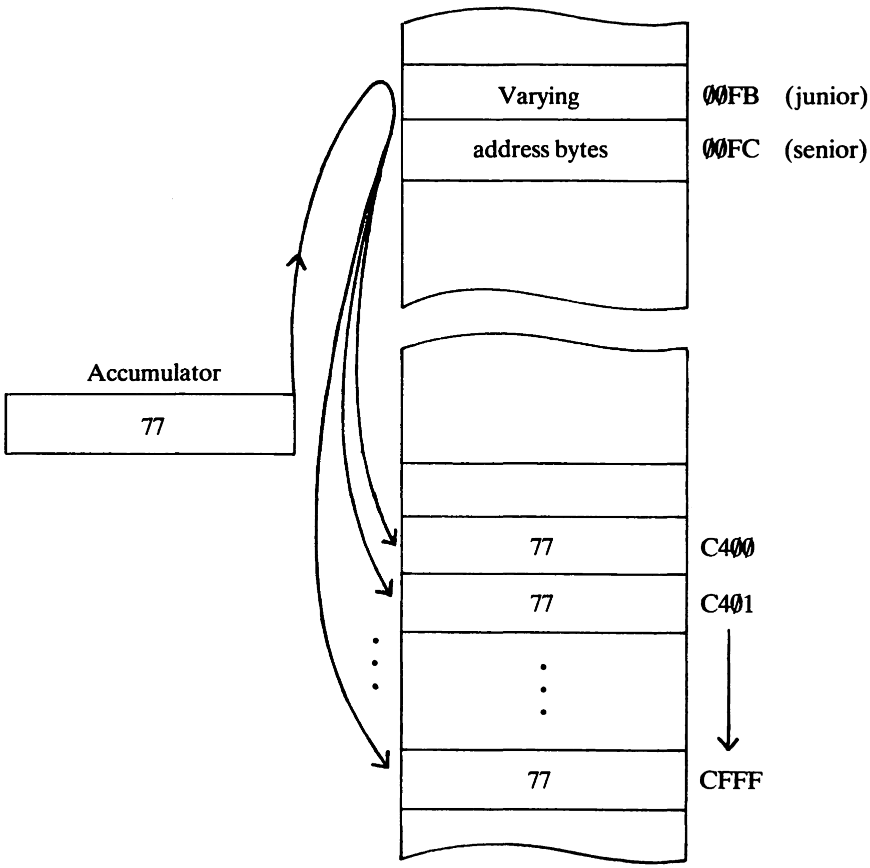


Why bother to go through this rigmarole, when you can perfectly well store the accumulator in CD37 directly? The answer is that we can easily put instructions in the program that change the contents of ØØFB–ØØFC, and hence change the address that we store stuff in. However, we don't have the danger of a self-modifying program, because ØØFB–ØØFC are part of the data area, not the program area. (They're not in the usual place we put data, but there's a good reason for that: see below.) So, from within the program we can redirect the contents of the accumulator to any place we wish in the entire 64K of RAM.

If you think it's a bit complicated, you'll no doubt be pleased to hear that this is the *simple* version of what's going on. There's also an indexed version—in fact, the indexing can be done in two different ways! But to begin with, I'll eliminate the role of the index by setting the relevant register to zero. Only when we've seen what pure unadulterated indirection can do, will I add indexing too.

FILLING SEVERAL PAGES WITH DATA

Let's go back to the problem of filling pages C4 to CF with the byte 77. The idea is to load 77 into the accumulator, and store it indirectly through ØØFB–ØØFC. A loop will make the contents of these two bytes run through the desired addresses C4ØØ–CFFF in turn.



It's no coincidence that I've used a zero-page *intermediate address* ØØFB–ØØFC. I had to. Indirection is only available through a zero-page intermediary. The bytes that specify the final destination *must* go in page zero. (That does *not* mean the final destination itself has to be on page zero: just the 'clue' that tells you where it really is.) As we saw earlier, this means that the only safe intermediaries are ØØFB–ØØFE, so kindly left for us by the designers of the operating system. (Thank you, Sirs and/or Ma'ams—we really *do* appreciate the thought.)

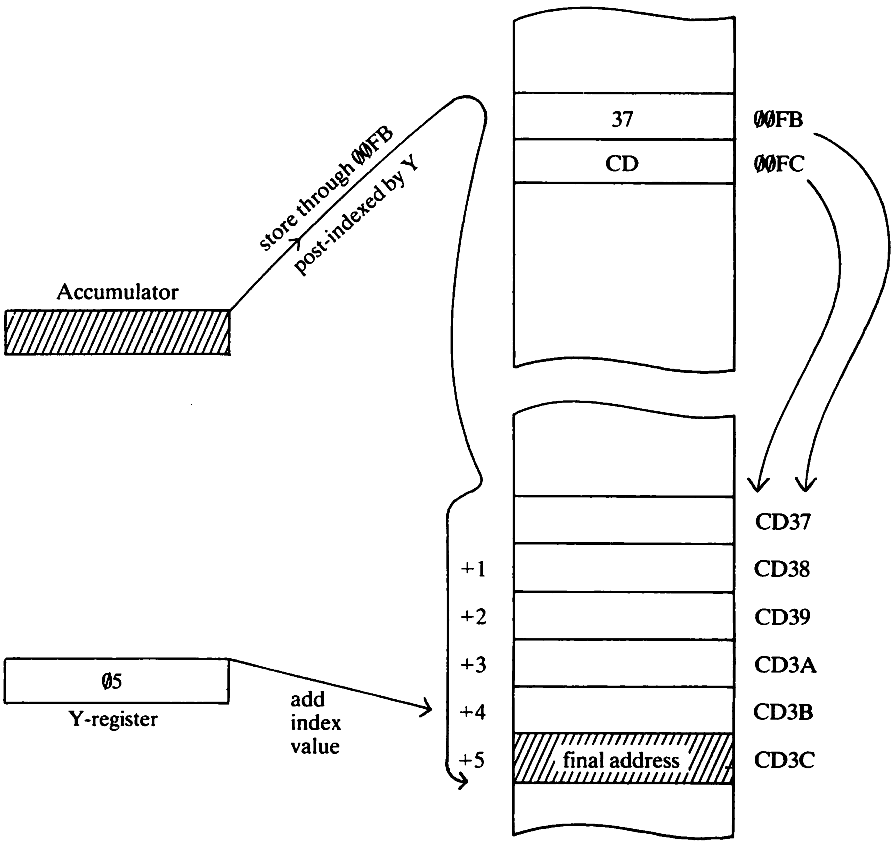
Here's the code:

```
       LDA #ØØ        A9 ØØ
       STA FB         85 FB      set indirect address to
       LDA #C4        A9 C4      bottom of area
       STA FC         85 FC
       LDY #ØØ        AØ ØØ      ignore indexing feature
loop:  LDA #77        A9 77      indirection through
       STA (FB), Y    91 FB      ØØFB ignore role of Y
       CLC            18         here
       INC FB         E6 FB
       LDA FB         A5 FB      increment 2-byte
       CMP #ØØ        C9 ØØ      intermediate address
       BNE skip       DØ Ø2
       INC FC         E6 FC
skip:  LDA #ØØ        A9 ØØ
       CMP FB         C5 FB      test junior byte
       BNE loop       DØ EB
       LDA #DØ        A9 DØ
       CMP FC         C5 FC      test senior byte
       BNE loop       DØ E5
       RTS            6Ø
```

Note that we test for an end at DØØØ, rather than CFFF, because the address is incremented *before* the test. So the loop deals with CFFF increments to get DØØØ, and then stops. It's very easy to loop once too few or once too many, so it's wise to check if you can, and think out just what happens at the start and the end.

POST-INDEXED INDIRECTION

The Y attached to the opcode STA (FB), Y above means that we're using what's called *post-indexed indirection*. That means that the contents of the Y-register are added to the 'destination' address specified by the intermediary. For instance, with the above addresses, and Ø5 in the Y-register, everything looks like this:



By using this we can get a more efficient program: leave ØØFB at zero, and increment the Y-register instead to run through a page; *then* increment the page number in ØØFC. The resulting code is:

```
        LDA #ØØ       A9 ØØ
        STA FB        85 FB
        LDA #C4       A9 C4
        STA FC        85 FC
loop1:  LDA #77       A9 77
        LDY #ØØ       AØ ØØ
loop2:  STA (FB), Y   91 FB
        INY           C8
        CPY #ØØ       CØ ØØ
        BNE loop2     DØ F9
```

```
INC FC        E6 FC
LDA FC        A5 FC
CMP #DØ       C9 DØ
BNE loop1     DØ ED
RTS           6Ø
```

This is ten bytes shorter, a 25% reduction in length.

Note the format of the opcode:

```
STA(FB), Y<-------------- indexed by Y
   ^ ^^
   | ||________________ comma
   | |_________________ brackets for indirection
   |___________________ one-byte (page Ø) address
```

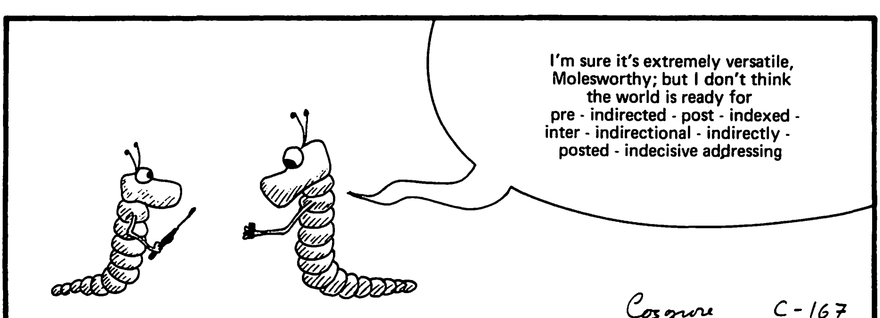
This amazing process is known as *post*-indexed indirection because the indirection is done first, and then the index is added to the destination address. There is also:

PRE-INDEXED INDIRECTION

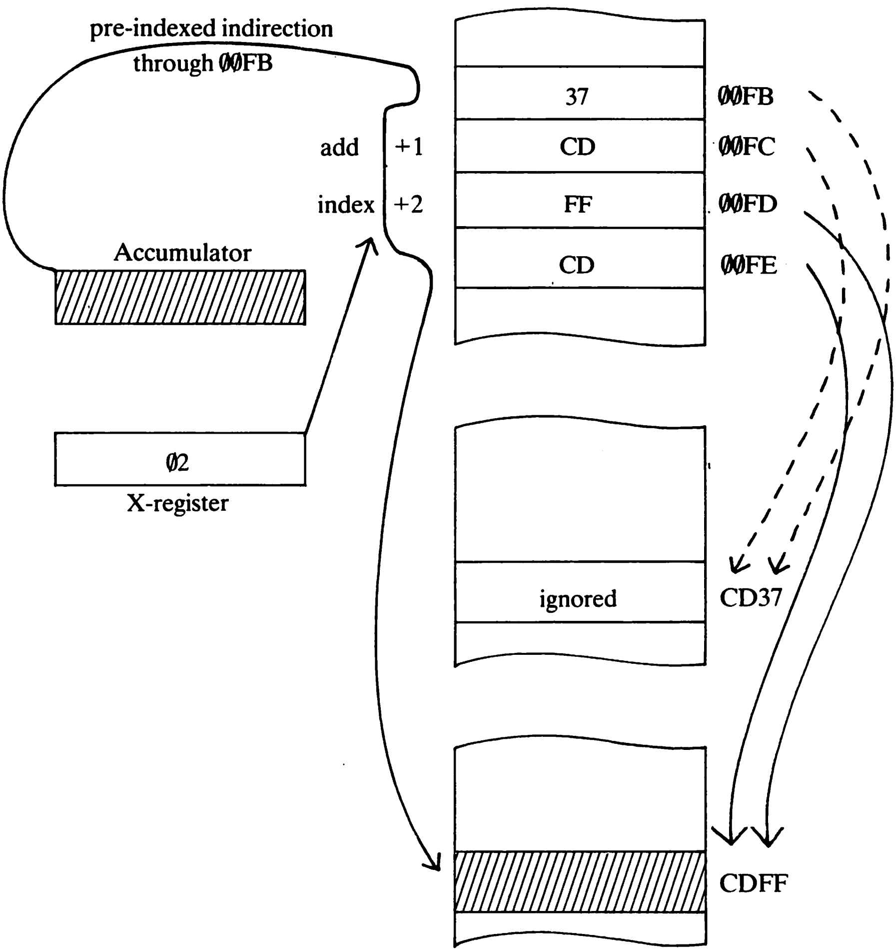
This uses the X-register, and the index is added to the address byte *before* the indirection is done—that is, to the intermediate address specified in the opcode. So the position of the intermediary ticks up one space if the X-register index is incremented. The opcode format reminds us that it is the X-register and that it is added to the intermediate address—for example:

STA (FB, X)

I'd love to make a big song and dance about how wonderful pre-indexed indirection is (because it's a smart idea with a lot of clever uses). But . . . usual snag, this time compounded. Since the *position* of the intermediate address moves around in page zero, we need plenty of spare space on that page to put our indirect addresses into. All we have is



four miserable bytes. So the only time it's worth using pre-indexed indirection is if we're doing pure indirection on one address (set index to zero and leave it there); or we want to hop around between two possible alternatives (flip index from 00 to 02 and back again—02 because they're two-byte addresses). In consequence, I'll say no more about it.



STRAIGHT INDIRECTION

The final addressing mode that uses indirection is available *only* on the JMP instruction, and it's pure-and-simple indirection with no fancy indices. For instance:

```
JMP (C0DE)
```

means 'look at address C0DE and take that as the junior byte of a two-byte address; use the next byte C0DF as the senior byte; jump to the address so formed'.

It's the Machine Code equivalent to BASIC's ON...GOTO, for those who've encountered this. The idea is to store a list of possible addresses you'd like to jump to; shovel the right one into CØDE–CØDF, and use that to direct the jump. I won't give an example here; but you might consider a routine that has to branch seven different ways depending on whether address CØØØ holds the number 1, 2, 3, 4, 5, 6 or 7; with the actual addresses stored in the next 14 bytes (in junior: senior pairs).

A subroutine is a jump with a variable return address—it remembers where it jumped from. The details are controlled by the machine stack, which can also be used for temporary storage.

15 Stacks and Subroutines

In BASIC, you can use subroutines to structure programs into nice, manageable chunks. This makes them easier to write, and easier to debug. Let's remind ourselves how they work. The subroutine is *called* using the command GOSUB followed by its line number in BASIC. The effect is just like a GOTO, except that the machine 'remembers' the line number that it jumped from. At the end of the subroutine, the command RETURN tells it to jump back, to the line immediately following the one it came from. So the same subroutine can be called from different places, and all the returns will be handled correctly. Moreover, you can call a subroutine from within another subroutine. Indeed a subroutine may call itself, a technique known as *recursive programming*.

It's much the same in Machine Code, but using the actual addresses of the instructions in place of their line numbers (because they don't have any). The analogue of GOSUB is:

JSR (opcode 2Ø) **J**ump to **S**ub**R**outine

which must be followed by a 2-byte absolute address—the address to be jumped to. The analogue of RETURN is:

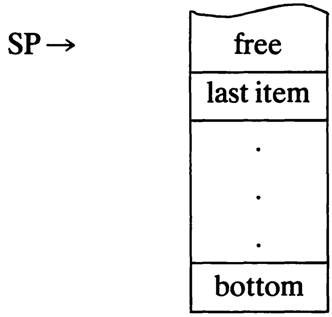
RTS (opcode 6Ø) **R**e**T**urn from **S**ubroutine

which we've seen already: it's the mandatory 'return to BASIC' ending of all our Machine Code routines. (In fact the Sixty-four treats our Machine Code routine as if it were a subroutine in its enormous BASIC operating system program, which is why we have to return in this way.)

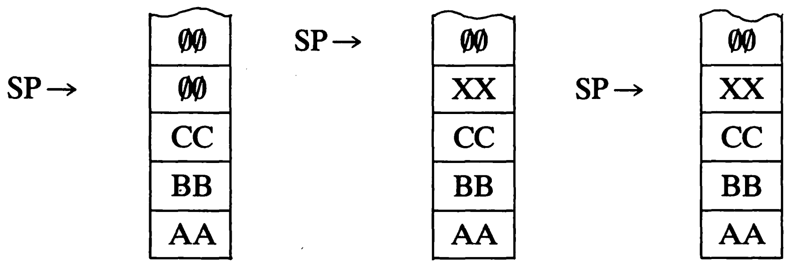
THE MACHINE STACK

How do these work? The 'jump' part is handled in much the same way as an ordinary jump, JMP: the new address is inserted into the two bytes of the PC-register, fooling the 6510 into looking at a different area of program memory. But if that were all that was happening, JSR would be the same as JMP. So JSR performs a second function: it stores the address of the instruction that follows the JSR, so that when an RTS is encountered, this address can be recovered from storage and popped back into the PC to continue the main program where it left off. This is done by using a *stack*.

A stack is a segment of memory with a fixed 'bottom' and a variable top. (In the Sixty-four, the machine stack is always page 1.) The stack pointer, or SP-register, holds the address of the top; it is called a pointer because that's what it does, like this:



Extra items can be pushed on to the stack by moving SP up one and putting the new item in memory; and they can be pulled off the stack by reducing the SP by one. (It's not actually necessary to delete the pulled item from memory: the stack routines ignore anything above the SP.) For example:



original push XX pull

In fact the SP points to the first *unused* location.

PUSH AND PULL

Although the JSR instruction takes care of all this pushing and pulling for you, there are some commands that let you deal with the stack directly. They're quite useful, too: you can push something you want to remember temporarily, then pull it when you need it. The only thing to watch out for is that you haven't pushed something else on top! The instructions are:

PHA	(opcode 48)	**PusH** Accumulator on to stack
PLA	(opcode 68)	**PulL** Accumulator from stack

which store and recall the accumulator contents; and:

PHP (opcode Ø8) **PusH P**-register on to stack

PLP (opcode 28) **PulL P**-register from stack

which do the same for all the flags.

Stacks work on the principle 'last in, first out'. Imagine a pile of books on a desk. PHA means 'add a book to the top of the pile'; and PLA means (in effect) 'take a book off the top'. So if you PHA three items in turn:

PHA 'Robinson Crusoe'

PHA 'Ulysses'

PHA 'Gorky Park'

then to get them off in the right order, you need:

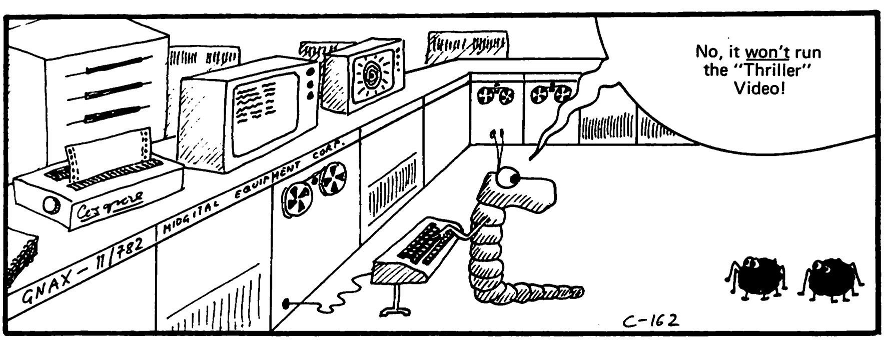
PLA 'Gorky Park'

PLA 'Ulysses'

PLA 'Robinson Crusoe'

HOW A SUBROUTINE USES THE STACK

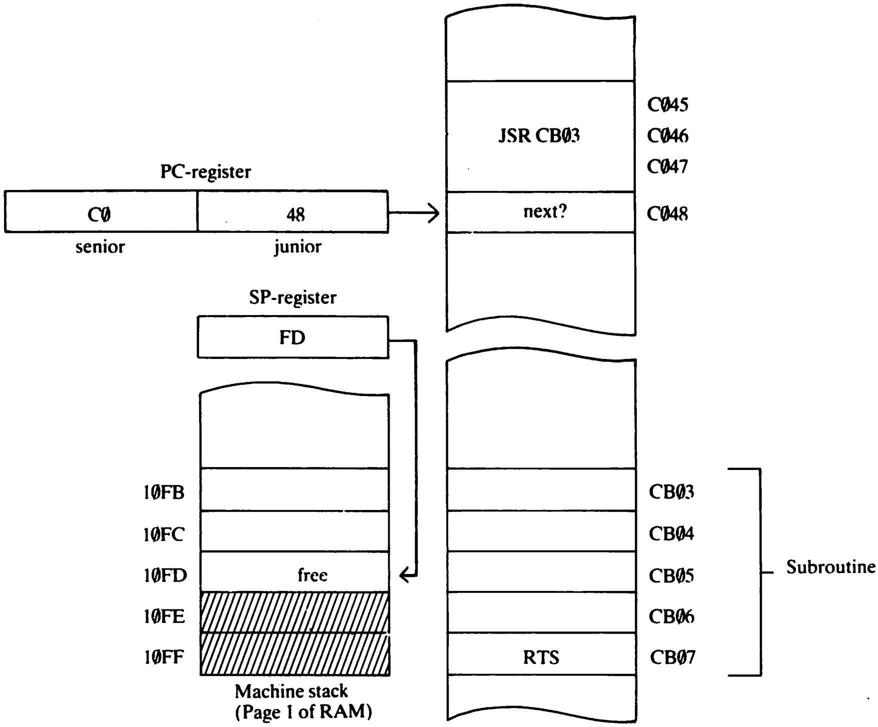
Subroutines push their return addresses on to the stack, and pull them off when an RTS is encountered. Because addresses occupy two bytes, they push and pull in two-byte chunks. (The senior byte is pushed first and pulled last, but we're not likely to care either way.) However, if you've been using the stack during a subroutine, make sure that you've pulled off everything that was pushed on, otherwise you'll return to the wrong address. This also applies to the final 'return to BASIC': *don't leave junk on the stack.*



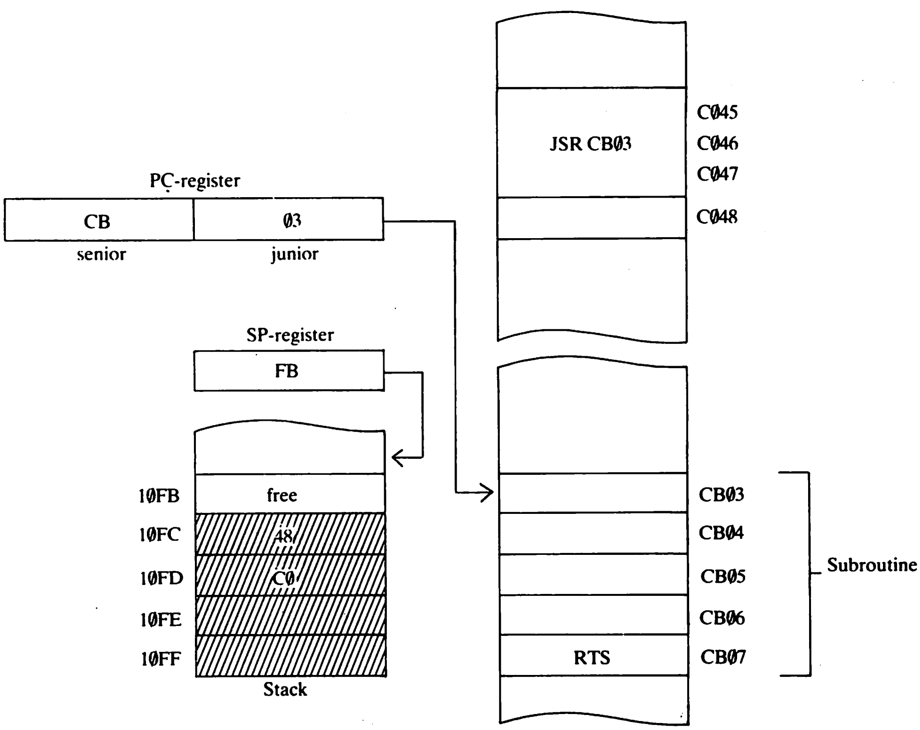
Here's an example. The CPU has just read the instruction:

JSR CBØ3

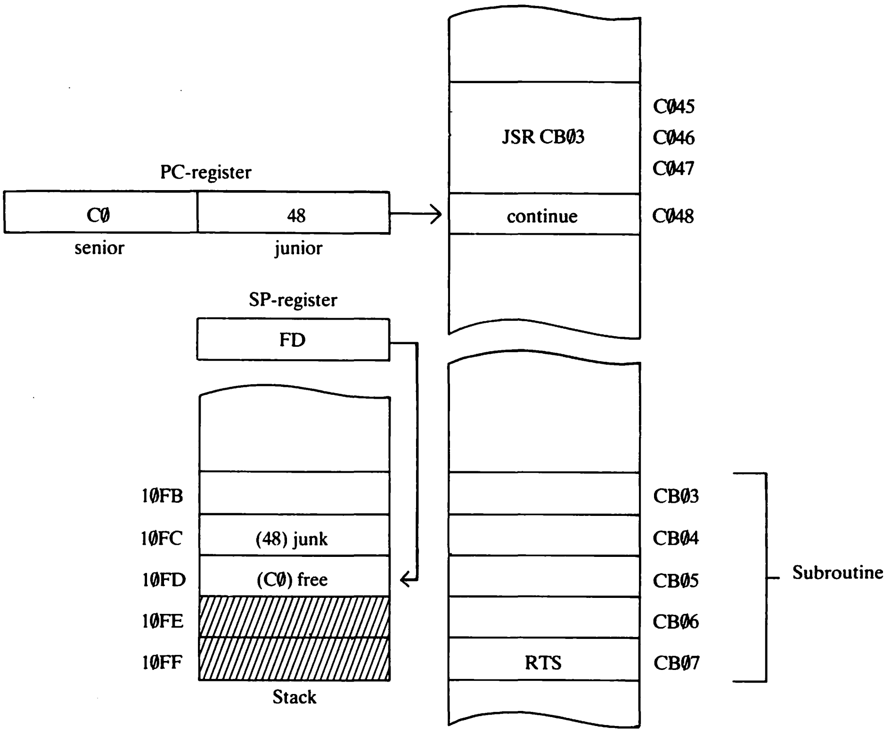
and has moved its PC on to the next instruction at CØ48:



Now it takes the two bytes from the PC and pushes them on to the stack; moves the SP to the new 'top' address; and places CBØ3 in the PC, to make the program jump to the subroutine:



The CPU then steps through the subroutine until it reaches the RTS. At this point, the PC is pulled off the stack (resetting the SP again) and control is back inside the main program:



I repeat that *all of this is done automatically*. But you ought to find it easier to understand how to use subroutines, and what can go wrong if you tinker with the stack, if you know exactly what's going on.

AN EXAMPLE

As an example of the use of subroutines, I'll write a routine that goes through a page of memory, and replaces all bytes that are not within a certain range (say 48–57 decimal, the ASCII codes for the digits ∅–9), by a specified byte (say 32, ASCII for 'space'). We'll have four data bytes:

C∅∅∅	Page number to be used
C∅∅1	Byte to be placed if out of range
C∅∅2	Bottom of range
C∅∅3	Top of range, plus 1

The subroutine will carry out the task 'replace the byte by 32'. It turns out (after writing the code) that this will start at address CØ29. The main program starts at CØØ4:

```
        LDA CØØØ      AD ØØ CØ
        STA FC        85 FC       load start of page
        LDA #ØØ       A9 ØØ       into ØØFB–ØØFC
                                  ready for
        STA FB        85 FB       indirection
        LDY #ØØ       AØ ØØ
loop:   SEC           38
        LDA (FB), Y   B1 FB
        CMP CØØ2      CD Ø2 CØ    see if byte below range
        BCS skip1     BØ Ø3
        JSR change    2Ø 29 CØ    subroutine: calculate
                                  address later
skip1:  SEC           38
        CMP CØØ3      CD Ø3 CØ    See if byte above range
        BCC skip2     9Ø Ø3
        JSR change    2Ø 29 CØ    Second use of
                                  subroutine
skip2:  INY           C8
        CPY #ØØ       CØ ØØ
        BNE loop      DØ E7       Relative jump by −25
        RTS           6Ø          Back to BASIC
```

When you're actually writing this, you don't know what the subroutine address will be, so you can't fill it in in the JSRs. Put two underlines, and fill them in later. (Use two so that the BNE displacement count is easy to make correctly.) Now we count up, and find that the next free address is CØ29. So we write the subroutine:

```
change:  LDA CØØ1      AD Ø1 CØ
         STA (FB), Y   91 FB      Indirection
         RTS           6Ø         Back to main program
```

To test this out, write a BASIC routine to fill page CF with random bytes. Then load it with data bytes:

CF 32 48 58

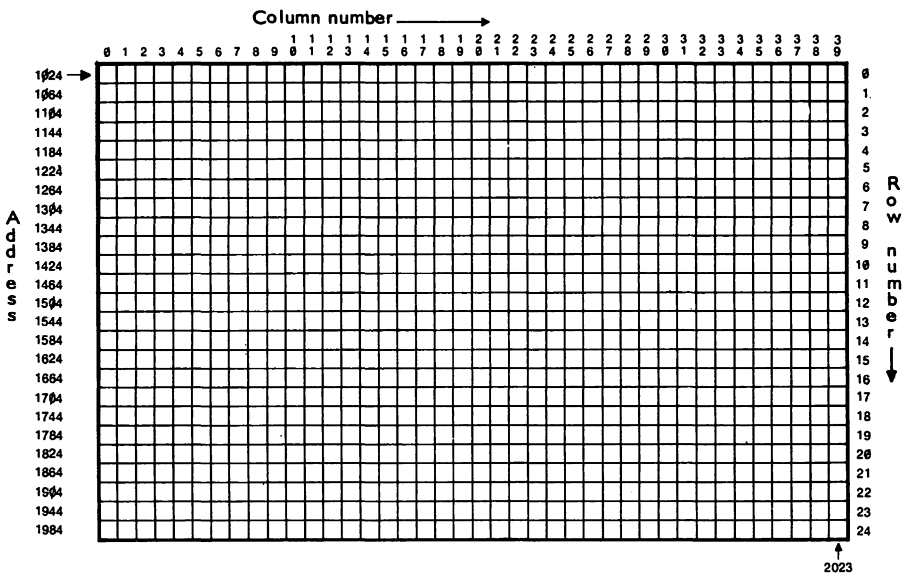
and run it. Check that only bytes in the range 48–57 are left: all others have become 32.

A more dramatic way to use this routine will appear in the next chapter, on the screen display. You'll be able to *see* the bytes change!

The display that you see on your monitor is produced using information stored in two areas of memory. By changing the contents of these areas, you can play tricks with the screen.

16 Screen and Colour Control

If you've read *Easy Programming*, Chapter 19 you'll know most of what's needed as regards the organization of the Screen and Colour Memory areas; but in case you haven't, I'll remind us all here. The screen display consists of 25 rows, each holding 40 characters. The rows are numbered 0–24, and the columns 0–39. That makes 1000 characters altogether:



SCREEN MEMORY

The memory area that specifies the characters is called the *Screen Memory* or *Video RAM*, and it runs from address 1024 to 2023 decimal (0400–07E7 hex). Since the computer's memory has no natural rectangular structure, everything is laid out in order as a single long line of addresses. The addresses run along the rows, and move down a row only when the row ends, going back to the first column just as you do when you read a book. So the hex addresses for Screen Memory correspond to these positions on the screen:

0400	0401	0402	...	...	...	...	0427
0428	0429	042A	...	...	...	...	044F
...	...	...	...	...	...	...	...
07C0	07C1	07C2	...	...	...	...	07E7

In general, the address for row R, column C, is (in decimal):

$$1024 + 40 * R + C$$

and to display a given character at this position we need only store the correct byte in this address.

The code required is *not* ASCII: it is the code listed in Appendix E of the *Manual*, page 132. With a few exceptions this is ASCII minus 64 for the alphabet, ASCII minus 32 for graphics, and plain old ASCII (how uninventive!) for the digits 0–9.

OK, let's give it a whirl. To display a round ball graphics character in row 12, column 20, we first calculate the address. It is:

$$1024 + 40 * 12 + 20 = 1524 \qquad \text{(05F4 hex)}$$

The code for a round ball is 81 according to Appendix E of the *Manual*, which is 51 hex. So we should use the following Machine Code routine:

```
LDA #51      A9 51
STA 05F4     8D F4 05
RTS          60
```

Load this, but instead of running it, use a BASIC routine:

```
5000 PRINT CHR$(147)
5010 SYS(49152)
5020 GOTO 5020
```

This starts us out with a nice clear screen, and avoids messy error messages until we break. Try it out, and check that it works. (On some early versions of the Sixty-four's ROM, it appears not to; but if you change the background colour by POKE 53281, 7 you'll see the ball. It just got printed in the same colour as the background.)

LINES OF CHARACTERS

Try out this routine, in the same way:

	LDA #Ø4	A9 Ø4
	STA FC	85 FC
	LDA #7B	A9 7B
	STA FB	85 FB
	LDX #12	A2 12
	LDY #ØØ	AØ ØØ
loop:	LDA #51	A9 51
	STA (FB), Y	91 FB
	CLC	18
	LDA FB	A5 FB
	ADC #28	69 28
	STA FB	85 FB
	LDA FC	A5 FC
	ADC #ØØ	69 ØØ
	STA FC	85 FC
	DEX	CA
	CPX #ØØ	EØ ØØ
	BNE loop	DØ <u>EA</u>
	RTS	6Ø

This runs through a loop, with X as loop counter, and uses indirection to store the byte 51 in a series of addresses that are 28 hex apart—that is, 4Ø decimal. In other words, it increases the row number but leaves the column fixed. The result is a vertical line of blobs. The start address is Ø47B, which is row 3 column 3.

If you change the ADC #28 to:

ADC #29 69 29

you'll get a diagonal row, because 29 hex is 41 decimal which adds 4Ø (1 to row number) plus 1 (1 to column number). If instead you try:

ADC #Ø1 69 Ø1

you get a horizontal line. To get a diagonal line going downwards to the left, you might expect to use:

ADC #27 69 27

Try it. Does it work? Well, sort of. What's the problem? Wrap-around!

COLOUR MEMORY

The screen colours are held in *Colour Memory* or *Colour RAM*. This is just like Screen Memory as regards its structure; but it runs from 55296 to 56295 decimal (D000–D3FF hex). The colour codes are the usual ones on the Sixty-four:

Black	00
White	01
Red	02
Cyan	03
Purple	04
Green	05
Blue	06
Yellow	07
Orange	08
Brown	09
Light red	0A
Dark grey	0B
Medium grey	0C
Light green	0D
Light blue	0E
Light grey	0F

The codes in Colour RAM give the foreground (ink) colour. To set the background and border colours you store the corresponding bytes in addresses 53281, 53280 respectively (D021, D020).

You can adapt the routine above so that it makes colour changes instead of printing blobs. Change LDA #04 to:

```
LDA #D0     A9 D0
```

and LDA #51 to:

```
LDA #07     A9 07
```

for yellow characters. To make the result show up, use this BASIC program:

```
5000  PRINT CHR$(147)
5010  FOR T = 1 TO 24
5020  PRINT "****************************************";
5030  NEXT
5040  SYS(49152)
5050  GOTO 5050
```

HOW YOUR APPETITE WAS WHETTED

We can now go back and take a look at the routine that I used to introduce Machine Code in Chapter 1. The first step is to *disassemble* it: translate from hex into mnemonics. You'll find Appendix 6 very useful for this: it lists all the opcodes in numerical order, with their addressing modes. The result is:

```
        LDX #00       A2 00
        LDA C0 00     AD 00 C0
store:  STA 0400, X   9D 00 04
        INX           E8
        CPX #00       E0 00
        BEQ end       F0 03
        JMP store     4C 06 C0
  end:  RTS           60
```

There is one data byte at C000 which starts out at 00 but is modified by POKEs later.

What this routine does is to fill page 04 with the byte specified in C000. Now page 04 is the start of Screen Memory; so you see a block of screen change to a single character. The rest of the BASIC makes random changes to the byte concerned every time you hit a key; and uses POKEs to alter the page number—first through Screen Memory, then through Colour Memory. Notice how quickly this simple Machine Code program achieves these effects.

DIGIT SIEVE

Now, as promised in the previous chapter, I'll write a routine that uses the screen to display the effect of a program that runs through a block of memory (here the Screen RAM) changing all bytes outside a selected range to a specific byte. You may like to guess just what its effect will be, before you try it out!

There are four data bytes in C000–C003: the page number (04), the byte to be inserted (20), the lower limit of bytes not to be changed (30), and the upper limit (3A). The code is:

```
LDA C0 00     AD 00 C0
STA FC        85 FC
LDA #00       A9 00
STA FB        85 FB
```

```
        LDY #00          A0 00
loop:   SEC              38
        LDA (FB), Y      B1 FB
        CMP C002         CD 02 C0
        BCS skip1        B0 03
        JSR change       20 31 C0
skip1:  SEC              38
        CMP C003         CD 03 C0
        BCC skip2        90 03
        JSR change       20 31 C0
skip2:  INY              C8
        CPY #00          C0 00
        BNE loop         D0 E7
        INC FC           E6 FC
        LDA FC           A5 FC
        CMP #08          C9 08
        BNE loop         D0 DF
        RTS              60
change: LDA C001         AD 01 C0
        STA (FB), Y      91 FB
        RTS              60
```

This is just like the example at the end of the previous chapter, except that instead of going through a single page, it goes through pages 04–07. That's the Screen Memory, *plus* a 'harmless' area from 07E8 to 07FF which includes the sprite data pointers. So, provided we're not using sprites, no trouble arises. If we are, then the test for the end of the loop has to be modified and is a little more complicated (test FB *and* FC).

What this routine does is eliminate from the screen display any character that is not a digit 0, 1, ..., 9. That's because it replaces any character code not in the range 48–57 (decimal) by a space (32 decimal). To see it in action, we need to set up an interesting screen:

```
5000  PRINT CHR$(147);
5010  FOR T = 1 TO 999
5020  PRINT CHR$(40 + INT(80 * RND(0) ) );
5030  NEXT
5040  GET A$: IF A$ = " " THEN 5040
```

```
5050  SYS(49156)
5060  GOTO 5060
```

Wait till the screen fills, then hit a key. Wham!

For an interesting variation, replace lines 5050 and 5060 by:

```
5050  FOR K = 120 TO 49 STEP -1
5060  POKE 49155, K
5070  SYS(49156)
5080  NEXT
5090  GOTO 5090
```

For yet another variation, use this last version, but add:

```
5005  POKE 49153, 83
```

which changes one data byte. This is a Valentine's day message, and a sad one: 'I gave you my heart and you left me with nothing.' Try it and you'll see what I mean!

SCREEN INVERTER

If you add 128 decimal to the contents of an address in Screen Memory, the corresponding character changes to 'inverse video'; that is, the foreground and background colours interchange. By looping through the whole Screen Memory area, you can switch the entire display to inverse video in a flash:

```
       LDX #04       A2 04
       LDA #04       A9 04
       STA FC        85 FC
       LDA #00       A9 00
       STA FB        85 FB
       LDY #00       A0 00
loop:  CLC           18
       LDA (FB), Y   B1 FB
       ADC #80       69 80          80 hex = 128 decimal
       STA (FB), Y   91 FB
       INY           C8
       CPY #00       C0 00
       BNE loop      D0 F4
```

```
INC FC        E6 FC
DEX           CA
SEC           38
CPX #00       E0 00
BNE loop      D0 EC
RTS           60
```

If you change the initial LDX #04 to LDX #01 or LDX #02 or LDX #03 then only the first 1, 2, or 3 pages of screen will invert. Here's a BASIC routine to illustrate the program's speed:

```
5000  PRINT CHR$(147);
5010  FOR T = 1 TO 24
5020  PRINT "1111222233334444555566667777888899990000";
5030  NEXT
5040  GET A$: IF A$ = "" THEN 5040
5050  IF A$ = "S" THEN STOP
5060  SYS(49152)
5070  GOTO 5040
```

This fills the screen with characters, and inverts every time you hit a key (other than S which stops the program).

PRINT AT

Another useful routine is a 'PRINT AT R, C' command, which lets you print a given character in a given row and column. Ordinary Sixty-four BASIC lacks this command; but you can obtain the same effect by cursor control. Let's write a Machine Code routine instead. (Actually, there's one in ROM already, which you can use—see Chapter 21—but it's instructive to write your own.) The idea is to compute 1024 + 40 * R + C and use indirection. In fact, the + C is done by indexing.

How do you multiply by 40 in Machine Code? A loop that adds 40 times would work, but it's slow. Instead, we shift left three times, getting 8 * R; remember that; shift left twice more to get 32 * R; then add 8 * R + 32 * R to get 40 * R. Easy!

There will be three data bytes: the screen code of the character to be printed, the row number, and the column number. These go in C000–C002 as usual. I suggest you use:

```
51  0A  0F
```

for a first test.

```
LDA #ØØ       A9 ØØ      ┐
STA FE        85 FE      │
LDA CØØ1      AD Ø1 CØ   ├ store R in FD–FE
STA FD        85 FD      ┘
CLC           18         ┐
ASL FD        Ø6 FD      │
ROL FE        26 FE      │
ASL FD        Ø6 FD      ├ 'quick and dirty'
ROL FE        26 FE      │  product by 8
ASL FD        Ø6 FD      │
ROL           26 FE      ┘
LDA FD        A5 FD      ┐
STA FB        85 FB      │
CLC           18         ├ add 1Ø24 and store
LDA FE        A5 FE      │  in FB—FC
ADC #Ø4       69 Ø4      │
STA FC        85 FC      ┘
CLC           18         ┐
ASL FD        Ø6 FD      ├ double twice more to
ROL FE        26 FE      │  get 32 * R
ASL FD        Ø6 FD      │
ROL FE        26 FE      ┘
CLC           18         ┐
LDA FB        A5 FB      │
ADC FD        65 FD      │
STA FB        85 FB      ├ 8 * R + 32 * R = 4Ø * R
LDA FC        A5 FC      │
ADC FE        65 FE      │
STA FC        85 FC      ┘
LDY CØØ2      AC Ø2 CØ     use indexing to add C
LDA CØØØ      AD ØØ CØ     character screen code
STA (FB), Y   91 FB        print character
RTS           6Ø
```

You should devise a BASIC routine to test this thoroughly. For instance:

```
5000 PRINT CHR$(147)
5010 FOR R = 0 TO 24
5020 FOR C = 0 TO 39
5030 POKE 49153, R: POKE 49154, C
5040 SYS(49155)
5050 NEXT: NEXT
```

will check out the screen positions; and suitable POKEs to 49152 will make sure you're printing the correct character.

A code representing the key currently being pressed is stored in address 197. You can use this for:

17 Keyboard Control

If you want to write Machine Code routines that respond to the keyboard (for example, controlling moving graphics), you have to find a way to detect, from inside Machine Code, which key is being pressed. You can do this by taking a look at the contents of address 197 decimal, 00C5 hex, which contains a (curiously coded) version of the key currently being held down—the code being 64 for 'no key'. The codes are neither ASCII nor Screen Codes; and I've listed them in Appendix 8. To check it out, try a simple BASIC program:

```
7000  PRINT PEEK(197)
7010  GOTO 7000
```

and GOTO 7000. Start pressing keys.

By testing to see what code is in 00C5 and branching accordingly, you can obtain keyboard control of your Machine Code.

LOOP-Y

Here's a simple example. The Y-register controls a loop which prints a character to the screen and erases the spaces on either side of it. If you press no keys, the character moves steadily to the right. If you press 'R' for *reverse* it moves left; and if you press 'S' the program stops. For simplicity, the character moves through one page of Screen Memory.

```
        LDA #00       A9 00
        STA FB        85 FB
        LDA #06       A9 06
        STA FC        85 FC
        LDY #00       A0 00
loop:   LDA #20       A9 20
        STA (FB), Y   91 FB
```

```
       INY            C8
       INY            C8
       STA (FB), Y    91 FB
       DEY            88
       LDA #51        A9 51
       STA (FB), Y    91 FB
test:  LDA #11        A9 11      code for key R
       CMP C5         C5 C5      see if it's being pressed
       BNE skip       D0 02
       DEY            88         Y has already moved 1
                                 place right: now move
       DEY            88         2 places left
skip:  LDA #0D        A9 0D      code for S
       CMP C5         C5 C5      see if it's being pressed
       BNE loop       D0 E5
       RTS            60
```

If you run this, you'll find that everything goes haywire. You see a lot of blinking blobs and precious little that resembles a moving one. The reason is simple: it's moving too fast! The TV can only display 50 pictures every second, and the blob is moving much faster than that.

This is a common problem in Machine Code: the answer is to add a time delay. The easiest method is to use a subroutine:

PUTTING IN A PATCH

We can add a JSR instruction that takes the program to a 'delay' routine. This puts a *patch* in the original program.

We begin as before:

```
       LDA #00        A9 00
       STA FB         85 FB
       LDA #06        A9 06
       STA FC         85 FC
       LDY #00        A0 00
loop:  LDA #20        A9 20
       STA (FB), Y    91 FB
       INY            C8
```

```
          INY          C8
          STA (FB), Y  91 FB
          DEY          88
          LDA #51      A9 51
          STA (FB), Y  91 FB
```

Now's a good place to put the patch:

```
          JSR delay    2Ø 29 CØ    compute destination
                                   from listing
```

After which we resume the original progam:

```
  test:   LDA #11      A9 11
          CMP C5       C5 C5
          BNE skip     DØ Ø2       relative jump un-
                                   changed by patch
          DEY          88
          DEY          88
  skip:   LDA #ØD      A9 ØD
          CMP C5       C5 C5
          BNE loop     DØ E2       jump changed by patch
          RTS          6Ø
```

Finally we add:

A DELAY ROUTINE

The idea here is to use the X-register to run through a loop of length 256 doing nothing, after which we return to the main program. The X-register is important in the main program, so we push it on to the stack (via the accumulator) at the start of the loop and pull it off at the end. Here's the code:

```
  delay:  TXA          8A
          PHA          48
          LDX #ØØ      A2 ØØ
  dloop:  DEX          CA
          CPX #ØØ      EØ ØØ
          BNE dloop    DØ FB
          PLA          68
```

```
        TAX          AA
        RTS          6Ø
```

Note the sequence:

```
┌──TXA   Transfer X to A
│ ┌─PHA   Push A (which holds X now) on to stack
│ │  .
│ │  .
│ │  .
│ └─PLA   Pull A off stack (still holding X value we wanted to
│         remember)
└──TAX   Transfer A to X (back to square one).
```

You might imagine that a loop of 256 operations would slow things down enough, but no! *It's still too fast*. So we use the Y-register to loop the whole delay 256 times. Surely 65536 operations will make it slow enough?

Change the above subroutine (but leave the main program intact) to the following:

```
delay:  TYA          98
        PHA          48
        TXA          8A
        PHA          48
        LDY #ØØ      AØ ØØ
        LDX #ØØ      A2 ØØ
dloop:  DEX          CA
        CPX #ØØ      EØ ØØ
        BNE dloop    DØ FB
        DEY          88
        CPY #ØØ      CØ ØØ
        BNE dloop    DØ F6
        PLA          68
        TAX          AA
        PLA          68
        TAY          A8
        RTS          6Ø
```

This time note that we pull the X- and Y-registers off the stack in the reverse order to how we pushed them:

```
TYA
PHA
TXA
PHA
.
.      — X-register      — Y-register
.
PLA
TAX
PLA
TAY
```

Well . . . now it's too slow. Snail's-pace moving graphics! But we can fix that, because we've now got a general purpose delay loop which we can fine-tune just by changing the start value of Y. You'll find that changing the LDY #ØØ (delay 256) to:

```
LDY #ØA A Ø ØA          delay 8 loops
```

produces a reasonable effect. Reduce ØA to Ø5 or Ø4 and it's really fast; increase to 12 or 16 and it's pretty slow. You can use this delay-loop routine, with suitable initial Y-values, whenever a time delay seems to be needed; and then adjust the Y-value to suit your tastes later.

Now a brief return to programming theory, to take a quick look at another important group of instructions:

18 Logic

There's a final group of Machine Code commands that you ought to be told about—if only because we'll need them in the next chapter on sprites. These are the *logic* instructions:

AND
ORA
EOR

First, a little bit of mathematical logic:

THE LEGACY OF GEORGE BOOLE

A mathematician called George Boole got the idea of using mathematical calculations to study logic around 1854, when he published a book called *The Laws of Thought*. He couldn't possibly have guessed what electronic engineers would be doing with his ideas a century later: his *Boolean algebra* is just what's needed to design computer circuits.

We can use the bits Ø and 1 to represent the logical values 'false' and 'true' respectively. And we can calculate with these using Boole's rules. For example, consider the sentence:

It's Tuesday AND it's raining.

When is this true? Would it be true if it were Wednesday? No—even if it were pouring pussy-cats and pooches. And if it *were* Tuesday, but the Sun was shining and the neighbours were lounging around in bikinis, it still wouldn't be a true statement. *Both* parts in an AND statement have to be true for the whole thing to be true. Or, as Boole essentially put it (in different symbols):

Ø AND Ø = Ø	(false AND false = false)
Ø AND 1 = Ø	(false AND true = false)
1 AND Ø = Ø	(true AND false = false)
1 AND 1 = 1	(true AND true = true)

You're no doubt familiar with this idea from BASIC, and it takes a similar form in Machine Code, as we'll see.

There's also the OR statement (ORA in 6510-ese):

Ø ORA Ø = Ø
Ø ORA 1 = 1
1 ORA Ø = 1
1 ORA 1 = 1

based on the idea that p OR q is true provided at least one of them is: 'it's snowing, OR I'm a blue-nosed skunk'. We don't insist on both!

Lastly in this order of ideas is the *exclusive OR*, otherwise known as EOR (which unaccountably makes me think of Winnie-the-Pooh). Here p EOR q means 'p OR q but *not* both', and we therefore have:

Ø EOR Ø = Ø
Ø EOR 1 = 1
1 EOR Ø = 1
1 EOR 1 = Ø ◄—— note the difference!

BYTE LOGIC

That's how the logic operations work on individual bits: what about bytes? In Machine Code (as in BASIC) they operate on each bit independently. Thus, to find:

1ØØ1Ø1Ø1 EOR 11ØØ1Ø11

we take bit 7 (left-hand ends) and work out:

1 EOR 1 = Ø

to get bit 7 of the result; then move on to bit 6:

Ø EOR 1 = 1

followed by bits 5, 4, 3, 2, 1, Ø:

Ø EOR Ø = Ø
1 EOR Ø = 1
Ø EOR 1 = 1
1 EOR Ø = 1
Ø EOR 1 = 1
1 EOR 1 = Ø

and stick them in line to get the answer:

Ø1Ø1111Ø

Similarly with AND and ORA.

The opcodes for the logic commands are listed in Appendix 4, in all addressing modes (of which there are eight).

MASKING

Perhaps the main use of logic operations in Machine Code programming is to test, or change, individual bits in a byte. Recall that the bits in an 8-bit byte are conventionally numbered:

7	6	5	4	3	2	1	Ø

so that the more senior bits have higher numbers. Suppose I want to test a byte to see what bit 3 is. How do I do it?

There are lots of numbers with bit 3 equal to 1—namely 128 of them; and 128 with bit 3 equal to Ø. There's no very nice pattern to them as far as their arithmetical properties go.

Consider the byte:

ØØØØ1ØØØ

which has Øs *everywhere* except bit 3, the one we're interested in. Call this number M, for *mask*. (It is equal to 8 decimal, of course). The idea is to AND the mask M with the byte concerned. Bits 7, 6, 5, 4 and 2, 1, Ø of the result *must* always be Ø, because Ø AND anything is Ø. If bit 3 of the number is Ø, then the final result is:

ØØØØØØØØ

whereas if bit 3 is 1 the result is:

ØØØØ1ØØØ

In other words, setting M = Ø8, we have:

p AND M = ØØ if bit 3 of p is Ø
p AND M = Ø8 if bit 3 of p is Ø

Similarly we can test bits Ø, 1, ..., 7 by using the masks:

ØØØØØØØ1	Ø1 (hex)	1 (decimal)	for bit Ø
ØØØØØØ1Ø	Ø2	2	for bit 1
ØØØØØ1ØØ	Ø4	4	for bit 2
ØØØØ1ØØØ	Ø8	8	for bit 3
ØØØ1ØØØØ	1Ø	16	for bit 4
ØØ1ØØØØØ	2Ø	32	for bit 5
Ø1ØØØØØØ	4Ø	64	for bit 6
1ØØØØØØØ	8Ø	128	for bit 7

Suppose that we're not so much interested in the value of bit 3: instead we want to set it to zero. Then we can form the difference:

p − (p ORA Ø8)

which knocks that digit out. There are other variations on these masking tricks, but once you've got the general idea, it's easy enough to see how they work.

An unusual and spectacular feature of the Sixty-four is the use of sprites*—large coloured graphic blocks that can be moved around the screen, overlapping as they pass. In BASIC, it's hard to make them move very quickly. Machine Code is different—you have to work hard to slow them down!*

19 Sprites

Sprites, or MOBs (Moveable Object Blocks), are moderately large graphic designs that are handled by a special VIC chip and can be moved about the screen as the programmer wishes. They can be made the basis of many attractive games and displays. They are not entirely straightforward to deal with, however: the aim of this chapter is to introduce some of the fundamental ideas—enough for you to use sprites yourself.

I'd like to start with a general run-down of the main techniques of sprite-handling, because even experienced BASIC programmers may find this a little tricky. Those of you who've read *Easy Programming* may find some sections of this chapter astonishingly familiar! Please bear with me: not everyone reading this book will have come across the material before.

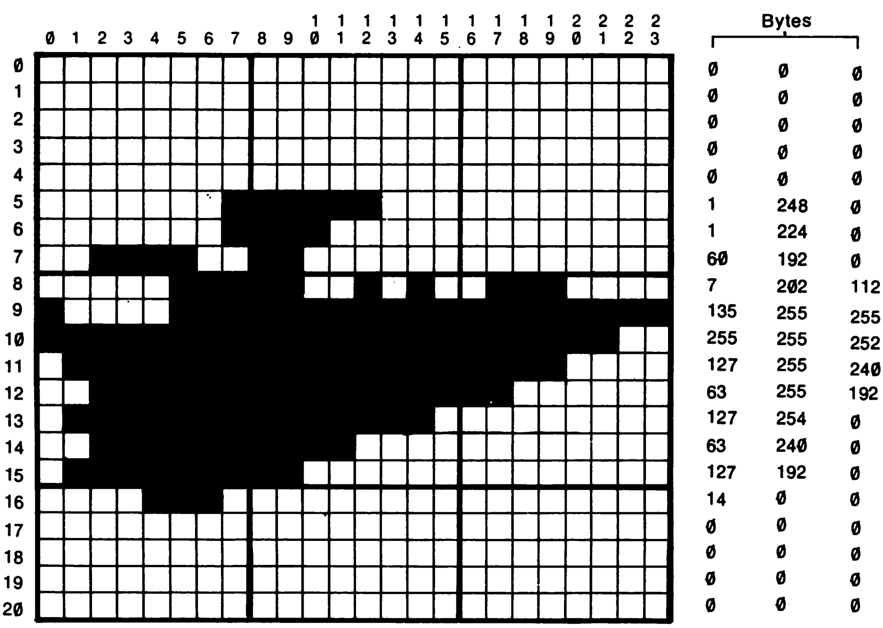


Figure 19.1

SPRITE DESIGN

The information that defines a sprite consists of a 21 × 24 grid, whose cells are either blank or blocked in. For example, Figure 19.1 shows a 'Star Cruiser' shaped sprite.

These blank or blocked in cells must be converted to a series of numbers, to be stored in the appropriate place (see below). To do this, replace every blank cell by a Ø and every full cell by a 1, as in Figure 19.1. Take each row of 24 digits and split it into three 8-digit pieces. For example row 8 of the figure breaks up as:

ØØØØØ111   11ØØ1Ø1Ø   Ø111ØØØØ

These look like binary bytes . . . and indeed that's the idea. Converted to decimal they become:

7   2Ø2   112

So each row of the sprite can be thought of as a series of three decimal numbers (between Ø and 255). The numbers for the entire sprite are listed down the side of Figure 19.1; and conventionally they are read in order from top left to bottom right; that is, the three bytes for row Ø, then the three for row 1, and so on until row 2Ø. That makes 63 numbers altogether.

You *can* design your sprite on squared paper, and work out the decimal numbers by hand. But wouldn't it be much nicer if the computer did all the hard work?

COMPUTER-AIDED SPRITE DESIGN

Here's a fairly simple program to let you design a sprite on screen and generate the list of data. To keep the listing within bounds, various possible improvements have been left out. If you want to make it more sophisticated, go ahead!

```
7Ø1Ø POKE 5328Ø, 4
7Ø2Ø PRINT CHR$(147)
7Ø3Ø FOR S = ØTO 2Ø
7Ø4Ø IF S = 8 * INT(S/8) THEN PRINT
     "------------------------"
                                   [24 – signs]
7Ø5Ø IF S < > 8 * INT(S/8) THEN PRINT
     ".......:.......:.......:"
7Ø6Ø NEXT S
7Ø8Ø PRINT: PRINT: PRINT
```

```
7100 DIM S(20, 23)
7110 FOR R = 0 TO 20
7120 FOR C = 0 TO 23
7130 CDE = 63: GOSUB 8000
7140 GET A$
7150 IF A$ < > "0" AND A$ < > "1" THEN 7140
7160 IF A$ = "0" THEN S(R, C) = 0: CDE = 32: GOSUB 8000
7170 IF A$ = "1" THEN S(R, C) = 1: CDE = 102:
     GOSUB 8000
7180 NEXT C
7190 NEXT R
7200 POKE 53280, 3
7210 GET A$: IF A$ < > "N" AND A$ < > "Y" THEN 7210
7220 IF A$ = "N" THEN POKE 53280, 4: GOTO 7110
7250 PRINT CHR$(19);
7260 FOR R = 0 TO 20
7270 FOR X = 0 TO 16 STEP 8
7280 V = 0
7290 FOR C = 0 TO 7
7300 IF S(R, X + C) = 1 THEN V = V + 2 ↑ (7 - C)
7310 NEXT C
7320 PRINT TAB(24 + X/2); V;
7330 NEXT X
7340 PRINT
7350 NEXT R
7360 GOTO 7360
8000 POKE 1024 + 40 * R + C, CDE
8010 RETURN
```

RUN this. The border turns purple, for reasons which will appear in a moment. You get a 21 × 24 grid of dots and dashes, ruled into 8 × 8 sections for convenience. There is a ? sign at top left. If you hit '1' it is replaced by a chequered pattern; if '0' by a space. It then moves on one place. You can continue in this way, plotting a block or a space, until the whole grid is filled.

At this point the border turns cyan, to remind you that you must press a key. (There's not much room for a message, so this is an easy way out.) 'Y' for 'yes' tells the program to continue; 'N' for 'no' means you made a mistake and want to try again. (On the rerun you must enter all the Øs and 1s again: one place where improvement would be possible.)

The computer then automatically lists out the data for the rows, down the right hand side. Copy them down on paper. (Or print them out to a printer, or copy to a file on cassette tape or disc.)

I've set up the numbers in decimal here, but of course you can convert to hex. The important thing to realize is that *loading* sprite data is fine in BASIC—it's moving the sprites, etc., where Machine Code becomes a must. To make things as easy as possible, I'll use BASIC where I can.

THE SPRITE REGISTERS

Special sections of memory are reserved for sprite-handling. The addresses start at 53248 decimal (DØØØ hex) and end at 53294 (DØ2E). From now on I'll use hex addresses since our final aim is Machine Code. Not all of the sprite registers are useful to a beginner, and I'll ignore the more esoteric ones. In addition there are several *pointers* in addresses Ø7F8–Ø7FF which tell the computer whereabouts to look for the 63 bytes of graphics data needed to define each sprite. I'll describe them in more detail in a moment; but first here's a quick run-down.

Sprite positions

Addresses DØØØ–DØØF hold the column number (or X-coordinate in Hi-res) and row number (Y-coordinate) for each of the eight sprites. These numbers range from Ø–255. Each is held as one byte in a single address.

Offset flag

The eight bits of a single byte at address DØ1Ø define an offset to the right of the X-coordinate (column number). If bit K is set to 1, then 256 is added to the column number. This is needed to place sprites towards the right-hand side of the screen.

Enable/disable

The eight bits of a single byte at address DØ15 enable (switch on) the Kth sprite if bit K is set to 1, and disable (switch it off) if bit K is Ø.

Expand vertically

The eight bits of a single byte at address DØ17 stretch the Kth sprite to twice its height if bit K is 1.

Expand horizontally

Similarly the eight bits in address DØ1D stretch Sprite K to twice its width if bit K is 1.

Collision flag

If two sprites 'collide' then the corresponding bits in DØ1E are set to 1.

Colours

Each address DØ27 to DØ2E holds the colour code (Ø–15), as in Chapter 16) for one sprite.

Data Pointers

Addresses Ø7F8–Ø7FF (top end of Colour RAM) hold pointers to the start addresses of the data for Sprites Ø–7 respectively. If the Kth pointer has value PTR, then the address for the data starts at 64 * PTR. We will call this the PTRth *block* of memory, from 64 * PTR to 64 * PTR + 63. This lets you define sprites anywhere in the first 16348 bytes of RAM. There are ways to use the other 49152 bytes, but they're messy: see the *Reference Guide* pages 101 and 133. *But* you can't just dump sprites in any old addresses: the BASIC system will clobber the data. See below for recommended addresses.

The addresses for controlling sprites are summarized in Tables 19.1 and 19.2, which are repeated for convenience as Appendix 7. For the meaning of the omitted addresses, see the *Reference Guide* pp. 131–181. That's 50 pages: I told you sprites weren't entirely straightforward!

Table 19.1 Sprite data pointers

Address	Contents
Ø7F8	Sprite Ø data pointer
Ø7F9	Sprite 1 data pointer
Ø7FA	Sprite 2 data pointer
Ø7FB	Sprite 3 data pointer
Ø7FC	Sprite 4 data pointer
Ø7FD	Sprite 5 data pointer
Ø7FE	Sprite 6 data pointer
Ø7FF	Sprite 7 data pointer

3. POKE the data into position.
4. Enable the sprite.
5. Define the colour of the sprite.
6. Set the row and column numbers for the sprite.

Let's take the Star Cruiser Sprite, above, and set it up as Sprite 1. This will do the job:

```
9000  V = 53248
9100  DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
9110  DATA 1, 248, 0, 1, 224, 0, 60, 192, 0, 7, 202, 112, 135, 255,
      255
9120  DATA 255, 255, 252, 127, 255, 240, 63, 255, 192, 127, 254, 0
9130  DATA 63, 240, 0, 127, 192, 0, 14, 0, 0, 0, 0, 0, 0, 0, 0
9140  DATA 0, 0, 0, 0, 0, 0
9200  POKE 2041, 13       [Sprite 1 pointer to 13th block;
                          2041 = 07F9 hex]
9210  FOR G = 0 TO 62
9220  READ H              [read data]
9230  POKE 832 + G, H     [POKE to block: note 832 = 64 * 13]
9240  NEXT G
9250  POKE V + 21, 2      [enable Sprite 1]
9260  POKE V + 40, 7      [Sprite 1 yellow]
9270  POKE V + 2, 100     [Sprite 1 in column 100]
9280  POKE V + 3, 100     [Sprite 1 in row 100]
```

Type this in *carefully* and RUN 9000: you should see the Star Cruiser in yellow, as required.

You can experiment with changing the positions by direct commands:

```
POKE V + 2, 110
```

moves it to the right:

```
POKE V + 3, 90
```

moves it up:

```
POKE V + 40, 5
```

turns it green. Try other values, see what happens.

MOVEMENT

You could have done all that in Machine Code, of course. But, as I said, initial setting-up is OK in BASIC. However, I'll have to use Machine Code to get any reasonable speed of movement.

I'll want to use much the same code, but modified on each occasion, for the next few stages. The modifications come at the beginning, which is awkward using LOADER. So I'll use a trick: the No OPeration command:

NOP (opcode EA)

This means 'ignore this instruction'. A block of NOPs will provide a program area which we can *edit* to fit extra bits in. Otherwise the NOPs will be harmless (though causing a tiny initial delay).

Put a block of 12 NOPs (an arbitrary, and unnecessarily large number) into the start of the program:

```
        NOP          EA
         .            .
         .            .
         .            .
        NOP          EA
```

and then continue with the guts of the thing:

```
        LDX #00      A2 00
loop:   STX D002     8E 02 D0      X-coordinate of
                                   Sprite 1
        JSR delay    20 1B C0
        CLC          18
        INX          E8
        CPX #00      E0 00
        BNE loop     D0 F4
        RTS          60
```

(To move vertically change STX D002 to STX D003.)

It's fairly likely we'll need a time delay, hence the JSR delay. The first time, I tried this with the long delay (looping Y and X registers), but that turned out to be over optimistic, and the sprite limped along like a one-legged tortoise. So I suggest you use the shorter version:

```
delay:  TXA          8A
        PHA          48
```

```
        LDX#8Ø        A2 8Ø      reasonable length for
                                 loop
dloop:  DEX           CA
        CPX#ØØ        EØ ØØ
        BNE dloop     DØ FB
        PLA           68
        TAX           AA
        RTS           6Ø
```

Now prepare for the movement by adding BASIC lines:

```
98ØØ  GET A$: IF A$ = " " THEN 98ØØ
981Ø  SYS (49152)
```

and RUN 9ØØØ.

You'll see the Star Cruiser Sprite build up in yellow. Hit a key: it will rapidly disappear from the centre of the screen and whiz across from left to right, stopping about three quarters of the way across. In fact its horizontal coordinate is now 255 decimal, the largest we can deal with using only address DØØ2. I'll show you one way to get round that in a moment; but first, we'll see how to make the sprite bigger.

EXPANSION

To make the sprite twice as wide, edit out the first five bytes of program, changing them from EA to:

```
LDA #Ø2       A9 Ø2
STA DØ1D      8D 1D DØ
```

now repeat the procedure: the sprite will be stretched horizontally. Edit the next three EA bytes to read:

```
STA DØ17      8D 17 DØ
```

and it will be twice as high too.

What did we do? We set bit 1 of registers DØ1D and DØ17 to 1, by storing 2 ↑ 1 = Ø2. If you look at Table 19.2 you'll see that these control the expansion.

WHOLE SCREEN MOVEMENT

If you want to position your sprite on the right-hand side of the screen, beyond column 255, you use the *offset flag* in DØ1Ø. If bit K of this is set to 1, then 256 is added to the column number for Sprite K. Here's an example. Load it in from C000 as usual (no block of NOPs now!):

```
        LDA D010     AD 10 D0   ┐  use masking with FD
        AND FD       29 FD      ├─ (11111101 binary) to
        8D 10 D0     8D 10 D0   ┘  reset offset for Sprite 1
                                   to 0
        LDY #00      A0 00
        LDX #00      A2 00
loop:   STX D002     8E 02 D0
        JSR delay    20 28 C0
        INX          E8
        CPX #00      E0 00
        BNE loop     D0 F5
        INY          C8
        CLC          18
                                   add 2 to offset to set
        INC D010     EE 10 D0   ┐  bit 2 of it to 1 and
        INC D010     EE 10 D0   ┘─ hence move Sprite 1 by
                                   256 columns
        CPY #02      C0 02
        BNE loop     D0 E9
        SEC          38
        DEC D010     CE 10 D0
        RTS          60
delay:  TXA          8A
        PHA          48
        LDX #00      A2 00         change 00 to 80 for
dloop:  DEX          CA            faster action
        CPX #00      E0 00
        BNE dloop    D0 FB
        PLA          68
        TAX          AA
        RTS          60
```

Now RUN 9000 as usual: this time the sprite whizzes across the screen and disappears off the right-hand edge. (A bit of it pokes out of the left side at the end: you can prevent this by resetting the enable/disable flag, bit 2, if you wish to experiment.)

KEYBOARD CONTROL

We've already seen how to read the keyboard in Machine Code programs, so let's modify the above routine to give us control of the vertical position of the star cruiser on the TV screen. Key 'U' will mean 'up'; 'D' is 'down'; and 'S' is 'stop', because I'm going to make the cruiser fly repeatedly across the screen. Now the code (loaded in at CØØØ) has a few extra wrinkles:

```
start:  CLC          18
        LDA #ØD      A9 ØD        look for key 'S' (code
        CMP C5       C5 C5        ØD) being pressed and
        BNE skip     DØ Ø1        RTS if it is
        RTS          6Ø
skip:   LDA DØ1Ø     AD 1Ø DØ
        AND FD       29 FD
        STA DØ1Ø     8D 1Ø DØ
        LDY #ØØ      AØ ØØ
        LDX #ØØ      A2 ØØ
loop:   STX DØØ2     8E Ø2 DØ
        JSR delay    2Ø 35 CØ     subroutine addresses
        JSR keys     2Ø 41 CØ     worked out after
                                  writing code and
                                  inserted
        INX          E8
        CPX #ØØ      EØ ØØ
        BNE loop     DØ F2
        INY          C8
        CLC          18
        INC DØ1Ø     EE 1Ø DØ
        INC DØ1Ø     EE 1Ø DØ
        CPY #Ø2      CØ Ø2
        BNE loop     DØ E6
        SEC          38
        DEC DØ1Ø     CE 1Ø DØ
        JMP start    4C ØØ CØ     repeat whole thing
delay:  TXA          8A           subroutine at CØ35
        PHA          48
```

```
        LDX #00       A2 00
dloop:  DEX           CA
        CPX #00       E0 00
        BNE dloop     D0 FB
        PLA           68
        TAX           AA
        RTS           60
keys:   LDA #12       A9 12         subroutine at C041
        CMP C5        C5 C5         look for key 'D'
        BNE skip1     D0 03
        INC D003      EE 03 D0 ---- increase row number
                                    by 1
skip1:  LDA #1E       A9 1E         key 'U'
        CMP C5        C5 C5
        BNE skip2     D0 03
        DEC D003      CE 03 D0
skip2:  RTS           60
```

Now when you start up with GOTO 9000 or RUN 9000 and hit a key the cruiser will traverse the screen repeatedly. Touch key 'U' for upward movement; key 'D' for downward. It's extremely fast! When you've tired of that, key 'S' will stop the thing. Hold 'S' down for a few seconds.

SPRITE PRIORITY

If two sprites overlap, the one with the smallest number appears to be on top of the other. The one 'underneath' will however show through any 'holes' in the one on top, just as you'd expect in real life.

To try this out, I'm going to set up another sprite. Same routine (for the moment this is good practice, but later I'll suggest a better approach if you want to use a *lot* of sprites):

```
9400 DATA 0, 255, 0, 3, 255, 192, 15, 195, 240, 63, 0, 252, 255,
     0, 255
9410 DATA 63, 0, 252, 127, 195, 254, 31, 255, 248, 3, 255, 192
9420 DATA 0, 255, 0, 0, 195, 0, 1, 129, 128, 3, 0, 192, 6, 0, 96
9430 DATA 15, 0, 240, 15, 0, 240, 7, 129, 224, 3, 195, 192
9440 DATA 1, 231, 128, 0, 0, 0, 31, 255, 248
```

```
9500  POKE 2040, 14        [Sprite 0 pointer to 14th block]
9510  FOR G = 0 TO 62
9520  READ H
9530  POKE 896 + G, H      [block 14: 896 = 64 * 14]
9540  NEXT G
```

To enable Sprite 0 as well as Sprite 1, we must change line 9250 above to:

```
9250  POKE V + 21, 3
```

because 3 = 00000011 in binary, so bits 1 and 0 are set to 1. Now we continue:

```
9560  POKE V + 39, 5       [Sprite 0 green]
9570  POKE V, 120          [Sprite 0 in column 120]
9580  POKE V + 1, 95       [Sprite 0 in row 95]
9590  POKE V + 29, 3       [expand Sprites 0, 1 horizontally]
9600  POKE V + 23, 1       [expand Sprite 0 vertically]
```

(Again, you could do a lot of this in Machine Code; but for the purposes of illustration BASIC is easier. You might like to work out a Machine Code routine for lines 9560–9600 though, as an exercise.)

You should still have the previous piece of Machine Code—the one with keyboard control—in memory. If you RUN 9000 you can make the star cruiser pass across the other monstrosity by judicious use of the 'U' and 'D' keys. See how it seems to go *behind*? That's because the green object (Sprite 0) has priority over the cruiser (Sprite 1).

Suppose we want the cruiser to pass *in front* of the Green Thing. Then we must change the priority. A simple way is to make the Green Thing be Sprite 2 rather than Sprite 1. This entails the following changes:

```
9500  POKE 2042, 14
9250  POKE V + 21, 6    [6 = 00000110]
9560  POKE V + 41, 5
9570  POKE V + 4, 120
9580  POKE V + 5, 95
9590  POKE V + 29, 6
9600  POKE V + 23, 4
```

Try it now: the cruiser goes in front, not behind.

USING THE SAME DATA FOR SEVERAL SPRITES

We can set more than one sprite to the same data, by making two or more pointers the same. Suppose we've got Sprites 1 and 2 set up as above; but now we want Sprite Ø to be a Black Thing (also double in size) in another position. We can do this: enable all three sprites by changing 925Ø yet again:

925Ø	POKE V + 21, 7	[7 = ØØØØØ111]

Now set up Sprite Ø:

97ØØ	POKE 2Ø4Ø, 14	[data for Sprite Ø from same block, 14]
		[Sprite Ø black]
977Ø	POKE V, 7Ø	[Sprite Ø in column 7Ø]
978Ø	POKE V + 1, 124	[Sprite Ø in row 124]
979Ø	POKE V + 29, 7	[all 3 sprites stretched horizontally]
9795	POKE V + 23, 5	[only Ø and 2 vertically]

If you RUN now you'll find two things, plus one cruiser.

COLLISION DETECTION

By using the collision register, DØ1E hex (or V + 3Ø with V = 53248 as in our BASIC programs), you can tell when a collision between sprites occurs. If two sprites collide, then those two bits are set to 1. For example if Sprites 1 and 2 collide, then DØ1E will hold:

ØØØØØ11Ø = Ø6

This value is updated at every collision. So to test for a 1:2 collision you'd need a piece of Machine Code like this:

```
        LDA #Ø6
        CMP 1E DØ
        BEQ action
        .
        .
        .
action: Whatever you want to happen
        when they collide.
```

Address DØ1F (V + 31) responds to a collision between sprites and text in foreground colour: bit K is set if Sprite K collides. (The *Reference Guide* says 'sprite-background collision' but means 'sprite-foreground collision'.)

WHERE TO STORE SPRITE DATA

For three or fewer sprites, you can use blocks 13, 14 and 15. These actually lie in the *cassette buffer*, an area of memory only used when the cassette recorder is operating. So it's a safe place to store sprites. Unfortunately, it's not long enough to hold all eight 64-byte blocks. So you need to try somewhere else. Unless you have a very long BASIC program, the *Reference Guide* suggests blocks 192–199. Again, if you want to know more, consult the *Reference Guide*.

THAT'S JUST THE START

This has been a long chapter, and we've barely scratched the surface. You can, for example, have multicoloured sprites. But space is running out, and I hope you've got enough ideas to keep you busy as it is. Once you've mastered what I've told you about sprite-handling, you might take a look at the *Reference Guide* for additional possibilities, beyond the scope of this book.

The Manual tells you how to use graphics characters, but it doesn't mention that the Sixty-four is capable of something much more impressive:

20 High-Resolution Graphics

Each character cell on the TV display is in fact made up of an 8 × 8 square of tiny cells, or *pixels* which are used to build up the character (deep down inside the electronics). By obtaining direct access to these cells, you can plot graphical displays on the *Hi-res* (High-resolution) screen. That means you have a display of 25 × 8 = 200 rows and 40 × 8 = 320 columns. It's almost the same number system that the sprites use, but restricted to the screen area (see Figure 20.1).

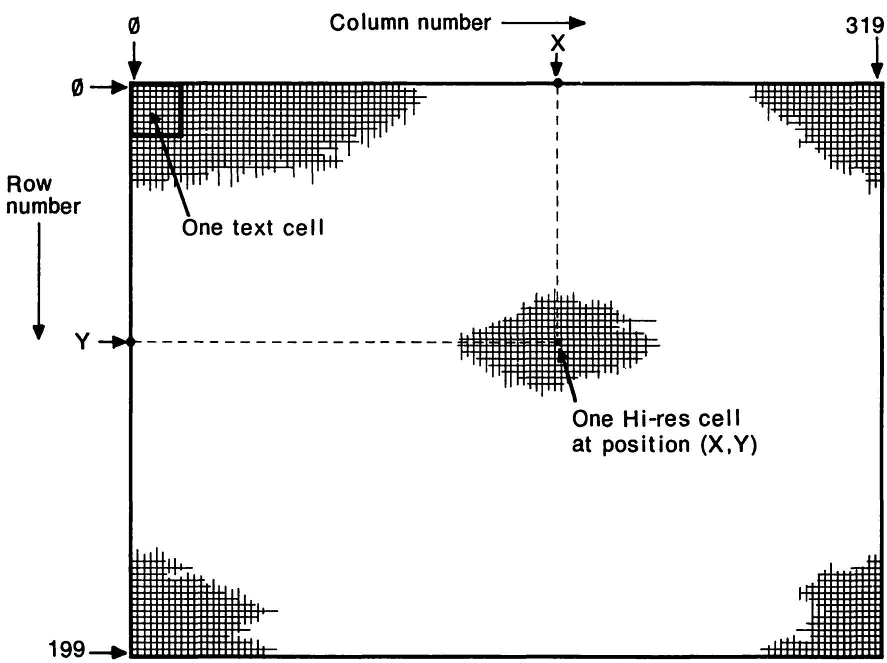


Figure 20.1

HI-RES MODE

In order to make your machine capable of high resolution graphics, you must put it into hi-res mode, set up an area of memory to hold the graphics data, and clear out that area. It is also necessary to assign colours. The *Reference Guide* explains this on page 123. Here's a BASIC program (so you can see what's involved) that clears the screen to light green. If you change the 13 in 11Ø8Ø to

16 * INK + PAPER

where INK and PAPER are the colour codes for foreground and background, you can get any combination of colours you want. The following routine will put the screen memory area at address 8192:

```
11ØØØ  REM HI-RES INITIALIZATION
11Ø1Ø  POKE 53265, PEEK(53265) OR 32
11Ø2Ø  POKE 53272, PEEK(53272) OR 8
11Ø3Ø  BM = 8192
11Ø4Ø  FOR U = BM TO BM + 7999
11Ø5Ø  POKE U, Ø
11Ø6Ø  NEXT U
11Ø7Ø  FOR U = 1Ø24 TO 2Ø23
11Ø8Ø  POKE U, 13
11Ø9Ø  NEXT U
111ØØ  RETURN
```

RUN this. First you get junk; then the screen clears to a mostly black background but with some coloured blobs where the text was; then it all clears to light green. (Change the 13 in line 11Ø8Ø to 16 * INK + PAPER where INK and PAPER are the colour codes you want. This program gives black ink on light green paper.)

Note that the screen memory clearing is rather slow: about 20 seconds in BASIC.

AND NOW IN MACHINE CODE

Since BASIC is so slow, here's the same program converted into Machine Code:

```
LDA DØ11    AD 11 DØ
ORA #2Ø     Ø9 2Ø
STA DØ11    8D 11 DØ
```

```
         LDA DØ18       AD 18 DØ
         ORA #Ø8        Ø9 Ø8
         STA DØ18       8D 18 DØ
         LDA #ØØ        A9 ØØ
         STA FB         85 FB
         LDA #2Ø        A9 2Ø
         STA FC         85 FC
         LDY #ØØ        AØ ØØ
loop:    LDA #ØØ        A9 ØØ
         STA (FB), Y    91 FB
         INC FB         E6 FB
         CMP FB         C5 FB
         BNE skip       DØ Ø2
         INC FC         E6 FC
skip:    LDA #3F        A9 3F
         CMP FB         C5 FB
         BNE loop       DØ EE
         CMP FC         C5 FC
         BNE loop       DØ EA
         LDA #ØØ        A9 ØØ
         STA FB         85 FB
         LDA #Ø4        A9 Ø4
         STA FC         85 FC
         LDY #ØØ        AØ ØØ
loop2:   LDA #ØD        A9 ØD
         STA (FB), Y    91 FB
         INC FB         E6 FB
         LDA #ØØ        A9 ØØ
         CMP FB         C5 FB
         BNE skip2      DØ Ø2
         INC FC         E6 FC
skip2:   LDA #E7        A9 E7
         CMP FB         C5 FB
```

```
BNE loop2    D0 EC
LDA #07      A9 07
CMP FC       C5 FC
BNE loop2    D0 E6
RTS          60
```

If you run this using SYS(49152), you'll find the screen clears in a trice!

PLOT

The hi-res columns and rows define a system of coordinates on the TV screen, as shown in Figure 20.1. The main job is to find a way to *plot* a single pixel at column X, row Y—that is, coordinates (X, Y). By combining such plots we can draw lines, curves, and fill in entire regions. Here's a BASIC routine to draw a single pixel in row Y, column X. It assumes that Y is between 0 and 199, X between 0 and 320. If you want to know why it works, see the *Reference Guide* or *Easy Programming*, Chapter 32.

```
12000 REM PLOT X, Y
12010 BY = BM + 320 * INT(Y/8) + 8 * INT(X/8) + (Y AND 7)
12020 BT = 7 - (X AND 7)
12030 POKE BY, PEEK(BY) OR (2 ↑ BT)
12040 RETURN
```

Assuming you've got the 'clear hi-res screen' Machine Code in place at C000, here's an example of how to use hi-res plotting:

```
13000 SYS(49152)
13010 FOR X = 0 TO 319
13020 Y = 100 + 80 * SIN(X/10)
13030 GOSUB 12000
13040 NEXT
```

In conjunction with the *plot* subroutine, this gives a wavy sine curve. Changing line 13020 leads to other curves.

HOW DOES THIS WORK?

This section gets a little technical, so you can skip it if you want to and come back later.

Each byte in the Hi-res Screen Memory holds data for an 8 × 1 row of pixels on the Hi-res screen. A binary 0 means 'no dot here' and a 1 means 'put a dot here'. So for example the byte 10110101 gives the effect shown in Figure 20.2.

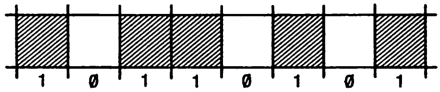


Figure 20.2

When you set the system variable in address 53265 to give Hi-res mode, the computer is instructed by the operating system to interpret the data in this way. This is called *bit-mapped* graphics.

The addresses for our Hi-res Screen Memory correspond to the actual screen positions as shown in Table 20.1.

Table 20.1

		0	1	2	...	39 ← Lo-res column number	
	0	8192	8200	8208	...	8504	
	1	8193	8201	8209	...	8505	
	2	8194	8202	8210	...	8506	
	3	8195	8203	8211	...	8507	
Hi-res	4	8196	8204	8212	...	8508	0 Lo-res row number ↓
row	5	8197	8205	8213	...	8509	
number ↓	6	8198	8206	8214	...	8510	
	7	8199	8207	8215	...	8511	
	8	8512	8520	...	...	...	
	9	8513	8521	...	...	...	
	10	8514	8522	...	...	...	
	11	8515	8523	...	...	...	1
	12	8516	8524	...	...	...	
	13	8517	8525	...	...	...	
	14	8518	8526	...	...	...	
	15	8519	8527	...	...	...	
	.	...	...	...	...	...	
	.	...	...	...	...	...	
	.	...	...	...	...	...	

In hex, these addresses start at 2000. So each character cell, which used to correspond to *one* address in Screen Memory, now corresponds to *eight* addresses: a block of memory eight bytes long. The blocks are arranged in the same order as the cells in Screen Memory: go along Lo-res rows first, and skip down a row after column 39.

Suppose we want to put a diagonal line in the top left corner, 5 pixels long. The addresses and contents take the form of Figure 20.3

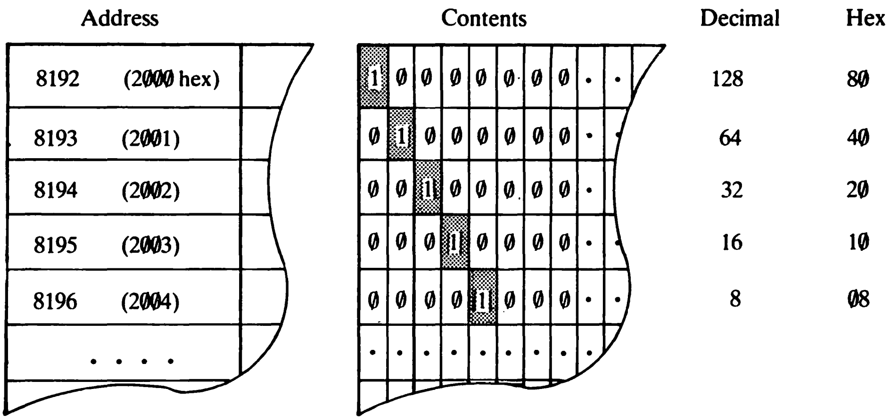


Figure 20.3

So this program should do the trick:

```
10  GOSUB 11000           [Enter Hi-res mode subroutine]
20  POKE 8192, 128
30  POKE 8193, 64
40  POKE 8194, 32
50  POKE 8195, 16
60  POKE 8196, 8
70  GOTO 70
```

Try it and see.

The same approach works in general:

1. Find the relevant address.
2. POKE it with the necessary value to produce the desired screen display. Or use a Machine Code STA command, as we'll see later.

Since we don't want to obliterate anything that's on the screen already, we must assume that the address may hold a non-zero value. That requires us to OR the contents with the new value (see Chapter 18).

Line 12010 calculates the correct address.
Line 12020 calculates the value to be POKEd in, to plot one new pixel.
Line 12030 ORs this with the existing contents and POKEs the result back in.
For more details, consult the *Reference Guide*, page 125.

A MACHINE CODE 'PLOT' ROUTINE

Unless you're very ambitious, you'll probably want to use a BASIC program to 'drive' the hi-res plotting. But there's no need to use BASIC for the actual PLOT X, Y routine at the heart of it. Let's do it in Machine Code.

I'll give it you as a bare routine: at the end I'll suggest ways to incorporate it, and the 'clear hi-res screen' routine, into a single package.

It uses four data bytes:

CØØØ	XJ-coord	junior byte of column number	(up to
CØØ1	XS-coord	senior byte of column number	319 total)
CØØ2	Y-coord	row number (up to 199)	
CØØ3	test	used during debugging	

So the program starts at CØØ4.

A 16-bit adder is going to be indispensable. First we write one which adds the contents of ØØFB–ØØFC to ØØFD–ØØFE and stores the result in ØØFB–ØØFC. Zero-page keeps the code simpler.

```
add:   CLC          18
       LDA FB       A5 FB
       ADC FD       65 FD
       STA FB       85 FB
       LDA FC       A5 FC
       ADC FE       65 FE
       STA FC       85 FC
       RTS          6Ø
```

It's just like the 16-bit adder from Chapter 8, but implemented in page zero. Note that I've written it as a subroutine (at CØØ4).

Next we start the main program, which is at CØ12 (4917Ø decimal). We've got to build up the equivalent of BASIC's:

BY = BM + 32Ø * INT(Y/8) + 8 * INT(X/8) + (Y AND 7)

where BM = 8192 = 2ØØØ hex. We start by getting 2ØØØ into place:

```
main:  LDA #2Ø      A9 2Ø
       STA FC       85 FC
       LDA #ØØ      A9 ØØ
       STA FB       85 FB
       STA FD       85 FD
```

The next job is to build up INT(Y/8). This is done by right-shifting it three times in a row:

```
LDA Y-coord     AD Ø2 CØ
LSR             4A
LSR             4A          — not worth looping!
LSR             4A
```

Now for the tricky bit. To multiply by 32Ø isn't *that* hard; but there's an easier way than direct multiplication. Note that 256 + 64 = 32Ø. To multiply by 256 is simple: move the junior byte to senior! Since I've cunningly put #ØØ into FD already, all we need is:

```
STA FE          85 FE
```

Now add it to the accumulating total in FB–FC:

```
JSR add         2Ø Ø4 CØ
```

You might imagine that the way to get 64 * INT(Y/8) is to double INT(Y/8) six times; but with what we've got already it's easier to halve 256 * INT(Y/8) twice!

```
CLC             18
LSR FE          46 FE
ROR FD          66 FD
                            — take care with carries
LSR FE          46 FE
ROR FD          66 FD
JSR add         2Ø Ø4 CØ
```

That's built up the equivalent of BM + 32Ø * INT(Y/8). Now for the 8 * INT(X/8). This is just X with its bits Ø–2 reset to Ø, so we can mask them off. We only have to work on the junior byte of X, too!

```
LDA XS-coord    AD Ø1 CØ
STA FE          85 FE
LDA XJ-coord    AD ØØ CØ
AND #F8         29 F8       F8 = 11111ØØØ binary:
                            mask in use
STA FD          85 FD
JSR add         2Ø Ø4 CØ
```

This leaves only the (Y AND 7) term in this part of the calculation, which doesn't take much effort at all:

```
LDA #ØØ         A9 ØØ
```

```
STA FE          85 FE
LDA Y-coord     AD Ø2 CØ
AND #Ø7         29 Ø7
STA FD          85 FD
JSR add         2Ø Ø4 CØ
```

We've now finished that part of the computation, and the address for storage of the relevant byte of screen is in ØØFB–ØØFC. Cunningly placed ready to use post-indexed indirection! (There's no flies on *this* baby, let me tell you.)

However (puff, pant), we're not finished. There's the next part, the stuff with BT. First we need to calculate 7 − (X and 7). Again, only the junior byte is required:

```
LDA XJ-coord    AD ØØ CØ
AND #Ø7         29 Ø7
STA FD          85 FD
LDA #Ø7         A9 Ø7
SEC             38
SBC FD          E5 FD
TAX             AA              BT is in X-register
```

I've shovelled it into the X-register because I want to use it to control a loop to build up 2 ↑ BT:

```
       INX         E8
       CLC         18
       LDA #Ø1     A9 Ø1
loop:  DEX         CA
       CPX #ØØ     EØ ØØ
       BEQ skip    FØ Ø4
       ASL         ØA
       CLC         18         forces a branch with a
       BCC loop    9Ø F7      relative displacement
                              (relocatable code,
                              not JMP)
```

Now all we have to do is OR this with the contents of the Screen Memory byte (indirect post-indexed addressing works wonders here) and store it (ditto) back again:

```
skip:  LDY #ØØ     AØ ØØ
```

```
ORA (FB), Y        11 FB
STA (FB), Y        91 FB
```

During development I added a line:

```
test:   STA C003        8D 03 C0
```

which let me find out what byte was ending up in the accumulator by PEEKing 49155. (And a good job I did, I can tell you, because I made an absolute bog of the first attempt, by missing out one line of program.) You can omit this; but don't *ever* omit the final:

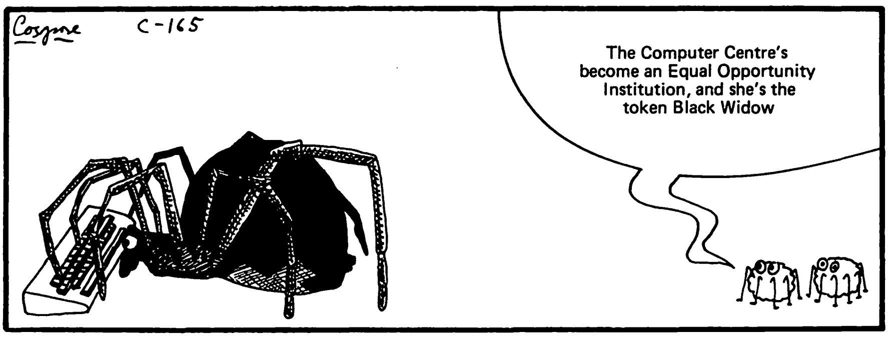
```
RTS        60
```

to get back to BASIC.

To use this routine, you have to load XJ, XS, and Y in place in the data area (C000, C001, C002) and then use:

```
SYS(49170)
```

to kick off from *main* and not *add*!



A HI-RES PACKAGE

All the above got developed a bit piecemeal. The final task is to put the bits together into an organized package that you can use reliably.

You've currently got the plot routine in memory. After the final RTS, you can add on the clear-screen routine we had before. This will be at address C074 (or C071 if you omitted the test line, as is your right): check with LOADER's print option to make sure. You've now got a plot routine at address 49170 and a clear-screen routine at 49268 (or whatever). Now you can write a BASIC 'driver' program: for example drawing a circle:

```
15000  SYS(49268): REM CLEAR HI-RES SCREEN
15010  FOR D = 0 TO 359
15020  DR = PI * D/180: REM CONVERT TO RADIANS
15030  X = INT(160 + 90 * COS(DR) )
```

```
15040  Y = INT(100 + 90 * SIN(DR) )
15050  XS = INT(X/256): XJ = X − 256 * XS
15060  POKE 49152, XJ: POKE 49153, XS: POKE 49154, Y
15070  SYS(49170): REM PLOT X, Y
15080  NEXT: NEXT
15090  GOTO 15090
```

You can modify this in lots of ways, of course. And you can write Machine Code routines at higher addresses still, to drive the *clear-screen* and *plot* routines.

What LOADER lacks is a good editor. But the Sixty-four already has an excellent editor, the one it uses for BASIC. Here we show you how to fool the computer into using the BASIC editor to edit Machine Code instead!

21 MINIASS – An Aid to Hand Assembly

So far, our techniques for assembling code and loading it into memory to be executed have been, shall we say, fairly primitive.

You can, of course buy an assembler to do the whole job for you (see Chapter 22) but that has two disadvantages. First of all, it costs you money, and secondly, you tend not to learn so much about the way Machine Code really works, because the assembler hides things from you. In any case, assemblers on cassette, are, by and large, less than ideal; you really need the disc versions if you want powerful utilities.

This chapter presents you with a compromise; a remarkably simple set of BASIC routines which will take away some of the hard work, and which will certainly make debugging easier.

THE EDITOR

It will have struck you by now that we need a way of editing code simply, to add subroutines, change the program for debugging, or just because you've forgotten to put in an instruction. Well, we've already got one—the BASIC editor. If only we could harness it in some way to edit Machine Code, half our problems go away before we start. This is where a feature of Commodore BASIC, which is usually a nuisance, suddenly comes into its own. If you write a line number followed by gibberish, BASIC will happily load it, and only complain when it tries to execute it. If the 'gibberish' is hex Machine Code, and we never try to execute it but, rather, execute only a loading routine with a higher starting line number, all will be well. So our code could look like this:

```
10 : A2 00          LDX #00
20 : A0 FF          LDY #FF
30 : BD 00 C0       LDA C000, X
40 : 09 F0          ORA #F0
50 *
```

Notice three things:

1. Each line starts with a colon. This separates the line number from the code, which doesn't matter for lines 10–30, but line 40 without the colon would be interpreted 4009, and so would come after line 50.
2. Each byte of code is separated by exactly one space. If two or more spaces appear, the program assumes the instruction is complete and ignores anything which follows. That allows you to comment every line, by writing the assembler equivalent for instance, as I've shown.
3. An asterisk in the colon position acts as a delimiter for the code, showing where it ends.

STORING BASIC

Now, to make this work, we need to know how BASIC code is stored in the Sixty-four. It's pretty straightforward. It starts from 2048 (decimal) which always contains a zero. The next two bytes hold a pointer to the beginning of the next line. The following two bytes hold the line number, and then comes the text of the line, delimited by a zero byte.

Here's an example:

```
10 : A2 00
20 : A0 FF
```

Machine Address	Contents	Interpretation
2048	0	always zero
2049	12	next line pointer = 8 × 256 +
2050	8	12 = 2060
2051	10	line no. = 0 × 256 + 10 = 10
2052	0	
2053	58	:
2054	65	A
2055	50	2
2056	32	space
2057	48	0
2058	48	0
2059	0	end of line
2060	23	next line pointer = 8 × 256 +
2061	8	23 = 2071
2062	20	line no = 20
2063	0	
2064	58	:
2065	65	A
2066	48	0
2067	32	space
2068	70	F
2069	70	F
2070	0	end of line

THE CODE

We'll make the main routine start at 10000. All it has to do is ask where the assembled code is to be loaded, initialize the line start address (LS) and then call a routine to handle a single instruction (i.e. one line). Then it simply resets the line start address using the line pointer bytes and repeats the process. The instruction decoder returns a flag called FINISH, which is zero (false), if there are instructions left to deal with, and −1 (true) if it has come across the terminating asterisk.

```
10000  INPUT "START ADDRESS FOR CODE"; SA
10010  LS = 2049: PB = SA
10020  GOSUB 12000: REM DECODE AN INSTRUCTION
10030  IF FINISH THEN END
10040  LS = PEEK(LS) + 256 * PEEK(LS + 1)
10050  GOTO 10020
```

The instruction decoder looks like this:

```
12000  REM DECODE AN INSTRUCTION
12010  PT = LS + 4: FINISH = 0
12020  IF PEEK(PT) = 172 THEN FINISH = -1: RETURN
12030  IF PEEK(PT) = 58 THEN GOSUB 14000: RETURN:
       REM NO LABEL
12040  GOSUB 16000: RETURN: REM LABEL
```

PT is set to LS + 4 to skip the next line pointer and line number. PT should now be pointing at an asterisk (172), in which case we've finished, or a colon (58), in which case we call a subroutine at 14000 which handles the instruction if there's no label. What's all this about labels? I never said anything about them. Well, no author likes to be accused of label.

We'll defer this discussion till later (a good thing if that pun is anything to go by). For the minute, we'll assume that the character following the line number is guaranteed to be either a colon or an asterisk, so line 12040 can't be reached.

Since PT is pointing at a colon, we have to increment it by 1 to point at the first hex digit of a byte. Then we'll call a subroutine at 20000 which decodes the byte and stores it in address PB. PB is bumped by 1 to be ready for the next byte, and PT is bumped by 2, which will leave it pointing at either a space between bytes, or several successive spaces, or an end of line number. In the latter two cases, the line is finished with so we can RETURN.

```
14ØØØ  PT = PT + 1
14Ø1Ø  GOSUB 2ØØØØ: REM DECODE A BYTE INTO PB
14Ø2Ø  PB = PB + 1
14Ø3Ø  PT = PT + 2
14Ø4Ø  IF PEEK(PT) = Ø OR (PEEK(PT) = 32 AND
       PEEK(PT + 1) = 32) THEN RETURN
14Ø5Ø  PT = PT + 1
14Ø6Ø  GOTO 14Ø1Ø
```

DECODING A BYTE

```
2ØØØØ  FOR N = Ø TO 1
2ØØ1Ø  D(N) = PEEK(PT + N)
2ØØ2Ø  IF D(N) > 64 THEN D(N) = D(N) - 7
2ØØ3Ø  D(N) = D(N) - 48
2ØØ4Ø  NEXT N
2ØØ5Ø  POKE PB, (D(Ø) * 16 + D(1) )
2ØØ6Ø  RETURN
```

This almost writes itself. We pick up the two alpha codes at PT and PT + 1. These could be 'Ø' to '9' (codes 48 to 57) or 'A' to 'F' (codes 65–7Ø). We now proceed much as in Chapter 2. For convenience, we want 'A' to carry on directly from '9' since it has the value 1Ø, which just means subtracting 7 from any letter. Now any code is just 48 larger than its true value so we subtract 48. Finally we multiply the first value by 16 and add the next to create the decimal equivalent to the hex code, and poke the result to PB.

LABELS

Now all this works like a charm, and you can insert, delete and modify lines to your heart's content, rerun the loader and everything is fine. Well, almost everything. The one remaining problem is that if any branches occur around the edited code, you'll have to alter the branch offsets. Wouldn't it be nice if the loader did that for you?

This implies that any branch instruction must be labelled somehow, and that the address part of the branch also contains a reference to the label. (See Chapter 11 for a discussion of labels in mnemonics.) To make the coding easy, I'm going to put some severe restrictions on the nature of an allowable label:

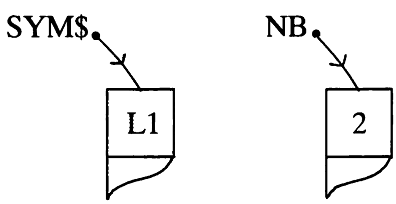
1. It must start with 'L' (I'll relax this restriction later).
2. It must contain exactly two characters.

It's easy to see why I'm making these restrictions. A two-character code looks pretty much like any other byte, so we don't have to muck about with the pointers, but starting with 'L' means it can't be a hex number, so it's easily distinguishable as a label.

Now a piece of code looks like this, for instance:

```
10      : A2 00
20L1    : A0 FF            Branch back to here
30      : BD 00 C0
40      : D0 L1            BNE L1
```

To make this work, we need to make some modifications and additions. First, we now know why there has to be a routine at 16000, to handle the condition that the line to be dealt with has neither a colon nor an asterisk as its first character. Second, the byte decoder (20000) has to be revised to account for a label in the address field. Finally, this routine needs to know what to put there instead, and this implies that we need an extra routine which, before anything else is done, searches through the code for labelled instructions, noting where they are and setting up a couple of arrays to keep a record, like this:



If the 'A2' in the above example is regarded as being in byte 0, then L1 refers to byte 2 (A0). The symbol array (SYM$) contains L1 and the corresponding element of NB (Number of bytes) is 2.

With this arrangement, the modification to the byte decoder is pretty straightforward. Lines 20050–20060 become:

```
20050  CODE = D(0) * 16 + D(1)
20060  IF CODE < = 255 THEN POKE PB, CODE: RETURN
```

Thus, the code to be poked is evaluated as before, but it is possible now for the result to be greater than 255 (FF) if the first character is 'L'. So if we reach 20070 we've found a label:

```
20070  S$ = CHR$(PEEK(PT) ) + CHR$(PEEK(PT + 1) )
20080  FOR L= 0 TO 50
20090  IF SYM$(L) = S$ THEN 20120
20100  NEXT L
20110  PRINT "LABEL"; S$; " NOT FOUND": END
20120  CODE = NB(L) - PB + SA - 1
20130  IF CODE > = 0 THEN POKE PB, CODE
20140  IF CODE < 0 THEN POKE PB, 256 + CODE
20150  RETURN
```

Line 20070 creates the label as a string, which is then searched for in SYM$. When it's found, L points to the number of bytes it is into the code in NB. Line 20120 then evaluates the offset. To take my example, and assuming that the code is loaded from 50000 onwards, we have:

SA = 50000

PB = 50008 (because it's pointing at L1 in line 40)

NB(1) = 2

So CODE = 2 − 50008 + 50000 − 1 = −7, which is the number of bytes to be skipped. However, since this is negative, it can't be poked directly. We have to form its complement by adding 256 (line 20140).

THE DECODE WITH LABEL ROUTINE (16000)

This one's a dolly. All we have to do is move the pointer PT along to the colon and call the 'decode without label' routine:

```
16000  PT = PT + 2
16010  GOSUB 14000
16020  RETURN
```

THE SYMBOL TABLE

All of which just leaves the problem of generating the symbol table (SYM$ and NB) in the first place.

To do this, we must count every byte in the program, which shouldn't be too difficult. The number of bytes per instruction is one more than the number of single spaces.

In outline, the code is going to look much like the decoding suite of routines we've already got, except that no decoding takes place:

```
22ØØØ  DIM SYM$(5Ø), NB(5Ø)
22Ø1Ø  BC = Ø: P = Ø: LS = 2Ø49
22Ø2Ø  GOSUB 24ØØØ: REM COUNT ONE LINE
22Ø3Ø  IF FINISH THEN RETURN
22Ø4Ø  LS = PEEK(LS) + 256 * PEEK(LS + 1)
22Ø5Ø  GOTO 22Ø2Ø
```

COUNTING A LINE (24ØØØ)

```
24ØØØ  PT = LS + 4: FINISH = Ø
24Ø1Ø  IF PEEK(PT) = 172 THEN FINISH = −1: RETURN
24Ø2Ø  IF PEEK(PT) = 58 THEN GOSUB 26ØØØ: RETURN:
       REM NO LABEL
24Ø3Ø  GOSUB 28ØØØ: RETURN: REM LABEL
```

THE 'NO LABEL' CONDITION (26ØØØ)

In this case, all we have to do is count the bytes and increment BC accordingly:

```
26ØØØ  PT = PT + 3
26Ø1Ø  IF PEEK(PT) = Ø THEN BC = BC + 1: RETURN
26Ø2Ø  IF PEEK(PT + 1) = 32 THEN BC = BC + 1: RETURN
26Ø3Ø  BC = BC + 1
26Ø4Ø  GOTO 26ØØØ
```

THE 'LABEL' CONDITION (28ØØØ)

This time we have to record the label and its relative address first, then call the 'No label' routine:

```
28ØØØ  S$ = CHR$(PEEK(PT) ) + CHR$(PEEK(PT + 1) )
```

```
28010  SYM$(P) = S$: NB(P) = BC: P = P + 1
28020  PT = PT + 2
28030  GOSUB 26000
28040  RETURN
```

And that's it! There are a few things to beware of. First, there's almost no error-trapping built in. It would be sensible to write a pre-processing routine which checked the syntax of the code, since an extra space in the wrong place, or a missing colon will confuse the program totally. Second, don't forget to add line:

```
10005  GOSUB 22000: REM SET UP SYMBOL TABLE
```

On the plus side, there are a couple of features which just 'happened'. I can't claim any credit for them, but they're there anyway.

First, you can add two leading spaces after each line number (except when there's a label), so that everything is columnated like this:

```
10     : A2 00          LDX#0
20L1   : A0 FF          LDY#FF
30     : BD 00 C0
40     : D0 L1          Loop back to L1 until zero
```

because BASIC removes them again. That makes it more difficult to make a mistake because everything lines up nicely. Of course, when you list, the extra spaces disappear, but it doesn't matter so much then.

Second, labels don't have to start with 'L'. The real restriction is that they must start with a symbol whose ASCII code is greater than 70, so that the computed 'byte' exceeds 255. This means any letter from 'G' onwards will do. So you can have more labels than you're likely to need, but if you want more than 51 you'll have to redimension SYM$ and NB. Incidentally, there's no check for more than 51 labels either!

To execute MINIASS, RUN 10000. 'GOTO' would work the first time you use it, but not subsequently because you would be re-dimensioning the symbol table. Then enter the start address for the code (not the data area!) in decimal. So for instance, if you have 6 bytes of data from C000 to C005, the start address is C006 = 49158.

PROJECT: SEPARATE ASSEMBLY

Here's an idea for a modification to MINIASS which could prove useful. Suppose you have several separate routines which you want assembled and loaded to a different area of memory.

They could appear like this:

```
 10  :
 20  :
 30  :        first routine
 40  :
 50  *
100  :
110  :        second routine
120  :
130  *
```

and so on.

As things stand, MINIASS can deal with all of them if you assemble the first routine, delete lines 10–50, rerun MINIASS and so on.

It would be nicer, though, if you could enter the starting line number for the routine to be assembled, so that the program would skip to that routine directly. It's easy to do, because we know the bytes which identify the line number are at LS + 2 and LS + 3. The only thing you need to worry about is where to dimension the symbol table!

Notice that, in either event, since the symbol table is re-initialized on every assembly, labels will be treated as local to each routine, so you can re-use label names in successive routines without MINIASS getting confused.

ANOTHER PROJECT

LOADER has several options not available in the current version of MINIASS: listing a Machine Code program to check it is stored correctly in memory; running it; recording it to cassette or disc; loading it back in from cassette or disc. You can easily pinch the relevant routines from LOADER and combine them with MINIASS to get a really versatile utility. The main item it lacks is something that will automatically assemble mnemonics into hex. If you're prepared to type in a table of all 151 mnemonics together with their opcodes; the number of bytes they require; and to add a few book-keeping routines, you can remedy this yourself. It's a nice project for a rainy February.

There's much more to Machine Code than I've been able to tackle here, of course; this is just a start. To go much further you'll need more sophisticated software aids—hand-assembly isn't really sensible for complicated programs. I'd like to finish by tidying up:

22 Some Loose Ends

There are a number of odd points and ideas that I haven't been able to make yet but shouldn't finish without mentioning. The first is:

SAVING MACHINE CODE AND BASIC IN ONE GO

Here's one way. Include the above two routines in your BASIC program. SAVE the BASIC program first, then use the 'F' option to save the Machine Code as a second file. In reverse: LOAD the BASIC; take option 'I', and input the file as a secondary stage.

An alternative is to POKE the system variables that determine where the BASIC program and variables areas go, to fool the Sixty-four into opening up a gap to put your Machine Code into. Transfer it down to that region. Now a simple SAVE will save BASIC *and* Machine Code in one go. See the *Reference Guide*, page 312 for the addresses of the system variables. (If you're *that* serious about programming, you'll already have bought it!)

RELOCATABLE CODE

All that talk of moving code around brings us to another topic. If you write code that avoids absolute addresses, it can be transferred to another region of memory without any problems. This is known as *relocatable* code. You can usually avoid JMP altogether; but JSR does pose problems so you may have to do a little editing on the code before or after transfer.

THE KERNAL

You can JSR to any routine in the Sixty-four's ROM: all you need to know is the address involved in a particular routine, and what state the registers must be in first.

Particularly useful are the Input/Output routines, which are available through a program called the KERNAL which sits in memory from E000 to FFFF. A full description may be found in the *Reference Guide*: a few routines that you may be especially interested in are described here.

Some KERNAL routines require other *preparatory routines* first. You must call those before you call the main routine.

CHKIN: Address FFC6
This opens a channel for input. The logical file number has to be put in the X-register. Unless you intend to use the keyboard as communication device you must use OPEN as a preparatory routine.

CHKOUT: Address FFC9
This works the same way as CHKIN, but for output.

CHRIN: Address FFCF
This gets a character from the input channel and puts it in the accumulator. Unless you are using the keyboard, preparatory routines OPEN and CHKIN are required. The X-register is used.

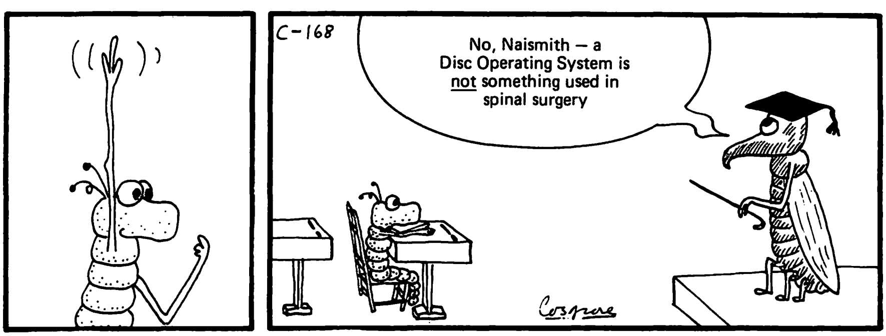
CHROUT: Address FFD2
This is like CHRIN but for output. Preparatory routines OPEN and CHKOUT, except for TV screen output.

CLALL: Address FFE7
This closes all files. Registers A, X are affected.

CLOSE: Address FFC3
This closes a single file. Load the logical file number into the accumulator. The X and Y registers will be affected.

GETIN: Address FFE4
Gets a character from keyboard (no preparatory routines, but note that it uses the keyboard buffer), or other input device (preparatory routines CHKIN, OPEN). It puts the character in the accumulator.

OPEN: Address FFC0
Opens a logical file. Preparatory routines SETLFS and SETNAM must be used.



PLOT: Address FFFØ
This sets the screen cursor position. The column number should go in the X-register, the row in the Y-register. The accumulator is used.

SETLFS: Address FFBA
Sets up a logical file. Load accumulator with logical file number, X-register with device number, Y-register with command (255 default). The device numbers are:

Ø	Keyboard
1	Cassette recorder
2	RS-232C device
3	TV display
4	Serial bus printer
5	Serial bus disc drive.

SETNAM: Address FFBD
Sets up a file name. The length of the name goes in the accumulator, and the X- and Y-registers get the junior and senior bytes of the address where the name starts in RAM.

For example, suppose you want to PRINT a character to the screen at row 7, column 5. Suppose the character is 'X' with ASCII code 58 hex. First you position the cursor at row 7, column 5, using PLOT:

```
LDX #Ø5        A2 Ø5
LDY #Ø7        AØ Ø7
JSR PLOT       2Ø FØ FF
```

Now you use CHROUT to output the character:

```
LDA #58        A9 58
JSR CHROUT     2Ø D2 FF
```

The CHROUT routine automatically updates the cursor position. So to print 'FRED' on the screen, you can use:

```
LDA #46        A9 46        ASCII for 'F'
JSR CHROUT     2Ø D2 FF
LDA #52        A9 52        ASCII for 'R'
JSR CHROUT     2Ø D2 FF
LDA #45        A9 45        ASCII for 'E'
JSR CHROUT     2Ø D2 FF
LDA #44        A9 44        ASCII for 'D'
JSR CHROUT     2Ø D2 FF
```

Don't forget to add the final:

```
        RTS             60
```

if you try these out.

ASSEMBLERS

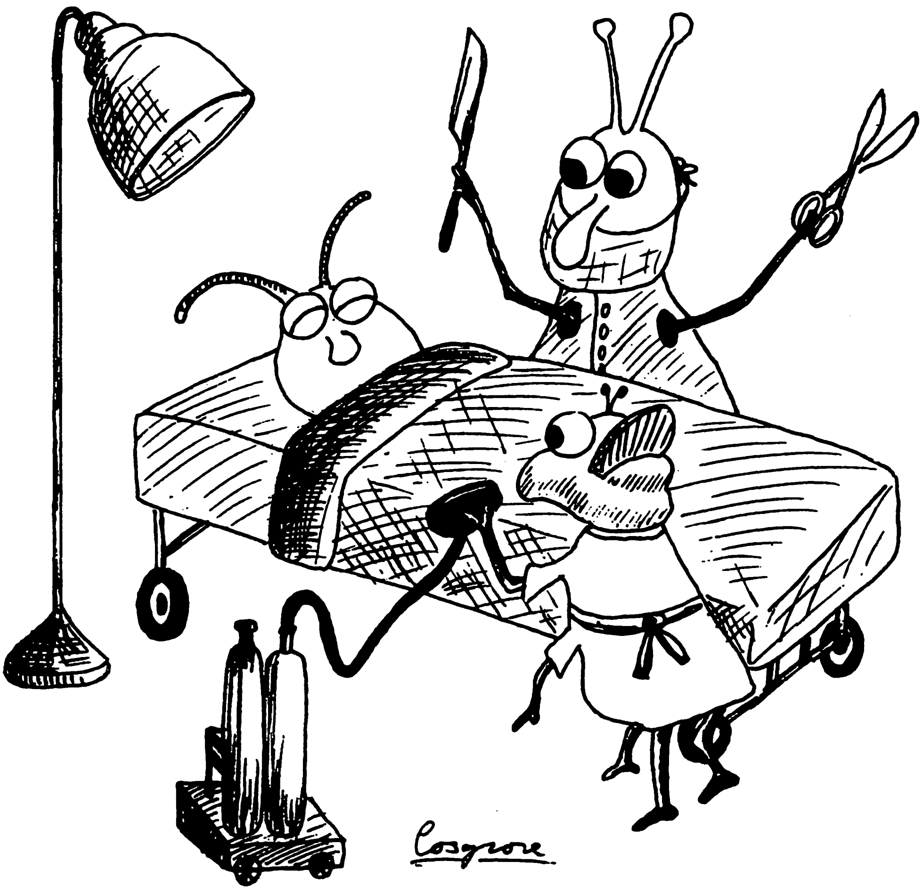
To help you edit and load Machine Code, there are a number of commercial assemblers available. These let you write in assembly mnemonics, using labels, etc., and convert to hex automatically.

One point to note is that almost all of them are *slow* to use, because they tend to come in sections that have to be loaded into the computer one at a time . . . load editing program; edit assembly code; save assembly code to a file; load assembler; read in assembly code from file; output hex code to another file; read that file in; load in place; and execute. For short programs, an assembler isn't really much use.

For longer programs, on the other hand, an assembler is a must. Commodore produces one called 64MON; several others are commercially available.

You could even write your own!

Appendices



1 Hex/Decimal Conversion

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
A	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
B	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
C	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
D	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
E	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
F	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

2's complement

ordinary

2 Mnemonics

ADC	Add with Carry
AND	Logical AND on each bit
ASL	Arithmetic Shift Left
BCC	Branch if Carry Clear
BCS	Branch if Carry Set
BEQ	Branch if result zero
BIT	Test bits from memory
BMI	Branch if minus (signed arithmetic)
BNE	Branch if result non-zero
BPL	Branch if plus (signed arithmetic)
BRK	Force break
BVC	Branch if overflow clear
BVS	Branch if overflow set
CLC	Clear Carry flag
CLD	Clear decimal mode flag
CLI	Clear interrupt disable bit
CLV	Clear overflow flag
CMP	Compare accumulator with memory
CPX	Compare index X with memory
CPY	Compare index Y with memory
DEC	Decrement by 1
DEX	Decrement index X
DEY	Decrement index Y
EOR	Exclusive OR
INC	Increment by 1
INX	Increment index X
INY	Increment index Y.
JMP	Jump to absolute address (or indirect address)
JSR	Jump to subroutine
LDA	Load accumulator
LDX	Load index X
LDY	Load index Y
LSR	Logical Shift Right
NOP	No operation
ORA	Logical OR

PHA	Push accumulator to stack
PHP	Push processor status to stack
PLA	Pull accumulator from stack
PLP	Pull processor status from stack
ROL	Rotate left one bit
ROR	Rotate right one bit
RTI	Return from interrupt
RTS	Return from subroutine
SBC	Subtract with Borrow
SEC	Set Carry flag
SED	Set decimal mode
SEI	Set interrupt disable status
STA	Store accumulator to memory
STX	Store index X
STY	Store index Y
TAX	Transfer accumulator to index X
TAY	Transfer accumulator to index Y
TSX	Transfer stack pointer to index X
TXA	Transfer index X to accumulator
TXS	Transfer index X to stack register
TYA	Transfer index Y to accumulator

3 Summary of Addressing Modes and Mnemonic Formats

Symbols used in this appendix:

MOP	Mnemonic for operation (e.g. STA)
jj	Zero page address (junior byte)
jj ss	Non-zero page address (junior byte, senior byte)
dd	Relative displacement: signed binary between −128 and 127
nn	Number byte
( )	Indirection
,X	Using X index
,Y	Using Y index

Implied and Accumulator addressing Either no address required, or the accumulator assumed. Format:

MOP

Immediate addressing Numerical data, not an address. Format:

MOP #nn

Absolute (non-zero page) addressing Using a two-byte address. Format:

MOP jj ss

Zero-page addressing Using a single byte to specify address on page ØØ. Format:

MOP jj

Pre-indexed by X Indirection via a byte whose address on page zero is the specified byte plus the contents of the X-register. Format:

MOP (jj,X)

Post-indexed by Y Indirection via an address on page zero, to which the contents of the Y-register are added. Format:

MOP (jj),Y

Indexed (four kinds) Zero or non-zero page, X or Y register used as index byte. Formats:

MOP jj, X
MOP jj, Y
MOP jj ss, X
MOP jj ss, Y

Indirect Two bytes specify the address at which the effective address may be found. Only used by JMP. Format:

MOP (jj ss)

Relative Signed displacement byte to be added to the Program Counter, used for branching. Format:

MOP dd

4 6510 Opcodes

This table shows all of the opcodes for the 6510 microprocessor, listed alphabetically by mnemonic, in all available addressing modes:

	Implied	Immediate	Absolute non-zero	Zero-page	Pre-indexed X	Post-indexed Y	Indexed X non-zero	Indexed X zero-page	Indexed Y non-zero	Indexed Y zero-page	Indirect	Relative
ADC	—	69	6D	65	61	71	7D	75	79	—	—	—
AND	—	29	2D	25	21	31	3D	35	39	—	—	—
ASL	ØA	—	ØE	Ø6	—	—	1E	16	—	—	—	—
BCC	—	—	—	—	—	—	—	—	—	—	—	9Ø
BCS	—	—	—	—	—	—	—	—	—	—	—	BØ
BEQ	—	—	—	—	—	—	—	—	—	—	—	FØ
BIT	—	—	2C	24	—	—	—	—	—	—	—	—
BMI	—	—	—	—	—	—	—	—	—	—	—	3Ø
BNE	—	—	—	—	—	—	—	—	—	—	—	DØ
BPL	—	—	—	—	—	—	—	—	—	—	—	1Ø
BRK	ØØ	—	—	—	—	—	—	—	—	—	—	—
BVC	—	—	—	—	—	—	—	—	—	—	—	5Ø
BVS	—	—	—	—	—	—	—	—	—	—	—	7Ø
CLC	18	—	—	—	—	—	—	—	—	—	—	—
CLD	D8	—	—	—	—	—	—	—	—	—	—	—
CLI	58	—	—	—	—	—	—	—	—	—	—	—
CLV	B8	—	—	—	—	—	—	—	—	—	—	—
CMP	—	C9	CD	C5	C1	D1	DD	D5	D9	—	—	—
CPX	—	EØ	EC	E4	—	—	—	—	—	—	—	—
CPY	—	CØ	CC	C4	—	—	—	—	—	—	—	—
DEC	—	—	CE	C6	—	—	DE	D6	—	—	—	—
DEX	CA	—	—	—	—	—	—	—	—	—	—	—
DEY	88	—	—	—	—	—	—	—	—	—	—	—
EOR	—	49	4D	45	41	51	5D	55	59	—	—	—
INC	—	—	EE	E6	—	—	FE	F6	—	—	—	—

	Implied	Immediate	Absolute non-zero	Zero-page	Pre-indexed X	Post-indexed Y	Indexed X non-zero	Indexed X zero-page	Indexed Y non-zero	Indexed Y zero-page	Indirect	Relative
INX	E8	—	—	—	—	—	—	—	—	—	—	—
INY	C8	—	—	—	—	—	—	—	—	—	—	—
JMP	—	—	4C	—	—	—	—	—	—	—	6C	—
JSR	—	—	2Ø	—	—	—	—	—	—	—	—	—
LDA	—	A9	AD	A5	A1	B1	BD	B5	B9	—	—	—
LDX	—	A2	AE	A6	—	—	—	—	BE	B6	—	—
LDY	—	AØ	AC	A4	—	—	BC	B4	—	—	—	—
LSR	4A	—	4E	46	—	—	5E	56	—	—	—	—
NOP	EA	—	—	—	—	—	—	—	—	—	—	—
ORA	—	Ø9	ØD	Ø5	Ø1	11	1D	15	19	—	—	—
PHA	48	—	—	—	—	—	—	—	—	—	—	—
PHP	Ø8	—	—	—	—	—	—	—	—	—	—	—
PLA	68	—	—	—	—	—	—	—	—	—	—	—
PLP	28	—	—	—	—	—	—	—	—	—	—	—
ROL	2A	—	2E	26	—	—	3E	36	—	—	—	—
ROR	6A	—	6E	66	—	—	7E	76	—	—	—	—
RTI	4Ø	—	—	—	—	—	—	—	—	—	—	—
RTS	6Ø	—	—	—	—	—	—	—	—	—	—	—
SBC	—	E9	ED	E5	E1	F1	FD	F5	F9	—	—	—
SEC	38	—	—	—	—	—	—	—	—	—	—	—
SED	F8	—	—	—	—	—	—	—	—	—	—	—
SEI	78	—	—	—	—	—	—	—	—	—	—	—
STA	—	—	8D	85	81	91	9D	95	99	—	—	—
STX	—	—	8E	86	—	—	—	—	—	96	—	—
STY	—	—	8C	84	—	—	—	94	—	—	—	—
TAX	AA	—	—	—	—	—	—	—	—	—	—	—
TAY	A8	—	—	—	—	—	—	—	—	—	—	—
TSX	BA	—	—	—	—	—	—	—	—	—	—	—
TXA	8A	—	—	—	—	—	—	—	—	—	—	—
TXS	9A	—	—	—	—	—	—	—	—	—	—	—
TYA	98	—	—	—	—	—	—	—	—	—	—	—

5 Effect of Operations on Flags

This table lists, for all operations that affect the Processor Status Register (flags), what the effect is. Operations not listed have no effect.

*	Set or reset according to result of operation
Ø	Always reset to Ø
1	Always set to 1
7	Bit 7 of the byte involved
6	Bit 6 of the byte involved

Operation	N	V	D	I	Z	C
ADC	*	*			*	*
AND	*				*	
ASL	*				*	*
BIT	7	6			*	
†BRK				*		
CLC						Ø
CLD			Ø			
CLI				Ø		
CLV		Ø				
CMP	*				*	*
CPX	*				*	*
CPY	*				*	*
DEC	*				*	
DEX	*				*	
DEY	*				*	
EOR	*				*	
INC	*				*	
INX	*				*	
INY	*				*	
LDA	*				*	
LDX	*				*	
LDY	*				*	
LSR	Ø				*	*
ORA	*				*	
PLA	*				*	
PLP	*	*	*	*	*	*
ROL	*				*	*

† BRK also sets the B flag.

Operation	N	V	D	I	Z	C
ROR	*				*	*
RTI	*	*	*	*	*	*
SBC	*	*			*	*
SEC						1
SED			1			
SEI				1		
TAX	*				*	
TAY	*				*	
TSX	*				*	
TXA	*				*	
TYA	*				*	

6 Opcodes in Numerical Order for Disassembly

This uses the same symbols as Appendix 3.

00	BRK
01	ORA (jj, X)
05	ORA jj
06	ASL jj
08	PHP
09	ORA #nn
0A	ASL
0D	ORA jj ss
0E	ASL jj ss
10	BPL dd
11	ORA (jj), Y
15	ORA jj, X
16	ASL jj, X
18	CLC
19	ORA jj ss, Y
1D	ORA jj ss, X
1E	ASL jj ss, X
20	JSR jj ss
21	AND (jj, X)
24	BIT jj
25	AND jj
26	ROL jj
28	PLP
29	AND #nn
2A	ROL
2C	BIT jj ss
2D	AND jj ss
2E	ROL jj ss
30	BMI dd
31	AND (jj), Y
35	AND jj, X
36	ROL jj, X
38	SEC
39	AND jj ss, Y
3D	AND jj ss, X
3E	ROL jj ss, X
40	RTI
41	EOR (jj, X)
45	EOR jj
46	LSR jj
48	PHA
49	EOR #nn
4A	LSR
4C	JMP jj ss
4D	EOR jj ss
4E	LSR jj ss
50	BVC dd
51	EOR (jj), Y
55	EOR jj, X
56	LSR jj, X
58	CLI
59	EOR jj ss, Y
5D	EOR jj ss, X
5E	LSR jj ss, X
60	RTS
61	ADC (jj, X)
65	ADC jj
66	ROR jj
68	PLA
69	ADC #nn
6A	ROR
6C	JMP (jj ss)
6D	ADC jj ss
6E	ROR jj ss
70	BVS dd
71	ADC (jj), Y
75	ADC jj, X
76	ROR jj, X
78	SEI
79	ADC jj ss, Y
7D	ADC jj ss, X
7E	ROR jj ss, X
81	STA (jj, X)
84	STY jj
85	STA jj
86	STX jj
88	DEY
8A	TXA
8C	STY jj ss
8D	STA jj ss
8E	STX jj ss
90	BCC dd
91	STA (jj), Y
94	STY jj, X
95	STA jj, X
96	STX jj, Y
98	TYA
99	STA jj ss, Y
9A	TXS
9D	STA jj ss, X
A0	LDY #nn
A1	LDA (jj, X)
A2	LDX #nn
A4	LDY jj
A5	LDA jj
A6	LDX jj
A8	TAY
A9	LDA #nn
AA	TAX
AC	LDY jj ss
AD	LDA jj ss
AE	LDX jj ss
B0	BCS dd
B1	LDA (jj), Y
B4	LDY jj, X
B5	LDA jj, X
B6	LDX jj, Y
B8	CLV
B9	LDA jj ss, Y
BA	TSX
BC	LDY jj ss, X
BD	LDA jj ss, X
BE	LDX jj ss, Y
C0	CPY #nn
C1	CMP (jj, X)
C4	CPY jj
C5	CMP jj
C6	DEC jj
C8	INY
C9	CMP #nn
CA	DEX
CC	CPY jj ss
CD	CMP jj ss
CE	DEC jj ss
D0	BNE dd
D1	CMP (jj), Y
D5	CMP jj, X
D6	DEC jj, X
D8	CLD
D9	CMP jj ss, Y
DD	CMP jj ss, X
DE	DEC jj ss, X
E0	CPX #nn
E1	SBC (jj, X)
E4	CPX jj
E5	SBC jj
E6	INC jj
E8	INX
E9	SBC #nn
EA	NOP
EC	CPX jj ss
ED	SBC jj ss
EE	INC jj ss
F0	BEQ dd
F1	SBC (jj), Y
F5	SBC jj, X
F6	INC jj, X
F8	SED
F9	SBC jj ss, Y
FD	SBC jj ss, X
FE	INC jj ss, X

7 Sprite Registers Made Easy

<table>
<tr><th>Address</th><th colspan="8">Contents</th><th>Function</th></tr>
<tr><td>DØØØ</td><td colspan="8">Sprite Ø column number</td><td rowspan="16">Sprite positions</td></tr>
<tr><td>DØØ1</td><td colspan="8">Sprite Ø row number</td></tr>
<tr><td>DØØ2</td><td colspan="8">Sprite 1 column number</td></tr>
<tr><td>DØØ3</td><td colspan="8">Sprite 1 row number</td></tr>
<tr><td>DØØ4</td><td colspan="8">Sprite 2 column number</td></tr>
<tr><td>DØØ5</td><td colspan="8">Sprite 2 row number</td></tr>
<tr><td>DØØ6</td><td colspan="8">Sprite 3 column number</td></tr>
<tr><td>DØØ7</td><td colspan="8">Sprite 3 row number</td></tr>
<tr><td>DØØ8</td><td colspan="8">Sprite 4 column number</td></tr>
<tr><td>DØØ9</td><td colspan="8">Sprite 4 row number</td></tr>
<tr><td>DØØA</td><td colspan="8">Sprite 5 column number</td></tr>
<tr><td>DØØB</td><td colspan="8">Sprite 5 row number</td></tr>
<tr><td>DØØC</td><td colspan="8">Sprite 6 column number</td></tr>
<tr><td>DØØD</td><td colspan="8">Sprite 6 row number</td></tr>
<tr><td>DØØE</td><td colspan="8">Sprite 7 column number</td></tr>
<tr><td>DØØF</td><td colspan="8">Sprite 7 row number</td></tr>
<tr><td>DØ1Ø</td><td>Sp 7</td><td>Sp 6</td><td>Sp 5</td><td>Sp 4</td><td>Sp 3</td><td>Sp 2</td><td>Sp 1</td><td>Sp Ø</td><td>Offset flag</td></tr>
<tr><td>DØ15</td><td>Sp 7</td><td>Sp 6</td><td>Sp 5</td><td>Sp 4</td><td>Sp 3</td><td>Sp 2</td><td>Sp 1</td><td>Sp Ø</td><td>Enable/disable</td></tr>
<tr><td>DØ17</td><td>Sp 7</td><td>Sp 6</td><td>Sp 5</td><td>Sp 4</td><td>Sp 3</td><td>Sp 2</td><td>Sp 1</td><td>Sp Ø</td><td>Expand vertically</td></tr>
<tr><td>DØ1D</td><td>Sp 7</td><td>Sp 6</td><td>Sp 5</td><td>Sp 4</td><td>Sp 3</td><td>Sp 2</td><td>Sp 1</td><td>Sp Ø</td><td>Expand horizontally</td></tr>
<tr><td>DØ1E</td><td>Sp 7</td><td>Sp 6</td><td>Sp 5</td><td>Sp 4</td><td>Sp 3</td><td>Sp 2</td><td>Sp 1</td><td>Sp Ø</td><td>Collision flag</td></tr>
<tr><td>DØ27</td><td colspan="8">Sprite Ø colour code</td><td rowspan="8">Colours</td></tr>
<tr><td>DØ28</td><td colspan="8">Sprite 1 colour code</td></tr>
<tr><td>DØ29</td><td colspan="8">Sprite 2 colour code</td></tr>
<tr><td>DØ2A</td><td colspan="8">Sprite 3 colour code</td></tr>
<tr><td>DØ2B</td><td colspan="8">Sprite 4 colour code</td></tr>
<tr><td>DØ2C</td><td colspan="8">Sprite 5 colour code</td></tr>
<tr><td>DØ2D</td><td colspan="8">Sprite 6 colour code</td></tr>
<tr><td>DØ2E</td><td colspan="8">Sprite 7 colour code</td></tr>
<tr><td>Ø7F8</td><td colspan="8">Sprite Ø data pointer</td><td rowspan="8">Pointers</td></tr>
<tr><td>Ø7F9</td><td colspan="8">Sprite 1 data pointer</td></tr>
<tr><td>Ø7FA</td><td colspan="8">Sprite 2 data pointer</td></tr>
<tr><td>Ø7FB</td><td colspan="8">Sprite 3 data pointer</td></tr>
<tr><td>Ø7FC</td><td colspan="8">Sprite 4 data pointer</td></tr>
<tr><td>Ø7FD</td><td colspan="8">Sprite 5 data pointer</td></tr>
<tr><td>Ø7FE</td><td colspan="8">Sprite 6 data pointer</td></tr>
<tr><td>Ø7FF</td><td colspan="8">Sprite 7 data pointer</td></tr>
</table>

8 Keyboard Scan Codes

This lists the contents of address 197 (ØØC5 hex) when a given key is pressed. Using PEEK(197) or STA C5 (Opcode 85 C5) permits detection of the key currently held down, bypassing the keyboard buffer.

Key	Code	Hex	Key	Code	Hex	Key	Code	Hex
(none)	64	4Ø		46	2E	T	22	16
*	49	31	A	1Ø	ØA	U	3Ø	1E
+	4Ø	28	B	28	1C	V	31	1F
,	47	2F	C	2Ø	14	W	9	Ø9
—	43	2B	D	18	12	X	23	17
.	44	2C	E	14	ØE	Y	25	19
/	55	37	F	21	15	Z	12	ØC
Ø	35	23	G	26	1A	RETURN	1	Ø1
1	56	38	H	29	1D	CLR/HOME	51	33
2	59	3B	I	33	21	INST/DEL	Ø	ØØ
3	8	Ø8	J	34	22	CRSR ↑↓	7	Ø7
4	11	ØB	K	37	25	CRSR →←	2	Ø2
5	16	1Ø	L	42	2A	←	57	39
6	19	13	M	36	24	f1	4	Ø4
7	24	18	N	39	27	f3	5	Ø5
8	27	1B	O	38	26	f5	6	Ø6
9	32	2Ø	P	41	29	f7	3	Ø3
:	45	2D	Q	62	3E	£	48	3Ø
;	5Ø	32	R	17	11			
=	53	35	S	13	ØD			

ORDER FORM

I should like to order the following Shiva titles:

Qty	Title	ISBN	Price
____	EASY PROGRAMMING FOR THE COMMODORE 64	0 906812 64 X	£6.95
____	THE COMMODORE 64 MUSIC BOOK	1 85014 019 7	£5.95
____	COMMODORE 64 ASSEMBLY LANGUAGE	0 906812 96 8	£7.95
____	GATEWAY TO COMPUTING WITH THE COMMODORE 64: BOOK ONE	1 85014 051 0 1 85014 017 0	£6.95 (hdbk) £4.95 (pbk)
____	GATEWAY TO COMPUTING WITH THE COMMODORE 64: BOOK TWO	1 85014 055 3 1 85014 035 9	£6.95 (hdbk) £4.95 (pbk)
____	BRAINTEASERS FOR BASIC COMPUTERS	0 906812 36 4	£4.95
____	....	....	....
____	....	....	....
____	....	....	....

Please send me a full catalogue of computer books and software: ☐

Name ....

Address ....

....

....

This form should be taken to your local bookshop or computer store. In case of difficulty, write to Shiva Publishing Ltd, Freepost, 64 Welsh Row, Nantwich Cheshire CW5 5BR, enclosing a cheque for £ ....

For payment by credit card: Access/Barclaycard/Visa/American Express

Card No .... Signature ....

Is BASIC programming on your 64 too limited for you? Then take a step beyond and get to know the core language of your Commodore: Machine Code.

This book will introduce you gently, but thoroughly, to the fundamentals of Machine Code programming. In no time at all, you will be exploring the possibilities of:

- Sprites
- Colour
- Keyboard control
- Moving graphics
- High and low resolution displays

Numerous appendices will help you to develop and build up your Machine Code. The 64's speed and versatility will astound you, as will your ability to exploit and manipulate them.

Accept the challenge and you will certainly reap the rewards!

Shiva Publishing Limited

GB £ NET +006.95

ISBN 1-85014-025-1

00695

9 781850 140252