The Complete Commodore Machine Code Programming Course

For the CBM 64 and 128

Includes an Assembler Program

Andrew Bennett
and
Surya

The Complete Commodore Machine Code Programming Course

The Complete Commodore Machine Code Programming Course

ANDREW BENNETT
and
SURYA

LONDON
Chapman and Hall/Methuen
NEW YORK

First published in 1986 by
Chapman and Hall Ltd/Methuen London Ltd
11 New Fetter Lane, London EC4P 4EE
Published in the USA by
Chapman and Hall/Methuen Inc
29 West 35th Street, New York NY 10001

Printed in Great Britain by J. W. Arrowsmith Ltd, Bristol

ISBN 0 412 27250 4

British Library Cataloguing in Publication Data

Bennett, Andrew
The complete Commodore machine code programming course: includes an assembler program for the CBM 64 and 128.
1. Commodore computers—Programming 2. Machine codes (Electronic computers) 3. Microcomputers —Programming
I. Title II. Surya
005.2'6 QA76.8.C6

ISBN 0–412–27250–4

Library of Congress Cataloging in Publication Data

Bennett, Andrew, 1964–
The complete Commodore machine code programming course.
(Chapman and Hall computing)
Bibliography: p.
Includes index.
1. Commodore 64 (Computer)—Programming.
2. Commodore 128 (Computer)—Programming. 3. Assembler language (Computer program language)
I. Surya, 1963– . II. Title. III. Series.
QA76.8.C64B46 1986 005.265 86–6797
ISBN 0–412–27250–4

Contents

Preface

There is a perpetual question faced by anybody wanting to present a technical subject to an audience unfamiliar with it: do you choose someone expert in the subject and hope that they will be able to translate their knowledge into plain English, or do you choose a professional communicator and hope that they have a sufficient grasp of the subject to get their facts right? That a large proportion of computer books are either badly written or technically inaccurate is testimony to the fact that neither approach offers a real solution.

This book takes a different approach. Instead of choosing between the machine code programming expert and the professional communicator, we've used both. Andrew is a professional programmer on the CBM 64, while I'm a computer journalist and consultant by profession. We believe that, by working together, we've come up with a guide which is far easier to use than any other on the same subject. We hope you'll agree.

London, March 1986

Surya

1

Learning machine code

Learning machine code is easy.

Machine code is often seen, felt or talked about as something complicated, mysterious and reserved for whizz-kids. When you see machine code listings in books and magazines, the code and the jargon often appear totally incomprehensible. You may have looked at other books or articles, supposedly written for complete beginners, and found them bewildering and confusing. You may have talked to experienced machine code programmers and ended up wondering if they are from the same planet or from one with a totally different language. Don't worry: you're not alone!

Most people find machine code confusing when they first encounter it. It's perfectly natural: you're learning a new language. You have probably forgotten how difficult and confusing BASIC seemed when you first began to learn it. Yet within a week or so you were quite happily writing your own programs and discovering more and more about the language. Machine code is no different. It may seem a little complicated to begin with, but you will be surprised when you realize how quickly your understanding has grown.

We don't know just how quickly or easily you will learn. What we do know is that all it takes is time and patience, plus a little effort. This book was written with the complete beginner in mind. We do, obviously, assume that you are reasonably familiar with BASIC, but we do not assume any prior knowledge or experience of machine code. We start right at the beginning.

We have designed this course with 'ease of use' as our prime concern. And you can make your learning even easier by following a few simple guidelines:

● **Take things slowly**

We know that you will probably be eager to dive in and start writing machine code masterpieces, but the time you spend covering the basics will make life a hundred times easier when you start learning more exciting techniques.

● **When in doubt, go back and re-read**
Again, it is very tempting to rush straight on to the next chapter as soon as you reach the end of the previous one. Don't! Your understanding will be much deeper and clearer if you take the time to make sure that you have understood everything in the chapter.

● **Reinforce your understanding**
We have already suggested re-reading any bits you are not sure about, but we also recommend that you regularly go back to earlier chapters to make sure you have not forgotten any of the earlier material. Machine code is very much interconnected, so this can be an extremely valuable way of developing your knowledge of the subject. You will find that the basics of the subject will make much more sense to you when you come back to them for a second and third time after covering some more advanced material.

● **Do the exercises!**
The exercises have been carefully designed to test your understanding and to give you the chance to put your new skills into practice. You will learn much more easily and effectively if you complete them, going back into the chapter for help as you need it. Just reading the exercise and then looking up the answer won't do you any good at all. Doing is the easiest way of learning.

● **Practice!**
As well as the exercises we have set, try out your own ideas along the way. You may find that many of your ideas don't work: that's fine – you'll actually learn more from finding out why they don't work!

● **Type in the example programs**
We know that typing in programs can be boring. It can also be extremely interesting if you pay attention as you enter them. You might wonder about how they work before reading the full explanation. We have deliberately kept these programs short to make them easy to enter, so please do take the trouble: the practice is well worth it. One of the main reasons for supplying an assembler as part of this course was because we intend the book to be used beside the computer, trying things out as you go along. If you learnt BASIC that way, you will know what a good system it is.

● **Enjoy it!**
Machine code is not some kind of a test. It is a tool designed to enable you to get even more power and enjoyment from your C64. Experiment. Have fun. Enjoy your new-found skills.

1.1 A word about the C128

The Commodore 128 is 100% upward-compatible with the Commodore 64. This means that ***all*** C64 programs, BASIC or machine code, will run on the C128. The assembler supplied with this book, and all the demonstration and example programs, will run perfectly on the C128 without modification. All references to the C64 should be taken to mean C64 or C128.

1.2 The assembler

In order to write machine code programs on your C64, you need a special program called an assembler. These normally cost between £8 and £50, and each one is slightly different from the rest. This means that learning machine code using a separate book and assembler is not only expensive, it is also hard work because the programs in the book may not work without modification on the assembler you have bought. We thought that this was not particularly helpful, so, with the willing co-operation of Chapman and Hall, the book's publisher, we decided to include an assembler with the book. Since the assembler and the book were written together, the two are, of course, 100% compatible.

The assembler supplied with this book was written in BASIC, with machine code subroutines: a technique you will be able to use in your own programs by the time you have completed this course! When you've typed it in, or loaded it from the optional tape available, it's ready to do its work. Not all of the explanation given below will make sense to you yet. Those of you with some experience should read it carefully, to note any differences between our assembler and the one you have used before. Beginners can just read quickly through it, referring back to it as required.

To type a machine code program into your C64, load the assembler and then enter the machine code just like a BASIC program except that the ***line numbers must be above 5000***. All the example programs in this book have line numbers of 6000 or over. If you accidentally enter anything with a line number of less than 5000, the assembler may be overwritten so enter **NEW** and then load it again. When you enter machine code into your C64, the code you type is called the source code. Once this has been assembled (we will

explain what this means later in the book), the resulting program is called the object code. A golden rule of writing machine code is: **always save your source code before assembling the program.** That way, if your machine code program crashes and causes the C64 to 'hang-up', you can just switch off and reload your source code to correct the mistake.

To enter source code, the first line of your program needs to tell the assembler whereabouts in memory to store the program. This is done by telling it the start address, and is done like so

5000 [⟨start address⟩

with one space between the left square bracket and the address. The normal start address is **$C000**, so you would enter

5000 [$C000

or, if you want to use decimal numbering

5000 [49152

All this is explained properly later in the book, but those with some machine code experience might like to note that the acceptable range of start addresses for this assembler is **$C000** to **$D000.**

To make life easy, the assembler can work with decimal, hex and binary numbers (see Chapters 2 and 3). It can also use ASCII codes. Normally, the assembler will assume that you are working in decimal. If you want to use hex, binary or ASCII, you tell the assembler this by putting a prefix in front of the number.

% = binary, e.g. **% 10011011**
$** = hex, e.g. **$A5
' = ASCII, e.g. **'E** means the ASCII code of **E** (NB: **'** = single quote mark)

If a number does not have any of these prefixes, the assembler will treat it as a decimal number.

The syntax for a line of assembler code is

⟨line number⟩ ⟨instruction⟩ ⟨number⟩ :⟨label⟩ ;⟨remark⟩

An example of this would be

6000 LDA $5000 :LOOP ;LOOP BACK TO HERE

Not all machine code instructions require a number (as we will see later), and labels and remarks are optional. Labels can be of any length, but must start with a colon. Remarks can be of any length, but must start with a semicolon. The total length of the line, however, must not be greater than 80 characters – just as in BASIC.

If a line does not have a label, the remark can go straight after the number

6000 LDA $5000 ;Like so

Constant labels (as opposed to line labels) must be on a line of their own. The format is-

⟨line number⟩:⟨label⟩=⟨value⟩

For example

```
6000 :COLOUR1=$07
```

You are also allowed simple maths

```
6010 :COLOUR2=COLOUR1+1
6020 :COLOUR3=COLOUR1-1
```

This assembler has three aditional commands, which we have called dot commands because they are preceded by a dot! You will not find these commands in a normal assembler.

The first dot command is **BYT**. This places single-byte numbers of up to 255 in memory after your machine code program. Thus

```
7010 .BYT $50 :DATA
```

would place the number **$50** in memory and label it **DATA**, so that

```
7020 LDA DATA
```

would place **$50** into the accumulator.

The second dot command, **WOR**, is the same as **BYT** except that it allows you to place numbers of up to 65535 into memory

```
7010 .WOR $0400 :DATA
```

Finally, **.TXT** places a string into memory

```
7030 .TXT "CBM 64" :LOGO
```

Note that dot commands must be on a line of their own, with one dot command per line.

Finally, you must end your source code (the program you type in) with **END** on a line of its own. This is to let the assembler know where the program ends. When you have finished entering your source code, just enter **RUN** to assemble it into object code. When assembly is complete, the assembler will print out the start and end addresses so that you can save the object code.

So, without further ado, let's begin at the beginning . . .

2

What is machine code?

This chapter will give you an overall understanding of what we mean by machine code, how it differs from BASIC and the purpose of programming in it. When you have read it, you should be able to answer the following questions:

- What is machine code?
- What is the point of programming in machine code?
- What is assembly language?
- What is hex?

When you first bought your C64, the salesperson probably told you that its natural programming language was BASIC. In fact, the C64's native language is machine code. (BASIC itself is actually written in machine code, but we will talk about this further on.) Later you will understand why we say this, but before then we have to explain exactly what we mean by machine code.

2.1 Programming in BASIC

To understand what we mean by machine code programming, we need to look firstly at what happens when we program in BASIC. In BASIC, we use variables to store and calculate values. To calculate the number of hours someone has been alive, for example, we would write a program something like that shown in Listing 2.1.

Listing 2.1

```
10 REM BASIC program 1
20 REM Calculates approx. age in hours
30 PRINT "[CLR]"
40 INPUT "Please enter your age in years";YEAR
50 INPUT "       and months";              MONTH
60 LET DAY(1)=YEAR*365.25
```

```
70 LET DAY(2)=MONTH*30.6
80 LET DAY=DAY(1)+DAY(2)
90 LET HOUR=INT(DAY*24)
100 PRINT "Congratulations!"
110 PRINT "You've been alive for over ";HOUR;" hours!"
120 END
```

In this program, we are using six variables. A variable, of course, is just a label for a value. When we tell BASIC to **LET HOUR=INT(DAY*24)**, it finds the value of **DAY**, multiplies it by 24, strips everything after the decimal point and places the result into the variable **HOUR**.

In reality, the C64 has stored the value of DAY in a **memory location**. We don't know which location, and it doesn't matter to us: all we have to do is refer to the variable and the C64 will examine the correct location to find the value. A simplified view of this process is shown later in this chapter.

In order for the C64 to 'remember' the value of a variable, it has to store both the variable name and the value somewhere in RAM. In reality, each letter of the variable, and the value itself, would occupy a separate memory location (**address**) but for now we will take a simple view and pretend that each variable name and value is stored in a single address. Later on in the book we will examine the process in more detail.

Let us take the example

```
10 LET A=5
```

BASIC would first check whether the variable **A** already exists. It does this by checking a **variable look-up table**: every time a variable is defined, it adds the variable to the table:

Variable look-up table

Variable	Stored at which address?

If it does not find the variable in the table, it assumes that it is being defined for the first time. It will then find the first free memory location in the **variable storage area** of memory and store the variable at this address:

Variable storage area

Address	Variable label	Value
49152	A	5

Now if we tell BASIC to alter the value of the variable

20 LET A=A+1

it will first check whether it exists by consulting its variable look-up table:

Variable look-up table

Variable	Stored at which address?
A	49152

Since the value of **A** is already stored at location 49152, BASIC will simply perform the calculation (5+1=6) and then store the new value at the same address as the old one:

Variable storage area

Address	Variable label	Value
49152	A	6

It is possible, of course, to tell BASIC to store a value in a particular memory location. We do this using the **'POKE address, value'** statement. The following piece of BASIC

10 POKE 49152,64

tells BASIC to store the value 64 in memory location 49152. Thus we could use memory locations instead of variables to store values. Instead of

10 LET A=5

we could say

10 POKE 49152,5

Likewise, **'PEEK(address)'** is used to look up the value stored at an address. So instead of

20 PRINT A

we would say

20 PRINT PEEK(49152)

2.2 Programming in machine code

So, what has all this to do with learning about machine code? Well, in machine code everything is done by working directly with memory locations – just like POKEing and PEEKing.

Let's suppose we wanted to print **"HELLO"** on the screen. In BASIC we would just say

```
10 PRINT "HELLO"
```

In machine code we would have to place each letter onto the screen individually. What we do, in effect, is to POKE the character code of each letter into the area of memory the C64 uses to display things on the screen. This is something you may have done in BASIC for speed – you can see an example of this technique in Listing 2.2.

There is nothing difficult about working directly with memory locations, it just takes a bit of getting used to. You don't have to have an IQ in four figures and you don't need to be a genius at programming. All you need is to be competent at BASIC programming and willing to concentrate.

Listing 2.2

```
10 REM (C) A.R.BENNETT 1985
20 :
30 REM BASIC VERSION
40 :
100 PRINT"■":TI$="000000"
110 FORI=0TO999:POKE1024+I,81:POKE55296+I,14:POKE1024+I,32:NEXT
120 PRINT"■";TI$,TI
130 END
READY.
```

2.3 Why program in machine code?

Ok, so machine code just requires concentration and practice, but why bother? What's wrong with BASIC? There are two main reasons for programming in machine code: power and speed.

Machine code is more powerful than BASIC because you have greater control over not only what the C64 does, but ***how*** it does it. This is something we will explain later, once we have got you writing your own machine code programs, but we can demonstrate the speed of machine code right now. Take a look at Listings 2.2 and 2.3. Both do the same job – they move a graphics character along each line of the screen, erasing the character currently there before moving to the next position. The difference is that Listing 2.2 is a straightforward BASIC program while Listing 2.3 sets up and

Listing 2.3

```
1000 REM MACHINE CODE VERSION
1005 :
1010 FORI=0TO70:READA:POKE49152+I,A:T=T+A:NEXT
1020 IFT<>11018THENPRINT"ERROR IN DATA STATEMENTS!!":STOP
1030 TI$="000000":SYS49152:PRINT"■";TI$;TI:END
1040 :
1050 DATA 169,0,133,247,169,4,133,248
1060 DATA 169,0,133,249,169,216,133,250
1070 DATA 162,0,160,0,169,81,145,247
1080 DATA 169,14,145,249,169,32,145,247
1090 DATA 200,192,250,208,239,24,165,247
1100 DATA 105,250,133,247,165,248,105,0
1110 DATA 133,248,24,165,249,105,250,133
1120 DATA 249,165,250,105,0,133,250,232
1130 DATA 224,4,208,206,96,255,0
READY.
```

runs a machine code program (we will explain what this means a little later; all you need to know for now is that the second program uses machine code). Type in and **RUN** each program in turn. Each program will time itself, and display the total time taken.

You will have noticed that the machine code version runs ***over 400 times faster*** than the BASIC one! A pretty good reason to learn machine code! Just think of what you can do with that kind of speed in your own programs. If you are wondering why machine code is so fast, you need to ask the question the other way around: Why is BASIC so slow?

So, let us go back to BASIC for a moment and find out what happens when we execute a piece of BASIC code. Let us suppose you type this into your C64

PRINT "Aren't example PRINT statements boring?"

The C64 may appear to display the sentence the instant you press the RETURN key, but there is actually a considerable delay – as we saw in Listing 2.2 above. The reason for this is that, as we mentioned earlier, BASIC is actually written in machine code. The C64 itself does not know how to PRINT anything: it needs a machine code program to display things on the screen, and this is in fact what happens.

When the C64 encounters a BASIC statement (like PRINT), it 'looks up' the machine code program. In this example, when it finds the PRINT program, the C64 will look at what follows the PRINT statement. In this case it finds a quotation mark, so it knows that we want to print everything up to but not including the next quote. The PRINT program then places each character, one at a time, onto the screen. After the closing quote, the PRINT program checks for variables or values to be printed before ending. The C64 then looks for the next BASIC statement.

The reason that BASIC is so slow is two-fold. First, it takes time to look up each BASIC statement. And second, because the machine code programs which make up BASIC have to perform a variety of jobs (PRINT, for example, is used to print the contents of quotes, straight values and variables), they are generally inefficient. Although the time taken to print a single item may not be noticeable, add up all the keywords in even a short program and you soon see that those short delays quickly multiply into long ones.

BASIC programming is a bit like talking to a foreigner through an interpreter. You have to speak to the interpreter in English, the interpreter then mentally translates the statement into the foreign language before finally passing on the translated message. Communication is slow and inefficient. You may have heard BASIC described as an **interpreter** or **interpreted language** for this reason.

2.4 Assemblers and assembly language

Ok, so now that we know a little about how BASIC works we can – at last – start talking about machine code! Something that often confuses people when they first come across machine code is what does machine code actually look like? You will have seen listings in books and magazines looking like Listing 2.3 above, consisting of masses of unintelligible numbers in DATA statements, described as machine code. And you have probably also seen listings like this

```
[ $C000
LDA #$08
STA $0400
LDA #$05
(etc)
```

described as machine code. So which one is really machine code? Well, if you want to be completely accurate then the answer is neither! True machine code is actually written in **binary** (base two) notation and would look something like this

```
01001100 00000100
10100000 01110001
(etc.)
```

But while the above might make perfect sense to a computer, we humans find it a little harder to cope with! For this reason, there are two alternative methods of writing machine code. The first, which we used in Listing 2.3 above, is known as a **BASIC loader**. All this does is use the decimal equivalents of binary instructions and addresses and uses BASIC to POKE them into memory. This is a useful technique to use in magazines, because

anyone can type them in without any understanding of machine code. But, while decimal numbers are an improvement on binary ones, this sort of line

```
2000 DATA 21,4,16,203,5,76
```

is not much easier to understand! So a third method of programming in machine code was developed: **assembly language** programming, often known simply as **assembler**. So what is assembly language?

Well, machine code commands take the form

⟨instruction⟩ ⟨number⟩

In binary, this might look like

```
01000010 00110101
```

And in decimal (as part of a BASIC loader)

```
DATA 66,56
```

The number can be either a value or an address in memory, while the instruction tells the C64 what to do with, or to, the value or address.

In assembly language, binary instructions are replaced by three-letter codes, known as **mnemonics**, and binary numbers are replaced by **hexadecimal** (base 16, usually known simply as **hex**) ones. We will find out about binary and hexadecimal numbering in the next chapter, but for now all we need to know is that hex numbers are indicated by placing a dollar sign ($) in front of them. So, **15** means decimal 15 while **$15** means hex 15 (decimal 21).

So, a line of assembler would look something like

```
LDA #$08
```

The assembler equivalent of the REM statement is a semicolon, so the same line might look like

```
LDA #$08        ; Load accumulator with H
```

When people say that they can program in machine code, they usually mean that they can program in assembler. Very few people program in binary these days (!), and it is assembler that you will be learning with the help of this book.

Because different computers have different types of **processors**, or **CPUs**, there are different types of assembler. The C64 has a **6510** processor and so is programmed in **6510 assembler**. Incidentally, the 6510 is almost identical to the **6502** (used in the BBC and Apple computers, among others) so you will also be able to program in 6502 assembler!

2.5 And finally . . .

The final thing to mention in this chapter is the '#' (hash sign). Normally when we use a (hex) number in assembler, the C64 will assume that we mean the memory location. So, if you typed

```
LDA $08          ; Don't worry about what this does just yet
```

into your ASSEMBLER, the C64 would use the memory location **$08**. If, however, you typed

```
LDA #$08         ; Or this, for that matter!
```

then the C64 would use the hex value **$08**. So the hash sign is just a way of telling the C64 that we mean the actual number and not the memory location.

2.6 Summary

Machine code is the direct manipulation of memory locations (just like POKEing and PEEKing). BASIC, on the other hand, normally organizes the C64's memory itself.

BASIC is slow firstly because the C64 has to look up the machine code program that actually does the work, and secondly because BASIC is designed for flexibility rather than efficiency.

True machine code programming is performed in binary (base 2) notation, but hardly anyone programs in binary these days. Assembler is a kind of half-way house between binary and BASIC and is what most people mean by machine code.

Assembly language programming uses hex (base 16) notation, and three-letter instructions known as mnemonics. Hex numbers are preceded by a dollar sign (for example, $15) to differentiate them from decimal numbers. In assembler listings, hex numbers are assumed to refer to memory locations unless they are preceded by a hash sign (for example, #$15). Numbers with a hash sign are taken to be literal values.

You should also know the meanings of the following terms: memory location (or address), variable look-up table, variable storage area,

interpreter (or interpreted language), binary, BASIC loader, ASSEMBLER (or assembly language), mnemonics, hexadecimal, 6510, 6502.

If there is anything in this summary you are not sure about, please go back and re-read the relevant parts of the chapter.

3

Hexadecimal and binary

This chapter explains the two numbering systems used in machine code: hexadecimal and binary. When you have read it, you should be able to

- Convert from hex to decimal using only a calculator
- Convert from binary to decimal using only a calculator
- Convert from decimal to hex using the program in this chapter
- Convert from decimal to binary using the program in this chapter
- Instantly recognize decimal, binary and hex numbers

The ability to count is one of civilization's most basic skills. Just think how much of our everyday lives are dependent of the concept of numbers. Without the ability to count we would have no calendars, clocks, money . . . hey, this sounds pretty good! Ah, well.

In order to count, we need some kind of numbering system. The most obvious numbering system would be to have a different symbol for each number. This would be fine for numbers up to about ten or twenty, but once you start going above this you are going to find it difficult to remember all the different symbols. For this reason, numbering systems use a small number of symbols divided into different columns. Thus in the decimal system, 1 represents one while 10 represents ten and 100 one hundred.

The decimal (**base 10**) system is the most obvious one since we each have ten digits on the end of our hands. It is not, however, the only system in use. Old money, for example, used base 12: twelve pennies made one shilling. Computers use three different numbering systems: decimal (base 10), hexadecimal (**base 16**) and binary (**base 2**). Since you are already familiar with base 10, let us use this as our first example. It might seem a bit like primary school maths (which it is), but it is important to understand the principle of decimal numbering before we look at other bases.

All numbering systems use columns to represent different values. The highest number any column can contain is one less than the base. Thus in

base 10, the highest number allowed in a single colum is nine. To write the number ten, we put a one in the tens column and a 0 in the units column:

10.

The meaning of the columns is the same in any base. The column before the point is units (ones), the column to the left is the base (in decimal, ten), the column to the left of this is the base squared (in decimal, ten times ten = one hundred) and the next the base cubed (in decimal, ten times ten times ten = one thousand) and so on

$Base^2$	Base	Units	
100s	10s	1s	Decimal value
1	0	0	One hundred
0	1	0	Ten
0	0	1	One
1	1	1	One hundred and eleven

When a number equals the base, we 'carry' into the next column. Let us take the example of a simple addition

	Base	Units	
Operation	10s	1s	Decimal value
	0	4	Four
+	0	6	Six
=	1	0	Ten

Ok, so this is all obvious stuff. So let us look at another base, say base 5. Remember that the highest number allowed in a single column is one less than the base, so base 5 will use only the digits 0, 1, 2, 3 and 4. To write decimal 5 in base 5, would carry over into the base column

$Base^2$	Base	Units	
25s	5s	1s	**Decimal** value
0	1	0	Five

Now let us do a simple addition in base 5

Operation	Base 5s	Units 1s	**Decimal** value
	0	4	Four
+	1	3	Eight
=	2	2	Twelve

To convert the answer 22, in base 5, to decimal we simply multiply the figure in the base column by the base (in this case, 2×5) and add on the units column: $(2 \times 5) + 2 = 10 + 2 = 12$. To convert a larger number, say 432 in base 5 to decimal, the same process applies

$Base^2$	Base	Units
25s	5s	1s
4	3	2

$= (4 \times 25) + (3 \times 5) + (2 \times 1)$
$= 100 + 15 + 2$
$= 117$ decimal.

3.1 Binary

You will remember that the highest figure used in a single column is one less than the base. So in binary (base 2), the only digits used are 0 and 1. These are sometimes referred to as off and on. Thus

$Base^2$	Base	Units	**Decimal** value
4	2	1	
1	0	0	Four
0	1	0	Two
0	0	1	One
1	1	1	Seven

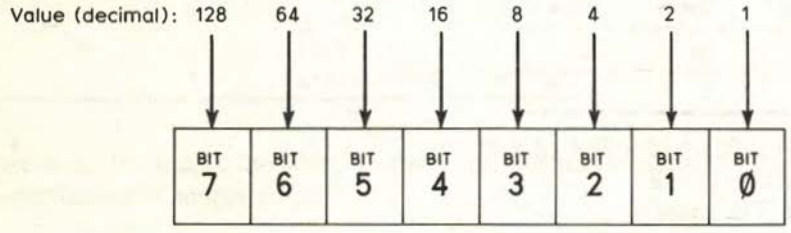
BInary digiTS are known as bits. You may have heard that the C64 is an 8-bit computer or has an 8-bit processor. This simply means that it can deal with up to eight columns of binary digits

$Base^7$	$Base^6$	$Base^5$	$Base^4$	$Base^3$	$Base^2$	Base	Units
128s	64s	32s	16s	8s	4s	2s	1s
1	1	1	1	1	1	1	1

Thus the larges number the C64 can handle at one go is binary 11111111. We can calculate this as

$(1 \times 128) + (1 \times 64) + (1 \times 32) + (1 \times 16) + (1 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1)$
$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1$
$= 255.$

Figure 3.1



You will probably already be familiar with the figure 255 as the maximum value allowed in certain functions, and you may know that 255 is the maximum value that can be sored in a single memory location (**byte**). To store a number larger than 255, the C64 uses two bytes. Thus the highest number the C64 can handle is binary 1111111111111111 which translates to decimal 65535, or 64K (1K = 1024 characters). Thus 65535 is the C64's highest memory location.

We have already seen how we convert from binary to decimal, but converting the other way around is more difficult. For this reason we have provided you with a simple program (Listing 3.1) to do the job for you. Simply type in any decimal number up to 255 and the program will convert it to binary. To avoid confusing decimal and binary numbers (does 101 mean one

Listing 3.1

```
10 REM DECIMAL TO BINARY
20 :
30 B$="01"
40 INPUT"INPUT DECIMAL NUMBER (0-65535):";D$:D=VAL(D$)
50 IFD<0ORD>65535THEN40
60 PRINT"BINARY NUMBER = %";
70 FORI=15TO0STEP-1:G=INT(D/2↑I):D=D-G*2↑I:PRINTMID$(B$,G+1,1);:NEXT
READY.
```

hundred and one, or five?), we will follow the standard computing convention of placing a percent sign (%) in front of all binary numbers used from now on. Thus **%101** is binary (decimal 5) while **101** is decimal.

3.2 Hexadecimal

Hex (base 16) is the numbering system you will use most when you write your own machine code programs. Since hex is base 16, the highest figure we can have in a single column is the equivalent of 15. There is, however, a problem here. Each digit must fit into a single column. Since there are no symbols for digits greater than nine, we will have to invent some. Rather than use unfamiliar symbols, hex uses the letters A to F to represent the numbers 10 to 15. Thus

Decimal:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Thus decimal 16 would be hex 10. As with binary, we have the problem of distinguishing hex from decimal. Is 11 decimal or hex (decimal 17)? Again, we use a simple convention to get round the problem. Hex numbers are preceded by a dollar sign (**$**). Thus **$10** is hex (decimal 16) while **10** is decimal.

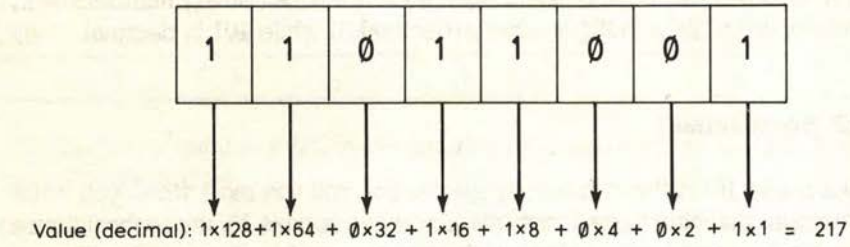
So let us look at how hex works

Base[2]	Base	Units	
256s	16s	1s	**Decimal** value
1	0	0	256
0	1	0	16
0	0	1	1
1	1	1	273

Since the highest number the C64 can handle is 65535, we are only going to deal with four digit hex numbers. To see the reason for this, let us calculate the highest four-digit hex number **($FFFF)**

= (4096 × F) + (256 × F) + (16 × F) + (1 × F)
= (4096 × 15) + (256 × 15) + (16 × 15) + (1 × 15)
= 61440 + 3840 + 240 + 15
= 65535

Figure 3.2



As with binary, converting ***into*** decimal is straightforward (although a calculator comes in handy if you are not too hot at mental arithmetic!). Converting ***from*** decimal is more difficult, so again we have written a program (Listing 3.2) to do the job for you. You will also find a useful chart in Appendix 1.

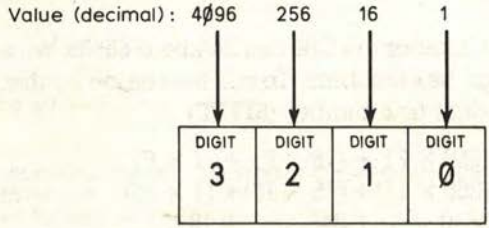
Listing 3.2

```
10 REM DECIMAL TO HEX
20 :
30 H$="0123456789ABCDEF"
40 INPUT"□INPUT DECIMAL NUMBER (0-65535):";D$:D=VAL(D$)
50 IFD<0ORD>65535THEN40
60 PRINT"■■HEX NUMBER = $";
70 FORI=3T00STEP-1:G=INT(D/16↑I):D=D-G*16↑I:PRINTMID$(H$,G+1,1);:NEXT
READY.
```

3.3 Why do we use hex?

A question which those new to machine code often ask is "Why does the C64 use hex?". The answer is simply that the C64, like any other computer, actually works in binary. Hex is used as an alternative because a number like **$0100** is easier to handle than **%100000000**! Decimal cannot be used since ten is not a power of two whereas sixteen is.

Figure 3.3



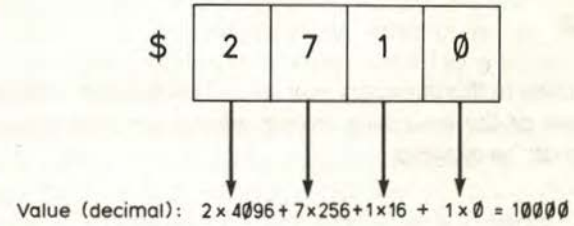
3.4 Memory addressing

We have already mentioned that the C64 has 65536 bytes (64K) memory available. But when you switch on, the C64 tells you that it has only 38911 bytes available to BASIC. What happened to the other 26625 bytes? We will discuss memory in more detail later, but let us take a quick look now at the way the C64's memory is arranged. The C64 has 64K of RAM. Some of this RAM, however, is needed by the C64 to take care of its own workings. The first 2047 bytes, for example, are used by the **Kernal** to store information. The Kernal is the name given to the C64's operating system: the set of machine code routines which take care of the workings of the C64. (The most useful Kernal routines are detailed in Appendix 7).

The Kernal uses this storage area to store such information as the current cursor position, the colour of the screen, which line of a BASIC program is being executed and so on. Also included in this area is the screen memory, used to store the contents of the screen (you may have POKEd to this in BASIC) and the sprite map pointers, used to keep track of sprite shape definitions. From 2048 to 40959 is the memory area available for your BASIC programs. The 8K following 40960 contains the BASIC programming language. BASIC is in fact made up of a number of machine code programs. These programs are stored here and automatically run whenever you run a BASIC program. The 4K beginning at 49152 is the area normally used for our machine code programs, and the rest of RAM is used for sound and graphics, input/output Kernal.

The last thing we need to explain in this chapter is two pieces of machine code jargon: paging and absolute addresses. For convenience, the C64's memory is divided into areas ('**pages**') of 256 bytes. The first 256 bytes are called zero-page, the next 256 page one, the next page two and so on. An absolute address is simply any address higher than **$FF**. A detailed, fully annotated C64 memory map is given in Appendix 2.

Figure 3.4



3.5 Summary

In order to be able to manipulate numbers, we need some kind of numbering system. The decimal system (base 10) is convenient for humans since we have ten fingers and thumbs, but computers work in binary (base 2). Binary digits are known as bits, and eight bits make one byte.

Because binary is inconvenient (the top of the C64's memory would be **%1111111111111111** in binary!), a compromise system is used. This system is known as hexadecimal, or hex (base 16). Any number between zero and the top of memory can be stored in hex using just two bytes.

In order to distinguish decimal, binary and hexadecimal numbers, a simple convention is used. A percent sign precedes binary numbers (for example, **%01010000**) and a dollar sign precedes hex ones (**$50**).

3.6 Exercises

In order to ensure that you understand how to convert from binary and hex into decimal, turn the following into decimal. The answers can be found in Appendix 6.

1. **$A0**
2. **%0110**
3. **$FC**
4. **%1100**
5. **$9009**
6. **%11000000**
7. **$C040**
8. **%01100011**
9. **$AD40**
10. **%11111000**

If there is anything in this summary you are not sure about, or if you got more than one or two of the exercises wrong, please go back and re-read the relevant parts of the chapter.

4

Machine code commands

Having covered the necessary background information, we are now ready to begin learning our first machine code commands. In this chapter, you will learn about the following commands:

- LDA
- STA
- LDX
- STX
- LDY
- STY
- RTS

You will also learn how to use the following BASIC statement:

- SYS

By the end of this chapter you will be able to write your first machine code program!

When we program in BASIC, most of the work is done using variables. BASIC programming could therefore be defined as the manipulation of data using variables (on the C64, the variables available are **A–Z, AA–ZZ, A$–Z$** and **AA$–ZZ$**). When we program in machine code, most of the work is done using memory locations directly rather than variables. Machine code programming could therefore be defined as the direct manipulation of memory.

In order to manipulate memory locations, we need some way of doing the manipulating. C64 machine code programming offers us three different manipulators which will fast become old friends: the accumulator, the **X** index and the **Y** index. Manipulators work in a broadly similar way to variables. They are used to select a value, perform an operation on that value, and place the result into the same or a different location. Each

manipulator acts on a single memory location and can therefore handle values up to 255 (**$FF, %11111111**).

The most important manipulator is the accumulator. Many of the mathematical commands used in machine code are only available through the accumulator. The **X** and **Y** indices are primarily used within loops. They are known as indices since they act as the 'index' (counter) of loops. An example of an index in BASIC would be the counter in a FOR . . . NEXT loop

```
FOR A=0 TO 999: POKE 1024+A, 32: NEXT A
```

When we assign a value to a variable in BASIC, we can do so in one of two different ways

```
A=50
B=A
```

In the first form a variable is assigned a constant, while in the second a variable is set to the value of another variable. As we mentioned in Chapter 2, in machine code we distinguish between constants and memory locations by preceding constants with a hash sign (#). Thus **#$FF** refers to the value 255 while **$FF** refers to memory location 255.

4.1 The LDA, LDX, LDY, STA, STX, and STY commands

In BASIC, to assign a value to a variable we use the LET statement

```
LET A=100
```

(although most BASICs, the C64 included, allow you to omit the LET). In machine code, the LDA (LoaD Accumulator) command is used to assign a value to the accumulator. We assign the value 32 (**$20**) to the accumulator like so

```
LDA #$20
```

Note that the hash sign (#) indicates that we mean the value **$20** and not the memory location.

```
LDA $20
```

would load the accumulator with the value stored in memory location **$20**. The difference can be likened to the difference between

```
LET A=32
```

and

```
LET A=PEEK(32)
```

The LDX and LDY commands work in an identical fashion. So

LDX #$20

loads the **X** index with the value 32 and

LDY #$20

does the same for the **Y** index.

So far we know how to store the contents of a memory location in the accumulator, but not how to do anything with the value once we have got it there. The most obvious things we will want to do are

(a) perform a calculation with it and place it back into the same memory location, and
(b) place it into a different memory location.

We will deal with calculations in later chapters, but first let us find out how to store the value of the accumulator in a memory location. This is performed using the STA (STore contents of Accumulator) command. To place the value of the accumulator into memory location 32, for example, we would use

STA $20

The accumulator retains its value until a new LDA command is issued, so

STA $21

would place the same value into location 33. The **X** and **Y** indices equivalents are STX and STY. The command format is identical to STA.

Ok, let us do something moderately useful. The border colour of the C64 is stored in memory location 53280. So to make it yellow, in BASIC we would POKE 53280 with the colour code for yellow (7)

POKE 53280,7

To achieve the same effect in machine code we would write

LDA #$07

to load the accumulator with the value 7, and then

STA $D020

to place this value into memory location 53280.

4.2 Our first machine code program

We know enough to write our first machine code program! This program will place a white letter A in the top left-hand corner of the screen. This kind of masterpiece is not likely to worry Jeff Minter over-much, admittedly, but we all have to start somewhere: even Minter probably started out by writing a similar program!

Before we begin writing our program, we have to tell the assembler whereabouts in memory to store it. We do this by a [(opening square bracket) followed by the start address in hex. The start address is the memory location used to store the first byte of the program

```
[ $C000
```

The address **$C000** is the start of the 4K area reserved for machine code programs. You can store them elsewhere (you have to if your programs require more than 4K RAM), but you can easily move them if required just by changing the above line.

The next line of our program assigns the screen code for the letter A (1) in the accumulator

```
LDA #$01
```

(A complete list of character codes can be found in Appendix 3: while colour codes can be found in Appendix 4.) To place a character onto the screen, we need to place the screen code of the character into the appropriate section of screen RAM. Screen RAM begins at the top left-hand corner of the screen with 1024. So we need to store the content of the accumulator at location 1024 (**$0400**)

```
STA $0400
```

On a relatively new C64, you may be able to see this character as soon as your program reaches this point. On older machines, however, you will not see anything because the character will be displayed in the background colour of the screen! We need to place a colour code into the top left-hand corner of colour memory, which begins at 55296 (**$D800**)

```
STA $D800
```

Since we have not changed the value of the accumulator, it will still contain its original value of 1. Since 1 is the colour code for white, the character in the top left-hand corner of screen memory (our letter A) will appear in white. We need one final line to complete our program, and that is RTS. RTS (ReTurn from Subroutine) is the equivalent of the BASIC RETURN statement. It is normally used to return from within a subroutine. If, however, it is used when you are not in a subroutine it instructs the C64 to return to BASIC.

If you have not already done so, type the above program into the assembler supplied with this book. It should look like this.

```
5000 [ $C000
5010 LDA #$01
5020 STA $0400
5030 STA $D800
5040 RTS
```

To assemble your program, simply enter RUN. The assembler will then work through your program, assembling a line at a time, until the finished program is ready to run at address **$C000**.

4.3 Running a machine code program

If you have followed the above steps, typing in the program and then entering RUN, the program is now ready to be run. The more impatient of you will already have discovered that the BASIC command RUN does not work with machine code programs! Instead, we use the command **SYS ⟨address⟩**. SYS tells the C64 to execute the machine code program which is at the specified address. Since our program is at address **$C000** (49152), we simply enter

```
SYS 49152
```

All being well, a white A will have appeared at the top left-hand corner of the screen and the READY prompt will have returned a couple of lines below the SYS command. Congratulations: you have just written your first machine code program!

There is another way of running a machine code program, and that is to use the USR command rather than SYS. USR is a little more complicated, but can be extremely useful. It is explained in Chapter 10.

4.4 What if my machine code program crashes?

Something which many people worry about is: What happens when you crash your C64 with a POKE or machine code program? If you are one of these people, we have both an assurance and a warning for you. The assurance is that ***you absolutely cannot damage your machine in any way by incorrect POKEing or crashing a machine code program.*** The warning is that you ***can*** cause the machine to 'lock up'. If this happens, you may have to switch it off and on again to cure the fault. While this will not do any damage, you will lose whatever you had in memory. For this reason, we offer you one golden rule when writing any machine code program: **Always save your program to tape or disk before executing it.** If you don't follow this advice, don't blame us if you spend three hours typing in a masterpiece and then find that the machine locks up when you try to run it because you made a simple mistake in the second line!

4.5 Remarks

Most books on BASIC programming advise you to include REM statements in your programs, remarks that remind you what the various bits of code do. This advice applies a hundred times more to machine code programs. Without remarks, you will find that you will have forgotten the workings of your programs within hours, let alone weeks or months. This is especially true when you are just beginning to learn.

In BASIC, we use the REM statement to indicate remarks. In machine code we use the semicolon (';'). This can be used, just like a REM, either on the end of a line or on a line of its own

```
LDA #$01 ;Load accumulator with 1
```

Semicolons can also be used on their own to separate sections of your programs. We will also be using remarks and separators extensively in the more complicated programs in later chapters.

4.6 Summary

Machine code programming can be defined as the direct manipulation of memory locations. Machine code uses three 'manipulators' (used like variables): the accumulator, the X index and the Y index. LDA is used to LoaD the Accumulator with a value, and STA to STore the contents of the Accumulator at a specified address. The equivalent commands for the X and Y indices are LDX, LDY, STX and STY. The accumulator and indices retain their values after an STA, STX or STY.

Before you can write a machine code program, you need to specify the start address. This is done as follows: [⟨start address⟩. To end a program, you use RTS. When you have written a program, RUN will assemble it and return you to BASIC. You then use SYS ⟨start address⟩ (remember that the start address must be in decimal) to execute it.

Finally, semicolon is the equivalent of the BASIC REM statement. We strongly suggest using it extensively, especially while you are fairly new to machine code.

4.7 Exercise

Write, assemble and execute a machine code program to place your first name at the top left-hand corner of the screen, with each letter in a different colour. You will need to refer to Appendix 3, Appendix 4 and Appendix 5. An example answer is shown in Appendix 6.

5

Labels, flags and branching

This chapter introduces labels, flags and branching. When you have read it, you should be able to answer the following questions:

- What is a label?
- How are labels assigned?
- What are the two uses for labels?
- What is a flag?
- How do we branch in machine code?

You will also be able to use the following machine code commands:

- INX
- INY
- INC
- DEX
- DEY
- DEC
- TAX
- TAY
- TXA
- TYA
- BEQ
- BNE
- BCC
- BCS
- CMP
- CPX
- CPY

5.1 Labels

A label is simply a name given to a line number or constant value in a machine code program. You can name (label) a line number or value at the beginning of a program, and then refer to the label. Labels are ***not*** variables for the simple reason that the value of a label cannot be altered once it has been set. Thus the following label definition is legal

```
6000 :VALUE=$01
```

(the colon is equivalent to the BASIC LET statement), but this

```
6010 :VALUE=VALUE+1
```

is not. Before we look at how labels are used in machine code, let us see how we might use them in BASIC.

5.2 Using labels in BASIC

The first thing to point out is that standard C64 BASIC Version 2.0 does not support labels, but some other BASICs do. Here we will explain how labels could be used in a BASIC which supports them.

There are two ways of using labels. The first is to label program lines. When you are writing a program, you will often want to GOTO or GOSUB to a routine which you have not yet written. A typical example might be something like

```
640 REM IF SCORE BEATS HIGH-SCORE GOTO WIN ELSE GOTO LOSE
650 IF SC)HS THEN GOTO 5000 ELSE GOTO 6000
```

You then have to remember to write the appropriate routines at these line numbers or, if you have to write them somewhere else, alter the line to suit. A much simpler and neater way would be to write the line as

```
650 IF SC)HS THEN GOTO WIN ELSE GOTO LOSE
```

The first line of each routine would then look like

```
5000 :WIN:
...
6000 :LOSE:
```

Not only can you then not worry about where the routines finally end up, but you also no longer need the REM statement in line 640 as the line becomes self-explanatory. All we have done is to label two lines and then branch to them by referring to these labels.

The second use of labels is to label constants. For example, if you intended to use the constant 3.14159265 (pi) more than once in a BASIC program, you might assign a variable **PI** somewhere near the beginning

```
120 PI=3.14159265
```

Next time you wanted to use pi, you could simply use **PI**

```
300 D=2*PI*R
```

You can do a similar thing in machine code

```
6000 :PI=3.14159265
7000 LDA #PI
```

5.3 Using labels for branching

To use a label for branching, you must first label the line you want to jump to. This is done by placing the label after a colon

```
6050 LDA #$04 :DISPLAYSCORE
```

To jump to this line, you would use one of the branching instructions we will learn about later in this chapter

```
7010 BEQ DISPLAYSCORE
```

We have already stressed the importance of using remarks in machine code programs; similarly, we strongly recommend using meaningful labels.

5.4 Labelling constants

Constants are labelled in a very similar way to BASIC; instead of LET, you use a colon

```
6010 :BORDERCOLOUR=$D020
```

or

```
6020 :RED=$02
```

Not only do labels make life easier while you are writing the program (meaningful words are a lot easier to remember than numbers), but they also make your programs much more readable. Compare

```
7000 LDA #$02
7010 STA $D020
```

with

```
7000 LDA #RED
7010 STA BORDERCOLOUR
```

Labels are also convenient if you want to change something throughout your program. If, for example, you were using the colours green and blue for your screen displays and wanted to change them to yellow and red, you need only change your colour labels. So, for example, if your original labels looked like this

```
6010:FOREGROUND=$05
6020:BACKGROUND=$06
```

you can change them to read

```
6010:FOREGROUND=$07
6020:BACKGROUND=$02
```

and re-assemble the program.

Right at the beginning of this chapter we said that labels are ***not*** the same as variables because their values cannot change within a program. The reason for this is that the assembler replaces labels with their values throughout a program during assembly. For example, let us take the following short program

```
6000 [ $C000
6010 ;This does not do anything useful so do not bother assembling it
6020 :SCREENPOSITION=$0400
6030 :COLOURPOSITION=$D800
6040 :COLOUR=$03
6050 :CHARACTER=$5E
6060;
6070 LDA #COLOUR
6080 STA COLOURPOSITION
6090 LDA #CHARACTER
7000 STA SCREENPOSITION
```

This would be assembled as if it were written

```
6070 LDA #$03
6080 STA $D800
6090 LDA #$5E
7000 STA $0400
```

Labels cannot change their value within a program because they are used while the program is being **assembled** and not while it is running. Thus

```
7000 STA CHARACTER+1
```

is legal (the assembler calculates the value of **CHARACTER+1** and uses this during assembly), while

```
7000:CHARACTER=CHARACTER+1
```

is not because **CHARACTER** has already been defined as a label and cannot

now be redefined. If you wanted to achieve the same effect, you would have to use another label like so

```
7000:CHARACTER2=CHARACTER+1
```

5.5 The increase commands: INX, INY and INC

The fact that we cannot alter the value of a label within a program presents us with a problem when we want to create a loop. In BASIC, we could create a loop to display the numbers one to ten like so

```
100 FOR A=1 TO 10
110 PRINT A
120 NEXT A
```

If we did not have the FOR . . . NEXT loop to help us, we would write it like this

```
100 A=A+1
110 PRINT A
120 IF A<10 THEN GOTO 100
```

But this method involves changing the value of the variable **A**. Since we can not do that with machine code labels, how would we write a loop in machine code? Well, 6502 assembler supplies us with three instructions for this purpose: INX, INY and INC. INX tells the assembler to INcrease the X index by one. In other words, it is equivalent to the BASIC statement **X=X+1**. INY is, of course, the Y index equivalent, adding one to the Y index. We will discuss INC in a moment.

All three instructions feature roll-over. To illustrate the point, enter and assemble the following program:

```
6000 [ $C000        ;Assemble at decimal 49152
6010 LDX #$FF       ;Load the X index with decimal 255
6020 INX            ;Increase the X index by one
6030 STX 828        ;Store the X index value in location decimal 828
6040 RTS            ;Return to BASIC
```

If you now run the program (SYS 49152) and PRINT PEEK(828) you will see that the value returned is zero.

The INC instruction works in exactly the same way as INX and INY, but instead of operating on one of the indices, it acts on a specified memory location. We will see some more complicated uses of INC when we deal with addressing modes in Chapter 6, but its simplest forms, and the only two we need worry about now, are

```
INC $XX
```

and

INC $XXXX

Both instruct the assembler to increase the value of the specified address by one. Thus

```
6010 LDA #$05     ;Load accumulator with five
6020 STA $FF      ;Store five in address $FF
6030 INC $FF      ;Increase value in address $FF by one
```

would leave address **$FF** with a value of **$06** stored in it (the original value of **$05** plus the INC increase of **$01**).

5.6 The decrease commands: DEX, DEY and DEC

In most loops, we want to increase the value of the loop counter (known as a 'forward' loop). There are, however, occasions when we would want to decrease it (a 'backward' loop). For example

```
100 FOR C=10 TO 1 STEP-1
110 PRINT C:FOR B=0 TO 250:NEXT B:REM Delay loop
120 NEXT C:PRINT "Take-off!"
```

or, without using a FOR . . . NEXT loop

```
100 C=10
110 C=C-1:PRINT C:B=0
120 B=B+1:IF B<250 THEN GOTO 120
130 IF C>1 THEN GOTO 110
140 PRINT "Take-off!"
```

In machine code, the method is exactly the same for backward loops as for forward loops, the only difference being that we use the decrease instructions instead of the increase ones. DEX, DEY and DEC are, as you would expect, the backward loop equivalents of INX, INY and INC. Thus DEX decreases the value of the X index by one, DEY decreases the value of the Y index by one and DEC decreases the value of the specified address by one.

The decrease instructions all feature roll-under, so that

```
6010 LDX #$00
6020 DEX
```

would leave the X index with a value of **$FF** (**$00** minus one). Again, you can demonstrate this by entering and assembling the following short program:

```
6000 [ $C000      ;Assemble at decimal 49152
6010 LDA #$00     ;Load accumulator with zero
6020 STA 828      ;STA accumulator value at address decimal 828
```

```
6030 DEC 828      ;Decrease value of address decimal 828 by one
6040 RTS          ;Return to BASIC
```

If you now SYS 49152 to run the program and PRINT PEEK(828) you will see a value of decimal 255.

5.7 The transfer commands: TAX, TAY, TXA AND TYA

So far we have seen how to increase and decrease the values of the X index, the Y index and any chosen address. We have not seen how to do the same with the value of the accumulator. There is not any direct way of adding to or subtracting from the value of the accumulator, but there is a simple way around the problem using the transfer commands. The transfer commands are used to transfer the value of the accumulator to either of the indices and vice-versa. They have a number of uses, but one of the main ones is to add to or subtract from the value of the accumulator. The instructions are

```
TAX          ;Transfer Accumulator value to the X index
TAY          ;Transfer Accumulator value to the Y index
TXA          ;Transfer X index value to the Accumulator
TYA          ;Transfer Y index value to the Accumulator
```

In all cases, the value you are transferring stays the same: a copy of the value is transferred. For example

```
6010 LDA #$07     ;Load accumulator with 7
6020 TAX          ;Transfer accumulator value (7) to X index
```

would leave both the accumulator and the X index with a value of 7. To increase the value of the accumulator by one, we would

```
6010 TAX          ;Transfer accumulator value to X index
6020 INX          ;Increase X index by one
6030 TXA          ;Transfer X index value back to accumulator
```

Similarly, to decrease the value of the accumulator you would replace line 6020 with

```
6020 DEX          ;Decrease X index by one
```

We could, of course, have used the Y index instead of the X one. In practice, you will probably already be using one of the indices and so would use the other one to increase or decrease the value of the accumulator.

5.8 The conditional branching commands: BEQ and BNE

A construct common to all computer languages is the conditional branch. This is just a fancy way of saying a branch (GOTO is a BASIC example of a branch instruction) which only happens if one or more conditions are met. The obvious example in BASIC is the IF . . . THEN statement

```
100 IF A=B THEN GOTO 500
110 IF C<>D THEN GOSUB 1500
```

Before we look at the first of the machine code examples, BEQ and BNE, let us take a slightly closer look at the IF . . . THEN construct.

The IF . . . THEN statement is in two parts. First, there is the comparison (following the IF statement) and second, the branch or action (following the THEN statement). What BASIC does is to set a flag dependent on the result of the comparison. Let us look at our first example, setting the values of A and B before we do so

```
90 A=10: B=10
100 IF A=B THEN GOTO 500
```

BASIC would first evaluate the comparison

```
A=10, B=10 therefore A=B
```

It then sets a flag telling itself to execute the branch or action following the next THEN statement

```
Set THEN flag to TRUE
```

Next it reads the code following the THEN statement and checks the status of the THEN flag

```
THEN flag is true, so execute code: GOTO 500
```

If A=5 and B=10, the same process would occur except that the THEN flag would be set to FALSE and the code following the THEN would be ignored. We can see from this that although the two keywords are used together, IF and THEN are actually separate operations. We need to understand this because machine code does the equivalent of an IF . . . THEN statement in two completely separate steps. Let us look first at the equivalent of IF.

The machine code equivalent of the IF statement is the comparison statement. This takes three forms, of course, for the accumulator, X index and Y index

```
CMP     ;CoMPare specified value with the accumulator
CPX     ;ComPare specified value with the X index
CPY     ;ComPare specified value with the Y index
```

The two uses of the CMP instruction are

```
CMP #$01        ;Compare accumulator with the value one
```

and

```
CMP $01         ;Compare accumulator with the value stored in address one
```

If the comparison is true, a COMPARISON EQUAL flag is set; otherwise one of the COMPARISON NOT EQUAL flags are set.

So far, we have made a comparison but we have not acted on it: all we have done is to set a flag. This is where the machine code equivalent of the THEN statement comes in. This is the branch instruction set

```
BEQ             ;Branch to the specified label if the comparison flag is EQual
                 (usually stated as 'Branch if EQual')
BNE             ;Branch to the specified label if the comparison is Not Equal
                 (usually stated as 'Branch if Not Equal')
```

So, for example, a machine code search routine might contain:

```
6010 BEQ FOUND  ;Record found, branch to FOUND line
6020 BNE SEARCH ;Not this one, branch back to SEARCH line
```

Ok, let us summarize all this, CMP is the basic comparison instruction. The format is **'CMP #$⟨value⟩'** or **'CMP $⟨address⟩'**. CMP instructs the assembler to compare the value, or the value stored in the specified address, with the value of the accumulator. If the two values are equal, an equal flag is set. If the two are different, a not-equal flag is set. CPX and CPY work in exactly the same way except that the comparison is made with the X and Y index respectively and not with the accumulator.

Once the equal or not-equal flag has been set, we can act on the status of the flag by the branch instructions: BEQ and BNE. The formats of the instructions are **'BEQ ⟨label⟩'** and **'BNE ⟨label⟩'** respectively. BEQ will branch to the line containing the specified label if the equal flag is set, while BNE will branch to the label if the not-equal flag is set.

Enter and assemble the following demonstration program:

```
6000 [ $C000
6010 :SCREEN=$0400          ;Top-left of screen
6020 :COLSCREEN=$D800       ;Top-left of colour screen
6030 LDA #$01               ;Character code of 'A'
6040 LDX 828                ;Load X index with decimal 828
6050 CPX #$00               ;Compare X index value with zero
6060 BEQ XISZERO            ;Branch to line with XISZERO label
6070 LDA #$02               ;Character code of 'B'
6080 STA SCREEN :XISZERO    ;Top-left of screen
6090 LDA #$01               ;Colour code of white
6100 STA COLSCREEN          ;Top-left of colour screen
6110 RTS                    ;Return to BASIC
```

Once you have assembled the program, enter

```
POKE 828,00:REM Place zero in location 828
SYS 49152
```

A white 'A' will have appeared at the top left-hand corner of the screen. This is because the comparison in line 6050 set the equal flag to true, and thus the program branched from line 6060 to 6080 and so line 6070 was never executed.

Now enter the following, remaining in BASIC

```
POKE 828,200:REM Place 200 in location 828
SYS 49152
```

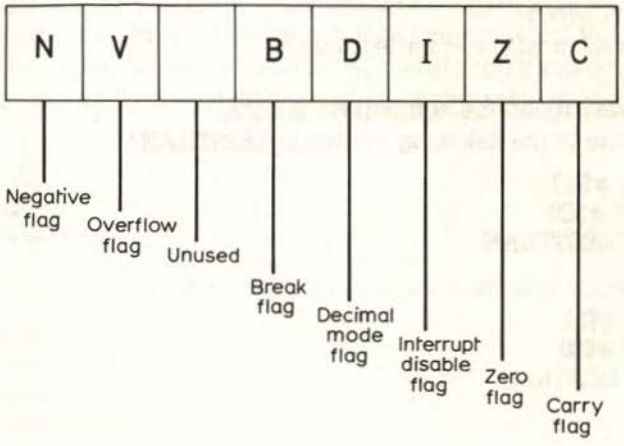
This time a white 'B' will have appeared. This is because the comparison in line 6050 failed (that is, the equal flag was not set), the branch in 6060 is therefore not carried out and so line 6070 is executed, loading 'B' into screen memory. The BASIC equivalent of this would be

```
100 A=1: IF X <> 0 THEN A=2
```

5.9 Flags

We had a very brief look at flags in Chapter 1, now it is time to see how they work. You will remember that a flag is simply a bit which can be set to either 1 (often known as true) or 0 (known as false). The C64 has a number of built-in 'status' flags held in a memory location known as the **Status Register** or **Process Register**. This looks something like that shown in Fig. 5.1. We have already dealt with one of these flags, albeit without knowing it, and that is the **Zero flag**. This is the flag used by our friends CMP, CPX and CPY. If the result

Figure 5.1



of a comparison is true (equal), then the zero flag is set to 1. If the comparison is false (not equal), the zero flag is set to 0.

To understand why it is called the zero flag, we have to know how the assembler carries out a comparison. In fact, it does it in the simplest way possible: subtract one from the other. So

```
6010 LDA #$01
6020 CMP #$01
```

is like saying

```
A=1
ZEROFLAG=A-1
```

Since #$01−#$01 equals zero (meaning that they are equal), the zero flag is set to 1 (zero=true). If the two were not equal, the subtraction would result in a value greater than zero and so the zero flag would be set to 0 (zero=false).

5.10 Greater-than and less-than comparisons

So far we have learnt how to use the comparison instructions together with BEQ and BNE to compare two values for equal or not-equal. At other times, though, we would want to know more than this: if the two are not equal, we will often want to know which one is greater. This is where the other two branch instructions come in

```
BCC          ;Branch if Carry Clear (tests for less-than)
BCS          ;Branch if Carry Set (tests for equal-to or greater-than)
```

These are used in exactly the same way as BEQ and BNE as we shall see in a moment (machine code is actually very easy once you get started – everything works in the same way!). There are, however, two things which often confuse beginners until they are familiar with BCC and BCS

1. Which is which?
   Does BCC or BCS test for less than?
   (It's BCC).
2. Which way round the comparison is done.
   Which one of the following branch to **LESSTHAN**?

   (a)
   ```
   LDA #$20
   CMP #$20
   BCC LESSTHAN
   ```

or

   (b)
   ```
   LDA #$21
   CMP #$20
   BCC LESSTHAN
   ```

(It's (a)).

Fortunately, you will have no such problems if you take two or three minutes now to memorize the following simple reminders:

(a) To remember which of BCC and BCS tests for less-than, remember that 'C' is lower than 'S' in the alphabet. Put another way, the ASCII value of C is less than the ASCII value of S.
(b) To remember which way round comparisons are done, just think of how you would do it in BASIC

```
100 IF A<6 THEN...
```

The value given after the less-than sign (in this case, 6) is compared with the value before ("is A less than 6?"). Thus, "Is the accumulator (or X or Y index) less than the value after the comparison instruction?"

Remember also that BCS tests for greater-than ***or equal***. If you want to test only for greater-than, you would have to do a BEQ ***before*** doing a BCS.

```
6010 CMP #$05
6020 BEQ EQUAL
6030 BCS GREATERTHAN
6040 BCC LESSTHAN
```

In BASIC, this would look like

```
100 IF A=5 THEN GOTO 1000:REM EQUAL ROUTINE
110 IF A>=5 THEN GOTO 2000:REM GREATER-THAN ROUTINE
120 IF A<5 THEN GOTO 3000:REM LESS-THAN ROUTINE
```

5.11 Out-of-range errors

There is one slight complication to branching: you will sometimes get an 'out of range' error during assembly. To explain why, we have to understand how the assembler stores branching instructions in memory. But before we do this, let us get our priorities right and tell you what to do if you get the error! The simple rule is: "If the branch is out of range, reverse the test." For example if

```
6010 CMP #$20
6020 BEQ EQUAL
6030 LDX #$30
```

resulted in an 'out-of-range' error, rewrite it the other way round

```
6010 CMP #$20
6020 BNE NOTEQUAL
6030 JMP EQUAL
6040 LDX #$30:NOTEQUAL
```

The JMP (JuMP) instruction simply tells the assembler to jump to the label following it (in this case, NOTEQUAL). In other words, it is just like a GOTO statement in BASIC except that we use a label instead of a line number. Ok, that's the solution, but why do we have the problem in the first place? Well, branch instructions are stored in two bytes. The first is the instruction itself (for example, BEQ), and the second is the number of bytes the C64 should jump.

The problem is that you can branch both forwards and backwards. Since the C64 has only one byte to store both the distance to be jumped and the direction, how does it do it? Well, it uses the seventh bit to store the direction: a zero means jump forward, and a one means backward. So, for example

```
%00000011
```

means jump three bytes forward, while

```
%10000011
```

means jump three bytes backward. The problem, of course, is that the maximum distance the C64 can store this way, leaving the seventh bit free for the direction, is **%1111111**, or decimal 127. Anything greater than this causes an 'out-of-range' error.

5.12 Exercise

Write a program to check the contents of 828. If it contains 2 then turn the border RED otherwise turn background colour equal to contents of 828.

6

Addressing modes

This chapter explains addressing modes. It covers

- Immediate
- Zero-page
- Absolute
- Relative
- Implied
- Absolute,X
- Absolute,Y
- Zero-Page,X
- Lo–Hi storage (used by the indirect addressing modes)
- Indirect,Y
- Indirect,X
- Indirect

Every machine code instruction will either branch or jump to a different section of the program, store a value somewhere or retrieve a value from somewhere. These operations break down into what are known as addressing modes. (A full list of machine code instructions, showing which one belongs in which addressing mode, is given in Appendix 8.)

In Chapter 2, you may remember that we mentioned the difference between absolute and zero-page numbers. We said that an absolute number has four hex digits and is between **$0100** (decimal 256) and **$FFFF** (decimal 65535) inclusive. A zero-page number, in contrast, has only two hex digits and is between **$00** and **$FF** (decimal 255) inclusive. Absolute and zero-page are two different addressing modes. Thus

```
LDA $22
```

is a zero-page instruction (that is, it is executed in zero-page addressing mode), while

```
LDA $0400
```

is an absolute instruction (that is, it is executed in absolute addressing mode).

Addressing modes are a very important part of machine code programming. In this chapter, we will examine each addressing mode in turn. Some of them have rather esoteric-sounding names! Do not worry too much about the names – you will learn them quickly enough as they become familiar to you – but do make sure that you understand the function of each.

This chapter presents quite a lot of information. None of it is particularly complex, but it is important to have a thorough understanding of it, so take it slowly and carefully. Let us start by looking at the modes we have already used.

6.1 Immediate

Immediate mode is used to load values into manipulators and memory locations. Examples of immediate mode instructions are

```
LDA #$50
STA $D020
```

and, since our assembler allows us to use decimal values as well as hex ones

```
LDA #10
```

One of the main uses of immediate mode is to set up and use labels

```
:BLUE=$06
:SCREENCOL=$D020
...
LDA BLUE
STA SCREENCOL
```

6.2 Zero-page

Zero-page occupies the first 256 bytes of memory (**$00** to **$FF** inclusive). Most of zero-page is required by the C64's BASIC and operating system, but some locations are available for use in your machine code programs. The memory map in Appendix 2 shows which ones. Because so few locations in zero-page memory are available to you, you would not normally store straightforward values in them. You would usually use them for some of the other addressing modes examined in this chapter.

The reason that zero-page memory is largely used by the C64 itself is that the closer to the start of memory a routine or value is, the less time it takes to locate it and thus the faster the C64 will operate. You may have noticed a similar effect in your BASIC programs, where the early sections of your

program run slightly faster than an identical routine later in the program. For this reason, it is a good idea to put the most commonly-called subroutines at the beginning, rather than end, of your programs.

6.3 Absolute

Examples of absolute mode commands are

```
LDA $C000
STA 49152
```

Absolute addressing is the one normally used in machine code programming. It allows you to address all of the C64's memory. Machine code programs and data can be stored anywhere in memory, provided that the addresses are not required by the C64 for other purposes. One point to watch out for is to make sure that you do not use memory required by the assembler.

As you become more experienced, you will get to know the C64's memory better, but for now we suggest you stick to the 4K block beginning at **$C000**. All the examples in this book use this block of RAM.

6.4 Relative

Relative addressing is the name given to all the branch instructions. They are so-called because they branch ***relative*** to the current program location. As we saw in Chapter 5, the assembler can jump up to 127 bytes either forwards or backwards. The calculation of how many bytes, and in which direction, to jump is performed automatically by the assembler.

Let us see how this works in practice

```
7000 LDA #$50
7010 CMP $25
7020 BEQ EQUAL
7030 RTS
7040 LDA $0400:EQUAL
```

This program simply compares the value of location **$25** (decimal 37) to the value **#$50** (decimal 80). If the two are equal (that is, location **$25** contains the value **#$50**), the program branches to line 7040. Otherwise it continues to line 7030 where it returns to BASIC.

When you assemble the program, the assembler calculates that line 7040 (labelled EQUAL) is one byte ahead of the current program position. (The reason that it is one, rather than two, bytes ahead is that the RTS command is the next line due to be executed and is therefore zero bytes ahead.) It thus converts line 7020 to read, in effect

```
7020 BEQ+$01    ;NB: This line is not valid, it simply illustrates the principle of
                what happens
```

The assembler supplied with this course is known as a 'two-pass' assembler. That is, it assembles the program in two stages. In the first stage (or pass), it assembles the commands but leaves the labels as they are. In the second pass, it calculates the values of all the labels and inserts these values in place of the labels. There is another type of assembler known as a single-pass or simple assembler. These do not allow the use of labels and are thus useless for anything but the smallest programs and persevering programmer!

The advantage of relative addressing, if used throughout a program, is that it is **relocatable**. This means that you can shift the program anywhere in memory without re-assembling it. You must, however, use relative addressing throughout, so you cannot use JMP or other absolute instructions. Writing relocatable programs requires considerable skill, but is ideal for short machine code utilities which other users might want to place somewhere else in memory to avoid conflicting with their own routines.

6.5 Implied

Implied addressing is one of the simplest addressing modes. It is the name given to all instructions which have no target, source or branch bytes following them. So far, the implied commands we have looked at include the transfer commands (TAX, TXA, TAY and TYA), the increase and decrease commands (INX, INY, DEX and DEY) and RTS. They are known as implied commands because they imply the value and/or address to be used. For example, INY means increase the Y index by one and place the result back into the Y index.

6.6 Absolute,X and Absolute,Y

We have already seen that the X and Y indices can be used as an alternative to the accumulator, but they also have their own special uses. The first of these are Absolute,X and Absolute,Y modes. Commands in this mode take the form

```
LDA ⟨address⟩,X
LDA ⟨address⟩,Y
```

where the address is in the range **$0100–$FFFF**. This mode is equivalent to the BASIC statement

```
100 POKE 1024+X,100
```

where the X in the above example would be replaced by the X or Y index. So, to achieve the equivalent of the above line in machine code you would

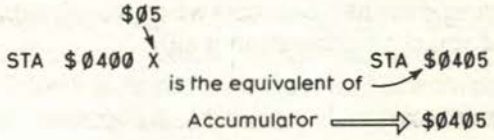
```
6010 LDA #$64      ;Set accumulator to decimal 100
6020 STA $0400,X   ;Add value of X index to the address and then carry out the
                    command
```

In other words, if the X index were set to **$10**, the above example would be equivalent to

```
6020 STA $0410
```

If you would like a more formal definition, the value of the index is added to the absolute number following the instruction, and the instruction is carried out on the resultant value. Figure 6.1 illustrates this process. Since both indices can hold up to **$FF** (decimal 255), the range of addresses you can control in this mode is ⟨**address**⟩ to ⟨**address**⟩+**$FF**.

Figure 6.1



Ok, so much for how it works (and if you are still not sure, the following program will make things clear), why would you want to use it? Well, as you may have already guessed, it is very useful for writing loops. Suppose you wanted to write a program to fill the first 100 positions of the screen with a reversed yellow block. In BASIC, we would write the program like so

```
100 FOR I=0 TO 99
110 POKE 1024+I, 160:REM 160 = character code for reversed block
120 POKE 55296+I, 7:REM 7 = colour code for yellow
130 NEXT I
```

In machine code, we would do it in Absolute,Y mode like this

```
6000 [ $C000
6010:SCREEN$0400                ;Define label as start of screen RAM
6020:COLSCREEN=$D800            ;Define label as start of colour RAM
6030;
6040 LDY #$00                   ;Initialize Y to zero
6050 LDA #$A0:LOOP              ;Load accumulator with decimal 160
6060 STA SCREEN,Y               ;Store reversed block in screen RAM
6070 LDA #$07                   ;Load accumulator with decimal 7
6080 STA COLSCREEN,Y            ;Store yellow code in colour RAM
6090 INY                        ;Increase the Y index by one
```

```
6100 CPY #$64                ;Check for decimal 100
6110 BNE LOOP                ;Branch back to loop if less than 100
6120;
6130 RTS
```

If you want to see this in action (and it is worth seeing the speed!), assemble it, clear the screen and SYS 49152. The first 100 character positions will turn almost instantaneously yellow. A very similar task to this, filling colour RAM with a particular colour, is of course a common requirement in many programs. In BASIC, we would do it like this

```
100 FOR I=0 TO 999:POKE 55296+I, 8:NEXT I:REM Orange
```

For other colours, of course, you would substitute another colour code for the 8 – see Appendix 4 for a list of colour codes.

Even though we are POKEing, this still takes a crawlingly slow 10.18 seconds to execute: not much good for swift colour changes in all-action games! The solution, of course, is to rewrite the above in machine code and SYS this routine instead. The machine code equivalent looks like this (this program introduces some new concepts which we will explain in a moment, so don't worry if you don't understand it all)

```
6000 [ $C000
6010:COLSCREEN=$D800         ;Start of colour RAM
6020;
6030 LDY #$00                ;Load Y index with zero
6040 LDA #$07                ;Load accumulator with 8 (orange)
6050 STA COLSCREEN,Y:LOOP    ;First quarter of colour RAM
6060 STA COLSCREEN+$0100,Y   ;Second quarter of colour RAM
6070 STA COLSCREEN+$0200,Y   ;Third quarter of colour RAM
6080 STA COLSCREEN+$02E8,Y   ;Final quarter of colour RAM
6090 DEY                     ;Decrease Y index by one
6100 BNE LOOP                ;Branch back to LOOP
6110;
6120 RTS
```

This takes 0.17 second to execute – an impressive difference!

Let us go through the above program in detail. There are four specific points you may have noticed:

1. The accumulator is initialized ***outside*** the loop. This means that it is only initialized once; if it had been inside the loop, it would have been initialized 255 times. Although loading a value into the accumulator takes only a minute fraction of a second, a minute fraction of a second multiplied by 255 can become quite a significant delay. Even machine code can be slowed down by careless programming.
2. We have used the fact that the accumulator, like the indices, retains its

value once set until loaded with a different value. (STA, remember, places a ***copy*** of the accumulator value in the specified address: it leaves the accumulator untouched.) We do not need to reload the accumulator with the colour code each time.

3. You will almost certainly have noticed that we have a BNE instruction without a CPY. This is because the decrease command (DEY) automatically sets the zero flag, and the BNE instruction will therefore compare the Y index with zero without a CPY **#$00** command. The same is true of the increase command, INX. You can, of course, put the CPY or CPX commands in anyway, and you may find it helps to do this to start with, but remember that every command you can eliminate will speed up your program, particularly within loops.
4. The screen is shaded in roughly four quarters. There is a slight overlap in that some parts of the screen get shaded twice, but the time loss in this is less than the time loss in writing a more elaborate loop to check for this.

Short routines like this are very handy to slot into your BASIC programs. By far the easiest way to do this is to convert the program into DATA statements, POKE it into RAM and then SYS it. Of course, converting the program manually would be a tedious task, so we have written a program to do the job for you – you will find it in Appendix 5. One word of warning concerning this mode. You must make sure that the result of a calculation does not exceed **$FFFF**; if it does, the result will roll-over into zero-page memory and your program will probably crash. It certainly will not work in the way you wanted it to!

6.7 Zero-page,X

This works in an identical fashion to Absolute,X except that it operates on zero-page addresses. Thus

```
LDA $D0,X
STA $FF,X
```

Note that the Y index cannot be used with the accumulator instructions in this mode. You can, however, achieve the same effect using the X index

```
LDX $D0,Y      ;Load X index with content of $D0+Y
TXA $D0        ;Transfer this value to the accumulator
```

6.8 Accumulator

The accumulator addressing mode is the name given to machine code commands which act directly on the accumulator. These include ROL and

ROR, and are discussed in Chapter 8. Appendix 8, detailing all the commands in the 6510 instruction set, shows which commands act on the accumulator.

6.9 Lo–Hi form

Lo–Hi form is not an addressing mode, but we need to discuss it before going any further since it is used by all the remaining addressing modes. Lo–Hi is the form in which the C64 stores numbers greater than **$FF** . Since each memory location can only hold one byte, that is a value up to decimal 255 (**$FF**), the C64 needs some way of handling numbers greater than this. It does so by using two consecutive locations. Let us look at an example

LDA $D000

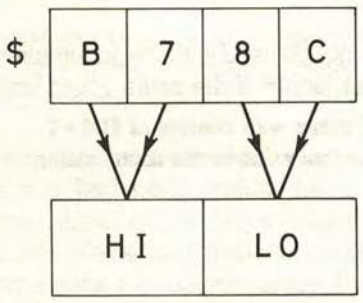
This command is stored in three consecutive bytes. The first byte is the code for the instruction itself, in this case **$AD** (decimal 173), the code for LDA. This is the code for absolute mode. The two remaining bytes are used to store the two halves of the number: **$D0** and **$00**. The reason for the name Lo–Hi, however, is that the low-byte, that is the smaller of the two numbers, is stored first (Fig. 6.2). So **$D000** is stored as **$00** and **$D0**. Thus the whole line of code is stored like so

$AD $00 $D0

Lo–Hi form may seem a little awkward at first, but it is simple enough once you have grown used to it.

To enable you to split up an absolute number into its Lo- and Hi-bytes, we have written two special commands into your assembler. These are ⟨and⟩. To load the accumulator with the Lo-byte of an address

Figure 6.2



LDA #<$0400

and the Hi part

LDA #>$0400

You can also do the same with labels, thus

LDA #>SCREEN

6.10 Indirect,Y

The first of the addresses to use Lo–Hi form is indirect, Y mode. Indirect, Y is one of the most useful commands you will come across once you start writing anything other than very simple programs because it allows you to act on large amounts of memory.

This mode uses two consecutive zero-page addresses to point to another address in memory like so

LDA ($FE),Y

The brackets around the address tells the assembler that a Lo–Hi form number has the Lo-byte stored at that address and the Hi-byte at the following address (in this case, **$FF**). Let us suppose that **$FE** contains **$00** and **$FF** contains **$04**. When we convert these two bytes from Lo–Hi form into absolute, we get **$0400**. In other words,

LDA ($FE),Y

is the same, in this case, as

LDA $0400,Y

which we looked at earlier in this chapter. Incidentally, in the above example, we happened to use **$FE** and **$FF** as the pointer to an absolute address, but you can use any free addresses in zero-page, bearing in mind that you need two ***consecutive*** free locations. Free addresses are shown in Appendix 2.

To illustrate the use of this mode, let us write a short routine to clear the high-resolution screen (which starts at **$8000**).

```
6000 [ $C000
6010:HIRES+$8000
6020 LDA #<HIRES
6030 STA $F7
6040 LDA #>HIRES
6050 STA $F8
6060 LDX #$00                ;USE X index as counter
6070 LDY #$00:LOOP1
```

```
6080 LDA #$00
6090 STA ($F7),Y:LOOP2
6100 DEY
6110 BNE LOOP2
6120 INC $F8          ;Increase Hi-byte
6130 INX              ;Add one to counter
6140 CPX #32          ;Finished?
6150 BNE LOOP1        ;No - go back to beginning of loop
6160 RTS              ;Yes - end
```

The important point about this program is that the indirect,Y command in line 6090 is used in place of the hundreds of absolute,Y which we would have otherwise needed.

6.11 Indirect,X

Indirect,X is – in contrast to indirect,Y – one of the least-used modes. It works in a slightly more complicated way to indirect,Y. A pair of consecutive zero-page locations are once again used as a pointer, but the resultant address forms another, second, pointer. Let us see what this means with an example:

```
LDA ($FE,X)       ;Note that the X is inside the brackets
```

If **$FE** holds **$00**, and **$FF** holds **$60**, the resultant address is **$6000**. This address, however, is used as a second pointer and the following operation is, in effect, carried out:

```
LDA ($6000,X)     ;NB:Not a legal command
```

Now, let us imagine the addresses around **$6000** hold these values:

Location	Value
$6000	$00
$6001	$04
$6002	$10
$6003	$04

If the X index were set to 0 (zero) when the **LDA ($FE,X)** command were carried out, the accumulator would be loaded with the contents of address **$0400**. If the X index equalled 1, the accumulator would be set to **$1004**. This is best shown in the following table:

X index value	Address loaded into accumulator
0	$0400
1	$1004
2	$0410

and so on. Compare this table with the one above to see how it works. If you want to set up a series of pointers in this way (and it is very rare to need to do so), make sure that the X index holds an even value to ensure the correct address is loaded.

6.12 Indirect

The last of our addressing modes is indirect. This is only available when using the JMP (JuMP) command and is often used in the C64's BASIC and Kernal routines. It is best left until you are fairly experienced, but comes in very useful in writing programs which will automatically adjust themselves to suit their requirements (simply changing a single memory location causes the pointer to lead to a completely different routine).

The command stores a pointer in an absolute address. The format of the command is

JMP ($6000)

If **$6000** contains **$00** and **$6001** contains **$C1**, the C64 would jump to the routine beginning at **$C100**.

6.13 Summary

The C64 has eleven different addressing modes (ways of treating memory). Each addressing mode has at least one command associated with it. Which command uses which addressing mode can be seen in Appendix 8.

We have seen new ways of using familiar commands, depending on the mode we are in. You do not need to worry about the names of the different modes, but you should know how to use the new command structures.

The chapter contains a lot of information, which you probably will not take in at one go. You might like to take a break at this point and re-read the chapter again later before continuing. We know this process of reading and

re-reading can seem tedious at times, but you will be glad you made the effort later on.

6.14 Exercise

Write a program to put the C64 character set on screen (starting in top left) in yellow.

7

Bit manipulation and logic (or 'truth') tables

This chapter explains the concepts of bit manipulation and logic (or 'truth') tables. This allows us to control individual bits within any given memory location. By the time you have completed the chapter, you will be able to answer the following questions:

- What is bit manipulation?
- What are the two main reasons for using it?
- What is a logic or truth table?

You will also be able to use the following machine code commands:

- AND
- ORA
- EOR

7.1 What is bit manipulation?

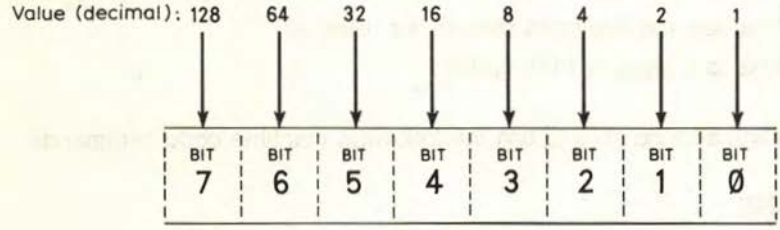
When we program in BASIC, we normally think of memory in terms of bytes, each location or address occupying one byte. But each byte is, of course, made up of eight bits (**bi**nary digits, remember). Thus decimal 10 is stored as **#%00001010**.

When we program in machine code, it is often useful to think of memory locations as eight bits instead of a simple byte. Rather than altering value as a whole byte, we may want to simply change a single bit. There are two main reasons for wanting to do this. First, there are certain bytes (known as **registers**) which are designed to be controlled by setting and resetting individual bits. Second, using individual bits to store simple on/off flags can give considerable memory savings. For example, if you are writing an adventure game and a certain room in it has four doors, you could use four bits of the same byte to store the open/closed door flags:

Door 1	Door 2	Door 3	Door 4	Spare	Spare	Spare	Spare
1	0	1	1	0	0	0	0

You could even use the four spare bits as other flags, perhaps for objects or characters. If you use all eight bits, you can store information in an eighth of the space you would use if you used a whole byte for each flag (Fig. 7.1). Put another way, your adventure could be eight times as big and still fit into the same memory! Bit manipulation is pretty useful! Bit manipulation, as you have probably already guessed, is performed in binary. So if you are less than 100% confident about your understanding of binary numbering, go back now and re-read Chapter 3!

Figure 7.1



Remember that binary numbers are preceded by a percentage sign ('%'), just as hex numbers are preceded by a dollar sign ('$'). So, for example

```
LDA %00000011
```

would load the content of location decimal 3 into the accumulator, while

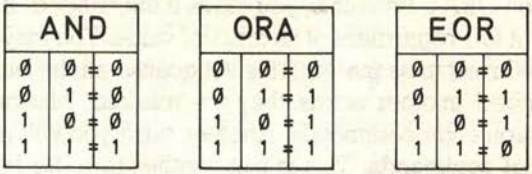
```
LDA #%00000011
```

would load the accumulator with the value decimal 3. Bit manipulation is actually carried out using logic tables, so let us find out about these.

7.2 Logic tables

Logic tables are simple tables (Fig. 7.2) showing what happens to a bit when it is acted on using one of the bit manipulation commands. Let us look in detail at each of the three bit manipulation commands, starting with ORA.

Figure 7.2



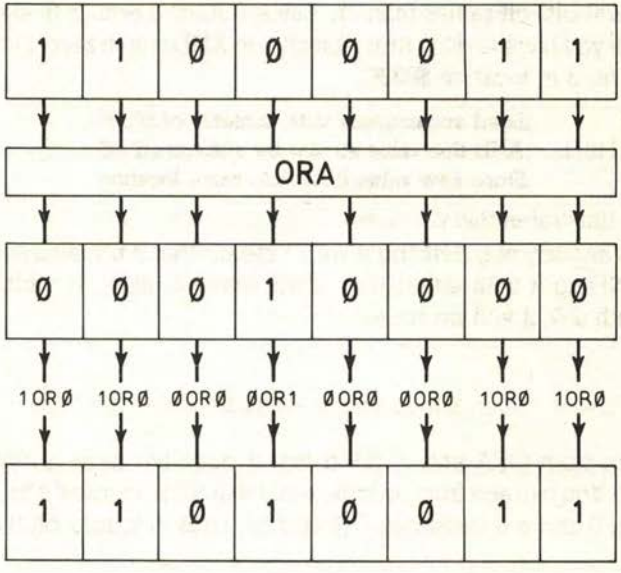
7.3 ORA

The ORA command allows you to set (make equal to 1) any individual bit or bits in a byte. For example, suppose that you wanted to set bit four in location **$033C** (decimal 828). You do not even need to know the current value of the bit, you can simply set it anyway: if it is not set (0) it will be set (1), and if it is already set it will stay that way. To set bit four

```
LDA $033C        ;Load current value into accumulator
ORA #%00010000   ;Set bit 4
STA $033C        ;Store new value at the same address
```

Let us suppose the original value of **$033C** was **%11000011**. After the ORA command, setting bit 4, it would become **%11010011**. Figure 7.3 shows how

Figure 7.3



this works. What happens is that the ORA logic table is performed on each bit in turn. If you ORA a bit with 0, you leave it unchanged. If you ORA a bit with 1, you set it to 1 regardless of its original value. You may hear of binary digits being referred to as **masks**. This is because all the bits ORAd with 0 are left unchanged: in other words, they are 'masked'. Masking is used with all the bit manipulation commands. Another term you will probably come across is **logical commands**. This is just another term for bit manipulation commands.

You will use the ORA command a lot when you use the C64's video and sound registers. For example, suppose you wanted to turn on sprites 3 and 6. To do this, you simply set bits 3 and 6 of the sprite-enable register **$15** (decimal 21). Using ORA to do this means that you do not run the risk of mistakenly turning off other sprites already in use. So, for example, if you had defined the label VIDEO as the beginning of the video registers, you would simply

```
LDA VIDEO+21    ;Load current value of sprite-enable register
ORA #%01001000  ;Set bits 3 and 6
STA VIDEO+21    ;Store new value in register
```

You do not risk changing any of your other sprites, because only bits 3 and 6 will be affected: all the others remain unchanged.

7.4 AND

AND works in exactly the same way as ORA, except that it is used to switch a bit or several bits off rather than on. Since 0 AND 0 equals 0, and 0 AND 1 equals 0, all you have to do to turn a bit off is to AND it with zero. For example, to turn off bit 3 in location **$033C**:

```
LDA $033C       ;Load accumulator with contents of $033C
AND #%11110111  ;AND this value so that bit 3 is turned off
STA $033C       ;Store new value back into same location
```

Figure 7.4 illustrates this process.

If a bit is already set, ANDing it with 1 means that it remains set. If a bit is not set, ANDing it with either 0 or 1 will leave it unset. If a bit is set and ANDed with a 0, it will be unset.

7.5 EOR

EOR differs from ORA and AND in that it does not have a direct BASIC equivalent. You can see from its logic table that EOR 'toggles' a bit, so that a 1 becomes a 0 and a 0 becomes 1 (Fig. 7.5). Thus to toggle bit 6 of location **$033C**

Figure 7.4

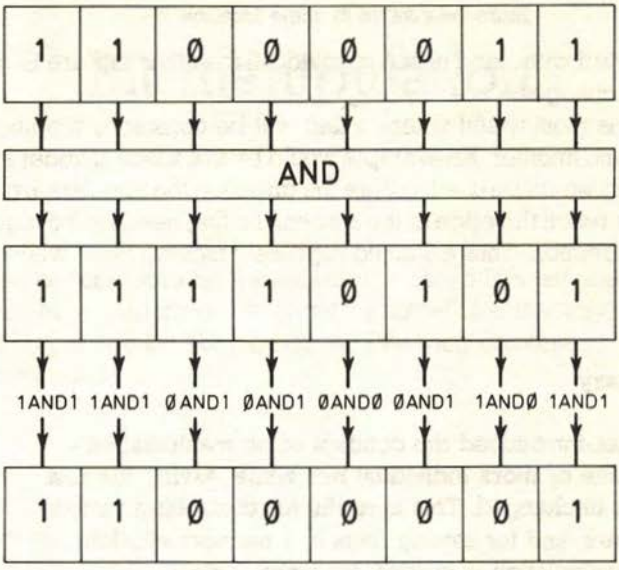
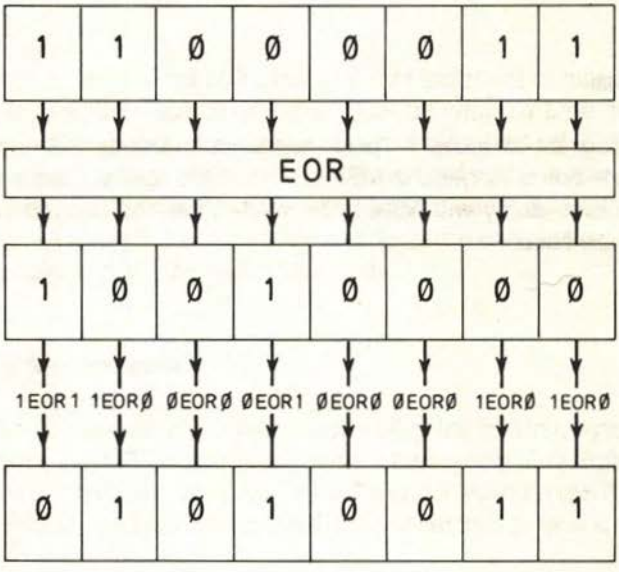


Figure 7.5



```
LDA $033C          ;Load accumulator as before
EOR #%01000000     ;Toggle bit 6, leave other bits unchanged
STA $033C          ;Store new value in same location
```

Bit 6 is EORed with 1 and is thus toggled, all the other bits are EORed with 0 and left unchanged.

EORing is most useful where a flag will be constantly flipping between one value and another. An example would be in a space-invader style game, where a flag would be used to store the direction the invaders are moving in. When they reach the edge of the screen, the flag needs to be toggled to the opposite direction. Thus a 0 could represent moving right, while a 1 means moving left.

7.6 Summary

This chapter introduced the concept of bit manipulation – changing one or more individual bits while leaving the rest of the byte unchanged. This is useful for controlling various C64 registers, and for storing flags in a memory-efficient way. Bit manipulation is carried out using logical commands and binary numbers, the technique being known as masking. The logic tables shown in this chapter illustrate how the three logical commands – ORA, AND and EOR – work. The following chapter examines more advanced bit manipulation.

7.7 Exercise

Write a program to move a sprite horizontally across the entire screen making sure not to forget the MSB bit when the sprite passes the X=256 position. A time delay will have to be written into the program so that the sprite can be seen.

8

Bit manipulation

This chapter covers using bit manipulation to move bits left and right, and check whether a particular bit is set or unset. By the time you have completed it, you will be able to use the following commands:
The shift commands

- **LSR**
- **ASL**

The rotation commands

- **ROR**
- **ROL**

Plus

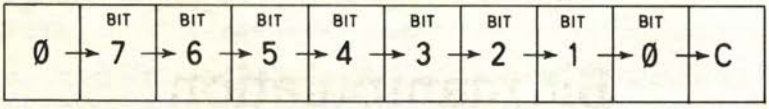
- **BIT**

In Chapter 7, we looked at logical (bit manipulation) commands to set, unset and toggle the value of any individual bit within a byte of the C64's RAM. This chapter introduces commands which allow us to shift all the bits in a byte left or right, and to check whether an individual bit is currently set or unset. These commands all use the accumulator addressing mode described in Chapter 6. As before, we will describe each in turn, beginning with the simplest of all: the shift commands.

8.1 The shift commands

The two shift commands allow you to move all of the bits in a byte either left or right. These are LSR and ASL. LSR stands for Logical Shift Right. It moves all eight bits of a byte one bit to the right. The right-most bit (bit 0) 'falls over' into the carry flag, and the left-most bit (bit 7) is set to zero as is shown in Fig. 8.1.

Figure 8.1

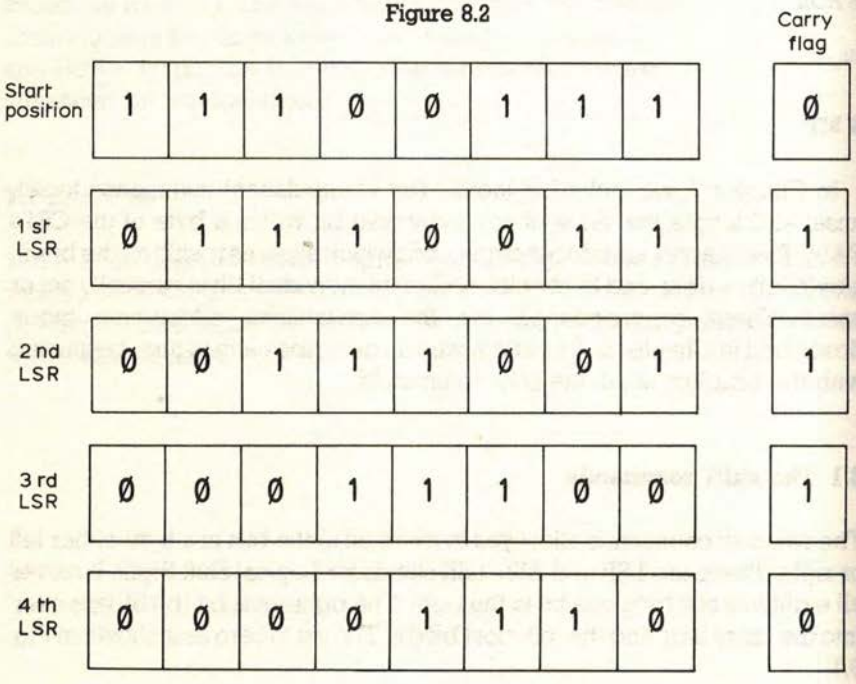


LSR can be used in a somewhat clumsy way to test the status of any bit. The way to do this is to use LSR as many times as required to move the bit you wish to test into the carry flag. Thus to test bit 3 of location **$033C**, you would load the value into the accumulator, perform four LSRs on the value (to move bit 3 into the carry flag) and then use BCC (Branch if Carry Clear) or BCS (Branch if Carry Set) to test the value

```
LDA $033C      ;Load the required value into the accumulator
LSR A          ;Bit 0 now in carry flag, bit 3 becomes bit 2
LSR A          ;Bit 1 in carry flag, original bit 3 becomes bit 1
LSR A          ;Bit 2 in carry flag, original bit 3 becomes bit 0
LSR A          ;Bit 3 is now in the carry flag
BCC ZERO       ;Branch to line labelled ZERO if carry flag=0
```

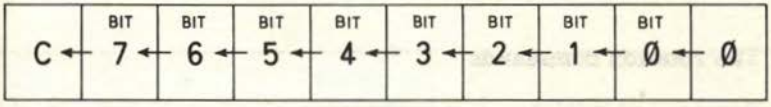
This process is illustrated in Fig. 8.2. The program would thus jump to the line labelled ZERO if the original bit 3 was not set (that is, it equalled zero). The

Figure 8.2



value of **$033C**, of course, remains as it was, so bit 3 is still where it originally was in the actual location being tested. Only the accumulator has changed value. The A following the LSR command tells the assembler to use the accumulator mode.

Figure 8.3

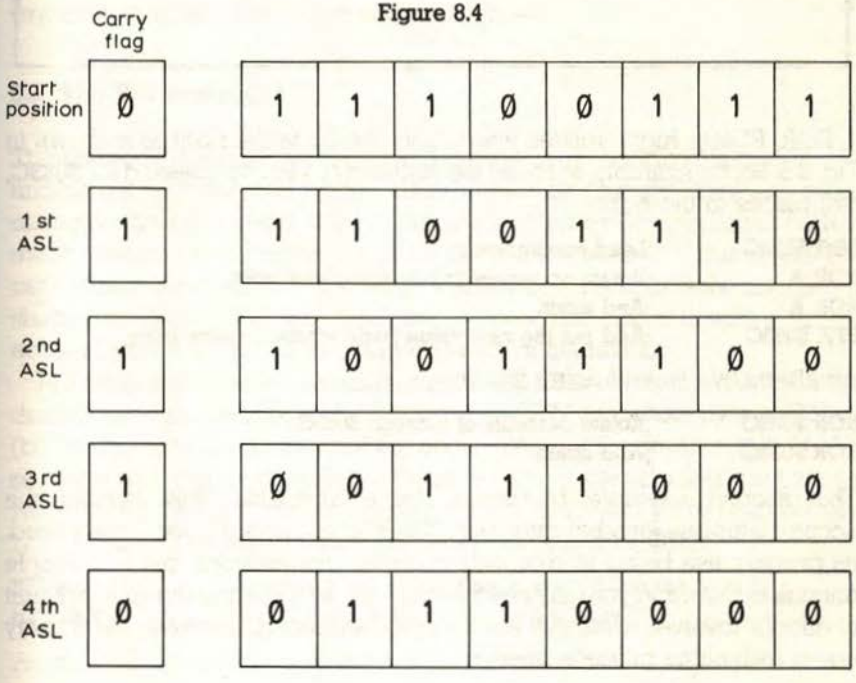


ASL, Arithmetic Shift Left, works in exactly the same way as LSR, but shifting to the left instead of to the right. ASL shifts each bit one position to the left, bit 7 moving into the carry flag and bit 0 being set to zero (Fig. 8.3). Again, to test a bit using ASL, you would shift the required bit into the carry flag. To test bit 5 of our old friend **$033C**, for example

```
LDA $033C       ;Load value
ASL A           ;Bit 7 moves into carry flag
ASL A           ;Bit 6 moves into carry flag
ASL A           ;Bit 5 moves into carry flag
BCC ZERO        ;Branch to line labelled ZERO if bit 7 was zero
```

Again, we can show this in diagrammatic form in Fig. 8.4. Which of the two shift

Figure 8.4

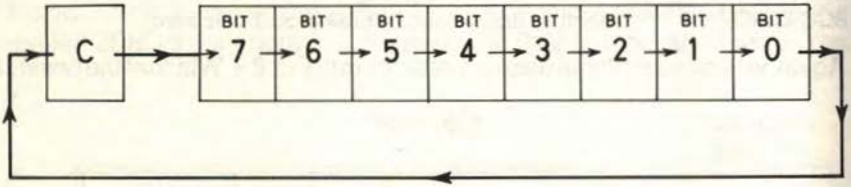


commands you use is up to you. We suggest that it makes sense to use LSR if the bit you want to test is nearer to the right of the byte (that is, bits 0, 1, 2 and 3) and ASL if the bit is nearer to the left (that is, bits 4 and 5 – we will show you a simpler way of testing bits 6 and 7 below, using the BIT command). In this way you use the minimum amount of memory necessary.

8.2 The rotation commands

The rotation commands are similar to the shift commands in that they allow you to shift bits left and right. The difference is that the rotation commands, as the name implies, shift the bits around in a circle. The easiest way to see what we mean is to look at Fig. 8.5. Briefly, the rotation commands shift all the bits round in a circle which includes the carry flag. So if you rotate everything once to the right, bit 0 will become the carry flag, and the carry flag is moved into bit 7. All other bits move down one.

Figure 8.5



ROR, ROtate Right, rotates everything one bit to the right as is shown in Fig. 8.5. So, for example, to rotate the contents of – you've guessed it – **$033C,** two places to the right

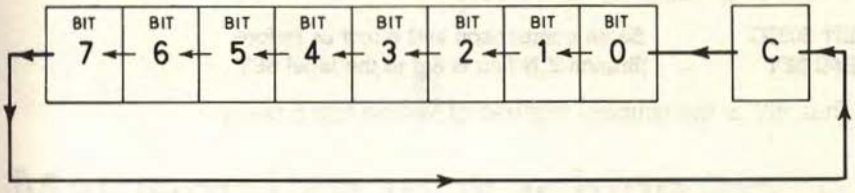
```
LDA $033C       ;Load accumulator
ROR A           ;Rotate accumulator bits one place right
ROR A           ;And again
STA $033C       ;And put the new value back where it came from
```

An alternative way of doing this is

```
ROR $033C       ;Rotate contents of location $033C
ROR $033C       ;And again
```

This second example by-passes the accumulator, thus leaving the accumulator free for other purposes. The ROR command is very rarely used, its primary use being in relatively complex mathematics, but it is simple enough to learn and you may need it someday, so it is worth the small amount of effort it involves. Until you are more experienced, however, file it away under 'will come in useful later'.

Figure 8.6



ROL, ROtate Left, is the complement of ROR. It rotates all the bits of the specified byte one place to the left. Bit 7 moves into the carry flag, and the carry flag moves into bit 0 as shown in Fig. 8.6. So to rotate two places to the left

```
LDA $033C       ;Load accumulator
ROL A           ;Rotate accumulator bits one place to the left
ROL A           ;One more time
STA $033C       ;Put the new value back where it belongs
```

or simply

```
ROL $033C       ;Rotate value of location one place left
ROL $033C       ;Once again
```

As with ROR, there are other uses for ROL, but these should be left until your experience in using machine code has grown.

8.3 The BIT command

The methods given above for testing the status of an individual bit (that is, finding out whether it is a one or zero) are a bit clumsy. They also mean changing the value of either the accumulator or a memory location or both. A much neater way of testing just bits 6 and 7 is to use the BIT command. BIT cannot be used to test any other bits. The BIT command uses the zero flag to show the status of a bit. Because it does not affect memory or the accumulator, it is known as a **non-destructive** command.

BIT is not available in immediate mode (see Chapter 6), so it must be used directly on a memory location. When you use BIT, the ***whole byte*** is ANDed (bit-by-bit) with the contents of the accumulator. If the result of this AND is zero, the zero flag is set to zero; if the result is one, the zero flag is set to one. Also, the sixth and seventh bits are moved into the N and V bits respectively. Let us use BIT to test the sixth bit of our old favourite, **$033C**

```
BIT $033C       ;Move bit 6 into the V flag, and bit 7 into the N flag
BVS SET         ;Branch if the V flag is Set to the label SET
```

Although BIT uses the accumulator as the comparison, it does not matter

what the accumulator is set to since we are only interested in the values of the V or N flags. Similarly, to test bit 7

```
BIT $033C        ;Same comparison and effect as before
BMI SET          ;Branch if N flag is Set to the label SET
```

Thus BIT is the simplest method of testing bits 6 and 7.

8.4 Summary

In this chapter we introduced the bit manipulation commands for moving all the bits in a byte left or right. The main purpose for this is to test the value of an individual bit. LSR and ASL are the normal commands used to test the values of bits 0 to 5, while BIT is usually used to test bits 6 and 7. The rotation commands, ROR and ROL, are normally reserved for complicated mathematics: we do not recommend using them until you are fairly experienced.

8.5 Exercise

Write a program to copy the actions of eight ROR commands on the contents of location 828 using the LSR command.

9

Mathematics in machine code

This chapter introduces mathematics in machine code. By the end of it, you will know how to use machine code for

- addition
- subtraction
- multiplication
- division

of both 8- and 16-bit numbers.

Don't panic! Machine code maths is easy once you've got the hang of it.

We give you this helpful advice at the beginning of the chapter because machine code maths has an undeserved reputation for being difficult. It's not, it just takes a bit of getting used to, that's all. In BASIC, maths is simple. You just use the relevant BASIC keyword or symbol together with any necessary parameters and the complete calculation is done for you. Thus no sooner have you entered

```
PRINT (79*(COS(5)+.98))/7-(2*.47)
```

and back comes the answer 21.36276874 (you always wanted to know that, didn't you?).

In machine code, however, there are only two formal arithmetic commands, for addition and subtraction. Multiplication and division have to be done using these and other commands. Also, the method used for any arithmetic operation depends on whether you are working with an 8- or 16-bit number. That's the bad news. The good news is: take this chapter nice and slowly, stopping after each example to make sure you understand it completely, and you will wonder what all the fuss was about. We are going to start with 8-bit arithmetic (that is, working with numbers in the range 0–255), and then move onto 16-bit work (numbers in the range 0–65535).

9.1 Eight-bit addition

Eight-bit addition is the addition of any two numbers in the range **$00–$FF** (decimal 0–255). To add two numbers together in BASIC, we would do this

```
A=200+20:REM A now equals 220
```

In machine code, we use the command ADC (ADd with Carry). This adds the specified value to the value in the accumulator, then places the result into the accumulator. So to add 200 (**$C8**) and 20 (**$14**)

```
LDA #$C8        ;Load first value into the accumulator
ADC #$14        ;Add the second value to the accumulator
```

The accumulator now contains the value **$DC** (**$C8** plus **$14**).

So what's all this 'with carry' bit? Well, the above description is a very slight simplification of what actually happens. ADC actually adds the specified value, the content of the accumulator ***and*** the content of the carry flag. So if the carry flag had been set to 1, the previous example would have left the accumulator with a value of **$DD** (**$C8** plus **$14** plus **$01**).

To get around this problem, when we do not want to add-in the value of the carry flag, we simply clear it first using the CLC (CLear Carry) command

```
LDA #$C8
CLC
ADC #$14
```

Figure 9.1

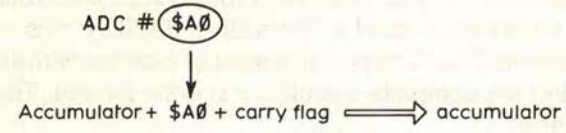


Figure 9.1 shows this process. You can also use the ADC command to add one to the accumulator. You may remember from Chapter 5 that while INX will increase the X index by one, and INY increases the Y index by one, there is no INA command. Well, you achieve the same affect by setting the carry flag to one and then performing an ADC to add zero to the accumulator

```
SEC             ;SEt Carry flag to 1
ADC #$00        ;Add zero plus the carry flag (1) to accumulator
```

Thus the accumulator is increased by one (original value plus zero plus the value of the carry flag, which we set to one).

The ADC command uses roll-over to cope with values larger than **$FF**.

Thus

```
LDA #$C8        ;Load accumulator with decimal 200
CLC             ;CLear Carry
ADC #$C9        ;Add decimal 201 to accumulator
```

would leave decimal 145 in the accumulator (200+201−256=145). To show that roll-over has occurred, the carry flag is set to one. So if you perform any addition where the result could be greater than 255, you will need to test the carry flag to make sure the result is correct. You do this using the BCC or BCS commands.

9.2 Eight-bit subtraction

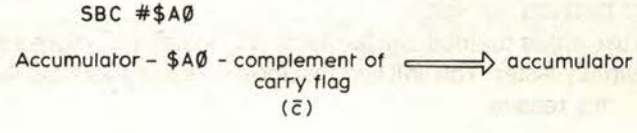
The command for subtracting two numbers in the range **$00** to **$FF** is SBC: SuBtract with Carry. This command subtracts the specified value from the accumulator, then places the result into the accumulator. Again, the carry flag also comes into the calculation but ***the inverse*** of the carry flag is subtracted, and ***not*** the carry flag itself. So, if the carry flag was set to zero, an extra one (the inverse of the carry flag) would be subtracted from the accumulator. If the carry flag was set to one, nothing extra would be subtracted (the inverse of one is zero). There is a good reason for subtracting the inverse, which we will explain in a moment.

It's obvious from this that, just as we had to clear the carry flag before adding, we must set it before subtracting

```
LDA #$CB        ;Decimal 203
SEC             ;SEt Carry flag to one, so that inverse is zero
SBC #$C8        ;SuBtract decimal 200 and inverse of carry flag
```

This would leave the accumulator set to 3 (203-200-0) as shown in Fig. 9.2. If you try to subtract a number larger than the value of the accumulator, roll-under occurs. If this happens, the carry flag is set to zero. This, of course, is the reason why the inverse of the carry flag is subtracted: if we left it at zero, we would not know if roll-under had occurred. Again, if your calculation could end up with a result less than zero, causing roll-under, you must check the carry flag using BCC or BCS.

Figure 9.2



9.3 Sixteen-bit addition

Sixteen-bit addition is just a simple extension of 8-bit addition. It is normally used to increase the Lo-Hi pointers (see Chapter 6). Let us see how we would add **$32** (decimal 50) to the contents of two locations labelled LO and HI. HI, of course, is LO+1. The two 8-bit numbers are really just a way of storing one 16-bit value.

Remember that Lo-Hi numbers are stored in reverse order. So **$033C** would be stored as **$3C** (in the Lo-byte) and **$03** (in the following Hi-byte). When we add a number to LO, we need to check the carry flag. If the carry flag is set, roll-over has occurred and we need to add one to HI. (This works in exactly the same way as manual addition, where you 'carry' one and add it to the next column up, or an abacus, where ten beads in one column are replaced by one bead in the next column up.) We do this like so

```
LDA LO          ;Load the accumulator with contents of LO
CLC             ;Clear the carry flag
ADC #$32        ;Add $32 to the accumulator
STA LO          ;Store the new result in location LO
BCC CLEAR       ;If carry flag is clear, skip next command
INC HI          ;Carry flag is set, so add one to value in HI
LDA #$20:CLEAR  ;If carry flag not set, program jumps to here
```

In other words, if the addition to the value stored in the location LO results in a value of greater than 255, the carry flag wil be set. The BCC test will fail and the INC command will be executed, increasing the value stored in location HI by one. If, however, the addition results in a value of 255 or under, the carry flag will not be set, the program will branch to the line labelled CLEAR and the value stored in HI will be left untouched.

A more elegant method is to alter the second part of the program like so

```
LDA LO
CLC
ADC #$32
STA LO
LDA HI          ;Load accumulator with value of location HI
ADC #$00        ;Add zero plus carry flag to the accumulator
STA HI          ;Store result back in location HI
```

If the addition to the LO component results in a value greater than 255, the carry flag is set to one and this value, the one, is added to the HI component. If the result was 255 or under, the carry flag remains set to zero and the HI component remains the same.

You can use either method, but the second is neater and, more importantly perhaps, slightly faster. You will find that most programmers use the second method for this reason.

9.4 Sixteen-bit subtraction

As with addition, 16-bit subtraction is a simple extension of 8-bit subtraction. Again, let us take the example of a pair of locations LO and HI, together forming a 16-bit number. To subtract **$32**, we would

```
LDA LO      ;Load accumulator with LO component
SEC         ;Set the carry flag to one
SBC #$32    ;Subtract $32
STA LO      ;Store the result back in location LO
LDA HI      ;Load accumulator with HI component
SBC #$00    ;Subtract zero plus value of the carry flag
STA HI      ;Store the result back in location HI
```

If the first subtraction resulted in a value of less than zero, thus causing roll-under, the carry flag is set to zero. The inverse of zero, i.e. one, is then subtracted from the HI component. If the first subtraction did ***not*** cause roll-under, the carry flag remains set to one and the inverse, i.e. zero, is subtracted from the HI component leaving it as it was.

9.5 Multiplication and division by two

In Chapter 8, we looked briefly at the shift commands ASL and LSR. We showed how they could be used to detect the status of a bit, but we can also use them to multiply and divide by two (the more mathematically inclined of you will probably have already realized how this is done). To multiply a number by two, simply use ASL. This shifts everything one place to the left and thus the value of each bit is doubled. In other words each bit, and therefore the whole byte, is multiplied by two. Remember that the left-most bit is shifted into the carry flag. If the carry flag is set after an ASL, therefore, you know that the new value is greater than **$FF** (decimal 255).

So, to multiply the contents of location **$033C** by two, simply

```
ASL $033C
```

Of course, if you shift left again, you again multiply by two. So two ASLs is the same as multiplying by four

```
ASL $033C   ;Multiply by two
ASL $033C   ;Multiply by two again, that is multiply by four
```

and so on in powers of two. So three ASLs would multiply by eight, four ASLs by sixteen, and so on. Do not forget that you would need to check the carry flag after ***each*** multiplication to check whether an overflow occurred.

9.6 Division by two

Division by two is, of course, the opposite of multiplication by two. So instead of using the ASL command, to shift left, you use LSR to shift right. As with multiplication, if you shift repeatedly, you divide by increasing powers of two. Thus if you use LSR three times, you will divide the byte by eight (2^3). So, to divide a byte by two

```
LSR BYTE
```

Of course, because machine code maths can only work in integer arithmetic (that is, whole numbers), the BASIC equivalent of this is

```
A=INT(BYTE/2)
```

So if BYTE contained the value one, LSR BYTE would return a value of zero.

9.7 Summary

This chapter explained how to add and subtract in machine code, as well as how to multiply and divide by two. If you want to multiply or divide by more than two, you have to multiply or divide by two several times. So to multiply a value by nine, for example, you would multiply the value by two three times and then add the value: (value $\times 2 \times 2 \times 2$) + value = value $\times$ 9.

Remember that you must always set the carry flag to zero before adding, and to one before subtracting. You should then check the value afterwards in case the result was greater than 255 or less than zero.

9.8 Exercise

Write a program to multiply the single byte (up to 255) contents of location 828 by three. Note that such a program cannot cope when the contents of 828 are greater than 256/3.

10

Machine code subroutines

This chapter introduces machine code subroutines. When you have finished reading it you will:

- be able to write your own machine code subroutines
- be able to use the C64's built-in machine code subroutines
- know the difference between a subroutine and a macro
- be able to use the BASIC function USR to call machine code programs

10.1 Machine code subroutines

Subroutines are a convenient way of saving memory and dividing a program into manageable chunks. In BASIC, you use **GOSUB** ⟨**line number**⟩ to jump to a subroutine; you then write the subroutine beginning at that line number and ending with a RETURN statement. You save memory because you can use the same piece of code in two or more different parts of your program. Machine code subroutines operate in exactly the same way, GOSUB and RETURN being replaced by the exact equivalents JSR and RTS. JSR stands for Jump to SubRoutine, and RTS, as we mentioned earlier in the book, for ReTurn from Subroutine. RTS, of course, returns to BASIC if it is used outside a subroutine. Suppose you wanted to write a subroutine to clear the screen. The first thing to do is to choose a name for the routine. If you have used Simons' BASIC, or a similar extended BASIC, you will have come across procedures. Procedures are simply named subroutines. Instead of GOSUB 5000, for example, you might use PROC HISCORES, and instead of RETURN you would use something like ENDPROC. Machine code subroutines work in a similar way in that you give them a name, and then use this name to call the routine. All you do is label the first line of the routine

```
7000 ;CLEARSCREEN
```

Then, whenever you wanted to call the subroutine, you just

```
JSR CLEARSCREEN      ;Jump to the SubRoutine called CLEARSCREEN
```

Simple!

Some extended BASIC procedures allow you to pass parameters to them. To explain what we mean by this, suppose you wanted to write a procedure (subroutine) to centre a piece of text horizontally. This makes screen displays look neater. In the standard C64 Version 2 BASIC supplied with the machine, you would write a subroutine something like this

```
2000 REM Centre the text in A$
2010 LM=(40-LEN(A$))/2
2020 FOR A=1 TO LM:PRINT CHR$(32);:NEXT A
2030 PRINT A$
2040 RETURN
```

To use the subroutine, you would then have to place the text you want centred in **A$** and then call the subroutine

```
150 A$="***ALIENS ATTACKING!!!***"
160 GOSUB 2000
```

With procedures, however, we would write the subroutine like so

```
2000 DEFROC CENTRETEXT(A$):REM DEFine PROCedure called CENTRETEXT
2010 LM=(40-LEN(A$))/2
2020 FOR A=1 TO LM:PRINT CHR$(32);:NEXT A
2030 PRINT A$
2040 ENDPROC
```

And call it by

```
150 PROC CENTRETEXT("***ALIENS ATTACKING!!!***")
```

In this case, the text '*****ALIENS ATTACKING!!!*****' is a parameter which is passed to the subroutine.

Some assemblers allow you to pass parameters to machine code subroutines: these special subroutines are called **macros**. Note that the assembler supplied with this course does not support macros, as you are unlikely to need them while you are learning machine code. Once you have become more experienced, however, you may like to buy an assembler with a macro facility.

The two other important differences between a subroutine and a macro, besides the fact that macros allow parameters to be passed to them, are:

(a) Macros can be stored on disc or, at a push, on tape. Some macro assemblers supply you with a ready-made library of macros on disc: this is well-worth looking out for if you do choose to buy a macro assembler.

(b) When you call a macro, the machine code is inserted into memory at the

current position. This can be wasteful of memory, particularly since library macros are, by their very nature, designed to be as general as possible. On the other hand, this does mean that you can write a complete program just by joining different macros together and adding in a bit of your own code.

10.2 The Kernal jump table

The C64's Operating System is known as the Kernal. It is made up of hundreds of small subroutines which carry out simple tasks like printing characters to the screen, reading data from the datasette and so on. Rather than have to write our own machine code programs to do that sort of thing, wouldn't it be great if we could just borrow whatever subroutines we needed from the Kernal? Good news: we can. We just use JSR to jump to the location of the subroutine you want to use.

At first glance, there would seem to be a slight problem in this idea. After all, Commodore has brought out a number of different versions of the Kernal ROM, and has moved some of the subroutines around. Don't we have to know exactly which version of the C64 we've got, and then look up the address of each subroutine we want for that particular version? Thankfully, no! In what an unkind person might describe as a rare example of forward-thinking on Commodore's part (we, of course, don't think anything of the sort), the company foresaw this problem way back in the days of the PET (the ***what?***).

What Commodore did was to create a special area of the Kernal called the Kernal Jump Table. This table is in exactly the same place in every C64, and is still in the same place in the Commodore 128. This table contains a pointer for each subroutine in the Kernal. The pointer is simply the address where the subroutine can be found in this particular version of the Kernal. So, all you have to do is to JSR to the address of the subroutine you want in the Kernal Jump Table, and the C64 will then automatically transfer you to the correct address. So any machine code program written on any of the C64's or, indeed, on the C128, will run on any other version of the machine. This is one of the reasons that all C64 software runs on the C128. To call a Kernal subroutine, then, you only need to know its pointer address in the Kernal Jump Table. You can look up this address in the Kernal routines in Appendix 2 of this book. You set the accumulator and indices to the required values (these are normally passed to the Kernal routine as parameters) and then JSR to the appropriate pointer address.

Let us see how this works in practice. Supposing that we wanted to print a character to the screen at the current cursor position. To do this, we would use the Kernal routine called PRINT. This takes the value of the accumulator, converts it to the ASCII equivalent (see Appendix 9 for a list of ASCII codes)

and prints this character at the current cursor position. If you look up PRINT in Appendix 8, you will see that its pointer address is **$FFD2** (decimal 65490). So, to print the letter 'A' (**$41**, decimal 65), we would

```
LDA #$41        ;Load accumulator with ASCII A
JSR $FFD2       ;Jump to pointer address of the PRINT subroutine
```

And that's all there is to it! The Kernal jump table looks up the actual address of the PRINT routine for you, and this routine then prints the A to the screen.

10.3 USR

Until now, the way we have run our machine code programs from BASIC is to use SYS followed by the start address of the program. There is, however, an alternative method of doing it: the USR command. The syntax is

⟨variable⟩=USR (⟨parameter⟩)

So, for example

```
A=USR (10)
```

USR is intended for a machine code program which takes a value, processes it in some way and then returns a different value. In other words, it is for use when you have written a machine code function.

A function is simply a name for a subroutine which takes one value and returns a different one. An example of a built-in BASIC function is

```
A=RND (10)
```

So, for example, you may have written a program to draw a circle, with the centre of the circle at the current graphics-cursor position. You could use USR to tell your program what diameter circle to draw

```
220 A=USR(25):REM Draw circle of diameter 25 pixels
```

In this case, we would not use the value returned since we only need to pass the value one way. Another example, where a value is passed both ways, might be a machine code program which performs a complex calculation (BASIC is very slow at certain types of calculations). In this case you would pass the original value to the machine code program and get the result of the calculation back

```
350 X=USR (211)
360 PRINT "THE ANSWER IS";X
```

You may be wondering how the C64 knows where to look for the machine code program, since we have not given it a start address. The answer is that the start address is stored in Lo–Hi form in decimal locations 785 and 786.

What happens when a USR command is executed is that the parameter (the number in brackets after the USR command) is converted to hex and placed into a special memory location called the Floating Point Accumulator (FPA). To read this number in your machine code program, **JSR $BC9B.** This calls a subroutine which places the Lo-byte of the FPA in location **$65** and the Hi-byte in location **$64**. This is the opposite way around to the usual Lo-Hi form. You can then use this value in your program in the usual way. When you have finished, and want to place a number back into the FPA, simply **JSR $B391.** This routine reads the Lo- and Hi-bytes from **$65** and **$64** respectively and places the resulting value into the FPA. This value is then converted to decimal and stored in the variable preceding the USR command.

10.4 Summary

Machine code subroutines work in an almost identical way to BASIC ones. To create a subroutine, simply label the first line and end with an RTS command. To call the subroutine, just JSR ⟨label⟩. There is a special kind of subroutine called a macro. Macros can have parameters passed to them, and can be stored on tape or disc for later inclusion in other programs. The assembler supplied with this book does not support macros, but you may find it useful to buy one which does once you are a more experienced machine code programmer.

The C64's operating system, called the Kernal, contains hundreds of useful machine code subroutines. You can use any of these in your own programs by JSRing to the appropriate pointer address in the Kernal Jump Table (see Appendix 7). These addresses are the same for any version of the C64 and C128. If you want to pass parameters (values) between BASIC and a machine code program, you use USR instead of SYS to run the program.

10.5 Exercise

Write a program to clear screen and write your name in the top left-hand corner using only the CHROUT (see Kernal routines) routine to output the letters.

11

Interrupts, the stack and adding commands to BASIC

This chapter introduces three important subjects: interrupts, the stack and adding commands to BASIC. All are fairly complex, so this chapter simply forms an introduction to the subject. When you have read it, you will understand:

- the interrupt program
- how to write interrupt-driven software
- what the stack is and how it operates
- how to add extra commands to BASIC

11.1 Interrupts

Whenever you turn on your C64, you are presented with a flashing cursor. This flashing is just one of the many jobs performed by a machine code program called the interrupt program. Fifty times per second, no matter what your C64 happens to be doing at the time, it is interrupted by the interrupt program. This program flashes the cursor, updates the built-in clock so that **TIME$** always contains the correct value, and checks to see whether any keys are being pressed. Having done all this, it then returns to whatever job it was doing before the interrupt. A fiftieth of a second later, the same thing occurs, and again a fiftieth of a second after that. And so on. You can see from this that the C64 is actually pretty busy even when it appears to be doing nothing! It has to make a note of exactly what it is doing, run the interrupt program and then carry on from wherever it left off.

There are times when it would be useful if we could persuade the interrupt program to do a few things for us while it is at it. We might, for example, want to display a real-time clock on the screen. If we tried to update this ourselves, we would have to JSR to our clock update routine at least once a second – our program would be nothing but JSRs and we would not have either room or time to do anything else! Fortunately, we can modify the interrupt program.

The interrupt program is stored at **$EA31**. This address is stored in Lo–Hi form in **$0314** and **$0315**. So, all we have to do is to replace the values in these locations with the address of our own routine and the C64 will jump to there fifty times a second instead. There are, of course, two important points to bear in mind. First, whatever you do within your routine must take considerably less than a fiftieth of a second. And second, the C64 cannot function without the standard interrupt program: for this reason, the last command in your own interrupt routine ***must*** be **JMP $EA31**. This means that the C64 will execute your routine first, and then jump to the normal interrupt program afterwards. Let us write an example interrupt routine (Listing 11.1) to flash the top line of the screen.

Listing 11.1

```
5000 [ $C000
5010 ;
5020 ;** INTERRUPT FLASH TOP LINE **
5030 ;
5040 :SCREEN=$0400
5050 ;
5060 SEI ;STOP INTERRUPTS
5070 LDA #<USERINT ;REPLACE INTERRUPT
5080 STA $0314
5090 LDA #>USERINT
5100 STA $0315
5110 CLI ;RESTART INTERRUPTS
5120 RTS ;RETURN TO BASIC
5130 ;
5140 LDY #$00 :USERINT ;START OF USER INTERRUPT
5150 LDA SCREEN,Y :LOOP
5160 EOR #%10000000
5170 STA SCREEN,Y ;PUT IT BACK-REVERSED
5180 INY
5190 CPY #40
5200 BNE LOOP
5210 JMP $EA31 ;GOTO USUAL INTERRUPT
5220 ;
5230 END
READY.
```

11.2 The stack

The stack is a special area of memory used by the interrupt program. Before the C64 jumps to the interrupt routine, it 'makes a note' of what it is doing so that it can carry on after the interrupt as if nothing had happened. To do this, it stores all the information it needs on the stack. The stack operates on a Last In, First Out (LIFO) basis. This is just like a stack of cards: the last one you put

onto the stack will (obviously) be on top, and will therefore be the first one to be taken when someone takes a card from the stack.

The command you need to place a number onto the stack is PHA, which stands for PusH Accumulator. This takes the value stored in the accumulator and puts a copy of it onto the top of the stack. The complementary command, PLA, PulL Accumulator, removes the value on the top of the stack and stores it in the accumulator. It is important to note that PHA leaves the original value in the accumulator, while PLA alters the accumulator.

The stack can be used by experienced machine code programmers to temporarily store values without using up other memory locations. This is ***not*** recommended to you at this stage as it is extremely easy to lose track of which value is at the top of the stack at any given moment. You will however, probably want to use it once you are more familiar with the stack.

11.3 Adding commands to BASIC

Ok, now for the exciting bit! You may or may not be surprised to know that you now have almost all the information you need to begin adding your own commands to BASIC! When the C64 runs a BASIC program, it checks each line of the program, character-by-character. It then checks to see if it recognizes any of the code as a BASIC keyword. The program which performs this check is called CHRGET. The CHRGET routine is read in from ROM and stored in RAM while the C64 is being used. Because CHRGET is stored in RAM, you can alter it to check for your own, additional, BASIC keywords.

The standard CHRGET routine is shown in Listing 11.2. Don't worry if you do not understand all of it, you can change it to include a check for extra

Listing 11.2

```
NORMAL CHRGET ROUTINE:

    0  0073  E67A      INC  $7A
    1  0075  D002      BNE  $0079
    2  0077  E67B      INC  $7B
    3  0079  AD2A02    LDA  $022A
    4  007C  C93A      CMP  #$3A
    5  007E  B00A      BCS  $008A
    6  0080  C920      CMP  #$20
    7  0082  F0EF      BEQ  $0073
    8  0084  38        SEC
    9  0085  E930      SBC  #$30
   10  0087  38        SEC
   11  0088  E9D0      SBC  #$D0
   12  008A  60        RTS
READY.
```

keywords without knowing how it works. To add your own keywords, simply modify CHRGET to branch to a machine code program of your own once it has reached the end of its own check. Your program performs its own search for your extra keywords; if it finds one, it then jumps to the machine code program (Fig. 11.1) you have written to carry out the command.

Figure 11.1

```
NEW CHRGET ROUTINE:

    0  0073  4C00C0     JMP  $C000
    1  0076  02         BYT  $02
    2  0077  E67B       INC  $7B
    3  0079  AD2702     LDA  $0227
    4  007C  C93A       CMP  #$3A
    5  007E  B00A       BCS  $008A
    6  0080  C920       CMP  #$20
    7  0082  F0EF       BEQ  $0073
    8  0084  38         SEC
    9  0085  E930       SBC  #$30
   10  0087  38         SEC
   11  0088  E9D0       SBC  #$D0
   12  008A  60         RTS
READY.


PROGRAM AT $C000:

    0  C000  E67A       INC  $7A
    1  C002  D002       BNE  $C006
    2  C004  E67B       INC  $7B
    3  C006  A919       LDA  #$19
    4  C008  8D0004     STA  $0400
    5  C00B  4C7900     JMP  $0079
READY.
```

12

Application and practice

You are now well on the way to becoming a fully-fledged machine code programmer! From here on in it's just a case of getting as much practice as possible, and learning from your own experience – just as you did with BASIC. This chapter wraps things up by covering:

- Designing a machine code program
- Choosing between BASIC, machine code and a combination of the two
- Doing several things at once
- Debugging
- Monitors

12.1 Program design

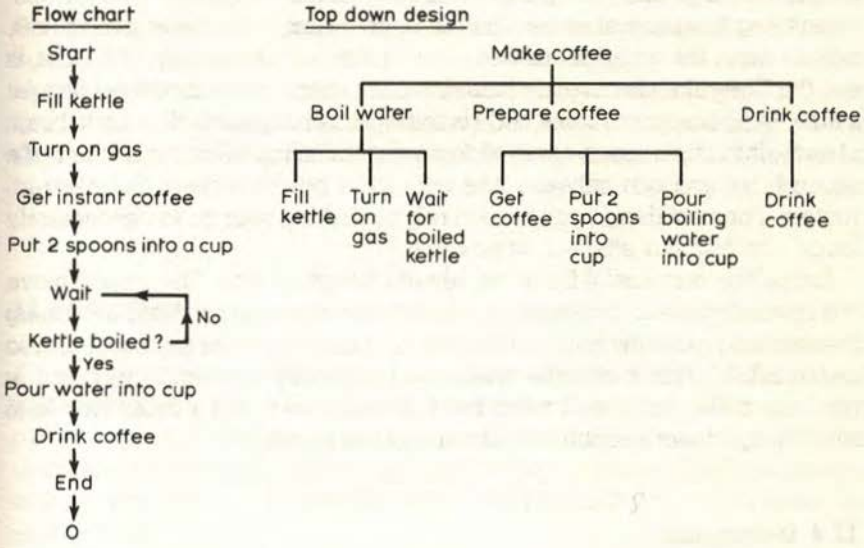
Designing machine code programs is no different in principle from designing BASIC programs. It is, however, more important since you can often get away with beginning a BASIC program with little or no planning; with machine code, however, you will probably end up totally confused if you try to do this – particularly while you are still relatively new to the game.

There are two main approaches to program design. The first is known as flowcharting, and the second as top-down design. You have probably heard of both, but in case you are not clear what the difference is, let us briefly explain the two systems. In flowcharting, you start at the beginning of the program and write down what happens at each stage of the program. You deal with any branches and so on as you meet them, and keep going until you reach the end of the program. In other words, flowcharting is a sequential approach: dealing with each section of the program in the sequence in which it will occur. Top-down design, which is gradually taking over from flowcharting as the most popular approach, involves taking an overall view, and then going into more detailed 'levels' of the program. So, for example, the top level of the program would be an address list program. The second

level might consist of putting original data in, modifying data and searching for addresses. The third level would split each of these tasks into their component parts, the next level divides these into their subtasks and so on. The bottom level is the code itself.

Figure 12.1 illustrates the difference between the two approaches. Personally, we prefer top-down programming, believing it to be easier and clearer, but choose whichever method you prefer.

Figure 12.1



12.2 BASIC, machine code or both?

Once you have planned your program in outline, the first decision to be made is whether to program in BASIC or machine code. BASIC may be slow, but if you can bash out a working program which does the job adequately in 20 minutes, why bother with machine code? Machine code is normally used where BASIC would be too slow.

In many cases, you will find that BASIC is fine for most of the program, it is just one or two places where everything slows down. If this is the case, the solution is to write the main program in BASIC and rewrite the offending routines in machine code. You can then SYS to these as required. A typical example where this approach would be useful is in colouring the screen. All you do is end your machine code program with a final RTS so that it returns to BASIC when it's finished. The BASIC program will then continue.

There are some cases, though, where speed is essential throughout the program. The main example, of course, is arcade-style games, where you want lots of different things to be happening very quickly. It is here where machine code comes into its own, and you would use it to write the entire program.

12.3 Doing several things at once

One of the things many people associate with machine code programs is that everything happens at once! The aliens fly around, the laser gun moves, bombs drop, the timer decreases, music plays. . . . The reality, of course, is that the C64 – like all currently available computers – can only do one thing at a time. But because machine code is so fast, it can ***appear*** to do a lot of things at once. Interrupts are one way of doing several things all within a fiftieth of a second, but you can only execute very short bits of code in the interrupt routine. For most things, you have to rely on making your code – particularly loops – as fast and efficient as possible.

Loops are very useful for doing several things at once. The loop to move the space-invaders, for example, should also check which keys are being pressed and move the gun, make a sound, update the timer and score and so on. In BASIC, this technique would be hopelessly slow and jerky, but in machine code it may well be so fast that you have to put a delay loop in to slow things down enough to make the game playable!

12.4 Debugging

When something goes wrong with a program, the fault is known as a bug. Debugging is the process of correcting the faults: removing the bugs. The old saying about a byte of prevention is worth a megabyte of cure (or something like that) holds especially true for machine code programming. Plan your programs properly, write them logically and type them in carefully and you will keep your debugging to a minimum.

To find the bug in a program, the first thing to do is to take a careful note of the symptoms. Even if the program crashes completely make a note of exactly where in the program it crashed: what was on the screen at the time? That way you will be able to work out what it was doing when it crashed and, therefore, the section of the code that is at fault. If the wrong value was displayed, make a note of the value that should have been displayed and the value actually shown. Check related values: Are they correct? If so, the fault lies in the calculation of the incorrect value. Otherwise the bug may be in the calculation of earlier values. If something appears on the screen in the wrong

place or in the wrong colour, where should it have appeared and what colour should it have been? Is everything else in the right place and in the correct colour? Make a careful note of the symptoms, and debugging is usually straightforward.

One common mistake (you probably made it when typing in one or more of the programs in this book) is to forget to enter a # sign when you want a value. Instead of using the value itself, the C64 would then use the value stored in that memory location. This simple mistake can cause all sorts of unexpected results, so be aware of it when you are typing programs in.

Once you have found the bug, go back to the source code and correct it. Save the corrected program to tape or disk and then re-assemble it and try again. We cannot emphasize this point too strongly, by the way: **always, always save your source code to tape or disc before assembling it.** This does not just apply to the original program, it applies every time you modify it. It is easy to correct one bug and, in doing so, create another one. And the second bug may cause a complete crash. So don't take the chance of losing your work: save it!

12.5 Monitors

A monitor, not the type that sits on your desk with your C64, is a collection of small programs that allow you to poke around in your C64's memory, examining and changing values. They can be extremely useful for debugging your own object code, and are even more useful if you want to modify someone else's machine code program and you do not have the source code.

There are plenty of monitors available for the C64, so it would be unfair of us to recommend any particular one. We do, however, suggest that you insist on the following features:

A disassembler. This is almost the opposite of an assembler. It allows you to convert object code into source code. Well, we say 'almost' because it will actually only display object code in source-code form. It will not actually allow you to alter the source code. You can, however, use it to make sense of object code and perhaps borrow ideas.

A hunt command. Quite often, when you are debugging, you will be looking for a particular value in memory. This might be a number, letter or command. A hunt command allows you to state the value you are looking for and the area of memory to search. It will then tell you whereabouts in memory the value can be found.

A fill command. Useful for testing purposes, a fill command allows you to fill a specified area of memory with a specified value.

A monitor command. All monitors will have this, since the name is derived from this function. A monitor allows you to specify an area of memory, and the contents of the area will be displayed on screen as either hex or decimal values, or, optionally with some monitors, ASCII characters. You can then use the cursor keys to move to an address and modify it. This is the method old-time machine code programmers had to enter their programs before assemblers were introduced.

A simple assembler. A simple assembler is one which works in the usual way except (a) it assembles each line of code straightaway, and (b) it does not allow the use of labels. It is not much use for writing proper programs, but it can be very useful for trying out little ideas and making small changes to your program without going back to your full assembler.

A save command. The normal BASIC SAVE command only allows you to save BASIC programs. If you want to save machine code, you either have to save the source code or you need a command to save the contents of an area of memory. This is what the save command in a monitor does. You tell it the start and end address of your program, and it will save the object code to either tape or disc. This allows you to create object code cassettes and discs to give to other people, or to sell.

To load object code, use

LOAD "⟨filename⟩",8,1:REM Disk

or

LOAD "⟨filename⟩",1,1:REM Tape

A trace (single-step) function. This allows you to execute a machine code program one line at a time. After each line, the monitor displays the values of the accumulator and indices, and the line just executed. You then press a key (usually the SHIFT key) to continue to the next line. This is a very handy aid to debugging.

Break points. This is similar to the trace function. Instead of stopping after every line, you tell it where to stop. This is useful if you know roughly where the bug is.

Afterword

If you've now completed the course: congratulations! If you're flicking idly through the book in your local bookshop: don't just stand there, buy it!

Provided you have worked your way carefully through each chapter, completing the exercises, you now have a firm grounding in 6510 machine code programming. All you need to do now is practise, practise, practise! Just like you did with BASIC.

You can learn a lot from looking at other people's programs, particularly ones which have been heavily annotated. Magazines are a good source of useful routines: check out ***Personal Computer World's*** Subset, for example. Computer clubs are also a great source of ideas and help: there is nearly always someone who has experienced the exact same problem as you, only 6 months ago. While it can be infuriatingly frustrating to have someone provide a solution in 10 seconds to a problem you have been working at for the past 10 days, it is also a tremendous help!

Happy programming, and we look forward to seeing your latest game on the computer store shelves 6 months from now!

APPENDIX 1

Quick Conversion Chart: Decimal/Hex/Binary

DECIMAL	HEX	BINARY
000	$00	%00000000
001	$01	%00000001
002	$02	%00000010
003	$03	%00000011
004	$04	%00000100
005	$05	%00000101
006	$06	%00000110
007	$07	%00000111
008	$08	%00001000
009	$09	%00001001
010	$0A	%00001010
011	$0B	%00001011
012	$0C	%00001100
013	$0D	%00001101
014	$0E	%00001110
015	$0F	%00001111
016	$10	%00010000
017	$11	%00010001
018	$12	%00010010
019	$13	%00010011
020	$14	%00010100
021	$15	%00010101
022	$16	%00010110
023	$17	%00010111
024	$18	%00011000
025	$19	%00011001
026	$1A	%00011010
027	$1B	%00011011
028	$1C	%00011100
029	$1D	%00011101
030	$1E	%00011110
031	$1F	%00011111
032	$20	%00100000
033	$21	%00100001
034	$22	%00100010
035	$23	%00100011
036	$24	%00100100
037	$25	%00100101
038	$26	%00100110

DECIMAL	HEX	BINARY
039	$27	%00100111
040	$28	%00101000
041	$29	%00101001
042	$2A	%00101010
043	$2B	%00101011
044	$2C	%00101100
045	$2D	%00101101
046	$2E	%00101110
047	$2F	%00101111
048	$30	%00110000
049	$31	%00110001
050	$32	%00110010
051	$33	%00110011
052	$34	%00110100
053	$35	%00110101
054	$36	%00110110
055	$37	%00110111
056	$38	%00111000
057	$39	%00111001
058	$3A	%00111010
059	$3B	%00111011
060	$3C	%00111100
061	$3D	%00111101
062	$3E	%00111110
063	$3F	%00111111
064	$40	%01000000
065	$41	%01000001
066	$42	%01000010
067	$43	%01000011
068	$44	%01000100
069	$45	%01000101
070	$46	%01000110
071	$47	%01000111
072	$48	%01001000
073	$49	%01001001
074	$4A	%01001010
075	$4B	%01001011
076	$4C	%01001100
077	$4D	%01001101
078	$4E	%01001110
079	$4F	%01001111
080	$50	%01010000
081	$51	%01010001
082	$52	%01010010
083	$53	%01010011
084	$54	%01010100
085	$55	%01010101
086	$56	%01010110
087	$57	%01010111
088	$58	%01011000
089	$59	%01011001
090	$5A	%01011010

DECIMAL	HEX	BINARY
091	$5B	%01011011
092	$5C	%01011100
093	$5D	%01011101
094	$5E	%01011110
095	$5F	%01011111
096	$60	%01100000
097	$61	%01100001
098	$62	%01100010
099	$63	%01100011
100	$64	%01100100
101	$65	%01100101
102	$66	%01100110
103	$67	%01100111
104	$68	%01101000
105	$69	%01101001
106	$6A	%01101010
107	$6B	%01101011
108	$6C	%01101100
109	$6D	%01101101
110	$6E	%01101110
111	$6F	%01101111
112	$70	%01110000
113	$71	%01110001
114	$72	%01110010
115	$73	%01110011
116	$74	%01110100
117	$75	%01110101
118	$76	%01110110
119	$77	%01110111
120	$78	%01111000
121	$79	%01111001
122	$7A	%01111010
123	$7B	%01111011
124	$7C	%01111100
125	$7D	%01111101
126	$7E	%01111110
127	$7F	%01111111
128	$80	%10000000
129	$81	%10000001
130	$82	%10000010
131	$83	%10000011
132	$84	%10000100
133	$85	%10000101
134	$86	%10000110
135	$87	%10000111
136	$88	%10001000
137	$89	%10001001
138	$8A	%10001010
139	$8B	%10001011
140	$8C	%10001100
141	$8D	%10001101
142	$8E	%10001110
143	$8F	%10001111

DECIMAL	HEX	BINARY
144	$90	%10010000
145	$91	%10010001
146	$92	%10010010
147	$93	%10010011
148	$94	%10010100
149	$95	%10010101
150	$96	%10010110
151	$97	%10010111
152	$98	%10011000
153	$99	%10011001
154	$9A	%10011010
155	$9B	%10011011
156	$9C	%10011100
157	$9D	%10011101
158	$9E	%10011110
159	$9F	%10011111
160	$A0	%10100000
161	$A1	%10100001
162	$A2	%10100010
163	$A3	%10100011
164	$A4	%10100100
165	$A5	%10100101
166	$A6	%10100110
167	$A7	%10100111
168	$A8	%10101000
169	$A9	%10101001
170	$AA	%10101010
171	$AB	%10101011
172	$AC	%10101100
173	$AD	%10101101
174	$AE	%10101110
175	$AF	%10101111
176	$B0	%10110000
177	$B1	%10110001
178	$B2	%10110010
179	$B3	%10110011
180	$B4	%10110100
181	$B5	%10110101
182	$B6	%10110110
183	$B7	%10110111
184	$B8	%10111000
185	$B9	%10111001
186	$BA	%10111010
187	$BB	%10111011
188	$BC	%10111100
189	$BD	%10111101
190	$BE	%10111110
191	$BF	%10111111
192	$C0	%11000000
193	$C1	%11000001
194	$C2	%11000010
195	$C3	%11000011
196	$C4	%11000100

DECIMAL	HEX	BINARY
197	$C5	%11000101
198	$C6	%11000110
199	$C7	%11000111
200	$C8	%11001000
201	$C9	%11001001
202	$CA	%11001010
203	$CB	%11001011
204	$CC	%11001100
205	$CD	%11001101
206	$CE	%11001110
207	$CF	%11001111
208	$D0	%11010000
209	$D1	%11010001
210	$D2	%11010010
211	$D3	%11010011
212	$D4	%11010100
213	$D5	%11010101
214	$D6	%11010110
215	$D7	%11010111
216	$D8	%11011000
217	$D9	%11011001
218	$DA	%11011010
219	$DB	%11011011
220	$DC	%11011100
221	$DD	%11011101
222	$DE	%11011110
223	$DF	%11011111
224	$E0	%11100000
225	$E1	%11100001
226	$E2	%11100010
227	$E3	%11100011
228	$E4	%11100100
229	$E5	%11100101
230	$E6	%11100110
231	$E7	%11100111
232	$E8	%11101000
233	$E9	%11101001
234	$EA	%11101010
235	$EB	%11101011
236	$EC	%11101100
237	$ED	%11101101
238	$EE	%11101110
239	$EF	%11101111
240	$F0	%11110000
241	$F1	%11110001
242	$F2	%11110010
243	$F3	%11110011
244	$F4	%11110100
245	$F5	%11110101
246	$F6	%11110110
247	$F7	%11110111
248	$F8	%11111000
249	$F9	%11111001

DECIMAL	HEX	BINARY
250	$FA	%11111010
251	$FB	%11111011
252	$FC	%11111100
253	$FD	%11111101
254	$FE	%11111110
255	$FF	%11111111

The program used to generate the above table.

```
10 H$="0123456789ABCDEF"
20 OPEN1,4:CMD1
30 PRINT"DECIMAL       HEX          BINARY"
35 PRINT"-------       ---          ------"
40 FORJ=0TO255:PRINT:J$=MID$(STR$(J),2):J$=RIGHT$("000"+J$,3):
   PRINTJ$,
50 GOSUB6000
60 GOSUB5000
70 NEXT
80 PRINT#1:CLOSE1:END
4999 :
5000 D=J
5020 PRINT"%";
5030 FORI=7TO0STEP-1:G=INT(D/2↑I):D=D-G*2↑I:PRINTMID$("01",
     G+1,1);:NEXT:PRINT,
5040 RETURN
5050 :
6000 D=J
6030 PRINT"$";
6040 FORI=1TO0STEP-1:G=INT(D/16↑I):D=D-G*16↑I:PRINTMID$(H$,
     G+1,1);:NEXT:PRINT,
6050 RETURN
READY.
```

APPENDIX 2

C64 memory map

HEX	DECIMAL	Description
00	0	6510 Data Direction
01	1	6510 Input/Output
02	2	Unused
03-04	3-4	Vector: Floating -> Integer
05-06	5-6	Vector: Integer -> Floating
07	7	Search character
08	8	Scan for quote
09	9	Last TAB position
0A	10	Flag: $00 = LOAD, $01 = VERIFY
0B	11	Pointer: Input buffer / No. of Subscripts
0C	12	Default DIM
0D	13	Data Type: $00 = Numeric, $FF = String
0E	14	Data Type: $00 = Floating, $80 = Integer
0F	15	Flag: DATA scan / LIST quote / Garbage call
10	16	Flag: Subscript / User Function (FNX)
11	17	Flag: $00 = INPUT, $40 = GET, $98 = READ
12	18	Flag: ATN sign / Comparison Result
13	19	Flag: Current INPUT prompt
14-15	20-21	Integer value
16	22	Pointer: Temporary string stack
17-18	23-24	Vector: Last temporary string
19-21	25-33	Stack for temporary strings
22-25	34-37	Utility pointer area
26-2A	38-42	Product of multiplication (Floating-point)
2B-2C	43-44	Pointer: Start of Basic program
2D-2E	45-46	Pointer: Start of Basic variables
2F-30	47-48	Pointer: Start of Basic arrays
31-32	49-50	Pointer: End of Basic arrays
33-34	51-52	Pointer: Bottom of string storage
35-36	53-54	Utility string pointer
37-38	55-56	Pointer: Limit of Basic memory
39-3A	57-58	Current Basic line number
3B-3C	59-60	Previous Basic line number
3D-3E	61-62	Pointer: Basic statement for CONT
3F-40	63-64	Current DATA line number
41-42	65-66	Current DATA item address
43-44	67-68	Vector: INPUT
45-46	69-70	Current Basic variable name
47-48	71-72	Pointer: Current Basic variable address
49-4A	73-74	Pointer: Variable in FOR/NEXT
4B-60	75-96	Temp pointer / Data area
61	97	F.P. Accumulator #1: Exponent
62-65	98-101	F.P. Accumulator #1: Mantissa
66	102	F.P. Accumulator #1: Sign

HEX	DECIMAL	Description
67	103	Pointer: Series evaluation constant
68	104	F.P. Accumulator #1: Overflow
69	105	F.P. Accumulator #2: Exponent
6A-6D	106-109	F.P. Accumulator #2: Mantissa
6E	110	F.P. Accumulator #2: Sign
6F	111	F.P. Acc#1 vs F.P. Acc#2 sign comparison
70	112	F.P. Accumulator #1: Rounding
71-72	113-114	Pointer: Cassette buffer
73-8A	115-138	CHRGET subroutine - Get next Basic character
7A-7B	122-123	Pointer: Next character of Basic text
8B-8F	139-143	F.P. RND seed
90	144	STatus
91	145	Flag: RVS Key / STOP Key
92	146	Constant for tape timing
93	147	Flag: $00 = LOAD, $01 = VERIFY
94	148	Flag: Serial output buffered character
95	149	Serial output buffered character
96	150	Tape End Of Tape received
97	151	Register save
98	152	Number of open files
99	153	Input device number
9A	154	Output (CMD) device number
9B	155	Tape character parity
9C	156	Flag: Tape byte received
9D	157	Flag: $00 = Program, $80 = Direct mode
9E	158	Tape pass 1 errors
9F	159	Tape pass 2 errors
A0-A2	160-162	Jiffy clock (TI)
A3	163	Number of serial bits
A4	164	Number of cycles
A5	165	Tape sync countdown
A6	166	Pointer: Tape buffer
A7	167	RS-232 Input bits / tape temp *
A8	168	RS-232 Input bit count / tape temp *
A9	169	RS-232 Check for start bit *
AA	170	RS-232 Input byte buffer / tape temp *
AB	171	RS-232 Parity / tape temp *
AC-AD	172-173	Pointer: Tape buffer / Screen scrolling
AE-AF	174-175	Tape end addresses / End of program
B0-B1	176-177	Tape timing constants
B2-B3	178-179	Pointer: Start of tape buffer
B4	180	RS-232 Bit count / tape temp
B5	181	RS-232 Next bit / tape End of Tape flag
B6	182	RS-232 Byte out buffer
B7	183	Length of Current file name
B8	184	Current logical file number
B9	185	Current secondary address
BA	186	Current device number
BB-BC	187-188	Pointer: Current file name
BD	189	RS-232 Parity out / tape temp
BE	190	Tape write block count
BF	191	Serial word buffer
C0	192	Tape motor interlock
C1-C2	193-194	I/O start address
C3-C4	195-196	Pointer: Tape load temps
C5	197	Current Key pressed $00 = No Key
C6	198	Number of characters in Keyboard queue
C7	199	Flag: $01 = Reverse chars, $00 = Ord chars
C8	200	Pointer: End of line for INPUT

HEX	DECIMAL	Description
C9-CA	201-202	Cursor position (X,Y) for INPUT
CB	203	Flag: Shifted characters
CC	204	Cursor flash, $00 = enable
CD	205	Cursor flash rate
CE	206	Character under cursor
CF	207	Flag: Last cursor flash on/off
D0	208	Flag: INPUT or GET from Keyboard
D1-D2	209-210	Pointer: Current screen line
D3	211	Cursor position (column only)
D4	212	Flag: Quote mode $00 = No
D5	213	Screen line length
D6	214	Cursor position (row only)
D7	215	Temp data area
D8	216	Number of Inserts
D9-F2	217-242	Screen line link table
F3-F4	243-244	Pointer: Colour screen position
F5-F6	245-246	Pointer: Keyboard decode table
F7-F8	247-248	Pointer: RS-232 Input buffer
F9-FA	249-250	Pointer: RS-232 Output buffer
FB-FE	251-254	Zero-page memory left free for users *
FF	255	Basic temp data area
0100-01FF	256-511	Stack area
0200-0258	512-600	Basic INPUT buffer
0259-0262	601-610	Logical file number table
0263-026C	611-620	Device number table
026D-0276	621-630	Secondary address table
0277-0280	631-640	Keyboard queue
0281-0282	641-642	Pointer: Start of Basic memory
0283-0284	643-644	Pointer: Top of Basic memory
0285	645	Flag: IEEE timeout
0286	646	Current colour code
0287	647	Colour under cursor
0288	648	Page containing screen memory
0289	649	Size of Keyboard buffer
028A	650	Key repeat $00=No repeat, $80=All repeat
028B	651	Repeat Key speed
028C	652	Repeat delay
028D	653	Flag: Keyboard shift / CBM Key / CTRL Key
028E	654	Last shift pattern
028F-0290	655-656	Pointer: Set up Keyboard table
0291	657	Flag: $00 = Disable / $80 = Enable shift
0292	658	Flag: $00 = scroll enable
0293	659	RS-232 control register
0294	660	RS-232 command register
0295-0296	661-662	RS-232 Bit timing
0297	663	RS-232 status register
0298	664	RS-232 number of bits to send
0299-029A	665-666	RS-232 speed
029B	667	RS-232 index to end of input buffer
029C	668	RS-232 start of input buffer
029D	669	RS-232 start of output buffer
029E	670	RS-232 index to end of output buffer
029F-02A0	671-672	IRQ save during tape operations
02A1	673	RS-232 enable
02A2	674	Timer for tape operation
02A3	675	Tape temp used during read
02A4	676	Tape temp used during read
02A5	677	Screen row marker
02A6	678	Flag: $00 = NTSC, $01 = PAL T.V. system

HEX	DECIMAL	Description	
02A7-02BF	679-703	Unused	*
02C0-02FE	704-766	Sprite map 11	*
02FF	767	Unused	*
0300-0301	768-769	Pointer: Basic error messages	
0302-0303	770-771	Pointer: Basic warm start	
0304-0305	772-773	Pointer: Tokenise Basic text	
0306-0307	774-775	Pointer: Basic text LIST	
0308-0309	776-777	Pointer: New Basic code link	
030A-030B	778-779	Pointer: Evaluate Basic token	
030C	780	Accumulator save	
030D	781	X index save	
030E	782	Y index save	
030F	783	Status register save	
0310	784	Jump instruction for USR	
0311-0312	785-786	Pointer: For Basic's USR	
0313	787	Unused	
0314-0315	788-789	Pointer: Hardware interrupt	
0316-0317	790-791	Pointer: Break interrupt (BRK)	
0318-0319	792-793	Pointer: Non-maskable interrupt	
031A-031B	794-795	Pointer: Kernal OPEN routine	
031C-031D	796-797	Pointer: Kernal CLOSE routine	
031E-031F	798-799	Pointer: Kernal CHKIN routine	
0320-0321	800-801	Pointer: Kernal CHKOUT routine	
0322-0323	802-803	Pointer: Kernal CLRCHN routine	
0324-0325	804-805	Pointer: Kernal CHRIN routine	
0326-0327	806-807	Pointer: Kernal CHROUT routine	
0328-0329	808-809	Pointer: Kernal STOP routine	
032A-032B	810-811	Pointer: Kernal GETIN routine	
032C-032D	812-813	Pointer: Kernal CLALL routine	
032E-032F	814-815	Pointer: User defined	
0330-0331	816-817	Pointer: Kernal LOAD routine	
0332-0333	818-819	Pointer: Kernal SAVE routine	
0334-033B	820-827	Unused	*
033C-03FB	828-1019	Tape buffer and Sprite maps 13, 14 and 15	*
03FC-03FF	1020-1023	Unused	*
0400-07E7	1024-2023	Screen in normal position	
07E8-07F7	2024-2039	Unused	*
07F8-07FF	2040-2047	Sprite map pointers	
0800-7FFF	2048-32767	Basic program area	
8000-9FFF	32768-40959	Cartridge area / More Basic program area	
A000-BFFF	40960-49151	Basic ROM	
C000-CFFF	49152-53247	Free RAM for user machine code programs	*
D000-D02E	53248-53294	Video registers	
D02F-D3FF	53295-54271	VIC II video chip	
D400-D41C	54272-54300	Sound registers	
D41D-D7FF	54301-55295	SID sound chip	
D800-DBFF	55296-56319	Colour screen	
DC00-DC0F	56320-56335	Interface chip 1	
DD00-DD0F	56576-56591	Interface chip 2	
E000-FFFF	57344-65535	Kernal Operating System ROM	
D000-DFFF	53248-57343	Alternate as character set ROM	
E000-FFFF	57344-65535	Alternate as RAM	

* signifies an area of memory which can be utilised for user machine code programs and variables.

Alternate areas of memory depend upon the contents of location 1.

APPENDIX 3

Screen codes

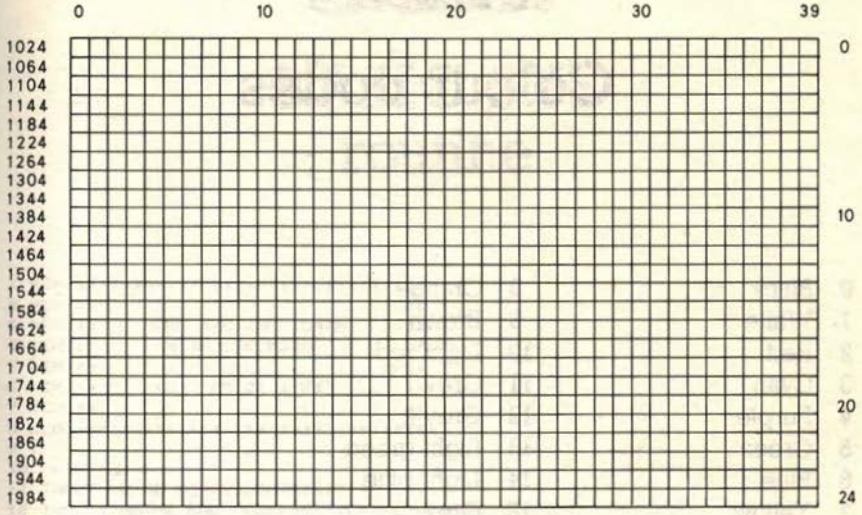
@	0
A	1
B	2
C	3
D	4
E	5
F	6
G	7
H	8
I	9
J	10
K	11
L	12
M	13
N	14
O	15
P	16
Q	17
R	18
S	19
T	20
U	21
V	22
W	23
X	24
Y	25
Z	26
[	27
£	28
]	29
↑	30
←	31
	32
!	33

"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63
—	64

♠	65
	66
—	67
—	68
—	69
—	70
	71
	72
	73
	74
	75
L	76
\	77
/	78
⌜	79
⌝	80
●	81
—	82
♥	83
	84
	85
X	86
o	87
♣	88
	89
◆	90
+	91
▮	92
	93
	94
◥	95
	96
▮	97

▬	98
▔	99
_	100
	101
▩	102
	103
▀	104
◤	105
▌	106
⊢	107
▪	108
⌞	109
⌝	110
▬	111
⌜	112
⊥	113
⊤	114
⊣	115
▌	116
▌	117
▌	118
▔	119
▬	120
▬	121
⌟	122
▪	123
▪	124
⌟	125
▪	126
▚	127

Screen memory map

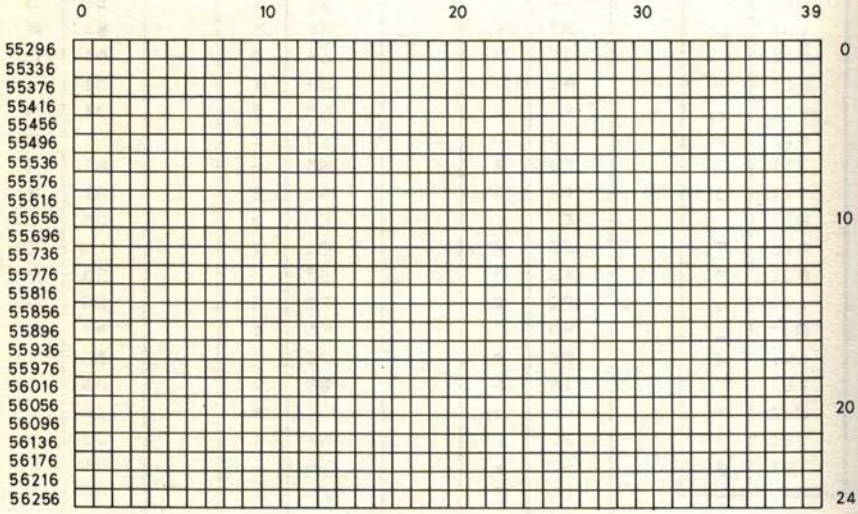


APPENDIX 4

Colour codes

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

Colour memory map



APPENDIX 5

BASIC SYS routine

```
1 REM ********************************
2 REM **                            **
3 REM **     MAKE M/C INTO DATA     **
4 REM **       WITH CHECKSUM        **
5 REM **                            **
6 REM **      (C) ARB 21/1/85       **
7 REM **                            **
8 REM ********************************
9 :
10 PRINT"■"
11 INPUT "■THE START ADDRESS:";S
12 INPUT "■■THE END ADDRESS:";E:IFE=<STHENRUN
13 INPUT "LINE NUMBER START:";LN:IFLN<50THEN13
14 PRINT"■":T=0
15 PRINT"■■■■";LN;"■■L="LN+20;"■■:FORI="S;"■■TO"E;
16 PRINT"■■:L=L+10:T=0:FORJ=0TO5:READA:POKEI,A:T=T+A"
17 LN=LN+10:PRINTLN;"■■I=I+1:";
18 PRINT"NEXTJ:READB:IFT<>BTHEN?"CHR$(34);"ERROR IN LINE:"CHR$(34);";L:STOP"
19 LN=LN+10:PRINTLN;"■■I=I-1:NEXTI:RETURN":REM ****
20 PRINT"LN="LN+10;":S="S;":E="E;":GOTO23"
21 PRINT"■■":POKE631,13:POKE632,13:POKE633,13:POKE634,13:POKE198,4:END
22 :
23 PRINT"■■■■";LN"■■DATA";
24 FORJ=0TO5:A$=STR$(PEEK(J+S)):A$=MID$(A$,2):PRINTA$",";
25 T=T+PEEK(J+S):NEXT:T$=MID$(STR$(T),2):PRINTT$:S=S+6:LN=LN+10
26 PRINT"S="S;":LN="LN;":E="E;":GOTO30":PRINT"■■"
27 :
28 POKE631,13:POKE632,13:POKE198,2:END
29 :
30 IFS<ETHEN23
31 :
32 I=0:RESTORE
33 PRINT"■■■■";:FORJ=0TO7:I=I+1:PRINTI:NEXTJ:REM DELETE THE LINES
34 PRINT"I="I;:IFI<40THENPRINT":GOTO37"
35 PRINT"■■":FORJ=0TO8:POKE631+J,13:NEXT:POKE198,9:END
36 :
37 IFI<40THEN33
38 END
READY.
```

APPENDIX 6

Answers to Exercises

Chapter 3

1 160
2 6
3 252
4 12
5 36873
6 192
7 49216
8 199
9 44352
10 248

Chapter 4

```
5000 [ $C000
5010 ;
5020 ;ANSWER - CHAPTER 4 EXERCISE
5060 ;
5070 LDA #$13 ;S
5080 STA $0400 ;SCREEN
5090 LDA #$15 ;U
5100 STA $0401
5110 LDA #$12 ;R
5120 STA $0402
5130 LDA #$19 ;Y
5140 STA $0403
5150 LDA #$01 ;A
5160 STA $0404
5170 ;
5180 LDA #$01 ;WHITE
5190 STA $D800 ;COLOUR SCREEN
5200 LDA #$03 ;CYAN
5210 STA $D801
5220 LDA #$04 ;PURPLE
5230 STA $D802
5240 LDA #$05 ;GREEN
5250 STA $D803
5260 LDA #$07 ;YELLOW
5270 STA $D804
5280 ;
5290 RTS
5300 END
READY.
```

Chapter 5

```
5000 [ $C000
5010 ;
5020 ;ANSWER - CHAPTER 5 EXERCISE
5030 ;
5040 :BORDER=$D020
5050 :BACKGROUND=$D021
5060 ;
5070 LDA 928
5080 CMP #$02 ;CHECK VALUE
5090 BNE NOTEQUAL ;BRANCH IF NOT = 2
5100 ;
5110 STA BORDER
5120 JMP EXIT ;GO TO EXIT
5130 ;
5140 STA BACKGROUND :NOTEQUAL
5150 ;
5160 RTS :EXIT
5170 END
READY.
```

Chapter 6

```
5000 [ $C000
5010 ;
5020 ;ANSWER - CHAPTER 6 EXERCISE
5030 ;
5040 :SCREEN=$0400
5050 :YELLOW=7
5060 :COLSCREEN=$D900
5070 ;
5080 LDY #$00 ;INITIALISE Y
5090 TYA :LOOP ;GET VALUE FOR ACC.
5100 STA SCREEN,Y ;PLACE IT ON SCREEN
5110 LDA #YELLOW
5120 STA COLSCREEN,Y ;PLACE YELLOW ON COLOUR SCREEN
5130 INY
5140 CPY #$00
5150 BNE LOOP
5160 ;
5170 RTS
5180 END
READY.
```

Chapter 7

```
5000 [ $C000
5010 ;
5020 ;ANSWER - CHAPTER 7 EXERCISE
5030 ;
5040 :VIDEO=$D000 ;VIDEO CHIP
5045 :MSB=VIDEO+$10 ;SPRITE MSB
```

```
5050 ;
5060 LDY #$00
5070 LDA #100
5080 STY VIDEO ;SPRITE 0 - X
5090 STA VIDEO+1 ;SPRITE 0 - Y
5100 LDA #01
5110 STA VIDEO+21 ;SPRITE 0 = ON
5120 LDA MSB
5130 AND #%11111110 ;TURN OFF MSB FOR SPRITE 0
5140 STA MSB
5150 ;
5160 INC VIDEO :LOOP1 ;MOVE SPRITE 0
5170 LDX #00
5180 DEX :DELAY1 ;DELAY
5190 BNE DELAY1
5200 INY
5210 BNE LOOP1
5220 ;
5230 ;Y IS NOW ZERO
5240 ;
5250 STY VIDEO ;SPRITE 0, X=0
5260 LDA MSB
5270 ORA #%00000001 ;MOVE SPRITE 0 ONTO RHS OF SCREEN
5280 STA MSB
5290 ;
5300 INC VIDEO :LOOP2
5310 LDX #00
5320 DEX :DELAY2
5330 BNE DELAY2
5340 INY
5350 BNE LOOP2
5360 ;
5370 RTS
5380 END
READY.
```

Chapter 8

```
5000 [ $C000
5010 ;
5020 ;ANSWER - CHAPTER 8 EXERCISE
5030 ;
5040 LDX #$00 ;INITIALISE X
5050 CLC ;CLEAR CARRY
5060 ;
5070 LDA 828 :LOOP
5080 LSR A
5090 INX
5100 CPX #$08
5110 BNE LOOP ;CARRY NOT CLEARED WITHIN LOOP
5120 ;
5130 RTS
5140 END
READY.
```

Chapter 9

```
5000 [ *C000
5010 ;
5020 ;ANSWER - CHAPTER 9 EXERCISE
5030 ;
5040 LDA 828
5050 TAY ;STORE ACC IN Y
5060 ;MULTIPLY ACC BY 2
5070 ASL A
5080 ;
5090 CLC ;CLEAR CARRY
5100 STA 828 ;828 NOW CONTAINS PEEK(828)*2
5110 TYA
5120 ADC 828
5130 STA 828 ;ADD ORIGINAL CONTENTS
5140 ;
5150 ;828 NOW CONTAINS PEEK(828)*3
5160 ;
5170 RTS
5180 END
READY.
```

Chapter 10

```
5000 [ *C000
5010 ;
5020 ;ANSWER - CHAPTER 10 EXERCISE
5030 ;
5040 ;CHROUT=$FFD2
5050 ;
5060 LDA #147
5070 JSR CHROUT ;CLEAR SCREEN
5080 ;
5090 LDA #'A
5100 JSR CHROUT ;DO EACH LETTER IN TURN
5110 LDA #'N
5120 JSR CHROUT
5130 LDA #'D
5140 JSR CHROUT
5150 LDA #'R
5160 JSR CHROUT
5170 LDA #'E
5180 JSR CHROUT
5190 LDA #'W
5200 JSR CHROUT
5210 ;
5220 RTS
5230 END
READY.
```

APPENDIX 7

The Kernal routine

Name: CHRIN

Operation: Get a stream of characters from the Keyboard.

Call Address: $FFCF, 65487

Registers Affected: Accumulator, X

Description: This routine takes input from the Keyboard and is the one used by the normal input on the C64. The routine flashes the cursor and awaits input. When a carriage return in inputed the routine returns. The routine is then called for each inputed character. For example:

```
5000 [ $C000
5010 ;
5012 ;CHRIN EXAMPLE
5014 ;
5020 :CHRIN=$FFCF
5030 :BUFFER=828 ;CASSETTE BUFFER
5040 ;
5050 LDY #$00 ;INITIALISE Y INDEX
5060 JSR CHRIN :CHRINTEST1
5070 STA BUFFER,Y ;STORE EACH CHARACTER
5080 INY
5090 CMP #13 ;RETURN PRESSED
5100 BNE CHRINTEST1 ;NO - GO BACK FOR ANOTHER CHARACTER
5105 ;
5110 RTS
READY.
```

Name: CHROUT

Operation: Output a character to the screen.

Call Address: $FFD2, 65490

Registers Affected: Accumulator

Description: This routine is one the most useful Kernal routines. It outputs any ASCII character held in the accumulator to the screen. You can use it to change colour, clear the screen, home the cursor or print any character that you wish.

```
5000 [ $C000
5010 ;
5015 ;CHROUT EXAMPLE
5020 ;
5030 :CHROUT=$FFD2
5040 ;
5050 LDY #$00
5060 LDA TEXT,Y :CHROUTTEST1 ;GET NEXT CHARACTER
5070 CMP #'@ ;IS IT AN '@'
5080 BEQ CHROUTTEST2 ;YES - EXIT
5090 JSR CHROUT ;NO - PRINT IT
5100 INY
5110 JMP CHROUTTEST1 ;GO BACK FOR NEXT CHARACTER
5120 ;
5130 RTS :CHROUTTEST2
5140 .TXT "ANDREW IS ACE@" :TEXT
READY.
```

Name: GETIN

Operation: Get a character from the Keyboard.

Call Address: $FFE4, 65508

Registers Affected: Accumulator, X, Y.

Description: This routine gets a single character from the Keyboard and returns it in the accumulator. The cursor is NOT flashed. If no Key has been pressed then zero is returned in the accumulator.

```
5000 [ $C000
5010 ;
5015 ;GETIN EXAMPLE
5020 ;
5030 :GETIN=$FFE4
5040 :TEMP=828
5050 ;
5060 LDY #$00
5070 STY TEMP :GETINTEST1 ;STORE Y TO PROTECT IT
5080 JSR GETIN ;GET A CHARACTER FROM KEYBOARD
5090 CMP #13 ;IS IT A RETURNPRINT
5100 BEQ GETINTEST2 ;YES - EXIT
5110 CMP #$00 ;NO KEY PRESSEDPRINT
5120 BEQ GETINTEST1 ;YES - GO BACK FOR ANOTHER CHARACTER
5130 LDY TEMP ;GET Y FROM STORE
5140 STA $0400,Y ;PLACE CHARACTER ON SCREEN
5150 INY
5160 JMP GETINTEST1 ;GO BACK FOR MORE
5165 ;
5170 RTS :GETINTEST2
5200 END
READY.
```

Name: LOAD

Operation: Load memory from cassette or disk.

Call Address: $FFD5, 65493

Registers Affected: Accumulator, X, Y.

Description: This routine will load an area of memory of m/c program from disk or tape into the C64. Before you can use it you must JSR to the SETLFS and SETNAM routines. The accumulator must be set to zero for load.

[* LOAD example *]

```
5000 [ $C000
5010 ;
5015 ;LOAD EXAMPLE
5020 ;
5030 :LOAD=$FFD5
5040 :SETLFS=$FFBA
5050 :SETNAM=$FFBD
5060 ;
5070 LDA #$01 ;FILE NUMBER = 1
```

```
5080 LDX #$01 ;TAPE DEVICE = 1
5090 LDY #$01 ;NOT A RELOCATED LOAD
5100 JSR SETLFS
5110 ;
5120 LDA #$00
5130 JSR SETNAM ;NO FILE NAME
5140 ;
5150 LDA #$00 ;LOAD NOT VERIFY
5160 JSR LOAD
5170 ;
5180 RTS
5190 ;
5200 END
READY.
```

Name: PLOT

Operation: Set or Read cursor position.

Call Address: $FFF0, 65520

Registers Affected: Accumulator, X, Y.

Description: This routine moves the position of the cursor to anywhere on the screen. If used with CHROUT, you can print characters anywhere on the screen. The carry flag must be clear and the x and y positions for the cursor must be held in the X and Y indexes. If the carry is set then the position of the cursor is returned in X and Y.

[* PLOT example *]

```
5000 [ $C000
5005 ;
5010 ;PLOT EXAMPLE
5015 ;
5020 :PLOT=$FFF0
5030 :CHROUT=$FFD2
5040 ;
5050 CLC
5060 LDY #18 ;COLUMN NUMBER
5070 LDX #$00 ;ROW NUMBER
5080 JSR PLOT
5090 JSR WRITEPLOT
5100 ;
5110 CLC
5120 LDY #18 ;COLUMN NUMBER
```

```
5160 LDX #$10 ;ROW NUMBER
5170 JSR PLOT
5180 JSR WRITEPLOT
5190 ;
5200 RTS
5210 ;
5220 LDY #$00 :WRITEPLOT
5230 LDA TEXT,Y :WRITEPLOT1 ;GET NEXT CHARACTER OF TEXT
5240 CMP #'@ ;IS IT '@'PRINT
5250 BEQ WRITEPLOT2 ;YES - EXIT
5260 JSR CHROUT ;PRINT THE CHARACTER
5270 INY
5280 JMP WRITEPLOT1
5290 RTS :WRITEPLOT2
5300 ;
5310 .TXT "PLOT@" :TEXT
5320 END
READY.
```

Name: SAVE

Operation: Save memory to cassette or disk.

Call Address: $FFD8, 65496

Registers Affected: Accumulator, X, Y.

Description: This routine will save any area of memory or m/c program to disk or tape. Before you use it you must call the SETLFS and SETNAM routines. You must place the start address in lo-hi format in page zero and the end address in lo-hi in the X and Y indexes. The accumulator must then be loaded with the page-zero offset of the start address pointer. So that if you use $F7 and $F8 as the pointer, you will load the accumulator with $F7.

[* SAVE example *]

```
5000 [ $C000
5010 ;
5015 ;SAVE EXAMPLE - SAVE $7000 TO $7100
5020 ;
5030 :SAVE=$FFD8
5040 :SETLFS=$FFBA
5050 :SETNAM=$FFBD
```

```
5070 LDY #$FF ;NO SECONDARY ADDRESS
5080 LDA #$01 ;FILE NUMBER = 1
5090 LDX #$01 ;TAPE DEVICE = 1
5100 JSR SETLFS
5110 ;
5120 LDA #$00 ;NO FILE NAME
5130 JSR SETNAM
5140 ;
5150 LDA #$00 ;LO PART OF START ADDRESS
5160 STA $F7
5170 LDA #$70 ;HI PART OF START ADDRESS
5180 STA $F8
5190 LDX #$00 ;LO PART OF END ADDRESS
5200 LDY #$71 ;HI PART OF END ADDRESS ($7100)
5210 LDA #$F7 ;OFFSET FOR START ADDRESS
5220 JSR SAVE
5230 ;
5240 RTS
5250 ;
5260 END
READY.
```

Name: SETLFS

Operation: Set up a file.

Call Address: $FFBA, 65466

Registers Affected: None.

Description: This routine sets up a file for the LOAD and SAVE routines. You must load the accumulator with the file number, the X index with the device number and the Y index with the secondary address. For cassette and disk operation, the Y index must be set to $FF (255). See the LOAD and SAVE examples for examples of SETLFS.

Name: SETNAM

Operation: Set up a file name.

Call Address: $FFBD, 65469

Registers Affected: None.

Description: This routine sets up a file name for the LOAD and SAVE routines. The accumulator is loaded with the length of the name and the X and Y indexes are loaded with the lo and hi parts of the address of the start of the name. For examples of SETNAM's useage see the LOAD and SAVE examples.

APPENDIX 8

A complete listing of the 6510 assembly language instruction set

Name: ADC
Operation: Add memory to Accumulator with Carry

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	ADC #$QQ	69	2
Zero Page	ADC $QQ	65	2
Zero Page,X	ADC $QQ,X	75	2
Absolute	ADC $QQQQ	6D	3
Absolute,X	ADC $QQQQ,X	7D	3
Absolute,Y	ADC $QQQQ,Y	79	3
(Indirect,X)	ADC ($QQ,X)	61	2
(Indirect),Y	ADC ($QQ),Y	71	2

Flags affected:	N	Z	C	I	D	V
	*	*	*			*

Name: AND
Operation: AND memory with Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	AND #$QQ	29	2
Zero Page	AND $QQ	25	2
Zero Page,X	AND $QQ,X	35	2
Absolute	AND $QQQQ	2D	3
Absolute,X	AND $QQQQ,X	3D	3
Absolute,Y	AND $QQQQ,Y	39	3
(Indirect,X)	AND ($QQ,X)	21	2
(Indirect),Y	AND ($QQ),Y	31	2

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: ASL
Operation: Shift left one bit (Accumulator or Memory)

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Accumulator	ASL #$QQ	0A	1
Zero Page	ASL $QQ	06	2
Zero Page,X	ASL $QQ,X	16	2
Absolute	ASL $QQQQ	0E	3
Absolute,X	ASL $QQQQ,X	1E	3

Flags affected:	N	Z	C	I	D	V
	*	*	*			

Name: BCC
Operation: Branch on Carry Clear

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BCC $QQ	90	2

Flags affected:	N	Z	C	I	D	V

Name: BCS
Operation: Branch on Carry Set

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BCS $QQ	B0	2

Flags affected:	N	Z	C	I	D	V

Name: BEQ
Operation: Branch on Zero

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BEQ $QQ	F0	2

Flags affected:	N	Z	C	I	D	V

Name: BIT
Operation: Test bits in memory with Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Zero Page	BIT $QQ	24	2
Absolute	BIT $QQQQ	2C	3

Flags affected:	N	Z	C	I	D	V
	*	*				*

Name: BMI
Operation: Branch on Minus

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BMI $QQ	30	2

Flags affected:

N	Z	C	I	D	V

Name: BNE
Operation: Branch on not Zero

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BNE $QQ	D0	2

Flags affected:

N	Z	C	I	D	V

Name: BPL
Operation: Branch on Plus

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BPL $QQ	10	2

Flags affected:

N	Z	C	I	D	V

Name: BRK
Operation: BreaK

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	BRK	00	1

Flags affected:

N	Z	C	I	D	V
			*		

Name: BVC
Operation: Branch on Overflow Clear

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BVC $QQ	50	2

Flags affected:

N	Z	C	I	D	V

Name: BVS
Operation: Branch on Overflow Set

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Relative	BVS $QQ	70	2

Flags affected:

N	Z	C	I	D	V

Name: CLC
Operation: Clear Carry flag

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	CLC	18	1

Flags affected:

N	Z	C	I	D	V
		*			

Name: CLD
Operation: Clear Decimal Mode

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	CLD	D8	1

Flags affected:

N	Z	C	I	D	V
				*	

Name: CLI
Operation: Clear Interrupt Disable flag

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	CLI	58	1

Flags affected:

N	Z	C	I	D	V
			*		

Name: CLV
Operation: Clear Overflow Flag

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	CLV	B8	1

Flags affected:	N	Z	C	I	D	V
						*

Name: CMP
Operation: Compare memory with Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	CMP #$QQ	C9	2
Zero Page	CMP $QQ	C5	2
Zero Page,X	CMP $QQ,X	D5	2
Absolute	CMP $QQQQ	CD	3
Absolute,X	CMP $QQQQ,X	DD	3
Absolute,Y	CMP $QQQQ,Y	D9	3
(Indirect,X)	CMP ($QQ,X)	C1	2
(Indirect),Y	CMP ($QQ),Y	D1	2

Flags affected:	N	Z	C	I	D	V
	*	*	*			

Name: CPX
Operation: Compare Memory with X index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	CPX #$QQ	E0	2
Zero page	CPX $QQ	E4	2
Absolute	CPX $QQQQ	EC	3

Flags affected:	N	Z	C	I	D	V
	*	*	*			

Name: CPY
Operation: Compare Memory with Y index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	CPX #$QQ	C0	2
Zero page	CPX $QQ	C4	2
Absolute	CPX $QQQQ	CC	3

Flags affected:	N	Z	C	I	D	V
	*	*	*			

Name: DEC
Operation: Decrease Memory by One

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Zero page	DEC $QQ	C6	2
Zero page,X	DEC $QQ,X	D6	2
Absolute	DEC $QQQQ	CE	3
Absolute,X	DEC $QQQQ,X	DE	3

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: DEX
Operation: Decrease X index by One

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	DEX	CA	1

Flags affected:

N	Z	C	I	D	V
*	*				

Name: DEY
Operation: Decrease Y index by One

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	DEY	88	1

Flags affected:

N	Z	C	I	D	V
*	*				

Name: EOR
Operation: Exclusive-OR Memory With Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	EOR #$QQ	49	2
Zero Page	EOR $QQ	45	2
Zero Page,X	EOR $QQ,X	55	2
Absolute	EOR $QQQQ	4D	3
Absolute,X	EOR $QQQQ,X	5D	3
Absolute,Y	EOR $QQQQ,Y	59	3
(Indirect,X)	EOR ($QQ,X)	41	2
(Indirect),Y	EOR ($QQ),Y	41	2

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: INC
Operation: Increase Memory by One

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Zero page	INC $QQ	E6	2
Zero page,X	INC $QQ,X	F6	2
Absolute	INC $QQQQ	EE	3
Absolute,X	INC $QQQQ,X	FE	3

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: INX
Operation: Increase X index by One

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	INX	E8	1

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: INY
Operation: Increase Y index by One

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	INY	C8	1

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: JMP
Operation: Jump

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Absolute	JMP $QQQQ	4C	3
Indirect	JMP ($QQQQ)	6C	3

Flags affected:	N	Z	C	I	D	V

Name: JSR
Operation: Jump To Subroutine

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Absolute	JSR $QQQQ	20	3

Flags affected:	N	Z	C	I	D	V

Name: LDA
Operation: Load Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	LDA #$QQ	A9	2
Zero Page	LDA $QQ	A5	2
Zero Page,X	LDA $QQ,X	B5	2
Absolute	LDA $QQQQ	AD	3
Absolute,X	LDA $QQQQ,X	BD	3
Absolute,Y	LDA $QQQQ,Y	B9	3
(Indirect,X)	LDA ($QQ,X)	A1	2
(Indirect),Y	LDA ($QQ),Y	B1	2

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: LDX
Operation: Load X Index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	LDX #$QQ	A2	2
Zero Page	LDX $QQ	A6	2
Zero Page,Y	LDX $QQ,Y	B6	2
Absolute	LDX $QQQQ	AE	3
Absolute,Y	LDX $QQQQ,Y	BE	3

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: LDY
Operation: Load Y Index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	LDY #$QQ	A0	2
Zero Page	LDY $QQ	A4	2
Zero Page,X	LDY $QQ,X	B4	2
Absolute	LDY $QQQQ	AC	3
Absolute,X	LDY $QQQQ,X	BC	3

Flags affected:

N	Z	C	I	D	V
*	*				

Name: LSR
Operation: Shift Accumulator Or Memory One bit Right

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Accumulator	LSR A	4A	1
Zero Page	LSR $QQ	46	2
Zero Page,X	LSR $QQ,X	56	2
Absolute	LSR $QQQQ	4E	3
Absolute,X	LSR $QQQQ,X	5E	3

Flags affected:

N	Z	C	I	D	V
*	*	*			

Name: NOP
Operation: No Operation

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	NOP	EA	1

Flags affected:	N	Z	C	I	D	V

Name: ORA
Operation: OR Memory With Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	ORA #$QQ	09	2
Zero Page	ORA $QQ	05	2
Zero Page,X	ORA $QQ,X	15	2
Absolute	ORA $QQQQ	0D	3
Absolute,X	ORA $QQQQ,X	1D	3
Absolute,Y	ORA $QQQQ,Y	19	3
(Indirect,X)	ORA ($QQ,X)	01	2
(Indirect),Y	ORA ($QQ),Y	11	2

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: PHA
Operation: Push Accumulator Onto Stack

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	PHA	48	1

Flags affected:

N	Z	C	I	D	V

Name: PHP
Operation: Push Status Onto Stack

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	PHP	08	1

Flags affected:

N	Z	C	I	D	V

Name: PLA
Operation: Pull Accumulator Off Stack

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	PLA	68	1

Flags affected:

N	Z	C	I	D	V
*	*				

Name: PLP
Operation: Pull Status Off Stack

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	PLP	28	1

Flags affected:	N	Z	C	I	D	V
	---- FROM STACK ----					

Name: ROL
Operation: Rotate Accumulator Or Memory One bit Left

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Accumulator	ROL A	2A	1
Zero Page	ROL $QQ	26	2
Zero Page,X	ROL $QQ,X	36	2
Absolute	ROL $QQQQ	2E	3
Absolute,X	ROL $QQQQ,X	3E	3

Flags affected:	N	Z	C	I	D	V
	*	*	*			

Name: ROR
Operation: Rotate Accumulator Or Memory One bit Right

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Accumulator	ROR A	6A	1
Zero Page	ROR $QQ	66	2
Zero Page,X	ROR $QQ,X	76	2
Absolute	ROR $QQQQ	6E	3
Absolute,X	ROR $QQQQ,X	7E	3

Flags affected:	N	Z	C	I	D	V
	*	*	*			

Name: RTI
Operation: Return From Interrupt

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	RTI	40	1

Flags affected:	N	Z	C	I	D	V
	---- FROM STACK ----					

Name: RTS
Operation: Return From Subroutine

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	RTS	60	1

Flags affected:	N	Z	C	I	D	V

Name: SBC
Operation: Subtract Memory From Accumulator With Borrow

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Immediate	SBC #$QQ	E9	2
Zero Page	SBC $QQ	E5	2
Zero Page,X	SBC $QQ,X	F5	2
Absolute	SBC $QQQQ	ED	3
Absolute,X	SBC $QQQQ,X	FD	3
Absolute,Y	SBC $QQQQ,Y	F9	3
(Indirect,X)	SBC ($QQ,X)	E1	2
(Indirect),Y	SBC ($QQ),Y	F1	2

Flags affected:	N	Z	C	I	D	V
	*	*	*			*

Name: SEC
Operation: Set Carry Flag

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	SEC	38	1

Flags affected:	N	Z	C	I	D	V
			*			

Name: SED
Operation: Set Decimal Flag

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	SED	F8	1

Flags affected:	N	Z	C	I	D	V
					*	

Name: SEI
Operation: Set Interrupt Disable Flag

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	SEI	78	1

Flags affected:	N	Z	C	I	D	V
				*		

Name: STA
Operation: Store Accumulator In Memory

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Zero Page	STA $QQ	85	2
Zero Page,X	STA $QQ,X	95	2
Absolute	STA $QQQQ	8D	3
Absolute,X	STA $QQQQ,X	9D	3
Absolute,Y	STA $QQQQ,Y	99	3
(Indirect,X)	STA ($QQ,X)	81	2
(Indirect),Y	STA ($QQ),Y	91	2

Flags affected:	N	Z	C	I	D	V

Name: STX
Operation: Store X Index In Memory

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Zero Page	STX $QQ	86	2
Zero Page,Y	STX $QQ,Y	96	2
Absolute	STX $QQQQ	8E	3

Flags affected:	N	Z	C	I	D	V

Name: STY
Operation: Store Y Index In Memory

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Zero Page	STY $QQ	84	2
Zero Page,X	STY $QQ,X	94	2
Absolute	STY $QQQQ	8C	3

Flags affected:	N	Z	C	I	D	V

Name: TAX
Operation: Transfer Accumulator to X Index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	TAX	AA	1

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: TAY
Operation: Transfer Accumulator to Y Index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	TAY	A8	1

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: TSX
Operation: Transfer Stack Pointer To X Index

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	TSX	BA	1

Flags affected:	N	Z	C	I	D	V

Name: TXA
Operation: Transfer X Index To Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	TXA	8A	1

Flags affected:	N	Z	C	I	D	V
	*	*				

Name: TXS
Operation: Transfer X Index To Stack Pointer

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	TXS	9A	1

Flags affected:	N	Z	C	I	D	V

Name: TYA
Operation: Transfer Y Index To Accumulator

Addressing Mode	Assembly Language Form	Opcode	Number of Bytes
Implied	TYA	98	1

Flags affected:

N	Z	C	I	D	V
*	*				

APPENDIX 9

ASCII codes

	0
	1
	2
	3
	4
	5
	6
	7
	8
	9
	10
	11
	12
	13
	14
	15
	16
	17
	18
	19
	20
	21
	22
	23
	24
	25
	26
	27
	28
	29
	30
	31
	32

!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63
@	64

A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90

[	91
£	92
]	93
↑	94
←	95
—	96
♠	97
▏	98
▔	99
▔	100
▔	101
▔	102
▏	103
▏	104
▏	105
▏	106
▁	107
▁	108
╱	109
╱	110
▁	111
▁	112
●	113
—	114
♥	115
▏	116
▁	117
╳	118
o	119
♣	120
▏	121
♦	122
+	123
▒	124
▏	125
▗	126
◥	127
	128
	129
	130
	131
	132

	133
	134
	135
	136
	137
	138
	139
	140
	141
	142
	143
	144
	145
	146
	147
	148
	149
	150
	151
	152
	153
	154
	155
	156
	157
	158
	159
	160
▮	161
▬	162
▔	163
_	164
▏	165
▉	166
▕	167
▀	168
◤	169
▌	170
▙	171
▪	172
▖	173
▗	174

▁	175
▝	176
▟	177
▜	178
▙	179
▌	180
▌	181
▌	182
▔	183
▬	184
▬	185
▟	186
▪	187
▪	188
▗	189
▪	190
▚	191
▁	192
◆	193
▏	194
▔	195
▔	196
▔	197
▁	198
▏	199
▏	200
▝	201
▝	202
▝	203
▙	204
╱	205
╱	206
▛	207
▜	208
●	209
▁	210
♥	211
▏	212
▝	213
X	214

o	215
♠	216
▏	217
◆	218
┼	219
▌	220
▏	221
◢	222
◥	223
	224
▌	225
▬	226
▔	227
_	228
▏	229
▉	230
▏	231
▀	232
◤	233
▌	234
▙	235
▪	236
▖	237
▗	238
▁	239
▝	240
▟	241
▜	242
▙	243
▌	244
▌	245
▌	246
▔	247
▔	248
▁	249
▟	250
▪	251
▪	252
▗	253
	254
◢	255

Index

Page numbers in **bold** refer to main entries.

The Complete Commodore Machine Code Programming Course

Andrew Bennett and Surya

Here, at last, is a really easy-to-follow introduction to machine code programming on the Commodore 64 and 128 computers.

Written with the absolute beginner in mind, the course starts with the question 'What is machine code?', explains the background concepts necessary for a complete understanding of the subject, then guides the reader gently through first the simple and later more complex programming steps.

The aim of the book is clear: to turn the reader into a competent machine code programmer as quickly and painlessly as possible. Every new command or concept is illustrated with a fully-annotated sample program and a thorough explanation.

Best of all, the course includes a free machine code assembler. Not only does this save you money, but you are guaranteed 100% compatibility between what you read in the book and what happens when you use the assembler.

By the end of the course, you will be able to write programs that not only contain features impossible to write in BASIC, but which run *hundreds* of times faster than a BASIC program!

A Chapman and Hall/
Methuen Paperback
COMPUTING

11 New Fetter Lane
London EC4P 4EE
29 West 35th Street
New York NY 10001

ISBN 0-412-27250-4