ADVANCED BASIC AND MACHINE CODE FOR THE COMMODORE 64

PETER GERRARD

Advanced Basic & Machine Code for the 64

Peter Gerrard

Duckworth

First published in 1984 by
Gerald Duckworth & Co. Ltd.
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 1785 0

British Library Cataloguing in Publication Data

Gerrard, Peter
Advanced BASIC and machine code for the 64
—(Duckworth home computing)
1. Computer 64 (Computer) — Programming
2. Basic (Computer Program language)
I. Title
001.64'24 QA76.8.C64

ISBN 0-7156-1785-0

Typeset by The Electronic Village, Richmond
from text stored on a Commodore 64
Printed in Great Britain by
Redwood Burn Ltd., Trowbridge
and bound by Pegasus Bookbinding, Melksham

Contents

Preface

In order to get the most out of your Commodore 64, it is necessary to move beyond BASIC and start doing some programming in machine code. All the best features of the 64 are inaccessible from BASIC other than by using POKEs and PEEKs all the time, which is unnecessarily slow.

This book aims to teach you the rudiments of machine code programming, with a couple of chapters on BASIC just to limber up. Commands are introduced slowly, with plenty of explanation and sample listings to work from. A feature used throughout the book is a small arcade game, which is gradually built up from chapter to chapter, using commands introduced in each chapter.

Subjects covered include interrupts, which provide us with the opportunity to produce some background music, adding commands to BASIC, manipulating sprites in machine code, how to perform 8 bit and 16 bit multiplication, and more. If you want to learn to program in machine code, this is where to start.

P.G.

I'd like to dedicate this book to John Ryan, whose ceaseless efforts to see it in print deserve a suitable reward. Without his enthusiasm, it would have taken much longer to produce. Thanks John.

1
Better BASIC Programming

Introduction

As anyone who has tried to use their Commodore 64 at all seriously will know, the version of BASIC that it is equipped with is not the most advanced version around. Indeed, by today's standards it is rather antiquated, as those of you who have had the opportunity to use, say, a BBC computer, will readily appreciate.

When it comes to internal specifications, however, the 64 is probably one of the better machines in the £100-200 price bracket. Given a structured BASIC there is no doubt that the 64 would be selling in even greater quantities than it already does. However, what you're given is a language that has no procedures, no easy-to-use commands for handling disk drives, and two of the most awkward commands for looking after graphics and sound: PEEK and POKE.

It is true that there are several packages on the market that enable you to upgrade the inherent BASIC, items such as BC Basic, Power and Morepower, Simon's Basic, and so on. But they all cost money, and if your budget won't stretch to purchasing anything other than the machine and a peripheral or two, then you'll have to learn to live with the Commodore environment and just make the best of a bad job.

Machine Code

Later on in this book we'll be looking at machine code programming, starting with a very general introduction to the language, and hopefully blowing away a few of the myths that surround it. People tend to speak of machine code programming as if it's on a par with learning to translate dwarfish runes, and only scholars hunched over ancient, musty manuscripts (or 12-year-old schoolkids hunched over copies of Revenge of the Mutant Camels) can ever hope to speak this strange tongue.

If this wasn't being written for such polite publishers as Duckworth, my response to that would be a short, but meaningful, word. There is nothing at all complicated about learning to program in machine code. You had to learn BASIC, which admittedly is more akin to the English language than some of the instructions we'll be encountering later, and machine code is simply another collection of symbols instructing the computer to do something.

By the end of this book you won't be writing anything as stunning as Easyscript or International Soccer, but at least the rudiments will be there, and you will be capable of writing acceptable programs that work much faster than the equivalent program in BASIC. Anyone can program in machine code: this book will teach you how.

Back to Basics

But for the first couple of chapters we're going to be concentrating purely on BASIC, and trying to make the best of the language that the machine presents us with at power-on. 38,911 bytes to play with is enough for most people, and some very good programs can be written within the constraints of Commodore BASIC, as we shall see.

There are some general rules to be observed when programming in BASIC, which will help to make programs run that little bit faster (later on in chapter 7 we'll be looking at ways to start slowing programs down, since machine code is so fast that at times we can't even begin to see what's happening) and make them more comprehensible to anyone who looks through your coding.

Writing legible, easy-to-follow programs is important. Not only for anyone else to whom you might show your listing, but also for yourself. If you write a program liberally littered with PEEKS, POKEs, GOSUBs and GOTOs, which at the time was a model of clarity, try looking at that program in six months time. It will be nothing more than a jumbled mess that no one, not even the author, will be able to figure out.

One of our allies in achieving legible programs is the REM statement. Although taking up valuable memory, a sprinkling of REM statements around a program can do a lot to ease understanding of that program later.

However, REMs can do a little more than that, as has been pointed out by one David Gristwood. They can be used to highlight portions of a program listing by printing the said portions in a different colour

from the rest of the code. Thus subroutines can be made to stand out, and parts of the listing that are performing vital tasks can also be highlighted, making them easier to find when you look at the program long after completing it.

Before the portion to be coloured, you'll need a line something like:

Line number REM " "[DEL,RVS T[SHIFTEDM]T[COLOURCODE]

followed by RETURN.

COLOURCODE is taken from the following table:

Shifted P	- Black	Shifted A	- Orange
Shifted U	- Brown	Shifted V	- Light Red
Shifted W	- Grey #1	Shifted X	- Grey #2
Shifted Y	- Light Green	Shifted Z	- Light Blue
Shift and up-arrow	- Yellow	Shifted +	- Grey #3
		CBM and -	- Purple
CBM and *	- Cyan	' on its own	- Red
E on its own	- White		

One thing to watch out for when using REM statements, either in this way or just as a conventional:

```
10 REM START OF PROGRAM
```

is to try to avoid jumping to them either as part of a subroutine, or with a straightforward GOTO statement. REMs eat up memory, and there may well come a point when you wish to discard some of them since you're running a little low on memory. There have been many laws of computing published in the past, but an additional one must state:

"If a REM statement is removed from a program, you can guarantee that a subroutine will commence execution at that removed statement."

In other words, you might have a line:

```
1000 REM START OF SORT ROUTINE
```

and somewhere else in the program a line that has a GOSUB 1000 in it. Unlike some other BASICs, Commodore 64 BASIC is not very

tolerant when it comes to being told to branch to a line that no longer exists, and the program will come to a grinding halt.

An easy way around this, in the above example, would be to have line 999 as the REM statement, and then commence the routine proper on line 1000. In this way it doesn't matter whether the REM statement stays or goes.

Conserving memory

Since we've been chiefly concerned with producing legible listings up till now, it comes as a bit of a body blow to learn that most of the techniques used for conserving memory make programs that little bit harder to understand. But there comes a time in every program's life when it is necessary to prune things down a little, and the following points will all help to save space and make the program run that little bit faster.

(1) Programs look much more legible if lines are spaced out, as in the following example:

```
10 FOR I = 1 TO 10 : PRINT I : NEXT I
```

However, that takes up a lot more space in memory than:

```
FORI=1TO10:PRINTI:NEXT
```

Not only have we removed the I parameter after the NEXT statement (although you can't always do this if more than one loop is operative at a time), but we've also got rid of the additional 10 superfluous spaces. The line may now be harder to read, but ten spaces is ten bytes, and taken over a long program can save at least as much as a K of memory.

(2) Each line number takes up a certain amount of memory, whereas putting colons between separate statements with the same line number takes up less. Thus, it might look neater to have:

```
10 A=2
20 B=5
30 C=A*B
40 PRINTC
```

But far less wasteful is:

```
10 A=2:B=5:C=A*B:PRINTC
```

(3) Variables should always be declared at the start of a program. Furthermore, whenever a numerical constant is to be used more than once (e.g. the mathematical constant PI), it saves an awful lot of memory to define a variable to be equal to PI. Thus, we might have something like:

```
10 PI=3.14159
20 PRINT "THE AREA OF A CIRCLE WITH RADIUS 10 IS ";
30 PRINT PI*10*10
```

Note the use of 10*10 rather than 10 to the power 2. It's a lot quicker to execute.

Since so many routines on the 64 depend on the use of sound (starting memory location 54272) and sprites (53248), it makes a lot of sense to define a couple of variables SO = 54272 and SP = 53248, and then just refer to other locations as SO plus something or SP plus something.

(4) String variables take up space as soon as they're declared, regardless of the length of that string. Make the string longer, and yet more space is eaten up, but making it shorter still leaves us with a certain amount of memory occupied. Obviously it isn't possible to avoid strings at all times, but their use should be kept to a sensible minimum.

(5) Matrices eat up memory, and if you're going to define a two dimensional matrix of any size, make sure that every element of the array is used. Whether it is used or not, the Commodore 64 still reserves some space for it, so if it remains unused it's a criminal waste of memory.

(6) Subroutines certainly save on memory, since one subroutine and five GOSUBs to it take up far less space than duplicating the code five times. But beware. If a subroutine is used only once in a program, it will occupy more space than merely typing out the code, since the commands GOSUB and RETURN themselves occupy memory.

(7) It is sometimes convenient to separate out mathematical expressions with extra brackets to make them more legible. But brackets take up memory space, and it is advisable to let operator precedence sort the expression out rather than unnecessary brackets.

(8) And finally, the biggest saving of all. As a number of people have asked me how certain techniques that I've used in the past in adventure game writing actually work, we'll devote a few pages to one method of programming that can save two, three, four or more kilobytes of memory.

Data conservation

When writing adventure games, somewhere in the program you'll have to have a collection of data that forms the map for the game. Since I usually only use the four cardinal compass points, North, South, East and West, and rely on movements in other directions (such as up and down) being sorted out by the program where appropriate, in a 250 location adventure I'll use a two-dimensional array, like this:

```
DIMP (249,3)
```

Remember that the first element of an array is always referred to as element zero. Now dimensioning an array like this takes up an awful lot of memory: 5009 bytes in fact. What it also does is to reserve a section of memory at the top of the area set aside for BASIC, that will be occupied no matter whether we use all the elements of the array or not.

Since my adventures are all resident in the machine, with no hopping off to the disk drive every now and again to get a room description, all the data for the games must be residing in the machine as well. A typical element in our P(249,3) array will be a two digit number. Since our room numbers are running from 0 to 249, and a lot of the rooms will not allow you to go in particular directions, on average each element will be about two digits long.

Since this data has to be read into the program at some point, you might think that a vast assortment of data statements will be necessary somewhere along the way, and that these will be used to get the data into our array. But just think for a moment. We've got one thousand numbers to type in, each on average about two digits long, so together with line numbers, data statements, and associated commas between each data element, incorporating a collection of data statements in the main body of the program will occupy around 3K of memory, which is a lot, especially for a complex adventure program.

The solution is to store all this data on tape, and read into the program

as soon as we start running it. This way, although we still lose our 5009 bytes for the array, we don't lose that additional 3K for however many data statements it takes to define everything. They have got to be typed in at some point, but that's just some work on my part: the user never gets to see it. So before even considering the rest of the program, there is one shorter program that consists of thousands of data statements (well, tens of them anyway), and a short program to file it all onto tape (or disk) when developing the program.

```
10 DATA0,1,2,3,4,4,3,4,1,0,5,3 ....
:
:
:
:
1000 OPEN2,8,2,"0:ROOM DATA,S,W"
1002 FORI=0TO249:FORJ=0TO3
1004 PRINT#2,P(I,J):CHR$(13);
1006 NEXT J,I
```

This takes each element of the array in turn and stores it on disk in the file ROOM DATA. Note the CHR$(13), a carriage return, separating each item of data. Then that program can be deleted and the rest of the adventure sorted out. It's usually more of an adventure getting the program to work than it is solving the thing!

When it comes to playing the completed game, the first part of the program contains a routine to read in all the data from disk, rather like this:

```
2000 OPEN2,8,2,"0:ROOM DATA,S,R"
2002 FORI=0TO249:FORJ=0TO3
2004 INPUT#2,P(I,J)
2006 NEXT J,I
2008 RETURN
```

A saving of around 3K is not to be sniffed at, and any programmer who uses vast amounts of data like this would be well advised to haul everything in from tape or disk, rather than having it stored in the main program. The computer sets aside some memory anyway, there's no need for you to set aside even more.

To conclude

There are many other techniques around when it comes to saving

memory, making progams run faster, making them more legible and so on, but a clear mind and a steady head are about the best weapons at your disposal.

In the next chapter we'll talk about building up a database from scratch, but to finish off for now, here is a reasonably short program that allows you to draw in multi-colour using a high resolution screen.

Full instructions are included in the program (lines 10000 to 12004), and we'll explain the rest of the program after the listing.

```
5 GOSUB10000
10 POKE53281,7:POKE53280,0:PRINT"[CLR,BLK]HANG ON
....."
20 BASE=2*4096
30 FOR I=BASE TO BASE+7999
40 POKE I,0:NEXT I
50 POKE 53272,PEEK(53272)OR8
55 POKE 53265,PEEK(53265)OR32
60 POKE 53270,PEEK(53270)OR16
70 FOR I=1024 TO 2023
80 POKE I,1:NEXT I
90 POKE53281,7
100 X=0:Y=0:A=8:B=A:D=1:E=1:F=1:J=1
110 GETA$
115 IFA$=" "THEN13000
120 IFA$="I"THENY=Y-1:IFY<0THENY=200
121 IFA$="M"THENY=Y+1:IFY>200THENY=0
122 IFA$="A"THENX=X-1:IFX<0THENX=159
123 IFA$="D"THENX=X+1:IFX>159THENX=0
125 IFA$="[F7]"THENPOKE53281,A:A=A+1:IFA>15THENA=0
:REMF7
126 IFA$="[SHIFTEDF7]"THENPOKE53280,B:B=B+1:IFB>15
THENB=0:REMSHIFTEDF7
130 IFA$="[F1]"THENGOSUB2000
131 IFA$="[F3]"THENGOSUB3000
132 IFA$="[F5]"THENGOSUB4000
134 IFA$="Q"THEN5000
135 IFA$="1"THENC=C+1:IFC>15THENC=0
136 IFA$="3"THENG=G+1:IFG>15THENG=0
138 IFA$="5"THENH=H+1:IFH>15THENH=0
140 IFA$="7"THENJ=1-J
200 ROW=INT(Y/8)
210 CHAR=INT(X/4)
220 LINE=YAND7
230 BIT=(7-(XAND3)*2):BIT=(2^BIT)*D+(2^(BIT-1))*E:
BIT=BIT*F
240 BYTE=8192+ROW*320+CHAR*8+LINE
245 IFJ=0THENPOKEBYTE,PEEK(BYTE)AND255-BIT:GOTO110
```

```
250 POKE BYTE,PEEK(BYTE)ORBIT
260 GOTO110
2000 POKE55296+ROW*40+CHAR.C
2012 D=1:E=1:F=1:RETURN
3000 POKE1024+ROW*40+CHAR,G OR (H*16)
3012 D=1:E=0:F=1:RETURN
4000 POKE1024+ROW*40+CHAR,(H*16) OR G
4012 D=0:E=1:F=1:RETURN
5000 POKE53265,PEEK(53265)AND223:POKE53270.PEEK(53
270)AND239:POKE53272,21
5002 END
10000 POKE 53272,23:POKE53280,0:POKE53281,0
10002 PRINT"[CLR,YEL]WELCOME TO MULTI-COLOUR HIGH
RES ARTIST.
10004 PRINT"[CD]TO DRAW IN HIGH RES USING MULTI-CO
LOUR  MODE ON THE ":
10006 PRINT"COMMODORE 64, CERTAIN       CONVENTION
S HAVE TO BE OBSERVED."
10008 PRINT"[CD]EACH 8 PIXEL BY 8 PIXEL SQUARE CAN
 SHOW FOUR DIFFERENT COLOURS."
10010 PRINT"[CD]ONE OF THESE IS THE SCREEN BACKGRO
UND   COLOUR, ALTERED BY POKEING":
10012 PRINT" MEMORY LOCA- TION 53281.  THIS BECOME
S THE STANDARD  BACKGROUND COLOUR.
10014 PRINT"[CD]THE OTHER THREE COLOURS COME FROM
:
10016 PRINT"[CD]SCREEN MEMORY (I.E. LOCATION 1024
      ONWARDS),":
10018 PRINT" WITH THE TOP FOUR BITS OF THE RELEVAN
T BYTE DETERMINING ONE":
10020 PRINT" COLOUR.   AND THE BOTTOM FOUR ANOTHER
 COLOUR."
10022 GOSUB12000
10024 PRINT"[CLR]THE FINAL COLOUR COMES FROM THE C
OLOUR  MEMORY REGISTER, ";
10026 PRINT"STARTING AT MEMORY     LOCATION 55296.
"
10028 PRINT"[CD]IN THIS MODE. OUR HORIZONTAL RESOL
UTION IS REDUCED TO 160 'PIXELS'";
10030 PRINT", SINCE IT    TAKES TWO BITS TO DETERM
INE THE STATE OFEACH PIXEL."
10032 PRINT"[CD]BIT PAIRING    COLOUR DISPLAYED"
10033 PRINT"[CD]   00          SCREEN BACKGROUND C
OLOUR"
10034 PRINT"   01          UPPER FOUR BITS OF":PRI
NT"              SCREEN MEMORY
10036 PRINT"   10          LOWER FOUR BITS"
10038 PRINT"   11          COLOUR MEMORY"
10039 PRINT"[CD]AND HENCE THE AVAILABILITY OF 4 CO
LOURS PER CHARACTER SQUARE."
10040 GOSUB12000:PRINT"[CLR]USE THE FOLLOWING KEYS
```

```
:"
10042 PRINT"KEY PRESSED  ACTION
10044 PRINT"[CD]    F1       CHANGE COLOUR MEMORY
10046 PRINT"    F3       CHANGE LOWER 4 BITS
10048 PRINT"    F5       CHANGE UPPER 4 BITS
10050 PRINT"    F7       CHANGE BACKGROUND COLOUR
10052 PRINT"SHIFTED F7   CHANGE BORDER COLOUR
10054 PRINT"[CD]    1        CHANGE VALUE FOR F1
10056 PRINT"    3        CHANGE VALUE FOR F3
10058 PRINT"    5        CHANGE VALUE FOR F5
10060 PRINT"    7        TOGGLE ERASE/DRAW MODE
10061 PRINT"             OFF AND ON"
10062 PRINT"[CD]SHIFTED Q    EXIT TO LOAD NEXT PRO
GRAM"
10063 PRINT"[CD]    A        MOVE CURSOR LEFT":PRI
NT"    D        MOVE CURSOR RIGHT
10064 PRINT"    I        MOVE CURSOR UP":PRINT"
 M       MOVE CURSOR DOWN"
10065 PRINT"    _        INSTRUCTION SCREEN":GOSUB
12000:IFZZ=1THENRETURN
10066 PRINT"[CLR]BASICALLY THIS PROGRAM IS JUST ME
ANT AS A BIT OF FUN, WHILST BEING";
10068 PRINT" AN INTRODUCT-ION TO MULTI-COLOUR HIGH
-RESOLUTION     GRAPHICS AT THE";
10070 PRINT" SAME TIME."
10072 PRINT"[CD]IF YOU FEEL ADVENTUROUS, DISPLAYS
CAN BESAVED TO TAPE OR DISK USING";
10074 PRINT" AN ASSEMBLERSUCH AS EXTRAMON: THE HIG
H-RES DISPLAY";
10076 PRINT"  FILLS MEMORY LOCATIONS 8192 TO 16191
."
10078 PRINT"[CD]HAVE FUN.":GOSUB12000:POKE53272,21
:RETURN
10998 END
10999 RETURN
12000 PRINT"[CD]PRESS SPACE TO CONTINUE"
12002 GETSP$:IFSP$<>" "THEN12002
12004 RETURN
13000 POKE53265,PEEK(53265)AND223:POKE53270,PEEK(5
3270)AND239:POKE53272,23
13001 POKE53281,0:PRINT"[YEL]
13002 ZZ=1:GOSUB10040
13004 POKE53272,PEEK(53272)OR8:POKE53265,PEEK(5326
5)OR32:POKE53270,PEEK(53270)OR16
13005 PRINT"[CLR]";:POKE53281,A-1:GOTO110

READY.
```

Certain conventions have been observed in this and the remaining BASIC programs in this book, so we'd better explain what they are.

This program operates in lower case when displaying the instructions. The printer used to produce the listing (an Epson FX80) is incapable of handling lower case when it comes to listing out a Commodore program, and so certain strange effects tend to occur. If you look at line 10002 you'll see what I mean. The first word, Welcome, is meant to appear just like that: capital W, and the rest in lower case. When it comes to entering the program, type the 'elcome' in as normal, and use the shift key for the W. Whenever you see any letters in italics, they are to be entered using the shift key.

Another thing that the Epson can't cope with is the control codes used to move the cursor around, change the colour of the printing, and so on, so these have all been amended to make them legible. These codes work as follows:

[CLR]	: Clear screen	[HOME]	: Cursor home
[DEL]	: Delete character	[INST]	: Insert character
[CD]	: Cursor down	[CU]	: Cursor up
[CR]	: Cursor right	[CL]	: Cursor left

[BLK] : Control and the key with the letters BLK on it.

And so on for the rest of the colours. Note that if we want more than one code to be represented, they're typed in like this on the listing:

[CLR,YEL]

as in line 10002. DON'T TYPE IN THE COMMA! It's there purely to make everything more legible to you.

Two other quirks of the Epson need explaining. The strange character in lines 115 and 10065 represents the left arrow key (honest), and the chinaman's hat in line 230 is the up arrow key.

Program explanation

Line 5: go off to the routine to give instructions.

Line 10: change background and border colours, and print hopeful message.

Lines 20-90: set up and clear multi-colour high res screen.

Line 100: declare a few variables, like X and Y position of cursor.

Lines 110-140: get and react on a key being pressed.

Lines 200-260: find location of cursor on high res screen, and update screen.

Lines 2000-4012: determine what multi-colour we're currently drawing in, and set variables accordingly.

Lines 5000-5002: set everything back to normal and end program.

Lines 10000-10078: instructions.

Lines 12000-12004: pause for space bar routine.

Lines 13000-13005: turn off high res, print up instructions, turn on high res again, and go back to main body of program.

As the program itself says, just a bit of fun, but it does show that you can do some interesting things in BASIC!

2
Building up a Database

Introduction

One of the more useful programs on any computer is the database. Opinions seem to differ over what a database should be capable of doing, but essentially it should be able to store and retrieve information, rather like a card filing system, and it should also be able to search through the files and sort them into order on a number of different fields, rather like an overworked and underpaid secretary.

For the purposes of this book, we're going to present the database in the form of an address file, and build it up from scratch, explaining what each routine does and how it does it. In this program there is enough room to store around 150 'cards', with each card holding 8 different 'fields' of information. These are as follows:

Name of person.
Four fields for the address.
Postcode.
Telephone number.
Date of birth.

The program can search through any of those eight fields for any part of information within that field. For instance, if the person's name was GERRARD, and you decided to search on the key word ARD, it would still find GERRARD, and also pull out any other names that happened to have ARD in them.

The program can also sort into order on the first seven fields. You can't sort on date of birth, unfortunately, since the time taken to do that would be a mite prohibitive. Still, if anyone wants to take up the challenge ...

The listing will be broken down into eleven major 'chunks', the ten parts of the program which manipulate, store, retrieve, file, sort,

display, load and save the data, and the eleventh 'chunk' will be any other part of the program that doesn't fit into any of those categories. Displaying the menu, for instance, and setting up some data in the first place.

There are some interesting routines contained within the main body of the listing, so without further ado we'll start with all those parts of the program that do not fit into one of the operations categories.

The peripheral parts of the program

```
5 POKE53280,0:POKE53281,0:POKE53272,23:F=0
10 REM ************************
15 REM *** DATABASE PROGRAM ***
20 REM *** P.G.      MAY84    ***
25 REM ************************
30 DIMF1$(150),F2$(150),F3$(150),F4$(150),F5$(150)
,F6$(150),F7$(150),F8$(150)
32 REM DIMENSION FIELD ARRAYS
40 OPEN15,8,15
42 REM OPEN CHANNEL FOR READING ERROR MESSAGES
4999 REM CLEAR SCREEN AND TURN PRINT COLOUR TO YEL
LOW
5000 PRINT"[CLR,YEL]64BASE : MAIN MENU SELECTION
5010 PRINT:PRINT:PRINT"0) LOAD FILE FROM DISK
5015 PRINT:PRINT"1) SAVE FILE TO DISK
5020 PRINT:PRINT"2) ADD FILE
5025 PRINT:PRINT"3) REMOVE FILE
5030 PRINT:PRINT"4) EXAMINE FILE
5035 PRINT:PRINT"5) AMEND FILE
5040 PRINT:PRINT"6) DISPLAY ALL FILES
5045 PRINT:PRINT"7) SEARCH FILE
5050 PRINT:PRINT"8) SORT FILE
5055 PRINT:PRINT"9) QUIT PROGRAM
5060 PRINT:PRINT"PRESS NUMERIC KEY FOR REQUIRED OP
TION."
5061 GETKEY$:IFKEY$=""THEN5061
5065 IFKEY$="0"THEN5500:REM LOAD ROUTINE
5070 IFKEY$="1"THEN6000:REM SAVE ROUTINE
5075 IFKEY$="2"THEN6500:REM ADD ROUTINE
5080 IFKEY$="3"THEN7000:REM REMOVE ROUTINE
5085 IFKEY$="4"THEN7500:REM EXAMINE ROUTINE
5090 IFKEY$="5"THEN8000:REM AMEND ROUTINE
5095 IFKEY$="6"THEN8500:REM DISPLAY ROUTINE
5100 IFKEY$="7"THEN9000:REM SEARCH ROUTINE
5105 IFKEY$="8"THEN9500:REM SORT ROUTINE
5110 IFKEY$="9"THEN10000:REM QUIT ROUTINE
5115 GOTO5061
```

Explanation

Line 5 - set black border, black background, lower case, and variable flag F.

Lines 10-25 - so you know who to blame for the program.

Lines 30-32 - dimension field arrays to 150 each, one for each field.

Lines 40-42 - open up a channel to the disk for reading the error messages whenever we're loading or saving data.

Lines 4999-5060 - clear screen, and display the program menu option.

Lines 5061-5115 - wait for a key to be pressed and branch to the appropriate part of the program. If nothing's pressed, loop back and wait until it is.

```
11999 REM READ ERROR CHANNEL
12000 REM CHECK ERROR CHANNEL
12002 INPUT#15,EN$,EM$,ET$,ES$
12004 IFEM$="OK"THENRETURN
12005 REM TURN ON REVERSE FIELD
12006 PRINT:PRINT"ERROR ON DISK [RVS]":EM$
12008 CLOSE15:CLOSE2:END
60000 REM INPUT ROUTINE
60002 CM$=""
60003 REM REVERSE ON,ASTERISK,REVERSE OFF,CURSOR L
EFT
60004 PRINT "[RVS]*[OFF,CL]";
60006 GETZ$:IFZ$=""THEN60006
60008 Z=ASC(Z$):IFZ<>13ANDZ<>20ANDZ<>32AND(Z<47ORZ
>57)AND(Z<65ORZ>90)THEN60006
60010 ZL=LEN(CM$):IFZL>27THEN60014
60012 IFZ>57THENZ=Z+128:Z$=CHR$(Z):CM$=CM$+Z$:PRIN
TZ$;:GOTO60004
60013 IFZ<>13ANDZ<>20THENCM$=CM$+Z$:PRINTZ$;:GOTO6
0004
60014 IFZ=13ANDZLTHENPRINT" ":RETURN
60016 IFZ=20ANDZLTHENCM$=LEFT$(CM$,ZL-1):PRINTZ$;
60018 GOTO60004
```

Lines 11999-12008 - read the error channel. If no errors, return to the main body of the program, but if there are then abort the program and close all channels to the disk.

Lines 60000-60018 - this is our multi-purpose input routine, and deserves closer examination.

Line 60002 - declare input string to be a null one.

Line 60004 - print up prompt.

Line 60006 - wait for a key to be pressed.

Line 60008 - get the ASCII value of the key pressed. If that key wasn't the RETURN key, the DELETE key, a letter, a number, or the back-slash key then we don't want to know, so back to line 60006.

Line 60010 - check the length of the input string. If it's greater than 27 characters, then check to see if the RETURN key's been pressed.

Line 60012 - if a letter has been pressed, turn it into an upper case one, add it to the input string, display it, and loop back for more.

Line 60013 - if it's numeric, or the backslash key, then add it to the input string, display it, and loop back for more.

Line 60014 - if RETURN has been pressed, and the input string actually has a character in it, then return from the subroutine. To enter a blank field then requires that you at least press the space bar.

Line 60016 - if the delete key has been pressed and the input string has some characters in it, then take off the rightmost character and echo that to the screen.

Line 60018 - back to the prompt again.

LOAD routine

This section of the program is used to load in any data previously saved onto disk. To convert this program to run on tape, you'll need to delete lines 40, 5507, and 11999-12008, and convert lines 5506 and 6006 to read as follows:

5506 OPEN1,1,1,"DATA"

6006 OPEN1,1,0,"DATA"

It will then take an exceedingly long time to file everything onto tape, but at least it will work.

```
5499 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
5500 POKE53280,11:POKE53281,11:PRINT"[CLR,BLK]LOAD
ING DATA FILE"
5502 PRINT:PRINT:PRINT"WHEN DATA DISK IS READY, PR
ESS SPACE    BAR."
5504 GETLO$:IFLO$<>" "THEN5504
5506 OPEN2,8,2,"@0:DATA,S,R"
5507 GOSUB12000
5508 FORI=1TO150
5510 INPUT#2,F1$(I),F2$(I),F3$(I),F4$(I),F5$(I),F6
$(I),F7$(I),F8$(I)
5512 NEXTI
5514 CLOSE2:POKE53280,0:POKE53281,0:GOTO5000
```

Explanation

Line 5499 - simple REM statement explaining what's going on.

Line 5500 - change background and border colours to grey, and print a message onto the screen.

Lines 5502-5504 - print another message, and wait while the user gets the disk ready and puts it in the drive.

Line 5506 - open a sequential file for reading the data file imaginatively called DATA.

Line 5507 - rush off to check the error channel.

Line 5508 - set up a loop to be performed 150 times.

Line 5510 - input the Ith element for each of the 8 data fields.

Line 5512 - continue until the end of the loop.

Line 5514 - close the file, revert the background and border colours to black, and go back to the menu again.

SAVE routine

This is the routine that saves all your precious data onto disk (or tape if you choose to amend the program).

Since Commodore disk drives don't particularly like null strings being saved onto them, we have to incorporate a check for every field on every 'card', and if it's a null field pad it out with a single space character. This takes a little while, but at least the file gets properly prepared for saving, and prevents any headaches later.

```
5999 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
6000 POKE53280,11:POKE53281,11:PRINT"[CLR,BLK]SAVI
NG DATA FILE"
6002 PRINT:PRINT:PRINT"WHEN DATA DISK IS READY, PR
ESS SPACE    BAR."
6004 GETSA$:IFSA$<>" "THEN6004
6006 OPEN2,8,2,"@0:DATA,S,W":GOSUB12000
6007 PRINT:PRINT:PRINT"PREPARING FILE.":GOSUB6020
6008 FORI=1TO150
6010 PRINT#2,F1$(I);CHR$(13);F2$(I);CHR$(13);F3$(I
);CHR$(13);F4$(I);CHR$(13);
6012 PRINT#2,F5$(I);CHR$(13);F6$(I);CHR$(13);F7$(I
);CHR$(13);F8$(I);CHR$(13);
6014 NEXTI
6016 CLOSE2:POKE53280,0:POKE53281,0:GOTO5000
6020 FORI=1TO150
6022 IFF1$(I)=""THENF1$(I)=" "
6024 IFF2$(I)=""THENF1$(I)=" "
6026 IFF3$(I)=""THENF1$(I)=" "
6028 IFF4$(I)=""THENF1$(I)=" "
6030 IFF5$(I)=""THENF1$(I)=" "
6032 IFF6$(I)=""THENF1$(I)=" "
6034 IFF7$(I)=""THENF1$(I)=" "
6036 IFF8$(I)=""THENF1$(I)=" "
6038 NEXTI
6040 PRINT:PRINT:PRINT"FILE READY FOR SAVING."
6042 RETURN
```

Explanation

Line 5999 - simple REM statement explaining what's going on.

Line 6000 - change background and border colours to grey, and print a message onto the screen.

Lines 6002-6004 - print another message, and wait while the user gets the disk ready and puts it in the drive.

Line 6006 - open a sequential file for writing the data file, and go off to check the error channel.

Line 6007 - tell the user that the file is being prepared, and go to the subroutine at line 6020.

Line 6008 - set up a loop to be performed 150 times.

Lines 6010-6012 - print all the data fields for each card onto disk, separating each one with a carriage return.

Line 6014 - carry on until the loop is finished.

Line 6016 - close the file, revert back to black background and border, and go off to the main menu again.

Lines 6020-6042 - go through every field on every 'card', and if that field is a null one then pad it out with an empty string. Then rush back to line 6008 again to carry on saving the file.

Adding a file to the record

This routine is called up whenever the user wishes to add a file to the main collection of 'cards'. The user can choose which number he wishes to call the file, and if you want to put file number 149 immediately after file 2 then that's fine by me. However, you might have to wait a long time if you then decide to flip through every file in turn before getting to the one you want.

```
6499 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
6500 POKE53281,1:POKE53280,1:PRINT"[CLR,BLK]ADDING
 FILE":PRINT:PRINT
6502 INPUT "FILE NUMBER FOR NEW FILE";F:IFF<0ORF>1
50THEN6502
6503 F(F)=F
6504 PRINT:PRINT:PRINT "NAME ";:GOSUB60000:F1$(F)=
CM$
6506 PRINT "ADDRESS 1 ";:GOSUB60000:F2$(F)=CM$
6508 PRINT "ADDRESS 2 ";:GOSUB60000:F3$(F)=CM$
6510 PRINT "ADDRESS 3 ";:GOSUB60000:F4$(F)=CM$
6512 PRINT "ADDRESS 4 ";:GOSUB60000:F5$(F)=CM$
6514 PRINT "POSTCODE ";:GOSUB60000:F6$(F)=CM$
6516 PRINT "TELEPHONE NUMBER ";:GOSUB60000:F7$(F)=
CM$
6518 PRINT "DATE OF BIRTH (DD/MM/YY) ";:GOSUB60000
:F8$(F)=CM$
6520 PRINT:PRINT:PRINT"RECORD ADDED.":FORI=1TO2000
:NEXT
6522 POKE53281,0:POKE53280,0
6524 GOTO5000
```

Explanation

Line 6499 - just the usual REM statement.

Line 6500 - change to a white background and a white border, and print up a message.

Line 6502 - input the number of the file to be added, and check that the user doesn't enter a number less than 1 or greater than 150

Line 6503 - set our file record to equal the number typed in.

Line 6504 - print up the message 'name', and go to the input routine starting at line 60000. On returning, the input string CM$ is assigned to the first field for this new file.

Lines 6506-6518 - ditto for the other seven fields in our file.

Line 6520 - print up appropriate message, and set up a loop to give the user time to read it.

Line 6522 - revert to a black border and a black background.

Line 6524 - back to the main menu again.

Removing a file

This routine is called whenever a record is to be removed. This just sets every field of that record to be a null one, and thus when searching through it or attempting to sort it this record will be treated by the program as if it no longer existed.

```
6999 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
7000 POKE53280,1:POKE53281,1:PRINT"[CLR,BLK]REMOVE
 FILE"
7002 PRINT:PRINT:INPUT "FILE NUMBER TO BE REMOVED"
:F:IFF<00RF>150THEN7002
7004 PRINT:PRINT:PRINT"ARE YOU SURE (Y OR N)?"
7006 GETRE$:IFRE$="Y"THEN7012
7008 IFRE$="N"THENPOKE53280,0:POKE53281,0:GOTO5000
7010 GOTO7006
7012 F1$(F)="":F2$(F)="":F3$(F)="":F4$(F)=""
7014 F5$(F)="":F6$(F)="":F7$(F)="":F8$(F)=""
7016 PRINT:PRINT:PRINT"RECORD REMOVED."
7018 FORI=1TO2000:NEXT
7020 POKE53281,0:POKE53280,0:GOTO5000
```

Explanation

Line 6999 - our usual friendly REM statement.

Line 7000 - change to a white border and a white background, and print up suitable message.

Line 7002 - ask the user for the file number to be removed, and check that they enter a number greater than 0 and less than 151.

Line 7004 - check that they really want to remove this file.

Line 7006 - they do, so carry on with the routine by going to line 7012.

Line 7008 - they chicken out, so revert to a black border and a black background and go to the menu routine again.

Line 7010 - nothing's been pressed, so back to line 7006 again.

Lines 7012-7014 - nullify every field on the Fth card.

Lines 7016-7018 - print out suitable message and set up a loop to give the user time to read it.

Line 7020 - revert to black background and black border, and go to the menu section again.

Examining a file

This menu option is included to give the user the chance to look at selected files simply by inputting a file number. The program will then display that file on the screen, before going back to the menu again at the press of a suitable key.

```
7499 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
7500 POKE53281,1:POKE53280,1:PRINT"[CLR,BLK]EXAMIN
E FILE"
7502 PRINT:PRINT:INPUT"FILE NUMBER TO BE EXAMINED"
:F:PRINT:PRINT
7503 IFF<0ORF>150THEN7502
7504 PRINT "NAME                ":F1$(F)
7506 PRINT "ADDRESS 1           ":F2$(F)
7508 PRINT "ADDRESS 2           ":F3$(F)
7510 PRINT "ADDRESS 3           ":F4$(F)
7512 PRINT "ADDRESS 4           ":F5$(F)
7514 PRINT "POSTCODE            ":F6$(F)
7516 PRINT "TELEPHONE NUMBER    ":F7$(F)
7518 PRINT "DATE OF BIRTH       ":F8$(F)
7520 PRINT:PRINT:PRINT"PRESS SPACE BAR TO CONTINUE
."
7522 GETANY$:IFANY$<>" "THEN7522
7524 POKE53280,0:POKE53281,0
7526 GOTO5000
```

Explanation

Line 7499 - the usual

Line 7500 - back to white display again, and a message telling you what's going on.

Line 7502 - get the user to input the number of the file he wants to examine.

Line 7503 - check that the file number is not less than zero and that it isn't greater than 150.

Lines 7504-7518 - display all the fields for the file number entered.

Line 7520 - tell the user to press the space bar to continue.

Line 7522 - and wait until he does.

Line 7524 - revert to black border and black background.

Line 7526 - back to the menu again.

Amending a file

This routine comes into play when the user wants to amend an existing file (if someone moves or changes their telephone number perhaps). It can also be used to add a file.

If a field is to remain as it is, pressing the RETURN key will take the user onto the next one. To change it, just use the delete key and the input routine at 60000 onwards takes care of the rest.

```
7999 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
8000 POKE53280,1:POKE53281,1:PRINT"[CLR,BLK]AMEND
FILE"
8002 PRINT:PRINT:INPUT "FILE NUMBER TO BE AMENDED"
;F
8003 IFF<00RF>150THEN8002
8004 PRINT:PRINT"HIT RETURN TO LEAVE A FIELD UNALT
ERED."
8006 PRINT:PRINT:PRINT"NAME ";:CM$=F1$(F):PRINTCM$
::GOSUB60004:F1$(F)=CM$
8008 PRINT"ADDRESS 1 ";:CM$=F2$(F):PRINTCM$;:GOSUB
60004:F2$(F)=CM$
8010 PRINT"ADDRESS 2 ";:CM$=F3$(F):PRINTCM$;:GOSUB
60004:F3$(F)=CM$
8012 PRINT"ADDRESS 3 ";:CM$=F4$(F):PRINTCM$;:GOSUB
60004:F4$(F)=CM$
8014 PRINT"ADDRESS 4 ";:CM$=F5$(F):PRINTCM$;:GOSUB
60004:F5$(F)=CM$
8016 PRINT"POSTCODE ";:CM$=F6$(F):PRINTCM$;:GOSUB6
0004:F6$(F)=CM$
8018 PRINT "TELEPHONE NUMBER ";:CM$=F7$(F):PRINTCM
$;:GOSUB60004:F7$(F)=CM$
8020 PRINT "DATE OF BIRTH ";:CM$=F8$(F):PRINTCM$;:
GOSUB60004:F8$(F)=CM$
8022 PRINT:PRINT:PRINT"RECORD AMENDED."
8024 FORI=1TO2000:NEXT
8026 POKE53280,0:POKE53281,0:GOTO5000
```

Explanation

Line 7999 - guess what?

Line 8000 - we're still with our white screen, and a simple message to let the user know what he's let himself in for.

Line 8002 - get the file number to be amended.

Line 8003 - and make sure it falls within the legal range.

Line 8004 - inform the user that hitting RETURN will leave a field as it was.

Line 8006 - print up the field to be altered, and put the field description (F1$(F))into the input string CM$. Print CM$ so that the user knows what he's changing, and go to the routine starting at line 60004 this time, so that the input string doesn't get set back to be a null one again. On returning from the routine, define F1$(F) to be whatever the input string now contains.

Lines 8008-8020 - as above, for the other seven fields of this file.

Line 8022 - tell the user the good (or bad?) news.

Line 8024 - set up a loop to give time to read the message.

Line 8026 - usual change to black screen, and off to the menu again.

Displaying all the files

This routine is used to allow the user to flick through every file in turn, until he's either gone through all 150 of them, or he gets bored and presses the 'Q' key to exit from the routine.

This is useful when scanning for some information that you know is in there somewhere, but for the life of you you can't remember where.

```
8499 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
8500 POKE53280,1:POKE53281,1:PRINT"[CLR,BLK]DISPLA
Y ALL FILES"
8502 PRINT:PRINT:FORI=1TO150
8503 PRINT"FILE NUMBER ";I:PRINT
8504 PRINT "NAME              ";F1$(I)
8506 PRINT "ADDRESS 1         ";F2$(I)
8508 PRINT "ADDRESS 2         ";F3$(I)
8510 PRINT "ADDRESS 3         ";F4$(I)
8512 PRINT "ADDRESS 4         ";F5$(I)
8514 PRINT "POSTCODE          ";F6$(I)
8516 PRINT "TELEPHONE NUMBER  ";F7$(I)
8518 PRINT "DATE OF BIRTH     ";F8$(I)
8520 PRINT:PRINT:PRINT"PRESS 'C' FOR NEXT RECORD
8522 PRINT"OR 'Q' TO RETURN TO MENU
8523 REM CLEAR SCREEN
8524 GETNR$:IFNR$="C"THENPRINT"[CLR]DISPLAY ALL FI
LES":PRINT:GOTO8530
8526 IFNR$="Q"THENPOKE53281,0:POKE53280,0:GOTO5000
8528 GOTO8524
8530 NEXTI
8532 NR$="Q":GOTO8526
```

Explanation

Line 8499 - friendly REM message.

Line 8500 - back to white screen again, and print up a straightforward message.

Line 8502 - set up a loop that can be performed 150 times.

Line 8503 - tell the user what file number he's currently looking at.

Lines 8504-8518 - display all the fields for that file.

Lines 8520-8522 - instructions for proceeding.

Line 8524 - check for a key press, and if he presses 'C' then it's off to the next file by going to line 8530 and taking the next step through the loop.

Line 8526 - the user wants to quit, so revert to our standard black screen and go back to the menu.

Line 8528 - nothing's been pressed, so wait until it is.

Line 8530 - next step through loop.

Line 8532 - the loop's finished, so fool the machine into thinking that a 'Q' has been pressed and go to line 8526 to finish everything off.

Searching through a file

This allows the user to search through any field, for any item that may be contained within that field.

For instance, a search on field 5 (usually the county in the person's address) for SHIRE, would pick out LANCASHIRE, BERKSHIRE, HAMPSHIRE, and so on. A search on field 1 for MI would pick out MIKE, MICHAEL, EMILY, etc. A powerful, and reasonably fast, routine.

```
8999 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
9000 POKE53280,9:POKE53281,7:PRINT"[CLR,BLK]SEARCH
 FILE":PRINT:PRINT
9002 PRINT"WHAT FIELD DO YOU WANT TO SEARCH ON?":P
RINT:PRINT
9004 PRINT"NAME                (FIELD 1)
9006 PRINT"ADDRESS 1           (FIELD 2)
9008 PRINT"ADDRESS 2           (FIELD 3)
9010 PRINT"ADDRESS 3           (FIELD 4)
9012 PRINT"ADDRESS 4           (FIELD 5)
9014 PRINT"POSTCODE            (FIELD 6)
9016 PRINT"TELEPHONE NUMBER    (FIELD 7)
9018 PRINT"DATE OF BIRTH       (FIELD 8)
9022 PRINT:PRINT:PRINT"PRESS APPROPRIATE NUMERIC K
EY"
9024 GETFS$:IFFS$=""THEN9024
9026 FS=VAL(FS$):IFFS=0ORFS>8THEN9024
9028 PRINT:PRINT:PRINT"FIELD NUMBER ";FS
9030 FORI=1TO2000:NEXT
9031 REM CLEAR SCREEN
9032 PRINT"[CLR]SEARCH FILE":PRINT:PRINT
9034 IFFS=8THEN9080
9040 PRINT"TEXT TO SEARCH FOR ? ";:GOSUB60000:TX$=
CM$
9042 ONFSGOTO9050,9053,9056,9059,9062,9065,9068,90
71
9050 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F1$(I))
9051 IFTX$=MID$(F1$(I),J,TX)THEN9112
9052 NEXTJ,I:GOTO9142
9053 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F2$(I))
9054 IFTX$=MID$(F2$(I),J,TX)THEN9112
9055 NEXTJ,I:GOTO9142
9056 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F3$(I))
9057 IFTX$=MID$(F3$(I),J,TX)THEN9112
```

```
9058 NEXTJ,I:GOTO9142
9059 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F4$(I))
9060 IFTX$=MID$(F4$(I),J,TX)THEN9112
9061 NEXTJ,I:GOTO9142
9062 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F5$(I))
9063 IFTX$=MID$(F5$(I),J,TX)THEN9112
9064 NEXTJ,I:GOTO9142
9065 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F6$(I))
9066 IFTX$=MID$(F6$(I),J,TX)THEN9112
9067 NEXTJ,I:GOTO9142
9068 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F7$(I))
9069 IFTX$=MID$(F7$(I),J,TX)THEN9112
9070 NEXTJ,I:GOTO9142
9071 TX=ZL:FORI=1TO150:FORJ=1TOLEN(F8$(I))
9072 IFTX$=MID$(F8$(I),J,TX)THEN9112
9073 NEXTJ,I:GOTO9142
9080 PRINT"SEARCH ON DAY(D), MONTH(M) OR YEAR(Y)?"
9082 GETSR$:IFSR$="D"THENL=1:GOTO9100
9084 IFSR$="M"THENL=4:GOTO9100
9086 IFSR$="Y"THENL=7:GOTO9100
9088 GOTO9082
9100 PRINT:PRINT:PRINT"INPUT NUMBER TO SEARCH FOR
";:GOSUB60000:IFZL>2THEN9100
9104 NS=VAL(CM$)
9106 FORI=1TO150
9108 IFNS=VAL(MID$(F8$(I),L,2))THEN9112
9110 NEXTI:GOTO9142
9111 REM CLEAR SCREEN
9112 PRINT "[CLR]RECORD NUMBER ";I:PRINT:PRINT
9114 PRINT "NAME                ";F1$(I)
9116 PRINT "ADDRESS 1           ";F2$(I)
9118 PRINT "ADDRESS 2           ";F3$(I)
9120 PRINT "ADDRESS 3           ";F4$(I)
9122 PRINT "ADDRESS 4           ";F5$(I)
9124 PRINT "POSTCODE            ";F6$(I)
9126 PRINT "TELEPHONE NUMBER    ";F7$(I)
9128 PRINT "DATE OF BIRTH       ";F8$(I)
9130 PRINT:PRINT"PRESS 'C' FOR NEXT RECORD
9132 PRINT"OR 'Q' TO RETURN TO MENU
9133 REM CLEAR SCREEN
9134 GETNR$:IFNR$="C"THENPRINT"[CLR]SEARCH FILES":
PRINT:GOTO9110
9136 IFNR$="Q"THENPOKE53281,0:POKE53280,0:GOTO5000
9138 GOTO9134
9142 PRINT:PRINT:PRINT"SEARCH CONCLUDED."
9144 FORI=1TO2000:NEXT
9146 POKE53280,0:POKE53281,0:GOTO5000
```

Explanation

Line 8999 - REM statement.

Line 9000 - different colours! An orange border and a yellow background, along with the message about what's going on.

Line 9002 - message to choose field to search through.

Lines 9004-9022 - which keys to press for what, and telling the user to press one of them.

Lines 9024-9026 - get a key press, and if it isn't one of the numbers 1 to 8 then go back and wait until it is.

Line 9028 - inform the user which field he's chosen to search through.

Line 9030 - and give him time to read it.

Lines 9031-9032 - clear screen and print message.

Line 9034 - if he's searching for a date (e.g. everyone whose birthday falls in August), then go off to line 9080, since this routine is handled differently from the rest.

Line 9040 - get the text to search on using the input routine at 60000.

Line 9042 - depending on the field to be searched, jump to the correct part of the program.

Line 9050 - set TX to equal the length of the string that we're looking for. Set up a loop to go through all 150 files, and set a loop to check through the entire length of the field.

Line 9051 - if a match is found for the search string anywhere in the Ith field then go to line 9112 to print everything out.

Line 9052 - nothing's been found, so continue the search. When it's all over, trot off to line 9142.

Lines 9053-9073 - ditto for all the other fields.

Line 9080 - ask the user if he wants to search on a day, a month or a year.

Lines 9082-9088 - get an input and set the variable L accordingly. L indicates at which point in the date string we're going to start looking.

Line 9100 - input the number to look for by using the routine at line 60000 onwards. If the user attempts to search for a string of more than two characters, forget it, and go back again to input a number.

Line 9104 - set NS to equal the VALue of the number.

Line 9106 - set up a loop to go through all 150 fields.

Line 9108 - if a match is found, then go to line 9112 to print everything up.

Line 9110 - carry on through all the fields, then trot off to line 9142.

Lines 9111-9128 - display the field where the match has been found.

Lines 9130-9134 - see if the user wants to look for another string, or he wants to return to the menu. If he decides to quit, it's back to our usual black screen again and the menu at line 5000.

Lines 9142-9146 - end of search and back to a black screen and the menu.

Sorting through the files

This set of routines allows the user to sort the files into order dependent on the contents of any of the first seven fields. The sort is reasonably quick on a low number of fields, but if you've got a full file you might as well go and make a cup of tea and settle down in front of the television.

```
9499 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
9500 POKE53280,9:POKE53281,8:PRINT"[CLR,BLK]FILE S
ORT":PRINT:PRINT
9502 PRINT"WHAT FIELD DO YOU WANT TO SORT ON?":PRI
NT:PRINT
9504 PRINT"NAME                (FIELD 1)
9506 PRINT"ADDRESS 1           (FIELD 2)
9508 PRINT"ADDRESS 2           (FIELD 3)
9510 PRINT"ADDRESS 3           (FIELD 4)
9512 PRINT"ADDRESS 4           (FIELD 5)
9514 PRINT"POSTCODE            (FIELD 6)
9516 PRINT"TELEPHONE NUMBER (FIELD 7)
9522 PRINT:PRINT:PRINT"PRESS APPROPRIATE NUMERIC K
EY"
9524 GETFF$:IFFF$=""THEN9524
9526 FF=VAL(FF$):IFFF=0ORFF>7THEN9524
9528 PRINT:PRINT:PRINT"FIELD NUMBER ";FF
9530 FORI=1TO2000:NEXT
9531 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
9532 PRINT"[CLR,BLK]FILE SORT":PRINT:PRINT
9533 FORJ=1TO148:IFF1$(J)=""ORF1$(J)=" "THEN9641
9534 ONFFGOTO9550,9560,9570,9580,9590,9600,9610
9550 FORI=1TO149:IFLEFT$(F1$(I+1),1)=" "THEN9641
9552 IFF1$(I)>F1$(I+1)THEN9632
9554 NEXTI
9556 GOTO 9641
9560 FORI=1TO149:IFLEFT$(F2$(I+1),1)=" "THEN9641
9562 IFF2$(I)>F2$(I+1)THEN9632
9564 NEXTI
9566 GOTO 9641
9570 FORI=1TO149:IFLEFT$(F3$(I+1),1)=" "THEN9641
9572 IFF3$(I)>F3$(I+1)THEN9632
9574 NEXTI
9576 GOTO 9641
9580 FORI=1TO149:IFLEFT$(F4$(I+1),1)=" "THEN9641
9582 IFF4$(I)>F4$(I+1)THEN9632
9584 NEXTI
```

```
9586 GOTO 9641
9590 FORI=1TO149:IFLEFT$(F5$(I+1),1)=" "THEN9641
9592 IFF5$(I)>F5$(I+1)THEN9632
9594 NEXTI
9596 GOTO 9641
9600 FORI=1TO149:IFLEFT$(F6$(I+1),1)=" "THEN9641
9602 IFF6$(I)>F6$(I+1)THEN9632
9604 NEXTI
9606 GOTO 9641
9610 FORI=1TO149:IFLEFT$(F7$(I+1),1)=" "THEN9641
9612 IFF7$(I)>F7$(I+1)THEN9632
9614 NEXTI
9616 GOTO 9641
9632 S$=F1$(I):F1$(I)=F1$(I+1):F1$(I+1)=S$
9633 S$=F2$(I):F2$(I)=F2$(I+1):F2$(I+1)=S$:S$=F3$(
I):F3$(I)=F3$(I+1):F3$(I+1)=S$
9634 S$=F4$(I):F4$(I)=F4$(I+1):F4$(I+1)=S$:S$=F5$(
I):F5$(I)=F5$(I+1):F5$(I+1)=S$
9635 S$=F6$(I):F6$(I)=F6$(I+1):F6$(I+1)=S$:S$=F7$(
I):F7$(I)=F7$(I+1):F7$(I+1)=S$
9636 S$=F8$(I):F8$(I)=F8$(I+1):F8$(I+1)=S$:GOTO955
4
9641 NEXTJ
9642 PRINT:PRINT:PRINT"SORT CONCLUDED.":FORI=1TO20
00:NEXT
9644 POKE53280,0:POKE53281,0:GOTO5000
```

Explanation

Line 9499 - REM statement.

Line 9500 - different colours again, and print up a message.

Lines 9504-9522 - display list of options and get the user to choose one.

Lines 9524-9530 - wait for a key to be pressed, and check that it fits into our chosen categories. If it doesn't, loop back until the user presses something that does, and if it does then tell him what field he's going to sort on, give him time to read the message, and zoom on to the next part of the program.

Lines 9531-9532 - just for clarity.

Line 9533 - start of the grand loop to riffle through the sort for each field. If an empty field is found then that's it for that field, so go to line 9641 to take the next step in the loop.

Line 9534 - go to the correct part of the program, dependent on which field we're looking through.

Line 9550 - go through each field in turn, and if a null string is found then go to line 9641 to take the next step in the grand J loop.

Line 9552 - compare the Ith field with its neighbour, and if a change has to be made go to line 9632 to swop everything over.

Line 9554 - next step in the I loop.

Line 9556 - and off for the next step in the J loop.

Lines 9560-9616 - as above for the other six fields.

Lines 9632-9636 - swop everything over. We may only be sorting on one field, but all the other records have to be altered as well!

Line 9641 - next step in the J loop.

Line 9642 - end of sort, so print a message and give the user time to read it.

Line 9644 : back to a black screen, and back to the menu at line 5000.

The QUIT routine

This routine simply switches everything off, but does allow the user the chance to return to the menu if he has a change of heart e.g. if he hasn't yet saved all the data to disk or tape.

```
9999 REM CLEAR SCREEN AND TURN PRINT COLOUR TO BLA
CK
10000 POKE53280,1:POKE53281,1:PRINT"[CLR,BLK]ARE Y
OU SURE (Y OR N)?"
10004 REM CLEAR SCREEN
10005 GETSURE$:IFSURE$="Y"THENPRINT"[CLR]":END
10010 IFSURE$="N"THENPOKE53280,0:POKE53281,0:GOTO5
000
10015 GOTO10005
```

Explanation

Line 9999 - REM statement.

Line 10000 - revert to white screen, and check that the user is convinced.

Line 10005 - he is, so end the program.

Line 10010 - he's not, so back to a black screen and the menu again.

Line 10015 - nothing suitable has been pressed yet, so back to line 10005.

Conclusion

This program is certainly not the world's most amazing database, although it certainly works and could easily be amended if necessary to file items other than names, addresses, birthdays and telephone numbers. Stock control is one application that springs to mind.

It serves its purpose in that it shows how what at first sight may seem a relatively complicated program can be simply built up in a series of stages, or modules. The prospect is not as daunting as you might at first think.

Before we leave BASIC entirely, and dive into the depths of machine code, one thing we're going to do throughout this book is build up a simple little arcade game (zap the aliens, the usual stuff) in machine code. However, this game has a BASIC opening, which prints up the title screen and draws a range of mountains. Type it in and save it to tape, but don't run it: we haven't got any machine code in there yet!

Fergus

Our hero is a character that my wife, for some reason best known to her, decided to call Fergus. Fergus can be moved around all over the screen, and has to dodge and shoot the various enemies that come at him: evil Horaces on skis, Pinball tables, hippies, and other assorted nasty beings.

Be very careful when you type in lines 18-26, which form the mountain range at the bottom of the screen. Follow the REM statements closely.

As I said, don't run the program yet. We won't be doing that till the end of the next chapter.

```
4 REM CLEAR SCREEN,PRINT 10 CURSOR RIGHTS, AND THE
N TURN COLOUR TO BLACK
5 POKE53280,7:POKE53281,7:PRINT"[CLR,10CL,BLK]****
  FERGUS  ****"
10 FORI=1TO50:A=INT(RND(.5)*728+41):B=INT(RND(.5)*
14+1)
12 POKE1024+A,46:POKE55296+A,B:NEXT
13 REM CURSOR HOME, THEN PRINT 23 CURSOR DOWNS
14 PRINT"[HOME,23CD]";:
```

```
15 REM TURN PRINT COLOUR TO GREEN, AND GO INTO REV
ERSE FIELD MODE
16 REM A REPRESENTS THE SHIFTED POUND SYMBOL, AND
B REPRESENTS THE
17 REM '*' KEY PRESSED IN CONJUNCTION WITH THE CBM
 LOGO KEY
18 PRINT"[GRN,RVS,A,CU,A,CU,A,CU,A,B,CD,B,CD,B,A,C
U,A,CU,A,B,CD,B,CD,B,A,CU,A]";
19 PRINT"[CU,A,CU,A,B,CD,B,CD,B,CD,B,CD,B,A,CU,A,C
U,A,B,CD,B,A,CU,A,CU,A,B,CD";
20 PRINT"[B,CD,B,CD,,B,A,CU,A,CU,A,B,CD,B,CD,B]";
21 PRINT"[4CU,16CR,RVS,2SP]"
22 PRINT"[RVS,3CR,2SP,4CR,2SP,4CR,4SP,10CR,2SP]"
24 PRINT"[RVS,2CR,4SP,2CR,4SP,2CR,6SP,4CR,2SP,2CR,
3SP,4CR,2SP]"
26 PRINT"[RVS,CR,20SP,2CR,10SP,2CR,4SP]";
27 REM CURSOR HOME, CHANGE PRINT COLOUR TO YELLOW,
 AND GO INTO REVERSE MODE
28 PRINT"[HOME,YEL,RVS]000000 SCORE  LIVES 3  HI-S
CORE   0000"
30 FORI=1984TO2023:POKEI,160:NEXT
32 FORI=56056TO56295:POKEI,5:NEXT
40 SYS 49152

READY.
```

3
Machine Code: a Starting Point

Introduction

Most of you will probably have seen some machine code routines in magazines and publications, and won't have the faintest idea about what it all means. This is reasonable, since most machine code listings look unintelligible enough to the best of people. However, by the end of the next couple of chapters you'll be able to pick out most of what's going on in a listing, as most machine code programs make extensive use of a very limited set of commands.

Machine code on the 64 only allows us to use 56 commands anyway, although it's fair to point out that most of these commands can operate in a variety of different ways. But only about half of these commands are commonly used, and only about half of that number are used very extensively. If you were asked to learn 28 different words in, say, an obscure African dialect, I'm sure that most of us could manage that in a few days. We are going to concentrate on those most commonly used instructions, even if we do cover the whole lot somewhere in this book.

Most introductions to machine code start off by telling you why you should learn about it. 'Programs can be written that operate at lightning speeds', is the main reason given. Personally I regard those sort of arguments as a waste of space. You know why you want to learn all about machine code, or presumably you wouldn't be holding a copy of this book in the first place.

Chapters four and five are going to be our first look at machine code proper, and in this chapter we're merely going to present you with a few listings, and an explanation of how they do what they do. If you get totally lost and confused, read the next two chapters and then come back here again.

To assemble, or not to assemble

Most people seem to think that you need some kind of an assembler to program in machine code. An assembler is a program that translates all the various symbols and numbers that you're going to be typing in over the course of the next couple of hundred pages, into something that looks a bit more sensible and intelligible to you. Good assemblers can go a lot further than that, but this is an introduction to machine code programming, so we'll ignore the more complicated stuff for now.

Simple assemblers abound for the Commodore 64, and, courtesy of 'legend in his own lunchtime' Jim Butterfield, Duckworth have in the past published the popular program Extramon. This, in retrospect, was a great mistake, since only about 2 people out of every thousand have managed to get the thing working, and the other 998 have all written in and told us that they can't get anywhere with it. Understandably frustrating for them, and not exactly satisfying for us either, so the program now exists on the tape of the book *Sprites and Sounds on the Commodore 64*, as well as being released by Argus Specialist Press a while ago in their 64 Tape magazine (issue number one, if you want to track it down).

There are others around, including the packages Sysres, PAL, and more, so if you're really earnest about getting to grips with machine code then I suggest that you acquire at least one of these programs before proceeding further. Without one life will not be impossible, but it will be very difficult.

Simple Border Routine

The following program is a reasonably short machine code listing for drawing a border around the screen. It serves to show the speed of machine code when compared to the speed of BASIC (and there's a more dramatic demonstration to follow later), and since it uses the grand total of 15 different commands, there won't be too much difficulty in following what's going on.

As I said, if you do get lost and your brain refuses to function any more, turn to chapters 4 and 5, and then come back here again.

BORDER ROUTINE

```
B*
   PC  SR AC XR YR SP
.;87D8 33 00 C0 00 F6
.
C000 A9 66       LDA #$66
C002 A2 27       LDX #$27
C004 A0 00       LDY #$00
C006 9D 00 04    STA $0400,X
C009 9D 00 D8    STA $D800,X
C00C 9D C0 07    STA $07C0,X
C00F 9D C0 DB    STA $DBC0,X
C012 CA          DEX
C013 D0 F1       BNE $C006
C015 8D 00 04    STA $0400
C018 8D 00 D8    STA $D800
C01B 8D C0 07    STA $07C0
C01E 8D C0 DB    STA $DBC0
C021 A2 F0       LDX #$F0
C023 9D 00 04    STA $0400.X
C026 9D 00 D8    STA $D800,X
C029 9D 27 04    STA $0427,X
C02C 9D 27 D8    STA $D827,X
C02F 9D F0 04    STA $04F0,X
C032 9D F0 D8    STA $D8F0,X
C035 9D 17 05    STA $0517,X
C038 9D 17 D9    STA $D917,X
C03B 9D E0 05    STA $05E0,X
C03E 9D E0 D9    STA $D9E0,X
C041 9D 07 06    STA $0607,X
C044 9D 07 DA    STA $DA07,X
C047 9D D0 06    STA $06D0,X
C04A 9D D0 DA    STA $DAD0,X
C04D 9D F7 06    STA $06F7,X
C050 9D F7 DA    STA $DAF7,X
C053 8D 00 C1    STA $C100
C056 8A          TXA
C057 38          SEC
C058 E9 28       SBC #$28
C05A AA          TAX
C05B AD 00 C1    LDA $C100
C05E E0 00       CPX #$00
C060 D0 C1       BNE $C023
C062 60          RTS
C063 00          BRK
.
-
```

Now for the difficult part: typing it in. Since we haven't yet explained what any of the symbols are, you're going to have to enter it like a parrot. Get an assembler up and running, and type in the command D C000. This means disassemble the contents of the computer's memory, starting at memory location C000.

Do what? You've probably heard of the terms decimal and hexadecimal. These are two different numerical counting systems, and the one that we're used to using is the decimal one. That is, numbers are counted to a base of 10, so that the number 1234 really stands for:

4 times 10 to the power 0, plus
3 times 10 to the power 1, plus
2 times 10 to the power 2, plus
1 times 10 to the power 3.

Since 10 to the power of 0 is mathematically defined to equal 1, our sum comes out to be 4 times 1, or 4, plus 3 times 10, or 30, plus 2 times 100, or 200, plus 1 times 1000, or 1000. Thus the end result is 1000 plus 200 plus 30 plus 4, or 1234: the number we started with.

When working in machine code you're not only going to have to understand decimal, but also hexadecimal. This counting system no longer uses the base 10, but instead we use the base 16. Thus, if we were referring to a hexadecimal number 1234, this would be translated to a decimal number as:

4 times 16 to the power 0, plus
3 times 16 to the power 1, plus
2 times 16 to the power 2, plus
1 times 16 to the power 3.

This in turn is equal to 4, plus 3 times 16, or 48, plus 2 times 256, or 512, plus 1 times 4096. This gives us our decimal equivalent of the hexadecimal number 1234 to be equal to 4 plus 48 plus 512 plus 4096, which equals 4660.

We use the numbers 0 to 9 to represent our counting system: quite handy for a numerical base of ten. However, you may be wondering how a numerical base of 16 manages to cope. Well, not only do we use the numbers 0 to 9, but we also use the letters A, B, C, D, E and F, where the hexadecimal letter A represents the decimal number 10, the hexadecimal letter B represents the decimal number 11, and so on, until we reach the hexadecimal letter F, which represents the decimal number 15.

Thus, the hexadecimal number A000 (to keep life simple), represents in decimal:

0 times 16 to the power 0, plus
0 times 16 to the power 1, plus
0 times 16 to the power 2, plus
10 times 16 to the power 3

which equals 40960.

Going back to our earlier command D C000, C000 is a hexadecimal number. This is equal to C (or 12) times 16 to the power 3, or 49152. This happens to be the start of the spare 4K of memory which sits inside the Commodore 64, and occupies memory locations 49152 to 53247. This is a useful area in which to put some short machine code routines, since it doesn't take up any of the computer's memory.

Let's get back to our program. If you look at the listing, you'll see that the first line consists of:

C000 A9 66 LDA #$66

This tells the computer (and us) that at memory location C000 (or 49152) sits the hexadecimal number A9, equivalent to the decimal number:

9 times 16 to the power 0, plus
10 times 16 to the power 1

which is equal to the decimal number 169. At memory location 49153, one further on, sits the hexadecimal number 66, equivalent to the decimal number:

6 times 16 to the power 0, plus
6 times 16 to the power 1

which is equal to the decimal number 102. Further on we come to the strange word LDA, followed by #$66. If you look at the table of machine code instructions at the back of this book, and track down the one headed LDA, you'll see that those letters stand for LoaD the Accumulator. You don't need to worry about what an accumulator is just yet, we'll be finding out more about that later. The next lot of symbols, the #$66, tells us that we're LoaDing the Accumulator with the hexadecimal number 66, or the decimal number 102. You can treat this as reasonably analogous to the BASIC statement:

LET A=66

In other words, we're assigning a value to this mysterious object the accumulator. Looking again through the listing, you'll see that the first number is always a memory location, the next little lot (a collection of one, two or three numbers) is all hexadecimal numbers, and the third lot is a collection of mnemonics and numbers.

To compare it to BASIC again for a moment, you can regard it as LINE NUMBER followed by STATEMENT followed by a REM statement explaining what's going on. In other words, the assembly listings as presented throughout this book are all fairly similar to BASIC listings, although what they achieve is way, way beyond what BASIC can ever hope to do.

Typing in the listing

If you've got an assembler, the command D C000 should have brought up one page full of memory locations, one lot of hexadecimal numbers (probably FF, which doesn't mean much to the computer: it's the decimal number 255 to save you working it out), and a collection of mnemonics, or more usually at this stage a lot of question marks, since the computer doesn't know what the hexadecimal number FF is meant to tell it to do. This is just as well, since it isn't telling it to do anything at all, as yet. Some disassemblers require that you press the stop key to prevent pages and pages of information being displayed: an annoying and unnecessary requirement.

If you now proceed to type in the first little lot of numbers (A9 66 in our example) by moving the cursor so that it stands over the first FF after the C000 message (and don't worry about moving the cursor down to type the 66 in: just enter it straight after the A9 number), followed by a carriage return, the screen display should change to resemble the first line of our program, with a lot of FFs and question marks further down the screen.

Typing in the next row (A2 27) next to the C002, which will now be immediately under the C000, should bring up the second line of the program, and so on. Type it all in slowly and carefully, and save it onto tape with the command:

S "BORDER",01,C000,C063

or onto disk with the command:

S "0:BORDER",08,C000,C063

The S stands for SAVE, the name in quotes follows the usual basic rules for filenames, the first number after the quotes represents the device number that we want to save our program onto, and the third and fourth numbers represent the first memory location that we want to save and the last memory location respectively.

When you're sure that everything is correct, you can run the program by coming out of the disassembler (most of them require you to type an X and a carriage return). Change the background colour to white so that we can see what's happening by typing in:

POKE 53281,1

and then type SYS 49152, followed by RETURN. A border will instantly be drawn around the screen, provided, of course, that you've typed the program in correctly. Control should then be returned to you again, and the usual READY message with the flashing cursor underneath should confidently be displayed on the screen.

Type SYS 49152 a few times (clear the screen before you do, otherwise nothing will appear to be happening) just to get a feel for the power of machine code. You don't yet understand how the program is doing what it is doing, but at least you can now look at machine code listings without feeling too daunted. Most of the commands used in our border routine are in that common group mentioned earlier that get used all the time, and they will, over the next couple of chapters, become very familiar friends indeed.

No assembler?

Well, I said it wasn't impossible without an assembler, merely very difficult, and that is indeed the case.

You really have to understand how to convert decimal to hexadecimal and back again, because without an assembler every number you see in that listing is going to have to be hand converted, and then POKEd into memory. To get you started, A9 represents the number 169, and 66 the number 192, so your first two POKEs will be:

POKE 49152,169:POKE49153,102 <RETURN>

The next line then says A2 27, which in decimal represents 162 and

39, so your next two POKEs will be:

POKE 49154,162:POKE49155,39 <RETURN>

Given time and a lot of patience you'll eventually get the program into memory, and can then type in the command SYS 49152 like everyone else. Only you'll find it very hard to save your program to tape or disk. Well, impossible in fact, so be prepared for a severe disappointment when you've finished drawing a few borders.

Starting to count

Now that you're getting a bit more familiar with what a machine code program looks like, how it's built up and how to enter it into memory, the following set of programs should serve to convince you of how fast this language really is.

These programs were first presented aeons ago by Mike Gross-Niklaus for the Commodore PET, but a quick dusting off and translating for the 64 will give a remarkable demonstration. Enter and run the BASIC program headed 'MILLION COUNT BASIC'.

```
MILLION COUNT BASIC

10 POKE53281,0:POKE53280,0
14 REM CLEAR SCREEN, TURN ON REVERSE MODE, GO INTO
 YELLOW PRINT MODE
15 PRINT"[CLR,RVS,YEL,6SP]"
20 SC=1024:N=5
30 FORI=1024TO1029
40 POKEI,48
50 NEXT
60 D=N
70 A=PEEK(SC+D):A=A+1
80 IFA<58THENPOKESC+D,A:GOTO60
90 POKESC+D,48
100 D=D-1
110 IFD=-1THENEND
120 A=PEEK(SC+D)
130 A=A+1
140 GOTO80

READY.
```

The idea of the program is that it counts up to a million in the top left hand corner of the screen, by the rightmost digit all the time (when that reaches 9, the one next to it is updated and the rightmost one set to zero again). If the next digit in reaches the total of nine when it comes round to updating, that is set to zero as well and the one next to that updated. The program goes on (and on, and on) until the count finally reaches a million. Don't bother waiting for it, as it takes around 7½ hours!

Yes, I know it can be done faster, but the program is meant to be a direct comparison with the machine code one, something we'll be coming back to later.

Okay, get the trusty disassembler out and type in the program headed 'MILLION COUNT M/C'.

```
MILLION COUNT M/C


B*
   PC  SR AC XR YR SP
.;678C 33 00 74 00 F6
.
C000 A2 05       LDX #$05
C002 A9 30       LDA #$30
C004 9D 00 04    STA $0400,X
C007 CA          DEX
C008 10 FA       BPL $C004
C00A A2 05       LDX #$05
C00C BD 00 04    LDA $0400,X
C00F 18          CLC
C010 69 01       ADC #$01
C012 C9 3A       CMP #$3A
C014 F0 06       BEQ $C01C
C016 9D 00 04    STA $0400,X
C019 4C 0A C0    JMP $C00A
C01C A9 30       LDA #$30
C01E 9D 00 04    STA $0400,X
C021 CA          DEX
C022 10 01       BPL $C025
C024 60          RTS
C025 BD 00 04    LDA $0400,X
C028 18          CLC
C029 69 01       ADC #$01
C02B 4C 12 C0    JMP $C012
C02E 60          RTS
```

Save it to tape or disk when you've finished, and then exit the disassembler as usual. Type a CLR and a NEW to sort BASIC out (don't worry, you won't lose the machine code program), and then enter the short program headed 'BASIC TESTING PROGRAM'.

```
BASIC TESTING PROGRAM


10 POKE53281,0:POKE53280,0
14 REM CLEAR SCREEN, TURN ON REVERSE MODE, GO INTO
 YELLOW PRINT MODE
15 PRINT"[CLR,RVS,YEL,6SP]"
20 T=TI
30 SYS49152
40 PRINT:PRINT:PRINT"TIME TAKEN = ";(TI-T)/60;" SE
CONDS"
50 END
```

When you're satisfied with it, RUN it, and if you can watch the digits changing you're a better man than I. The count this time takes a mere 27.7 seconds (approximately), as compared to about 7 ½ hours. Quite an improvement. And in case you're thinking that the program isn't really counting to a million, it is, as we shall see in the next couple of chapters.

Final programs

You may by now be wondering what is the point of typing in these programs when you haven't a clue what's really happening. Well, we've constantly referred to the next two chapters as the ones that get the ball rolling, so to speak. The object of this one is to get you used to entering listings, seeing how fast machine code is, and perhaps whetting your appetite for more.

By now you should be au fait with entering machine code listings, saving programs onto tape or disk, and generally getting an idea of how powerful this language is. (As a by-line, if you want to load the programs back from tape into the computer when there's no disassembler resident, you'll have to LOAD "BORDER",1,1 instead of the usual LOAD "BORDER". This tells the computer NOT to load the program in at the start of BASIC, where it would normally go, but at the place where it was saved from: in our case, starting at memory location 49152 decimal, C000 hexadecimal).

It is convenient, for the purposes of this book, to have these programs

typed in and working by the time we get to explaining precisely how they work. If you have them on the screen in front of you while we go through changing a few things, explaining how this command works, how that one operates, and so on, it's a lot easier to grasp when the programs are already up and running than it is when you're still trying to type them in. You can see what's happening, having already overcome the horrors of typing them in.

Sprite data

The games listing, when it reaches its final stages, uses a number of sprites, and the data for these in hexadecimal form is given next.

```
B*
   PC  SR AC XR YR SP
.;87D8 33 00 C0 00 F6
.
.:0340 C0 18 03 C0 18 03 C0 3C
.:0348 03 60 7E 06 30 C3 0C 1F
.:0350 81 F8 0F 66 F0 0F 66 F0
.:0358 0F 00 F0 07 24 E0 01 99
.:0360 80 00 C3 00 00 FF 00 01
.:0368 BD 80 03 00 C0 06 00 60
.:0370 06 00 60 06 00 60 06 00
.:0378 60 06 00 60 0F 00 F0 00
.
.
.:0380 00 18 00 00 18 00 00 18
.:0388 00 00 3C 00 00 3C 00 00
.:0390 7E 00 00 7E 00 00 FF 00
.:0398 00 FF 00 01 DB 80 03 99
.:03A0 C0 07 3C E0 07 66 E0 07
.:03A8 66 E0 07 C3 E0 0F E7 F0
.:03B0 0F 7E F0 0F 3C F0 1F 18
.:03B8 F8 3B 00 DC 73 00 CE 00
.
.
.:3E40 80 3E 01 80 22 01 80 41
.:3E48 01 80 FF 81 C1 E3 C3 41
.:3E50 FF C2 61 DD C6 31 C9 CC
.:3E58 1F C9 F8 00 EB 80 00 EB
.:3E60 80 00 FF 80 00 FF 80 0F
.:3E68 C3 F0 0F 81 F0 1F 81 F8
.:3E70 39 C3 9C 39 C3 9C 73 81
.:3E78 CE E7 00 E7 C6 00 63 00
.:3E80 00 0F C0 00 1F E0 00 3B
.:3E88 F0 00 33 F8 00 7F F8 00
```

```
.:3E90 5E F8 00 3E F8 00 31 F0
.:3E98 00 1F F0 00 1F F0 00 1C
.:3EA0 E0 10 39 C0 10 73 80 90
.:3EA8 E7 00 88 E7 00 84 73 80
.:3EB0 42 39 C0 21 FF FF 10 1C
.:3EB8 00 08 0E 00 07 FF FF 00
.:3EC0 C6 78 C0 C6 FC C0 C6 CC
.:3EC8 C0 FE CC C0 FE CC C0 C6
.:3ED0 CC C0 C6 FC 00 C6 78 C0
.:3ED8 00 00 00 00 00 00 06 33
.:3EE0 C6 06 37 E6 06 36 66 07
.:3EE8 F6 66 07 F6 66 06 36 66
.:3EF0 06 37 E0 06 33 C6 00 00
.:3EF8 00 00 00 00 00 00 00 00
.:3F00 00 00 00 00 7E 00 03 FF
.:3F08 C0 07 FF E0 0F FF F0 0F
.:3F10 FF F0 1F FF F8 1F FF F8
.:3F18 3F FF FC 3F E7 FC 3F 81
.:3F20 FC 3F A5 FC 3F A5 FC 3F
.:3F28 81 FC 3E 99 7C 7E 99 7E
.:3F30 7C 42 3E 7C 5A 3E 78 7E
.:3F38 1E 70 E7 0E 60 99 06 00
.:3F40 00 38 00 00 7C 00 00 7C
.:3F48 00 00 7C 00 18 7C 30 18
.:3F50 7C 30 1F C7 F0 1F C7 F0
.:3F58 18 44 30 18 44 30 00 7C
.:3F60 00 18 44 30 18 44 30 1F
.:3F68 FF F0 1F FF F0 18 7C 30
.:3F70 18 7C 30 00 38 00 00 38
.:3F78 00 00 38 00 00 10 00 00
.:3F80 00 1F F8 00 10 08 00 16
.:3F88 68 00 10 08 00 15 A8 00
.:3F90 14 28 00 13 C8 00 10 08
.:3F98 00 1F F8 00 20 18 00 5B
.:3FA0 28 00 9B 48 01 6C 88 02
.:3FA8 6D 08 0C 03 00 0F FC 00
.:3FB0 08 84 00 08 04 00 08 04
.:3FB8 00 08 04 00 08 04 00 00
.
.
.:37C0 00 18 00 00 18 00 00 3C
.:37C8 00 00 7E 00 00 C3 00 0F
.:37D0 81 F0 3F 66 FC 6F 66 F6
.:37D8 CF 00 F3 C7 18 E3 C1 A5
.:37E0 83 C0 C3 03 00 FF 00 01
.:37E8 BD 80 03 00 C0 06 00 60
.:37F0 06 00 60 06 00 60 06 00
.:37F8 60 06 00 60 0F 00 F0 00
.
.
```

As you can see, this is in four distinct blocks, one for our hero Fergus, one for the missiles which he launches at the enemy, one large group for the six different kinds of enemy, and one final one for poor Fergus when he loses a life.

To type them in, if we use the first group as an example, get the assembler out and type M 0340,0378. You'll see that the numbers displayed after the M correspond to the first line of numbers and the last line, and the M incidentally stands for display Memory. The screen will fill with numbers, probably a lot of zeros, and to enter your Fergus data just move the cursor up to the first set of zeros after the number 0340 (this is a memory location expressed in hexadecimal, just as the C000 was earlier), and type in the next eight hexadecimal numbers. Then hit RETURN, and those numbers will be entered into the computer's memory.

Enter the entire 8 lines (8 by 8, or 64 bytes of data for a sprite, remember) and then save it onto tape or disk as per usual. Then enter and save the rest of the sprite data, before moving on to the final program.

Moving Around

This is our first attempt at something lengthy, and you may be disappointed to learn that all it does is set a sprite up and allow it to move around the screen under keyboard control.

More complicated than BASIC, I agree, but you try getting a sprite to move this fast in BASIC. Type it in as usual, but don't worry about the blocks that have got ??? or BRK next to them, just move on to the next lot. Those are gaps that will be filled in later.

In particular, there's a large gap from location C0CD to C0FF, and all of that can be missed out. We're just interested there in the little routine from location C100 to C10B, which is there to slow everything down. Slow it down? In BASIC you'd be trying to speed it up, but here, without this delay loop, you wouldn't be able to see what's happening. Try it, and find out.

```
B*
   PC  SR AC XR YR SP
.;8FEB 33 00 D3 00 F6
.
C000 A9 0D      LDA #$0D
C002 8D F8 07   STA $07F8
C005 A9 01      LDA #$01
C007 8D 15 D0   STA $D015
C00A A9 32      LDA #$32
C00C 8D 00 D0   STA $D000
C00F 8D 01 D0   STA $D001
C012 A9 00      LDA #$00
C014 8D 17 D0   STA $D017
C017 8D 1D D0   STA $D01D
C01A A2 00      LDX #$00
C01C 8E 10 D0   STX $D010
C01F A0 00      LDY #$00
C021 AD C5 00   LDA $00C5
C024 C9 0A      CMP #$0A
C026 F0 1F      BEQ $C047
C028 C9 12      CMP #$12
C02A F0 45      BEQ $C071
C02C C9 21      CMP #$21
C02E F0 70      BEQ $C0A0
C030 C9 24      CMP #$24
C032 F0 7F      BEQ $C0B3
C034 C9 01      CMP #$01
C036 F0 03      BEQ $C03B
C038 4C 21 C0   JMP $C021
C03B 20 C7 C0   JSR $C0C7
C03E 4C 21 C0   JMP $C021
C041 FF         ???
C042 FF         ???
C043 FF         ???
C044 FF         ???
C045 FF         ???
C046 FF         ???
C047 E0 00      CPX #$00
C049 D0 03      BNE $C04E
C04B 4C 21 C0   JMP $C021
C04E CA         DEX
C04F 20 00 C1   JSR $C100
C052 8E 00 D0   STX $D000
C055 EA         NOP
C056 EA         NOP
C057 EA         NOP
C058 E0 00      CPX #$00
C05A F0 03      BEQ $C05F
C05C 4C 21 C0   JMP $C021
C05F EC 10 D0   CPX $D010
C062 F0 F8      BEQ $C05C
```

```
C064 A2 FF      LDX #$FF
C066 A9 00      LDA #$00
C068 8D 10 D0   STA $D010
C06B 8E 00 D0   STX $D000
C06E 4C 21 C0   JMP $C021
C071 E0 FF      CPX #$FF
C073 D0 03      BNE $C078
C075 4C 21 C0   JMP $C021
C078 E8         INX
C079 20 00 C1   JSR $C100
C07C 8E 00 D0   STX $D000
C07F E0 FF      CPX #$FF
C081 D0 F2      BNE $C075
C083 A9 00      LDA #$00
C085 CD 10 D0   CMP $D010
C088 D0 EB      BNE $C075
C08A A9 01      LDA #$01
C08C 8D 10 D0   STA $D010
C08F A2 01      LDX #$01
C091 8E 00 D0   STX $D000
C094 4C 21 C0   JMP $C021
C097 FF         ???
C098 FF         ???
C099 FF         ???
C09A FF         ???
C09B FF         ???
C09C FF         ???
C09D FF         ???
C09E FF         ???
C09F FF         ???
C0A0 C0 00      CPY #$00
C0A2 D0 03      BNE $C0A7
C0A4 4C 21 C0   JMP $C021
C0A7 88         DEY
C0A8 20 00 C1   JSR $C100
C0AB 8C 01 D0   STY $D001
C0AE 4C 21 C0   JMP $C021
C0B1 00         BRK
C0B2 00         BRK
C0B3 C0 FF      CPY #$FF
C0B5 D0 03      BNE $C0BA
C0B7 4C 21 C0   JMP $C021
C0BA C8         INY
C0BB 20 00 C1   JSR $C100
C0BE 8C 01 D0   STY $D001
C0C1 4C 21 C0   JMP $C021
C0C4 FF         ???
C0C5 FF         ???
C0C6 FF         ???
C0C7 A9 1E      LDA #$1E
C0C9 8D 27 D0   STA $D027
```

```
C0CC 60          RTS
C0CD FF          ???
C0CE CF          ???
C0CF FF          ???
C0D0 FF          ???
C0D1 FF          ???
C0D2 FF          ???
C0D3 FF          ???
C0D4 FF          ???
C0D5 FF          ???
C0D6 FF          ???
C0D7 FF          ???
C0D8 FF          ???
C0D9 FF          ???
C0DA FF          ???
C0DB FF          ???
C0DC FF          ???
C0DD FF          ???
C0DE FF          ???
C0DF FF          ???
C0E0 80          ???
C0E1 00          BRK
C0E2 00          BRK
C0E3 00          BRK
C0E4 00          BRK
C0E5 00          BRK
C0E6 00          BRK
C0E7 00          BRK
C0E8 00          BRK
C0E9 00          BRK
C0EA 00          BRK
C0EB 00          BRK
C0EC 00          BRK
C0ED 00          BRK
C0EE 00          BRK
C0EF 00          BRK
C0F0 00          BRK
C0F1 00          BRK
C0F2 00          BRK
C0F3 00          BRK
C0F4 00          BRK
C0F5 00          BRK
C0F6 77          ???
C0F7 00          BRK
C0F8 00          BRK
C0F9 00          BRK
C0FA 00          BRK
C0FB 00          BRK
C0FC 00          BRK
C0FD 00          BRK
C0FE 00          BRK
```

```
C0FF 00          BRK
C100 8E 00 C2    STX $C200
C103 A2 FF       LDX #$FF
C105 CA          DEX
C106 D0 FD       BNE $C105
C108 AE 00 C2    LDX $C200
C10B 60          RTS
C10C BF          ???
.
.
```

When you've finished, check it carefully and save it to tape or disk with the command:

S "MOVING AROUND",01,C000,C10C

or

S "0:MOVING AROUND",08,C000,C10C

depending on which you're using.

Exit the disassembler and type SYS 49152 to get everything going. Fergus is controlled by the 'A' and 'D' keys to move him left and right, and 'I' and 'M' to move him up and down. Don't worry if you can't see him at first, as he starts off life off the screen. So far we're not worried about where he moves, but later on we'll confine him to the visible areas of the screen.

Now that you're totally baffled and lost, let's start unravelling a few mysteries in chapters 4 and 5.

4
Machine Code: First Instructions

Introduction

You'll have noticed by now, if you typed in and ran the program MOVING AROUND at the end of the last chapter, that our hero Fergus has a habit of sometimes racing to the left of the screen from way over on the right. Don't worry, this will be corrected in later chapters. It's a result of there being 320 possible sprite locations on the horizontal axis, and any memory location can only store a value up to 255. Consequently you need two bytes to store the required horizontal position of any sprite, and this early version of the program neglects to check that.

Values, bytes, 255, what is all this? Let's get back to basics (with a small b) and start explaining things in greater detail.

First lessons

It's important, in these early days of learning how to program in machine code, to get certain facts very, very clear indeed. A little time spent now going over what to some will be fairly simple stuff, will save a lot of time for all of us later on.

First of all, what is a byte? Put simply, a byte is the equivalent of one character, such as the letter A, and when we speak of a computer having X thousand bytes of useable memory, then that is just another way of saying that that computer is capable of storing X thousand characters in its memory. A byte is not the smallest amount of information that a computer can concern itself with, however. Bytes are split up (on the Commodore 64 anyway) into eight bits, with four bits combining to form a nibble. Each bit has a value associated with it, and we can talk in terms of the 2nd bit in the 4th byte of memory, or whatever.

To put it pictorially:

Bit No	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

If you add all those numbers up you'll see that they total 255, which is why we said earlier that the largest value that can be stored in any byte is 255: you just can't get in any number higher than that.

To alter the value stored in any particular byte, you're probably well aware of the commands POKE and PEEK: on the Commodore 64 you have to be.

If we, say, POKE 49152 with 169, what precisely are we doing? Look at the number 169, and try to figure out what set of values in the above combinations add up to 169, remembering of course that you can't have two bits set to equal 64: only one bit can be set to have that value in it, and that is the seventh one (cunningly referred to as bit 6 just to confuse things).

The number 169 is in fact made up of bits 7 (128), 5 (32), 3 (8) and 0 (1). So, the result of POKEing 49152 with 169 is to turn bits 7, 5, 3 and 0 on, and turn bits 6, 4, 2 and 1 off.

To turn bits 0, 3, 5 and 7 on without affecting the status of bits 1, 2, 4 and 6, which may already be on or off, we have to use the following statement:

```
POKE 49152,PEEK(49152)OR169
```

Okay, so that's bits and bytes sorted out, and we can already cope with hexadecimal and decimal, so let's take a look at some of the more commonly encountered commands in machine code.

Getting Started

To take the example of a simple BASIC program first of all, we'll add two numbers together: the numbers 4 and 6. In BASIC, this might be written as:

```
10 A=4
20 B=6
30 C=A+B
40 PRINTC
```

The machine code equivalent might go something like this:

```
C000 A9 04      LDA #$04
C002 A2 06      LDX #$06
C004 8E 00 04   STX $0400
C007 6D 00 04   ADC $0400
C00A 8D 02 04   STA $0402
C00D 60         RTS
```

What does all this mean? Line by line, we are:

Loading the accumulator with the hexadecimal number 4.

Loading the X register with the hexadecimal number 6.

Storing the X register at memory location 0400 (hexadecimal), or 1024 (decimal).

Adding to the accumulator (which contains the value 6) the contents of memory location 0400 (which now contains 4, because we've just put it there).

Storing the new value in the accumulator (4 + 6 = 10) at memory location 0402 (hexadecimal) or 1026 (decimal).

Returning from this subroutine.

Well, that makes a bit more sense, but probably not much. What is the accumulator, this mysterious X register that has cropped up from nowhere, why should storing something in a memory location be the same as printing the result of adding 4 and 6?

The accumulator is the heart of the 6510, the processor which looks after the Commodore 64, and it is the accumulator that does most of the work in all machine code programs. In itself, it is nothing more exciting than an 8 bit storage area, and as such it can store any number up to 255, as we've seen. But why put a four into it?

To revert back to a BASIC way of thinking for a while, we're all used to assigning values to variables in BASIC. A = 4, B = 6 and so on. In machine code there is no such thing as a variable in this sense. What we have to do is to use the accumulator, the X register and the Y register to store and retrieve numbers. These are the only three things that are capable of storing and retrieving numbers, and all three come into extensive use in most machine code programs.

Thus, the result of putting a 4 into the accumulator can be thought of as being reasonably equivalent to assigning the value of 4 to a variable. Similarly, putting a 6 into the X register (another 8 bit storage area) is similar to assigning the value of 6 to another variable. Finally, if we'd wanted to, we could have assigned a value to the Y register (our third and final 8 bit storage area), which would again have been similar to giving a BASIC variable a value.

With only three 'variables' to play with, you might well imagine that life can get pretty hairy at times, and you are so, so right. Fortunately, since machine code programs operate at amazingly fast speeds, this apparent limitation doesn't really worry us, as long as we remember where everything is stored.

Storing values in memory

To backtrack to the little machine code program given earlier, you'll see that we stored the value in the X register at a certain memory location. In machine code you put values into registers and store them all over the place (remembering of course where you've put them), retrieving them when the need arises. On the 64 memory location 0400 happens to be on the screen, and storing the content of the X register at location 0400 is equivalent to POKEing a number (whatever happens to be in the X register) into location 0400 in hexadecimal, or 1024 in decimal. This, of course, is the start of screen memory, since the screen is just another set of memory locations like everything else.

As we're dealing with the 64, life is never as easy as we'd like it to be. Just as in BASIC POKE1024,6 doesn't achieve very much, neither does storing the X register at location 0400. This is because we also need to put some colour there so we can see what's happening. In BASIC, you might POKE 55296,0 to get a black character appearing at the top of the screen (a black letter D if we've got a four there). In machine code, we'd have to:

```
C00D  A2 01     LDX #$01
C00F  8E 00 D8  STX $D800
C012  8E 02 D8  STX $D802
C015  60        RTS
```

In other words, load the X register with a 1, and store it at locations D800 and D802, the hexadecimal equivalents of colour memory locations 55296 and 55928. You'll note our RTS has been moved down a bit to accomodate these new instructions. Without having a RTS

at the end of a routine, unlike BASIC which grinds to a halt when there are no more statements to execute, machine code merrily trundles on through memory until it finds something to do: and that something may not be very pleasant for the machine. You won't damage the computer, but you may well cause it to crash, losing the program that you've so lovingly typed in.

So loading a register (or the accumulator) is the equivalent of defining a variable, and storing that register somewhere in memory is the equivalent of POKEing a variable into memory.

Incidentally, to run that little program given earlier you can type the command SYS 49152, which transfers program execution to location 49152, and only returns to BASIC Ready mode when it encounters an RTS (or something that causes the machine to throw a wobbler. A technical term, that one).

Some formal definitions

We'll give some formal definitions of the commands used so far:

STA

STore the contents of the Accumulator at the memory location specified.

LDA

LoaD the Accumulator with the numeric value specified.

LDX

LoaD the X register with the numeric value specified.

STX

STore the contents of the X register at the memory location specified.

ADC

ADd to the accumulator with Carry the contents of the specified memory location.

Don't worry about the 'with Carry' part. We'll be hearing a lot more about that later.

All the definitions given above are basic ones, as most of those commands have variations on the way that they've been used so far. That is, going back to our 'foreign language' analogy given earlier, we've taught you one tense of a verb, but there are others that we haven't looked at yet. We'll take care of those variations as and when we get to them.

More simple programs

To accustom you to entering simple machine code programs, and help you think in machine code rather than in BASIC, here are a couple of simple programs.

One popular program places a heart (or diamond, or whatever) on the screen. Not very exciting perhaps, but it is something. What you're not usually told is why the program works. Why should I put a 65 into the accumulator? Read on...

```
C000 A9 41     LDA #$41
C002 8D 00 04  STA $0400
C005 A2 00     LDX #$00
C007 8E 00 D8  STX $D800
C00A 60        RTS
```

Here we're putting the value 65 into the accumulator. Why? If you look at the manual that accompanies the 64, and turn to page 133, you'll see a set of screen display codes. The code for a heart is 65, so if, in BASIC, you typed:

POKE 1024,65

A heart would appear in the top left hand corner of the screen (if you could see it of course). That POKE command is the equivalent of the first 5 bytes of that little machine code program above. Load the accumulator with the code for a heart, and store it at memory location 0400 hexadecimal, or 1024 decimal.

Now, what about the next part? Loading the X register with a 0 (we might just as easily have loaded the accumulator with a 0 instead), and storing it at location DB00, is the equivalent of:

POKE 55296,0

which turns our little heart into a black heart.

Finally, we return from the subroutine and end up in BASIC Ready mode again.

An example like this doesn't really show any great advantage over BASIC, so we'll print out 5 rows of the things (200 hearts in all) by introducing a few new machine code statements. But first, a possible BASIC equivalent.

```
10 FORI=0TO199
20 POKE1024+I,65
30 POKE55296+I,0
40 NEXTI
```

This will put 200 little black hearts at the top of the screen. In machine code, this might be written as:

```
C000 A2 C8      LDX #$C8
C002 A9 41      LDA #$41
C004 9D FF 03   STA ($03FF),X
C007 A9 00      LDA #$00
C009 9D FF D7   STA ($D7FF),X
C00C CA         DEX
C00D D0 F3      BNE 02 C0
C00F 60         60
```

There are a few new commands here. Before considering them in detail, we'll explain what the program is doing in plain English.

First of all, put a value of C8 hexadecimal, or 200 decimal, into the X register.

Put a value of 41 hexadecimal, or 65 decimal, into the accumulator.

Store it at memory location 03FF offset with the X register. In other words (working in decimal for a while), the first time around the loop the X register contains 200, so we're going to store the contents of the accumulator at location 1023 offset with 200, or 1223.

Put a value of zero into the accumulator.

Store it at memory location D7FF offset with the X register.

Decrease the content of the X register by 1. That is, the first time around decrease it to 199, then 198, and so on.

If the result of doing that isn't equal to zero (i.e. the X register doesn't contain a zero), then branch back to location C002 and load the accumulator with a 41, or decimal 65 again and repeat as before.

We finally get to when the X register does contain a zero and program execution comes to a halt. So, plenty of new commands, and if you run this program with a SYS 49152, you'll be amazed at the difference between the BASIC version and this machine code one. You can see BASIC performing, but this just appears to happen instantaneously.

One final point may be puzzling you. Why are numbers seemingly reversed when entered into the disassembler? For example, if you look at the line starting at memory location C004, why do we have FF 03 in one part, and 03FF in the mnemonic part? As you know, a number greater than 255 cannot be stored in one single byte, and so when we're talking about numbers as large as 03FF hexadecimal, or 1023 decimal, this number has to be spread over two bytes.

Later on we'll be talking about something called the Stack, a place for storing useful information. The Stack, like seemingly everything else in the computer world, stores numbers on a 'last-in, first out' basis. That is, the number it was last told to remember is the first number that it will retrieve. Like a stack of plates, the last one to be put on top of the pile is the first one to be taken off it. So the first number the computer sees when it's looking at two bytes storing a large number is the last one that was put in there: in this example, the 03. Then it sees the next one, the FF, and remembers the number as 03FF.

New commands explained

What we've seen in that last program are the commands BNE, DEX and STA offset with X. In order then:

BNE

Branch if the result of the previous instruction does not yield a value of zero. In other words, is Not Equal to zero.

DEX

DEcrement the content of the X register by one.

STA offset

STore the content of the Accumulator at a specified memory location, offset with the contents of the X (or Y) register.

We're already beginning to see some variations on the earlier command theme that we gave you (STA offset is a very different animal to STA), and there'll be plenty more to come.

For a little bit of light (light?) diversion, try comparing the following listing with our original MOVING AROUND one. Type in the differences (there's a number hidden away in there, including a new routine to fire the missile when the RETURN key is pressed), and run the program with a SYS 49152 command.

Don't forget to load your sprite data in first, and remember to save this new version of the program before you run it. One mistake could cost you a lot of time and trouble.

```
B*
   PC  SR AC XR YR SP
.;7FC5 33 00 AD 00 F6
.
C000 A9 0D      LDA #$0D
C002 8D F8 07   STA $07F8
C005 A9 03      LDA #$03
C007 8D 15 D0   STA $D015
C00A A9 0E      LDA #$0E
C00C 8D F9 07   STA $07F9
C00F 20 00 C3   JSR $C300
C012 A9 00      LDA #$00
C014 8D 17 D0   STA $D017
C017 8D 1D D0   STA $D01D
C01A A2 00      LDX #$00
C01C 8E 10 D0   STX $D010
C01F A0 35      LDY #$35
```

```
C021 AD C5 00    LDA $00C5
C024 C9 0A       CMP #$0A
C026 F0 1F       BEQ $C047
C028 C9 12       CMP #$12
C02A F0 45       BEQ $C071
C02C C9 21       CMP #$21
C02E F0 70       BEQ $C0A0
C030 C9 24       CMP #$24
C032 F0 7F       BEQ $C0B3
C034 C9 01       CMP #$01
C036 F0 03       BEQ $C03B
C038 4C 21 C0    JMP $C021
C03B 20 C7 C0    JSR $C0C7
C03E 4C 21 C0    JMP $C021
C041 FF          ???
C042 FF          ???
C043 FF          ???
C044 FF          ???
C045 FF          ???
C046 FF          ???
C047 E0 00       CPX #$00
C049 D0 03       BNE $C04E
C04B 4C 21 C0    JMP $C021
C04E CA          DEX
C04F 20 00 C1    JSR $C100
C052 8E 00 D0    STX $D000
C055 EA          NOP
C056 EA          NOP
C057 EA          NOP
C058 E0 00       CPX #$00
C05A F0 03       BEQ $C05F
C05C 4C 21 C0    JMP $C021
C05F EC 10 D0    CPX $D010
C062 F0 F8       BEQ $C05C
C064 A2 FF       LDX #$FF
C066 A9 00       LDA #$00
C068 8D 10 D0    STA $D010
C06B 8E 00 D0    STX $D000
C06E 4C 21 C0    JMP $C021
C071 E0 FF       CPX #$FF
C073 D0 03       BNE $C078
C075 4C 21 C0    JMP $C021
C078 E8          INX
C079 20 00 C1    JSR $C100
C07C 8E 00 D0    STX $D000
C07F E0 FF       CPX #$FF
C081 D0 F2       BNE $C075
C083 A9 00       LDA #$00
C085 CD 10 D0    CMP $D010
C088 D0 EB       BNE $C075
C08A A9 03       LDA #$03
```

```
C08C 8D 10 D0    STA $D010
C08F A2 01       LDX #$01
C091 8E 00 D0    STX $D000
C094 4C 21 C0    JMP $C021
C097 FF          ???
C098 FF          ???
C099 FF          ???
C09A FF          ???
C09B FF          ???
C09C FF          ???
C09D FF          ???
C09E FF          ???
C09F FF          ???
C0A0 C0 35       CPY #$35
C0A2 D0 03       BNE $C0A7
C0A4 4C 21 C0    JMP $C021
C0A7 88          DEY
C0A8 20 00 C1    JSR $C100
C0AB 8C 01 D0    STY $D001
C0AE 4C 21 C0    JMP $C021
C0B1 00          BRK
C0B2 00          BRK
C0B3 C0 E0       CPY #$E0
C0B5 D0 03       BNE $C0BA
C0B7 4C 21 C0    JMP $C021
C0BA C8          INY
C0BB 20 00 C1    JSR $C100
C0BE 8C 01 D0    STY $D001
C0C1 4C 21 C0    JMP $C021
C0C4 FF          ???
C0C5 FF          ???
C0C6 FF          ???
C0C7 A9 07       LDA #$07
C0C9 8D 28 D0    STA $D028
C0CC 8C 01 D2    STY $D201
C0CF 8E 02 D0    STX $D002
C0D2 8C 03 D0    STY $D003
C0D5 C0 10       CPY #$10
C0D7 F0 0A       BEQ $C0E3
C0D9 88          DEY
C0DA 20 00 C1    JSR $C100
C0DD 8C 03 D0    STY $D003
C0E0 4C D5 C0    JMP $C0D5
C0E3 AC 01 D2    LDY $D201
C0E6 60          RTS
C0E7 00          BRK
C0E8 00          BRK
C0E9 00          BRK
C0EA 00          BRK
C0EB 00          BRK
C0EC 00          BRK
```

```
C0ED 00          BRK
C0EE 00          BRK
C0EF 00          BRK
C0F0 00          BRK
C0F1 00          BRK
C0F2 00          BRK
C0F3 00          BRK
C0F4 00          BRK
C0F5 00          BRK
C0F6 77          ???
C0F7 00          BRK
C0F8 00          BRK
C0F9 00          BRK
C0FA 00          BRK
C0FB 00          BRK
C0FC 00          BRK
C0FD 00          BRK
C0FE 00          BRK
C0FF 00          BRK
C100 8E 00 C2    STX $C200
C103 A2 FF       LDX #$FF
C105 CA          DEX
C106 D0 FD       BNE $C105
C108 20 00 C3    JSR $C300
C10B AE 00 C2    LDX $C200
C10E 8C 01 D4    STY $D401
C111 60          RTS
.


.
C300 A9 0F       LDA #$0F
C302 8D 18 D4    STA $D418
C305 A9 22       LDA #$22
C307 8D 05 D4    STA $D405
C30A A9 86       LDA #$86
C30C 8D 06 D4    STA $D406
C30F A9 81       LDA #$81
C311 8D 04 D4    STA $D404
C314 A9 00       LDA #$00
C316 8D 20 D0    STA $D020
C319 8D 21 D0    STA $D021
C31C A9 93       LDA #$93
C31E 8D 77 02    STA $0277
C321 A9 01       LDA #$01
C323 8D 00 C6    STA $C600
C326 60          RTS
C327 00          BRK
.
.
```

Let's take a look, using the commands we already know, along with a couple of new ones, at how some of this program manages to work in the way that it does.

MOVING AROUND revisited

Look at the listing from memory locations C000 to C01F first of all. Taking these in turn, this is what is happening.

Load the accumulator with the hexadecimal number 0D, or decimal number 13.

Store that number in memory location 2040. This is equivalent to POKE 2040,13, and tells the computer that the data for the first sprite is stored in the 13th block of memory.

Load the accumulator with a 3, and store that in location D015 hexadecimal, or 53269 decimal. In other words, POKE53269,3: turn on the first two sprites.

Load the accumulator with a 14, and store it in memory location 2041 (decimal 07F9). That is, POKE 2041,14: tell the computer that the data for the second sprite is stored in the 14th block of memory.

JSR $C300 means Jump to the SubRoutine starting at memory location C300. Essentially it acts like a GOSUB in BASIC, but we'll come back to this one later.

Load the accumulator with a zero, and then store it at locations D017 and D01D hexadecimal, or 53271 and 53277 decimal. Equivalent to POKE 53271,0:POKE53277,0. This turns off the horizontal and vertical expansion for all sprites, so that we don't get any strange displays appearing on the screen.

Load the X register with a zero and store it in location D010 hexadecimal, or 53264 decimal. Equivalent to POKE 53264,0: switch off the location that determines whether the horizontal location of the sprites is beyond location 255 or not.

Finally, load the Y register with 35 hexadecimal, or 53 decimal. This will determine the start location on the vertical for our friend Fergus.

So to set everything up isn't too complicated. The routine that controls most of the game is found in locations C021 through to C040,

so we'll go through that before leaving this program and going back to some of the earlier material we showed you: the border program and the million count program.

Main program routine

The very first line loads the accumulator with the content of memory location C5 hexadecimal, or 197 decimal. This, if you like, is the equivalent of PEEKing at location 197 to see which key has been pressed.

Then we CoMPare the value in the accumulator with the hexadecimal number 0A: 10 in decimal. This is the value returned if the 'A' key has been pressed, and if the two values are equal (that in the accumulator and the value 10) then we branch to memory location C047.

The program then checks in turn for the 'D' key, the up and down movements, and finally to see whether a missile has been fired or not.

If none of these events have taken place, go back to location C021 again to wait until something does happen. Needless to say, on later versions of this program we won't be just sitting around waiting for the player to press something: the aliens will start having some ideas of their own.

You're probably by now in a reasonable enough position to take a look through the rest of the listing and try to figure out what it's all doing. So far there's been nothing complicated, but to clear up on the three new commands we've met in the last couple of pages:

JSR

Jump to the SubRoutine at the specified memory location.

CMP

CoMPare the contents of the accumulator with a specified number. We can also compare them with the contents of a specified memory location, amongst other things.

BEQ

Branch if the result of the previous instruction is EQual to zero: similar in operation to the BNE instruction met earlier. That one was, if you remember, branch if the result of the previous instruction does not give a value of zero.

How do we decide how far to branch? Obviously that will depend on where you want program execution to continue, but you should note that with all these branching commands there is a limit to how far we can go, and that limit is either 127 memory locations further on in the program or 128 memory locations further back in the program.

Sticking with decimal numbers for a while, if we could have a command such as:

BEQ 30

program execution would jump forward 30 bytes if the result of an operation was zero.

If we had something like:

BEQ 210

program execution would jump back (256-210) 46 bytes if the result of an operation was zero.

We'll now go back to those earlier programs to examine how they worked, starting with the border one, so get the assembler in there, load the border program (which you did save, didn't you?), and let's take a proper look through the listing.

Out on the border

As you can see, this relies quite heavily on the STA offset feature. This apparent inconvenience of having to type out endless lists of STA instructions doesn't really matter, since machine code whizzes along at a fair old rate of knots. The equivalent performance in BASIC would take an eternity ... well, almost.

Starting with the very first line of the program, you can see that we load the accumulator with a value of 66 hexadecimal, or 102 decimal. This determines not only what character is going to form our border,

but also what colour that character will be displayed in, since we use the same value for both things.

Then, load the X register with a 27 hexadecimal, or 39 decimal (one less than the screen width), and the Y register with a zero. Looking back on the program I can't for the life of me remember why I did that, but I'm sure there must have been a reason at the time! It certainly doesn't affect this program.

Then we store the value in the accumulator at memory location 0400 (the start of the screen) offset with the value in X. Since this is 27 hexadecimal to start with, that's where the value 66 goes: at 0427. The equivalent of POKE 1063,102. We similarly put the value of 66 into the colour memory for that screen location.

The next two lines do the same for the bottom row of the screen, before decrementing the X register and seeing if we've reached a value of zero yet. If we haven't, then trot back to location C006 and go through the whole performance again, remembering that this time around the X register will have the value 26 hexadecimal in it, and so we now alter screen location 0426 and colour memory location DB26.

This continues until the X register finally has a zero in it, when we colour in the top left corner of the screen, and the leftmost character on the bottom line of the screen. This has to be done since, when the the contents of the X register reached zero, we never went round the loop again to store the accumulator at 0400 offset with zero, i.e. 0400.

Now for the complicated part. The X register is loaded with a value of F0 hexadecimal, or 240 decimal. This is the equivalent of six lines of the screen (40 columns per line), and allows us to put four characters down the left hand side of the screen and four down the right, remembering to put some colour there as well of course.

Then we temporarily store the value held in the accumulator at memory location C100: just somewhere to store the value where it won't come to any harm. We then transfer the contents of the X register to the accumulator, because although we can perform mathematical operations on the accumulator, we can't on the X register, and we want to do a bit of subtraction. The SEC command simply tells the computer that there's a bit of subtraction coming up, and the next line (SBC #$28) subtracts the hexadecimal value of 28, or the decimal value of 40, from the value in the accumulator. The new value in the accumulator is now transferred back to the X register, and the old value

of the accumulator (102, from way back at the start of the program) is picked up from where it was dropped off in memory location C100: its temporary storage position.

The value in the X register is now compared with 0, and if the result of this operation doesn't give us a zero we branch back to memory location C023 and start this whole performance again. Remember that now the X register contains a value 40 less than it previously did, and so our STA offset command affects one line further up the screen.

This continues until we've filled in all the edges of the screen, and the program comes to a halt with the RTS command at location C062.

To go through the four new commands encountered in this program:

TXA

Transfer the contents of the X register to the contents of the Accumulator, leaving the contents of the X register unaffected.

SEC

SEt the Carry flag. In other words, tell the computer that there's some mathematics coming up which involves subtraction. All the flags will be covered in the next chapter, so don't worry about them just yet.

SBC

SuBtract from the accumulator with Carry the specified number, or we can also subtract from the accumulator the value stored in a specified memory location. Again, this mysterious word Carry will be explained in the next chapter.

TAX

Transfer the contents of the Accumulator to the X register, leaving the contents of the accumulator unaffected.

Well, that's one reasonably complicated program explained in great detail, so we might as well get the other one out of the way now, and go step by step through the Million Count machine code program.

Again, it'll be a great help if you can get this one up on the screen and follow it through as we explain what's happening.

If I had a million

Again, we'll go through this one instruction at a time, so that you can get a clear understanding of what is happening. To start, the first two instructions are reasonably straightforward: load the X register with the value of 5, and the accumulator with a value of 30 hexadecimal, or 48 decimal. If you turn to the manual supplied with the 64, and find page 133 again, you'll find that decimal 48 is the value associated with the numeric figure 0.

Then we store our accumulator value (the figure 0) at memory location 0400 offset with the value of the X register. Decrement the X register, and check to see that it is either positive or zero. If it is, go back to memory location C004 and store the accumulator at 0400 offset with the new value of X. This continues until X becomes negative and we then have 6 zeroes up on the screen. The program has begun.

The X register is re-loaded with 5 again, and the accumulator in the next line uses the LDA offset command. Analogous to the STA offset, only this time we're receiving a value, not placing one. CLC stands for CLear the Carry flag, and tells the computer that there's going to be some addition going on in the very near future.

We then add 1 to the value stored in the accumulator, and compare that with 3A hexadecimal. This is one way of checking to see if the accumulator is displaying the code for the number 9 (hexadecimal 39) or has gone over that limit. This is checked for in the next line, because if the accumulator does contain 3A program execution continues at memory location C01C. If it doesn't, then we store this updated value on the screen and JuMP back to memory location C00A.

Now, if the accumulator contains a value greater than nine, we need to switch this back to zero, and update and check the next digit on the left. LDA # $30 at location C01C puts the numeric code for zero into the accumulator, and stores it on the screen. The value in the X register is then decreased by one, and if the result of doing this is negative (Branch on PLus checks for a number being either positive or equal to zero: all will be revealed in the section on flags in the next chapter) then we've reached a million and the program ends.

If, however, the X register still contains a positive number or zero we

branch to memory location C025 and carry on. This loads the accumulator with the content of memory location 0400 offset with X, but remember we have now decremented X and so are looking at the character to the left of the one that we've just set to zero.

Another CLC command heralds a further bit of addition coming up, and one is added to the content of the accumulator. Then program execution jumps to location C012 to start the whole series of checks off all over again.

As with the Border program, we'll give you some formal definitions of the new instructions encountered in this program.

BPL

Branch if the result of an operation is either positive or zero. As long as it isn't negative, that's fine by us. It actually stands for Branch on PLus: for once, a confusing mnemonic.

CLC

CLear the Carry flag. In other words, prepare for some addition.

ADC

ADd with Carry the contents of the accumulator to the number specified. This can also be used to add the contents of the accumulator to the contents of a specified memory location.

JMP

JuMP to a specified memory location. Equivalent to GOTO in BASIC really, and it's as good an idea to try to avoid it in machine code as it is in BASIC. This is because we will often want to change a couple of bytes of a program, which will involve juggling a few things around. If the location that we're jumping to happens to move, the machine code program will not change where we've specified to jump to, and program execution will undoubtedly end up in no-man's land.

LDA offset

LoaD the Accumulator with the content of a specified memory location, offset with the value of the X register, or Y register for that matter.

In the next chapter, we'll move on to various flags and processor architecture, but for now here's the latest update to our Fergus program, in which we've added some better sound, and also managed to confine the little blighter to the contents of the visible screen. The annoying occasional jump when moving left has also been cured in this one.

See if you can manage to figure out what's happening. You should be getting used to all this by now!

```
B*
   PC  SR AC XR YR SP
.;5766 33 00 4E 00 F6
.
C000 A9 0D      LDA #$0D
C002 8D F8 07   STA $07F8
C005 A9 03      LDA #$03
C007 8D 15 D0   STA $D015
C00A A9 0E      LDA #$0E
C00C 8D F9 07   STA $07F9
C00F 20 00 C3   JSR $C300
C012 A9 00      LDA #$00
C014 8D 17 D0   STA $D017
C017 8D 1D D0   STA $D01D
C01A A9 00      LDA #$00
C01C 8D 10 D0   STA $D010
C01F EA         NOP
C020 EA         NOP
C021 AD C5 00   LDA $00C5
C024 C9 0A      CMP #$0A
C026 F0 1F      BEQ $C047
C028 C9 12      CMP #$12
C02A F0 45      BEQ $C071
C02C C9 21      CMP #$21
C02E F0 70      BEQ $C0A0
C030 C9 24      CMP #$24
C032 F0 7F      BEQ $C0B3
C034 C9 3C      CMP #$3C
C036 F0 03      BEQ $C03B
C038 4C 21 C0   JMP $C021
C03B 4C C7 C0   JMP $C0C7
C03E 4C 21 C0   JMP $C021
```

```
C041 FF          ???
C042 FF          ???
C043 FF          ???
C044 FF          ???
C045 FF          ???
C046 FF          ???
C047 A9 00       LDA #$00
C049 CD 10 D0    CMP $D010
C04C D0 0E       BNE $C05C
C04E E0 18       CPX #$18
C050 F0 07       BEQ $C059
C052 CA          DEX
C053 20 00 C1    JSR $C100
C056 8E 00 D0    STX $D000
C059 4C 21 C0    JMP $C021
C05C E0 00       CPX #$00
C05E F0 03       BEQ $C063
C060 4C 52 C0    JMP $C052
C063 A9 00       LDA #$00
C065 8D 10 D0    STA $D010
C068 A2 FF       LDX #$FF
C06A 8E 00 D0    STX $D000
C06D 4C 21 C0    JMP $C021
C070 00          BRK
C071 E0 FF       CPX #$FF
C073 D0 03       BNE $C078
C075 4C 21 C0    JMP $C021
C078 E8          INX
C079 20 00 C1    JSR $C100
C07C 8E 00 D0    STX $D000
C07F A9 00       LDA #$00
C081 CD 10 D0    CMP $D010
C084 D0 11       BNE $C097
C086 E0 FF       CPX #$FF
C088 D0 EB       BNE $C075
C08A A9 03       LDA #$03
C08C 8D 10 D0    STA $D010
C08F A2 01       LDX #$01
C091 8E 00 D0    STX $D000
C094 4C 21 C0    JMP $C021
C097 E0 40       CPX #$40
C099 D0 DA       BNE $C075
C09B CA          DEX
C09C 4C 21 C0    JMP $C021
C09F FF          ???
C0A0 C0 32       CPY #$32
C0A2 D0 03       BNE $C0A7
C0A4 4C 21 C0    JMP $C021
C0A7 88          DEY
C0A8 20 00 C1    JSR $C100
C0AB 8C 01 D0    STY $D001
```

```
C0AE 4C 21 C0    JMP $C021
C0B1 00          BRK
C0B2 00          BRK
C0B3 C0 E5       CPY #$E5
C0B5 D0 03       BNE $C0BA
C0B7 4C 21 C0    JMP $C021
C0BA C8          INY
C0BB 20 00 C1    JSR $C100
C0BE 8C 01 D0    STY $D001
C0C1 4C 21 C0    JMP $C021
C0C4 FF          ???
C0C5 FF          ???
C0C6 FF          ???
C0C7 A9 07       LDA #$07
C0C9 8D 28 D0    STA $D028
C0CC A9 21       LDA #$21
C0CE EA          NOP
C0CF 8D 04 D4    STA $D404
C0D2 8C 02 C2    STY $C202
C0D5 8E 02 D0    STX $D002
C0D8 8C 03 D0    STY $D003
C0DB C0 10       CPY #$10
C0DD F0 0A       BEQ $C0E9
C0DF 88          DEY
C0E0 20 00 C1    JSR $C100
C0E3 8C 03 D0    STY $D003
C0E6 4C DB C0    JMP $C0DB
C0E9 AC 02 C2    LDY $C202
C0EC A9 81       LDA #$81
C0EE 8D 04 D4    STA $D404
C0F1 4C 21 C0    JMP $C021
C0F4 00          BRK
C0F5 00          BRK
C0F6 77          ???
C0F7 00          BRK
C0F8 00          BRK
C0F9 00          BRK
C0FA 00          BRK
C0FB 00          BRK
C0FC 00          BRK
C0FD 00          BRK
C0FE 00          BRK
C0FF 00          BRK
C100 8E 00 C2    STX $C200
C103 A2 FF       LDX #$FF
C105 CA          DEX
C106 D0 FD       BNE $C105
C108 EA          NOP
C109 EA          NOP
C10A EA          NOP
C10B 8C 01 D4    STY $D401
```

```
C10E EA          NOP
C10F EA          NOP
C110 EA          NOP
C111 AE 00 C2    LDX $C200
C114 EA          NOP
C115 EA          NOP
C116 EA          NOP
C117 60          RTS
C118 FF          ???
C119 FF          ???
C11A FF          ???
C11B FF          ???
C11C FF          ???
C11D FF          ???
C11E FF          ???
C11F FF          ???
C120 00          BRK
C121 00          BRK
C122 00          BRK
C123 00          BRK
C124 00          BRK
C125 00          BRK
C126 00          BRK
C127 00          BRK
.
.
C300 A9 0F       LDA #$0F
C302 8D 18 D4    STA $D418
C305 A9 22       LDA #$22
C307 8D 05 D4    STA $D405
C30A A9 86       LDA #$86
C30C 8D 06 D4    STA $D406
C30F A9 81       LDA #$81
C311 8D 04 D4    STA $D404
C314 A9 00       LDA #$00
C316 8D 20 D0    STA $D020
C319 8D 21 D0    STA $D021
C31C A2 18       LDX #$18
C31E 8E 00 D0    STX $D000
C321 A4 32       LDY $32
C323 8C 01 D0    STY $D001
C326 60          RTS
C327 00          BRK
.
.
```

5
Machine Code: Flags and Registers and Other Wonders

Introduction

We mentioned in the last chapter such things as STA offset, an alternative version of the STA command, instructions such as ADC, ADd with Carry, setting and clearing the carry flag, and offering dark hints about registers such as the X and Y register. Before we encounter the horrors of double precision arithmetic and using built-in subroutines in the 64's memory map in later chapters, it's about time to get formal again and explain what all these things mean.

To start, we'll take a look at the way the 6510 microprocessor really executes a program.

How the 6510 executes a program

Rather like a program written in BASIC, the 6510 operates quite simply by fetching an instruction from memory, acting on and executing that instruction, and then going to fetch another one. But how does it know which instruction is the next one? A special register is set aside to control all this, and this register is known as the program counter. It is this register that informs the 6510 which instruction it has to execute next.

Fortunately for us, this program counter is automatically incremented after every instruction, and as a programmer you don't have to worry about updating it yourself. This program counter register is actually two bytes stuck together (rather than the one byte of the accumulator, the X register, and the Y register), and as such it comprises 16 bits. This enables it to look at any location within the 64K of memory that the Commodore 64 contains.

You may remember our earlier dissection of a byte, which looked like this:

Bit number	7	6	5	4	3	2	1	0
Value	128	64	32	16	8	4	2	1

Well, the program counter, being two bytes long, looks something like this:

Bit no.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1

Adding all those values up gives us a grand total of 65535, which is the largest number that the program counter can accomodate. Since 64K is the equivalent of (64 times 1024) bytes, or 65536 bytes, the program counter can readily access any byte within those 65536 (0 to 65535 being 65536 numbers).

As you've seen from the listings given earlier, machine code instructions can consist of one, two or three bytes. The first one is always the machine code equivalent of the operation code (known for brevity as the op-code), as a glance at the table of machine code instructions at the back of the book will reveal. This byte is directed off to the instruction register, to find out what it means, and then routed to something known as the instruction decode logic. This sends out appropriate signals to all the other elements of the microprocessor, warning them all that something is about to happen.

The second byte, and third if appropriate, are then pushed off into the data bus buffer, and from there sent either to the arithmetic logic unit (known as the ALU) if they merely contain data, or to our old friend the program counter if they contain the address of a memory location.

Now aren't you glad that all this sort of stuff is handled automatically, and you don't have to worry about it?

More registers

We've already had extensive experience of looking at the X and Y registers, and the accumulator. Since the accumulator is the only register on which arithmetical and logical operations can be performed, it is the most powerful register within the 6510. It is also the only one on which logical operations (such as AND and OR: they're coming

up in chapter six) can be carried out. The program counter is another register that we've discussed.

But there are other registers, and the first one that we'll come to is known as the Processor Status Register.

Processor Status Register

This is, as usual, an 8 bit register, although we can in fact only get useful information from seven of these bits. In diagrammatic form, the register looks like this:

Bit	7	6	5	4	3	2	1	0
Flag	N	V	-	B	D	I	Z	C

Flags are something that we encountered in chapter 4, but never explained properly. The status of these flags is tested by a number of machine code instructions, and some of them we've already met: BNE, BPL, SEC and so on. We'll go through each of these flags in turn, starting with one that we've already met.

The Carry flag (C)

This is the one handled by byte zero of the status register.

It really comes into play when we want to deal with numbers that are greater than 255. As we've already seen, 255 is the largest number that can be stored in any one byte and if, as with the program counter, we want to handle numbers that are larger than that, we have to use two bytes that are next to each other.

To link two bytes together there has to be something that joins the first one to the second, and this is the purpose of the carry flag. Think of it as a number carrying over from one byte to the next, if you like. Rather like the way you did addition at primary school: if a number is bigger than 10, say 18, then you put the 8 down and carry 1.

There are four important instructions for using and acting on the carry flag, and these are:

BCC: Branch on Carry Clear.

This instructs the program to branch off somewhere else if the carry

flag is clear, i.e. it isn't set, and the value stored in it is currently zero. To make sure that this doesn't happen accidentally, but only when we want it to, the next instruction clears the carry flag before we perform any mathematical calculations.

CLC: CLear the Carry flag.

This sets the carry flag to zero, and thus ensures that nothing untoward happens to our programs.

BCS: Branch on Carry Set.

The opposite of BCC. If the carry flag is not clear, i.e. it has been set and the value currently stored in it is a one, then branch to whatever part of the program we wish to go to. Again, there is an analogous command to CLC to make sure that this doesn't happen accidentally, although this is usually only used when we want to perform some subtraction.

SEC: SEt the Carry flag.

This sets the carry flag to one, i.e. it is no longer clear.

To illustrate this, here is a short program:

```
.; C000 18       CLC
.; C001 A9 01    LDA #$01
.; C003 69 01    ADC #$01
.; C005 B0 03    BCS $C00A
.; C007 4C 03 C0 JMP $C003
.; C00A 8D 00 04 STA $0400
.; C00D 8D 00 D8 STA $D800
.; C010 60       RTS
```

This first of all clears the carry flag, then loads the accumulator with the number one. The next instruction adds one to the value currently in the accumulator and then checks to see if a carry has been set. That is, has the number in the accumulator exceeded 255 and flipped around to 0 again? If it has, we go off to C00A and perform the instructions there, but if it hasn't we go back to C003 and add another one to the accumulator.

The instructions STA $0400 and STA $D800 then store the content of the accumulator (a zero now, since it's flipped over from 255 to zero as the carry flag was set) in the top left hand corner of the screen.

The result is that an '@' sign appears.

The three other flags in the status register that are affected by numerical operations are the zero flag, the overflow flag, and the negative flag:

The Zero flag (Z)

This is the second bit of the status register, and it is set if the result of a mathematical operation is zero. It is this flag that is being looked at in the BNE and BEQ commands we covered earlier. If the result of an operation is equal to zero, then the zero flag is set and a BEQ command, seeing that the flag is set, would then send program execution off somewhere else.

The Overflow flag (V)

This is the seventh bit of the status register, and is set when two positive or negative numbers are added together, and the result exceeds either +$7F or -$80. Hexadecimal numbers, to avoid writing the word out thousands of times, are usually prefixed with a dollar sign, as in the above example. This convention we will use throughout the rest of the book.

The Negative flag (N)

The 8th bit of the status register, this is set when two signed numbers, if added together, give a negative result. If they don't, it isn't set.

The three flags that we haven't looked at yet are the B, D and I flags, or respectively the BRK command, Decimal mode, and IRQ disable flags. Briefly, the first one indicates whether an interrupt request to the 6510 was caused by a 'break' instruction (op-code 00) or by some externally generated interrupt, the second one determines whether the ALU will operate in binary mode or decimal mode, and the final one shuts out all interrupts to the 6510, should you decide not to let anything interrupt you for a while.

From flags and registers, we'll move on to another important concept, which you must grasp before you can start to use machine code at all seriously. This is concerned with what are termed modes of addressing. We said earlier that many of the commands that we've used (LDA, STX and so on) have variations on a main theme, and these

variations are governed by which addressing mode we are currently operating in.

Modes of addressing

In all, there are some thirteen different modes of addressing, and these can be summarised as:

MODE	OPERAND FORMAT
Immediate	#aa
Absolute	aaaa
Zero Page	aa
Implied	
Indirect absolute	(aaaa)
Absolute indexed,X	aaaa,X
Absolute indexed,Y	aaaa,Y
Zero page indexed,X	aa,X
Zero page indexed,Y	aa,Y
Indexed indirect	(aa,X)
Indirect indexed	(aa),Y
Relative	aa or aaaa
Accumulator	A

In the table, the letter 'a' represents a hexadecimal digit, so aaaa is a four digit hexadecimal number, such as 4F2C. X and Y refer to the X and Y registers.

We'll give a brief rundown on each of these modes over the next few pages, but since the understanding of some of them (in particular indexed indirect and indirect indexed) is really beyond the preliminary stages that we've reached so far, we'll be coming back to the more complicated ones later.

Immediate Addressing

This form of addressing allows you to specify a single byte constant as the operand (i.e. the thing to be operated with). Thus any number between 0 and 255 can act as the operand, and it has to be prefixed with a hash sign. Thus the instruction:

```
LDX #$5B
```

loads the hexadecimal number $5B (or decimal 92) into the X register. Instructions using immediate addressing are always 2 bytes long, and the second byte is always the operand.

Absolute Addressing

This allows us to address any one of the 65536 memory locations in the 6510, since, as we've seen, two bytes put together allow us to look at such large numbers. The table above shows us that absolute addressing uses two consecutive bytes. Since the first byte in a machine code instruction is always the op-code, it follows then that an instruction using absolute addressing will always take up three bytes of memory, with the second byte being the lower part of the operand address and the third byte being the higher part. For instance:

```
AD 00 04   LDA $0400
```

will take the content of memory location $0400 and store it in the accumulator.

Zero Page Addressing

This is an alternative form of absolute addressing, in which the operand now consists of a single byte. This means that we can only cover the first 256 bytes of memory, and this is referred as page zero. Since we're only using a single byte for the operand, it follows that zero page addressing instructions will always consist of two bytes: the op-code and the operand. For instance:

```
LDA $1B
```

will load the content of memory location $1B into the accumulator.

Apart from JSR and JMP, any command that can use absolute addressing can also use zero page addressing. What are the advantages of using zero page? Well, since we're using two byte instructions instead of three byte, we're taking up less memory, and this two byte instruction takes one cycle less to process than the normal three byte. Zero page is also useful as a temporary storage area for data values, since there is a reasonable amount of empty space in there.

However, be very careful when using zero page. Since it is faster to do this, and also takes up less memory, the 6510 wants to use it as much as you do, so watch out that your two interests don't run into conflict.

Implied Addressing

About half the instructions that the 6510 is capable of processing do no more than clear registers, transfer data from one register to another, increment or decrement registers, and so on. These commands need no operand, since the 6510 receives enough information from the op-code: the rest is implied, hence the name for this mode of addressing.

Some examples include DEX for decrementing the content of the X register, TXA for transferring the content of the X register to the accumulator, and so on. All implied commands take up just one byte of memory.

Indirect Absolute Addressing

There is only one instruction in the entire repertoire of the 6510 that can use this mode of addressing, and that is the JMP instruction. Using this mode, the JMP command causes the program counter to be loaded with a new address from which the 6510 is to fetch its next instruction. When used in absolute mode, the JMP command simply puts the destination address into the program counter, as in:

```
JMP $C021
```

This causes the memory location C021 to be stored in the program counter, hence program execution will continue at that location. However, when used in implied mode, something totally different happens. For instance, the command:

```
JMP ($01F0)
```

causes the program counter to be loaded with the low-order address stored in location $01F0, and the high order address stored in location $01F1. It's probably easier to see what's going on by way of an illustration, rather than talking about it:

Location	Content
C000	$6C
C001	$F0
C002	$01
:	:
:	:
:	:
01F0	$21
01F1	$C0

Thus what gets stored in the program counter is the content of memory location $01F0 and $01F1. The program counter would now contain the memory address $C021, and program execution would now continue at this point. Isn't all this a bit complicated, when you could probably use an ordinary JMP instead? Well, most of the time you would use an ordinary JMP, but there are times when this sort of indirect addressing is most useful.

For example, there are a lot of systems now available that allow several keyboards to be connected up to one main terminal. That main terminal then has to act according to which keyboard is currently being used to access it. Using indirect addressing, we could then start acting upon different parts of the program in the main terminal according to which keyboard was currently in use, as the 6510 would change the contents of some of its memory locations as different devices were attached to it (or in this case, as different keyboards came into play). Thus the indirect jump to those locations that change would cause program execution to continue at the correct part of the program for that keyboard.

Absolute Indexed Addressing

We've referred to this in the past as STA offset. To put it more formally, the address of the operand is computed by taking the absolute address implied in the instruction and adding the content of the X or Y register to it, depending on which is being used. This sort of addressing makes for a three byte machine code instruction. For example:

```
LDA $C020,X
```

will load the accumulator with the content of memory location $C020 plus X. That is, if the X register contains a 5, then the accumulator

will be loaded with the content of memory location $C025. This sort of addressing is useful in many ways, whether for accessing long lists of data (we'll have a look at this in the section on adding commands to BASIC), providing some animation on the screen, and so on. It was used effectively in the section on drawing the border, for example.

Zero Page Indexed Addressing

This is the two byte equivalent of absolute indexed addressing, and since we only use one byte for the operand we are restricted to accessing the first 256 bytes of memory: zero page again. As with absolute indexed, the operand is found by adding the content of the X or Y register to the zero page address specified in the second byte of the instruction. For example:

```
LDA $57,X
```

If the X register contains a 7, then the accumulator will be loaded with the content of memory location $57 + $7, or $5E. As with other zero page equivalents, this command takes up one less byte of memory than its absolute equivalent, and it will usually run one cycle faster per instruction as well.

However, a word of warning. Since we are only using one byte for the operand, we are restricted to just those first 256 bytes. But what about if the operand plus the X register exceeds 256? The answer is that it just wraps around, so that the command:

```
LDA $FA,X
```

would be okay as long as the X register didn't contain anything greater than 5, but if it contained 6, then the effective address thus produced ($FA plus 69) would be one more than 255, so it would wrap around to zero again, causing chaos in your programs.

Indexed Indirect Addressing

If you're a drinking man, now's the time to reach for the whisky, because this is going to get a mite hairy.

This addressing mode is a combination of two that we've already discussed: indexed addressing and indirect addressing. If you re-read those sections now, you'll remember that indexed addressing required

us to add a value to the operand to get the effective address for our data, and indirect addressing required us to go to the address of the first of the two memory locations that contained the data, rather than acting on the data itself.

By combining these two, we can use a two byte instruction to access all 64K of memory in the 64. Swings and roundabouts of course, since this command now takes 6 cycles to operate rather than the usual 3 or 4 of the ordinary zero-page indexed and absolute indexed.

This is how it works. The value in the X register is added to the zero page operand specified in the instruction, to produce an indirect zero page address. So far, fairly similar to our ordinary indexed addressing mode. Then, and this is where the indirect part comes in, the effective memory address comes from the first byte of the indirect zero page address (the low order byte) and the second byte of the indirect zero page address (the high order byte). The content of this effective address is then stored in the accumulator.

Let's take a look at an example:

LDA ($1B,X)

If the X register contains, say, $40, then the 6510 will compute an address of $5B ($1B offset with $40). It then looks at location $5B to get the low order address and $5C to get the high order address of the effective memory address. If $5B contained $21 and $5C contained $C0, then the effective memory address would be $C021. The content of $C021 would then be loaded into the accumulator.

It's best to play around with scraps of paper and little diagrams to figure out this one!

Indirect Indexed Addressing

If you thought that one was bad...

This combines the same two addressing modes as Indexed Indirect, but uses them in reverse order. For example:

LDA ($1B),Y

Say the Y register contained $40. The base address would be fetched from location $1B (low order) and $1C (high order). So if $1B contained

$21, and $1C contained $C0, then the base address would be $C021. Our effective address is found by adding the content of the Y register to that address, to give an effective address of $C061. The content of that address is then stored in the accumulator.

Note that these last modes are the only real difference between the X and the Y registers. You have to use the X register with indexed indirect, and you have to use the Y register with indirect indexed.

We'll be coming back to both of these at a later date, since they aren't the easiest of concepts to grasp at this stage in the game.

Relative Addressing

This is the mode used by all the branching instructions, some of which we've already seen, such as BNE, BEQ and BPL. The address to transfer program execution to (or, to put it another way, the value that is to be stored in the program counter), depends on the operand after the op-code.

To give an example:

BNE $05

will cause program execution to continue 5 bytes further on if the result of the last operation was not equal to zero. The command:

BNE $F9

would cause program execution to branch backwards 6 bytes. As we saw earlier on, we can only branch forward a maximum of 127 bytes, and backwards a maximum of 128 bytes, since all of these branching instructions have a single byte for their operand.

Accumulator Addressing

This is a collection of four instructions that affect the content of the accumulator, and we'll be looking at all of them later.

Right then, enough of theory, let's see how Fergus is getting on.

Scrolling along

Since we last looked at the listing, a number of important changes and additions have been made to the program, including putting in some left and right scrolling depending on which way Fergus is going at the time. We've also, at last, brought the aliens into play and they now majestically sweep down the screen and generally get in the way.

Your fire button is now the space bar, since this is easier to tap when using the four movement keys than reaching over to find the RETURN key and probably missing it.

```
B*
   PC  SR AC XR YR SP
.:4740 33 00 28 00 F6
.
C000 A9 0D      LDA #$0D
C002 8D F8 07   STA $07F8
C005 A9 03      LDA #$03
C007 8D 15 D0   STA $D015
C00A A9 0E      LDA #$0E
C00C 8D F9 07   STA $07F9
C00F 20 00 C3   JSR $C300
C012 A9 00      LDA #$00
C014 8D 17 D0   STA $D017
C017 8D 1D D0   STA $D01D
C01A A9 00      LDA #$00
C01C 8D 10 D0   STA $D010
C01F EA         NOP
C020 EA         NOP
C021 4C 00 C5   JMP $C500
C024 AD C5 00   LDA $00C5
C027 C9 0A      CMP #$0A
C029 F0 1C      BEQ $C047
C02B C9 12      CMP #$12
C02D F0 42      BEQ $C071
C02F C9 21      CMP #$21
C031 F0 6D      BEQ $C0A0
C033 C9 24      CMP #$24
C035 F0 7C      BEQ $C0B3
C037 C9 3C      CMP #$3C
C039 F0 09      BEQ $C044
C03B 20 00 C1   JSR $C100
C03E 20 00 C1   JSR $C100
C041 4C 21 C0   JMP $C021
C044 4C C7 C0   JMP $C0C7
```

```
C047 A9 00      LDA #$00
C049 CD 10 D0   CMP $D010
C04C D0 0E      BNE $C05C
C04E E0 18      CPX #$18
C050 F0 01      BEQ $C053
C052 CA         DEX
C053 4C 16 C4   JMP $C416
C056 8E 00 D0   STX $D000
C059 4C 21 C0   JMP $C021
C05C E0 00      CPX #$00
C05E F0 03      BEQ $C063
C060 4C 52 C0   JMP $C052
C063 A9 00      LDA #$00
C065 8D 10 D0   STA $D010
C068 A2 FF      LDX #$FF
C06A 8E 00 D0   STX $D000
C06D 4C 21 C0   JMP $C021
C070 00         BRK
C071 E0 FF      CPX #$FF
C073 D0 03      BNE $C078
C075 4C 21 C0   JMP $C021
C078 E8         INX
C079 4C 00 C4   JMP $C400
C07C 8E 00 D0   STX $D000
C07F A9 00      LDA #$00
C081 CD 10 D0   CMP $D010
C084 D0 11      BNE $C097
C086 E0 FF      CPX #$FF
C088 D0 EB      BNE $C075
C08A A9 07      LDA #$07
C08C 8D 10 D0   STA $D010
C08F A2 01      LDX #$01
C091 8E 00 D0   STX $D000
C094 4C 21 C0   JMP $C021
C097 E0 40      CPX #$40
C099 D0 DA      BNE $C075
C09B CA         DEX
C09C 4C 21 C0   JMP $C021
C09F FF         ???
C0A0 C0 32      CPY #$32
C0A2 D0 03      BNE $C0A7
C0A4 4C 21 C0   JMP $C021
C0A7 88         DEY
C0A8 20 00 C1   JSR $C100
C0AB 8C 01 D0   STY $D001
C0AE 4C 21 C0   JMP $C021
C0B1 00         BRK
C0B2 00         BRK
C0B3 C0 E5      CPY #$E5
C0B5 D0 03      BNE $C0BA
C0B7 4C 21 C0   JMP $C021
```

Binary, as you know, is a system of representing numbers as a series of 0s or 1s, rather than using the digits 0 to 9 as we do. This is because computers can only understand two states (an electronic circuit can only be on or off, it can't be anything in between), and the binary notation follows logically from that.

So, in binary, 27 becomes 00011011
128 becomes 10000000

Remembering the rules for AND and OR, 27 AND 128 now becomes 00000000 (at no point do we have two 1s together), and 27 OR 128 becomes 10011011 (if a 1 occurs in either number, the result is a 1).

In plain English then, 27 AND 128 equals 0, and 27 OR 128 equals 155.

One final example:

53 in binary equals 00000110101
1111 in binary equals 10001010111

53 AND 1111 equals 00000010101 = 21
53 OR 1111 equals 10001110111 = 1143

Other logical operators

These have been mentioned earlier, in statements like:

IF A>5

and so on.

Knowing how truth tables work, it now becomes a simple matter to understand all these logical operators, and calculate the results that they will give.

To sum up, the remaining operators are:

= : equal to
< : less than
<= : less than or equal to
> : greater than
>= : greater than or equal to
<> : not equal to

We've already been through the routine from $C000 to $C01E, as well as the routine from $C021 to $C046, but it's wise to note that there are a few changes from the listing that we last looked at. In the second block, room has had to be made to include a JMP $C500 command (which sets up the aliens and controls their movement), so everything has been shoved down a little. This has meant changing all the branch instructions, so watch out for them when it comes to typing in the changes.

What we haven't looked at in any great detail is the routines for moving our hero around the screen, and for firing the missile at the oncoming enemy. The missile doesn't do anything yet, by the way, other than travel serenely up the screen. That will have to wait till chapter 7.

Moving left

Let's take a look at the routine from $C047 to $C070: the one that moves Fergus to the left. Don't worry about the 00 BRK command at location $C070: program execution never reaches it.

The first thing that this routine does is to load the accumulator with a zero and compare it with memory location $D010. This location is the most significant bit of the horizontal position of the sprite, and determines whether the sprite is to the left of location 255 or the right. If it's set to 1 we branch to location $C05C to carry out some tests over there, but if it's set to zero we carry on with location $C04E.

This checks the content of the X register with $18, to see if he's still on the screen or not. If he is, we can decrease the content of the X register by one in the next instruction, but if he's at the screen border we won't decrease the X register at all, so that instruction is missed out with a simple branch command.

Program execution then goes to the routine at location $C416, which we'll come to in a moment, and then back to $C021 after storing the content of the X register in location $D000: the X position of sprite zero, our hero Fergus.

From $C05C to $C070 the most significant bit of the X position is set to one, so all we're concerned about is whether or not we're at the transition point to set it back to zero again. So this is what we check for: does the X register contain a zero, because if it does we have to set the most significant bit to zero, load the X register with 255 to move Fergus over, store the X register in location $D000 and go back to the

As you can see, quite complex decision making can be achieved using these operators.

NOT is a peculiar one, and doesn't seem to be used very much. Still, just about every decision-making process you will require can be achieved with AND and OR, as the following example should serve to show.

```
10 IF A<10 OR B>5 AND A$="Y" THEN 200
```

Here, the program will branch off to line 200 if A is less than 10 OR B is greater than 5, but only if A$ is equal to "Y" as well. If all these conditions are not met, the program just falls through to the next line.

What we're doing here is testing to see whether various statements are true or false. Try the following short example:

```
10 A=20
20 PRINT (A>15)
```

When run, this program will print out the value -1, because the statement A>15 is a true one: we've just defined A to be equal to 20.

So if something is true, the computer prints out a -1, and if it is false it prints out a zero. For instance:

```
10 A=20
20 PRINT (A>25)
```

will result in 0 being printed, as the statement is patently not true.

All of this is based on what are called TRUTH TABLES, and the table for AND works as follows:

Truth tables

A	B	C
0	0	0
0	-1	0
-1	0	0
-1	-1	-1

The table operates in the following way: if the first statement A is false

The Y position of the missile is then compared with $10, to see if it's gone off the screen or not. If it has, then we branch to location $C0EE to do a bit of tidying up: retrieve the Y position of Fergus, change the waveform for voice 1 back to a noise one, and go back to location $C021 again.

If it hasn't, then we decrease the Y register, store it in the memory location that holds the Y co-ordinate of the missile sprite, and jump to three subroutines. The first updates the aliens, the second is a delay loop, and the third checks to see if anything has hit anything else. We'll come to those later on.

Then the program loops back to $C0DA to test for Y being equal to $10 again.

The delay loop

This is the routine that slows everything down. Without this, you wouldn't be able to see what was happening. It occupies locations $C100 to $C117, with a few gaps in between which we'll fill up later. The command EA NOP simply means do nothing for a couple of cycles, which doesn't slow the program down a great deal, but which does give us space to add some routines later on.

First of all, the content of the X register is stored in a safe place before loading it with $FF, or decimal 255. The next couple of instructions just decrease the content of the X register and compare it to zero. A simple but effective delay loop.

When X is equal to zero, after a few NOPs (No OPeration) we store the content of the Y register at location $D401, which is the location that holds the high frequency for voice 1, pick up the original content of the X register again, and finally return from the subroutine.

Start up routine

This is called once only at the start of the program, and sets all the sprite positions up, as well as initialising the sound.

```
.
C300 A9 0F      LDA #$0F
C302 8D 18 D4   STA $D418
C305 A9 22      LDA #$22
```

the screen. Altering this value alters how many lines will be scrolled). Store the accumulator at location $57, the first spare byte in page zero.

The accumulator is then loaded with the value $04, and stored at location $58, the second spare value. Loading the Y register with a zero and loading the accumulator with the content of location $57 offset with Y has the effect of putting a 04 in the accumulator the first time around the loop, and this value is then stored in location $59. The program then looks at every screen memory location in turn, and stores whatever happens to be there in the location immediately to the left of it. Remember the value $04 which we originally loaded into the accumulator at $CA06? $0400 happens to be the start of the screen memory, and so what is happening here is that in the order low-byte high-byte everything is successively moved one location to the left.

At location $CA23 $28 (or decimal 40) is added to the value held in the accumulator, which has the effect of stepping us down onto the next screen line. At locations $CA2B to $CA2D we check to see if all 24 lines on the screen have been covered. If they have, then return from this routine, and if they haven't, go back and move another line along one character.

Conclusion

The game is slowly evolving into something, and you should by now be quite at home looking at what are still relatively simple machine code routines. In the early part of this chapter a lot of theory was covered, and it's well worth reading through it a couple of times to try to grasp what is going on.

The second part of this chapter took an extensive look through the games listing. It isn't finished yet, and in chapter 7 we'll be taking a look at the final version: at least, the final version that you'll be presented with in this book. It's up to you to turn it into a fully fledged arcade game with high score tables, opening credits, and so on. The rudiments are here, but the final polish is up to you.

After the fun of working through a games listing though, it's back to the blackboard once more, and a look at logical operators and the four commands that we mentioned earlier in the section 'Accumulator Addressing'.

The X and Y registers are then loaded with $A0 (or decimal 160), and these values are stored in the two locations that handle the X and Y co-ordinate positions for sprite 0, thus indicating where our character Fergus will appear at the start of the game. A zero is then put into the accumulator, and stored at the six locations that control the vertical position of the next six sprites (the enemy!). Finally, the accumulator is loaded with one, and this value is stored in location $D027: the machine code equivalent of typing in BASIC POKE 53287,1. This indicates that the colour of our first sprite, Fergus, will be white.

This routine finishes with an RTS instruction, which sends program execution back to wherever it came from. In this case, memory location $C012.

Calling the scroll routine

This routine is called up whenever the character is moving to the left or the right, in which case we also want to scroll the background right or left to give the illusion that he's actually moving.

```
.
C400 8E 12 C4    STX $C412
C403 8C 13 C4    STY $C413
C406 20 00 CA    JSR $CA00
C409 AE 12 C4    LDX $C412
C40C AC 13 C4    LDY $C413
C40F 4C 7C C0    JMP $C07C
C412 7E E5 FF    ROR $FFE5,X
C415 FF          ???
C416 8E 12 C4    STX $C412
C419 8C 13 C4    STY $C413
C41C 20 32 CA    JSR $CA32
C41F AE 12 C4    LDX $C412
C422 AC 13 C4    LDY $C413
C425 4C 56 C0    JMP $C056
C428 00          BRK
.
```

The routine itself doesn't do very much. It just stores the current values in the X and Y registers at a couple of safe memory locations that won't get altered by anything else, then jumps to the appropriate scroll

collided with. The AND #$02 is a check to see if the missile has collided with something, and if it has then carry on at location $C694. There is a slight error in the program at this point, since this should really be a branch to location $C690: you can't win them all, and the mistake is corrected later.

If nothing has collided with either the missile or Fergus, then jump back to location $C646 and carry on from there.

At $C67D onwards, a check is carried out to see if it's a missile to hero collision. If it is, then jump to $C68A and from there jump back to $C646, but if it isn't then we have to assume that Fergus has collided with one of the enemy, so change his colour and exit from the program with a BRK command.

Similarly, from $C690 (where program execution should have gone to from location $C678) onwards we check for a missile to Fergus collision again. If it is, then jump to $C69E and from there jump to $C646 again. Otherwise, load the accumulator with $00 and store it at location $D028 (which has the effect of changing the missile's colour to black) and exit from the program with a BRK command.

The scroll routines

Two of these, occupying locations $CA00 to $CA2E and $CA30 to $CA5E. The first one is used to scroll the background to the left, and the second scrolls it to the right. Some interesting techniques are involved in these two routines, although both of them are really quite similar. We'll take the first one and go through that fairly exhaustively. By then you should be able to follow the second one: it uses much the same logic.

```
.
CA00 A9 28      LDA #$28
CA02 A2 18      LDX #$18
CA04 85 57      STA $57
CA06 A9 04      LDA #$04
CA08 85 58      STA $58
CA0A A0 00      LDY #$00
CA0C B1 57      LDA ($57),Y
CA0E 85 59      STA $59
CA10 C8         INY
CA11 B1 57      LDA ($57),Y
CA13 88         DEY
CA14 91 57      STA ($57),Y
```

To begin with, the accumulator is loaded with a value of $3F, or decimal 63. If all the sprites are active, this is the value that will be held in the register to turn sprites on. If they're not active, then something different will be stored there and we'll have to turn them all on.

The next couple of instructions store the X and Y registers in reasonably safe locations, before doing the check to see how many sprites are turned on (CMP $D015: a direct comparison between the value stored in the accumulator, which is the value we've just put there, and the value currently held in memory location $D015, the location that turns various sprites off and on). If the required number are turned on, then program execution branches off to location $C538 and continues from there.

Otherwise, we go through a lengthy set of instructions to turn all the sprites on. This is done by going through a loop 5 times (LDY #$06 and stop when the value in the Y register reaches zero), telling the computer that the data for the enemy sprites is held in the $FBth memory location: equivalent to POKE 2042,241:POKE 2043,241 etc. This is later altered in the final version of the program to step through each enemy sprite in turn, to give the impression that there are a number of different enemies out there, just waiting to get you.

All the sprites are turned on, and then the accumulator is loaded with a random value (LDA $00A2). This location is the one where the seconds part of the internal clock of the 64 is stored. Since this is being updated all the time, the value held there will always be changing, and therefore our sprites will not keep re-appearing at the same old locations at the top of the screen.

The program then stores whatever value happens to be in location $00A2 in the register that looks after the horizontal co-ordinate position of sprite 2, adds a number to it, stores that for sprite 3, and so on. The accumulator is then loaded with $32 (or decimal 50), which is stored in location $D005: the vertical co-ordinate position for sprite 2.

A check is made to see if we've covered all the sprites. If we haven't, then loop back and do it all again, but if we have then we recover the values that were originally in the X and Y registers and jump off to the subroutine at $C600 which updates the positions of all the enemy sprites, amongst other things.

Finally, we jump back to the routine which checks to see if any keys have been pressed. Memory locations $C547 to $C556 can be ignored for the time being, since they are only duplicates of earlier instructions:

```
C656 A9 00      LDA #$00
C658 8D 03 D0   STA $D003
C65B A0 10      LDY #$10
C65D 60         RTS
C65E A9 04      LDA #$04
C660 CD 1E D0   CMP $D01E
C663 EA         NOP
C664 EA         NOP
C665 EA         NOP
C666 EA         NOP
C667 30 03      BMI $C66C
C669 4C 46 C6   JMP $C646
C66C AD 1E D0   LDA $D01E
C66F 29 01      AND #$01
C671 D0 0A      BNE $C67D
C673 AD 1E D0   LDA $D01E
C676 29 02      AND #$02
C678 D0 1A      BNE $C694
C67A 4C 46 C6   JMP $C646
C67D A9 03      LDA #$03
C67F CD 1E D0   CMP $D01E
C682 F0 06      BEQ $C68A
C684 A9 04      LDA #$04
C686 8D 27 D0   STA $D027
C689 00         BRK
C68A 4C 46 C6   JMP $C646
C68D EA         NOP
C68E EA         NOP
C68F EA         NOP
C690 A9 03      LDA #$03
C692 CD 1E D0   CMP $D01E
C695 F0 07      BEQ $C69E
C697 EA         NOP
C698 A9 00      LDA #$00
C69A 8D 28 D0   STA $D028
C69D 00         BRK
C69E 4C 46 C6   JMP $C646
C6A1 FF         ???
.
```

At the start of the routine, we store the current values held in the X and Y registers in a couple of memory locations that will not be altered by anything else. These can later be retrieved from those memory locations and program execution can continue without anything untoward happening.

Program execution then leaps off to the routine starting at memory location $C640, which we'll come to in a moment, before loading the Y register with the content of memory location $D005: the current

```
C656 A9 00      LDA #$00
C658 8D 03 D0   STA $D003
C65B A0 10      LDY #$10
C65D 60         RTS
C65E A9 04      LDA #$04
C660 CD 1E D0   CMP $D01E
C663 EA         NOP
C664 EA         NOP
C665 EA         NOP
C666 EA         NOP
C667 30 03      BMI $C66C
C669 4C 46 C6   JMP $C646
C66C AD 1E D0   LDA $D01E
C66F 29 01      AND #$01
C671 D0 0A      BNE $C67D
C673 AD 1E D0   LDA $D01E
C676 29 02      AND #$02
C678 D0 1A      BNE $C694
C67A 4C 46 C6   JMP $C646
C67D A9 03      LDA #$03
C67F CD 1E D0   CMP $D01E
C682 F0 06      BEQ $C68A
C684 A9 04      LDA #$04
C686 8D 27 D0   STA $D027
C689 00         BRK
C68A 4C 46 C6   JMP $C646
C68D EA         NOP
C68E EA         NOP
C68F EA         NOP
C690 A9 03      LDA #$03
C692 CD 1E D0   CMP $D01E
C695 F0 07      BEQ $C69E
C697 EA         NOP
C698 A9 00      LDA #$00
C69A 8D 28 D0   STA $D028
C69D 00         BRK
C69E 4C 46 C6   JMP $C646
C6A1 FF         ???
.
```

At the start of the routine, we store the current values held in the X and Y registers in a couple of memory locations that will not be altered by anything else. These can later be retrieved from those memory locations and program execution can continue without anything untoward happening.

Program execution then leaps off to the routine starting at memory location $C640, which we'll come to in a moment, before loading the Y register with the content of memory location $D005: the current

To begin with, the accumulator is loaded with a value of $3F, or decimal 63. If all the sprites are active, this is the value that will be held in the register to turn sprites on. If they're not active, then something different will be stored there and we'll have to turn them all on.

The next couple of instructions store the X and Y registers in reasonably safe locations, before doing the check to see how many sprites are turned on (CMP $D015: a direct comparison between the value stored in the accumulator, which is the value we've just put there, and the value currently held in memory location $D015, the location that turns various sprites off and on). If the required number are turned on, then program execution branches off to location $C538 and continues from there.

Otherwise, we go through a lengthy set of instructions to turn all the sprites on. This is done by going through a loop 5 times (LDY #$06 and stop when the value in the Y register reaches zero), telling the computer that the data for the enemy sprites is held in the $FBth memory location: equivalent to POKE 2042,241:POKE 2043,241 etc. This is later altered in the final version of the program to step through each enemy sprite in turn, to give the impression that there are a number of different enemies out there, just waiting to get you.

All the sprites are turned on, and then the accumulator is loaded with a random value (LDA $00A2). This location is the one where the seconds part of the internal clock of the 64 is stored. Since this is being updated all the time, the value held there will always be changing, and therefore our sprites will not keep re-appearing at the same old locations at the top of the screen.

The program then stores whatever value happens to be in location $00A2 in the register that looks after the horizontal co-ordinate position of sprite 2, adds a number to it, stores that for sprite 3, and so on. The accumulator is then loaded with $32 (or decimal 50), which is stored in location $D005: the vertical co-ordinate position for sprite 2.

A check is made to see if we've covered all the sprites. If we haven't, then loop back and do it all again, but if we have then we recover the values that were originally in the X and Y registers and jump off to the subroutine at $C600 which updates the positions of all the enemy sprites, amongst other things.

Finally, we jump back to the routine which checks to see if any keys have been pressed. Memory locations $C547 to $C556 can be ignored for the time being, since they are only duplicates of earlier instructions:

collided with. The AND #$02 is a check to see if the missile has collided with something, and if it has then carry on at location $C694. There is a slight error in the program at this point, since this should really be a branch to location $C690: you can't win them all, and the mistake is corrected later.

If nothing has collided with either the missile or Fergus, then jump back to location $C646 and carry on from there.

At $C67D onwards, a check is carried out to see if it's a missile to hero collision. If it is, then jump to $C68A and from there jump back to $C646, but if it isn't then we have to assume that Fergus has collided with one of the enemy, so change his colour and exit from the program with a BRK command.

Similarly, from $C690 (where program execution should have gone to from location $C678) onwards we check for a missile to Fergus collision again. If it is, then jump to $C69E and from there jump to $C646 again. Otherwise, load the accumulator with $00 and store it at location $D028 (which has the effect of changing the missile's colour to black) and exit from the program with a BRK command.

The scroll routines

Two of these, occupying locations $CA00 to $CA2E and $CA30 to $CA5E. The first one is used to scroll the background to the left, and the second scrolls it to the right. Some interesting techniques are involved in these two routines, although both of them are really quite similar. We'll take the first one and go through that fairly exhaustively. By then you should be able to follow the second one: it uses much the same logic.

```
.
CA00 A9 28      LDA #$28
CA02 A2 18      LDX #$18
CA04 85 57      STA $57
CA06 A9 04      LDA #$04
CA08 85 58      STA $58
CA0A A0 00      LDY #$00
CA0C B1 57      LDA ($57),Y
CA0E 85 59      STA $59
CA10 C8         INY
CA11 B1 57      LDA ($57),Y
CA13 88         DEY
CA14 91 57      STA ($57),Y
```

The X and Y registers are then loaded with $A0 (or decimal 160), and these values are stored in the two locations that handle the X and Y co-ordinate positions for sprite 0, thus indicating where our character Fergus will appear at the start of the game. A zero is then put into the accumulator, and stored at the six locations that control the vertical position of the next six sprites (the enemy!). Finally, the accumulator is loaded with one, and this value is stored in location $D027: the machine code equivalent of typing in BASIC POKE 53287,1. This indicates that the colour of our first sprite, Fergus, will be white.

This routine finishes with an RTS instruction, which sends program execution back to wherever it came from. In this case, memory location $C012.

Calling the scroll routine

This routine is called up whenever the character is moving to the left or the right, in which case we also want to scroll the background right or left to give the illusion that he's actually moving.

```
.
C400 8E 12 C4    STX $C412
C403 8C 13 C4    STY $C413
C406 20 00 CA    JSR $CA00
C409 AE 12 C4    LDX $C412
C40C AC 13 C4    LDY $C413
C40F 4C 7C C0    JMP $C07C
C412 7E E5 FF    ROR $FFE5,X
C415 FF          ???
C416 8E 12 C4    STX $C412
C419 8C 13 C4    STY $C413
C41C 20 32 CA    JSR $CA32
C41F AE 12 C4    LDX $C412
C422 AC 13 C4    LDY $C413
C425 4C 56 C0    JMP $C056
C428 00          BRK
.
```

The routine itself doesn't do very much. It just stores the current values in the X and Y registers at a couple of safe memory locations that won't get altered by anything else, then jumps to the appropriate scroll

the screen. Altering this value alters how many lines will be scrolled). Store the accumulator at location $57, the first spare byte in page zero.

The accumulator is then loaded with the value $04, and stored at location $58, the second spare value. Loading the Y register with a zero and loading the accumulator with the content of location $57 offset with Y has the effect of putting a 04 in the accumulator the first time around the loop, and this value is then stored in location $59. The program then looks at every screen memory location in turn, and stores whatever happens to be there in the location immediately to the left of it. Remember the value $04 which we originally loaded into the accumulator at $CA06? $0400 happens to be the start of the screen memory, and so what is happening here is that in the order low-byte high-byte everything is successively moved one location to the left.

At location $CA23 $28 (or decimal 40) is added to the value held in the accumulator, which has the effect of stepping us down onto the next screen line. At locations $CA2B to $CA2D we check to see if all 24 lines on the screen have been covered. If they have, then return from this routine, and if they haven't, go back and move another line along one character.

Conclusion

The game is slowly evolving into something, and you should by now be quite at home looking at what are still relatively simple machine code routines. In the early part of this chapter a lot of theory was covered, and it's well worth reading through it a couple of times to try to grasp what is going on.

The second part of this chapter took an extensive look through the games listing. It isn't finished yet, and in chapter 7 we'll be taking a look at the final version: at least, the final version that you'll be presented with in this book. It's up to you to turn it into a fully fledged arcade game with high score tables, opening credits, and so on. The rudiments are here, but the final polish is up to you.

After the fun of working through a games listing though, it's back to the blackboard once more, and a look at logical operators and the four commands that we mentioned earlier in the section 'Accumulator Addressing'.

The Y position of the missile is then compared with $10, to see if it's gone off the screen or not. If it has, then we branch to location $C0EE to do a bit of tidying up: retrieve the Y position of Fergus, change the waveform for voice 1 back to a noise one, and go back to location $C021 again.

If it hasn't, then we decrease the Y register, store it in the memory location that holds the Y co-ordinate of the missile sprite, and jump to three subroutines. The first updates the aliens, the second is a delay loop, and the third checks to see if anything has hit anything else. We'll come to those later on.

Then the program loops back to $C0DA to test for Y being equal to $10 again.

The delay loop

This is the routine that slows everything down. Without this, you wouldn't be able to see what was happening. It occupies locations $C100 to $C117, with a few gaps in between which we'll fill up later. The command EA NOP simply means do nothing for a couple of cycles, which doesn't slow the program down a great deal, but which does give us space to add some routines later on.

First of all, the content of the X register is stored in a safe place before loading it with $FF, or decimal 255. The next couple of instructions just decrease the content of the X register and compare it to zero. A simple but effective delay loop.

When X is equal to zero, after a few NOPs (No OPeration) we store the content of the Y register at location $D401, which is the location that holds the high frequency for voice 1, pick up the original content of the X register again, and finally return from the subroutine.

Start up routine

This is called once only at the start of the program, and sets all the sprite positions up, as well as initialising the sound.

```
.
C300 A9 0F      LDA #$0F
C302 8D 18 D4   STA $D418
C305 A9 22      LDA #$22
```

As you can see, quite complex decision making can be achieved using these operators.

NOT is a peculiar one, and doesn't seem to be used very much. Still, just about every decision-making process you will require can be achieved with AND and OR, as the following example should serve to show.

```
10 IF A<10 OR B>5 AND A$="Y" THEN 200
```

Here, the program will branch off to line 200 if A is less than 10 OR B is greater than 5, but only if A$ is equal to "Y" as well. If all these conditions are not met, the program just falls through to the next line.

What we're doing here is testing to see whether various statements are true or false. Try the following short example:

```
10 A=20
20 PRINT (A>15)
```

When run, this program will print out the value -1, because the statement A>15 is a true one: we've just defined A to be equal to 20.

So if something is true, the computer prints out a -1, and if it is false it prints out a zero. For instance:

```
10 A=20
20 PRINT (A>25)
```

will result in 0 being printed, as the statement is patently not true.

All of this is based on what are called TRUTH TABLES, and the table for AND works as follows:

Truth tables

A	B	C
0	0	0
0	-1	0
-1	0	0
-1	-1	-1

The table operates in the following way: if the first statement A is false

We've already been through the routine from $C000 to $C01E, as well as the routine from $C021 to $C046, but it's wise to note that there are a few changes from the listing that we last looked at. In the second block, room has had to be made to include a JMP $C500 command (which sets up the aliens and controls their movement), so everything has been shoved down a little. This has meant changing all the branch instructions, so watch out for them when it comes to typing in the changes.

What we haven't looked at in any great detail is the routines for moving our hero around the screen, and for firing the missile at the oncoming enemy. The missile doesn't do anything yet, by the way, other than travel serenely up the screen. That will have to wait till chapter 7.

Moving left

Let's take a look at the routine from $C047 to $C070: the one that moves Fergus to the left. Don't worry about the 00 BRK command at location $C070: program execution never reaches it.

The first thing that this routine does is to load the accumulator with a zero and compare it with memory location $D010. This location is the most significant bit of the horizontal position of the sprite, and determines whether the sprite is to the left of location 255 or the right. If it's set to 1 we branch to location $C05C to carry out some tests over there, but if it's set to zero we carry on with location $C04E.

This checks the content of the X register with $18, to see if he's still on the screen or not. If he is, we can decrease the content of the X register by one in the next instruction, but if he's at the screen border we won't decrease the X register at all, so that instruction is missed out with a simple branch command.

Program execution then goes to the routine at location $C416, which we'll come to in a moment, and then back to $C021 after storing the content of the X register in location $D000: the X position of sprite zero, our hero Fergus.

From $C05C to $C070 the most significant bit of the X position is set to one, so all we're concerned about is whether or not we're at the transition point to set it back to zero again. So this is what we check for: does the X register contain a zero, because if it does we have to set the most significant bit to zero, load the X register with 255 to move Fergus over, store the X register in location $D000 and go back to the

Binary, as you know, is a system of representing numbers as a series of 0s or 1s, rather than using the digits 0 to 9 as we do. This is because computers can only understand two states (an electronic circuit can only be on or off, it can't be anything in between), and the binary notation follows logically from that.

So, in binary, 27 becomes 00011011
128 becomes 10000000

Remembering the rules for AND and OR, 27 AND 128 now becomes 00000000 (at no point do we have two 1s together), and 27 OR 128 becomes 10011011 (if a 1 occurs in either number, the result is a 1).

In plain English then, 27 AND 128 equals 0, and 27 OR 128 equals 155.

One final example:

53 in binary equals 00000110101
1111 in binary equals 10001010111

53 AND 1111 equals 00000010101 = 21
53 OR 1111 equals 10001110111 = 1143

Other logical operators

These have been mentioned earlier, in statements like:

IF A>5

and so on.

Knowing how truth tables work, it now becomes a simple matter to understand all these logical operators, and calculate the results that they will give.

To sum up, the remaining operators are:

= : equal to
< : less than
<= : less than or equal to
> : greater than
>= : greater than or equal to
<> : not equal to

```
C047 A9 00      LDA #$00
C049 CD 10 D0   CMP $D010
C04C D0 0E      BNE $C05C
C04E E0 18      CPX #$18
C050 F0 01      BEQ $C053
C052 CA         DEX
C053 4C 16 C4   JMP $C416
C056 8E 00 D0   STX $D000
C059 4C 21 C0   JMP $C021
C05C E0 00      CPX #$00
C05E F0 03      BEQ $C063
C060 4C 52 C0   JMP $C052
C063 A9 00      LDA #$00
C065 8D 10 D0   STA $D010
C068 A2 FF      LDX #$FF
C06A 8E 00 D0   STX $D000
C06D 4C 21 C0   JMP $C021
C070 00         BRK
C071 E0 FF      CPX #$FF
C073 D0 03      BNE $C078
C075 4C 21 C0   JMP $C021
C078 E8         INX
C079 4C 00 C4   JMP $C400
C07C 8E 00 D0   STX $D000
C07F A9 00      LDA #$00
C081 CD 10 D0   CMP $D010
C084 D0 11      BNE $C097
C086 E0 FF      CPX #$FF
C088 D0 EB      BNE $C075
C08A A9 07      LDA #$07
C08C 8D 10 D0   STA $D010
C08F A2 01      LDX #$01
C091 8E 00 D0   STX $D000
C094 4C 21 C0   JMP $C021
C097 E0 40      CPX #$40
C099 D0 DA      BNE $C075
C09B CA         DEX
C09C 4C 21 C0   JMP $C021
C09F FF         ???
C0A0 C0 32      CPY #$32
C0A2 D0 03      BNE $C0A7
C0A4 4C 21 C0   JMP $C021
C0A7 88         DEY
C0A8 20 00 C1   JSR $C100
C0AB 8C 01 D0   STY $D001
C0AE 4C 21 C0   JMP $C021
C0B1 00         BRK
C0B2 00         BRK
C0B3 C0 E5      CPY #$E5
C0B5 D0 03      BNE $C0BA
C0B7 4C 21 C0   JMP $C021
```

The V flag is loaded with the original contents of the sixth bit being tested.

All these logical operators really come into their own when talking about binary arithmetic and multiplication, but that must wait until chapter 8.

Now for the four instructions that can shift each individual bit in the accumulator, or in a memory location, to the left or the right.

Rotating and shifting

For all these four instructions, we have also got to consider the carry flag, since it effectively acts as the 'ninth bit' for the operand, or the number that we're working on. In either of the two shift operations, the bit position at the opposite end of the byte from the one that gets moved off is reset to zero. In the two rotate operations, the bit position at the opposite end of the byte from the one that gets moved off is given the value that was stored in the carry flag before the operation took place. The symbols used to represent these instructions are as follows:

ASL : Accumulator Shift Left
LSR : Logical Shift Right
ROL : ROtate Left
ROR : ROtate Right

These operations can work either on a location in memory, or on the accumulator. They can also be used in any of the four addressing modes: absolute, zero page, zero page offset with X, and absolute offset with X.

As well as affecting the carry flag, which we'll illustrate with a diagram or two in a moment, these four instructions affect two of the flags in the status register.

ASL, ROL and ROR cause the negative (N) flag to be set if bit 7 of the shifted result is set to a 1, otherwise it is reset to a zero.

LSR always causes the negative flag to be reset, as it always puts a zero into bit 7.

If the shifted result is zero, then the zero (Z) flag is set, otherwise it is reset to 1.

All of this is probably best illustrated by a diagram. We'll look at the decimal number 84 (binary 01010100), and say that the carry flag has been set to 1.

Carry Flag	Bit Position 7	6	5	4	3	2	1	0	
1	0	1	0	1	0	1	0	0	Before shift (decimal 84)
0	1	0	1	0	1	0	0	0	After ASL (decimal 168)
0	0	0	1	0	1	0	1	0	After LSR (decimal 42)
0	1	0	1	0	1	0	0	1	After ROL (decimal 169)
0	1	0	1	0	1	0	1	0	After ROR (decimal 170)

A quick glance at the numbers in the decimal column after these operations will reveal a number of interesting things. Performing a shifted left or right will either double or half the number under consideration, which can be useful in many kinds of arithmetic, as well as in certain kinds of sort routines. With a little bit of logical thought, you should be able to see what kind of powerful uses we could put these instructions to, and we'll see some of those uses in later chapters.

Before taking our last look at Fergus in chapter 7, we'll tie up a few lose ends and cover some of the instructions and commands that haven't yet been dealt with. But first of all, vital to a true understanding of machine code, we'll start with a little something known as the stack pointer.

The stack pointer

The stack is a 256 byte (I bet you knew that number was coming up) block of memory, that fills memory locations 256 to 511, but it fills them in a rather strange fashion.

First, what does it fill them with?

Well, one function of the stack is to hold all the addresses during subroutine jumps, and this it does automatically.

Another purpose is to transfer data rapidly from register to register,

memory location to memory location, and whether it is storing data or jump addresses, anything that goes in there is recorded from memory location 511 downwards.

In other words, location 256 is the last one to be filled.

When pulling data back off the stack, it is retrieved on a 'last-in, first-out' basis, usually abbreviated to LIFO. Thus, the last item of information that went in there is the first one to come back out again.

What keeps track of where the next empty byte of stack space is? Well, just as the machine code program itself has a program counter, so the stack has a stack pointer, which stores where the next block of information can go.

To illustrate this properly, let's look at a concrete example.

The following (short) machine code program illustrates all the necessary points:

```
C000  : LDA #$01     : Load the Accumulator with 1.
C002  : JSR $F6ED    : Jump to Internal Subroutine to check
                       stop key.
C005  : TAX          : Transfer Contents of Accumulator into
                       X register.
```

Step by step, here's what happens:

(1) Find address of next instruction, i.e. C002, and put this in the program counter.

(2) Execute instruction, and get back address for next one from program counter (i.e.C002)

(3) Fetch next instruction, i.e. JSR $F6ED

(4) Find address for next instruction (C005), and put this onto stack.

(5) Put next vacant position (509, as C005 occupies two bytes) into stack pointer.

(6) Put F6ED into program counter, and jump to subroutine at $F6ED.

(7) Come back and look at stack pointer to find where last data stored : stack pointer says 509, so last data stored in 510 and 511.

(8) Get that (i.e. C005) and put it into program counter.

(9) Go and execute instruction at C005.

A program may well have to find its way back through several subroutines, as programs grow in complexity. Thankfully, the stack, stack pointer and program counter take care of all this for you.

The contents of the stack can be altered by just about everything, and two of the commands which do this concern the Accumulator. These are:

PHA : PusH contents of Accumulator onto stack, but don't change the value in the accumulator.
PLA : PuLl top of stack into Accumulator.

The status register and stack pointer can also be altered, although with care, and the commands to do this are:

PHP : PusH status register onto stack.
PLP : PuLl status register from stack.
TSX : Transfer stack pointer to X register.
TXS : Transfer X register to stack pointer.

Addition and subtraction

Using the knowledge that we now have, it is a simple matter to add or subtract two numbers together, provided that the result or the numbers are not greater than 255, or less than 0.

To add numbers together that are greater than this, the following example program should help to make things clearer.

What we have to do is store each number as two bytes: the double precision mentioned earlier.

For instance, the number 1926 in decimal is equivalent to $0786. To use this number, we split it up into two parts, namely $07 and $86. These are referred to as the Most Significant Byte (MSB) and Least Significant Byte (LSB).

First of all, we need to add the two LSBs, and see if there is a carry. Well, $86 plus $86 is equal to $16,12 or carry +0C.

```
86
86
--------
16,12  =  carry plus 0C
```

The two MSBs, $07, added together give 14, or $0E. With the carry from the addition of the two LSBs this becomes $0F, and so the final total is $0F0C, or decimal 3852, which is indeed correct.

Let's put this into a program.

Example program

Our code could look something like this:

```
.,  C000 18       CLC
.,  C001 D8       CLD
.,  C002 A9 86    LDA #$86
.,  C004 69 86    ADC #$86
.,  C006 8D 02 04 STA $0402
.,  C009 A9 07    LDA #$07
.,  C00B 69 07    ADC #$07
.,  C00D 8D 00 04 STA $0400
.,  C010 60       RTS
```

The RTS at the end of the program indicates ReTurn from Subroutine. This is there so that, when we run the program with a SYS 49152 (since this is the decimal equivalent of C000), the start of the program, we don't end up back in the monitor again, but remain in Basic.

Line by line then:

Clear the carry flag.
Clear the decimal flag (4th bit of the status register, this indicates whether or not arithmetic is to be performed in decimal or binary: the 6510 is happier in decimal, but it doesn't really matter in a program such as this. Why put it in? It's one way of introducing a new command).
Load the accumulator with $86.
Add with carry $86.
Store the result in memory location $0402.
Load the accumulator with $07.
Add with carry $07.

Store the result in memory location $0400.

The result of running this program is that an O appears in the corner of the screen, with an L close by. O is screen character code 15, or $0F, and L is character code 12, or $0C.

Thus our answer is $0F0C.

You may think that we haven't added anything up at all, but just displayed things on the screen.

The answer is put on the screen so that it can be seen, and could just as well be stored in any other two memory locations in the usual MSB - LSB format, where it could have been used as part of a future calculation.

It all depends on how you look at it.

Making comparisons

The ability to make decisions in a machine code program relies a great deal on the ability to make comparisons between numbers, registers, and memory locations.

There are a number of commands which allow us to do this (some we've seen already, some we haven't), and these are:

CPX : ComPare the contents of a memory location with the contents of the X register.

Syntax : CPX $0404

CPY : ComPare the contents of a memory location with the contents of the Y register.

Syntax : CPY $0402

CMP : CoMPare the contents of a memory location with the contents of the accumulator.

Syntax : CMP $0400

When encountering the compare command, for example CPX $0404, the program reads the contents of memory location $0400, subtracts

that from the contents of the X register, and sets various flags depending on the result.

This can be used in various ways, and the following program illustrates just one example:

```
.,  C000 A9 53    LDA #$53
.,  C002 8D 04 04 STA $0404
.,  C005 A2 01    LDX #$01
.,  C007 E8       INX
.,  C008 EC 04 04 CPX $0404
.,  C00B F0 03    BEQ $C010
.,  C00D 4C 07 C0 JMP $C007
.,  C010 8E 00 04 STX $0400
.,  C013 60       RTS
```

This program loads a 'heart' symbol into the accumulator, then stores it at memory location $0404 (the screen). The X register is then loaded with a 1, and incremented.

Next, we compare location $0404 with the contents of the X register, and subtracting that (53) from the X register value (currently 29) does not give us a value of zero.

Hence the Branch if EQual instruction is not obeyed, and the program jumps back to increment X again.

Finally, when X equals 53, the BEQ is obeyed and the result, another heart, is printed out onto the screen.

Simple animation

One of the major uses so far (or at least that's the way the market appears to be heading) for machine code programming on the 64 is to produce some amazing games.

For the present, a couple of simple illustrations will serve to show the power of machine code, and the speed with which animated displays can be moved.

This first program simply prints a row of 255 hearts onto the screen: about 6 ½ lines worth.

You may have to change the background colour to be able to see them,

but the point to note is the sheer speed with which things happen.

```
.,  C000 A9 53    LDA #$53
.,  C002 A2 01    LDX #$01
.,  C004 E8       INX
.,  C005 F0 06    BEQ $C00D
.,  C007 9D 00 04 STA $0400,X
.,  C00A 40 04 C0 JMP $C004
.,  C00D 60       RTS
```

Quite simply, the heart character code is loaded into the accumulator, and a 1 is loaded into the X register, which is then incremented.

A test is made for X being equal to zero, which it will be when it is incremented up from being equal to 255: it flips back to 0 again.

However, until then it isn't zero, and so the contents of the accumulator are stored at location $0400, offset with X, and we jump back to increase X again.

You may have been puzzled by the different natures of jumping and branching commands: well, jumping usually uses direct addresses (you tell it which location to go to), and branching uses relative ones, as we've seen.

Some more animation

A well-known technique when moving things about on the screen is to print something, and then fill the space behind it with a space character, thus obliterating the previous image. Then, move the character on one stage and obliterate the image in the place just moved from, and so on.

The following program accomplishes this in machine code, but is so fast that you won't be able to see what's happening!

Later on we'll look at program timing, and having said very early on that one of the chief advantages of machine code is its speed of operation, we'll also look at ways of slowing programs down!

```
.,  C000 A9 53    LDA #$53
.,  C002 A2 00    LDX #$00
.,  C004 A0 20    LDY #$20
.,  C006 8C 25 C0 STY $C025
.,  C009 8D 26 C0 STA $C026
.,  C00C 9D 00 04 STA $0400,X
.,  C00F 98       TYA
.,  C010 9D FF 03 STA $03FF,X
.,  C013 E8       INX
.,  C014 EA       NOP
.,  C015 EA       NOP
.,  C016 EA       NOP
.,  C017 EA       NOP
.,  C018 EA       NOP
.,  C019 EA       NOP
.,  C020 D0 ED    BNE $C00C
.,  C022 60       RTS
```

This program puts a heart into the accumulator, a zero into X, and the code for a space into Y. Y is then stored in memory location $C025, and the contents of the accumulator in $C026: both safely out of the way.

The accumulator contents are then stored at location $0400, offset with X, the contents of the Y register transferred to the accumulator and then stored at $03FF (i.e. a space is stored one memory location (or one screen 'square') behind the heart each time), and the X register incremented.

When X tops 255 the BNE instruction fails as X is flipped back to zero, and the proram exits. Until then, we loop back and print out more hearts and spaces.

This technique is the basis for almost all the screen animation displays that you see in the popular arcade games.

And the NOPs? They just tell the computer to do nothing for 2 cycles. They're there so that, after reading the next section on timing, you can come back to this program and alter it so that you can actually see what's going on!

Timing

The appendix lists all the mnemonics used in machine code for you, and with each one you'll see that we've included the cycle times. In

other words, you know how long each instruction will take to execute.

That is why the previous program is impossible to watch: it takes some 30 cycles, or micro-seconds, to print and then overwrite a heart shape - that's pretty fast, and much too fast for the eye to see.

So the use of NOPs, as mentioned in the last program, can slow us down a little, but 2 micro-seconds isn't exactly a long time. Thus we have to build up various delay programs, and the simplest one must surely be:

```
LDX #$01    - 2 cycles
INX         - 2 cycles
BNE back to INX again. - 3 cycles
```

This just loads a 1 into the X register, checks to see if X is zero, which it will be after flipping over from 255 to zero again, and if it isn't going back and incrementing X again.

However, this takes up a huge 1277 cycles, which still isn't very long. Thus we have to extend the program a little, rather like this:

```
LDY #$01    - 2 cycles
LDX #$01    - 2 cycles
INX         - 2 cycles
BNE         - 3 cycles (go back and increase X again until
              it equals zero)
INY         - 2 cycles
BNE         - 3 cycles (go and increment Y again).
```

Thus we run through our original delay loop 255 times, which gives us a realistic delay of slightly over a quarter of a second. A little better.

There are other methods of course, but if we're going to use the 6510 for precise timing of instruments, program control or whatever, it is obviously better to use this timer than the jiffy clock. Also, every micro-second counts, so use the table in the appendix: some operations take longer under different circumstances.

Charget and the Interrupt

These are the names given to two of the most important functions within the computer.

Charget

Charget, short for CHARacter GET, is a machine code program that resides in page zero of the Commodore 64: in fact it sits in locations 115 through 138. Thus it is quite a short routine. However, it is also a very useful one, as it provides the link from BASIC to the Interpreter.

Charget acts as follows:

When a BASIC program is running, each program line is copied from the RAM area into which you typed it, into the BASIC input buffer. There the charget routine scans through it (ignoring spaces, so you could modify charget to remove the code that checks for spaces, which will make Basic run a bit faster, but does mean that you can't type in spaces any more!), until it finds a byte that it knows. This is then remembered in the accumulator, and program execution returns to the interpreter, where the byte is dealt with.

It is by modifying this charget routine that new commands can be added to Basic, and we'll be looking at this in more detail in chapter 11.

Using the Assembler, and disassembling the code from locations 115 to 138, allows you to do this, but be careful : charget is operating all the time, so any changes will have to make sense to it.

The Interrupt

An Interrupt is precisely what it says it is: an interruption. Computers, like humans, don't like being interrupted on occasions, and so a number of commands in the 6510 instruction set allow you to switch off, and switch back on again, any interrupts.

These commands are:

SEI : SEt Interrupt disable. This stops all interruptions (i.e. stop keys, external devices and the like, although it can't cover everything).

CLI : CLear Interrupt flag. This resets everything.

RTI : ReTurn from Interrupt.

Now, just to complicate things a little further, the 6510 has two different input pins. The NMI, or Non Maskable Input pin cannot be blocked, but the IRQ, or Interrupt ReQuest pin can: it is this one that is set or

cleared with SEI and CLI, which are themselves only operating on the Interrupt flag, the third bit of the status register.

The interrupt procedure, from which RTI returns you, is only instigated in the case of an IRQ interrupt, whereupon the 64 makes a good attempt at keeping everything like it was before the interrupt happened.

BRK: BReaK.

This starts up another interrupt sequence, and when used in machine code programs that are running on a machine with a monitor (e.g. a 64 with the assembler working), BRK will halt program operation and drop you into the monitor, whereupon you will get the usual display of PC, IRQ, XR, YR and so on, with PC as usual displaying the address of the current statement.

7
Machine Code: Goodbye to Fergus

Introduction

Fergus has been with us for some time now, but since the remaining chapters in this book are concerned with some of the more serious things that can be done with machine code, it's time to bid him farewell with this final listing of the program.

As you can see from the length of the listing given, our program has grown a bit from the days when it would do nothing more than just move a sprite around the screen while under machine code control. The rudiments of the original are still there, but now there all sorts of new routines built into it.

Keeping track of the high score, determining when you've lost all three of your lives, and so on, are all essential features for a proper arcade game, and they're all in here somewhere, as we shall shortly see. It is by no means a finished game, but you should by now (and certainly by the end of the book) be in a position to put the little finishing touches to it yourself. As it stands, when all three lives have gone, the program just loops back to the beginning again, after checking to see whether the high score needs updating or not. You could, if you like, drop back into BASIC again at that point and put in a request for playing a new game.

Since the program doesn't occupy that much memory (barely a couple of K) there isn't much to it. But at least it does show how one would go about building up an all machine code game, incorporating such features as scrolling screen displays, a variety of enemies, and so on.

Rather than give the lengthy descriptions we have done before, when the code was sometimes analysed on a byte by byte basis, we'll simply

split the listing up into its different sections and tell you what each of those sections does. You can probably sort out what's going on for yourself by now.

As a bonus, there's a BASIC (aarrgghh!) program at the end of this chapter, which will check itself and then load a machine code disassembler into memory, provided that you follow the instructions CAREFULLY!! But for now, back to Fergus, and the first part of the listing.

```
B*
   PC  SR AC XR YR SP
.;371A 33 00 02 00 F6
.
C000 A9 0D      LDA #$0D
C002 8D F8 07   STA $07F8
C005 A9 03      LDA #$03
C007 8D 15 D0   STA $D015
C00A A9 0E      LDA #$0E
C00C 8D F9 07   STA $07F9
C00F 20 00 C3   JSR $C300
C012 A9 00      LDA #$00
C014 8D 17 D0   STA $D017
C017 8D 1D D0   STA $D01D
C01A A9 00      LDA #$00
C01C 8D 10 D0   STA $D010
C01F EA         NOP
C020 EA         NOP
C021 4C 00 C5   JMP $C500
C024 AD C5 00   LDA $00C5
C027 C9 0A      CMP #$0A
C029 F0 1C      BEQ $C047
C02B C9 12      CMP #$12
C02D F0 42      BEQ $C071
C02F C9 21      CMP #$21
C031 F0 6D      BEQ $C0A0
C033 C9 24      CMP #$24
C035 F0 7C      BEQ $C0B3
C037 C9 3C      CMP #$3C
C039 F0 09      BEQ $C044
C03B 20 5E C6   JSR $C65E
C03E 20 00 C1   JSR $C100
C041 4C 21 C0   JMP $C021
C044 20 C7 C0   JSR $C0C7
C047 A9 00      LDA #$00
C049 CD 10 D0   CMP $D010
C04C D0 0E      BNE $C05C
C04E E0 18      CPX #$18
C050 F0 01      BEQ $C053
C052 CA         DEX
```

```
C053 4C 16 C4    JMP $C416
C056 8E 00 D0    STX $D000
C059 4C 21 C0    JMP $C021
C05C E0 00       CPX #$00
C05E F0 03       BEQ $C063
C060 4C 52 C0    JMP $C052
C063 A9 00       LDA #$00
C065 8D 10 D0    STA $D010
C068 A2 FF       LDX #$FF
C06A 8E 00 D0    STX $D000
C06D 4C 21 C0    JMP $C021
C070 00          BRK
C071 E0 FF       CPX #$FF
C073 D0 03       BNE $C078
C075 4C 21 C0    JMP $C021
C078 E8          INX
C079 4C 00 C4    JMP $C400
C07C 8E 00 D0    STX $D000
C07F A9 00       LDA #$00
C081 CD 10 D0    CMP $D010
C084 D0 11       BNE $C097
C086 E0 FF       CPX #$FF
C088 D0 EB       BNE $C075
C08A A9 07       LDA #$07
C08C 8D 10 D0    STA $D010
C08F A2 01       LDX #$01
C091 8E 00 D0    STX $D000
C094 4C 21 C0    JMP $C021
C097 E0 40       CPX #$40
C099 D0 DA       BNE $C075
C09B CA          DEX
C09C 4C 21 C0    JMP $C021
C09F FF          ???
C0A0 C0 32       CPY #$32
C0A2 D0 03       BNE $C0A7
C0A4 4C 21 C0    JMP $C021
C0A7 88          DEY
C0A8 20 00 C1    JSR $C100
C0AB 8C 01 D0    STY $D001
C0AE 4C 21 C0    JMP $C021
C0B1 00          BRK
C0B2 00          BRK
C0B3 C0 E5       CPY #$E5
C0B5 D0 03       BNE $C0BA
C0B7 4C 21 C0    JMP $C021
C0BA C8          INY
C0BB 20 00 C1    JSR $C100
C0BE 8C 01 D0    STY $D001
C0C1 4C 21 C0    JMP $C021
C0C4 FF          ???
C0C5 FF          ???
```

```
C0C6 FF          ???
C0C7 A9 3F       LDA #$3F
C0C9 8D 15 D0    STA $D015
C0CC A9 21       LDA #$21
C0CE 8D 04 D4    STA $D404
C0D1 8C 04 C2    STY $C204
C0D4 8E 02 D0    STX $D002
C0D7 8C 03 D0    STY $D003
C0DA C0 10       CPY #$10
C0DC F0 13       BEQ $C0F1
C0DE 88          DEY
C0DF 8C 03 D0    STY $D003
C0E2 20 00 C1    JSR $C100
C0E5 20 40 C6    JSR $C640
C0E8 20 00 C6    JSR $C600
C0EB 20 00 C1    JSR $C100
C0EE 4C DA C0    JMP $C0DA
C0F1 AC 04 C2    LDY $C204
C0F4 A9 81       LDA #$81
C0F6 8D 04 D4    STA $D404
C0F9 A9 3D       LDA #$3D
C0FB 8D 15 D0    STA $D015
C0FE 60          RTS
C0FF FF          ???
C100 8E 00 C2    STX $C200
C103 A2 FF       LDX #$FF
C105 CA          DEX
C106 D0 FD       BNE $C105
C108 20 5E C6    JSR $C65E
C10B 8C 01 D4    STY $D401
C10E 20 5E C6    JSR $C65E
C111 AE 00 C2    LDX $C200
C114 20 5E C6    JSR $C65E
C117 20 5E C6    JSR $C65E
C11A 20 5E C6    JSR $C65E
C11D 20 5E C6    JSR $C65E
C120 20 5E C6    JSR $C65E
C123 20 5E C6    JSR $C65E
C126 20 5E C6    JSR $C65E
C129 20 5E C6    JSR $C65E
C12C 60          RTS
C12D 00          BRK
```

This is the routine that handles moving left, right, up and down, as well as firing. The main difference now is in the delay loop from locations $C100 to $C12C, which goes off to check for a collision as often as possible (the routine starting at $C65E).

```
C2FF 00        BRK
C300 A9 0F     LDA #$0F
C302 8D 18 D4  STA $D418
C305 A9 22     LDA #$22
C307 8D 05 D4  STA $D405
C30A A9 86     LDA #$86
C30C 8D 06 D4  STA $D406
C30F A9 81     LDA #$81
C311 8D 04 D4  STA $D404
C314 A9 00     LDA #$00
C316 8D 20 D0  STA $D020
C319 8D 21 D0  STA $D021
C31C A2 A0     LDX #$A0
C31E 8E 00 D0  STX $D000
C321 A0 A0     LDY #$A0
C323 8C 01 D0  STY $D001
C326 A9 00     LDA #$00
C328 8D 03 D0  STA $D003
C32B 8D 05 D0  STA $D005
C32E 8D 07 D0  STA $D007
C331 8D 09 D0  STA $D009
C334 8D 0B D0  STA $D00B
C337 8D 0D D0  STA $D00D
C33A A9 01     LDA #$01
C33C 8D 27 D0  STA $D027
C33F 8D 02 D0  STA $D002
C342 A9 07     LDA #$07
C344 8D 28 D0  STA $D028
C347 A9 00     LDA #$00
C349 8D 00 C7  STA $C700
C34C A9 30     LDA #$30
C34E 8D 00 C8  STA $C800
C351 8D 01 C8  STA $C801
C354 8D 02 C8  STA $C802
C357 8D 03 C8  STA $C803
C35A 8D 04 C8  STA $C804
C35D 8D 05 C8  STA $C805
C360 8D 00 04  STA $0400
C363 8D 01 04  STA $0401
C366 8D 02 04  STA $0402
C369 8D 03 04  STA $0403
C36C 8D 04 04  STA $0404
C36F 8D 05 04  STA $0405
C372 A9 01     LDA #$01
C374 8D 00 D8  STA $D800
C377 8D 01 D8  STA $D801
C37A 8D 02 D8  STA $D802
C37D 8D 03 D8  STA $D803
C380 8D 04 D8  STA $D804
C383 8D 05 D8  STA $D805
C386 A9 F9     LDA #$F9
```

```
C388 8D FF CC     STA $CCFF
C38B EA           NOP
C38C A9 30        LDA #$30
C38E 8D 08 C8     STA $C808
C391 8D 09 C8     STA $C809
C394 8D 0A C8     STA $C80A
C397 8D 0B C8     STA $C80B
C39A 8D 0C C8     STA $C80C
C39D 8D 0D C8     STA $C80D
C3A0 8D 27 04     STA $0427
C3A3 8D 26 04     STA $0426
C3A6 8D 25 04     STA $0425
C3A9 8D 24 04     STA $0424
C3AC 8D 23 04     STA $0423
C3AF 8D 22 04     STA $0422
C3B2 A9 01        LDA #$01
C3B4 8D 22 D8     STA $D822
C3B7 8D 23 D8     STA $D823
C3BA 8D 24 D8     STA $D824
C3BD 8D 25 D8     STA $D825
C3C0 8D 26 D8     STA $D826
C3C3 8D 27 D8     STA $D827
C3C6 8D 14 D8     STA $D814
C3C9 A9 33        LDA #$33
C3CB 8D 14 04     STA $0414
C3CE 8D FF CB     STA $CBFF
C3D1 60           RTS
C3D2 FF           ???
```

Considerably lengthened since last time, there aren't really that many changes to this. It sets up the sound and some sprite positions as before, but the bulk of the code is now concerned with storing a zero in the memory locations that we'll use to keep the current score ($C400 to $C405) and the high score ($C808 to $C80D), as well as storing on the screen a collection of zeroes next to the display that says high score and score. These scores are then given a colour by storing a 1 in colour memory at the appropriate points.

The sprite pointer to let the computer know which enemy sprite is to be displayed is stored at location $CCFF and the number of lives left is stored at location $CBFF before returning from this subroutine. Remember, when dealing with zeroes or whatever being stored in locations, we have to use the CHR$ codes (which happens to be a 30 for the number 0).

```
C3FF 00        BRK
C400 8E 12 C4  STX $C412
C403 8C 13 C4  STY $C413
C406 20 00 CA  JSR $CA00
C409 AE 12 C4  LDX $C412
C40C AC 13 C4  LDY $C413
C40F 4C 7C C0  JMP $C07C
C412 AD E5 FF  LDA $FFE5
C415 FF        ???
C416 8E 12 C4  STX $C412
C419 8C 13 C4  STY $C413
C41C 20 30 CA  JSR $CA30
C41F AE 12 C4  LDX $C412
C422 AC 13 C4  LDY $C413
C425 4C 56 C0  JMP $C056
C428 00        BRK
C429 80        ???
C42A 00        BRK
C42B 00        BRK
C42C 00        BRK
C42D 00        BRK
C42E 00        BRK
```

A small piece of code that we've seen before. This looks after the X and Y registers while scrolling the screen around.

```
C500 A9 3D     LDA #$3D
C502 8E 00 C2  STX $C200
C505 8C 02 C2  STY $C202
C508 CD 15 D0  CMP $D015
C50B F0 2E     BEQ $C53B
C50D A0 06     LDY #$06
C50F A9 FB     LDA #$FB
C511 AD FF CC  LDA $CCFF
C514 99 F9 07  STA $07F9,Y
C517 A9 3D     LDA #$3D
C519 8D 15 D0  STA $D015
C51C AD A2 00  LDA $00A2
C51F 8D 04 D0  STA $D004
C522 69 30     ADC #$30
C524 8D 06 D0  STA $D006
C527 69 90     ADC #$90
C529 8D 08 D0  STA $D008
C52C 69 D0     ADC #$D0
C52E 8D 0A D0  STA $D00A
C531 A9 32     LDA #$32
C533 8D 05 D0  STA $D005
```

```
C536 88          DEY
C537 C0 00       CPY #$00
C539 D0 D4       BNE $C50F
C53B AE 00 C2    LDX $C200
C53E AC 02 C2    LDY $C202
C541 20 00 C6    JSR $C600
C544 4C 00 C9    JMP $C900
C547 4C 24 C0    JMP $C024
C54A FF          ???
```

Another old friend, for updating the enemy sprites if necessary.

```
C5FF 00          BRK
C600 8E 00 C2    STX $C200
C603 8C 02 C2    STY $C202
C606 20 40 C6    JSR $C640
C609 AC 05 D0    LDY $D005
C60C C8          INY
C60D C0 E8       CPY #$E8
C60F D0 10       BNE $C621
C611 A9 03       LDA #$03
C613 8D 15 D0    STA $D015
C616 AE 00 C2    LDX $C200
C619 AC 02 C2    LDY $C202
C61C 20 5E C6    JSR $C65E
C61F 60          RTS
C620 00          BRK
C621 8C 05 D0    STY $D005
C624 8C 07 D0    STY $D007
C627 8C 09 D0    STY $D009
C62A 8C 0B D0    STY $D00B
C62D 20 00 C1    JSR $C100
C630 20 00 C1    JSR $C100
C633 20 00 C1    JSR $C100
C636 20 00 C1    JSR $C100
C639 AE 00 C2    LDX $C200
C63C AC 02 C2    LDY $C202
C63F 60          RTS
C640 AC 04 C2    LDY $C204
C643 20 5E C6    JSR $C65E
C646 AD C5 00    LDA $00C5
C649 EA          NOP
C64A C9 0A       CMP #$0A
C64C F0 08       BEQ $C656
C64E C9 12       CMP #$12
C650 F0 04       BEQ $C656
C652 AC 03 D0    LDY $D003
C655 60          RTS
C656 A9 00       LDA #$00
```

```
C658 8D 03 D0     STA $D003
C65B A0 10        LDY #$10
C65D 60           RTS
C65E A9 04        LDA #$04
C660 CD 1E D0     CMP $D01E
C663 30 01        BMI $C666
C665 60           RTS
C666 EA           NOP
C667 EA           NOP
C668 EA           NOP
C669 EA           NOP
C66A EA           NOP
C66B EA           NOP
C66C AD 1E D0     LDA $D01E
C66F 29 01        AND #$01
C671 D0 0A        BNE $C67D
C673 AD 1E D0     LDA $D01E
C676 29 02        AND #$02
C678 D0 16        BNE $C690
C67A 60           RTS
C67B EA           NOP
C67C EA           NOP
C67D A9 03        LDA #$03
C67F CD 1E D0     CMP $D01E
C682 F0 08        BEQ $C68C
C684 A9 04        LDA #$04
C686 8D 27 D0     STA $D027
C689 4C 00 CC     JMP $CC00
C68C 4C 24 C0     JMP $C024
C68F EA           NOP
C690 A9 03        LDA #$03
C692 CD 1E D0     CMP $D01E
C695 F0 F5        BEQ $C68C
C697 8E 08 C2     STX $C208
C69A 20 AA C6     JSR $C6AA
C69D 20 00 CD     JSR $CD00
C6A0 4C 14 CD     JMP $CD14
C6A3 AE 08 C2     LDX $C208
C6A6 60           RTS
C6A7 EA           NOP
C6A8 EA           NOP
C6A9 EA           NOP
C6AA AE 00 C8     LDX $C800
C6AD E8           INX
C6AE EA           NOP
C6AF E0 39        CPX #$39
C6B1 10 0C        BPL $C6BF
C6B3 8E 00 C8     STX $C800
C6B6 8E 05 04     STX $0405
C6B9 A9 01        LDA #$01
C6BB 8D 05 D8     STA $D805
```

```
C6BE 60          RTS
C6BF A2 30       LDX #$30
C6C1 8E 00 C8    STX $C800
C6C4 AE 01 C8    LDX $C801
C6C7 E8          INX
C6C8 EA          NOP
C6C9 E0 39       CPX #$39
C6CB 10 0C       BPL $C6D9
C6CD 8E 01 C8    STX $C801
C6D0 8E 04 04    STX $0404
C6D3 A9 01       LDA #$01
C6D5 8D 04 D8    STA $D804
C6D8 60          RTS
C6D9 A2 30       LDX #$30
C6DB 8E 01 C8    STX $C801
C6DE AE 02 C8    LDX $C802
C6E1 E8          INX
C6E2 EA          NOP
C6E3 E0 39       CPX #$39
C6E5 10 0C       BPL $C6F3
C6E7 8E 02 C8    STX $C802
C6EA 8E 03 04    STX $0403
C6ED A9 01       LDA #$01
C6EF 8D 03 D8    STA $D803
C6F2 60          RTS
C6F3 A2 30       LDX #$30
C6F5 8E 02 C8    STX $C802
C6F8 AE 03 C8    LDX $C803
C6FB E8          INX
C6FC EA          NOP
C6FD E0 39       CPX #$39
C6FF 10 00       BPL $C701
C701 8E 03 C8    STX $C803
C704 8E 00 04    STX $0400
C707 A9 01       LDA #$01
C709 8D 02 D8    STA $D802
C70C 60          RTS
C70D A2 30       LDX #$30
C70F 8E 03 C8    STX $C803
C712 AE 04 C8    LDX $C804
C715 E8          INX
C716 EA          NOP
C717 E0 39       CPX #$39
C719 10 0C       BPL $C727
C71B 8E 04 C8    STX $C804
C71E 8E 01 04    STX $0401
C721 A9 01       LDA #$01
C723 8D 01 D8    STA $D801
C726 60          RTS
C727 A2 30       LDX #$30
C729 8E 04 C8    STX $C804
```

```
C72C AE 05 C8     LDX $C805
C72F E8           INX
C730 EA           NOP
C731 E0 39        CPX #$39
C733 10 0C        BPL $C741
C735 8E 05 C8     STX $C805
C738 8E 00 04     STX $0400
C73B A9 01        LDA #$01
C73D 8D 00 D8     STA $D800
C740 60           RTS
C741 A9 30        LDA #$30
C743 8D 05 C8     STA $C805
C746 8D 04 C8     STA $C804
C749 8D 03 C8     STA $C803
C74C 8D 02 C8     STA $C802
C74F 8D 01 C8     STA $C801
C752 8D 00 C8     STA $C800
C755 60           RTS
C756 FF           ???
C757 FF           ???
C758 FF           ???
```

A much extended routine, which updates the sprite positions, checks for collisions, and from $C6AF onwards updates the score and high score displays if necessary.

```
B*
   PC  SR AC XR YR SP
.;26F4 33 00 DC 00 F6
.
C900 AD 00 C8     LDA $C800
C903 8D 05 04     STA $0405
C906 AD 01 C8     LDA $C801
C909 8D 04 04     STA $0404
C90C AD 02 C8     LDA $C802
C90F 8D 03 04     STA $0403
C912 AD 03 C8     LDA $C803
C915 8D 02 04     STA $0402
C918 AD 04 C8     LDA $C804
C91B 8D 01 04     STA $0401
C91E AD 05 C8     LDA $C805
C921 8D 00 04     STA $0400
C924 4C 47 C5     JMP $C547
C927 00           BRK
C928 00           BRK
.
```

A tiny little routine which is constantly called to update your score on the screen.

```
.
CA00 A9 28      LDA #$28
CA02 A2 18      LDX #$18
CA04 85 57      STA $57
CA06 A9 04      LDA #$04
CA08 85 58      STA $58
CA0A A0 00      LDY #$00
CA0C B1 57      LDA ($57),Y
CA0E 85 59      STA $59
CA10 C8         INY
CA11 B1 57      LDA ($57),Y
CA13 88         DEY
CA14 91 57      STA ($57),Y
CA16 C8         INY
CA17 98         TYA
CA18 C9 27      CMP #$27
CA1A D0 F4      BNE $CA10
CA1C A5 59      LDA $59
CA1E 91 57      STA ($57),Y
CA20 A5 57      LDA $57
CA22 18         CLC
CA23 69 28      ADC #$28
CA25 85 57      STA $57
CA27 90 02      BCC $CA2B
CA29 E6 58      INC $58
CA2B CA         DEX
CA2C D0 DC      BNE $CA0A
CA2E 60         RTS
CA2F 00         BRK
CA30 A9 28      LDA #$28
CA32 A2 18      LDX #$18
CA34 85 57      STA $57
CA36 A9 04      LDA #$04
CA38 85 58      STA $58
CA3A A0 27      LDY #$27
CA3C B1 57      LDA ($57),Y
CA3E 85 59      STA $59
CA40 88         DEY
CA41 B1 57      LDA ($57),Y
CA43 C8         INY
CA44 91 57      STA ($57),Y
CA46 88         DEY
CA47 98         TYA
CA48 C9 00      CMP #$00
CA4A D0 F4      BNE $CA40
CA4C A5 59      LDA $59
CA4E 91 57      STA ($57),Y
CA50 A5 57      LDA $57
CA52 18         CLC
CA53 69 28      ADC #$28
CA55 85 57      STA $57
```

```
CA57 90 02      BCC $CA5B
CA59 E6 58      INC $58
CA5B CA         DEX
CA5C D0 DC      BNE $CA3A
CA5E 60         RTS
CA5F FF         ???
.
```

Our old friend the scrolling routine.

```
.
CC00 8E 1A C2   STX $C21A
CC03 8C 1B C2   STY $C21B
CC06 A9 DF      LDA #$DF
CC08 8D F8 07   STA $07F8
CC0B 88         DEY
CC0C 20 4D CC   JSR $CC4D
CC0F 8C 01 D0   STY $D001
CC12 C0 FF      CPY #$FF
CC14 D0 F5      BNE $CC0B
CC16 A9 0D      LDA #$0D
CC18 8D F8 07   STA $07F8
CC1B A9 01      LDA #$01
CC1D 8D 27 D0   STA $D027
CC20 A9 00      LDA #$00
CC22 8D 1E D0   STA $D01E
CC25 A0 E5      LDY #$E5
CC27 8C 01 D0   STY $D001
CC2A A2 A0      LDX #$A0
CC2C EA         NOP
CC2D 8E 00 D0   STX $D000
CC30 8D 10 D0   STA $D010
CC33 8E 08 C2   STX $C208
CC36 A9 3D      LDA #$3D
CC38 8D 15 D0   STA $D015
CC3B EA         NOP
CC3C A9 00      LDA #$00
CC3E 8D 05 D0   STA $D005
CC41 20 5B CC   JSR $CC5B
CC44 A2 A0      LDX #$A0
CC46 EA         NOP
CC47 4C 8C C6   JMP $C68C
CC4A FF         ???
CC4B FF         ???
CC4C FF         ???
CC4D A2 00      LDX #$00
CC4F E8         INX
CC50 E0 FF      CPX #$FF
CC52 D0 FB      BNE $CC4F
CC54 AE 08 C2   LDX $C208
```

```
CC57 20 00 C1     JSR $C100
CC5A 60           RTS
CC5B AE FF CB     LDX $CBFF
CC5E CA           DEX
CC5F E0 30        CPX #$30
CC61 D0 05        BNE $CC68
CC63 4C 00 CE     JMP $CE00
.
```

A new routine, which is used to fetch in the new sprite when our hero gets hit, and whizz him to the top of the screen. The latter part of the routine is checking to see how many lives you have left. If all your lives have gone, jump to the routine at $CE00.

```
.
CD00 AE FF CC     LDX $CCFF
CD03 E8           INX
CD04 E0 FF        CPX #$FF
CD06 F0 04        BEQ $CD0C
CD08 8E FF CC     STX $CCFF
CD0B 60           RTS
CD0C A2 F9        LDX #$F9
CD0E 8E FF CC     STX $CCFF
CD11 4C A0 C6     JMP $C6A0
CD14 38           SEC
CD15 A9 3F        LDA #$3F
CD17 ED 1E D0     SBC $D01E
CD1A 8D 15 D0     STA $D015
CD1D 4C A3 C6     JMP $C6A3
CD20 00           BRK
CD21 00           BRK
.
```

This just flips in the next set of enemy sprites when necessary.

```
.
CE00 AD 05 C8     LDA $C805
CE03 CD 0D C8     CMP $C80D
CE06 F0 04        BEQ $CE0C
CE08 10 53        BPL $CE5D
CE0A D0 3C        BNE $CE48
CE0C AD 04 C8     LDA $C804
CE0F CD 0C C8     CMP $C80C
CE12 F0 04        BEQ $CE18
CE14 10 47        BPL $CE5D
CE16 D0 30        BNE $CE48
CE18 AD 03 C8     LDA $C803
```

```
CE1B CD 0B C8    CMP $C80B
CE1E F0 04       BEQ $CE24
CE20 10 3B       BPL $CE5D
CE22 D0 24       BNE $CE48
CE24 AD 02 C8    LDA $C802
CE27 CD 0A C8    CMP $C80A
CE2A F0 04       BEQ $CE30
CE2C 10 2F       BPL $CE5D
CE2E D0 18       BNE $CE48
CE30 AD 01 C8    LDA $C801
CE33 CD 09 C8    CMP $C809
CE36 F0 04       BEQ $CE3C
CE38 10 23       BPL $CE5D
CE3A D0 0C       BNE $CE48
CE3C AD 00 C8    LDA $C800
CE3F CD 08 C8    CMP $C808
CE42 F0 04       BEQ $CE48
CE44 10 17       BPL $CE5D
CE46 D0 00       BNE $CE48
CE48 A9 30       LDA #$30
CE4A 8D 00 04    STA $0400
CE4D 8D 01 04    STA $0401
CE50 8D 02 04    STA $0402
CE53 8D 03 04    STA $0403
CE56 8D 04 04    STA $0404
CE59 4C 93 CE    JMP $CE93
CE5C 60          RTS
CE5D AD 00 C8    LDA $C800
CE60 8D 08 C8    STA $C808
CE63 8D 27 04    STA $0427
CE66 AD 01 C8    LDA $C801
CE69 8D 09 C8    STA $C809
CE6C 8D 26 04    STA $0426
CE6F AD 02 C8    LDA $C802
CE72 8D 0A C8    STA $C80A
CE75 8D 25 04    STA $0425
CE78 AD 03 C8    LDA $C803
CE7B 8D 0B C8    STA $C80B
CE7E 8D 24 04    STA $0424
CE81 AD 04 C8    LDA $C804
CE84 8D 0C C8    STA $C80C
CE87 8D 23 04    STA $0423
CE8A AD 05 C8    LDA $C805
CE8D 8D 0D C8    STA $C80D
CE90 8D 22 04    STA $0422
CE93 A9 30       LDA #$30
CE95 8D 00 C8    STA $C800
CE98 8D 01 C8    STA $C801
CE9B 8D 02 C8    STA $C802
CE9E 8D 03 C8    STA $C803
CEA1 8D 04 C8    STA $C804
```

```
CEA4 8D 05 C8     STA $C805
CEA7 8D 05 04     STA $0405
CEAA 4C 66 CC     JMP $CC66
CEAD 00           BRK
.
.
```

A check to see if there's a new high score, and update all the scores if there is.

And now, the promised BASIC listing.

Entering and using Extramon

(1) Switch your Commodore 64 off and on again.

(2) Type in the program 'Monmaker'. Please note that, due to the way the program was formatted before printing, lines 500,502,504 and 1450 have wrapped around on the printout. For instance, line 500 has a series of data statements ... 7068,7420,7431,6285 ... Enter these lines on your machine as just very long individual lines.

(3) To check that the data has been entered correctly, 'Monmaker' includes a checksum program. Alter line 1425 to read:

```
1425 READA:B=B+A:NEXTJ
```

Then RUN the program.

The program steps through all the blocks of data, and if it finds any errors it will identify the group of lines in which the error occurs. You'll have to check lines 1400- yourself.

(4) SAVE the program when entered correctly, remembering to change line 1425.

(5) Switch the 64 off and on again.

(6) Enter the commands: POKE 8192,0:POKE44,32:NEW <RETURN>

(7) LOAD and RUN 'Monmaker'. It takes some time to run, so be patient.

(8) Enter the commands: POKE44,08:POKE45,235:POKE46,17:CLR <RETURN>

(9) Type SYS 2168. You are now 'into' Extramon, and can save the program using its own SAVE command (see later).

Extramon command set summary

These will be given in the form COMMAND, followed by an example syntax.

*Simple assembler. .A C000 LDA #$12
.A C002 STA $8000,X
.A C005 RTS
The user starts assembling at $C000, and after entering the first line the assembler prompts for the next one.

*Disassembler .D C000
Clear the screen and print a page of disassembled output, starting at $C000.

*Printing Disassembler .P C000,C200
Send disassembled code to the printer. Engage the printer beforehand with OPEN4,4:CMD4 and enter the monitor with SYS 2168.

*Fill memory .F C000 C100 FF
Fill the block of memory from $C000 to $C100 with the byte $FF.

*Go run .G C000
Go to address $C000 and execute the code found there.

*Hunt memory .H C000 C100 'READ
Hunt through memory from $C000 to $C100 for the ASCII string READ, and print out the location(s) where it was found.

*Load .L "FRED",08
Load the program FRED from device 8 (or device 1, or device ...)

*Memory display .M C000 C100
Display the bytes of memory from $C000 to $C100. These can then be altered by typing over them and hitting RETURN.

*Register display .R
Display the state of the registers when the monitor was first entered.

*Save .S "FRED",01,C000,C100
Save the block of memory from $C000 to $C100 onto device 01, and call it FRED.

*Transfer memory .T C000 C100 C500
Transfer memory in the range $C000 to $C100 and start storing it at $C500.

*Exit to Basic .X
Return to Basic ready mode.

```
MONMAKER - BASIC LOADER FOR DISASSEMBLER

500 DATA6395,6669,5692,7248,7628,6783,7068,7420,74
31,6285,8644,7437,8180
502 DATA6918,5441,6366,8341,6710,6920,6460,6755,71
09,6851,6799,7710,6215
504 DATA7172,7043,6911,6860,7882,7656,7607,7689,58
88,3983,4118,5337,6028,7231
1001 DATA169,11,141,32,208,141
1002 DATA33,208,165,45,133,34
1003 DATA165,46,133,35,165,55
1004 DATA133,36,165,56,133,37
1005 DATA160,0,165,34,208,2
1006 DATA198,35,198,34,177,34
1007 DATA208,60,165,34,208,2
1008 DATA198,35,198,34,177,34
1009 DATA240,33,133,38,165,34
1010 DATA208,2,198,35,198,34
1011 DATA177,34,24,101,36,170
1012 DATA165,38,101,37,72,165
1013 DATA55,208,2,198,56,198
1014 DATA55,104,145,55,138,72
1015 DATA165,55,208,2,198,56
1016 DATA198,55,104,145,55,24
1017 DATA144,182,201,79,208,237
1018 DATA165,55,133,51,165,56
1019 DATA133,52,108,55,0,79
1020 DATA79,79,79,173,230,255
1021 DATA0,141,22,3,173,231
1022 DATA255,0,141,23,3,169
1023 DATA128,32,144,255,0,0
1024 DATA216,104,141,62,2,104
1025 DATA141,61,2,104,141,60
1026 DATA2,104,141,59,2,104
1027 DATA170,104,168,56,138,233
1028 DATA2,141,58,2,152,233
1029 DATA0,0,141,57,2,186
1030 DATA142,63,2,32,87,253
1031 DATA0,162,66,169,42,32
1032 DATA87,250,0,169,82,208
```

```
1033 DATA52,230,193,208,6,230
1034 DATA194,208,2,230,38,96
1035 DATA32,207,255,201,13,208
1036 DATA248,104,104,169,144,32
1037 DATA210,255,169,0,0,133
1038 DATA38,162,13,169,46,32
1039 DATA87,250,0,169,5,32
1040 DATA210,255,32,62,248,0
1041 DATA201,46,240,249,201,32
1042 DATA240,245,162,14,221,183
1043 DATA255,0,208,12,138,10
1044 DATA170,189,199,255,0,72
1045 DATA189,198,255,0,72,96
1046 DATA202,16,236,76,237,250
1047 DATA0,165,193,141,58,2
1048 DATA165,194,141,57,2,96
1049 DATA169,8,133,29,160,0
1050 DATA0,32,84,253,0,177
1051 DATA193,32,72,250,0,32
1052 DATA51,248,0,198,29,208
1053 DATA241,96,32,136,250,0
1054 DATA144,11,162,0,0,129
1055 DATA193,193,193,240,3,76
1056 DATA237,250,0,32,51,248
1057 DATA0,198,29,96,169,59
1058 DATA133,193,169,2,133,194
1059 DATA169,5,96,152,72,32
1060 DATA87,253,0,104,162,46
1061 DATA76,87,250,0,169,144
1062 DATA32,210,255,162,0,0
1063 DATA189,234,255,0,32,210
1064 DATA255,232,224,22,208,245
1065 DATA160,59,32,194,248,0
1066 DATA173,57,2,32,72,250
1067 DATA0,173,58,2,32,72
1068 DATA250,0,32,183,248,0
1069 DATA32,141,248,0,240,92
1070 DATA32,62,248,0,32,121
1071 DATA250,0,144,51,32,105
1072 DATA250,0,32,62,248,0
1073 DATA32,121,250,0,144,40
1074 DATA32,105,250,0,169,144
1075 DATA32,210,255,32,225,255
1076 DATA240,60,166,38,208,56
1077 DATA165,195,197,193,165,196
1078 DATA229,194,144,46,160,58
1079 DATA32,194,248,0,32,65
1080 DATA250,0,32,139,248,0
1081 DATA240,224,76,237,250,0
1082 DATA32,121,250,0,144,3
1083 DATA32,128,248,0,32,183
```

```
1084 DATA248,0,208,7,32,121
1085 DATA250,0,144,235,169,8
1086 DATA133,29,32,62,248,0
1087 DATA32,161,248,0,208,248
1088 DATA76,71,248,0,32,207
1089 DATA255,201,13,240,12,201
1090 DATA32,208,209,32,121,250
1091 DATA0,144,3,32,128,248
1092 DATA0,169,144,32,210,255
1093 DATA174,63,2,154,120,173
1094 DATA57,2,72,173,58,2
1095 DATA72,173,59,2,72,173
1096 DATA60,2,174,61,2,172
1097 DATA62,2,64,169,144,32
1098 DATA210,255,174,63,2,154
1099 DATA108,2,160,160,1,132
1100 DATA186,132,185,136,132,183
1101 DATA132,144,132,147,169,64
1102 DATA133,187,169,2,133,188
1103 DATA32,207,255,201,32,240
1104 DATA249,201,13,240,56,201
1105 DATA34,208,20,32,207,255
1106 DATA201,34,240,16,201,13
1107 DATA240,41,145,187,230,183
1108 DATA200,192,16,208,236,76
1109 DATA237,250,0,32,207,255
1110 DATA201,13,240,22,201,44
1111 DATA208,220,32,136,250,0
1112 DATA41,15,240,233,201,3
1113 DATA240,229,133,186,32,207
1114 DATA255,201,13,96,108,48
1115 DATA3,108,50,3,32,150
1116 DATA249,0,208,212,169,144
1117 DATA32,210,255,169,0,0
1118 DATA32,239,249,0,165,144
1119 DATA41,16,208,196,76,71
1120 DATA248,0,32,150,249,0
1121 DATA201,44,208,186,32,121
1122 DATA250,0,32,105,250,0
1123 DATA32,207,255,201,44,208
1124 DATA173,32,121,250,0,165
1125 DATA193,133,174,165,194,133
1126 DATA175,32,105,250,0,32
1127 DATA207,255,201,13,208,152
1128 DATA169,144,32,210,255,32
1129 DATA242,249,0,76,71,248
1130 DATA0,165,194,32,72,250
1131 DATA0,165,193,72,74,74
1132 DATA74,74,32,96,250,0
1133 DATA170,104,41,15,32,96
1134 DATA250,0,72,138,32,210
```

```
1135 DATA255,104,76,210,255,9
1136 DATA48,201,58,144,2,105
1137 DATA6,96,162,2,181,192
1138 DATA72,181,194,149,192,104
1139 DATA149,194,202,208,243,96
1140 DATA32,136,250,0,144,2
1141 DATA133,194,32,136,250,0
1142 DATA144,2,133,193,96,169
1143 DATA0,0,133,42,32,62
1144 DATA248,0,201,32,208,9
1145 DATA32,62,248,0,201,32
1146 DATA208,14,24,96,32,175
1147 DATA250,0,10,10,10,10
1148 DATA133,42,32,62,248,0
1149 DATA32,175,250,0,5,42
1150 DATA56,96,201,58,144,2
1151 DATA105,8,41,15,96,162
1152 DATA2,44,162,0,0,180
1153 DATA193,208,8,180,194,208
1154 DATA2,230,38,214,194,214
1155 DATA193,96,32,62,248,0
1156 DATA201,32,240,249,96,169
1157 DATA0,0,141,0,0,1
1158 DATA32,204,250,0,32,143
1159 DATA250,0,32,124,250,0
1160 DATA144,9,96,32,62,248
1161 DATA0,32,121,250,0,176
1162 DATA222,174,63,2,154,169
1163 DATA144,32,210,255,169,63
1164 DATA32,210,255,76,71,248
1165 DATA0,32,84,253,0,202
1166 DATA208,250,96,230,195,208
1167 DATA2,230,196,96,162,2
1168 DATA181,192,72,181,39,149
1169 DATA192,104,149,39,202,208
1170 DATA243,96,165,195,164,196
1171 DATA56,233,2,176,14,136
1172 DATA144,11,165,40,164,41
1173 DATA76,51,251,0,165,195
1174 DATA164,196,56,229,193,133
1175 DATA30,152,229,194,168,5
1176 DATA30,96,32,212,250,0
1177 DATA32,105,250,0,32,229
1178 DATA250,0,32,12,251,0
1179 DATA32,229,250,0,32,47
1180 DATA251,0,32,105,250,0
1181 DATA144,21,166,38,208,100
1182 DATA32,40,251,0,144,95
1183 DATA161,193,129,195,32,5
1184 DATA251,0,32,51,248,0
1185 DATA208,235,32,40,251,0
```

```
1186 DATA24,165,30,101,195,133
1187 DATA195,152,101,196,133,196
1188 DATA32,12,251,0,166,38
1189 DATA208,61,161,193,129,195
1190 DATA32,40,251,0,176,52
1191 DATA32,184,250,0,32,187
1192 DATA250,0,76,125,251,0
1193 DATA32,212,250,0,32,105
1194 DATA250,0,32,229,250,0
1195 DATA32,105,250,0,32,62
1196 DATA248,0,32,136,250,0
1197 DATA144,20,133,29,166,38
1198 DATA208,17,32,47,251,0
1199 DATA144,12,165,29,129,193
1200 DATA32,51,248,0,208,238
1201 DATA76,237,250,0,76,71
1202 DATA248,0,32,212,250,0
1203 DATA32,105,250,0,32.229
1204 DATA250,0,32,105,250,0
1205 DATA32,62,248.0,162.0
1206 DATA0,32,62,248,0,201
1207 DATA39,208.20.32,62,248
1208 DATA0,157,16,2,232,32
1209 DATA207.255,201,13,240,34
1210 DATA224,32,208,241,240,28
1211 DATA142,0,0,1,32,143
1212 DATA250,0,144,198,157,16
1213 DATA2,232,32,207,255,201
1214 DATA13,240,9,32,136,250
1215 DATA0,144,182,224,32,208
1216 DATA236,134,28,169,144,32
1217 DATA210,255,32,87,253,0
1218 DATA162,0,0,160,0,0
1219 DATA177,193,221,16,2,208
1220 DATA12,200,232,228,28,208
1221 DATA243,32,65,250,0,32
1222 DATA84,253,0,32,51,248
1223 DATA0,166,38,208,141,32
1224 DATA47,251,0,176,221,76
1225 DATA71,248,0,32,212,250
1226 DATA0,133,32,165,194,133
1227 DATA33,162,0,0,134,40
1228 DATA169,147,32,210,255,169
1229 DATA144,32,210,255,169,22
1230 DATA133,29,32,106,252,0
1231 DATA32,202,252,0,133,193
1232 DATA132,194,198,29,208,242
1233 DATA169,145,32,210,255,76
1234 DATA71,248,0,160,44,32
1235 DATA194,248,0,32,84,253
1236 DATA0,32,65,250,0,32
```

```
1237 DATA84,253,0,162,0,0
1238 DATA161,193,32,217,252.0
1239 DATA72,32,31,253,0.104
1240 DATA32,53,253,0,162,6
1241 DATA224,3,208.18,164,31
1242 DATA240,14,165,42.201,232
1243 DATA177,193,176,28,32,194
1244 DATA252,0,136,208,242.6
1245 DATA42,144.14,189,42,255
1246 DATA0,32,165,253,0,189
1247 DATA48,255,0,240,3,32
1248 DATA165,253,0.202,208,213
1249 DATA96,32,205,252,0,170
1250 DATA232,208,1,200,152.32
1251 DATA194,252.0.138,134,28
1252 DATA32,72,250,0,166,28
1253 DATA96,165,31,56,164,194
1254 DATA170,16,1,136,101,193
1255 DATA144,1,200,96,168,74
1256 DATA144,11,74,176,23,201
1257 DATA34,240,19,41,7,9
1258 DATA128,74,170,189,217,254
1259 DATA0,176,4,74,74,74
1260 DATA74,41,15,208,4,160
1261 DATA128,169,0,0,170,189
1262 DATA29,255,0,133,42,41
1263 DATA3,133,31,152,41,143
1264 DATA170,152,160,3,224,138
1265 DATA240,11,74,144,8,74
1266 DATA74,9,32,136,208,250
1267 DATA200,136,208,242,96,177
1268 DATA193,32,194,252,0,162
1269 DATA1,32,254,250,0,196
1270 DATA31,200,144,241,162,3
1271 DATA192,4,144,242,96,168
1272 DATA185,55,255,0,133,40
1273 DATA185,119,255,0,133,41
1274 DATA169,0,0,160,5,6
1275 DATA41,38,40,42,136,208
1276 DATA248,105,63,32,210,255
1277 DATA202,208,236,169,32,44
1278 DATA169,13,76,210,255,32
1279 DATA212,250,0,32,105,250
1280 DATA0,32,229,250,0,32
1281 DATA105,250,0,162,0,0
1282 DATA134,40,169,144,32,210
1283 DATA255,32,87,253,0,32
1284 DATA114,252,0,32,202,252
1285 DATA0,133,193,132,194,32
1286 DATA225,255,240,5,32,47
1287 DATA251,0,176,233,76,71
```

```
1288 DATA248,0,32,212,250,0
1289 DATA169,3,133,29,32,62
1290 DATA248,0,32,161,248,0
1291 DATA208,248,165,32,133,193
1292 DATA165,33,133,194,76,70
1293 DATA252,0,197,40,240,3
1294 DATA32,210,255,96,32,212
1295 DATA250,0,32,105,250,0
1296 DATA142,17,2,162,3,32
1297 DATA204,250,0,72,202,208
1298 DATA249,162,3,104,56,233
1299 DATA63,160,5,74,110,17
1300 DATA2,110,16,2,136,208
1301 DATA246,202,208,237,162,2
1302 DATA32,207,255,201,13,240
1303 DATA30,201,32,240,245,32
1304 DATA208,254,0,176,15,32
1305 DATA156,250,0,164,193,132
1306 DATA194,133,193,169,48,157
1307 DATA16,2,232,157,16,2
1308 DATA232,208,219,134,40,162
1309 DATA0,0,134,38,240,4
1310 DATA230,38,240,117,162,0
1311 DATA0,134,29,165,38,32
1312 DATA217,252,0,166,42,134
1313 DATA41,170,188,55,255,0
1314 DATA189,119,255,0,32,185
1315 DATA254,0,208,227,162,6
1316 DATA224,3,208,25,164,31
1317 DATA240,21,165,42,201,232
1318 DATA169,48,176,33,32,191
1319 DATA254,0,208,204,32,193
1320 DATA254,0,208,199,136,208
1321 DATA235,6,42,144,11,188
1322 DATA48,255,0,189,42,255
1323 DATA0,32,185,254,0,208
1324 DATA181,202,208,209,240,10
1325 DATA32,184,254,0,208,171
1326 DATA32,184,254,0,208,166
1327 DATA165,40,197,29,208,160
1328 DATA32,105,250,0,164,31
1329 DATA240,40,165,41,201,157
1330 DATA208,26,32,28,251,0
1331 DATA144,10,152,208,4,165
1332 DATA30,16,10,76,237,250
1333 DATA0,200,208,250,165,30
1334 DATA16,246,164,31,208,3
1335 DATA185,194,0,0,145,193
1336 DATA136,208,248,165,38,145
1337 DATA193,32,202,252,0,133
1338 DATA193,132,194,169,144,32
```

```
1339 DATA210,255,160,65,32,194
1340 DATA248,0,32,84,253,0
1341 DATA32,65,250,0,32,84
1342 DATA253,0,169,5,32,210
1343 DATA255,76,176,253,0,168
1344 DATA32,191,254,0,208,17
1345 DATA152,240,14,134,28,166
1346 DATA29,221,16,2,8,232
1347 DATA134,29,166,28,40,96
1348 DATA201,48,144,3,201,71
1349 DATA96,56,96,64,2,69
1350 DATA3,208,8,64,9,48
1351 DATA34,69,51,208,8,64
1352 DATA9,64,2,69,51,208
1353 DATA8,64,9,64,2,69
1354 DATA179,208,8,64,9,0
1355 DATA0,34,68,51,208,140
1356 DATA68,0,0,17,34,68
1357 DATA51,208,140,68,154,16
1358 DATA34,68,51,208,8,64
1359 DATA9,16,34,68,51,208
1360 DATA8,64,9,98,19,120
1361 DATA169,0,0,33,129,130
1362 DATA0,0,0,0,89,77
1363 DATA145,146,134,74,133,157
1364 DATA44,41,44,35,40,36
1365 DATA89,0,0,88,36,36
1366 DATA0,0,28,138,28,35
1367 DATA93,139,27,161,157,138
1368 DATA29,35,157,139,29,161
1369 DATA0,0,41,25,174,105
1370 DATA168,25,35,36,83,27
1371 DATA35,36,83,25,161,0
1372 DATA0,26,91,91,165,105
1373 DATA36,36,174,174,168,173
1374 DATA41,0,0,124,0,0
1375 DATA21,156,109,156,165,105
1376 DATA41,83,132,19,52,17
1377 DATA165,105,35,160,216,98
1378 DATA90,72,38,98,148,136
1379 DATA84,68,200,84,104,68
1380 DATA232,148,0,0,180,8
1381 DATA132,116,180,40,110,116
1382 DATA244,204,74,114,242,164
1383 DATA138,0,0,170,162,162
1384 DATA116,116,116,114,68,104
1385 DATA178,50,178,0,0,34
1386 DATA0,0,26,26,38,38
1387 DATA114,114,136,200,196,202
1388 DATA38,72,68,68,162,200
1389 DATA58,59,82,77,71,88
```

```
1390 DATA76,83,84,70,72,68
1391 DATA80,44,65,66,249,0
1392 DATA53,249,0,204,248,0
1393 DATA247,248,0,86,249,0
1394 DATA137,249,0,244,249,0
1395 DATA12,250,0,62,251,0
1396 DATA146,251,0,192,251,0
1397 DATA56,252,0,91,253,0
1398 DATA138,253,0,172,253,0
1399 DATA70,248,0,255,247,0
1400 DATA237,247,0,13,32,32
1401 DATA32,80,67,32,32,83
1402 DATA82,32,65,67,32,88
1403 DATA82,32,89,82,32,83
1404 DATA80,52,55,44,49,54
1405 DATA57,44,54,52,0,16
1406 DATA18,78,4,131,49,51
1407 DATA51,44,49,56,55,44
1408 DATA49,54,57,44,50,44
1409 DATA49,51,51,44,49,56
1410 DIMC(40):FORI=1TO40:READC(I):NEXT:I=0
1420 I=I+1:FORJ=0TO5
1422 IFI=404THENPOKE4586,80:END
1425 READA:B=B+A:POKE2162+I*6+J,A:NEXTJ
1430 IFI/10=INT(I/10)THEN1450
1440 GOTO1420
1450 IFB<>C(INT(I/10))THENPRINT"ERROR IN LINES ";9
91+I;"-";1000+I;" B=";B
1460 B=0:GOTO1440

READY.
```

8
Machine Code: Mathematical Operations

Multiplication

We saw in chapter 6 how to handle addition and subtraction in machine code, and you could be forgiven for thinking that, just like the commands SBC and ADC for those mathematical operations, there would be a couple of commands to handle multiplication and division.

Unfortunately there aren't, and so multiplication has to be handled as a series of additions. To multiply, say, 4 by 5, you have to perform the calculation 4+4+4+4+4. In other words, add 4 to 0 five times, which can easily be achieved by a simple looping procedure.

For example:

```
C000 A0 05      LDY #$05
C002 A9 00      LDA #$00
C004 69 04      ADC #$04
C006 88         DEY
C007 D0 FB      BNE $C004
C009 8D 00 04   STA $0400
C00C A9 00      LDA #$00
C00E 8D 00 D8   STA $D800
C011 60         RTS
```

Step by step then:

Load the Y register with 5.

Load the accumulator with 0.

Add 4 to the contents of the accumulator.

Decrement the Y register.

If the result of decrementing the Y register is not equal to zero then branch back and add another 4 to the accumulator.

Store the accumulator at memory location $0400 (the top left hand corner of the screen).

Load the accumulator with 0.

Store it at the colour memory location that handles the top left hand corner of the screen.

Return from this subroutine.

Now this is all very well if we don't wish to add up numbers greater than 255. If we do, however, the accumulator will keep flipping back to zero and a carry will be registered each time our sum exceeds 255, and thus 256 will be lost from our answer each time that happens. One way to check for this is, if a carry is set, add one to the memory location that will be handling the high order value of the number, which can be done using the INC command. The INC command adds one to the contents of a specified memory location. If the answer is going to exceed 256 times 256, or 65536, then yet another memory location will be needed to check for another carry. For now, we'll stick to smaller numbers.

This small program multiplies 25 by 24, and displays the result on the screen. Of course, since the screen codes used for displaying characters are never the same as the character to be represented (that is, the number 16 doesn't have a screen code of 16), instead of numbers being printed up we get letters appearing instead. Oh well.

```
C000 A2 00      LDX #$00
C002 8E 25 C0   STX $C025
C005 A2 19      LDX #$19
C007 A9 00      LDA #$00
C009 18         CLC
C00A 69 18      ADC #$18
C00C 90 03      BCC $C011
C00E EE 25 C0   INC $C025
C011 D0 F6      BNE $C009
C013 8D 02 04   STA $0402
C016 AD 25 C0   LDA $C025
C019 8D 00 04   STA $0400
C01C A9 00      LDA #$00
```

```
C01E 8D 02 D8    STA $D802
C021 8D 00 D8    STA $D802
C024 60          RTS
```

As usual, we'll go through this one instruction at a time.

Load the X register with a zero.

Store it in location $C025, which is where we'll handle the high order value of the result of multiplying 25 by 24.

Load the X register with $19, or decimal 25, which is the number of times we'll perform this addition.

Load the accumulator with zero to get it ready for receiving the low order value.

Clear the carry flag before addition.

Add 24 to the value held in the accumulator.

If a carry isn't set, then we Branch on Carry Clear forward three locations.

Otherwise, increment the high order value at $C025.

Decrease the value in the X register by one.

If the result of that isn't equal to zero, then branch back and add another 24 to the accumulator.

Finally, store the result on the screen by placing the value in the accumulator at location $0402, picking up the high order value from $C025, and placing that at $0400. Put some colour into the locations, and retreat from this routine.

In the above example, it doesn't really matter whether we multiply the numbers in the order 25 by 24, or 24 by 25, since we'll still be making roughly the same number of passes around our main routine. However, if you wanted to multiply 5 by 1000 it would obviously be a lot quicker to do it one way than another. You'd probably have to write a short routine to make sure that the quickest route was taken, although the time required to perform that routine might well outweigh the advantages of running it in the first place.

Division

Just as multiplication has to be treated as a series of additions, so division has to be treated as a series of subtractions. In the multiplication program above, we had to clear the carry flag before adding the numbers up. With subtraction, we have to set the carry flag before subtracting them. In the following example, we'll divide 86 by 4.

```
C000 A0 00      LDY #$00
C002 A2 04      LDX #$04
C004 AE 40 C0   STX $C040
C007 A9 56      LDA #$56
C009 38         SEC
C00A E9 04      SBC #$04
C00C C8         INY
C00D CD 40 C0   CMP $C040
C010 B0 F7      BCS $C009
C012 8C 00 04   STY $0400
C015 8D 02 04   STA $0402
C018 A9 00      LDA #$00
C01A 8D 00 04   STA $0400
C01D 8D 02 04   STA $0402
C020 60         RTS
```

One line at a time:

Load the Y register with a zero.

Load the X register with 4, since this is what we want to divide by.

Store the X register at location $C040, so we can keep an eye on it.

Load the accumulator with $56, or decimal 86, since this is what we want to divide.

Set the carry flag before subtraction.

Subtract 4 from the value held in the accumulator.

Increment Y, since this is holding the quotient.

Compare the accumulator with the value previously stored at $C040.

If the carry flag is still set then branch back and take four away again.

It isn't set, so we can store the quotient on the screen, and the remainder is the value held in the accumulator. We finally colour our result on the screen in black so that you can see it, and then exit from the program.

There is a further way of carrying out mathematical operations, which involves something known as Binary Coded Decimal arithmetic, or BCD for short, which is switched on or off with the commands:

SED : SEt Decimal mode of operation.

CLD : CLear the Decimal flag.

You may recall the decimal flag (D) from our earlier discussion on the status register. When this flag is set, the 6502 is all ready to handle BCD. You must clear the flag after you've finished, otherwise dire consequences will occur.

BCD is a strange animal. The main difference between this and ordinary arithmetic, is that a carry occurs after each half-byte (or four bits) exceeds 9. What possible use is this, you may wonder. Well, for one thing this notation acts as a link between binary (which the machine likes to work in) and decimal (which we like to work in). For instance, if the answer to an addition sum was decimal 23, in ordinary binary this would have been stored as:

0 0 0 1 0 1 1 1

In other words, 23. However, in BCD this is looked at not as one byte, but as two nibbles, like this:

0 0 0 1 0 1 1 1

Which represents 1 times 10 from the left hand nibble, plus 7 from the right hand nibble, which of course equals 17.

Binary multiplication

We've already seen that there are a number of commands available to us for manipulating the bits within a byte: ROL, ASL and so on. Hence it would seem to make sense to use these commands in some kind of sensible fashion. One of their most powerful uses comes into

effect when we consider the topic of binary multiplication. Up until now, we've been handling our arithmetical calculations in the kind of way that we've been used to doing ever since leaving school. However, the 6510 doesn't particularly like this manner of dealing with numbers, and so the techniques involved for handling multiplication and division in a conventional way can, at times, get very complicated.

The 6510 prefers to work in binary, and using the aforementioned shifting and rotating instructions, arithmetical operations become much more straightforward.

Let us consider the way we would normally multiply two numbers together, say 123 and 89.

```
    123
     89
  ------
   1107      - known as partial product 1.
   0984      - known as partial product 2.
  ------
  10947
```

You can see that, as we multiply by each number, we move along one row of digits. More correctly, we are increasing the power of ten by 1 each one each time we produce a new partial product. Binary arithmetic follows much the same rules, and using the same numbers our sum now becomes, in binary:

```
        01111011 - 123
        01011001 - 89
----------------
        01111011
       00000000
      00000000
     01111011
    01111011
   00000000
  01111011
00000000
----------------
010101011000011 - 10947
```

The only difference here is that each time we move the partial product to the left, we're increasing by a power of two, not a power of ten as in ordinary decimal arithmetic.

However, there are two fundamental differences in the way that the computer performs this operation and the way that we've just done it.

(1) It keeps a running total as it goes along, which is updated after each partial product has been calculated, whereas we wait until the end and then add the whole lot up.

(2) When we multiply, each digit is examined from right to left, and each line is moved along one power of ten or two depending on what system we're working in. On the 6510 it's far easier to rotate the current partial product one row to the left (as we would) or to the right. This latter option is known as high order to low order multiplication.

Such sums as the one we covered above come under the heading of 16 bit multiplication, since the answer is a number way in excess of 255. For the time being we'll just look at 8 bit multiplication (since it's easier to understand!) and multiply together the two numbers 11 and 8.

```
C000 A2 08      LDX #$08
C002 A0 0B      LDY #$0B
C004 8E 40 C0   STX $C040
C007 8C 41 C0   STY $C041
C00A A0 08      LDY #$08
C00C A9 00      LDA #$00
C00D 18         CLC
C00E 4E 41 C0   LSR $C041
C011 90 04      BCC $C018
C014 18         CLC
C015 6D 40 C0   ADC $C040
C018 0E 40 C0   ASL $C040
C01B 88         DEY
C01C D0 F0      BNE $C00E
C01E 8D 00 04   STA $0400
C021 A9 00      LDA #$00
C023 8D 00 DB   STA $DB00
C026 60         RTS
```

In our usual way, we'll examine this listing one line at a time.

In the first line, we load the X register with an 8.

Then, load the Y register with $0B, or decimal 11. These are the two numbers that we'll be multiplying.

Store the X register at a 'safe' location, $C040.

Ditto for the Y register, at location $C041.

Load the Y register with an 8. Since we're dealing with 8 bit numbers only, we're going to pass through this loop 8 times. Unfortunately, 16 bit numbers cannot be handled purely by loading this register with a 16, as we shall see.

Load the accumulator with a zero.

Clear the carry flag.

Shift the content of location $C041 logically one place to the right (re-read the section on the shift and rotate commands if you've forgotten the result of doing this).

If no carry has been set, then branch forward four bytes.

Clear the carry flag again.

Add to the accumulator, with carry, the content of memory location $C040.

Shift every bit in the accumulator one bit to the left.

Decrease the Y register.

If we haven't done all 8 passes, then switch back to location $C00E.

We have finished, so store the result on the screen and give it some colour so that we can actually see it.

Unfortunately this routine will not work on two numbers that, when multiplied together, give an answer that is greater than 255. The reason for this lies in the way that the ASL command works. Each time we progress through the main program loop, every bit is shifted one bit to the right, and by the eighth time around the bit that originally started off at the right hand edge of the byte will have 'fallen off' the left hand edge.

This loss of leftmost bits is obviously quite important, since it would result in errors creeping into the calculations. However, the bit isn't actually lost, it is caught up in a carry. So as long as we can keep track of any carry that might be generated, it becomes possible to

multiply together any combination of numbers up to and including 255. We're still only working on 8 bits, remember: 16 bit numbers come later.

So, by using our old friend the ROL command, we should be able to multiply two 8 bit numbers together regardless of their size, and the following program demonstrates this in action.

```
B*
   PC  SR AC XR YR SP
.:5F79 33 00 61 00 F6
.
C000 A9 F0      LDA #$F0
C002 8D 00 C1   STA $C100
C005 A9 F1      LDA #$F1
C007 8D 04 C1   STA $C104
C00A A9 00      LDA #$00
C00C 8D 02 C1   STA $C102
C00F 8D 06 C1   STA $C106
C012 8D 07 C1   STA $C107
C015 A0 08      LDY #$08
C017 4E 00 C1   LSR $C100
C01A 90 13      BCC $C02F
C01C AD 06 C1   LDA $C106
C01F 18         CLC
C020 6D 04 C1   ADC $C104
C023 8D 06 C1   STA $C106
C026 AD 07 C1   LDA $C107
C029 6D 02 C1   ADC $C102
C02C 8D 07 C1   STA $C107
C02F 0E 04 C1   ASL $C104
C032 2E 02 C1   ROL $C102
C035 88         DEY
C036 D0 DF      BNE $C017
C038 AD 06 C1   LDA $C106
C03B 8D 01 04   STA $0401
C03E AD 07 C1   LDA $C107
C041 8D 00 04   STA $0400
C044 A9 00      LDA #$00
C046 8D 01 D8   STA $D801
C049 8D 00 D8   STA $D800
C04C 60         RTS
C04D FF         ???
.
.
```

The numbers are stored at locations $C001 and $C006, if you want to multiply numbers other than $F0 and $F1, or in decimal terms 240 and 241.

To multiply two larger numbers together, the next program multiplies 1000 by 250, to get the obvious result of 250000. Those numbers, in low-high order format, are stored in locations $C009 and $C00E, $C013 and $C018 respectively.

See if you can figure out what's happening. The key to the routine lies in locations $C03D to $C048. The result is stored as a 32-bit number, in locations $C105 to $C108 respectively.

```
.
C000 A9 00      LDA #$00
C002 8D 06 C1   STA $C106
C005 8D 07 C1   STA $C107
C008 A9 E8      LDA #$E8
C00A 8D 00 C1   STA $C100
C00D A9 03      LDA #$03
C00F 8D 01 C1   STA $C101
C012 A9 FA      LDA #$FA
C014 8D 02 C1   STA $C102
C017 A9 00      LDA #$00
C019 8D 03 C1   STA $C103
C01C A9 00      LDA #$00
C01E EA         NOP
C01F EA         NOP
C020 EA         NOP
C021 EA         NOP
C022 A2 10      LDX #$10
C024 4E 01 C1   LSR $C101
C027 6E 00 C1   ROR $C100
C02A 90 10      BCC $C03C
C02C AD 06 C1   LDA $C106
C02F 18         CLC
C030 6D 02 C1   ADC #C102
C033 8D 06 C1   STA $C106
C036 AD 07 C1   LDA $C107
C039 6D 03 C1   ADC $C103
C03C 6A         ROR
C03D 8D 07 C1   STA $C107
C040 6E 06 C1   ROR $C106
```

```
C043 6E 05 C1     ROR $C105
C046 6E 04 C1     ROR $C104
C049 CA           DEX
C04A D0 DB        BNE $C024
C04C 60           RTS
.
.
```

9
Machine Code: Indirect Addressing

Introduction

We're coming to the end of our first foray into machine code, and now it's time to tidy up a few loose ends, including a quick look at using built-in subroutines, adding commands to BASIC, and one last discussion on indirect addressing, as promised.

I can do little better at this point than to quote the authoritative voice of Dave Parkinson, of Ariadne Software, in a letter he once sent to me regarding this subject. I'd spoken briefly about indirect addressing in an earlier book, and Dave was quick to point out that one or two things weren't quite right. And I quote:

'Like the rest of humanity, you are confused about indirect addressing!

'Life is made complicated by the fact that all the indirect modes (except JMP indirect) are indexed using the X or Y registers, so let's suppose for the time being that they weren't. If so, immediate, absolute and indirect addressing would form a hierarchy as follows:

'LDA # FRED : means load the accumulator with the byte FRED immediately following the op-code.

'LDA FRED : means look in memory location FRED, and load the accumulator with the byte that you find there.

'LDA (FRED) : means look in memory locations FRED and FRED + 1, interpret the bytes that you find there as an address, look in the location with that address, and load the accumulator with the byte that you find there. In other words, FRED is interpreted as a POINTER to the byte that you want.

'Unfortunately, for reasons known only to Chuck Peddle (and he's probably forgotten), life is more complicated than this, as the 6502 (father of the 6510 used in the 64) was designed without a simple LDA (FRED), but with two (different) indexed indirect addressing modes, LDA (FRED),Y and LDA (FRED,X).

'The first of these is known as INDIRECT INDEXED addressing, and it is vital to know how to use it. If you don't, then you end up with code at least four times longer than it needs to be, plus you make an idiot of yourself by writing letters to the computer press saying that the 6502 is an inferior chip to the Z80 (e.g. the authors of the Hobbit).

'The effect of LDA (FRED),Y is to look in the zero-page addresses FRED and FRED + 1 and interpret the bytes that you find there as an address. You then add Y onto this address, look at the address which results, and load the accumulator with the byte that you find there.

'An example is a subroutine to print a string terminated by a carriage return. This routine is called by loading A and Y with the high and low bytes of the address of the start of the string, then calling the print routine. For example:

```
LDA # <STRING
LDY # >STRING
JSR PRINT

(more code).
```

(Here we're using descriptors, as encountered in some assemblers, rather than our more familiar #$C000, etc. Just interpret the words as addresses, e.g. JSR PRINT means jump to the PRINT subroutine in the 64's internals).

'The print subroutine would then be as follows:

```
PRINT   STA POINTER       ; set up pointer to string in
                          page zero

        STY POINTER + 1
        LDY #0            ; start with first byte.

PRIN10  LDA (POINTER),Y   ; get a byte
        JSR CHROUT        ; Kernal character out routine
```

```
INY             ; prepare for next byte
CMP #$0D        ; reached the end
BNE PRIN10      ; if not, keep going
RTS             ; the end
```

'The other indirect addressing mode, INDEXED INDIRECT addressing using the X register, LDA (FRED,X) is hardly ever used in 6502-based computers. The effect of it is to add X onto FRED, look in FRED + X and FRED + X + 1, interpret these bytes as an address, and load the accumulator with the byte to be found at this address. The mode is useful in 6502-based control applications, when you might have a whole load of I/O chips, and a table of their addresses in page zero. The only time you might use it on the 64 is with X equal to zero; LDA (FRED),Y and LDA (FRED,X) are equivalent if X and Y are both zero.

'Confusing, isn't it?'

It certainly is, but I hope Dave's comments will shed some light on the subject. All writs and lawsuits will be forwarded to him as soon as they arrive at the publishers!

10
Machine Code: Built-in Subroutines

Introduction

We are now going a little bit beyond the scope of this present introductory tome (it is after all an introduction, not a machine code Bible), but a few general words of advice might be in order here.

Linking to BASIC

There are two ways of accessing and using machine code from BASIC, rather than simple PEEKs and POKEs, which allow us to look at and alter the contents of various memory locations.

The first that we'll look at is USR. As you may know, this sends a variable (A = USR(B)) B to the floating point accumulator.

Control then jumps to the machine code routine whose address is found from locations 784 and 785. Thus by altering those two locations before the USR call, we can pass variables to and from machine code subroutines, and hence mix BASIC and machine code within the same program.

The following short example should serve to demonstrate this:

```
10 PRINT "[CLR]"
20 POKE 784,0 : REM LSB
30 POKE 785,192 : REM MSB (=$C0)
40 B=123
50 A=USR(B)
60 PRINT "[CD]A =";A
```

Provided that we have earlier put an RTS into memory location $C000

(this can be achieved by using the statement POKE 49152,96), control will go to that memory location, encounter the RTS and come back, and our BASIC program will then print out the content of the floating point accumulator.

SYStem calling

As you will see from the memory maps at the end of this section, the Commodore 64 is equipped with a wealth of internal machine code subroutines that can be accommodated within your own programs. However, our usual conflict arises in that the 64 wants to use them as much as you do. Consequently, using these routines for the inexperienced programmer can be a daunting prospect, and we're not going to go into any great detail in this book.

However, a few words of advice:

(1) It may seem obvious, but do study the memory maps to see where these routines start.

(2) Use an assembler before attempting to use any of them. Disassemble the listing for the various routines, and see where they start and end, and where they go to while in operation. Some of them will have additional subroutines built into them, which cause program control to skip about all over the place.

(3) Find out where these routines pick up their data from and where they subsequently store the result.

(4) Bear in mind that they will be using the accumulator as much as you want to, and programs will need careful thought if they are to work correctly.

With the disassembler option in most assemblers, and the complete memory maps to guide you around, using built-in routines is not as difficult as it might first appear.

Musical interlude

Before we start dissecting some of the workings of the 64 in the next chapter, a musical interlude. We've already taken a look at graphics, and sprites in particular, so it's about time that the other great feature of the 64 got a look in as well.

The programs that follow are to a large extent based on the theories outlined in chapter 11, and the method used there to start adding commands to BASIC. See if you can follow the methods used now, rather than waiting for the explanation then.

A couple of BASIC driver programs will be given first. No real difference between them, except that they produce slightly different sounds when used with the various programs. Either can be used with any of the musical data to follow.

The music, when played, will continue to carry on whatever else the 64 is doing, provided that the 'whatever else' doesn't involve loading or saving to and from disk, or using some other external device. This is demonstrated (simply, I must admit) in the first driver program, which just prints the value stored in the 64's internal clock in the top left hand corner of the screen.

This is just one way of producing those irritating background tunes so common in arcade games for the 64!

MUSIC DRIVER 1

```
10 S=54272:POKES+3,124
15 POKES+5,121:POKES+6,33
20 POKES+24,15:POKE49216,10:POKE49218,0
25 POKE49219,65:POKE251,178:POKE252,192
30 POKE253,178:POKE254,192:SYS49152:POKES+8,0
40 PRINT"[CLR]":FORI=1TO10000:PRINT"[HOME]";TI
45 NEXTI

READY.
```

MUSIC DRIVER 2

```
10 S=54272:POKES+3,255:POKES+5,142:POKES+6,150
15 POKES+5,142:POKES+6,150
20 POKES+24,15:POKE49216,10:POKE49218,0
25 POKE49219,65:POKE251,178:POKE252,192
30 POKE253,178:POKE254,192:SYS49152:POKES+8,0

READY.
```

```
B*
   PC  SR AC XR YR SP
.;7FC5 33 00 AD 00 F6
.
C000 EA          NOP
C001 EA          NOP
C002 EA          NOP
C003 78          SEI
C004 A9 50       LDA #$50
C006 8D 14 03    STA $0314
C009 A9 C0       LDA #$C0
C00B 8D 15 03    STA $0315
C00E 58          CLI
C00F EA          NOP
C010 EA          NOP
C011 60          RTS
C012 78          SEI
C013 A9 31       LDA #$31
C015 8D 14 03    STA $0314
C018 A9 EA       LDA #$EA
C01A 8D 15 03    STA $0315
C01D 58          CLI
C01E EA          NOP
C01F 60          RTS
C020 EA          NOP
C021 DE DE DE    DEC $DEDE,X
C024 DE DE DE    DEC $DEDE,X
C027 DE DE DE    DEC $DEDE,X
C02A DE DE DE    DEC $DEDE,X
C02D DE DE DE    DEC $DEDE,X
C030 DE DE DE    DEC $DEDE,X
C033 DE DE DE    DEC $DEDE,X
C036 DE DE DE    DEC $DEDE,X
C039 DE DE DE    DEC $DEDE,X
C03C DE DE DE    DEC $DEDE,X
C03F DE 0D 08    DEC $080D,X
C042 00          BRK
C043 41 00       EOR ($00,X)
C045 00          BRK
C046 00          BRK
C047 00          BRK
C048 00          BRK
C049 00          BRK
C04A 00          BRK
C04B 00          BRK
C04C 00          BRK
C04D 00          BRK
C04E 00          BRK
C04F 00          BRK
C050 EA          NOP
C051 EA          NOP
```

```
C052 EA          NOP
C053 AD 42 C0    LDA $C042
C056 F0 04       BEQ $C05C
C058 4C 31 EA    JMP $EA31
C05B EA          NOP
C05C CE 41 C0    DEC $C041
C05F D0 F7       BNE $C058
C061 EA          NOP
C062 EA          NOP
C063 EA          NOP
C064 AD 40 C0    LDA $C040
C067 8D 41 C0    STA $C041
C06A EA          NOP
C06B E6 FB       INC $FB
C06D D0 02       BNE $C071
C06F E6 FC       INC $FC
C071 EA          NOP
C072 A0 00       LDY #$00
C074 B1 FB       LDA ($FB),Y
C076 C9 FF       CMP #$FF
C078 F0 28       BEQ $C0A2
C07A EA          NOP
C07B EA          NOP
C07C EA          NOP
C07D EA          NOP
C07E A0 00       LDY #$00
C080 8D 01 D4    STA $D401
C083 E6 FB       INC $FB
C085 D0 02       BNE $C089
C087 E6 FC       INC $FC
C089 EA          NOP
C08A B1 FB       LDA ($FB),Y
C08C 8D 00 D4    STA $D400
C08F AD 43 C0    LDA $C043
C092 48          PHA
C093 A9 00       LDA #$00
C095 8D 04 D4    STA $D404
C098 68          PLA
C099 8D 04 D4    STA $D404
C09C EA          NOP
C09D EA          NOP
C09E 4C 31 EA    JMP $EA31
C0A1 EA          NOP
C0A2 EA          NOP
C0A3 EA          NOP
C0A4 A5 FD       LDA $FD
C0A6 85 FB       STA $FB
C0A8 A5 FE       LDA $FE
C0AA 85 FC       STA $FC
C0AC 4C 64 C0    JMP $C064
C0AF 00          BRK
```

```
C0B0 EA          NOP
C0B1 FF          ???
C0B2 FF          ???
C0B3 12          ???
C0B4 12          ???
C0B5 13          ???
C0B6 13          ???
C0B7 15 15       ORA $15,X
C0B9 21 21       AND ($21,X)
C0BB 00          BRK
C0BC 00          BRK
C0BD 15 15       ORA $15,X
C0BF 21 21       AND ($21,X)
C0C1 00          BRK
C0C2 00          BRK
C0C3 15 15       ORA $15,X
C0C5 00          BRK
C0C6 00          BRK
C0C7 21 21       AND ($21,X)
C0C9 00          BRK
C0CA 00          BRK
C0CB 00          BRK
C0CC 00          BRK
C0CD 00          BRK
C0CE 00          BRK
C0CF 00          BRK
C0D0 00          BRK
C0D1 00          BRK
C0D2 00          BRK
C0D3 25 25       AND $25
C0D5 27          ???
C0D6 27          ???
C0D7 2A          ROL
C0D8 2A          ROL
C0D9 21 21       AND ($21,X)
C0DB 25 25       AND $25
C0DD 00          BRK
C0DE 00          BRK
C0DF 2A          ROL
C0E0 2A          ROL
C0E1 21 21       AND ($21,X)
C0E3 25 25       AND $25
C0E5 00          BRK
C0E6 00          BRK
C0E7 21 21       AND ($21,X)
C0E9 00          BRK
C0EA 00          BRK
C0EB 00          BRK
C0EC 00          BRK
C0ED 00          BRK
C0EE 00          BRK
```

```
C0EF 00          BRK
C0F0 00          BRK
C0F1 00          BRK
C0F2 00          BRK
C0F3 00          BRK
C0F4 00          BRK
C0F5 00          BRK
C0F6 00          BRK
C0F7 00          BRK
C0F8 00          BRK
C0F9 FF          ???
.
.
```

The hub of the music program lies from locations $C000 to $C0B2, with the actual 'tune' that is played following on behind. The music being played goes back to the first note again when an FF is encountered, as at location $C0F9 in the first program. The notes are then stored in-between, with the notes coming in the traditional high-low format. The values to use for any particular note can be found on pp. 152-4 of the manual supplied with your 64. You will, however, have to convert the values given there into hexadecimal ones.

To insert a gap between notes being played, just insert a few 00 (BRKs) to suit. The more BRKs, the longer the delay between notes.

```
B*
   PC  SR AC XR YR SP
.;6F9F 33 00 87 00 F6
.
C0B3 02          ???
C0B4 04          ???
C0B5 05 05       ORA $05
C0B7 06 06       ASL $06
C0B9 07          ???
C0BA 07          ???
C0BB 08          PHP
C0BC 08          PHP
C0BD 07          ???
C0BE 07          ???
C0BF 06 06       ASL $06
C0C1 05 05       ORA $05
C0C3 02          ???
C0C4 04          ???
C0C5 05 05       ORA $05
C0C7 06 06       ASL $06
C0C9 07          ???
```

```
C0CA 07          ???
C0CB 08          PHP
C0CC 08          PHP
C0CD 07          ???
C0CE 07          ???
C0CF 06 06       ASL $06
C0D1 05 05       ORA $05
C0D3 09 09       ORA #$09
C0D5 08          PHP
C0D6 09 09       ORA #$09
C0D8 07          ???
C0D9 07          ???
C0DA 07          ???
C0DB 06 06       ASL $06
C0DD 06 04       ASL $04
C0DF 04          ???
C0E0 03          ???
C0E1 03          ???
C0E2 03          ???
C0E3 04          ???
C0E4 05 06       ORA $06
C0E6 06 07       ASL $07
C0E8 09 05       ORA #$05
C0EA 04          ???
C0EB 03          ???
C0EC 04          ???
C0ED 05 06       ORA $06
C0EF 07          ???
C0F0 08          PHP
C0F1 09 0A       ORA #$0A
C0F3 0A          ASL
C0F4 09 08       ORA #$08
C0F6 07          ???
C0F7 07          ???
C0F8 06 07       ASL $07
C0FA 08          PHP
C0FB 09 0A       ORA #$0A
C0FD 05 06       ORA $06
C0FF 0B          ???
C100 0B          ???
C101 06 07       ASL $07
C103 08          PHP
C104 07          ???
C105 08          PHP
C106 0C          ???
C107 0C          ???
C108 0A          ASL
C109 0A          ASL
C10A 0B          ???
C10B 0B          ???
C10C 0A          ASL
```

```
C10D 09 08        ORA #$08
C10F 07           ???
C110 06 06        ASL $06
C112 03           ???
C113 03           ???
C114 03           ???
C115 03           ???
C116 03           ???
C117 03           ???
C118 FF           ???
.


.

AND SOME BOOGIE


B*
   PC  SR AC XR YR SP
.;5F79 33 00 61 00 F6
.
COB3 04           ???
COB4 49 05        EOR #$05
COB6 67           ???
COB7 06 6A        ASL $6A
COB9 07           ???
COBA 35 07        AND $07,X
COBC A3           ???
COBD 07           ???
COBE 35 06        AND $06,X
COCO 6A           ROR
COC1 05 67        ORA $67
COC3 04           ???
COC4 49 05        EOR #$05
COC6 67           ???
COC7 06 6A        ASL $6A
COC9 07           ???
COCA 35 07        AND $07,X
COCC A3           ???
COCD 07           ???
COCE 35 06        AND $06,X
CODO 6A           ROR
COD1 05 67        ORA $67
COD3 05 B9        ORA $B9
COD5 07           ???
COD6 45 08        EOR $08
COD8 93           ???
COD9 09 9F        ORA #$9F
CODB 0A           ASL
CODC 3C           ???
CODD 09 9F        ORA #$9F
```

```
C0DF 08          PHP
C0E0 93          ???
C0E1 07          ???
C0E2 45 04       EOR $04
C0E4 49 05       EOR #$05
C0E6 67          ???
C0E7 06 6A       ASL $6A
C0E9 07          ???
C0EA 35 07       AND $07,X
C0EC A3          ???
C0ED 07          ???
C0EE 35 06       AND $06,X
C0F0 6A          ROR
C0F1 05 67       ORA $67
C0F3 06 6A       ASL $6A
C0F5 08          PHP
C0F6 17          ???
C0F7 09 9F       ORA #$9F
C0F9 08          PHP
C0FA 17          ???
C0FB 05 B9       ORA $B9
C0FD 07          ???
C0FE 35 08       AND $08,X
C100 93          ???
C101 07          ???
C102 35 FF       AND $FF,X
.
.
```

I'm sure you'll be able to produce your own musical experiments before too long.

And now, as promised, a map of the internal machine code subroutines as used by the ROM in the Commodore 64. With the assembler that you've become familiar with using from the listing MONMAKER, take these listings to pieces and see how they work. They're a lesson in themselves.

Commodore 64 – ROM Memory Map

A000; ROM control vectors
A00C; Keyword action vectors
A052; Function vectors
A080; Operator vectors
A09E; Keywords
A19E; Error messages
A328; Error message vectors
A365; Misc messages
A38A; Scan stack for FOR/GOSUB
A3B8; Move memory
A3FB; Check stack depth
A408; Check memory space
A435; 'out of memory'
A437; Error routine
A469; BREAK entry
A474; 'ready.'
A480; Ready for Basic
A49C; Handle new line
A533; Re–chain lines
A560; Receive input line
A579; Crunch tokens
A613; Find Basic line
A642; Perform [NEW]
A65E; Perform [CLR]
A68E; Back up text pointer
A69C; Perform [LIST]
A742; Perform [FOR]
A7ED; Execute statement
A81D; Perform [RESTORE]
A82C; Break
A82F; Perform [STOP]
A831; Perform [END]
A857; Perform [CONT]
A871; Perform [RUN]
A883; Perform [GOSUB]
A8A0; Perform [GOTO]
A8D2; Perform [RETURN]
A8F8; Perform [DATA]
A906; Scan for next statement
A928; Perform [IF]
A93B; Perform [REM]
A94B; Perform [ON]
A96B; Get fixed point number
A9A5; Perform [LET]
AA80; Perform [PRINT#]
AA86; Perform [CMD]
AAA0; Perform [PRINT]
AB1E; Print string from (y.a)
AB3B; Print format character
AB4D; Bad input routine
AB7B; Perform [GET]
ABA5; Perform [INPUT#]
ABBF; Perform [INPUT]
ABF9; Prompt & input
AC06; Perform [READ]
ACFC; Input error messages
AD1E; Perform [NEXT]
AD78; Type match check
AD9E; Evaluate expression
AEA8; Constant – pi
AEF1; Evaluate within brackets
AEF7; ')'
AEFF; comma..
AF08; Syntax error
AF14; Check range
AF28; Search for variable
AFA7; Setup FN reference
AFE6; Perform [OR]
AFE9; Perform [AND]
B016; Compare
B081; Perform [DIM]
B08B; Locate variable
B113; Check alphabetic
B11D; Create variable
B194; Array pointer subrtine
B1A5; Value 32768
B1B2; Float–fixed
B1D1; Set up array
B245; 'bad subscript'
B248; 'illegal quantity'
B34C; Compute array size
B37D; Perform [FRE]
B391; Fix–float
B39E; Perform [POS]
B3A6; Check direct
B3B3; Perform [DEF]
B3E1; Check fn syntax
B3F4; Perform [FN]
B465; Perform [STR$]
B475; Calculate string vector
B487; Set up string
B4F4; Make room for string
B526; Garbage collection
B5BD; Check salvageability
B606; Collect string
B63D; Concatenate
B67A; Build string to memory
B6A3; Discard unwanted string
B6DB; Clean descriptor stack
B6EC; Perform [CHR$]
B700; Perform [LEFT$]
B72C; Perform [RIGHT$]
B737; Perform [MID$]
B761; Pull string parameters
B77C; Perform [LEN]
B782; Exit string–mode
B78B; Perform [ASC]
B79B; Input byte paramter
B7AD; Perform [VAL]
B7EB; Parameters for POKE/WAIT
B7F7; Float–fixed
B80D; Perform [PEEK]
B824; Perform [POKE]
B82D; Perform [WAIT]

B849; Add 0.5
B850; Subtract-from
B853; Perform [subtract]
B86A; Perform [add]
B947; Complement FAC#1
B97E; 'overflow'
B983; Multiply by zero byte
B9EA; Perform [LOG]
BA2B; Perform [multiply]
BA59; Multiply-a-bit
BA8C; Memory to FAC#2
BAB7; Adjust FAC#1/#2
BAD4; Underflow/overflow
BAE2; Multiply by 10
BAF9; +10 in floating pt
BAFE; Divide by 10
BB12; Perform [divide]
BBA2; Memory to FAC#1
BBC7; FAC#1 to memory
BBFC; FAC#2 to FAC#1
BC0C; FAC#1 to FAC#2
BC1B; Round FAC#1
BC2B; Get sign
BC39; Perform [SGN]
BC58; Perform [ABS]
BC5B; Compare FAC#1 to mem
BC9B; Float-fixed
BCCC; Perform [int]
BCF3; String to FAC
BD7E; Get ascii digit
BDC2; Print 'IN..'
BDCD; Print line number
BDDD; Float to ascii
BF16; Decimal constants
BF3A; TI constants
BF71; Perform [SQR]
BF7B; Perform [power]
BFB4; Perform [negative]
BFED; Perform [EXP]
E043; Series eval 1
E059; Series eval 2
E097; Perform [RND]
E0f9; ?? breakpoints ??
E12A; Perform [SYS]
E156; Perform [SAVE]
E165; Perform [VERIFY]
E168; Perform [LOAD]
E1BE; Perform [OPEN]
E1C7; Perform [CLOSE]
E1D4; Parameters for LOAD/SAVE
E206; Check default parameters
E20E; Check for comma
E219; Parameters for open/close
E264; Perform [COS]
E26B; Perform [SIN]
E2B4; Perform [TAN]
E30E; Perform [ATN]
E37B; Warm restart

E394; Initialize
E3A2; CHRGET for zero page
E3BF; Initialize Basic
E447; Vectors for $300
E453; Initialize vectors
E45F; Power-up message
E500; Get I/O address
E505; Get screen size
E50A; Put/get row/column
E518; Initialize I/O
E544; Clear screen
E566; Home cursor
E56C; Set screen pointers
E5A0; Set I/O defaults
E5B4; Input from keyboard
E632; Input from screen
E684; Quote test
E691; Setup screen print
E6B6; Advance cursor
E6ED; Retreat cursor
E701; Back into previous line
E716; Output to screen
E87C; Go to next line
E891; Perform <return>
E8A1; Check line decrement
E8B3; Check line increment
E8CB; Set color code
E8DA; Color code table
E8EA; Scroll screen
E965; Open space on screen
E9C8; Move a screen line
E9E0; Synchronize color transfer
E9F0; Set start-of-line
E9FF; Clear screen line
EA13; Print to screen
EA24; Synchronize color pointer
EA31; Interrupt - clock etc
EA87; Read keyboard
EB79; Keyboard select vectors
EB81; Keyboard 1 - unshifted
EBC2; Keyboard 2 - shifted
EC03; Keyboard 3 - 'comm'
EC44; Graphics/text contrl
EC4F; Set graphics/text mode
EC78; Keyboard 4
ECB9; Video chip setup
ECE7; Shift/run equivalent
ECF0; Screen ln address low
ED09; Send 'talk'
ED0C; Send 'listen'
ED40; Send to serial bus
EDB2; Serial timeout
EDB9; Send listen SA
EDBE; Clear ATN
EDC7; Send talk SA
EDCC; Wait for clock
EDDD; Send serial deferred
EDEF; Send 'untalk'

EDFE; Send 'unlisten'
EE13; Receive from serial bus
EE85; Serial clock on
EE8E; Serial clock off
EE97; Serial output '1'
EEA0; Serial output '0'
EEA9; Get serial in & clock
EEB3; Delay 1 ms
EEBB; RS-232 send
EF06; Send new RS-232 byte
EF2E; No-DSR error
EF31; No-CTS error
EF3B; Disable timer
EF4A; Compute bit count
EF59; RS232 receive
EF7E; Setup to receive
EFC5; Receive parity error
EFCA; Receive overflow
EFCD; Receive break
EFD0; Framing error
EFE1; Submit to RS232
F00D; No-DSR error
F017; Send to RS232 buffer
F04D; Input from RS232
F086; Get from RS232
F0A4; Check serial bus idle
F0BD; Messages
F12B; Print if direct
F13E; Get..
F14E; ..from RS232
F157; Input
F199; Get.. tape/serial/rs232
F1CA; Output..
F1DD; ..to tape
F20E; Set input device
F250; Set output device
F291; Close file
F30F; Find file
F31F; Set file values
F32F; Abort all files
F333; Restore default I/O
F34A; Do file open
F3D5; Send SA
F409; Open RS232
F49E; Load program
F5AF; 'searching'
F5C1; Print filename
F5D2; 'loading/verifying'
F5DD; Save program
F68F; Print 'saving'
F69D; Bump clock
F6BC; Log PIA key reading
F6DD; Get time
F6E4; Set time
F6ED; Check stop key
F6FB; Output error messages
F72D; Find any tape headr
F76A; Write tape header
F7D0; Get buffer address
F7D7; Set buffer start/end pointers
F7EA; Find specific header
F80D; Bump tape pointer
F817; 'press play..'
F82E; Check tape status
F838; 'press record..'
F841; Initiate tape read
F864; Initiate tape write
F875; Common tape code
F8D0; Check tape stop
F8E2; Set read timing
F92C; Read tape bits
FA60; Store tape chars
FB8E; Reset pointer
FB97; New character setup
FBA6; Send transition to tape
FBC8; Write data to tape
FBCD; IRQ entry point
FC57; Write tape leader
FC93; Restore normal IRQ
FCB8; Set IRQ vector
FCCA; Kill tape motor
FCD1; Check r/w pointer
FCDB; Bump r/w pointer
FCE2; Power reset entry
FD02; Check 8-rom
FD10; 8-rom mask
FD15; Kernal reset
FD1A; Kernal move
FD30; Vectors
FD50; Initialize system constnts
FD9B; IRQ vectors
FDA3; Initialize I/O
FDDD; Enable timer
FDF9; Save filename data
FE00; Save file details
FE07; Get status
FE18; Flag status
FE1C; Set status
FE21; Set timeout
FE25; Read/set top of memory
FE27; Read top of memory
FE2D; Set top of memory
FE34; Read/set bottom of memory
FE43; NMI entry
FE66; Warm start
FEB6; Reset IRQ & exit
FEBC; Interrupt exit
FEC2; RS-232 timing table
FED6; NMI RS-232 in
FF07; NMI RS-232 out
FF43; Fake IRQ
FF48; IRQ entry
FF81; Jumbo jump table
FFFA; Hardware vectors

11
Machine Code: Adding Commands to BASIC

Introduction

The deficiencies inherent in Commodore Basic are well known. But it's interesting to trace these deficiencies back through time to the very early Commodore machines.

The first Commodore PET, as well as coming complete with its own cassette deck and monitor, and having a paltry 8K of RAM (and also costing some £625 when it first appeared back in 1979!), had what Commodore themselves termed Basic 1.

As a Basic language it was fine at the time, but there were a number of things missing from it. For example, there was no way of accessing the machine code monitor, as it didn't have one built in.

A utility to overcome this soon came on the market, but this took up precious space from the meagre amount of RAM that you had, and so Commodore followers had to wait a couple of years before Basic 2 appeared.

Basic 2

When it did appear it caused instant confusion among the Commodore ranks, since some people were calling it Basic 2, and others referred to it as Basic 3. Seemingly the so-called 'Basic 2' never appeared, and although this particular version of the language was always called Basic 2, theoretically it should have been referred to as Basic 3.

Still, whatever number you gave it it was a great improvement over its predecessor, and did have access to a machine code monitor. Moreover, the ROM installed was now capable of looking after disk

drives, something that the earlier machines could not do.

Time went by, Basic 4 appeared, and for a long time it was rumoured that there was to be a fifth version of the language as well, with all the features that existing ones had lacked, of which more in a moment.

However, at the time of writing Basic 5 is in the realms of fantasy, and is not likely to appear now.

64 Basic

This brings us to the Commodore 64, sidestepping the Vic along the way.

The version of Basic that they've installed in the machine is that which we referred to earlier as Basic 2/3, although Commodore have apparently now decided that it should be called Basic 2, and indeed this is what the machine greets you with when you turn it on.

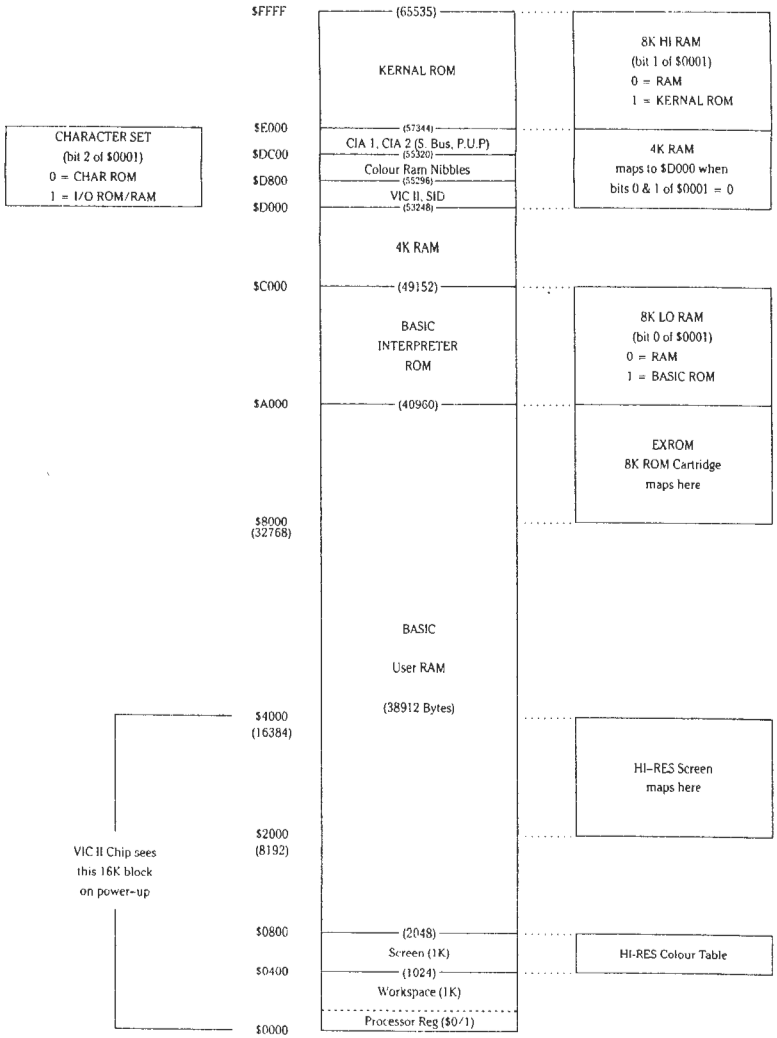
However, not only have Commodore taken a retrograde step and installed an old version of their popular Basic language, but they've also managed to take out a great deal of what was already in there.

So we see no machine code monitor - and hence the need for programs such as Extramon and the like.

What we are left with instead is an extremely flexible memory management system, but a very poor Basic with which to manage it.

The memory architecture looks something like this:

Commodore-64 Architecture Map



To look after all this requires a lot of work, and to understand it all properly requires even more!

Still, what you buy is what you get, so let's see precisely what we have got.

Basic advantages

The version of Basic in the Commodore 64 is a pretty standard version of what is usually referred to as Microsoft Basic.

This is based on the original Beginners All-purpose Symbolic Instruction Code, from which the language takes its name. This language was devised a number of years ago, and the cracks are now beginning to show, but for a beginner it is still possibly the easiest of languages to learn.

Apart from the interface to machine code, which is not good, the commands you have at your disposal are not too difficult to understand and get to grips with, and owing to the great similarity between Basic words and English words, most beginners can soon start writing programs in Basic.

And disadvantages

However, most beginners also soon come to realise that the version of Basic as supplied by Commodore is sadly lacking in a number of departments.

The concepts of structured programming, the computer flavour of the month, are impossible to simulate on the 64, and there is a distinct lack of such commands as PRINT AT, PRINT USING, and so on.

In particular, when it comes to using graphics and sound, the number of commands is strictly limited to two : PEEK and POKE. No other commands exist to cope with the vast number of PEEKs and POKEs needed to set up a high resolution screen and draw things on it, or to play a few musical notes, or do just about anything with either graphics or sound.

If you want to make music, or display various images on the screen, it has all got to be done the long way, by using a laborious series of POKEs.

Given that this version of Basic is so appalling in these particular departments, it is no wonder that people go to great lengths to try to improve it.

There are now many packages on the market that, in a variety of different ways, have set out to try to improve on the language that we are originally offered.

Whether they succeed in their chosen aims is, of course, a completely different matter, but what they all have in common is that they are adding commands to the existing version of Basic, and through those commands are seeking to make life easier for the person using the machine.

The rest of this chapter will be devoted to showing you one way in which commands could be added, as well as giving you a number of routines to try out for yourself.

But first, the concepts involved.

Adding commands: the concepts

There are many different ways in which you can add commands to Commodore's existing command set. Commands can be added either as words or symbols, or indeed we could also use the function keys: re-define them to be able to accept existing Basic keywords, and then put our new words (or symbols) in their place.

We'll be looking at the two simplest options in this chapter, namely defining various symbols to act as commands, rather than adding new words, and re-defining the function keys to accept these symbols.

These are certainly easier than trying to add new command words to Basic, as this involves altering a lot more things than we are going to do, and for the first time user can seem to be incredibly complicated. So complicated in fact that you probably wouldn't even want to try it!

Still, what we are going to do is fairly straightforward, and shouldn't present any major difficulties.

Getting a character

Anything that you type onto the screen is interpreted and executed by the Commodore 64 as soon as you press the return key. Once this key has been pressed there are a number of routines built into the 64 which will act upon everything that you typed in, and depending on precisely what you typed a number of things will happen.

You can generate a syntax error, and a subroutine exists within the Basic ROM to print out a suitable message and return to await your next input. Since it is in ROM (it starts at location $AF08) we can't alter it, but there's nothing to stop us copying this ROM into RAM and altering it there, so that SYNTAX ERROR becomes something *a lot more meaningful. Or a lot more rude, if you're feeling in that kind* of mood!

You could have entered a line of a program, in which case you won't get any error messages (or for that matter any other messages) coming back at all, but a great many pointers inside the machine will have been altered to cope with the new line.

You might have entered a direct command, and in this case the machine will just execute whatever it was that you typed in.

Character get routine

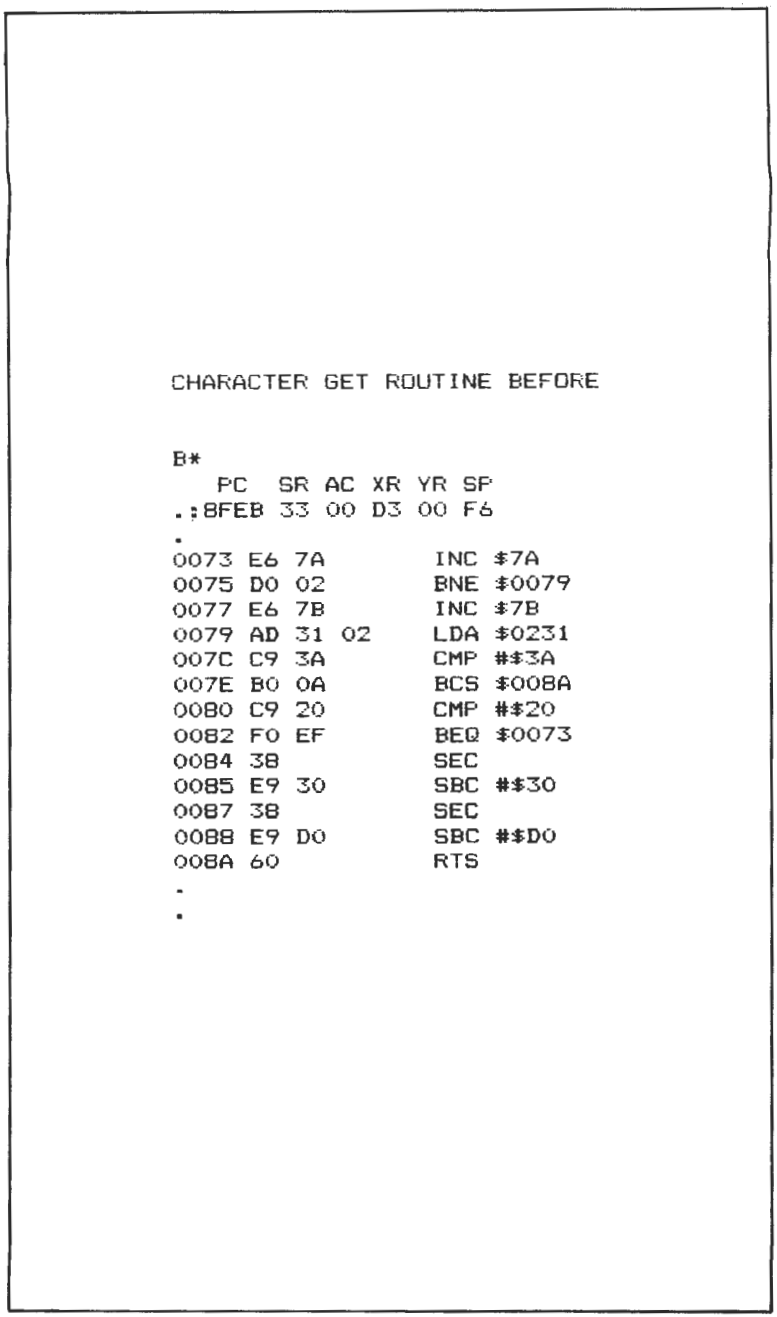
How does the machine know what to do? In other words, how does it interpret what you've typed in? Understanding this is the key to generating our own commands, because if we can intercept the Basic routine that looks after all the commands and alter it, we are then well on the way to adding our own commands into the machine.

The machine knows what to do because of the ROM that's built into it, but there must be a routine somewhere in the machine that looks at what you've typed in and thinks 'ahah!', and then does (or attempts to do) whatever you've told it.

There is indeed such a routine, which lives in locations $0073 to $008A (or decimal locations 115 to 138), and this is usually referred to as the CHARGET routine, or character get.

This is the routine that gets a character that you've typed in and acts upon that character.

The routine looks, in its original form, like this:


CHARACTER GET ROUTINE BEFORE
B*
PC SR AC XR YR SP
.:8FEB 33 00 D3 00 F6
.
0073 E6 7A INC $7A
0075 D0 02 BNE $0079
0077 E6 7B INC $7B
0079 AD 31 02 LDA $0231
007C C9 3A CMP #$3A
007E B0 0A BCS $008A
0080 C9 20 CMP #$20
0082 F0 EF BEQ $0073
0084 38 SEC
0085 E9 30 SBC #$30
0087 38 SEC
0088 E9 D0 SBC #$D0
008A 60 RTS
.
.

What we are going to do is alter that routine so that it no longer behaves in quite the same way.

As it stands at the moment, it interprets everything in the following way:

Locations $73-$77 : update the pointer in memory
locations $7A and $7B.

Locations $79-$7B : this is the pointer.

Locations $7C-$7F : if it's a colon or greater,
then end.

Locations $80-$83 : if it's a space, then loop
back to start again.

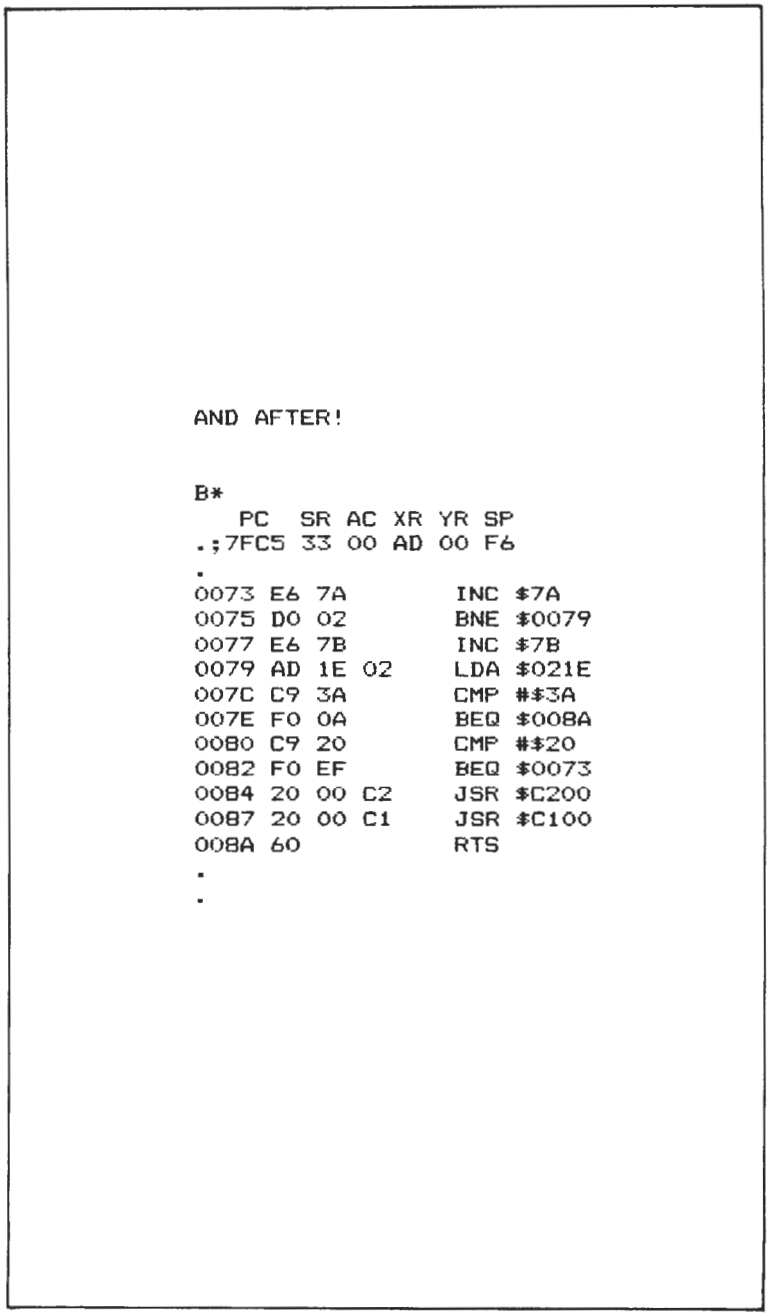
Locations $84-$8A : set flags for character type,
and return from subroutine.

Comments

This routine is the key to adding commands to Basic, since by altering it we can make it jump to some code of our own which will check for a special character, and if that character has been entered then do something! If we find that a special character has not been typed, then it's back to the routine again and carry on as normal.

We'll see later on how we can actually load a program into the computer which, when executed, alters the routine to behave in the way we want.

Instead, a couple of JSRs (jumps to subroutines) will be incorporated in it, and when we've finished with it it will look like this:



```
AND AFTER!


B*
   PC  SR AC XR YR SP
.;7FC5 33 00 AD 00 F6
.
0073 E6 7A        INC $7A
0075 D0 02        BNE $0079
0077 E6 7B        INC $7B
0079 AD 1E 02     LDA $021E
007C C9 3A        CMP #$3A
007E F0 0A        BEQ $008A
0080 C9 20        CMP #$20
0082 F0 EF        BEQ $0073
0084 20 00 C2     JSR $C200
0087 20 00 C1     JSR $C100
008A 60           RTS
.
.
```

It would be wise, at this point, to make an effort to get Extramon typed up and loaded into the computer, since this will make life a lot easier from now on. Without it we can still proceed with a lot of POKEs, but in order to see precisely what is happening, Extramon is a great help.

Altering the CHARGET routine

When everything is running normally, on pressing the Return key the system will come out of ROM into this routine to fetch the next character of Basic text, then trundle back into ROM again to ponder on its next move.

What will happen now is that the system will come out of ROM, to our changed subroutine, and when it hits the first JSR command it will jump to the routine sitting at location $C200 onwards. This will determine whether or not we're going to be interpreting a special command, and if we are jump somewhere else to process it.

If we're not, then the system goes back to the altered CHARGET routine, and finds that it now has to make yet another jump, this time to location $C100. This is simply a direct copy of what used to exist in the portion of CHARGET that we have changed, so that execution can continue as normal in the event of a special character not being found.

After that, it returns into ROM again to work out what will happen next.

The program to alter the CHARGET routine sits at locations $C10B onwards, and together with the direct replacement for the altered parts, which starts at location $C100, it all looks like this :

```
B*
   PC  SR AC XR YR SP
.:8FEB 33 00 D3 00 F6
.
C100 C9 3A        CMP #$3A
C102 B0 06        BCS $C10A
C104 38           SEC
C105 E9 30        SBC #$30
C107 38           SEC
C108 E9 D0        SBC #$D0
C10A 60           RTS
C10B A9 20        LDA #$20
C10D 85 84        STA $84
```

```
C10F 85 87       STA $87
C111 A9 00       LDA #$00
C113 85 85       STA $85
C115 85 88       STA $88
C117 A9 C2       LDA #$C2
C119 85 86       STA $86
C11B A9 C1       LDA #$C1
C11D 85 89       STA $89
C11F A9 F0       LDA #$F0
C121 85 7E       STA $7E
C123 A9 04       LDA #$04
C125 85 7A       STA $7A
C127 85 7B       STA $7B
.
.
```

The next routine that we need is the one to separate the extra code and the processing of that code from ordinary Basic. In this routine we check the current character being processed against a table stored at locations $C300 onwards, and if we find what we're looking for, branch to the appropriate subroutine by reading the most significant byte and least significant byte of the subroutine address from a table which is stored immediately after the character data.

```
B*
   PC  SR AC XR YR SP
.;371A 33 00 02 00 F6
.
C200 08          PHP
C201 86 04       STX $04
C203 A2 04       LDX #$04
C205 DD 00 C3    CMP $C300,X
C208 F0 07       BEQ $C211
C20A CA          DEX
C20B 10 F8       BPL $C205
C20D A6 04       LDX $04
C20F 28          PLP
C210 60          RTS
C211 BD 06 C3    LDA $C306,X
C214 8D 1E C2    STA $C21E
C217 BD 08 C3    LDA $C308,X
C21A 8D 1F C2    STA $C21F
C21D 20 00 C0    JSR $C000
C220 20 74 A4    JSR $A474
.
.
```

To explain what's happening, the program first of all saves the current status register onto the stack, and the current value held in the X register into location $0004 in the event of not finding a special character.

If that is the case, then everything is read back into the appropriate registers and it's off to the CHARGET routine again.

Finding special characters

However, if a special character is found then we branch out to location $C211 where we get the least significant byte and the most significant byte from our table. These are then stored at the appropriate registers, and then the program branches off to the subroutine to carry out the command.

The characters, and their LSBs and MSBs look like this :

```
B*
   PC  SR AC XR YR SP
.;26F4 33 00 DC 00 F6
.
C300 5F          ???
C301 21 21       AND ($21,X)
C303 21 21       AND ($21,X)
C305 00          BRK
C306 00          BRK
C307 C0 C0       CPY #$C0
.
.
```

This may not look very sensible as a disassembly, but it's the data that we're after, not the annotations.

The only thing we need now is a routine to execute, and in this case we've used an OLD routine. This can be used without the rest of this code by just typing in SYS49152, and it will then recover any program lost after a NEW command had been issued.

On the other hand, there's something infinitely more satisfying about seeing your own code being executed at the press of a key, rather than typing in boring old SYS commands all the time.

```
B*
   PC  SR AC XR YR SP
.;6F9F 33 00 87 00 F6
.
C000 A5 2B       LDA $2B
C002 A4 2C       LDY $2C
C004 85 22       STA $22
C006 84 23       STY $23
C008 A0 03       LDY #$03
C00A C8          INY
C00B B1 22       LDA ($22),Y
C00D D0 FB       BNE $C00A
C00F C8          INY
C010 98          TYA
C011 18          CLC
C012 65 22       ADC $22
C014 A0 00       LDY #$00
C016 91 2B       STA ($2B),Y
C018 A5 23       LDA $23
C01A 69 00       ADC #$00
C01C C8          INY
C01D 91 2B       STA ($2B),Y
C01F 88          DEY
C020 A2 03       LDX #$03
C022 E6 22       INC $22
C024 D0 02       BNE $C028
C026 E6 23       INC $23
C028 B1 22       LDA ($22),Y
C02A D0 F4       BNE $C020
C02C CA          DEX
C02D D0 F3       BNE $C022
C02F A5 22       LDA $22
C031 69 02       ADC #$02
C033 85 2D       STA $2D
C035 A5 23       LDA $23
C037 69 00       ADC #$00
C039 85 2E       STA $2E
C03B 60          RTS
.
.
```

Now that we've got everything together, it only remains to run the program by going to the various parts of it, and then, just by pressing the left arrow key and Return, we can instantly recover any program that may have been lost due to an accidental NEW.

To add yet more commands, you'll need to store more data for characters, and more data for LSBs and MSBs at locations $C3000 and onwards (or anywhere else for that matter, as long as the program

is pointed to the correct location!). The data for the characters is just the ASCII code for that character.

If, however, your forte is typing words rather than symbols, the following program shows how you might use the word BAK to get back a program, rather than typing in the left-arrow key.

```
B*
   PC  SR AC XR YR SP
.;8D55 31 72 9F 00 F6
.
C200 08          PHP
C201 86 04       STX $04
C203 A2 00       LDX #$00
C205 DD 00 C3    CMP $C300,X
C208 F0 26       BEQ $C230
C20A CA          DEX
C20B 10 F8       BPL $C205
C20D A6 04       LDX $04
C20F 28          PLP
C210 60          RTS
C211 BD 06 C3    LDA $C306,X
C214 8D 1E C2    STA $C21E
C217 BD 08 C3    LDA $C308,X
C21A 8D 1F C2    STA $C21F
C21D 20 C0 FF    JSR $FFC0
C220 E6 C9       INC $C9
C222 D0 02       BNE $C226
C224 E6 CA       INC $CA
C226 28          PLP
C227 A2 00       LDX #$00
C229 A1 C9       LDA ($C9,X)
C22B A6 04       LDX $04
C22D 60          RTS
C22E 00          BRK
C22F 00          BRK
C230 E6 7A       INC $7A
C232 D0 02       BNE $C236
C234 E6 7B       INC $7B
C236 A2 00       LDX #$00
C238 A1 7A       LDA ($7A,X)
C23A 38          SEC
C23B E9 41       SBC #$41
C23D F0 03       BEQ $C242
C23F 4C 5A C2    JMP $C25A
C242 E6 7A       INC $7A
C244 D0 02       BNE $C248
C246 E6 7B       INC $7B
C248 A2 00       LDX #$00
C24A A1 7A       LDA ($7A,X)
```

```
C24C 38          SEC
C24D E9 4B       SBC #$4B
C24F F0 03       BEQ $C254
C251 4C 08 AF    JMP $AF08
C254 20 00 C0    JSR $C000
C257 20 74 A4    JSR $A474
```

Early experiments

You could start your experiments with adding commands by trying to interface the graphics routines given earlier to act at the press of a key. Where commands will need parameters added to them, you could have another table of addresses to interpret things like !1, !2, !3 and so on where, having found that an '!' symbol has been entered, you then check to see what the next number is and go to the correct subroutine.

Where you'll need other parameters to be separated by commas, it is most practical to go to the internal ROM routines and let them do the checking, as was done with the graphics routine for setting up a high resolution screen in a suitable colour.

Function keys

We said earlier that we'd be giving you a program to use the function keys, and here it is. As it stands, it allows you to define the function keys to be any of the existing Basic keywords, although of course if you add your own commands you can also define a key to be one or more of those as well.

When you run the program by typing in SYS 49152, the prompt F1? will appear, at which point you enter whatever you want function key 1 to be (e.g. PRINT). If you want to make it equivalent to typing in PRINT and then hitting the Return key, enter PRINT followed by the left arrow key, which has been used in this program to stand for the Return key.

Appendix
Some Useful Information

Introduction

Some old, and some new material here, starting with:

M/C instruction set

The following notation applies to this summary:

A	Accumulator
X, Y	Index registers
M	Memory
P	Processor status register
S	Stack Pointer
√	Change
–	No change
+	Add
∧	Logical AND
−	Subtract
V	Logical Exclusive-OR
→, ←	Transfer to
∀	Logical (inclusive) OR
PC	Program counter
PCH	Program counter high
PCL	Program counter low
#dd	8-bit immediate data value (2 hexadecimal digits)
aa	8-bit zero page address (2 hexadecimal digits)
aaaa	16-bit absolute address (4 hexadecimal digits)
↑	Transfer from stack (Pull)
↓	Transfer onto stack (Push)

ADC

Add to Accumulator with Carry

Operation: A + M + C → A, C

N Z C I D V
✓ ✓ ✓ — — ✓

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	ADC	#dd	69	2	2
Zero Page	ADC	aa	65	2	3
Zero Page, X	ADC	aa,X	75	2	4
Absolute	ADC	aaaa	6D	3	4
Absolute, X	ADC	aaaa,X	7D	3	4*
Absolute, Y	ADC	aaaa,Y	79	3	4*
(Indirect, X)	ADC	(aa,X)	61	2	6
(Indirect), Y	ADC	(aa),Y	71	2	5*

*Add 1 if page boundary is crossed.

AND

AND Memory with Accumulator

Logical AND to the accumulator
Operation: A ∧ M → A

N Z C I D V
✓ ✓ — — — —

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	AND	#dd	29	2	2
Zero Page	AND	aa	25	2	3
Zero Page, X	AND	aa,X	35	2	4
Absolute	AND	aaaa	2D	3	4
Absolute, X	AND	aaaa,X	3D	3	4*
Absolute, Y	AND	aaaa,Y	39	3	4*
(Indirect, X)	AND	(aa,X)	21	2	6
(Indirect), Y	AND	(aa),Y	31	2	5*

*Add 1 if page boundary is crossed.

ASL

Accumulator Shift Left

Operation: C ← [7|6|5|4|3|2|1|0] ← 0

N Z C I D V
√ √ √ – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	0A	1	2
Zero Page	ASL aa	06	2	5
Zero Page, X	ASL aa,X	16	2	6
Absolute	ASL aaaa	0E	3	6
Absolute, X	ASL aaaa,X	1E	3	7

BCC

Branch on Carry Clear

Operation: Branch on C = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC aa	90	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BCC aaaa), and convert it to a relative address.

BCS

Branch on Carry Set

Operation: Branch on C = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS aa	B0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to next page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BCS aaaa), and convert it to a relative address.

BEQ

Branch on Result Equal to Zero

Operation: Branch on Z = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ aa	F0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to next page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BEQ aaaa), and convert it to a relative address.

BIT

Test Bits in Memory with Accumulator

Operation: A M, $M_7 \rightarrow N$, $M_6 \rightarrow V$
Bit 6 and 7 are transferred to the Status Register. If the result of A M is zero then Z = 1, otherwise Z = 0

N Z C I D V
M_7 √ – – – M_6

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT aa	24	2	3
Absolute	BIT aaaa	2C	3	4

BMI

Branch on Result Minus

Operation: Branch on N = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI aa	30	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BMI aaaa), and convert it to a relative address.

BNE

Branch on Result Not Equal to Zero

Operation: Branch on Z = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE aa	D0	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BNE aaaa), and convert it to a relative address.

BPL

Branch on Result Plus

Operation: Branch on N = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL aa	10	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BPL aaaa), and convert it to a relative address.

BRK

Force Break

Operation: Forced Interrupt PC + 2 ↓ P ↓

B N Z C I D V
1 – – – 1 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	00	1	7

BVC

Branch on Overflow Clear

Operation: Branch on V = 0

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC aa	50	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BVC aaaa), and convert it to a relative address.

BVS

Branch on Overflow Set

Operation: Branch on V = 1

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS aa	70	2	2*

*Add 1 if branch occurs to same page.
Add 2 if branch occurs to different page.
Note: AIM 65 will accept an absolute address as the operand (instruction format BVS aaaa), and convert it to a relative address.

CLC

Clear Carry Flag

Operation: 0 → C

N Z C I D V
– – 0 – – –

Addressing Mode	Assembly Language Form	OP CODE	No Bytes	No. Cycles
Implied	CLC	18	1	2

CLD

Clear Decimal Mode

Operation: 0 → D

N Z C I D V
– – – – 0 –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLI

Clear Interrupt Disable Bit

Operation: 0 → I

N Z C I D V
– – – 0 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLV

Clear Overflow Flag

Operation: 0 → V

N Z C I D V
– – – – – 0

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CMP

Compare Memory and Accumulator

Operation: A − M

N Z C I D V
√ √ √ − − −

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	CMP	#dd	C9	2	2
Zero Page	CMP	aa	C5	2	3
Zero Page, X	CMP	aa,X	D5	2	4
Absolute	CMP	aaaa	CD	3	4
Absolute, X	CMP	aaaa,X	DD	3	4*
Absolute, Y	CMP	aaaa,Y	D9	3	4*
(Indirect, X)	CMP	(aa,X)	C1	2	6
(Indirect), Y	CMP	(aa),Y	D1	2	5*

*Add 1 if page boundary is crossed.

CPX

Compare Memory and Index X

Operation: X − M

N Z C I D V
√ √ √ − − −

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	CPX	#dd	E0	2	2
Zero Page	CPX	aa	E4	2	3
Absolute	CPX	aaaa	EC	3	4

CPY

Compare Memory and Index Y

Operation: Y − M

N Z C I D V
√ √ √ − − −

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	CPY	#dd	C0	2	2
Zero Page	CPY	aa	C4	2	3
Absolute	CPY	aaaa	CC	3	4

DEC

Decrement Memory by One

Operation: M — 1 → M

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC aa	C6	2	5
Zero Page, X	DEC aa,X	D6	2	6
Absolute	DEC aaaa	CE	3	6
Absolute, X	DEC aaaa,X	DE	3	7

DEX

Decrement Index X by One

Operation: X — 1 → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY

Decrement Index Y by One

Operation: Y — 1 → Y

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR

Exclusive-OR Memory with Accumulator

Operation: A V M → A

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	EOR	#dd	49	2	2
Zero Page	EOR	aa	45	2	3
Zero Page, X	EOR	aa,X	55	2	4
Absolute	EOR	aaaa	4D	3	4
Absolute, X	EOR	aaaa,X	5D	3	4*
Absolute, Y	EOR	aaaa,Y	59	3	4*
(Indirect, X)	EOR	(aa,X)	41	2	6
(Indirect), Y	EOR	(aa),Y	51	2	5*

*Add 1 if page boundary is crossed.

INC

Increment Memory by One

Operation: M + 1 → M

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Zero Page	INC	aa	E6	2	5
Zero Page, X	INC	aa,X	F6	2	6
Absolute	INC	aaaa	EE	3	6
Absolute, X	INC	aaaa,X	FE	3	7

INX

Increment Index X by One

Operation: X + 1 → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY

Increment Index Y by One

Operation: Y + 1 → Y

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP Code	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP

Jump

Operation: (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JMP aaaa	4C	3	3
Indirect	JMP (aaaa)	6C	3	5

JSR

Jump to Subroutine

Operation: PC + 2↓, (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR aaaa	20	3	6

LDA

Load Accumulator with Memory

Operation: M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDA	#dd	A9	2	2
Zero Page	LDA	aa	A5	2	3
Zero Page, X	LDA	aa,X	B5	2	4
Absolute	LDA	aaaa	AD	3	4
Absolute, X	LDA	aaaa,X	BD	3	4*
Absolute, Y	LDA	aaaa,Y	B9	3	4*
(Indirect, X)	LDA	(aa,X)	A1	2	6
(Indirect), Y	LDA	(aa),Y	B1	2	5*

*Add 1 if page boundary is crossed.

LDX

Load Index X with Memory

Operation: M → X

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDX	#dd	A2	2	2
Zero Page	LDX	aa	A6	2	3
Zero Page, Y	LDX	aa,Y	B6	2	4
Absolute	LDX	aaaa	AE	3	4
Absolute, Y	LDX	aaaa,Y	BE	3	4*

*Add 1 when page boundary is crossed.

LDY

Load Index Y with Memory

Operation: M → Y

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	LDY	#dd	A0	2	2
Zero Page	LDY	aa	A4	2	3
Zero Page, X	LDY	aea,X	B4	2	4
Absolute	LDY	aaaa	AC	3	4
Absolute, X	LDY	aaaa,X	BC	3	4*

*Add 1 when page boundary is crossed.

LSR

Local Shift Right

Operation: 0 → | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | → C

N Z C I D V
0 ✓ ✓ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	LSR	A	4A	1	2
Zero Page	LSR	aa	46	2	5
Zero Page, X	LSR	aa,X	56	2	6
Absolute	LSR	aaaa	4E	3	6
Absolute, X	LSR	aaaa,X	5E	3	7

NOP

No Operation

Operation: No Operation (2 cycles)

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA

OR Memory with Accumulator

Operation: A V M → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	ORA	#dd	09	2	2
Zero Page	ORA	aa	05	2	3
Zero Page, X	ORA	aa,X	15	2	4
Absolute	ORA	aaaa	0D	3	4
Absolute, X	ORA	aaaa,X	1D	3	4*
Absolute, Y	ORA	aaaa,Y	19	3	4*
(Indirect, X)	ORA	(aa,X)	01	2	6
(Indirect), Y	ORA	(aa),Y	11	2	5*

*Add 1 on page crossing.

PHA

Push Accumulator on Stack

Operation: A ↓

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP

Push Processor Status on Stack

Operation: P↓

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	08	1	3

PLA

Pull Accumulator from Stack

Operation: A ↑

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP

Pull Processor Status from Stack

Operation: P ↑

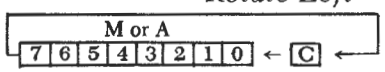
N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

Rotate Left

Operation:

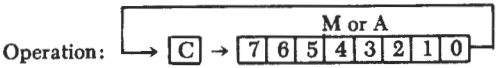


N Z C I D V
✓ ✓ ✓ – – –

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Accumulator	ROL	A	2A	1	2
Zero Page	ROL	aa	26	2	5
Zero Page, X	ROL	aa,X	36	2	6
Absolute	ROL	aaaa	2E	3	6
Absolute, X	ROL	aaaa,X	3E	3	7

ROR

Rotate Right

Operation: 

N Z C I D V
✓ ✓ ✓ – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR aa	66	2	5
Zero Page, X	ROR aa,X	76	2	6
Absolute	ROR aaaa	6E	3	6
Absolute, X	ROR aaaa,X	7E	3	7

RTI

Return from Interrupt

Operation: P↑ PC↑

N Z C I D V
From Stack

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	40	1	6

RTS

Return from Subroutine

Operation: PC↑, PC + 1 → PC

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	60	1	6

SBC

Subtract from Accumulator with Carry

Operation: A − M − $\overline{C}$ → A
Note: $\overline{C}$ = Borrow

N Z C I D V
✓ ✓ ✓ – – ✓

Addressing Mode	Assembly Language Form		OP CODE	No. Bytes	No. Cycles
Immediate	SBC	#dd	E9	2	2
Zero Page	SBC	aa	E5	2	3
Zero Page, X	SBC	aa,X	F5	2	4
Absolute	SBC	aaaa	ED	3	4
Absolute, X	SBC	aaaa,X	FD	3	4*
Absolute, Y	SBC	aaaa,Y	F9	3	4*
(Indirect, X)	SBC	(aa,X)	E1	2	6
(Indirect), Y	SBC	(aa),Y	F1	2	5*

*Add 1 when page boundary is crossed.

SEC

Set Carry Flag

Operation: 1 → C

N Z C I D V
– – 1 – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED

Set Decimal Mode

Operation: 1 → D

N Z C I D V
– – – – 1 –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI

Set Interrupt Disable Status

Operation: 1 → I

N Z C I D V
– – – 1 – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA

Store Accumulator in Memory

Operation: A → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA aa	85	2	3
Zero Page, X	STA aa,X	95	2	4
Absolute	STA aaaa	8D	3	4
Absolute, X	STA aaaa,X	9D	3	5
Absolute, Y	STA aaaa,Y	99	3	5
(Indirect, X)	STA (aa,X)	81	2	6
(Indirect), Y	STA (aa),Y	91	2	6

STX

Store Index X in Memory

Operation: X → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX aa	86	2	3
Zero Page, Y	STX aa,Y	96	2	4
Absolute	STX aaaa	8E	3	4

STY

Store Index Y in Memory

Operation: Y → M

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY aa	84	2	3
Zero Page, X	STY aa,X	94	2	4
Absolute	STY aaaa	8C	3	4

TAX

Transfer Accumulator to Index X

Operation: A → X

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY

Transfer Accumulator to Index Y

Operation: A → Y

N Z C I D V
√ √ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TSX

Transfer Stack Pointer to Index X

Operation: S → X

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA

Transfer Index X to Accumulator

Operation: X → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS

Transfer Index X to Stack Pointer

Operation: X → S

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

TYA

Transfer Index Y to Accumulator

Operation: Y → A

N Z C I D V
✓ ✓ – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

M/C mnemonics

6502 INSTRUCTION REPERTOIRE

MNEMONIC	FUNCTION
ADC	Add memory to accumulator with carry.
AND	AND memory with acuumulator.
ASL	Shift left 1 bit.
BCC	Branch on carry clear.
BCS	Branch on carry set.
BEQ	Branch on 0.
BIT	Test bits in memory with accumulator.
BMI	Branch on result negative.
BNE	Branch on result <> 0.
BPL	Branch on result positive.
BRK	Force break.
BVC	Branch on overflow clear.
BVS	Branch on overflow set.
CLC	Clear carry flag.
CLD	Clear decimal mode.
CLI	Clear interrupt disable.
CLV	Clear overflow flag.
CMP	Compare memory with accumulator.
CPX	Compare memory with X register.
CPY	Compare memory with Y register.
DEC	Decrement memory.
DEX	Decrement X register.
DEY	decrement Y register.
EOR	Exclusive OR memory with accumulator.
INC	Increment memory.
INX	Increment X register.
INY	Increment Y register.
JMP	Jump to specified location.
JSR	Jump to subroutine.
LDA	Load accumulator.
LDX	Load X register.
LDY	Load Y register.
LSR	Shift right 1 bit.
NOP	No operation.
ORA	OR memory with accumulator.
PHA	Push accumulator onto stack.
PHP	Push processor status onto stack.
PLA	Pull accumulator from stack.
PLP	Pull processor status from stack.

```
ROL     Rotate left 1 bit.
ROR     Rotate right 1 bit.
RTI     Return from interrupt.
RTS     Return from subroutine.

SBC     Subtract memory with borrow from accumulator.
SEC     Set carry flag.


SED     Set decimal mode.
SEI     Set interrupt disable.
STA     Store accumulator.
STX     Store X register.
STY     Store Y register.

TAX     Transfer accumulator to X register.
TAY     Transfer accumulator to Y register.
TSX     Transfer stack pointer to X register.
TXA     Transfer X register to accumulator.
TXS     Transfer X register to stack pointer.
TYA     Transfer Y register to accumulator.
```

6510 Flag Guide

N-Negative Flag

Instruction to condition				Instruction to test
ADC	DEY	LSR	TAX	BMI
AND	EOR	ORA	TAY	BPL
ASL	INC	PLA	TSX	
CMP	INX	PLP	TXA	
CPX	INY	ROL	TYA	
CPY	LDA	ROR		
DEC	LDX	RTI		
DEX	LDY	SBC		

V-Overflow Flag

Instruction to condition			Instruction to test
ADC	CLV	RTI	BVC
BIT	PLP	SBC	BVS

B-BREAK Flag

Instruction to condition	Instruction to test
BRK PLP RTI	

D-Decimal Flag

Instruction to condition	Instruction to test
CLD PLP RTI SED	

I-Interrupt Flag

Instruction to condition	Instruction to test
BRK CLI PLP RTI SEI	

Z-Zero Flag

Instruction to condition	Instruction to test
ADC DEC LDA ROL AND DEX LDX ROR ASL DEY LDY RTI BIT EOR LSR SBC CMP INC ORA TAX CPY INX PLA TAY CPX INY PLP TXA TYA	BEQ BNE

C-Carry Flag

Instruction to condition	Instruction to test
ADC CPX ROL SEC ASL CPY ROR BCS CLC LSR RTI CMP PLP SBI	BCC

Hex/Dec convertor

Decimal & Hexadecimal Conversions

HEXADECIMAL COLUMNS											
6		5		4		3		2		1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,897	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

Notes.

To convert from hexadecimal to decimal, first find the corresponding column position for each hexadécimal digit. Make a note of the decimal equivalents, then add the noted values together to obtain the converted decimal value.

To convert from decimal to hexadecimal, find the largest decimal value in the table that will fit into the number to be converted. Next make a note of the hex equivalent and column position. Calculate the decimal remainder, and repeat the process on this and any subsequent remainders.

Index

Duckworth Home Computing

ADVANCED BASIC & MACHINE CODE FOR THE 64
by Peter Gerrard

For the more serious user of the Commodore 64, this book teaches you all about programming in machine code. It includes sections on building a database, double precision arithmetic, flags and registers, logic operators, indirect addressing, built-in subroutines, adding commands to Basic and animation. The all-important link to Basic is not forgotten: the opening chapters show you how to improve your Basic programming techniques, offering many program examples.

Peter Gerrard, former editor of *Commodore Computing International*, is the author of two top-selling adventure games for the Commodore 64 and a regular contributor to *Personal Computer News, Which Micro? and Software Review* and *Commodore Horizons*.

ISBN 0-7156-1785-0

9 780715 617854

Duckworth
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 1785 0

IN UK ONLY £6.95 NET