# INTRODUCING COMMODORE 64™ MACHINE CODE

IAN SINCLAIR

A Spectrum Book

# Introducing Commodore 64 Machine Code

# Introducing Commodore 64 Machine Code

**Ian Sinclair**


PRENTICE-HALL, INC.
Englewood Cliffs, New Jersey 07632

*Library of Congress Cataloging in Publication Data*

Sinclair, Ian Robertson.
Introducing Commodore 64 machine code.

"A Spectrum Book."
Includes index.
1. Commodore 64 (Computer)—Programming. 2. Basic (Computer program language) I. Title. II. Title: Introducing Commodore sixty-four machine code.
ISBN 0-13-477316-0



10 9 8 7 6 5 4 3 2 1

ISBN 0-13-477316-0

Prentice-Hall International, Inc., London
Prentice-Hall of Australia Pty. Limited, Sydney
Prentice-Hall Canada Inc., Toronto
Prentice-Hall of India Private Limited, New Delhi
Prentice-Hall of Japan, Inc., Tokyo
Prentice-Hall of Southeast Asia Pte. Ltd., Singapore
Whitehall Books Limited, Wellington, New Zealand
Editora Prentice-Hall do Brasil Ltda., Rio de Janeiro

# Contents

# Preface

Many computer users are content to program in BASIC for all of their computing lives. A large number of others are eager to find out more about computing and their computer than the use of BASIC can lead to. Few, however, make much progress to the use of machine code, which allows so much more control over the computer. The reason for this, to my mind, is that so many books which deal with machine code programming seem to start with the assumption that the reader is already familiar with the ideas and the words of this type of programming. In addition, many of these books treat machine code programming as a study in itself, leaving the reader with little clue as to how to apply machine code to his or her computer.

This book has two main aims. One is to introduce the Commodore 64 owner to some of the details of how the Commodore 64 works, so allowing for more effective programming even without delving into machine code. The second aim is to introduce the methods of machine code programming in a simple way. I must emphasise the word 'introduce'. No single book can tell you all about machine code, even the machine code for one computer. All I can claim is that I can get you started. Getting started means that you will be able to write short machine code routines, understand machine code routines that you see printed in magazines, and generally make more effective use of your Commodore 64. It also means that you will be able to make use of the many more advanced books on the subject, and progress to greater mastery of this fascinating topic.

Understanding the operating system of your Commodore 64, and having the ability to work in machine code can open up an entirely new world of computing to you. This is why you find that most of the really spectacular games are written in machine code. You will also find that many programs which are written mainly in BASIC will

incorporate pieces of machine code in order to make use of its greater speed and better control of the computer.

I am most grateful to several people who made the production of this book possible. Of these, Richard Miles of Granada Publishing commissioned the book and produced a Commodore 64 and matching printer. He and Sue Moore, also of Granada, then did their usual miracles of meticulous effort on my manuscript. I am also most grateful to someone I have never met – Milton Bathurst, whose book *Inside The Commodore 64* has been a most valuable reference work.

Ian Sinclair

# Chapter One
# ROM, RAM, Bytes and Bits

One of the things that discourage computer users from attempts to go beyond BASIC is the number of new words that spring up. The writers of many books on computing, especially on machine code computing, seem to assume that the reader has an electronics background and will understand all of the terms. I shall assume that you have no such background. All that I shall assume is that you possess a Commodore 64, and that you have some experience in programming your Commodore 64 in BASIC. This means that we start at the correct place, which is the beginning. I don't want in this book to have to interrupt important explanations with technical or mathematical details, and these will be found in the Appendices. This way, you can read the full explanation of some points if you feel inclined, or skip them if you are not.

To start with, we have to think about memory. A unit of memory for a computer is, as far as we are concerned, just an electrical circuit that acts like a switch. You walk into a room, switch a light on, and you never think that it's remarkable in any way that the light stays on until you switch it off. You don't go about telling your friends that the light circuit contains a memory – and yet each memory unit of a computer is just a kind of miniature switch that can be turned on or off. What makes it a memory is that it will stay the way it has been turned, on or off, until it is changed. One unit of computer memory like this is called a *bit* – the name is short for *binary digit*, meaning a unit that can be switched one of two possible ways.

We'll stick with the idea of a switch, because it's very useful. Suppose that we wanted to signal with electrical circuits and switches. We could use a circuit like the one in Fig. 1.1. When the switch is on, the light is on, and we might take this as meaning YES. When the switch is turned off, the light goes out, and we might take this as meaning NO. You could attach any two meanings that you liked to these two conditions (called 'states') of the light, so long as

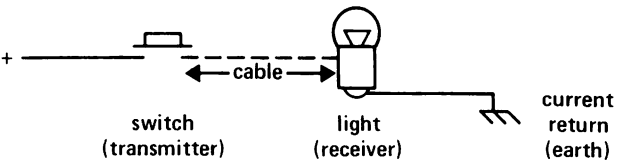
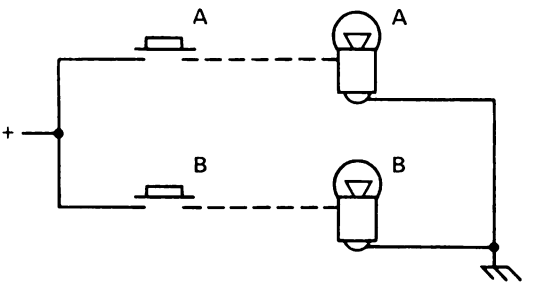



*Fig. 1.1.* A single-line switch and bulb signalling system.




| A | B |
|---|---|
| off | off |
| off | on |
| on | off |
| on | on |

*Fig. 1.2.* Two-line signalling. Four possible signals can be sent.

there are only two. Things improve if you can use two switches and two lights, as in Fig. 1.2. Now four different combinations are possible: (a) both off; (b) A on, B off; (c) A off, B on; (d) both on. This set of four possibilities means that we could signal four different meanings. Using one line allows two possible codes, using two lines allows four codes. If you feel inclined to work them all out, you'll find that using three lines will allow eight different codes. A moment's thought suggests that since 4 is 2×2, and eight is 2×2×2, then four lines might allow 2×2×2×2, which is 16, codes. It's true, and since we usually write 2×2×2×2 as $2^4$ (two to the power 4), we can find out how many codes could be transmitted by any number of lines. We would expect eight lines, for example, to be able to carry $2^8$ codes, which is 256. A set of eight switches, then, could be arranged so as to convey 256 different meanings. It's up to us to decide how we might want to use these signals.

One particularly useful way is called *binary code*. Binary code is a way of writing numbers using only two digits, ∅ and 1. We can think of ∅ as meaning 'switch off' and 1 as meaning 'switch on', so that 256 different numbers could be signalled, using eight switches, by thinking of ∅ as meaning off and 1 as meaning on. This group of eight is called a *byte*, and it's the quantity that we use to specify the memory size of our computers. This is why the numbers 8 and 256 occur so much in machine code computing.

The way that the individual bits in a byte are arranged so as to indicate a number follows the same way as we use to indicate a number normally. When you write a number such as 256, the 6 means six units, the 5 is written to the immediate left of the 6 and means five tens, and the 2 is written one more place to the left and means two hundreds. These positions indicate the importance or significance of a digit, as Fig. 1.3 shows. The 6 in 256 is called the 'least significant digit', and the 2 is the 'most significant digit'. Change the 6 to 7 or 5, and the change is just one part in 256. Change the 2 to 1 or 3 and the change is one hundred parts in 256 – much more important.

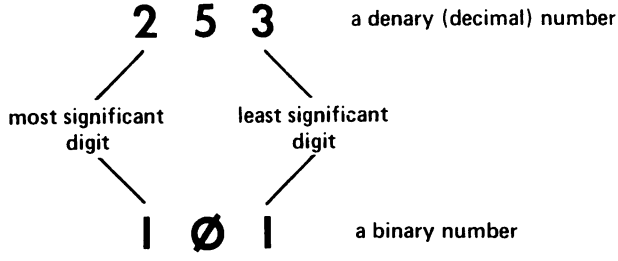



*Fig. 1.3.* The significance of digits. Our numbering system uses the position of a digit in a number to indicate its significance or importance.

Having looked at bits and bytes, it's time to go back to the idea of memory as a set of switches. As it happens, we need two types of memory in a computer. One type must be permanent, like mechanical switches or fixed connections, because it has to be used for retaining the number-coded instructions that operate the computer. This is the type of memory that is called *ROM*, meaning read-only memory. This implies that you can find out and copy what is in the memory, but not delete it or change it. The ROM is the most important part of your computer, because it contains all the instructions that make the computer carry out the actions of BASIC.

When you write a program for yourself, the computer stores it in the form of another set of number-coded instructions in a part of memory that can be used over and over again. This is a different type of memory that can be 'written' as well as 'read', and if we were logical about it we would refer to it as RWM, meaning read-write memory. Unfortunately, we're not very logical, and we call it *RAM* (meaning random-access memory). This was a name that was used in the very early days of computing to distinguish this type of memory from one which operated in a different way. We're stuck with the name of RAM now and probably forever!

## The number-code caper

Now we can get back to the bytes. We saw earlier that a byte, which is a group of eight bits, can consist of any one of 256 different arrangements of these bits. The most useful arrangement, however, is one that we call *binary code*. These different arrangements of bits in binary code represent numbers which we write in ordinary form as Ø to 255 (not 1 to 256, because we need a code for zero). Each byte of the 38929 bytes of RAM that are available in the Commodore 64 can store a number which is in this range of Ø to 255.

Numbers by themselves are not of much use, and we wouldn't find a computer particularly useful if it could deal only with numbers between Ø and 255, so we make use of these numbers as codes. Each number code can, in fact, be used to mean several different things. If you have worked with ASCII codes in BASIC, you will know that each letter of the alphabet, each of the digits Ø to 9, and each punctuation mark, is coded in ASCII as a number between 32 (the space) and 127 (the left-arrow). That selection leaves you with a large number of ASCII code numbers which can be used for other purposes such as graphics characters. The ASCII code is not the only one, however. The Commodore 64 uses its own coded meanings for numbers in this range of Ø to 255. For example, when you type the word PRINT in a program line, what is placed in the memory of the Commodore 64 (when you press ENTER) is not the sequence of ASCII codes for PRINT. This would be 8Ø,82,73,78,84, one byte for each letter. What is put into memory, in fact, is one byte – the binary form of the number 153. This single byte is called a *token* and it can be used by the computer in two ways. One way is to locate the ASCII codes for the characters that make up the word PRINT. These are stored in the ROM, so that when you LIST a

program, you will see the word PRINT appear, not a character whose code is 153. The other, even more important, use of the 'token' is to locate a set of instructions which are also held in the ROM in the form of number codes. These instructions will cause characters to be printed on the screen, and the numbers that make up these codes are what we call *machine code*. They control directly what the 'machine' does. That direct control is our reason for wanting to use machine code. When we use BASIC, the only commands we can use are the ones for which 'tokens' are provided. By using machine code, we can make up our own commands and do what we please.

Incidentally, the fact that PRINT generates one 'token' is the reason why it is possible to use ? in place of PRINT. The Commodore 64 has been designed so that a ? which is not placed between quotes will also cause 153 to be put into memory.

## Do-it-yourself spot

As an aid to digesting all that information, try a short program. This one, in Fig. 1.4, is designed to reveal the keywords that are stored in the ROM, and it makes use of the BASIC instruction word PEEK.

```
10 PRINT41118;" ";:FOR N=41118 TO 41373
20 K=PEEK(N)
30 IF K<128THENPRINTCHR$(K);
40 IFK>=128THENPRINTCHR$(K-128):PRINTN+1;" ";
50 NEXT
```

*Fig. 1.4.* A program that reveals the keywords of Commodore 64 BASIC.

PEEK has to be followed by a number or number variable within brackets, and it means 'find what byte is stored at this address number'. All of the bytes of memory within your Commodore 64 are numbered from zero upwards, one number for each byte. Because this is so much like the numbering of houses in a road, we refer to these numbers as *addresses*. The action of PEEK is to find what number, which must be between Ø and 255, is stored at each address. The Commodore 64 automatically converts these numbers from the binary form in which they are stored into the ordinary decimal (more correctly, *denary*) numbers that we normally use. By using CHR$ in our program, we can print the character whose ASCII code is the number we have PEEKed at. The program uses the

variable N as an address number, and then checks that PEEK(N) gives a number less than 128 – in other words a number which is an ASCII code. If it is, then the character is printed.

Now the reason that we have to check is that the last character in each set of words, or word, is coded in a different way. The number that we find for the last character has had 128 added to the ASCII code. For example, the first three address locations that the program PEEKs at contain the numbers 69, 78 and 196. The number 69 is the ASCII code for E, 78 is the code for N, and then 196–128=68, which is the ASCII code for D. This is where the word END is stored, then. The reason for treating the last letter so differently is to save memory! If a gap were left between words, this would be a byte of memory wasted. As it is, there is no waste, because the last letter of a group always has a code number that is greater than 128, so the computer can recognise it easily. We have followed the same scheme in the BASIC program of Fig. 1.4 by using line 4Ø to print the correct letter and to take a new line and print the address number. There is another set of numbers stored earlier on, which consists of more addresses. These are the addresses of subroutines which carry out the actions of BASIC, and they are stored in the same order as these words.

## C 64 cutaway

Now take a look at a diagram of the Commodore 64 in Fig. 1.5. It's quite a simple diagram because I've omitted all of the detail, but it's enough to give us a clue about what's going on inside. This is the type of diagram that we call a *block diagram*, because each unit is drawn as a block with no details about what may be inside. Block diagrams are like large-scale maps which show us the main routes between towns but don't show side-roads or town streets. A block diagram is enough to show us the main paths for electrical signals in the computer.

The names of two of the blocks should be familiar already, ROM and RAM, but the other two are not. The block that is marked MPU is a particularly important one. MPU means Microprocessor Unit – but some block diagrams use the letters CPU (Central Processing Unit). The MPU is the main 'doing' unit in the system, and it is, in fact, one single unit. The MPU is a single plug-in chunk, one of these silicon chips that you read about, encased in a slab of black plastic and provided with forty connecting pins that are arranged in two

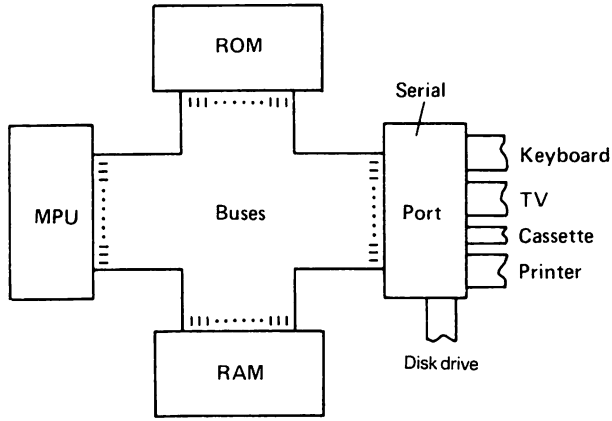



*Fig. 1.5.* A block diagram of C 64. The connections marked 'Buses' consist of a large number of connecting links which join all of the units of the system.

rows of twenty (Fig. 1.6). There are several different types of MPU made by different manufacturers, and the one in your Commodore 64 is called 6502 (or 6502A).

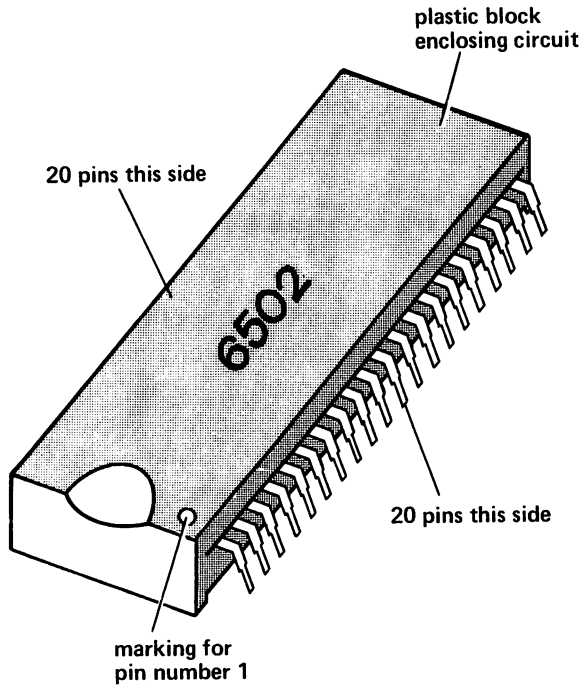



*Fig. 1.6.* The 6502 MPU. The actual working part is smaller than a fingernail, and the larger plastic case (52 mm by 14 mm wide) makes it easier to work with.

What does the MPU do? The answer is practically everything, and yet the actions that the MPU can carry out are remarkably few and simple. The MPU can *load* a byte, meaning that a byte which is stored in the memory can be copied into another store in the MPU. The MPU can also *store* a byte, meaning that a copy of a byte that is stored in the MPU can be placed in any address in the memory.

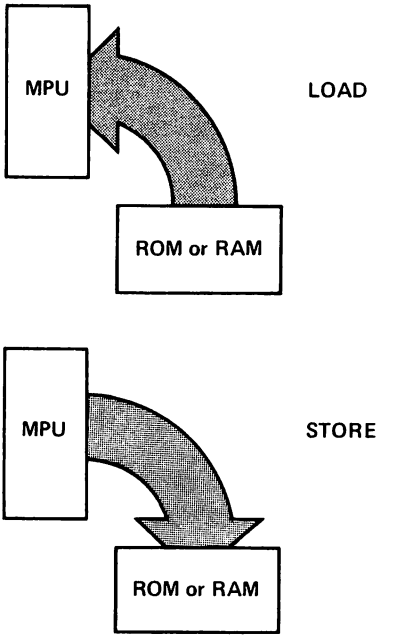



*Fig. 1.7.* Loading and storing. 'Loading' means signalling to the MPU from the memory, so that the digits of a byte are copied into the MPU. 'Storing' is the opposite process.

These two actions (see Fig. 1.7) are the ones that the MPU spends most of its working life in carrying out. By combining them, we can copy a byte from any address in memory to any other. You don't think that's very useful? That copying action is just what goes on when you press the letter H on the keyboard and see the H appear on the screen. The MPU treats the keyboard as one piece of memory and the screen as another, and copies bytes from one to the other as you type. That's a considerable simplification, but it will do for now – just to show how important the action is.

Loading and storing are two very important actions of the MPU, but there are several others. One set of actions is the *arithmetic set*. For most types of MPU, these consist of addition and subtraction only, using only single-byte numbers. Since a single-byte number

means a number between Ø and 255, how does the computer manage to carry out actions like multiplication of large numbers, division, raising to powers, logarithms, sines, and all the rest? The answer is by machine code programs that are stored in the ROM. If these programs were not there you would have to write your own. There aren't many computer users who would like to set about that task.

There's also the *logic set*. MPU logic is, like all MPU actions, simple and subject to rigorous rules. Logic actions compare the bits of two bytes and produce an 'answer' which depends on the bit values that are being compared and on the logic rule that is being followed. The three main logic rules are called AND, OR and XOR, and Fig. 1.8 shows how they are applied.

Another set of actions is called the *jump set*. A jump means a change of address, rather like the action of GOTO in BASIC. A combination of a test and a jump is the way that the MPU carries out its decision steps. Just as you can program in BASIC:

```
IF A = 36 THEN GOTO 1Ø5Ø
```

so the MPU can be made to carry out an instruction which is at an entirely different address from the normal next address. The MPU is a programmed device, meaning that it carries out each of its actions as a result of being fed with an instruction byte which has been stored in the memory. Normally when the MPU is fed with an instruction from an address somewhere (usually in ROM), it carries out the instruction and then reads the instruction byte that is stored in the next address up. A jump instruction would prevent this from happening, and would instead cause the MPU to read from another address, the one that was specified in the jump instruction. This jump action can be made to depend on the result of a test. The test will usually be carried out on the result of the previous action, whether it gave a zero, positive or negative result, for example.

That isn't a very long or exciting list, but the actions that I've omitted are either unimportant at this stage, or not particularly different from the ones in the list. What I want to emphasise is that the magical microprocessor isn't such a very smart device. What makes it so vital to the computer is that it can be programmed and that it can carry out its actions very quickly. Equally important is the fact that the microprocessor can be programmed by sending it electrical signals.

These signals are sent to eight pins, called the *data pins*, of the MPU. It doesn't take much of a guess to realise that these eight pins correspond to the eight bits of a byte. Each byte of the memory can

AND

The result of ANDing two bits will be 1 if both bits are 1, Ø otherwise:

1 AND 1 = 1 { 1 AND Ø = Ø ; Ø AND 1 = Ø } Ø AND Ø = Ø

For two bytes, corresponding bits are ANDed

```
      1Ø11Ø111
AND   ØØØØ1111
      ØØØØØ111
```

only these bits exist in both bytes.

OR

The result of ORing two bits will be 1 if either or both bits is 1, Ø otherwise:

1 OR 1 = 1 { 1 OR Ø = 1 ; Ø OR 1 = 1 } Ø OR Ø = Ø

For two bytes, corresponding bits are ORed

```
      1Ø11Ø111
OR    ØØØØ1111
      1Ø111111
```

only bit which is Ø in both.

XOR (Exclusive-OR)

Like OR, but result is zero if the bits are identical

1 XOR 1 = Ø { 1 XOR Ø = 1 ; Ø XOR ! = 1 } Ø XOR Ø = Ø

```
      1Ø11Ø111
XOR   ØØØØ1111
      1Ø111ØØØ
```

if two bits are identical the result is zero.

*Fig. 1.8.* The rules for the three logic actions, AND, OR and XOR.

therefore affect the MPU by sharing its electrical signals with the MPU. Since this is a long-winded description of the process, we call it 'reading'. Reading means that a byte of memory is connected along eight lines to the MPU, so that each 1 bit will cause a 1 signal on a data pin, and each ∅ bit will cause a ∅ signal on a data pin. Just as reading a paper or listening to a recording does not destroy what is written or recorded, reading a memory does not change the memory in any way, and nothing is taken out. The opposite process of writing does, however, change the memory. Like recording a tape, writing wipes out whatever existed there before. When the MPU writes a byte to an address in the memory, whatever was formerly stored at that address is no longer there; it has been replaced by the new byte. This is why it is so easy to write new BASIC lines replacing old ones at the same line number.

## Table d'Hôte?

Do you really write programs in BASIC? It might sound like a silly question, but it's a serious one. The actual work of a program is done by coded instructions to the MPU, and if you write only in BASIC, you don't write these. All that you do is to select from a menu of choices that we call the BASIC keywords, and arrange them in the order that you hope will produce the correct results. Your choice is limited to the keywords that are designed into the ROM. We can't alter the ROM, and if we want to carry out an action that is not provided for by a keyword, we must either combine a number of keywords (a BASIC program) or operate directly on the MPU with number codes (machine code). When you have to carry out actions by combining a number of BASIC commands, the result is clumsy, especially if each command is a collection of other commands. Direct action is quick, but it can be difficult. The direct action that I am talking about is machine code, and a lot of this book will be devoted to understanding this 'language' which is difficult just because it's simple!

Take a situation which will illustrate this paradox. Suppose you want a wall built. You could ask a builder. Just tell him that you want a wall built across the back garden, and then sit back and wait. This is like using BASIC with a command-word for 'build a wall'. There's a lot of work to be done, but you don't have to bother about the details.

Now think of another possibility. Suppose you had a robot which

could carry out instructions mindlessly but incredibly quickly. You couldn't tell it to 'build a wall' because these instructions are beyond its understanding. You have to tell it in detail, such as: 'Stretch a line from a point 85 feet from the kitchen edge of the house, measured along the fence southwards, to a point 87 feet from the lounge end of the house measured along that fence southwards. Dig a trench eighteen inches deep and one foot wide along the path of your line. Mix three bags of sand and two of cement with four barrow-loads of pebbles for three minutes. Mix water into this until a pail filled with the mixture will take ten seconds to empty when held upside down. Fill the trench with the mixture ...' The instructions are very detailed – they have to be for a brainless robot – but they will be carried out flawlessly and quickly. If you've forgotten anything, no matter how obvious, it won't be done. Forget to specify how much mortar, what mixture and where to place it, and your bricks will be put up without mortar. Forget to specify the height of the wall, and the robot will keep piling one layer on top of another, like the Sorcerer's Apprentice, until someone sneezes and the whole wall falls down.

The parallel with programming is remarkably close. One keyword in BASIC is like the 'build a wall' instruction to the builder. It will cause a lot of work to be done, drawing on a lot of instructions that are not yours – but it may not be done as fast as you like. If you can be bothered with specifying the detail, machine code is a lot faster because you are giving your instructions direct to an incredibly fast but mindless machine, the microprocessor. We can stretch the similarity further. If you said to your builder 'mend the car', he might be unwilling or unable to do so. The correct set of detailed instructions to the robot would, however, get this job done. Machine code can be used to make your computer carry out actions that are simply not provided for in BASIC, though it's fair to say that many modern computers allow a much greater range of commands than early models, and this aspect of machine code is not quite so important as it used to be.

One last look at the block diagram is needed before we start on the inner workings of the Commodore 64. The block which is marked 'Port' includes more than one chip. A port in computing language means something that is used to pass information, one byte at a time, into or out from the rest of the system – the MPU, ROM and RAM. The reason for having a separate section to handle this is that inputs and outputs are important but slow actions. By using a port we can let the microprocessor choose when it wants to read an input or write

an output. In addition, we can isolate inputs and outputs from the normal action of the MPU. This is why nothing appears on the screen in a BASIC program except where we have a PRINT command in the program. It's also why pressing the PLAY key of the cassette recorder has no effect until you type LOAD (with a filename) and press ENTER. The port keeps the action of the computer hidden from you until you actually need to have an input or an output.

We have now looked at all of the important sections of your Commodore 64. I've used some terms loosely – purists will object to the way I've used the word 'port', for example – but no-one can quarrel with the actions that are carried out. What we have to do now is to look at how the computer is organised to make use of the MPU, ROM, RAM and ports so that it can be programmed in BASIC and can run a BASIC program. It looks like a good place to start another chapter!

Chapter Two

# Digging Inside the Commodore 64

I don't mean 'digging inside' literally – you don't have to open up the case! What I do mean is that we are going to look at how the Commodore 64 is designed to load and run BASIC programs. We'll start with a simplified version of the action of the whole system, omitting details for the moment.

The ROM of your Commodore 64, which starts at address 4ø96ø, consists of a large number of short programs – *subroutines* – which are written in machine code, along with sets of values (tables) like the table of keywords. There will be at least one machine code subroutine for each keyword in BASIC, and some of the keywords may require the use of many subroutines. When you switch on your Commodore 64, the piece of machine code that is carried out is called the 'initialisation routine'. This is a long piece of program but, because machine code is fast, carrying out instructions at the rate of many thousands per second, you see very little evidence of all this activity. All that you notice is a delay between switching on and seeing the MICROSOFT copyright notice. In this brief time, though, the action of the RAM part of the memory has been checked, some of the RAM has been 'written' with bytes that will be used later, and most of the RAM has been cleared for use. Cleared for use does not mean that nothing is stored in the RAM. When you switch off the computer, the RAM loses all trace of stored signals, but when you switch on again the memory cells don't remain storing zeros. In each byte, some of the bits will switch to 1 and some will switch to ø when power is applied. This happens quite at random, so that if you could examine what was stored in each byte just after switching on, you would find a set of meaningless numbers. These would consist of numbers in the range ø to 255, the normal range of numbers for a byte of memory. These numbers are 'garbage' – they weren't put into memory deliberately, nor do they form useful instructions or data. The first job of the computer, then, is to clean up. In place of the

random numbers, the computer substitutes a very much more ordered pattern of thirty-two bytes of 255 followed by thirty-two bytes of ∅. Try this – switch on, and type (no line number):

```
FOR N =2∅49 TO 2249:?PEEK(N); "  ";:NEXT
```

and then press RETURN. The range of memory addresses we have used is the 'start of BASIC' range, where the first bytes of a BASIC program are normally stored. If we have just switched on, and haven't used a line number for the command, there will be nothing stored here except for the pattern that was left after the initialisation and as a result of the command. As you'll see on the screen, the pattern consists mainly of the chain of 255's and ∅'s. At the start, though, and scattered through the memory, you will find other numbers. The numbers at the start are placed there by the FOR N=2∅49... command that you used to discover the pattern, and the other numbers are put there by the action of this command as it is executed.

The initialising program has a lot more to do. The first section of RAM, from address ∅ to 818, is for 'system use'. This is because the machine code subroutines which carry out the actions of BASIC need to store quantities in memory as they are working. Address numbers 43 and 44, for example, hold the address of the first byte of a BASIC program. A much larger section of the memory is used for storing numbers that make text and graphics appear on the screen. In addition, some RAM has also to be used to hold quantities that are created when a program runs. That's what we are going to look at now.

SEE P.24

## Variables on the table

BASIC programs make a lot of use of variables, meaning the use of letters to represent numbers and words. Each time you 'declare a variable' by using a line like:

```
N=2∅   or   A$="SMITH"
```

the computer has to take up memory space with the name (N or A$ or whatever you have used) and the value (like 2∅ or SMITH) that you have assigned to it. The piece of memory that is used to keep track of variables is called the *variable list table* (VLT). It doesn't occupy any fixed place in the memory, but is stored in the free space just above your program. If you add one more line to your program,

the VLT address has to be moved to a set of higher address numbers. If you delete a line from your program, the VLT will be moved down in the same way so that it is always kept just following the last line of BASIC.

Now because the variable list table address can and does move around as the program is altered, the computer must at all times keep a note of where the table starts. This is done by using two bytes of a piece of memory that is reserved for system use, the addresses 45 and 46. You may wonder why two addresses are used. The reason is that one byte can hold a number only up to 255 in value. If we use two bytes, however, we can hold the number of 256's in one byte and the remainder in the other. A number like 257, for example, is one 256 and one remaining. We could code this as 1,1. This means that a 1 is stored in the byte that is reserved for 256's, and 1 in the byte reserved for units. The order of storing the numbers is low-byte then high-byte. To find the number that is stored, we multiply the second byte by 256 and add the first byte. For example, if you found 3,8 stored in two consecutive addresses that were used in this way, this would mean the number:

SEE P. 23

8*256 + 3 = 2Ø51

The biggest number that we can store using two bytes like this is 255,255, which means 255*256+255=65535. This is the reason that you can't use very large numbers like 7ØØØØ as line numbers in the Commodore 64 - the operating system uses only two bytes to store its line numbers. In fact, for other reasons, the maximum number that you can use is 63999.

All of this means that we can find the address that is stored in addresses 45 and 46 by using the formula:

?PEEK(46)*256+PEEK(45)

If you use this just after you have switched on your Commodore 64, then the result on the C 64 is the address number 2Ø51. This is just above the address at which the first byte of a BASIC program would be stored. To see this in action, type the line:

1Ø N=2Ø

and try ?PEEK(46)*256+PEEK(45) again. If you typed this line as I did, with a space between the line number and the 'N', then the address that you get is 2Ø6Ø. The variable list table has moved upwards in the memory, by 9 bytes. That's more than the number of bytes that you typed, you'll notice - the reasons for this will come later.

VLT = WHERE VARIABLE ARE KEPT

Quite a lot of important addresses that the computer uses are *dynamically allocated* like this. 'Dynamically allocated' means that the computer will change the place where groups of bytes are to be kept. It will then keep track of where they have been stored by altering an address that is held in a pair of bytes such as this example. This has important implications for how you use your computer. For example, if you shift the VLT by poking new numbers into addresses 45 and 46, the computer can't find its variable values. Try this - after finding the VLT address, but without running the one-line program, 1Ø N=2Ø, type ?N. The answer will be zero. Why? Because the program has not been run. The address 2Ø6Ø is where the VLT will start, but there's no VLT created until the program runs. This makes it easy for you to add or delete lines at this stage. All that will have to be altered is the pair of numbers in addresses 45 and 46. The VLT values are put in place only when the program runs, and a new table is created all over again each time you type RUN and press RETURN. Each time a new table is created, a new pair of bytes will be put into 45 and 46. That's why you can't resume a program after editing - you have to RUN again to create a new VLT at a new address. If you RUN the one-line program now, and then type ?N you will get the expected answer of 2Ø. Now type (no line number) POKE 46,9, and press RETURN. This has changed the address of the VLT to an address where there is no VLT. Try ?N and see what you get. On my C 64, it was a very large number, because the correct value of variable N can be found no longer. If your Commodore 64 locks up during this exercise, then you may have to switch off to regain control. This seldom happens but if you do have to switch off, the program will be lost. Note, incidentally, the use of POKE to place a new value into a memory address. The correct form of the command is POKE A,D. A is an address, and will be in the range Ø to 4Ø959 (the range of values of RAM memory) for the C 64. D is the data that you place into this memory address, and it must be a value between Ø and 255. If you try to poke a number greater than 255, you will get an 'FC ERROR' message instead.

## A look at the table

It's time now to do something more constructive, and take a look at what is stored in the VLT. When we do these investigations, it's important to ensure that the computer is clear of the results of previous work. Therefore, it's advisable to switch off and then on

again, before each effort. Simply pressing the RESTORE key does not alter values that you may have poked into the memory. It's tedious, I know, but that's machine code for you!

To work, then. After switching off and on again, type the line:

```
10 N=20
```

again, and find the VLT address by using

```
?PEEK(46)*256+PEEK(45)
```

This gave me the address 2060 again. Now type RUN, so that values are put into the VLT, and take a look at what has been stored there. This is done by using the command:

```
FORX=2060 TO 2070:?X;"  ";PEEK(X):NEXT
```

and pressing RETURN. This gives the listing that is illustrated in Fig. 2.1. Now can we recognise anything here? We ought to recognise the first byte of 78, because that's the ASCII code for N!

```
2060   78
2061   0
2062   133
2063   32
2064   0
2065   0
2066   0
2067   88
2068   0
2069   140
2070   1
```

*Fig. 2.1.* The variable list table entry for a simple number variable.

The next byte is zero because our variable is called N, not N1 or NG or any other two-letter name. If we used a two-letter name, then both addresses 2060 and 2061 would have been occupied. The next five bytes, then, must be the way that the number 20 has been coded. At this point, don't worry about how these numbers are used to represent 20 just accept that they do! How do I know that it's the next five bytes that represent the number 20? Easy, the byte in address 2067 is 88, which is the ASCII for X, and that's the variable that we used to print the table values! The C 64 always used just five bytes for any value of number variable, no matter whether it's a small number like 20, or a very much larger one like 1427068315, or a

fraction, or negative. This makes the storage of number variables simple, and it also makes it easy for the computer to find variables. If, for example, it is looking for the value of a variable called Y, then when it finds 'N' (coded as ASCII 78) it need not waste time with the next six bytes (one for a second letter, five for the value), but can move to the next place where a variable name will be stored. If you are curious, and have a head for mathematics, Appendix A shows what method of coding is used to convert numbers into five bytes. For the purposes of this book you don't, however, need to understand how the coding is done as long as you know how the code is stored and how many bytes are needed.

## Tying up a string

Now we need to take a look at how a string variable is stored. Switch off and on again, and then type the line:

```
10 AB$="THIS IS A STRING"
```

RUN this one-liner, and then find the VLT address by using addresses 45 and 46 as before. I obtained 2078 for this address. Now use:

```
FOR X=2078 TO 2088:?X;"  ";PEEK(X):NEXT
```

to find what is in the VLT. This time, it's as Fig. 2.2 shows. The first value in this table is 65, which is the ASCII code for A. The second,

```
2078   65
2079   194
2080   16
2081   10
2082   8
2083   0
2084   0
2085   88
2086   0
2087   140
2088   2
```

*Fig. 2.2.* The VLT entry for a string variable.

however, is 194. Now this is the ASCII code for B with 128 added to it, and it's the way that the C 64 recognises that this is a string variable. If you had used the variable name A$ rather than AB$,

2079

then the second number (at address 2679) would have been 128, not ∅. When you use a number variable, the second ASCII code of the name will be ∅, or one of the ASCII code numbers, never greater than 127. Good thinking, designers!

Now take a look at the rest of the entry for this string. It doesn't look much like the ASCII codes for the letters, does it? In fact, the entry consists of seven bytes only, just the same total length as a number variable. The clue to what is being done emerges when we take a look at the numbers. The number that follows the code for B (194, because 128 has been added to the ASCII code) is 16. Now 16 is the number of characters in the string. If you count the number of letters and spaces you'll see that this is what it comes to. The next two bytes are 1∅ and 8. Now two bytes together are always likely to be an address, and if we combine them in the usual way, using 8*256+1∅, we get 2∅58. The next step in the trail is PEEK(2∅58). Sure enough, it's 84, which is the ASCII code for 'T'. 2∅58, then, is the address of the first byte of the string.

Let's gather all this up. The C 64 stores an entry of seven bytes in its VLT for each string. Of these seven bytes, the first two are for the string name, and the second will be 128 or more. When a two-character name is used, 128 is added to the ASCII code for the second letter. This allows the computer to distinguish a string variable from a number variable. The next five bytes then contain the length of the string and the address in memory of its first byte. As it happens, only three bytes are needed to keep track of a string. One byte is needed for the length – no string will exceed 255 characters (in fact, you are not allowed to enter more than 24∅). Two bytes are needed for the address, so that two of the seven bytes that are used in the string VLT entry are not needed except as separators. The convenience of having the same total length of VLT entry for a string as for a number outweighs the slight waste of the last two bytes in each string entry.

In this example, the string is stored at an address lower than the VLT, in the 'BASIC text' part of the memory. This is the part of the memory which contains the program, and since the ASCII codes for the string are placed here when you type the program, it's as good a resting place as any. Numbers must be transferred to the VLT because they are not stored as ASCII codes. The question now is, what happens when a string is created which does not exist in the program? Switch off, then on again, and type:

```
1∅ A$="AB":B$="CD":C$=A$+B$
```

Now RUN this, and you will find that your VLT is longer, as you might expect. You will have to look at memory addresses from 2ø8ø to 21øø this time. You will find the entries for A$ and B$, just as you would expect, giving addresses inside the program memory region, as shown in Fig. 2.3. The variable C$, however, gives the bytes

```
2080   65
2081   128
2082   2
2083   9
2084   8
2085   0
2086   0
2087   66
2088   128
2089   2
2090   17
2091   8
2092   0
2093   0
2094   67
2095   128
2096   4
2097   252
2098   159
2099   0
2100   0
```

*Fig. 2.3.* The VLT entry for a string which is not stored in the program part of memory.

252,159 for its address. This corresponds to an address of 4ø956 (it's 159*256 + 151, remember) for this string. We can take a look at these addresses. If you type:

```
FORX=4ø956TO4ø959:?X;"  ";PEEK(X);"  ";CHR$
(PEEK(X)):NEXT
```

then all will be revealed. The ASCII codes for letters ABCD are now stored here, and the use of CHR$ in the program reveals them.

## Into integers

We've looked at the storage of numbers and of strings, but we should not forget that the C 64 allows two types of numbers to be stored.

One of these number types is called 'real', the other is called 'integer'. A variable for a real number uses a letter, or pair of letters, or letter and digit like A, AB or A2 to represent it. A real number is coded in the way that we have already seen, using a total of seven bytes. Of these, two are for the name, and five for the value. A real number can be positive, negative, or fractional, and its range of size can be from about $10^{38}$ to $10^{-39}$. The integer numbers are whole numbers, no fractions allowed, and can range from −32768 to +32767. It's time to take a look at these numbers in the VLT.

Start, as usual, by switching off and then on again to clear the memory. Now type the one-liner:

```
1Ø A%=15:B%=3ØØ
```

and RUN this. Find the position of the VLT as usual by peeking addresses 45 and 46. My C 64 came up with 2Ø68. Now type:

```
FOR X=2Ø68TO2Ø85:PRINTX;"  ";PEEK(X):NEXT
```

and press RETURN. This will give you the sequence which is shown in Fig. 2.4.

```
2068   193
2069   128
2070   0
2071   15
2072   0
2073   0
2074   0
2075   194
2076   128
2077   1
2078   44
2079   0
2080   0
2081   0
2082   88
2083   0
2084   140
```

*Fig. 2.4.* The VLT entry for two integer numbers.

It's not the same as the real number sequence that we saw in Fig. 2.1. To start with, something has been done to the first byte, the one that should represent the variable name of A%. The byte is 193, which is just 65+128. It's the code for A with 128 added. The second

byte is 128. This suggests that when we create an integer variable name, the machine adds 128 to both of the letters of the name. Can you see the pattern in all this? For a real number, the letters of the name use ASCII codes. For a string, the second code for the name is 128, or an ASCII code with 128 added. For an integer, 128 has been added to the first code as well. This is how the machine can distinguish between different letter variables. The signs % and $ are *not* stored in the memory!

We need now to look at how the numbers are stored, though. The bytes that follow the two 'variable name' bytes are Ø,15,Ø,Ø,Ø, and this seems to indicate that the number 15 has been stored unaltered. This is quite unlike the transformation that we saw carried out for the storage of a real number. The number 3ØØ, however, is stored as 1,44. Now could this be a two-byte storage system? We can try it - 256*1+44=3ØØ. The number has been stored as two bytes, with the high byte first, unlike the order that is used for line numbers or memory address numbers. The VLT entry is still of seven bytes, with three of them unused this time.

SEE P.16

How does this explain the use of integers? Well, for one thing, we can explain the range of integer numbers. If we use only two bytes for storage, we can't store a number greater than the one which uses 255 stored in each byte. This number would be 255*256+255, which is 65536. Now we know that the range of integers is from −32768 to +32767 and these numbers add up to 65535! Instead of making integer numbers cover just Ø to 65535, then, the C 64 covers the more useful range of −32768 to +32767. The reasons for this range are illustrated in more detail in Appendix A. Using integers has several other side-effects. One is that integers are stored with perfect precision. When you declare A%=17412, then that's the number in the store, not 17411.9999999. A 'real' number is almost always an approximation, and when we use 'real' numbers, we have to be prepared for some 'rounding'. You've probably met calculators that told you that the answer to a problem was 3.9999999 rather than 4.Ø. This is caused by these approximations in storing real numbers, and some calculators would round the figure to 4, others would not. If you work on the C 64 using integers, these problems do not bother you. The other advantage of using integers is speed. Because an integer number is stored in two bytes, the computer can deal with it much more quickly than with a 'real' number which requires five bytes, and which needs more elaborate decoding. If you want speed, then, use integers but you probably knew that already!

C%= 32767 MAX

## Program time

It's time now to look at how a program is stored in the memory of your C 64. As before, we shall rely on PEEKs at parts of the memory to find out what is happening. The first thing we need to know, however, is where the bytes that form the address of the start of a program are stored. As it happens, they are stored at 43 and 44.

SEE P. 15

We can therefore start looking at a program as it exists in the memory. Type the program as shown in Fig. 2.5, but don't run it. Now type:

```
?PEEK(44)*256+PEEK(43)
```

and you will find the address at which the first byte of this program starts. In this example, my C 64 gave the address 2049. Now when you use the usual loop to print values of the PEEK numbers from this address onwards, you get the list as shown in Fig. 2.6. At first

```
10 A=10
20 PRINT A
30 C$="CBM 64"
```

*Fig. 2.5.* A simple BASIC program.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 10 | 8 | 10 | 0 | 65 | 178 | 49 | 48 | 0 | 18 | 8 |
| 20 | 0 | 153 | 32 | 65 | 0 | 34 | 8 | 30 | 0 | 67 |
| 36 | 178 | 34 | 67 | 77 | 66 | 32 | 54 | 52 | 34 | 0 |

*Fig. 2.6.* The bytes that represent the program in memory.

sight it looks like a stream of meaningless numbers, but when you look more carefully, you can see some pattern in it. As usual, the ASCII codes act as useful signposts. At 2053, for example, you can see the number 65, which is the ASCII code for 'A'. Since we know that the line is 'A=10', we can look for the rest of this line. The 10 is recognisable as 49 (ASCII '1') and 48 (ASCII '0'), so that the number 178 must represent the '=' sign. Now this is *not* the ASCII code for '=', but one of these 'tokens' that I mentioned in Chapter 1. It's a token because the computer is required to carry out an action, not just store an ASCII code here. The 0 at address 2057 marks the end of this line.

Now we have to grapple with the first four bytes. The first two are, as you might suspect from looking at them, an address. The 10,8

makes up the address 8*256+1ø, which is 2ø58. What is this address? Why, it's the address of the first byte of the next line! This is how the operating system of the C 64 can pick out lines, and put them into the correct sequence, no matter what order you use to enter them. The final mystery is easily solved. Looking at the third and fourth bytes of each of the lines shows the sequence 1ø, 2ø, 3ø – the line numbers. There are two bytes reserved for the line numbers because we want to have line numbers higher than 255. For line numbers smaller than 256, the second of these bytes – the more significant byte – is not used.

Now take a look at the other lines, as they appear stored in the memory. We have met the PRINT token of 153 before, and all the rest should be familiar by now. The only novelty is the end of the program. The last line ends with a ø as usual, but following it, in the place where the address bytes for the next line would be, is another pair of zeros. This is the marker that the computer uses for END.

We can carry out some interesting changes on a program like this. Suppose, for example, that we poke the addresses that are used to carry the line numbers. If you type:

```
POKE2ø6ø,1ø:POKE2ø68,1ø
```

and press RETURN, you will have placed the number 1ø in each line number address for the lines 2ø and 3ø. Now LIST and look at the result! It's a program of line 1ø's. Contrary to what you might expect, this will RUN perfectly normally. The action of running, you see, depends on the 'next line' addresses being correct, not on how the lines are numbered. A program that has been altered in this way, however, is certainly not normal. Try, for example, using the screen editing system to change the second line A to B, so that the line reads PRINT B. Now RUN and then LIST. You will find that the previous first line has disappeared, and the new first line is PRINT A, with PRINT B as the second line! You can, however, record a program which has been altered in this way, and replay it normally. This is the germ of a method by which you can make a program difficult to alter.

## Running the program

Now that we have looked at the way in which a program is coded and stored in the memory of the C 64, we can give a bit of thought as to how it runs. This action is carried out by the most complicated part

of the operating system, and it has to be given a starting address. This address comes, as you might expect, from the locations 43 and 44 which we have used. Suppose we go through the actions, omitting detail, of the three line program of Fig. 2.5. At the first address in BASIC, the RUN subroutine will read the first two bytes, and store them temporarily. These bytes will be used in place of the 'start of BASIC' address when the next line is carried out. The line number bytes are then read and stored. Why? So that if there is a syntax error in the line, the computer will be able to print out the message: 'SN ERROR IN 1Ø' rather than 'SN ERROR SOMEWHERE'! The next byte is an ASCII code, and the computer will take this as being a variable name. In the old days, the word LET had to be used to 'declare a variable'. There was a token for LET and, in modern machines, this token is still used. The difference is that the token is now put in automatically when a letter immediately follows a line number or a colon. If you do type LET, then the same token is used.

Following the assignment token, the special token for the '=' sign then causes a subroutine to swing into action. This one creates an entry in the variable list table, at the first available address, and puts the ASCII code for A in that place. The next address in the VLT is left blank – there's no second letter for this variable name. The number 1Ø is then read and converted to the special binary form, as noted in Appendix A. This set of bytes is also placed in the VLT as the entry for A. The next byte of the program is then read – it's Ø, so that the address for the next line, which was read as the first action, is now placed into the microprocessor. The type of action that we have considered in detail for line 1Ø then repeats with line 2Ø. This time, more has to be done when the action token is read. Since this is the token for PRINT, the subroutine for PRINT must be called up. It will locate the address of the next vacant place on the screen. This is done by keeping a note of the address in a couple of bytes of RAM – read these bytes, and you have the address. The value of A is then found in the VLT, and the bytes converted back to ASCII code form. The codes are then placed, one by one, in the screen memory. Doing this causes the characters to become visible on the screen, because of another subroutine. Once again, the zero at the end of the line causes the next line number to be used. At the end of the third line, however, the 'next line' number is zero, and the program ends. The computer goes back to its waiting state, ready for another command.

It's not quite so simple as that description makes it sound, but the essentials are there. The important thing to realise is that there is a

lot of action to be done, and it has to be done one step at a time. What makes BASIC slow is that each token calls up a subroutine, which has to be found. For example, if you have a program that consists of a loop like:

```
1Ø FOR N=1TO5Ø
2Ø PRINT N
3Ø NEXT
```

then the action of reading the PRINT token of 153, and finding where the correct subroutine is stored, will be carried out fifty times. There is no simple way of ensuring that the subroutine is located once and then just used fifty times. The kind of BASIC that you have on your C 64 is *interpreted* BASIC. This means that each instruction is worked out as the computer comes to it. If that means finding the address of the PRINT subroutine fifty times, so be it. The alternative is a scheme called *compiling*, in which the whole program is converted to efficient machine code before it is run. Compiling is done using another program, called a *compiler*. The use of a compiler for a BASIC program very greatly speeds up the running of the program, but it makes the program less easy to edit, because it converts the program into a different form. You can't win them all!

See P. 29

# Chapter Three
# The Microprocessor

In this chapter, we'll start to get to grips with the 6502 microprocessor of the C 64. The microprocessor or MPU is, you remember, the 'doing' part of the computer as distinct from the storing part (memory) or the input/output part (ports), so that what the microprocessor does will control what the rest of the computer does.

The MPU itself consists of a set of memory stores for numbers, but with a lot of organisation added. By means of circuits that are very aptly called *gates*, the way in which bytes are transferred between different parts of the MPU's own memory can be controlled, and it is these actions that constitute the addition, subtraction, logic and other actions of the MPU. Each of the actions is programmed. Nothing will happen unless an instruction byte is present in the form of a 1 or a Ø signal at each of the 8 data terminals, and these bytes are used to control the gates inside the MPU. What makes the whole system so useful is that because the program instructions are in the form of electrical signals on eight lines, these signals can be changed very rapidly. The speed is decided by another electrical circuit called a 'clock-pulse generator', or 'clock' for short. The speed that has been chosen as standard for the clock of the C 64 is very fast, so that something like a million operations can be carried out per second.

## Machine code

The program for the MPU, as we have seen, consists of number codes, each being a number between Ø and 255 (a single byte number). Some of these numbers may be instruction bytes which cause the MPU to do something. Others may be data bytes, which are numbers to add, or store or shift, or which may be ASCII codes

for letters. The MPU can't tell which is which – it simply does as it is instructed. It's up to the programmer to sort out the numbers and put them into the correct order.

The correct order, as far as the MPU is concerned, is quite simple. The first byte that is fed to the MPU after switching on or after completing an instruction, is taken as being an instruction byte. Now many of the 6502 instructions consist of just one byte, and need no data. Others may be followed by one or two bytes of data, and some instructions need two bytes. When the MPU reads an instruction byte, then it analyses the instruction to find if the instruction is one that has to be followed by one or more other bytes. If, for example, the instruction byte is one that has to be followed by two data bytes, then when the MPU analyses the first byte, it will treat the next two bytes that are fed to it as being the data bytes for that instruction. This action of the MPU is completely automatic, and is built into the MPU. The snag is that the machine code programmer must work to the same rules, and get the program right. 100% correct is just about good enough! If you feed a microprocessor with an instruction byte when it expects a data byte, or with a data byte when it expects an instruction byte, then you'll have trouble. Trouble nearly always means an endless loop, which causes the screen to go blank and the keys to have no effect. Even the combination of STOP and RESTORE keys can sometimes fail to break the C 64 out of such a loop, and the only remedy is to switch off. You will generally lose whatever program you had in store, so that it's vitally important to save any machine code program, or a BASIC program that causes machine code actions (by using POKE) on tape before you use it.

What I want to stress at this point is that machine code programming is tedious. It isn't necessarily difficult – you are drawing up a set of simple instructions for a simple machine – but it's often difficult for you to remember how much detail is needed. When you program in BASIC, the machine's error messages will keep you right, and help to detect mistakes. When you use machine code, you're on your own, and you have to sort out your own mistakes. In this respect, a type of program called an *assembler* helps considerably. We'll look at that point again later. In the meantime, the best way to learn about machine code is to write it, use it, and make your own mistakes. We'll start looking at how this is done, and we'll begin with the ways of writing the numbers that constitute the bytes of a machine code program.

SEE P.27

## Binary, denary and hex

A machine code program consists of a set of number codes. Since each number-code is a way of representing the 1's and Ø's in a byte, it will consist of numbers between Ø and 255 when we write it in our normal scale of ten (denary scale). The program is useless until it is fed into the memory of the C 64, because the MPU is a fast device, and the only way of feeding it with bytes as fast as it can use them is by storing the bytes in the memory, and letting the MPU help itself to them in order. You can't possibly type numbers fast enough to satisfy the MPU, and even methods like tape or disk are just not fast enough.

Getting bytes into the memory, then, is an essential part of making a machine code program work, and we shall look at methods in more detail later on. At one time, simple and very short programs would be put into a memory by the most primitive possible method, using eight switches. Each switch could be set to give a 1 or Ø electrical output, and a button could be pressed to cause the memory to store the number that the switches represented, and then select the next memory address. Programming like this is just too tedious, though, and working with binary numbers of 1's and Ø's soon makes you cross-eyed. Now that we have computers, it makes sense to use the computer itself to put numbers into memory, and an equally obvious step is to use a more convenient number scale.

Just what is a convenient number scale is a matter that depends on how you enter the numbers and how much machine code programming you do. The C 64 contains subroutines which convert the binary numbers in its memory to the form of denary numbers to print on the screen, and will also carry out the reverse action. When you use PEEK, the address that you want can be written in denary, and the result of the PEEK will be a number in denary, between Ø and 255. When you use POKE, similarly, you can type both the address number and the byte to be poked in denary.

Serious machine code programmers, however, find the use of denary anything but convenient. A denary number for a byte may be one figure (like 4) or two (like 17) or three (like 143). A much more convenient code is the one called *hex* (short for hexadecimal) code. All one-byte numbers can be represented by just two hex digits. In conjunction with this, serious machine code programmers write their programs in what is called *assembly language*. This uses command words which are shortened versions of the names of commands to the MPU. Programs that are called

*assemblers* then convert these command words into the correct binary codes. Practically all assemblers show these codes on the screen in hex form rather than in denary. In addition, when you type data numbers, you will have to make use of hex code. 'Hexadecimal' means scale of sixteen, and the reason that it is used so extensively is that it is naturally suited to representing binary bytes. Four bits, half of a byte, will represent numbers which lie in the range Ø to 15 in our ordinary number scale. This is the range of one hex digit (see Fig. 3.1). Since we don't have symbols for digits higher than 9, we have to

| Hex | Denary | Hex | Denary |
|---|---|---|---|
| Ø | Ø | C | 12 |
| 1 | 1 | D | 13 |
| 2 | 2 | E | 14 |
| 3 | 3 | F | 15 |
| 4 | 4 | then | |
| 5 | 5 | 1Ø | 16 |
| 6 | 6 | 11 | 17 |
| 7 | 7 | to | |
| 8 | 8 | 2Ø | 32 |
| 9 | 9 | 21 | 33 |
| A | 1Ø | 22 | 34 |
| B | 11 | etc. | |

*Fig. 3.1.* Hex and denary digits.

use the letters A,B,C,D,E, and F to supplement the digits Ø to 9 in the hex scale. The advantage is that a byte can be represented by a two-digit number, and a complete address by a four-digit number. The number codes that are used as instructions have been designed in hex code, so that we can see much better how commands are related. For example, we may find that a set of related commands all start with the same digit when they are written in hex. In denary, this relationship would not appear. In addition, it's much easier to write down the binary number which the computer actually uses when you see the hex version. The use of the C 64 assembler and monitor programs, such as the excellent MIKRO assembler, demand familiarity with hex, and books of information on the 6502 MPU will all be written assuming that you know hex. It sounds as if we ought to make a start on it!

## The hex scale

The hexadecimal scale consists of sixteen digits, starting as usual with Ø and going up in the usual way to 9. The next figure is not 1Ø, however, because this would mean one sixteen and no units. Since we aren't provided with symbols for digits beyond 9, we use the letters A to F. The number that we write as 1Ø (ten) in denary is written as ØA in hex, eleven as ØB, twelve as ØC and so on up to fifteen, which is ØF. The zero doesn't have to be written, but programmers get into the habit of writing a data byte with two digits and an address with four even if fewer digits are needed. The number that follows ØF is 1Ø, sixteen in denary, and the scale then repeats to 1F, thirty-one, which is followed by 2Ø. The maximum size of byte, 255 in denary, is FF in hex. When we write hex numbers, it's usual to mark them in some way so that you don't confuse them with denary numbers. There's not much chance of confusing a number like 3E with a denary number, but a number like 26 might be hex or denary. The convention that is followed by 6502 programmers is to use the dollar sign ($) to mark a hex number, with the sign placed before the number. For example, the number $47 means hex 47, but plain 47 would mean denary forty-seven. When you write hex numbers for a 6502 program, it's advisable to follow this convention.

| Hex | Binary | Hex | Binary |
|---|---|---|---|
| Ø | ØØØØ | 8 | 1ØØØ |
| 1 | ØØØ1 | 9 | 1ØØ1 |
| 2 | ØØ1Ø | A | 1Ø1Ø |
| 3 | ØØ11 | B | 1Ø11 |
| 4 | Ø1ØØ | C | 11ØØ |
| 5 | Ø1Ø1 | D | 11Ø1 |
| 6 | Ø11Ø | E | 111Ø |
| 7 | Ø111 | F | 1111 |

*Fig. 3.2.* Hex and binary digits.

Now the great value of hex code is how closely it corresponds to binary code. If you look at the hex-binary table of Fig. 3.2, you can see that $9 is 1ØØ1 in binary and $F is 1111. The hex number $9F is therefore just 1ØØ11111 in binary – you simply write down the binary digits that correspond to the hex digits. The conversion in the opposite direction is just as easy – group the binary digits in fours, starting at the least significant (right-hand side of the number) and then convert each group into its corresponding hex digit. Figure 3.3

```
            Conversion: Hex to Binary

Example: 2CH ..............  2H is ØØ1Ø binary
                            CH is 11ØØ binary

So 2CH is ØØ1Ø11ØØ binary          (data byte)


Example: 4A7FH ............ 4H is Ø1ØØ binary
                            AH is 1Ø1Ø binary
                            7H is Ø111 binary
                            FH is 1111 binary

So 4A7FH is Ø1ØØ1Ø1ØØ1111111 binary (an address)


            Conversion: Binary to Hex

Example: Ø11Ø1Ø11 ..............  Ø11Ø is 6H
                                  1Ø11 is BH

So Ø11Ø1Ø11 is 6BH


Example: 1Ø11Ø1ØØ1ØØ1Ø ... note that this is
not a complete number of bytes.

   Group into fours, starting with lsb:
               ØØ1Ø is 2H
               1ØØ1 is 9H
               11Ø1 is DH and
    the remaining 1Ø is 2, making 2D92H
```

*Fig. 3.3.* Converting between hex and binary.

shows examples of the conversion in each direction so that you can see how easy it is.

The C 64 has no built-in programs for converting between denary and hex, and the most convenient method of converting is a BASIC program. Figure 3.4 shows a denary-to-hex conversion program which builds up a string called H$ as a hex number. The conversion is started by dividing the number by 4Ø96 and taking the integer (whole number) part of the result. If the denary number was smaller than 4Ø96, then the result of this is Ø, and this will also be the

first hex digit. The conversion of hex codes is done by making use of the ASCII code values, because there is a simple relationship between the ASCII codes for digits ∅ to 9 and the digits themselves.

```
10 PRINT"□":PRINT"PLEASE TYPE DENARY NUMBER-"
   :INPUT D
20 IF D>65535THENPRINT"TOO LARGE-LIMIT IS 65535"
   :GOSUB300:GOTO10
30 IF D<1THENPRINT"NO NUMBERS LESS THAN 1, PLEASE"
   :GOSUB300:GOTO10
40 IF D<>INT(D)THENPRINT"NO FRACTIONS, PLEASE"
   :GOSUB300:GOTO10
50 F=4096:H$=""
60 Y=INT(D/F)
70 GOSUB200
80 D=D-Y*F:F=INT(F/16)
90 IF F<1THEN110
100 GOTO60
110 IF LEFT$(H$,2)="00"THEN H$=RIGHT$(H$,2)
120 H$="$"+H$
130 PRINT"HEX NUMBER IS ";H$
140 END
200 IF Y<=9THEN H$=H$+CHR$(Y+48)
210 IFY>9THEN H$=H$+CHR$(Y+55)
220 RETURN
300 FORN=1TO1000:NEXT:RETURN
```

*Fig. 3.4.* A program for converting denary numbers to hex.

If you add 48 to the digit, then you have its ASCII code, so that 48+1=49 is the code for '1'. This ASCII code can then be incorporated into a string. A slight complication arises when we get to ten, because in hex this is A, and the ASCII code for A is 65, which is 55 greater than 1∅. The conversion program must therefore add 48 to a digit of 9 or less, and add 55 to a digit in the range 1∅ to 15 so as to make the correct conversion. This isn't difficult to do in a BASIC program. The next step is to take the remainder after dividing by 4∅96, and divide this by 256, which is 4∅96/16. The integer part of this is then put into hex string form, and the process repeats until there is nothing left. If you want to make this routine super-efficient, then use integer variables for all of the whole numbers.

Hex-to-denary can be done in much the same way, and a suitable program is shown in Fig. 3.5. This relies on converting the ASCII code for each digit of the hex number into its number digit, and then multiplying by the correct place factor (numbers 1,16,256 and 4∅96

```
10 PRINT"□":Y=1:D=0:PRINT"PLEASE TYPE HEX NUMBER"
   :INPUTH$
30 L=LEN(H$):FORN=0TOL-1
40 GOSUB200:NEXT
50 PRINT"DENARY NUMBER IS ";D
100 END
200 P$=MID$(H$,L-N,1):A=ASC(P$)
210 IFA<48 OR A>102 THEN GOSUB300:GOTO280
220 IF A<65 AND A>57THEN GOSUB300:GOTO280
230 IF A<=97 AND A>70 THEN GOSUB300:GOTO280
240 IF A<=57 THEN Q=A-48
250 IF A>=65THEN Q=A-55
260 IF A>=97THEN Q=A-87
270 D=D+Q*Y:Y=Y*16
280 RETURN
300 PRINT"BAD HEX...PLEASE TRY AGAIN":END
500 FORX=1TO1000:NEXT:RETURN
```

*Fig. 3.5.* A program for converting hex numbers to denary.

for up to 4 digits) The numbers obtained in this way are then added into a total, D. Once again, the BASIC program is fairly simple, which is why the 'we pay for each program published' page of every magazine gets choked with denary-hex conversion programs for each new computer! Just in case you want to do these conversions when you are not near a friendly C 64, though, the methods are shown in Appendix B.

Assuming, which is reasonable, that you don't want to commit yourself to the cost of a full-scale assembler at this point, what do you do to create machine code programs? The answer is that you design your program in assembly language, which is by far the easiest way to design machine code programs, and then you convert into hex code. Converting means looking up in a set of tables (called the *instruction set*) the hex number that represents each instruction. Instruction sets are provided by the manufacturers of all microprocessors, and Mostek, who designed the 6502, provide one for this chip. Just to assist you, a quick-reference guide has been included in this book, in Appendix C. Don't refer to it at the moment - it'll put you off!

## Negative numbers

Useful as these denary-hex conversion programs for the C 64 are,

they are deficient in one respect - they don't handle negative numbers. This is unfortunate, though understandable. Negative numbers are very important in machine code programs, particularly if you are working without an assembler. The reason is that you sometimes want the MPU to do the equivalent of a GOTO, perhaps jumping to a step which is 3Ø steps ahead of its present address. This sort of thing is usually programmed by supplying a data number which is the number of steps that you want to skip. If you want to jump back to a previous step, however, you will need to use a negative number for this data byte. This is very common, because it's the way that a loop is programmed in machine code. We need,

```
Binary: Number (8 bits)          ØØ11Ø1Ø1 (denary 53)

Step 1: change each Ø to
        1, each 1 to Ø           11ØØ1Ø1Ø

Step 2: add 1                          +1
                                 --------
Result is negative
form of number                   11ØØ1Ø11 (denary 2Ø3)
                                 --------

Denary: Number                         53

Step 1: subtract from 256             2Ø3

This is negative form (in denary)

Hex: Number                           $35   \ Remember that F
                                            | represents 15
Step 1: subtract from $FF              FF   | denary, and
                                       35   | 15 - 5 = 1Ø
                                      ----  | denary, which
                                      $CA   / is A in hex.

Step 2: add 1                          +1   \ This is easier
                                      ----  | than a subtraction
                                      $CB   | from $1ØØ, which
                                            | involves the use
                                            / of carries in hex.

Result is hex form of negative number

An alternative is to do the denary conversion, then
convert back to hex.
```

*Fig. 3.6.* The two's complement, or negative form, of a binary number.

therefore, to know how to write a negative number in hex.

What makes it awkward is that there is no negative sign in hex arithmetic. There isn't one in binary either. The conversion of a number to its negative form is done by a method called *complementing*, and Fig. 3.6 shows how this is done. At first sight, and very often at second, third, and fourth, it looks entirely crazy. When you are dealing with a single byte number, for example, the denary form of the number −1 is 255! You are using a large positive number to represent a small negative one! It begins to make more sense when you look at the numbers written in binary. The numbers that can be regarded as negative all start with a 1 and the positive numbers all start with a ∅. The MPU can find out which is which just by testing the left-hand bit, the most significant bit.

It's a simple method, which the machine can use efficiently, but it does have disadvantages for humans. One of these disadvantages is that the digits of a negative number are not the same as those of a positive number. For example, in denary −4∅ uses the same digits as +4∅. In hex, −4∅ becomes $D8 and +4∅ becomes $28. The denary number −85 becomes $AB and +85 becomes $55. The second disadvantage is that humans cannot distinguish between a single byte number which is negative and one which is greater than 127. For example, does $9F mean 159 or does it mean −97? The short answer is that the human operator doesn't have to worry. The microprocessor will use the number correctly no matter how we happen to think of it. The snag is that we have to know what this correct use is in each case. Throughout this book, and in others that deal with machine code programming, you will see the words 'signed' and 'unsigned' used. A signed number is one that may be negative or positive. For a single byte number, values of ∅ to $7F are positive, and values of $8∅ to $FF are negative. This corresponds to denary numbers ∅ to 127 for positive values and 128 to 255 for negative. Unsigned numbers are always taken as positive. If you find the number $9C described as signed, then you know it's treated as a negative number (it's more than $8∅). If it's described as unsigned, then it's positive, and its value is obtained simply by converting. How do we convert a signed single-byte hex number into denary, using our programs? It's simple – if the number is greater than $7F, then subtract 256 from its denary value. If you get 24∅ as a result, for example, then 24∅−256 =−16, and that's the signed value in denary.

## Light relief

Just to take a break from all this arithmetic, let's look at screen displays on the C 64. Each part of the screen can be controlled by whatever is stored in part of the memory, but there are two varieties of this memory. The simplest one to work with is the part which is called *text screen memory*. It takes up memory addresses $4ØØ to $7E7, which is 1Ø24 to 2Ø23 in denary. What we mean by 'text screen memory' is that this piece of memory is treated in a special way. Anything that is stored here will be used to display text on the screen. That means that any code number that you store at an address in this range will produce the corresponding letter or graphics shape on the screen. It's not quite so simple as this with the C 64, though. When a number is placed in any part of this memory, its effect is not visible unless one of two conditions is met. Condition one is that there should be a character at that screen position already. Condition two is that a new background colour must be used. Readers of my basic guide to the C 64, *Commodore 64 Computing*, will already be aware of these restrictions. Try this – press the CLEAR key to clear the screen, and then type:

```
POKE53281,3:POKE1524,1
```

and press RETURN. The result is the letter A appearing in the middle of the screen. The computer has converted its 'internal' code of 1 that was stored at position 1524 into a set of numbers that will produce the letter 'A' on the screen at this position. You can program this sort of thing in a loop, as Fig. 3.7 shows. You should restore normal screen conditions by using STOP/RESTORE first.

```
10 POKE53281,3
20 FORN=1024TO2023
30 POKEN,1:NEXT
```

*Fig. 3.7.* A program which fills the screen with the letter A.

The effect of this loop is to fill the screen with A's. The filling is not particularly fast, because we're using BASIC in the loop. Later, we'll look at the same sort of thing in machine code, which is stunningly fast! For the moment, though, look at the effect on this program of replacing 1 by another number which is the code for one of the graphics blocks, like 65. It's quite useful, and if you want to avoid the 'hole' in the pattern caused by the OK message and the cursor, then add an endless loop to the program, like:

```
40 GOTO 40
```

The odd thing here is the use of 'internal' codes. The C 64 uses the standard set of ASCII code numbers in its BASIC. When you use BASIC instruction words like ASC and CHR$, you are making use of ASCII code numbers. For its own purposes, however, the C 64 converts these ASCII codes to other numbers. These are the 'internal' code numbers, and they are listed in Appendix E of the C 64 manual. When we poke a number into the part of memory that is used for the screen, the character that will appear will be one selected from this 'internal' list, rather than what you might expect from the ASCII codes.

# Chapter Four
# 6502 Details

## Registers - PC and accumulator

A microprocessor consists of sets of memories, which are called *registers*. These memories are of a rather different type compared to ROM or RAM. The registers are connected to each other and to the pins on the body of the MPU by the circuits that are called *gates*. In this chapter, we shall look at some of the most important registers of the 6502 and how they are used. A good starting point is the register which is called the *PC* - short for Program Counter.

No, it doesn't count programs - what it does is to count the steps in a program. The PC is a sixteen-bit (two byte) register which can store a full-sized address number, up to $FFFF (65535 denary). Its purpose is to count the address number, and the number that is stored in the PC will be incremented (increased by 1) each time an instruction is completed, or when another byte is needed. For example, if the PC holds the address $1F3A (denary 7994), and this address contains an instruction byte, then the PC will increment to $1F3B (denary 7995) whenever the MPU is ready for another byte. The next byte will then be read from this new address.

What makes the PC so important is that it's the automatic way by which the memory is used. When the PC contains an address number, the electrical signals that correspond to the Ø's and 1's of that address appear on a set of connections, collectively called the *address bus*, which link the MPU to all of the memory, RAM and ROM. The number that is stored in the PC will select one byte from the memory, the byte which is stored at that address number. At the start of a read operation, the MPU will send out a signal called the read signal on another line, and this will cause the memory to connect up the selected parts to another set of lines, the data bus. The signals on the data bus then correspond to the pattern of Ø's and 1's that is stored in the byte of memory that has been selected by the

address in the PC. Each time the number in the PC changes, another byte of memory is selected, so that this is the way by which the MPU can keep itself fed with bytes. When the MPU is ready for another byte, the PC increments, and another read signal is sent out.

There are other ways in which the PC number can be changed, but for the moment we'll pass over that and look at another register, the *accumulator*. The accumulator of a microprocessor is the main 'doing' register of the MPU. This means that you would normally use it to store any number that you wanted to transfer somewhere else, or add to or carry out any other operation upon. The name 'accumulator' comes from the way in which this register operates. If you have a number stored in the accumulator, and you add another number to it, then the result is also stored in the accumulator. The nearest equivalent in BASIC is using a variable A, and writing the line:

```
A=A+N
```

where N is a number variable. The result of this BASIC line is to add N to the old value of A, and make A equal this new value. The old value of A is then lost. The accumulator acts in the same way, with the difference that an accumulator can't store a number greater than 255 (denary).

The 6502 has one accumulator register, often labelled as the 'A register'. The importance of this is that it is used much more than the other registers, because so many actions can be carried out more quickly, more conveniently, or perhaps only, in the accumulator. When we read a byte from the memory, we usually place it in the accumulator. When we carry out any arithmetic or logic action, it will normally be done in the accumulator and the result will also be stored in the accumulator.

## Addressing methods

When we program in BASIC, we don't have to worry about memory addresses at all unless we are using PEEK or POKE. The task of finding where bytes are stored is dealt with by the operating system of the machine. When a variable is allocated a value in a BASIC program as, for example, by a line like:

```
1Ø N = 12
```

we never have to worry about where the number 12 is stored, or in what form. Similarly, when we add the line:

```
2Ø K=N
```

we don't have to worry about where the value of N was stored or where we will store the value of K. Remembering our comparison with wall-building, we can expect that when we carry out machine code programming, we shall have to specify each number that we use, or alternatively the address at which the number is stored. This way in which we obtain a number, or find a place to store it, is called the *addressing method.* What makes the choice of addressing method particularly important is that a different code number is needed for each different addressing method for each command. This means that each command exists in several different versions, with a different code for each addressing method. A list of all the 6502 addressing methods at this stage would be rather baffling, and for that reason has been consigned to Appendix D. What we shall do here is to look at some examples of selected addressing methods and the way that we write them in assembly language.

## Assembly language

Trying to write down machine code directly as a set of numbers is a very difficult process which is liable to errors from beginning to end. The most useful way of starting to write a program is to write it in a set of steps in what is called *assembly language* (or *assembler language*). This is a set of abbreviated command words, called 'mnemonics', and numbers which are the data or address numbers. The numbers can be in hex or in denary, provided they are supplied to the computer in the correct form. Each line of an assembly language program indicates one microprocessor action, and this set of instructions is later 'assembled' into machine code, hence the name.

The aim of each line of an assembly language program is to show the action and the data or address that is needed to carry out that action. Just as when we make use of TAB in BASIC we need to complete the command with a number. The part of the assembly language that specifies what is to be done is called the *operator*, and the part which specifies what the action is done to or on is called the *operand.* A few instructions need no operand, and we'll look at some later.

An example makes this easier. Suppose we look at the assembly language line:

```
LDA #$12
```

The operator is LDA, a shortened version of LOAD A, meaning that the accumulator register A is to be loaded with a byte. The operand is #$12, of which the $12 means that this is 12 hexadecimal, rather than twelve denary. The other mark, the hashmark, #, is used to show the addressing method that is to be used, a method called 'immediate addressing'.

The whole line, then, should have the effect of placing the number $12 into the accumulator register A. It is the equivalent in machine code terms of the BASIC instruction:

```
A=18   (remember that $12 is denary 18)
```

You could imagine that the memory which held the number was inside the microprocessor rather than part of the RAM memory, and was labelled with the name of 'A'.

A command such as LDA #$12 is said to use immediate addressing, because the byte which is loaded into the accumulator must be placed in the memory byte whose address immediately follows that of the instruction byte. It's like leaving a note for your milkman that says 'money in envelope next door'. There is one code number for the LDA # part of the whole instruction, and this byte is $A9, so that the hex sequence in memory of A9 12 will represent the entire command LDA #$12. It's a lot easier to remember what LDA #$12 means than to interpret A9 12, however, which is why we use assembly language as much as possible.

Immediate addressing like this can be convenient, but it ties you down to the use of one definite number and one fixed memory address. It's rather like programming in BASIC:

```
N = 4*12 + 3
```

rather than

```
N= A*B+C
```

In the first example, N can never be anything else but 51, and we might just as well have written: N = 51. The second example is very much more flexible, and the value of N depends on what values we choose for the variables A, B and C. When a machine code program is held in RAM, then the numbers which are loaded by this immediate addressing method can be changed if we must change

them. However, when the program is held in ROM no change is possible - and that's just one reason for needing other addressing methods. One of these other methods is 'absolute addressing'.

Absolute addressing uses a complete two-byte address as its operand. This creates a lot of work for the 6502. This is because, when it has read the code for the operator, it will then have to read two more bytes to find the memory address at which the data is stored. It will then have to place this address in the PC, read in the data byte, carry out the operation, and then restore the next correct address into the PC. Figure 4.1 shows in diagram form what has to be done. An absolute-addressed operation is therefore a lot slower to carry out than an immediate one, but since any byte may be stored at the address which is specified, it's easy to alter the data.

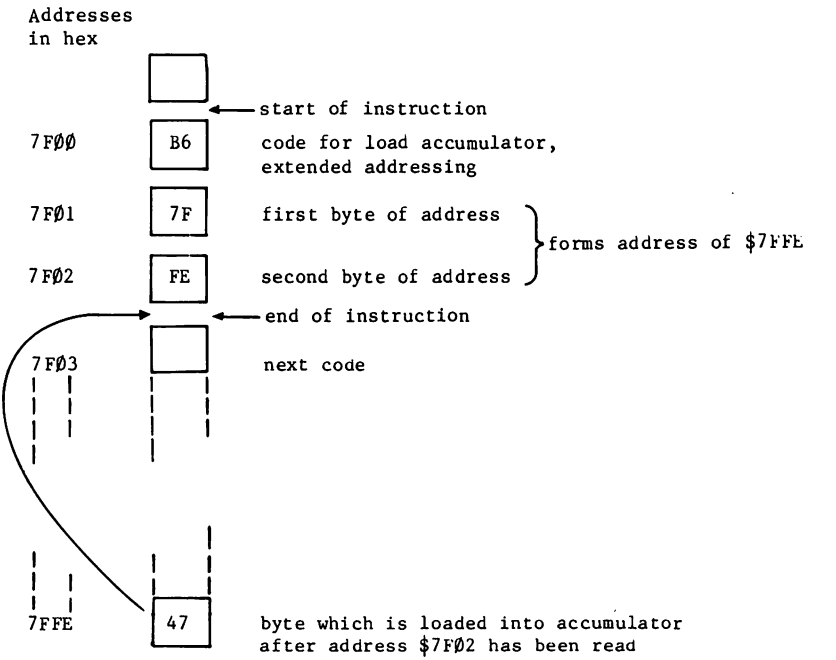



*Fig. 4.1.* How the absolute addressing method works.

Suppose, for example, that we have the instruction:

LDA $7FFE

In this slice of assembly language, the operator is LDA (load the accumulator A), and the operand is the address $7FFE. What you have to remember is that what is put into the register A is not 7FFE, which is a two-byte address, but the data byte which is stored in

memory at this address. The effect of the complete instruction, then, is to place a copy of the byte which is stored at $7FFE into the accumulator A of the 6502. When the instruction has been completed, the address $7FFE will still hold its own copy of the byte, because reading a memory does not change the content of the memory in any way.

We can also use the absolute addressing method in a command which will store a byte into the memory. The command:

```
STA $7FFF
```

means that the byte that is stored in the accumulator A is to be copied to memory at address $7FFF. This action does change the content of this memory address, but the accumulator A will still hold the same byte after the instruction has been carried out.

### Zero page addressing

Zero page addressing is a method that allows you to specify a full address by using only one byte! The secret is that the upper byte of the address is taken to be zero. This is how we get the name of 'zero page' for this type of addressing. Suppose, for example, that we used the command:

```
LDA $3F
```

There's no # sign here, so it does not mean that we load the number $3F into the accumulator. What it does mean is that we load the accumulator with the byte that has been stored in the address $ØØ3F. The ØØ part is the zero page, and the 3F part is the part that we have specified in the instruction. The range of addresses that we can use with this method is from $ØØØØ to $ØØFF only. This is only 256 (denary) bytes, but it's a very important 256 bytes. In the 6502, zero page addressing allows us to get access to any of these addresses very rapidly, and with only one byte of address (the lower byte). This makes these addresses the favoured ones for any programmer who wants to carry out rapid loading and storing. For this reason, the 'zero page' addresses of the RAM in any 6502 machine are favourites for storing important quantities. We have already seen them being used for storing quantities like the address of the start of BASIC and the address of the Variable List Table. The same addresses are used for all sorts of important quantities, and as we go on in this book, we shall look at several of them in more detail. Because the C64 makes a

lot of use of zero page addresses for its own purposes, we have to be careful how we make use of them in our own programs. If we try to use an address that the computer needs, we may lock up the whole system. That's just one reason why it's so important to record a machine code program before you try it out.

## Indexed addressing

Indexed addressing is a method which is particularly useful on the 6502. The principle is that an eight-bit register is used to hold a byte. This byte is then added to a base address when the indexed addressing is used. A base address means an address to which we can add a number before using it. For example, suppose that we had the number $4C stored in an index register. If we then specify that we want to load from an address $7ØØØ with indexing, then what happens is that the number in the index register is added to the address $7ØØØ. This makes the address number $7Ø4C, and this is the number that will be used as the address number. The effect is that the byte in address $7Ø4C will be loaded into the accumulator.

There are two registers in the 6502 that can be used in this way, the X and Y registers. They are both eight-bit registers, and they are almost identical, but there are some differences that will be more important to you later on. One of the differences that we can look at immediately is zero page indexed addressing. When we load the X register with a byte such as $4A, and then issue the command (in assembly language):

```
LDA $7ØØØ,X
```

this means that the address that is to be used is $7ØØØ plus the contents of the X register. The result is that the address $7Ø4A is used. This is 'absolute indexed addressing'. 'Absolute' means that we are using absolute addressing for the 'base address' of $7ØØØ, and we are then adding the 'index' number of 4A from the X index register. We can use the Y index register of the 6502 in exactly the same way, with instructions such as:

```
LDA $7ØØØ,Y
```

in assembly language. The use of the X index register, however, allows us to use zero page addressing. We can, for example, use assembly language commands such as:

```
LDA ØC,X
```

This means that the base address is $ØØØC, and that the number stored in the X register will be added to $ØØØC before use. If the number stored in the X register is $23, then $ØØØC + $23 gives $ØØ2F, and this is the address number that will be used. The accumulator will therefore be loaded from address $ØØ2F. Zero page indexed addressing cannot be used with the Y index register, so that a command such as:

```
LDA $1F,Y
```

is impossible – there is no instruction code for it.

Now the use of indexed addressing might not seem particularly useful at first sight. All you are doing, after all, is to add a number to an address. What makes the method so very useful is that you can alter the number that is stored in the X or Y registers. More particularly, you can increment or decrement the number in either of these index registers. The command INX means 'increment X'. Its effect is to add one to the number that is stored in the X register. Now since the number in the X register is added to a 'base address' when we use indexed addressing, incrementing X means that we shall increment the address that we get from an X indexed load or store operation.

Suppose, for example, that you wanted to store ten bytes at ten consecutive addresses. It's a very common problem, because it's just what you need to do to print ten characters on the screen, for example. The use of indexing means that you can set up a loop which will store, using indexing, then increment the index and repeat the store operation. Carry this out for a total of ten times, and the action is complete. Perhaps it's a bit early to mention the use of a loop, but we'll soon come to it. The example is a useful one, because it illustrates one of the most common uses for indexed addressing.

In addition to incrementing the X and Y registers (mnemonics INX, INY), we can decrement these registers. The assembly language command DEX means decrement X (subtract 1 from the number stored in X), and DEY means decrement Y. Since we have two index registers it's even possible to load from one base address, X-indexed, and then increment X. The byte that has been loaded can then be stored, indexed to the Y register this time and using another base address. The Y register can then be decremented. If all this is done in a loop, the effect will be to place bytes from a set of addresses, moving up from a starting address, into another set of addresses, moving down from another starting address. Complicated, you think? It might sound so in words, but in fact it's a very simple

and neat method of shifting bytes from one part of memory to another, and that's an operation which is very often essential in computing.

## Indirect addressing

'Indirect addressing ' means going to an address to pick up another address at which a byte is located. It's like going to the address of a tourist office to find the address of a hotel (for a quick byte?). The 6502 allows two main forms of indirect addressing to be used. These are relatively complicated, and we won't make much use of them in this book, because this is an introduction, not an encyclopaedia. We can, however, look at the principles that are involved, because the two indirect methods are similar in many respects.

The main principle is to make use of the zero page addresses in pairs. In any of these pairs, we can store an address in two bytes. The order of the bytes is the one that should be familiar to you by now, low-byte and then high-byte. An indirect addressing method makes use of the first of a pair of these zero page addresses. The effect of a command which makes use of indirect addressing is therefore to read the byte in the first of the page zero addresses, and place this into the lower half of the Program Counter register. The next zero page address is then read, and the byte in it is put into the higher half of the Program Counter. The Program Counter now contains a full address, and this is used for loading or storing, depending on which operation has been specified. For example, if address $1Ø contained $3D and address $11 contained $7F, the effect of an indirect load from $1Ø would be to place the address $7F3D into the Program Counter, and so load the accumulator with the byte that is stored at $7F3D. Once again, it seems very complicated and unnecessary until you realise the special advantages of such a method. The special advantage is that you can alter the address that is used by altering numbers stored in the zero page memory. The operating system of the C 64 makes extensive use of indirect addressing along with the addresses that are stored in its page zero part of RAM. The indirect addressing methods of the 6502 are, in fact, rather more complicated than I have indicated here, because the index registers are also used. One of the indirect addressing methods is illustrated in examples in Chapter 7 and in Chapter 9.

## Relative addressing

Relative addressing is one of the first addressing methods that was ever used, and it is not used for many commands nowadays. Relative addressing means that the operand of an instruction can be one or two bytes, and the address that is going to be used is found by adding this number (called the 'offset') to the 'current address', which is the number in the Program Counter. It's rather like the old-style Treasure Island maps which specify 'one step left, two forward, three right ...' and so on. You don't know where this will get you until you know where to start, but when relative addressing is used in a microprocessor, the starting place is usually the address in the PC. The 6502 uses relative addressing only for its BRANCH commands, and the offset is a single byte which is treated as a signed number. The use of a single byte signed number means that we can jump to a new address which is up to 127 steps forward or 128 steps back from the present one. These branches are the machine code equivalents of GOTO, but with the difference that they can be made to depend on a condition, like the accumulator containing zero. It's as if there were one single BASIC instruction which carried out the effect of:

IF A=ϕ THEN GOTO...

We'll look at branch instructions in a lot more detail later.

## The other registers

Of the other registers, the S register is an eight-bit register that we shall leave strictly alone for the moment. It is the type of register that is called a 'stack pointer', and is used to locate bytes which the MPU has stored temporarily. If you interfere with what is stored in the S register, you may upset the operating system of the computer. The other register that is important to us is the *Processor Status* (P) register and we'll deal with it in more detail now.

## The P register

The Processor Status register, sometimes called the Flag register, isn't really a register like the others. You can't do anything with the bits in this register, and they don't even fit together as a number. What the Status register is used for is as a sort of electronic notepad.

Seven of the bits in the register (there are eight of them altogether) are used to record what happened at the previous step of the program. If the previous step was a subtraction that left the A register storing zero, then one of the bits in the Status register will go from value Ø to value 1 to bring this to the attention of the MPU. If you add a number to the number in an accumulator, and the result consists of nine bits instead of eight (Fig. 4.2) then another of the bits in the Status register is 'set', meaning that it goes from Ø to 1. If the most significant bit in a register goes from Ø to 1 (which might mean a negative number), then another of the Status bits is set. Each bit, then, is used to keep a track of what has just happened. What makes this register important is that you can make branch commands depend on whether a Status bit is set (to 1) or reset (to Ø).

```
Number in accumulator    1Ø11Ø11Ø
Number added             11ØØØ1Ø1

-------------------------------

Result                  1Ø1111Ø11

This consists of nine bits, and the accumulator
can hold only eight.  The most significant bit
is transferred to the carry flag of the status
register.

Accumulator now holds Ø1111Ø11
Carry bit is set (equal to 1)
```

*Fig. 4.2.* Why the carry bit is needed.

Figure 4.3 shows how the bits of the Status register of the 6502 are arranged. Of these bits, Ø,1 and 7 are the ones that we are most likely to use at the start of a machine code career. The use of the others is rather more specialised than we need at the moment. Bit Ø is the

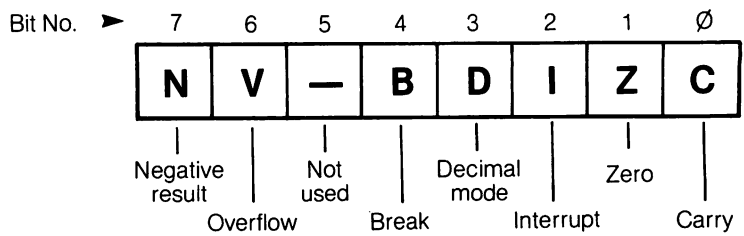



*Fig. 4.3.* The bits of the Processor Status register. Only three of these, N,Z, and C, are extensively used in most programs.

Carry Bit (or Carry Flag). This is set (to 1) if a piece of addition has resulted in a carry from the most significant bit of a register. If there is no carry, the bit remains reset. When a subtraction is being carried out (or a similar operation like comparison), then this bit will be used to indicate if a 'borrow' has been needed. It can for some purposes be used as a ninth bit for the accumulator, particularly for shift and rotate operations in which the bits in a byte are all shifted by one place (Fig. 4.4).

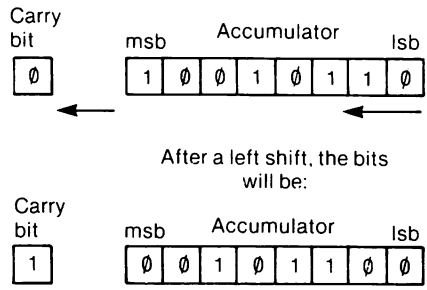



*Fig. 4.4.* Using the Carry Bit (or Carry Flag) in a shift operation, in which all the bits of a byte are shifted one place to the left.

The Zero Flag is bit 1. It is set if the result of the previous operation was exactly zero, but will be reset (0) otherwise. It's a useful way of detecting equality of two bytes – subtract one from the other, and if the zero flag is set, then the two were equal. The Negative Flag is set if the number resulting in a register after an operation has its most significant bit equal to 1. This is the type of number that might be a negative number if we are working with signed numbers. This bit is therefore used extensively when we are working with signed numbers.

You can alter some of the bits of the Status register selectively – but that's not beginner's work! For the most part, we don't load anything into this register, or store its content. It is used almost exclusively as a way of keeping track of what has just been done, and that's how we shall illustrate its use in this book. Other dodges can wait until you're an expert.

Chapter Five

# Register Actions

## Accumulator actions

Since the accumulator is the main single-byte register, we must now list its actions and describe them in detail. Of all the accumulator actions, simple transfer of a byte is by far the most important. We don't, for example, carry out any form of arithmetic on ASCII code numbers, so that the main actions that we perform on these bytes are loading and storing. We load the accumulator with a byte copied from one memory address, and store it at another. Very few computer systems allow a byte to be moved directly from one address to another, so that the rather clumsy-looking method of loading from one address and storing to another is used almost exclusively.

The next most important group of actions is the arithmetic and logic group, which contains addition, subtraction, AND, and OR. We can add to this group the SHIFT and ROTATE actions which we looked at briefly in the previous chapter. The effects of the 6502's shift and rotate commands, with their assembly language mnemonics, is shown in Fig. 5.1. A shift always results in a register losing one of its stored bits, the one at the end which is shifted out. Both types of shifts cause the register to gain a zero at the opposite end. The carry bit is used as a ninth bit of the accumulator in both of these shifts. The shift action can be carried out on either the 'A' register (the accumulator) or on a byte that is stored in the memory. The effect of a shift on a binary number stored in the register is to multiply the number by two if the shift is left, or to divide it by two if the shift is right (Fig. 5.2). A rotation, by contrast, always keeps the same bits stored in the register, but the positions of the bits are changed. The 6502 has two rotate commands, one for rotate left and the other for rotate right. Once again, they use the carry bit as the ninth bit of the register. Either the accumulator, A, can be used, or the action can be carried out on a byte stored in the memory.

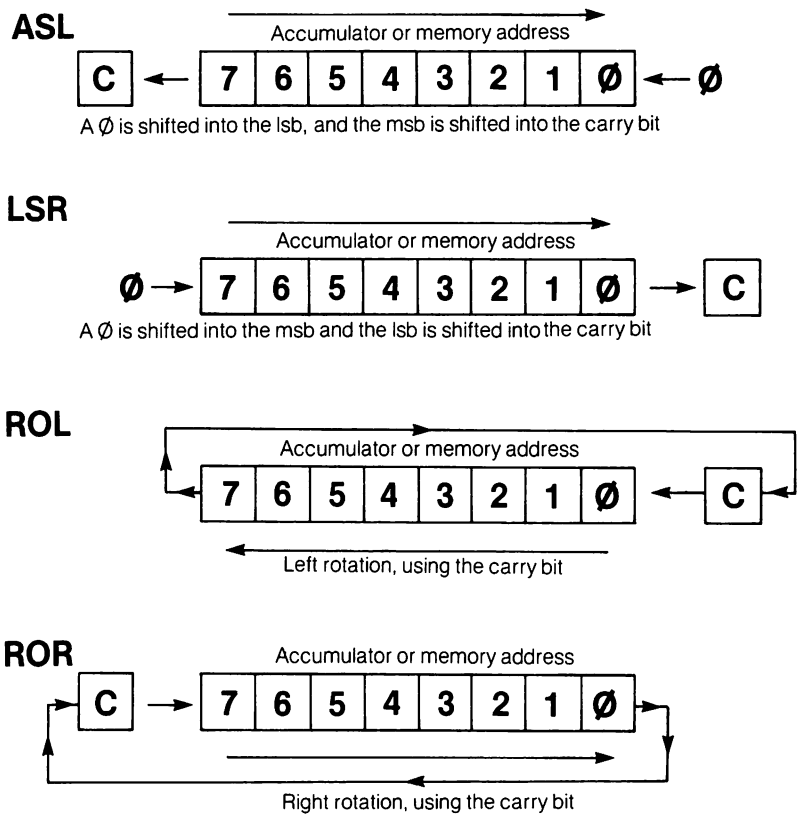



*Fig. 5.1.* The 6502 shift and rotate instructions: ASL (Arithmetic Shift Left), LSR (Logic Shift Right), ROL (Rotate Left) and ROR (Rotate Right).

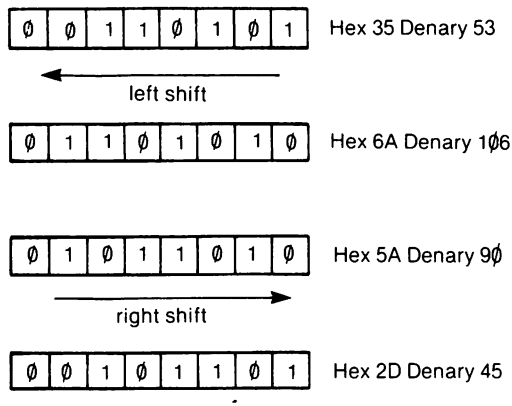



*Fig. 5.2.* The effect of a shift on a number.

The 6502, unlike most other microprocessor types, does not allow the accumulator to be used for increment or decrement actions. Increment means adding 1 and decrement means subtracting 1. These actions can be carried out only on bytes stored in the memory, so that the 6502 programmer must increment or decrement bytes before or after use. This may mean that a byte has to be returned to memory to be incremented or decremented. It's just as easy, however, to use ADD immediate to perform an increment. An immediate subtract is not quite so simple, because the carry flag in the P register (the processor status register) must be cleared to zero before a subtraction is carried out. If the carry flag is not cleared, the result of subtracting 1 will be to subtract 2! The accumulator, however, can be used for the important comparison commands.

The CMP (Compare) instruction is a particularly useful one. CMP is the mnemonic and it must use one of the standard memory addressing methods. The effect of CMP is to compare the byte that was copied from the memory with the byte that is already present and stored in accumulator A. 'Compare' in this respect means that the byte copied from memory is subtracted from the byte in the accumulator. The difference between this instruction and a true subtraction is that the result is not stored anywhere! The result of the subtraction is used to set flags in the P register, but nothing else, and the byte in the accumulator is unchanged. For example, suppose that the accumulator contained the byte $4F, and we happen to have the same size of byte stored at address $327F. If we use the command:

```
CMP $327F
```

then the zero flag in the CC register will be set (to 1), but the byte in the accumulator will still be $4F, and the byte in the memory will still be $4F. A subtraction would have left the content of the accumulator equal to zero.

Why should this be important? Well, suppose you want a program to do one thing if the 'Y' key is pressed, and something different if the 'N' key is pressed. If you arrange for the machine code program to store into the accumulator the ASCII code for the key that was pressed, you can compare it. By comparing it with $4E (the ASCII code for 'N'), we can find if the 'N' key was pressed. If it was, the zero flag will be set. If not, we can test again. By comparing with $59, we can find if the 'Y' key was pressed – once again, this would cause the zero flag to be set. If neither of these comparisons caused the zero flag to be set, we know that neither the 'Y' nor the 'N' key was

pressed, and we can go back and try again. If it looks very much like the action that you can program using the GET$ loop in BASIC, you're right – it is.

Finally, we have the test-and-branch actions. These, as the name suggests, allow the flags in the P register to be tested, and will make the program branch to a new address if a flag was set. Which flag? That depends on which branch-and-test instruction you use, because there's a different one for each main flag, and for each state of a flag. For example, consider the two tests whose mnemonics are BEQ and BNE. BEQ means 'branch if equal to zero'. As this suggests, it will cause a branch if the result of a subtraction or comparison is zero. In other words, it causes a branch to take place if the zero flag is set. Its 'opposite number', BNE, means 'branch if not equal to zero'. It will cause a branch to take place if the zero flag is not set. There are, therefore, two branch instructions which test the zero flag, but in opposite ways. The same sort of thing goes for several of the other flags. There's also a different type of branch

---

*BCC Branch on carry clear:* Jump to a new address if the carry flag is reset (at ∅).

*BCS Branch on carry set:* Jump to a new address if the carry flag is set (at 1).

*BEQ Branch if equal to zero:* Jump to a new address if the zero flag is set (at 1).

*BNE Branch if not equal to zero:* Jump to a new address if the zero flag is not set (zero flag at ∅).

*BMI Branch on result minus:* Jump to a new address if the result of the previous operation was a negative number (N flag set).

*BPL Branch on result positive:* Jump to a new address if the result of the previous operation was a positive number, or zero (N flag reset).

*BVC Branch on overflow clear:* Jump to a new address if the overflow flag is ∅.

*BVS Branch if overflow set:* Jump to a new address if the overflow flag is set. This will happen if the action of adding or subtracting causes the sign bit (the most significant bit) to change incorrectly.

*Note:* All of the above instructions use PC-relative addressing. The command byte has to be followed by a single byte, called the displacement, which is added to the address in the Program Counter register to obtain the new address.

*JMP Jump to new address:* Jump to the new address given by the two bytes that follow the JMP code.

---

*Fig. 5.3.* The complete list of 6502 BRANCH instructions.

instruction, mnemonic JMP, which doesn't carry out any tests, like a GOTO with no IF preceding it.

The complete list of all the available branch instructions is shown in Fig. 5.3. Many of these are instructions that you'll probably never use, and the really important ones are the ones that use the zero, carry and negative flags. All of them, with the exception of JMP, use relative addressing. This means that one single byte must follow the code for the branch. This number is treated as a signed number (in other words, if it's more than $7F, denary 127, it's treated as negative) and is added to the address which is in the PC at the instant when the branch is carried out. The result of this addition is the address to which the branch goes, so the next instruction that is carried out will be the one at this new address. This type of branch, which uses a single byte 'displacement' number, permits a shift of up to 127 (denary) places forward, or 128 backward. That's because a single signed byte can't exceed these values.

## Interacting with C 64

The time has come now to start some practical machine code programming of your C 64. This is not simply a matter of typing the assembly language lines as if they were lines of BASIC. Unless you happen to have an assembler program cartridge fitted, the C 64 will simply give you an 'SN ERROR' message when you try to run these programs. Since we want to start on a small scale, we'll forget about assemblers at the moment, and assemble 'by hand'. This means that we find the machine code bytes that correspond to the assembly language instructions by looking them up in a table. We then have to convert the hex codes and data numbers into denary numbers. We then poke these numbers into the memory of the C 64, place the address of the first byte into the PC of the 6502, and watch it all happen. It sounds simple, but there is quite a lot to think about, and a number of precautions to take. To start with, the C 64 uses quite a lot of its RAM, as we have seen, for its own purposes. If we simply POKE a number of bytes into the memory without heeding which part of memory we use, the chances are that we shall either replace bytes that the C 64 needs to use, or our program bytes will be replaced by the action of the C 64. What we need is a piece of memory that is safely roped off for our use only.

This can be done by making use of the fact that the C 64 can shift its BASIC programs about in the same way as it can shift the

Variable List Table. The easiest shift to carry out is the end of memory. The last byte of RAM that a program of your own can normally make use of is 4Ø96Ø (denary number). Addresses right at the end of memory like this are normally used for storing strings that are not already present lower in memory. We have already looked at this principle in Chapter 2. Now the C 64 is not programmed to stop at this number automatically. What is done is to store this 'end-of-RAM' number, as two bytes, in memory locations 55 (low byte) and 56 (high byte). These are page zero addresses, and in the course of operating a BASIC program, the computer will be continually testing that it does not try to use locations higher than the number stored in these addresses.

Suppose, then, that we alter this number. We don't have to alter both bytes, because if we reduce the high byte by one, we will have reserved 256 bytes for our own uses. This is because a number stored as two bytes is 256 times the high byte plus the low byte. When we switch the C 64 on, the number that is stored in location 56 is 16Ø. If we use POKE56,159, then, we will have reserved 256 bytes, starting at address 4Ø7Ø5 and going up to 4Ø96Ø. Remember that we count both the first byte and the last byte. Once we have done this, we can be sure that any BASIC program that we run will not use addresses above 4Ø7Ø4, and therefore cannot interfere with our machine code programs.

The other problem, then, is how to place the starting address for your program into the program counter of the 6502. Fortunately, the designers of the C 64 have been kind to you. There is a BASIC command SYS which will do this for you. SYS has to be followed by a number, and this number will be placed into the PC. It will therefore be used as the address of the first byte of your program. Incidentally, I've taken this as meaning 'starting byte'. It's possible to write programs in which the first few bytes are data, so that the program starts at, say, the tenth byte. This creates no problems; you simply use the address of the starting byte as the number for SYS.

Lastly, for the moment at least, you have to ensure that your machine code program will stop in an orderly way. Nothing that we have done so far will indicate to the 6502 of the C 64 where your program ends. As a result, the 6502 could continue to read bytes after the end of your program, until it encounters some byte which causes a 'crash'. This might, for example, be a byte which causes an endless loop. Some programmers doubt if there are any bytes which do not cause an endless loop in these circumstances! To return correctly to the operating system of the C 64, you need to end each

machine code program with a 'return from subroutine' instruction, whose mnemonic is RTS and whose code is $6Ø.

There's another headache that we don't have to worry about at the moment. When you run a machine code program along with a BASIC program in your C 64, you are using the same 6502 microprocessor for both jobs. It can't cope with both at the same time, so it runs one, then the other. If you make use of the 6502 registers in your machine code program, as you are bound to do, then you have to be quite certain that you are not destroying information that the BASIC program needs. For example, if at the instant when your machine code program started, the registers of the 6502 contained the address of a reserved word in the ROM, then it will need this address in these registers when your machine code program ends. When a machine code program is called into action by using the SYS command, this is taken care of automatically. The contents of the registers of the 6502 are placed into a part of the RAM memory which is called the 'stack'. This, incidentally, is another good reason for being careful as to where you place your machine code in the memory. If you wipe out the stack, the C 64 will quite certainly not like it! The stack is located in the range of memory between addresses 256 and 511. When the RTS instruction is encountered at the end of your machine code, the bytes that have been stored in the 'stack' are replaced into their registers, and normal action resumes. If you call a machine code program into action by any other method, not using SYS, you may have to attend to this salvage operation for yourself as part of your machine code program. This involves using the PUSH and PULL commands – but more of that later.

## Practical programs at last

With all of these preliminaries out of the way, we can at last start on some programs which are very simple, but which are intended to get you familiar with the way in which programs are placed into the memory of the C 64. You will also get some experience in the use of assembly language and machine code, and with how a machine code program can be run.

We'll start with the simplest possible example – a program which just places a byte into the memory. In assembly language, it reads:

```
ORG 4Ø7Ø5; start placing bytes here
LDA #$55; place hex 55 in accumulator
STA $9FC4; store them at 9FC4
RTS; go back to BASIC
```

The first line contains a mnemonic, ORG, which you haven't seen before. It isn't part of the instructions of the 6502, but it is an instruction to the assembler, which in this case is you! ORG is short for origin, and it's a reminder that this is the first address that will be used for your program. We've chosen to use an address which leaves space for longer programs than we shall be writing in the course of this book, and we could have chosen a higher number. It will do as well as any other, however, and it leaves plenty of room for longer programs. When you program using an assembler, this line can be typed and the assembler will then automatically place the bytes of the program in the memory starting at this address. As it is, with assembly being done 'by hand' it simply acts as a reminder of what addresses to use. Note the comments which follow the semicolons. The semicolon in assembly language is used in the same way as a REM in BASIC. Whatever follows the semicolon is just a comment which the assembler ignores, but which the programmer may find useful.

Now we need to look at what the program is doing. The first real instruction is to load the number $55 into the 'A' accumulator. This uses immediate addressing, so the number $55 will have to be placed immediately following the instruction. The hashmark, '#', is used in assembly language to indicate that immediate loading is to be used. The next line commands the byte in the accumulator (now $55) to be stored at address $9FC4. In denary, this is 4Ø9ØØ. It's an address well above the ones that we shall use for the program. Obviously, we wouldn't want to use an address which was also going to be used by the program. This instruction uses absolute addressing. Finally, the program ends with the RTS instruction, essential for ensuring that C 64 life continues normally after our program ends.

The next step in programming is to write down the codes in hex. Each code has to be looked up, taking care to select the correct code for the addressing method. The code for LDA immediate is $A9, so this is the first byte of the program which will be stored at address (denary) 4Ø7Ø5. We can start a table of address and data numbers with this entry:

```
4Ø7Ø5   $A9
```

and then move on. The byte that we want to load is $55, and this has to be put into the next memory address, because this is how immediate addressing works. The table now looks like this:

| | |
|---|---|
| 4Ø7Ø5 | $A9 |
| 4Ø7Ø6 | $55 |

The next byte we need is the instruction byte for STA, with absolute addressing. This byte is $8D, and it has to be followed by the two bytes of the address at which we want the bytes stored. The address 4Ø9ØØ translates into hex as $9FC4, so we can use the bytes $C4 and $9F following the STA instruction. Remember that these bytes have to be in low-high order. The last code has to be the RTS code of $6Ø, so that the table now looks as in Fig. 5.4. It uses addresses 4Ø7Ø5 to 4Ø71Ø, six bytes in all, and will place a byte into 4Ø9ØØ, using denary numbers. Now we have to put it into memory and make it work!

| | |
|---|---|
| 4Ø7Ø5 | A9 |
| 4Ø7Ø6 | 55 |
| 4Ø7Ø7 | 8D |
| 4Ø7Ø8 | C4 |
| 4Ø7Ø9 | 9F |
| 4Ø71Ø | 6Ø |

*Fig. 5.4.* The coded program, using denary addresses and hex bytes of data.

This requires a BASIC program which will clear the memory, and poke the bytes in one by one. Before we can write this program, we have to convert each hex byte into denary, because the POKE instruction of the C 64 uses only denary numbers. You can convert by using a calculator or by means of the program which was illustrated in Chapter 3. The BASIC poke program is shown in Fig. 5.5. By using POKE56,159 we ensure that all memory addresses above 4Ø7Ø4 are left unused by the C 64. We declare the variable A

```
10 POKE56,159:A=40704
20 FORN=1TO6:READ D%
30 POKEA+N,D%:NEXT
40 SYS40705
100 DATA169,85,141,196,159,96
```

*Fig. 5.5.* The BASIC program which pokes the bytes into place. Note how the integer D% has been used for the data, and how the program is made to run by SYS4Ø7Ø5.

as 4Ø7Ø4, so that we can make use of this in the POKE commands. Lines 2Ø to 4Ø then poke data numbers into addresses that start at 4Ø7Ø5. Why 4Ø7Ø5? Well, we have used POKEA+N, and with A=4Ø7Ø4 and N=1, the first address just has to be 4Ø7Ø5. All of the codes have to be put into denary form for use with the BASIC poke program, and the only problems arise when you have an address number in denary, and you have to convert it into two denary byte numbers. The method of doing this is shown in Appendix B, following the denary-hex conversions. It's better in many ways, however, to work in hex as much as possible, and convert to denary only when you must. Since you have to make use of hex when you graduate to the MIKRO assembler, or any other assembler, it's as well to start getting familiar with the principles of working in hex now.

The last program line, line 4Ø, contains SYS4Ø7Ø5. This is the BASIC instruction which will cause your machine code program to run, with the start address specified. Line 1ØØ then contains the six bytes of data that we have worked out. When you RUN this, there's no obvious effect. That's because you can't see what's in address 4Ø9ØØ. If you use:

```
?PEEK(4Ø9ØØ)
```

then you should find the value of 85, which is the denary version of $55, the number that the program put there. Now try this: type POKE4Ø9ØØ,255, press RETURN, and then delete line 5Ø of your program. This is the SYS line. RUN the program again, and use ?PEEK(4Ø9ØØ) to find what's there. It should be 255. Now type SYS4Ø7Ø5 and press RETURN. Using ?PEEK(4Ø9ØØ) should now give you 85 again. This is because poking the bytes of the program into memory won't make the program run, only SYS does this. You can, therefore, poke values into memory early in a BASIC program, and then make use of them later with an SYS wherever you like.

Now this program isn't an ambitious piece of work, it does no more than POKE4Ø9ØØ,85 would do in BASIC, but it's a start. The main thing at this point is to get used to the way in which machine code operates, and how you place it into memory and run it. Another point, incidentally, is that the machine code is safe in memory. If you type NEW (RETURN), the BASIC program will be cleared out, but your machine code remains. If you POKE4Ø9ØØ,255 now, and test with ?PEEK(4Ø9ØØ), you will find that this address can still be changed by using SYS4Ø7Ø5. These bytes will remain there until you make an effort to change them, or you switch off.

You can preserve the machine code program on tape if you like, and this is a technique that we'll look at later. One step at a time, if you please! Another thing we'll leave for later is the alternative method of calling up a program, using the USR command.

Now let's try something a lot more ambitious in terms of our use of machine code – though the example is simple enough. Figure 5.6 shows the assembly language version of the program. What we are going to do is to load a byte into the accumulator, shift it one place left, and then put it into memory at an address one step higher than the address from which we took it. This looks like an open-and-shut

| | |
|---|---|
| LDX #$Ø | A2 ØØ |
| LDA $9FC$,X | BD C4 9F |
| ASL A | ØA |
| INX | E8 |
| STA $9FC4,X | 9D C4 9F |
| RTS | 6Ø |

*Fig. 5.6.* The assembly language program for 'multiply by two'. The listing shows the assembly language on the left, and the hex codes on the right.

case for indexed addressing, so we shall start by placing a zero into the X register. This is the LDX #$ØØ step. As before, the '#' means immediate addressing. The next line, LDA $9FC4,X means that the accumulator is to be loaded from the address $9FC4, plus the number in the X register. This makes the load come from $9FC4 (4Ø9ØØ denary). The third step is ASL A, arithmetic left shift of the byte in the accumulator, so that the bits of the byte are shifted left. Fourthly, we increment the number in the X register, using INX. This makes the Ø into 1, and we now store the byte in the accumulator at address $9FC5 by using STA 9FC4,X. This time, because 1 has been added to the number in the X register, the byte is stored at address $9FC5. We end, as always, with the RTS instruction.

Now we can put this into code form. It's just as easy as before, despite the use of indexing. The LDX instruction needs the immediate loading code of $A2, and this has to be followed by the byte of the data, $ØØ. The LDA with indexed addressing is coded as $BD, and it has to be followed by the two bytes of the address $9FC4, in their usual low-high order. The ASL byte is ØA, and then we perform the incrementing of X with INX, code $E8. We then store the byte that is in the accumulator back into memory, using STA

$9FC4,X. The STA X indexed code is $9D, and this is followed by the now-familiar address bytes. Finally, $6ø is the RTS command.

Now we have to code this in BASIC. If we choose a small number to place into $9FC4, the effect of the left shift will be to double the number, so we can use this to obtain a bit of arithmetic wizardry.

```
10 POKE 56,159:A=40704
20 FOR N=1 TO 11:READ D%
30 POKEA+N,D%:NEXT
40 POKE40900,7:SYS40705
50 PRINT"TWICE ";PEEK(40900);" IS ";PEEK(40901)
100 DATA162,0,189,196,159,10,232,157,196,159,96
```

*Fig. 5.7.* The BASIC program which pokes the bytes into place and then makes use of the machine code program.

The BASIC program is shown in Fig. 5.7. We start, as usual, by clearing memory space. You needn't worry if you have had another program in this part of the memory before. The new program will replace it completely, and provided that your program ends correctly with the RTS instruction byte, the old program bytes cannot interfere with the new ones. The values are poked into places in the usual way in lines 2ø to 3ø. In line 4ø, however, we place a number, 7 denary, into the address $9FC4 (4ø9øø denary). Now this is the address which will be used by the program, and the byte which is 7 in binary form, øøøøø111, will be placed in this address. In the second part of line 4ø, SYS4ø7ø5 will carry out the machine code program, which should left-shift this byte, making it øøøø111ø. In denary, this is 14, twice 7. Line 5ø prints this result, and line 1øø contains the data bytes.

It's simple enough, but if you knew nothing about machine code, you would wonder how on earth the number became multiplied by two. Once again, the program does nothing that could not be done more easily and as quickly by using BASIC. The important thing, from our point of view, is that you have now used indexed addressing, and a shift instruction, as well as getting more experience in putting a machine code program into your C 64 by the hardest method of all. If, incidentally, you have made any mistakes, particularly with DATA, then it's likely that the C 64 will go into a trance and refuse to do anything. When you have typed in a BASIC program like this which pokes bytes into the memory, always record the BASIC program before you RUN it. This way, if the effect of an incorrect byte is to zonk out half the RAM, you can switch off, then on again, and reload your program. If you didn't record it, then

you'll have to type it all over again. That's hard work, and life is hard enough as it is. Another final point concerns the number that you poke into 4Ø9ØØ. If this is a small number, then the program works. You can't, however, poke a number greater than 255 into any single memory position. In addition, if the number that you place there amounts to more than 127 when it is multiplied by two, then the results of this program will appear very strange! That's because a number greater than 127 is treated as being negative. The routines that multiply numbers in your C 64 are a lot more sophisticated than this simple example!

Chapter Six

# Taking a Bigger Byte

The simple programs that we looked at in Chapter 5 don't do much, though they are useful as practice in the way that machine code programs are written. Practising assembly language writing and its conversion into machine code is essential at this stage, because you can more easily find if you are making a mistake when the programs are so simple. It's not so easy to pick up a mistake in a long machine code program, particularly when you are still struggling to learn the language!

Most beginners' difficulties arise, oddly enough, because machine code is so simple, rather than because it is difficult. Because machine code is so simple, you need a large number of instruction steps to achieve anything useful, and when a program contains a large number of instruction steps, it's more difficult to plan. The most difficult part of that planning is breaking down what you want to do into a set of steps that can be tackled by assembly language instructions. For this part of the planning, flowcharts are by far the most useful method of finding your way around. I never think that flowcharts are ideally suited for planning BASIC programs, but they really come into their own for planning machine code.

## Flowcharts

Flowcharts are to programs as block diagrams are to hardware – they show what is to be done (or attempted) without going into any more detail than is needed. A flowchart consists of a set of shapes, with each shape being the symbol for some type of action. Figure 6.1 shows some of the most important flowchart shapes for our purposes (taken from the standard set of flowchart shapes). These are the terminator (start or end), the process (or action), the input/output and the decision steps. Inside the shapes, we can write

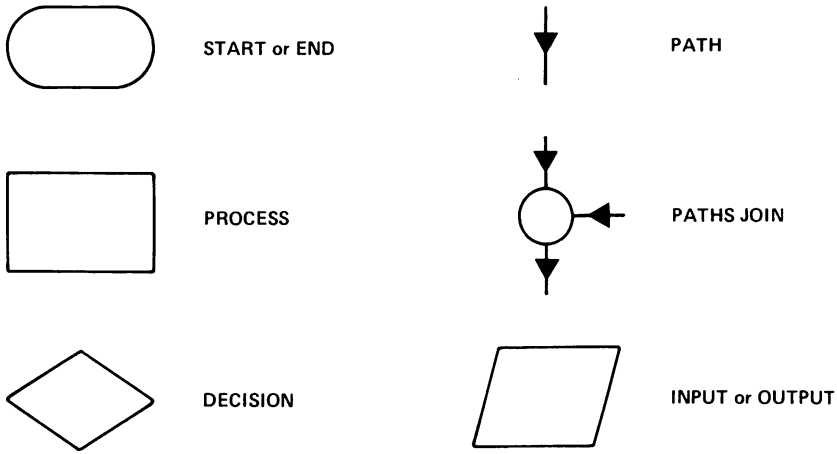



*Fig. 6.1.* The main flowchart shapes.

brief notes of the action that we want, but once again without details.

An example is always the best way of showing how a flowchart is used. Suppose that you want a machine code program that takes the ASCII code for a key that has been pressed, and prints the character corresponding to that key. A flowchart for this action is shown in Fig. 6.2. The first terminator is 'START', because every program or piece of program has to start somewhere. The arrowed line shows that this leads to the first 'action' block, which is labelled 'get character in A'. This describes what we want to do – get the code

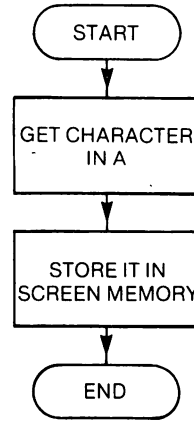



*Fig. 6.2.* A flowchart for a 'print-a-character' program.

number for a character in the accumulator. We don't know how we're going to do this at present – that comes later. After getting the character, the arrow points to the next action, storing the byte in screen memory. That's how we carry out the 'print' part of the action, and it's something that we've looked at earlier. The END terminator then reminds us that this is the end of this piece of program, it's not an endless loop.

This is a very simple flowchart, but it is enough to illustrate what I mean. Note that the descriptions are fairly general ones – you don't ever put assembly language instructions inside the boxes of your flowchart. Strictly speaking, I should not have referred to the accumulator A in the 'get character' box, but my excuse is that I need to be reminded of where the code is to be stored. A flowchart should be written so that it will show anyone who looks at it what is going on. It should never be something that only the designer of the program can understand and use, and which just confuses anyone else. A good flowchart, in fact, is one that could be used by any programmer to write a program in any variety of machine code – or in any other computer 'language', such as BASIC, FORTH, PASCAL and so on. A lot of flowcharts, alas, are constructed after the program has been written (usually by lots of trial and error) in the hope that they will make the action clearer. They don't, and you wouldn't do that, would you?

Once you have a flowchart, you can check that it will do what you want by going over it very carefully. In the example, the actions of 'get character' and 'store in screen memory' are going to be done using machine code, so we'll concentrate on them. Getting the ASCII code for a character looks tricky at first. A lot of computers, however, put the ASCII code for the last character that was used into an address in memory. This is where a good knowledge of the way that the C 64 uses its memory comes in handy! Appendix E shows some of these important addresses. One is of particular interest, the address 512 denary ($2ØØ). This is the address of the start of the 'keyboard buffer'. A buffer is a section of RAM memory which the computer uses to store data temporarily. As the name suggests, the keyboard buffer is used to store the code numbers for keys that you have pressed. These codes then remain in the buffer until the RETURN key is pressed. Pressing the RETURN key then shifts everything that has been in the keyboard buffer into another part of memory. The importance of this address is that we can use it to look for a code number for the last key that was pressed. If we load the accumulator from this address we may, as the advertise-

ments say, learn something to our advantage. The first step, then, looks like loading the accumulator, using extended addressing, so that we can make use of address 512.

The second step, of storing the byte in the screen memory, is straightforward. The memory that we shall use is the 'text memory' which is in the range of 1Ø24 to 2Ø23 denary. How about a spot right in the centre of the screen, at 1524? You want to know how I got to that number? Well, if we take the range 1Ø24 to 2Ø23, that's 1ØØØ addresses, including the first and the last. Half of that is 5ØØ, so if we add 5ØØ to 1Ø24, we get 1524, which must be the address for the centre of the middle line on the screen. A 'store accumulator to 1524' should therefore get us where we want.

Now we can design the assembly language part of the code. We can follow the path we have trod before, and start the code at address 4Ø7Ø5 (denary). This makes our assembly language code look like this:

```
ORG 4Ø7Ø5
LDA $Ø2ØØ
STA $Ø5F4
RTS
```

We can now assemble this by hand, finding the codes that are needed to make this run. The codes – all in hex – are:

```
AD ØØ Ø2
8D F4 Ø5
96
```

If we now convert these into denary, we have the codes that we can use in a 'screen-poke' program. We can then put this into the form of a BASIC program which pokes the codes into memory, and then calls the machine code program. It will look as Fig. 6.3, so we can enter it and run it. Another small step for a C 64 user!

```
10 POKE56,159:A=40704
20 FORN=1TO7:READ D%
30 POKEA+N,D%:NEXT
40 GET A$:IF A$=""THEN 40
50 SYS40705
60 POKE53281,243
100 DATA173,0,2,141,244,5,96
```

*Fig. 6.3.* The BASIC poke program for printing a character.

Well, it works, but not as we might expect, and the program shows what has had to be added. To start with, we need some way of getting a code into the first buffer address of 512 ($Ø2ØØ). The program achieves this by using a GET A$ loop in line 4Ø. The second point is that nothing appears on the screen until the colours have been adjusted, and this is done in line 6Ø. If you type this program in and run it, it will work. To make it work again, you will have to restore the colour registers by using the STOP/RESTORE keys together. If you don't, the program still works, but you can't see any results on the screen.

Now in our present state of knowledge, we are doing the looping in BASIC, and then calling the machine code program as soon as the BASIC has detected a key being pressed. Rather than waste any more time over this rather half-hearted method, though, let's try the complete machine code approach, even at the risk of making some mistakes on the way.

## Loop back in hope

Since this is a simple program, it looks like a good opportunity to get an introduction to looping. If you have done anything more than the most elementary BASIC programming, you will know what a loop involves. A loop exists when a piece of program can be repeated over and over again until some test succeeds. In BASIC, you can cause a loop to happen by using a line which might read, for example:

```
2ØØ IF A=Ø THEN GOTO 1ØØ
```

This contains a test (is A=Ø?), and if the test succeeds (yes, A is Ø), then the program goes back to line 1ØØ and repeats all the steps from there to line 2ØØ again. That sort of loop in BASIC corresponds very closely to how we create a loop in machine code. Instead of using line numbers, however, we are using address numbers. Instead of testing a variable called 'A', we shall test the contents of a register, which in this case can be the 'A' register.

Let's start the proper way with a flowchart. Figure 6.4 shows how this might look. The first step is the same – get the character code in the accumulator A. The next step, however, is a 'decision' step. The decision is 'is it Ø?'. All decision steps in flowcharts must be worded so that there can only be two possible answers, yes or no. This is indicated by having two arrowed paths from the decision step. One of these is labelled 'YES'. It leads back to the first step of the

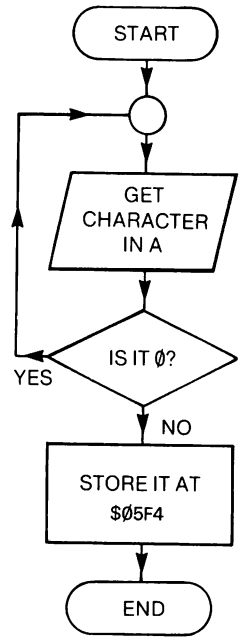



*Fig. 6.4.* Another flowchart - this one has a loop which rejects the zero character.

program, the step that requires the memory to be loaded to the accumulator. Why? Because if we find that we have a zero in the accumulator it means that there's no key pressed, and we have to go back and try again. The other path, the one that is labelled 'NO', leads to the next action step, storing the accumulator byte in the screen memory.

The action, then, will be that the accumulator is loaded from address 512, and the byte in the accumulator is tested to see if it is zero. If it is, we repeat the loading. If it isn't (which means that a key was pressed), then we store the byte in the screen memory. Now we have to put this into assembly language form - and that's going to introduce some new items and problems to you.

Figure 6.5 shows an assembly language program which should carry out the effect of the flowchart. There's one more step in this flowchart, and one alteration to an existing step. The new step is the 'BEQ LOOP', and the change is to the first step, which now has LOOP: stuck in front of it. This word LOOP is a 'label'. It's being used here in place of an address, and it means the address at which the instruction starts. With LOOP: placed in front of the LDA 512 instruction, the word LOOP means the address at which the LDA

```
LOOP:   LDA $Ø2ØØ
        BEQ LOOP
        STA $Ø5F4
        RTS
```

*Fig. 6.5.* The assembly language program corresponding to the flowchart.

instruction byte is stored. By using words in this way, we avoid having to think about address numbers until we actually write the machine code. If we use an assembler, we usually don't have to worry about address numbers at all – the assembler automatically puts in address numbers in place of label words. The same label word is also used in the next step. BEQ means 'branch if the register is equal to zero', so the effect of BEQ LOOP is that the program should go back to the address of the LDA instruction if the accumulator contains zero. It's rather like using a version of BASIC which allowed variables to be used in place of line numbers (as some do).

In assembly language, this all looks quite neat and straightforward. If we were using an assembler it would be straightforward, but when we assemble by hand, it's not so simple. The reason is that we have to follow the BEQ instruction by a single byte which will give the address of the LDA instruction. This is PC-relative addressing, so that we have to use a signed byte that can be added to the address in the program counter to give the address of the LDA step. The formula is shown in Fig. 6.6. What you have to do is to find the address that you want to jump to, and the address of the branch command. Subtract these, then subtract 2 from the result. What you have now is the size of the 'displacement byte' that you need to follow the branch instruction. Since this number is negative, we have to convert it to the form of a signed byte, using the procedure that we looked at earlier.

*Destination:* address you want to jump to (which has label in front of assembly instruction).

*Source:* address you want to jump from (address of branch code – in assembly language; it is the instruction with the label name *after* it).

*Displacement:* Destination minus Source minus 2 put into hex form.

*Fig. 6.6.* The formula for finding the size of a displacement byte.

| | |
|---|---|
| 4Ø7Ø5 | 173 |
| 4Ø7Ø6 | Ø |
| 4Ø7Ø7 | 2 |
| 4Ø7Ø8 | 24Ø |
| 4Ø7Ø9 | displacement byte |

The 'source' address is 4Ø7Ø8, where the BEQ byte is placed.
The 'destination' address is 4Ø7Ø5, the LDA command. The procedure is:

Destination address − source address = 4Ø7Ø5 − 4Ø7Ø8 = −3
Now subtract another 2, so that −3−2=−5.

In denary, the equivalent byte for −5 is 256−5=251. This must be the byte at address 4Ø7Ø9.

*Fig. 6.7.* An example of finding a displacement byte.

If all that sounds complicated, take a look at it in practice, in Fig. 6.7. Assuming that we are going to place the first byte of the program at address 4Ø7Ø5 then the address of the BEQ instruction is at 4Ø7Ø8. The number 4Ø7Ø8 is the source address, where we're coming from, and 4Ø7Ø5 is the destination address, where we're going to. Subtract source from destination numbers, and we get −3. Subtract another 2 from this, and we get −5. −5 in hex is FB, so that's the displacement byte that is placed following the BEQ instruction code.

```
10 POKE56,159:A=40704
20 FORN=1TO9:READ D%
30 POKEA+N,D%:NEXT
40 GET A$:IF A$=""THEN 40
50 SYS40705
60 POKE53281,243
100 DATA173,0,2,240,251,141,244,5,96
```

*Fig. 6.8.* The BASIC program of the assembly language of Fig. 6.5.

Now when you try this program, listed in Fig. 6.8, you will find that it works, and the fact that it does gives us a clue about the C 64. A lot of computers won't run a program like this. If you have, in fact, figured out why not, award yourself a gold star. What is happening is that our program loops until a number code is put into address 512, and a code should be put into that address by the act of pressing a key. Now on a lot of computers, if the microprocessor is spending all its time looping round your program, it can't be scanning the keyboard looking for a key to be pressed! There's only one

microprocessor in the C 64, and it has to do everything. As it happens, it doesn't run your program continually. At intervals, it interrupts what it is doing to test the keyboard. If a key is pressed, then the code number for that key is put into the buffer. This type of action is called, appropriately enough, an interrupt, and the action is a very useful one. The use of this action makes it much more difficult for the C 64 to be upset by looping programs which loop incorrectly. In addition, the microprocessor does not have to keep the screen display going – if it did, you would see the screen picture disappear when you ran this program.

Is there another way? What we have to do is to write a piece of program that will attend to reading the keyboard, and place that piece of program in our loop. That's possible, but it takes a lot of time, and needs a lot of knowledge of the C 64. It also seems a trifle unnecessary, because there must be a routine in the ROM of the C 64 which will do all this for us. There is – and Appendix E lists the address for this and other useful routines. The routine which starts at address $E112 will scan the keyboard looking for a key being pressed. If no key is pressed, the number in the accumulator will be zero. If a key is pressed, the number in the accumulator will be the number code for that key. Using this routine, we don't have to make use of any memory address like 512.

The next step, then, is to see how this routine at $E112 can be used. The instruction that we need is called 'jump to subroutine', and it's abbreviated to JSR. Each subroutine in the ROM ends with the RTS code, which returns it to whatever program called it, so if we use JSR followed by the address $E112, then the subroutine will run and then return to our own program. Figure 6.9 shows a flowchart for what we are going to attempt now. We shall call up the subroutine to get the byte into the accumulator, then test it. If the byte is zero, we shall return to the 'scan keyboard' step. If not, we shall make use of another subroutine, the one which prints the code on the screen. So far, so good. Figure 6.10 shows the assembly language version of this flowchart, with the label word LOOP once more used to indicate the address to which the program is to return if the byte in the accumulator is zero. Figure 6.11 shows the program put into the form of a set of BASIC poke instructions. When this runs, pressing a key will cause a letter or other character to appear at the cursor position. This doesn't look any different from what happens when you press a key at any other time, so how do we know that this program works? Easy – just run it, and press a key. Now press the RETURN key. You will see the letter printed again, and

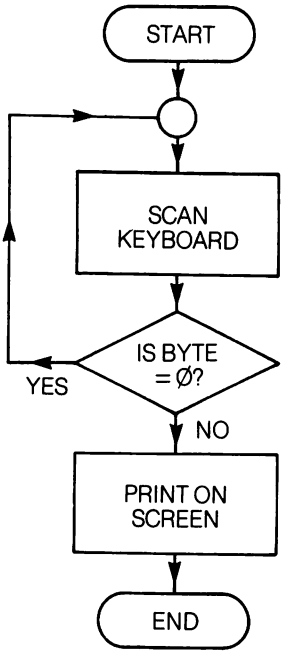



*Fig. 6.9.* A flowchart for a character printing program.

```
LOOP:   JSR $E112 ;keyboard
        BEQ LOOP
        JSR $E1ØC ;screen
```

*Fig. 6.10.* The assembly language version of the program.

```
10 POKE56,159:A=40704
20 FORN=1TO9:READ D%
30 POKE A+N,D%:NEXT
40 SYS40705
100 DATA32,18,225,240,251,32,12,225,96
```

*Fig. 6.11.* The BASIC poke program.

the READY on the next line. There's no ?SYNTAX ERROR message, as you would get if you simply typed a letter and then pressed RETURN. Why is the letter duplicated? Because the input routine also places the letter code in the keyboard buffer, and pressing RETURN deals with this. The ?SYNTAX ERROR is missing because we have short-circuited the routine. We've broken a lot of new ground in this short piece of program, so perhaps this is a

good time to go over it all carefully and make sure that you know what it has all been about before we plunge deeper into the business.

## More loops

The loop that we have tried out was a simple loop that is classified as a 'holding loop'. Its job was to keep a piece of program repeating until something happened. It's time now to take a look at another type of loop, called a *counting loop*. The importance of this one is twofold – it's the way that we program a time-delay in machine code, and it also gives me an excellent opportunity to demonstrate just how fast machine code can be.

The type of loop that you use most in BASIC is the FOR...NEXT loop. This uses a 'counter' variable to keep a score of how many times you have used the loop, and compares the value of the counter with the limit number that you have set each time the loop returns. Now the action of a FOR...NEXT loop can be simulated in BASIC without using FOR or NEXT, and the method is shown in Fig. 6.12.

```
10 C=0:ND=10
20 PRINT"ACTION ";C
30 C=C+1
40 IF C<=ND THEN 20
50 PRINT "FINISHED"
```

*Fig. 6.12.* A simple loop in BASIC which gives the action of FOR...NEXT.

The count number is C, and its limit is ND. At the end of the program, the value of C will be 11, just like the value of the counter in a FOR N=1 TO 1Ø type of loop. The next thing, then, is to take a look at the flowchart for this type of program, and that's shown in Fig. 6.13.

This method of forming a counting loop is the one that we use in machine code. We can write some assembly language that will do the same job – but, as usual, we have to give a lot more thought to how the task will be done. For one thing, we don't have variable names in machine code. We have to decide where we shall store a number, and in what register we shall carry out the task of decrementing it. The decision step is easier – we can use a BNE test this time to keep the program looping back until the content of the register that we have tested is zero. In case you're wondering how we specify which

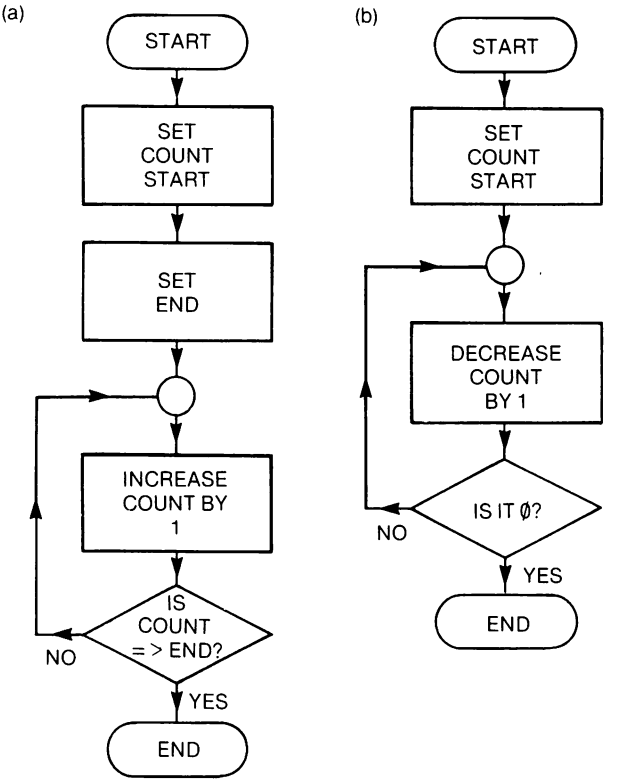



*Fig. 6.13.* Flowcharts for loops. (a) Incrementing the count number. (b) Decrementing the count number, which is simpler.

register we're testing, the answer is that it's always the one that we used just before the BNE (or any other) test.

Figure 6.14(a) shows what we end up with as an assembly language program. The register that we use is the X register, rather than the accumulator. This is because the X register (along with the Y one) is better adapted for counting operations. What makes it better adapted is the presence of increment and decrement commands. INX and DEX will, respectively, increment or decrement the X register. The commands INY and DEY will perform the same operations on the Y register. There is no corresponding pair of commands for the accumulator, and if we want to carry out counting operations using the accumulator, we have to go about them in a slightly different way. In this example, then, the X register is loaded with $FF, which is 255 in denary. This is the largest number that we can load into an eight-bit register. Having loaded the accumulator, we then decrement it, and mark this address as 'LOOP', the place we want to return to if the register

(a)

```
            LDX #$FF  A2 FF
LOOP        DEX       CA
            BNE LOOP  DØ FD
            RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO6:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"START"
50 SYS40705
60 PRINT"STOP"
100 DATA162,255,202,208,253,96
```

*Fig. 6.14.* (a) A counting loop in assembly language. (b) BASIC poke program.

content is not zero. The test is carried out by BNE (branch if not equal to zero) because we want the program to repeat the decrementing action until the contents of the X register reach zero. The BASIC program which pokes the bytes into memory and then carries out the program is shown in Fig. 6.14(b). Now when you run this one, you will not see much of a time delay between the printing of 'START' and the printing of 'STOP'. This isn't because nothing has happened, it is because the machine code countdown is so fast! If you try a BASIC version of this:

```
1Ø A=255:?"START"
2Ø A=A-1
3Ø IF A<>ØTHEN 2Ø
4Ø ?"STOP"
```

you will see that there is a noticeable pause. The difference does not reflect the comparative speeds, however, because quite a lot of time is spent in the printing actions of the BASIC part of each program. To see just how great the advantage of machine code can be in terms of speed, we need to work with much larger numbers. Now there are several ways of doing this, but one which we can look at right now involves two loops. You have probably met nested loops in BASIC. The principle is that there is an inner loop and an outer loop. On each pass of the outer loop, the whole of the inner loop is carried out.

This allows us to create much longer time delays, by doing one count inside another. Suppose we had, in BASIC, the lines:

```
1Ø X=1ØØ:?"START"
2Ø X=X-1
3Ø Y=255
4Ø Y=Y-1
5Ø IF Y<>Ø THEN 4Ø
6Ø IF X<>Ø THEN 2Ø
7Ø ?"STOP"
```

then these would carry out a countdown of Y from 255 to Ø each time the value of X was decremented. Try this one - and time it. You won't need a stop-watch - anything with a minute hand will do!

(a)

```
            LDX #$FF   A2 FF
LOOP2:      LDY #$64   AØ 64
LOOP1:      DEY        88
            BNE LOOP1  DØ FD
            DEX        CA
            BNE LOOP2  DØ F8
            RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO11:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"START"
50 SYS40705
60 PRINT"STOP"
100 DATA162,255,160,100,136,208,253,202,208,248,96
```

*Fig. 6.15.* (a) Assembly language for a two-loop counter. (b) The BASIC poke program.

For a contrast, let's see how the same numbers could be dealt with in a machine code countdown. Figure 6.15(a) shows the assembly language version. The X register is loaded with $FF, and the Y register with $64 (1ØØ denary). This second instruction is labelled 'LOOP2'. Then comes DEY, so that the Y register is decremented, and this is labelled 'LOOP1'. The BNE test then returns to this LOOP1 point until the Y register has reached Ø. After that, the X register is decremented, and then tested. Note that the order is not

quite the same as in the BASIC version. In a machine code decrement and test action, you must have the decrement done just before the test, otherwise the register that is tested may not be the correct one. If the X register has not reached zero, the program loops back, this time to LOOP2, to fill up the Y register again and perform the 'inner loop' yet again.

When you try this – it's still almost too fast to follow! It's a good illustration of the speed advantage of machine code as compared to BASIC. If you are not quite convinced that the count has been carried out, then alter the number in the outer count from $64 to $FF (replace 1ØØ by 255 in the data line 1ØØ of the BASIC program). This makes the delay slightly more noticeable.

## Accumulator INC and DEC

I pointed out earlier that there are no INC and DEC commands for the accumulator. This does not, however, mean that we can't use the accumulator for counting operations, just that it's not so well equipped as the X and Y registers. Supposing we have to carry out a count in the accumulator, however, we could make use of SBC #1, meaning subtract 1 from the content of the accumulator. This would lead us to a countdown program of the sort that is shown in Fig. 6.16(a). We start by using CLC, the command which clears the carry

(a)

```
        CLC       18
        LDA #$FF  A9 FF
LOOP:   SBC #1    E9 Ø1
        BNE FC    DØ FC
        RTS       6Ø
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO8:READ D%
30 POKE A+N,D%
40 PRINT"START"
50 SYS40705
60 PRINT"STOP"
100 DATA24,169,255,233,1,208,252,96
```

*Fig. 6.16.* How to carry out a countdown using a byte in the accumulator. (a) Assembly language. (b) BASIC poke program.

bit. This is needed, because if the carry bit happens to be set, then the first SBC action will subtract 2 rather than 1. All the remaining SBC actions will be normal, and it would not make much difference to a time delay like this to have an extra decrement. In some kinds of decrement action, however, like counting bytes, the subtraction of 2 instead of 1 could be a disaster. For this reason, it's a good habit to clear the carry bit before you carry out a subtraction of this sort. The SBC #1 is the decrementing step, and the loop runs pretty much like the X register loop that we looked at earlier. The BASIC poke version is shown in Fig. 6.16(b).

The 6502 does not confine you to decrementing numbers that are stored in the registers, however. The DEC action can be applied to any address in memory, so you can write countdown programs that use as many bytes as you like. The most convenient addresses to use for storing counting bytes are the page zero addresses (from Ø to 255) but, when your 6502 is installed in a Commodore 64, you have to be careful! This is because the C 64 uses a lot of page Ø addresses

(a)

```
        LDA   #$FF
        STA   $6A
LOOP1:  STA   $6B
LOOP2:  STA   $6C
LOOP3:  DEC   $6C
        BNE   LOOP3
        DEC   $6B
        BNE   LOOP2
        DEC   $6A
        BNE   LOOP1
        RTS
```

(b)

```
10 POKE56,159:A=40704
20 FORN=1TO21:READ D%
30 POKE A+N,D%:NEXT
40 PRINT"START"
50 SYS40705
60 PRINT"STOP"
100 DATA169,255,133,150,133,151,133,152,198,152,208
110 DATA252,198,151,208,246,198,150,208,240,96
```

*Fig. 6.17.* A much longer count, using page zero addresses for storage. (a) Assembly language. (b) BASIC poke program.

for its own purposes, and placing bytes in some of these addresses will cause havoc to the operation of the machine. A quick peek at the page ∅ numbers shows that addresses 15∅ to 177 appear to be comparatively unused during simple programs, so perhaps we could make use of these for a really large count. Figure 6.17(a) shows the assembly language version. The three addresses which I have used are $96, $97, and $98 (denary 15∅, 151 and 152). The program starts by loading $FF into the accumulator, and then storing this number in all three page ∅ addresses. Because these are page ∅ addresses, we can make use of page ∅ addressing. The program makes no other use of the accumulator, so that the number $FF will remain in the accumulator throughout. This allows us to store the accumulator in each memory address without reloading the accumulator.

The technique for the countdown is very much the same as before, except that there are three loops. You might try drawing a flowchart for yourself to see how these are arranged. The effect is that we are counting down the number 16,777,215 (denary) to zero. As you might expect, this takes a lot longer than a two-register countdown – several minutes. The BASIC version is in Fig. 6.17(b). If you use a stopwatch to measure the time between the START and STOP messages, and divide this by the number shown above, you'll get some idea of how long (on average) each loop takes. Don't try to count down this number in a BASIC program – life's too short!

Chapter Seven

# Ins and Outs and Roundabouts

## Video loops

Of all the loop programs that we can make use of in machine code, loops that involve the video display addresses are among the most useful. We have seen earlier that we can make things happen on the video display by making use of POKE commands from BASIC. The techniques that you used for POKE displays on the screen can also be used for machine code, and the main differences are that machine code is faster and involves more work on your part!

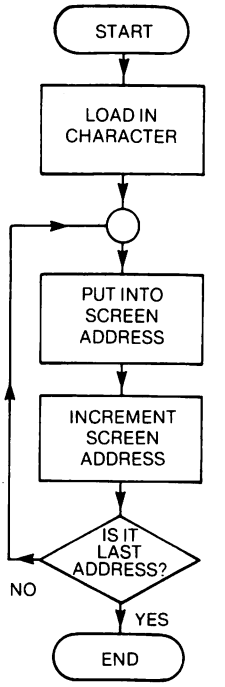



*Fig. 7.1.* A flowchart for filling the screen with a character.

Take a look, for starters, at Fig. 7.1. This shows the flowchart for a program that will fill part of the text screen with one character. The idea is that we load a register (the accumulator is usually the best bet) with a code number for a character, and we store this at the first screen address, which is $4ØØ. We must then increment this address, store the accumulator again, and repeat this process for as long as we need to, until we reach the last address. The flowchart shows what we have to do, but you need to know how to carry it out with the 6502. This is the kind of program which can make use of indexed addressing, so we'll start by recalling what this involves.

There are two index registers, X and Y, each of which can store a single byte number. When we carry out an indexed load or store (or any other operation that copies a byte from memory), the number that is held in the index register is added to the address that we have specified. This forms a new address, and it's this address that is used for the load or store. If, for example, we have 2 stored in the X register, and we specify a store as:

```
STA 1Ø24,X
```

(denary numbers), then this means that the address which will be used is 1Ø24 + 2=1Ø26, and the byte in the accumulator will be stored to address 1Ø26. One of the features that makes this so useful is that we have INX and DEX commands which will, respectively, increment or decrement the number that is held in the X index register. There are corresponding INY and DEY commands for the Y index register.

Let's get down to the assembly language version, which is shown, along with the BASIC program, in Fig. 7.2. The first step is straightforward - load the accumulator with 124. This is the ASCII code number that will produce a half-chequer pattern - obviously, you could try any other code number that you liked to use. We then load the X register with zero. The next item is the loop. We start the loop by storing the byte in the accumulator to the address given by 1Ø24 (denary) plus the byte in the X register, and then incrementing the X register. In assembly language, these steps are written as:

```
STA 1Ø24,X
INX
```

The next step is to compare the content of the X register with zero, using BNE. Since the X register started at zero, and has just been incremented to 1, the comparison will not give zero, and so BNE will cause a loop back to store the character at another address. When

(a)

```
        LDA #124        ;124 denary
        LDX #Ø          ;X starts at zero
LOOP:   STA 1Ø24,X      ;put into 1Ø24+X
        INX             ;increment X
        BNE LOOP        ;until end of count
        RTS             ;back to BASIC
```

(b)

```
10 POKE56,159:A=40704
20 FOR N=1TO11:READ D%
30 POKE A+N,D%:NEXT
35 POKE53281,3
40 SYS40705
100 DATA169,124,162,0,157,0,4,232,208,250,96
```

*Fig. 7.2.* (a) The assembly language program. (b) The BASIC poke program. This version, however, fills only the top quarter of the screen.

the number in the X register is equal to 255 (denary), then the next increment action will make the content of the X register equal to zero again. This is because the register can hold only one byte. At this point, the BNE test fails, and the program breaks out of the loop, and returns to BASIC.

Now this does some of what we want, but not quite all. It will place a character in 256 screen addresses, but the screen consists of a thousand addresses. What has limited us in this case is the size of the X register, one byte. This makes certain that an indexed load or store, carried out in a loop, cannot deal with more than 256 bytes. Working on the basis that half a loaf (or quarter of a screen) is better than nothing, we'll try it out. Later on, we'll see how we can get round this limitation.

Transforming this assembly language by hand into machine code is reasonably straightforward. Once the machine code bytes have been written down (check the displacement byte that follows the BNE), then the BASIC program that pokes the bytes into memory can be written down (Fig. 7.2(b)). It shouldn't take long – apart from the value of N and the DATA line, it's almost the same as any of the other programs so far. We must, however, include the POKE53281,3 step to ensure that the effect of the program will be visible. At the end of the run, we must use the STOP and RESTORE keys to return

the screen conditions to normal. When this runs, there is a short delay while the slow BASIC pokes the numbers into the memory, and then the machine code does its stuff in its usual lightning way.

## The rest of the way

Writing a program that will fill all of the screen is not quite so easy, because of the limitation of the size of the index register. There are several ways of carrying out the action that we need, but the most straightforward method makes use of what is called *indirect addressing*. This was briefly mentioned in Chapter 4, and now that we're up against it, we'll have to take a closer look at one of the two methods that the 6502 can use. The two indirect addressing methods of the 6502 are often known as 'indexed indirect' and 'indirect indexed'. Titles like these are rather confusing so, in this book, I'll stick to the simpler descriptions of X-indirect and Y-indirect. This is because one method makes use of the X index register, and the other method makes use of the Y index register. The one we need for the loop program is the Y-indirect method.

The way that Y-indirect addressing operates is as follows. The first address that we want to use, such as $\$\emptyset 4\emptyset\emptyset$ in our example, is stored as two bytes in two consecutive addresses in memory. As always, the bytes are stored in the low-then-high order, and they must be in page zero memory. A number is also placed in the Y index register. Now when we carry out a memory operation such as a load or a store, we use the Y-indirect form. In assembly language, a Y-indirect store would be written in the form:

```
STA address,Y
```

Its effect is fairly complicated at first sight. The low byte of the address is copied from the memory, and the content of the Y register is added to it. If there is any carry from this addition, it is added to the high byte. The two new bytes of address are then used for the store (in this example) operation. Suppose, for example, that we stored the first screen address of $\$\emptyset 4\emptyset\emptyset$ in zero page memory, using (denary) addresses $15\emptyset$ and 151. Using the low-high order of storing means that address $15\emptyset$ will store $\$\emptyset\emptyset$ and 151 will store $\$\emptyset 4$. Now if we have the number $26 (hex) stored in the Y register, then the effect of:

```
STA (15Ø),Y
```

will be to add $26 to the number stored in 15ø (which was zero), and so form the address $ø426. This is the address to which the byte in the accumulator will be copied.

(a)

```
        LDA   #124
        LDX   #ø
        STX   15ø
        LDX   #4
        STX   151
        LDY   #ø
LOOP:   STA   (15ø),Y
        INY
        BNE   LOOP
        INC   151
        LDX   151
        CPX   #8
        BNE   LOOP
        RTS
```

(b)

```
10 POKE56,159:A=40704
20 FOR N=1TO26:READ D%
30 POKE A+N,D%:NEXT
35 POKE53281,3
40 SYS40705
50 GOTO50
100 DATA169,124,162,0,134,150,162,4,134,151,160,
    0,145,150
110 DATA200,208,251,230,151,166,151,224,8,208,
    243,96
```

*Fig. 7.3.* Filling the entire screen. There's a flaw in this one too, but it does not cause any trouble in this case. (a) Assembly language; (b) BASIC poke version.

Figure 7.3(a) shows the complete assembly language version of the program. The first six steps place the correct values into the registers and into memory. This has not been done in the most efficient way, but at least it's easy to follow. Where things start to get more difficult is step 7, at the start of the loop. What causes the problem is the need to carry out the incrementing of the screen address for a thousand times. We'll take a look at this part of the program in close detail.

At the start of the loop, the number in the Y register is Ø, and the 'base address' for the STA operation is stored in zero page addresses 15Ø and 151 (denary). The address 15Ø stores Ø and address 151 stores $Ø4, so that the two bytes make up the first screen address of $Ø4ØØ, denary 1Ø24. When the STA (15Ø),Y step is used for the first time, then, the byte in the accumulator will be put into address $Ø4ØØ. The INY step then increments the number in the Y register from Ø to 1. When the BNE LOOP step is used, the fact that the Y register contains 1 (not equal to zero) will cause the program to loop back. This looping will cause consecutive addresses of the screen memory to be used. Once again, this continues until the Y register increments from 255 (denary) to zero.

This breaks out of the loop. Since we started address 15Ø with zero, there has been no automatic addition to the number in address 151, and we have to attend to this for ourselves. Now I have taken a short-cut here, and you need to know what it is and why we can get away with it. At the end of the first loop, we increment the number in address 151. This automatically selects the next quarter of the screen if we repeat the first loop. We need to be able to stop the program, however, because we don't want the byte 124 placed in every part of the memory – it would soon zonk out our program and a lot more besides. To stop the action, then, the number in address 151 is copied into the X register, and compared with the number 8. If it's not equal to 8, then the second BNE will return to the first loop to fill up another piece of memory. When the number in 151 reaches 8, the program stops. Figure 7.3(b) shows the BASIC poke program which places the bytes in memory, and runs the machine code.

Now this fills up the screen all right, but you can't always use it. The reason is that the screen addresses actually go from $Ø4ØØ (1Ø24 denary) to $Ø7E7 (2Ø23 denary). We have used $Ø4ØØ to $Ø7FF, rather more. What we have filled between $Ø7E7 and $Ø7FF is the piece of memory that is reserved for 'sprite pointers'. If your program does not use sprites, all is well. If you are going to use sprites, and you need this piece of memory later, you can poke into it later. If, however, you want to store sprite information in this piece of memory, and then run the machine code program, you will need to stop filling the screen after address $Ø7E8. This needs rather more than just the CPX 8 step that we used in this example. You would need a CPX 7 in place of the CPX 8. Following the BNE loop, you would then need LDX 15Ø and CPX $E8 to test the lower byte. This would be followed by another BNE LOOP step. The result of these steps would be to halt the program when the address $Ø7E8 was

reached, saving your sprites from a fate worse than death.

### Take a bigger cast-list ...

We can modify the program of Fig. 7.3 in an interesting way. Suppose we started with the accumulator containing zero, and we incremented the accumulator on each pass through the loop. This would mean that we would produce on the screen each character for the numbers ∅ to 255 (denary). What would happen then? Well, since the accumulator can hold only eight bits, and 255 (denary) is the largest number it can hold, is simply goes back to ∅ again next time it is incremented. Figure 7.4 shows the flowchart for this action,

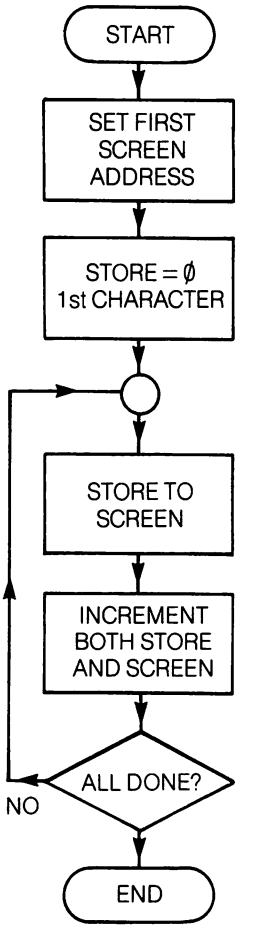



*Fig. 7.4.* A flowchart for printing the entire character set.

(a)

```
        CLC
        LDA  #Ø
        TAX
        STX  15Ø
        LDX  #4
        STX  151
        TAY
LOOP:   STA  (15Ø),Y
        ADC  #1
        INY
        BNE  LOOP
        INC  151
        LDX  151
        CPX  #8
        BNE  LOOP
        RTS
```

(b)

```
10 POKE56,159:A=40704
20 FOR N=1TO27:READ D%
30 POKE A+N,D%:NEXT
35 POKE53281,3
40 SYS40705
50 GOTO50
100 DATA24,169,0,170,134,150,162,4,134,151,168,
    145,150
110 DATA105,1,200,208,249,230,151,166,151,224,
    8,208,241,96
```

*Fig. 7.5.* (a) The assembly language version and (b) the BASIC listing for the flowchart of Fig. 7.4.

and Fig. 7.5 shows the assembly language and the BASIC poke program. There's nothing here that should cause you any great amount of head-scratching, because the only difference between this and the program of Fig. 7.3 involves incrementing the accumulator. As before, we have to clear the carry bit at the start of the program. The step ADC #1 in the loop will then add 1 to the accumulator on each pass through the loop. Try it, and you'll see the full set of text-mode characters appear. This is a good example of how a simple program can be extended so as to do much more than the original version. It's an important point, because a lot of machine code

programming is of this type. If you keep a note of all the assembly language programs that you have ever used, along with what they did, then you'll find that this 'library' is a very precious asset. Very often, you'll find that any new program that you want to write can be done by modifying or combining (or both) old routines that you are familiar with. Another big advantage of this is that an old routine is a trustworthy routine – a split-new one needs a lot of testing before you can rely on it.

Oh yes, before I forget. Have you thought what happens when the accumulator contains 255 (denary) and then has 1 added to it? What will this do to the carry bit? Might we need a CLC somewhere in the loop to counteract the effect? See if you can spot what the effect is, and how to get round it.

## Save it!

At this point, when our programs are getting slightly longer, and doing more interesting things, it's time to look at the topic of saving machine code program on tape. Now you aren't in any way obliged to do this. Our machine code programs have all been, so far, in the form of a BASIC program that poked numbers into the memory. You can, obviously, just save this BASIC program, and it will create the machine code for you whenever you want. When you are using a mixture of BASIC and machine code, this is the ideal way of saving and loading the machine code bytes. There are times, however, when you need as much of the memory as possible, and another piece of BASIC program is as welcome as an elephant in a space capsule. You may also want to put on cassette a program that is a mixture of BASIC and machine code, and to make it difficult for anyone to copy it. Either way, you'll want to make a direct recording of the bytes that are stored in the memory. The ordinary BASIC commands of the C 64 can cope with this but, as we'll see, in a rather roundabout way.

Unlike most other machines, the C 64 has no special BASIC commands for saving or reloading a machine code program. When you use a machine code monitor program written for the C 64, you will usually find that it includes routines for saving and loading machine code. This is fine if you are using the monitor continually, but another method of saving and loading would be useful.

Fortunately, the ordinary BASIC commands of SAVE and LOAD can be pressed into service. When the SAVE command is

used, the C 64 saves a BASIC program. This means that all the bytes, starting from address 2ø49 and going up to the last byte of BASIC will be saved. The first two bytes that are saved, in fact, are always the two which give the end of the BASIC program. Where does the machine obtain these addresses? From its zero page memory. The start of BASIC, as you know by now, is held in addresses 43 and 44. The end of BASIC, which is the same as the start of the variable list table, is held in addresses 45 and 46. If we change the numbers in these addresses, we should be able to control the saving or loading for any other part of the memory.

```
10 POKE56,159:A=40704
20 FOR N=1TO100
30 POKEA+N,N:NEXT
40 POKE43,255:POKE44,158
50 POKE45,101:POKE46,159
60 SAVE"MC"
70 POKE43,1:POKE44,8
80 POKE45,3:POKE46,8
```

*Fig. 7.6.* A program which pokes numbers into memory and then saves them on tape. This illustrates how machine code programs can be saved and reloaded.

Let's try it out. Figure 7.6 shows a BASIC program which pokes numbers into memory, and then saves them. We could, of course, have used genuine machine code, but the sequence of numbers is easier to recognise. As usual, memory is reserved, so that the hundred numbers are not in danger of being changed by the action of the machine. Lines 4ø and 5ø then carry out the pokes that are needed to reset the page ø numbers. In place of the normal start-of-BASIC number that we place in 43 and 44, we put an address which is two bytes below the start of our set of numbers. The subtraction of 2 is important, because if we use the true starting address, the operating system will replace the first two numbers by two bytes of an address, the end-of-program address. We then poke the last address of our program into addresses 45 and 46. Following that, we can use an ordinary SAVE command. This gives the usual tape message, and lines 7ø and 8ø then restore the normal numbers into the page ø locations.

Now we have to prove that it works. RUN the program, with a spare blank cassette in the recorder. Carry out the usual PRESS RECORD AND PLAY action when requested, and let the program finish. Now rewind the tape, which should contain all the

numbers that were poked into memory. Now switch off the C 64. When you switch on again, all the addresses will be reset, and the numbers will be lost. To replay the tape, you need to reset the page zero numbers again, LOAD, and then restore the page ∅ numbers. You can't carry out all this in a program, because the normal LOAD action does not operate if there are program lines following a LOAD.

To recover the program, then, first reserve the memory by typing POKE56, 159 (RETURN). Then change the numbers in addresses 43 and 44, using:

```
POKE43,255:POKE44,158
```

as before. Now LOAD"MC" to get the bytes in place. After loading is complete, type:

```
POKE43,1:POKE44,8:POKE45,3:POKE46,8
```

This will restore the normal numbers in the zero page addresses. You can then check that the numbers have been replayed. To do so, type:

```
FORN=1TO1∅∅:?PEEK (4∅7∅4+N);"  ";:NEXT
```

and press RETURN. You should see the numbers listing on the screen. This proves that a machine code program can be saved and loaded by using only the ordinary BASIC commands.

Don't imagine, by the way, that this applies only to machine code programs. Any section of the memory can be saved and reloaded by using this method. Since the screen is controlled by the bytes in memory from 1∅24 to 2∅23 (denary), you can SAVE and LOAD screen patterns in this way. The patterns are rather spoiled by the messages that appear during loading, though, so for best results, you should disconnect the screen display from the keyboard output. You can do this with POKE154,4. The screen can be reconnected with POKE154,3. While the screen is disconnected, you will see nothing appearing when you type POKE154,3. The effect will be obvious, however, when you press RETURN and then make any other use of the keyboard, such as listing a program.

### Take a message ...

After that brief interlude on the subject of saving and reloading machine code, let's get back to the programs. We left Fig. 7.5, you remember, writing characters on the screen. It's time now that we

looked at ways of putting something more interesting on the screen, and letters look like a reasonably simple start to this type of programming. What do we have to do? Well, to start with, we need to store some ASCII codes for letters somewhere in the memory; we can't just use a string variable as we would in BASIC. We will have to know the address at which the first of the letters is stored, and how many letters are stored starting at this address. After that, we should be able to work a loop which takes a byte from the 'text space' (where the letter codes are stored) and put it into the screen space (the screen addresses). We've already used the main type of instruction that we need for this sort of thing – the auto-incremented load or save. To work, then.

We start, as always, with a flowchart. It's not so easy this time, because we need a different way of ending the loop. We could count the number of letters that we want to place on the screen, but I want to look at a different technique this time – using a *terminator*. You are probably familiar with this idea used in BASIC programs. A 'terminator' is a byte which the program can recognise as a special character – one which is not, for example, part of a message. A convenient terminator for a lot of purposes is $\emptyset$, so we'll try that. The difficulty arises because we don't want this terminator printed on the screen. Because of the way that the C 64 uses its code numbers, the number $\emptyset$ actually would give us a printed character – the @. We don't want this to appear, so we must test the accumulator between loading the byte from memory and placing it into the screen memory. That, as you'll see, makes the loops more complicated.

The flowchart that we need is shown in Fig. 7.7. What we have to do is to store two addresses. One of these will be familiar, it will be part of the screen memory. This has to be the address of the first byte that we will want to put on the screen. The other address has to be a store address, the start of a string of bytes that will be used to store ASCII codes, and which will not be used for anything else. We can clear things up for ourselves by giving these two addresses label names. I've chosen SCRN for the screen memory, and TXT for where we're storing the codes. What we do, then, having allocated these addresses, is to load a code from TXT, and increment the TXT address. We then test the character, to see if it's our terminator of $\emptyset$. If it is, we want to leave the program at once. If it's not, then we store this byte at SCRN, increment the SCRN address, and go back for another character. Now this gives us a flowchart which has two jumps. One of these is the 'go back' part, the other is the part that goes to the end. What is this going to look like in assembly language?

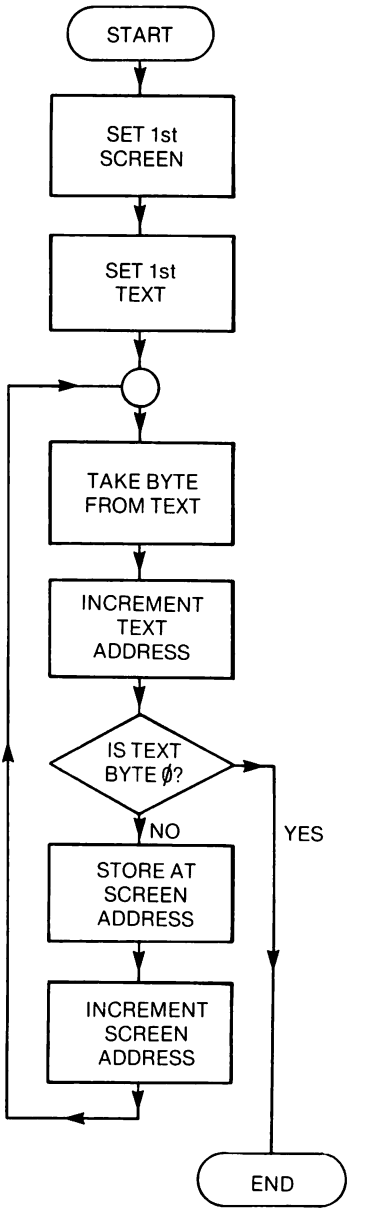



*Fig. 7.7.* A flowchart for printing a message on the screen.

The answer appears in Fig.7.8. It follows the flowchart pretty exactly, and the parts we particularly need to look at are the BEQ and the BNE steps. At the BEQ step, the accumulator has been loaded from the TXT piece of memory, whose starting address is

(a)

```
        LDX #Ø
LOOP:   LDA TXT,X ;get character
        CMP #Ø    ;is it Ø?
        BEQ OUT   ;end if so
        STA SCRN,X;put on screen
        INX       ;increment index
        BNE LOOP  ;round again
OUT:    RTS       ;back to BASIC
```

(b)

```
10 POKE56,159:A=40704:B=40863
20 FOR N=1 TO 16:READ D%
30 POKE A+N,D%:NEXT
40 REM POKE LETTER CODES
50 FORJ=1TO13:READ D%:POKE B+J,D%:NEXT
60 PRINT"J"
70 POKE53281,3
80 SYS40705
100 DATA162,0,189,160,159,201,0,240,6
110 DATA157,224,5,232,208,243,96
120 DATA3,15,13,13,15,4,15,18,5,32,54,52,0
```

*Fig. 7.8.* (a) The assembly language for the message routine, and (b) the BASIC listing.

$9FAØ. This translates into denary as the two bytes 16Ø,159; this is the denary address 4Ø864. I have picked an address which is well clear of the addresses that we are using for the program. The accumulator is loaded using X indexed addressing, and the X register has been loaded with zero. As a result, the first load of the accumulator will come from the address 4Ø864. This will be the byte 3, which is the C 64 internal code number for the letter 'C' when we are using this method of placing text on the screen. The CMP Ø step is put following the load so that the Ø byte can be detected at the end of the message. BEQ means 'branch if equal to zero', and the displacement that follows this instruction byte will take the program to the RTS instruction, skipping over the steps between the BEQ and the RTS. If the byte is not zero, however, it is stored at the SCRN address, using the X register once again for indexing. We then have to get back for another character. Since this needs a jump, and one that must always be made at this point, we could use the

JMP instruction. JMP, however, has to be followed by a full two-byte address, and it's a lot easier to use BNE. At this point in the program, the byte in the X register can never be zero (unless you're trying to place too many letters on the screen). As a result, BNE will always return the program to the LOOP position. This will continue until a zero is loaded into the accumulator, and the BEQ OUT step forces the program to return to BASIC.

That explained, we can convert into the form of a BASIC program, and try it out. We'll place the bytes into memory in a fairly simple way, by poking them into memory from a DATA line. This is done in line 5Ø. Lines 2Ø to 3Ø have previously put the machine code into place, and line 8Ø runs it. When it runs, you see the message appearing.

Perhaps we can deal now with one annoying point about placing text on the screen. In all of the preceding programs that use text on the screen, we have had to poke the colour background to make the characters visible. In a looping program of this sort, we can do a lot better. What we shall do is to place the number 1 into each letter position in the colour addresses. If you remember your C 64 BASIC, you will recall that adding 54272 denary (which is $D4ØØ) to a text memory address gives the corresponding colour address. By poking a number between 1 and 15 into this address, we can get different 'foreground' colours of characters.

This sort of thing needs only a comparatively small change to a machine code program. Figure 7.9 shows the assembly language version. Following the STA SCRN,X step, which places the character code into text memory, we have a LDA #1 and STA

```
        LDX  #Ø
LOOP:   LDA  TXT,X
        CMP  #Ø
        BEQ  OUT
        STA  SCRN,X
        LDA  #1
        STA  COLR,X
        INX
        BNE  LOOP
OUT:    RTS
```

*Fig. 7.9.* Modifying the routine so that the colour memory is poked. This ensures that the text can be seen without having to change the background colour.

COLR,X pair of steps. These place the number 1 into the correct colour memory addresses, provided we get the base address correct. The base address for text is $Ø5EØ, and adding $D4ØØ to this gives $D9EØ. In denary, it's 55776. By using this method of placing characters on the screen we avoid having to poke the address 53281, and we also avoid having to press the STOP and RESTORE keys at the end of the program. Another small step forward!

## Sailing out of the port

When I described the action of the computer system in Chapter 1, the idea of a 'port' was raised. As far as a computer is concerned, a port is any chip or collection of chips that carries out the actions of sending bytes out or taking bytes in. As it happens, the port arrangements of the C 64 are rather complicated, and they are organised in a rather complicated way. Fortunately, until we start to make use of really advanced graphics, or unless we want to make non-standard cassette recordings, we don't have to take control of the ports. The sound system and the special video effects, such as sprites, are dealt with by special chips.

As it happens, carrying out these types of action in machine code is particularly simple, because to do so we don't have to understand the port system. The C 64 uses POKE commands in BASIC for its sound instructions. That means that we can use identical methods in machine code, but using LDA (immediate) to get a byte and STA to place it into memory. Wherever anything can be done in BASIC by using POKE, we can use this LDA STA combination in machine code. We very seldom have to bother, though. That's because we can do all that's needed from BASIC. There's no point in using machine code for sound, because the speed of machine code is no advantage – you will have to use a delay loop in any case to get the correct duration of sound. It's only where you have to achieve video results that are impossible with normal sprite graphics that you really need to get to grips with the video interface chip of the C 64. That's definitely not a job for the beginner to machine code, and we'll leave it strictly alone!

# Chapter Eight
# Debugging, Checking and MIKRO

## Debugging delights

Now that you have experienced some of the delights of machine code programming, it seems fair to mention some of the drawbacks. One of these is debugging. A 'bug' is a fault in a program, and debugging is the process of finding it and eliminating it. It all sounds rather insecticidal, but it's nothing like as easy as that!

It's easy to say, I know, but the first part is prevention. Check your flowchart carefully to make sure that it really describes what you want to do. When you are satisfied with the flowchart, turn to the assembly language to make sure that it will carry out the instructions of the flowchart. When you are happy with this, then check that the bytes you intend to poke into memory are the bytes which correspond to the assembly language instructions. One thing to watch very carefully is that you have the correct code for the addressing method that you are using. If you check each stage in the development of a program in this way, you will eliminate a lot of bugs before they are up and flying. Don't feel that you are a failure if the program still doesn't run – unless a machine code program is very simple, there's a very good chance that there will be a bug in it somewhere. It happens to all of us – and it's only by experience that you can get to the stage where the bugs will be few in number and easy to find.

If you use an assembler, one source of bugs completely disappears. Human frailty means that the process of converting assembly language instructions into machine code bytes is error-prone. That's because it means looking up tables, and anything which involves looking from one piece of paper to another is highly likely to introduce mistakes. I shall briefly describe the action of the MIKRO assembler later in this chapter. At the time of writing, there were several assemblers available for the C 64, but MIKRO has

several advantages, one of which is that it can be obtained in one cartridge along with a monitor program called TIM. There's more on monitors later in this chapter as well! If machine code has really caught your imagination, and you feel that you want to branch out into more advanced work than we have space for in this book, then a good assembler and monitor program are essential. You will have to be prepared to pay quite a lot for such a program. If, however, you intend to be just a dabbler, spawning the odd drop of machine code now and again, then the poke-to-memory methods that we have used so far will be perfectly adequate.

Using these methods, however, means that there will be bugs lurking in each corner of the code. The main cause of these bugs is weariness. Converting an assembly language program into hex bytes, and writing them in the form of DATA lines for a BASIC poke program is a tedious job, and all tedious jobs result in mistakes (ever driven a 'Friday car'?). Faulty address methods are one common result of tedium, and simply writing down the wrong code is another. One very potent source of trouble is with branch displacements. You may get the number wrong somewhere between subtracting addresses and converting a number (particularly a negative number) to hex. Another problem arises when you modify a program, and add code between a jump instruction and its destination. Having done that, you then forget to alter the size of the displacement byte! This is a problem which simply doesn't arise when an assembler is used. An incorrect jump will nearly always cause the computer to lock up. You can often restore control with the use of the STOP and RESTORE keys, but not always, and you will sometimes lose your program (you did record it, didn't you?). Another form of incorrect branch is doing the opposite of what you intended, like using BEQ in place of BNE or the other way round. Careful thought about what the jump will do for different sizes of bytes should eliminate this one.

A lot of problems, as I have already said, can be eliminated by meticulous checking, and it pays to be extra careful about branch displacements, and about the initial contents of registers. A very common fault is to make use of registers as if they contained zero at the start of the program. You can never be certain of this. It's safer, in fact, to assume that each register will contain a value that will drive the computer bananas if it is used. With all that said, and with all the effort and goodwill in the world, though, what do you do if the program still won't run?

There's no single, simple, answer. It may be that your flowchart

doesn't do what you expect it to do, and if you didn't draw a flowchart, then you have got what you deserve. It may be that you are trying to make use of a C 64 ROM routine and it doesn't operate in the way that you expect. When you have enough experience in machine code to follow more elaborate programs, you can make use of a disassembly of the ROM. At the time of writing, there is a very useful book called *Inside the Commodore 64* by Milton Bathurst, published by DataCap in Belgium. It's available from major booksellers. This book is a complete listing, with comments, on the ROM of the Commodore 64. There is also much on the use of the RAM, in particular the page zero addresses. Once again, this is for the really serious C 64 machine code programmer. All I can do here is to give you general guidance on removing the bugs from a program that seems to be well-constructed but which simply doesn't work according to plan.

The first golden rule is never to try out anything new in the middle of a large program. Ideally, your machine code program will be made up from subroutines on tape, each of which you have thoroughly tested before you assembled them into a long program. In real life, this is not so easy, particularly when the subroutines exist only as DATA lines for BASIC poke programs. As usual, users of an assembler have the best of it, because they can keep assembly language instructions stored like BASIC programs, and merge and edit them as they choose.

The next best thing to keeping a subroutine library on tape is to have extensive notes about subroutines. In addition to routines of your own, you can keep notes on routines which you have seen in magazines. *Personal Computer World* runs a series called *SUBSET* (and I wish they would reprint it as a book!). This consists of several general-purpose machine code routines each month. Most of these are for the two most-used microprocessors, the Z80 and the 6502. Even if you don't use the routines, the way in which they are documented should give you some ideas about how you should keep a record of your own routines - I personally would buy the magazine for this feature alone! If you are going to use a new routine in a program, it makes sense to try it on its own first so that you can be sure of what has to be placed in each register before the routine is called, and what will be in the registers after. Look at the examples in *SUBSET*, and see how well this information is presented.

Planning of this type should eliminate a lot of bugs, but if you are still faced with a program that doesn't work, and which you don't want to have to pull apart, then you will have to use *breakpoints*. A

breakpoint, as far as the C 64 operating system is concerned, is the byte $6Ø. This is the RTS byte, and its effect is to return to BASIC. When you are back in BASIC, you can examine the contents of memory by using PEEK instructions. The principle is to pick a point in the program at which something is put into memory. If you place a $6Ø byte following this, then when the program runs, it will return to BASIC immediately after the memory is used. By using a PEEK, you can then check that what has been loaded into the memory is what you expect. If it isn't, you should know where to look for the fault. If all is well at this point, then substitute the original byte that belongs in place of the $6Ø, and place the $6Ø at the next address following a memory store command. This type of action, however, is much more easily carried out with the help of a good monitor program, of which more later.

The most awkward fault to find by this or any other method is a faulty loop. A faulty loop always causes the computer to lock up. Though the STOP and RESTORE keys will usually get you out of trouble, this will not always be the case. For example, it's possible for a program that runs wild to alter one of the bytes that controls the use of the keyboard, so that you find you can't use the keys even if you regain control! The action of the STOP and RESTORE keys, for example, can be disabled by poking to address 8Ø8 (denary). The main cause of this sort of thing is a loop back to the wrong position. For example, if we had a program, part of which read:

```
        LDX,#$FF
LOOP:   DEX
        BNE LOOP
```

we could encounter problems. Suppose that this was assembled by hand, and we made the branch back to the LDX instruction rather than to the DEX instruction. This would result in the X register being kept 'topped up', and never decremented to zero, so that the loop would be endless. A mistake like this is easily spotted in assembly language, because the position of the label name is easy to check. It is very much more difficult to find when you have only the machine code bytes to look at. As always, taking care over loops is the only answer, and the method that has been shown in this book, of calculating and checking displacements, is a good precaution.

## The TIM monitor

I mentioned monitors briefly earlier on in this chapter. This has nothing to do with a TV monitor, which is a sort of superior quality TV display for signals. A monitor in the software sense is a program, one which checks (or monitors) each action of a machine code program. A monitor is (or should be!) a machine code program which can be put into the memory at a set of addresses that you aren't likely to use for anything else. Once there, a monitor allows you to display the contents of any section of memory (in hex), alter the contents of any part of RAM, and inspect or alter the register contents of the 6502. These are the most elementary monitor actions, and it's useful if a section of program can be run, breakpoints inserted, and registers inspected on a working program. The ideal monitor would be one which could carry out the steps of a machine code program one at a time, displaying the register and memory contents at each step. Such a monitor was available for the old TRS-80 Mk.1, and would be a very welcome item for serious machine code programmers on other machines.

## MIKRO monitor

The MIKRO assembler/monitor cartridge contains, among its many facilities, an excellent monitor which is modelled on the TIM monitor which was available for the older 'PET' computers. The C 64 has to be switched off while the cartridge is inserted but, once in place, the cartridge can be left in until you need to use another type of cartridge. Simply placing the cartridge in position does not cause the monitor to run – you have to call it up by typing TIM (then RETURN). This causes the TIM display to appear on the screen (Fig. 8.1). This shows the TIM starting address, and underneath it the contents of the main registers of the 6502 along with important

```
CALL @   814B
  ADDR  IRQ  SR  AC  XR  YR  SP
 .;814B  EA31 4D  ØØ  ØØ  ØØ  FB
```

*Fig. 8.1.* The TIM monitor message on the screen. This will appear each time you start the monitor by typing TIM and pressing RETURN.

addresses. If we ignore the addresses for the moment (at this stage in learning to use machine code they are not important), we can look at the register bytes. These are the five main register contents of Status Register (shown as SR), Accumulator, X index, Y index, and Stack Pointer. If you have just switched on the C 64, the accumulator and the index register will contain zero. The status register normally has the value of 4D (several flags including the carry flag are set), and the SP register is not at its starting value of $FF.

All this may not be of much use to you right now, but as we move into more advanced machine code work, you'll see that a register display like this can be very useful indeed. You can call up this register display whenever you want by typing the letter R (then RETURN). All the actions of the monitor can be called up using the combination of a full-stop and a letter, such as .R. While the monitor is waiting for another command, however, it places a full-stop on the screen so that you only have to press the letter of your choice, and then press the RETURN key. Figure 8.2 shows what options are available from these single-letter commands.

Now you won't necessarily want to use all of these options - some of them might never be of particular interest to you. Instead of dealing with them in order, then, we'll pick a few that are the most

---

M - Display contents of memory in hex. M has to be followed by a start address and an end address, in four-digit hex.

S - Save a machine code program on tape or disk.

L - Load a machine code program from tape or disk.

G - Execute a program. G has to be followed by the starting address of the program, in four-digit hex.

H - Hunt through memory for a group of bytes. H has to be followed by a start address, an end address, and the bytes that are to be found, all in hex.

T - Transfer a block. T must be followed by the start address of the block, the end address, and the new starting address.

D - Disassemble code. The starting address and the end address for disassembly must follow. The display fills the screen and remains until a key is pressed to display the next piece of the disassembly.

X - Return to BASIC.

---

*Fig. 8.2.* The TIM menu for the MIKRO package - typical functions of a monitor.

useful to the beginner in machine code. Of these the '.M' command (Memory inspection) is the most important. When you press .M (usually just M for the reason above), you should then type a space. This has to be followed by the starting memory address that you are interested in. This must be typed as a four-digit hex number. For example, the address 2B must be typed as ØØ2B. You then have to type another space, and then the ending address of the section of memory you want to investigate. Again, this must be in the form of a four-digit hex number. Nothing happens until you press RETURN, so that you have time to change your mind. When you press RETURN, you will see the screen display the section of memory which you have requested. The column at the left-hand side contains a full stop, a colon, and then the starting address for each row of numbers, and there are eight numbers in each row. The numbers that are stored in the addresses are then shown in these rows. Suppose, for example, that we look at the row whose starting number (on the left-hand side) is ØØ4Ø. The first byte in this row is the byte which is stored at $ØØ4Ø, the next byte is the one stored at $ØØ41, and so on.

While TIM is displaying contents of memory in this way, you can alter any of the bytes in any of the displayed addresses. This is done in exactly the same way as you edit a BASIC program. You place the cursor over the first digit of the byte, using the arrowed cursor control keys. You then type the digit that you want, and follow up by typing the second digit of the byte. Of course, if you want only to alter one digit, you can place the cursor over it and type one digit. This must be a valid hex digit. When you press RETURN, the value will be placed in the memory. If you typed a letter which does not make sense, no change is made. One particularly useful feature of this ability to alter code is that you can make any byte in your code equal to $ØØ. This is the BRK (Break) command, and its effect will be to return you to the TIM monitor. This is not the same as RTS, which normally returns you to BASIC. When you use BRK to return to the monitor, you can then make use of the .R command to inspect the registers of the 6502, and the .M command to see what has happened in the memory. Placing the $ØØ code in a machine code program is called 'setting a breakpoint', and it's a particularly useful way of finding out what a program is doing at some particular point in its path.

The next most important instruction, then, as far as we are concerned at the moment, is '.R', as we mentioned earlier. Pressing the R key causes the register contents of the 6502 to be displayed

when you press RETURN. Unless the machine has been interrupted in the middle of a program, there won't be much to look at here, but when you come to make use of the more advanced features of TIM, like the use of breakpoints, it will be very useful. The contents of all the registers are shown, and normally we'll be looking at the contents of the A,B,X and Y registers in particular. What this means is that when you run your machine code program with the breakpoint inserted, the program will stop at the breakpoint, and the screen will show the contents of the registers, when you use the '.R' command. This is very often all that you need to spot where a program has gone wrong. TIM allows you to set up as many breakpoints as you like, though you can only have a breakpoint in place of an operating code. If, for example, you have a piece of machine code for the instruction LDA $4Ø5Ø, then this consists of the opcode $AD followed by the address bytes $5Ø and $4Ø. If you want to break here, you can only replace the opcode $AD by $ØØ, not the address bytes. Only an opcode represents an instruction to the 6502. When the program stops at a breakpoint, you can inspect the contents of registers, use .M to examine memory, and then allow the program to continue by pressing '.G'. You can even alter the contents of the registers before you start the program again, but this is not the sort of thing you want to try until you have rather more experience with machine code. It's useful, though, if you are checking the action of a loop, and you want to see what happens when the content of a register reaches some value like $FF or $ØØ. Instead of going round the loop dozens of times, you can go round once to check that the loop is working, and then alter the register contents so as to make the loop stop – and then check that it does. Altering the 6502 registers is done after you have used .R to display the register contents. All you need to do is to place the cursor over a digit, and then type the value that you want, and press RETURN. This value will then be placed into the register when the program is resumed. The .G has to be followed by a space, then a four-digit address for the next instruction that you want to carry out. Wherever you have used breakpoints, you should clear them after you have finished investigating, by using .M, followed by editing.

These are very powerful methods of debugging and sorting out a program, and yet they form only part of the facilities that this very useful monitor offers. The lack of SAVE and LOAD facilities for machine code programs, for example, is remedied in TIM. The .S command will save a machine code program, for which you have to specify a filename, cassette or disk storage, the starting address for

the code, and the end address plus 1. All numbers must be in hex, using four digits for addresses and two for single bytes. For example, suppose you want to save on cassette a program which started at $9F6Ø and ended at $9FE2. The SAVE command would be typed as:

.S"MCPROG",Ø1,9F6Ø,9FE3

Of this lot, the .S is the save command, and MCPROG is a filename. The Ø1 specifies that we are using cassette – Ø8 would mean that you wanted to save on disk. You must use Ø1 or Ø8, not 1 or 8. The start address then follows, and the end address has 1 added to it. Commas are used to separate the numbers. Reloading a program from cassette is much easier – all you need to type is:

.L"MCPROG"

or even .L if you are content to load the first program on the tape.

Last, but quite certainly not least, the TIM monitor allows you to search for bytes, transfer bytes, and disassemble memory. Suppose you want to see if the byte 2D occurs between 9FØØ and 9FFF. If you type .H 9FØØ 9FFF 2D, and then press RETURN, the screen will display each address in the range where this byte occurs. You can look for groups of bytes also, which is often more useful. For example, if you are looking in the ROM for a load from the address 7A (code 85 7A), and you think that this might be in the region between $AØØØ and $AFFF, then .H AØØØ AFFF 85 7A will find the ten addresss at which this combination of codes appears. If you want to place a piece of ROM code into your own program without having to copy it byte by byte, you can use the .T command. The .T has to be followed by the starting address of the bytes you want to copy, then the finishing address, then the new starting address. If, for example, you want to place the code that lies between $AB7B and $ABA4 into memory at $9FØØ, then the command you need is:

.T AB7B ABA4 9FØØ

Finally, the .D command allows you to disassemble the memory. This is particularly useful for ROM, if you have no printed disassembly. The .D command has to be followed by a starting and a finishing address, both in four-digit hex. The result of pressing RETURN is a disassembly display. This consists of 25 lines of print on the screen (unless you have requested only a small amount of code). Each line contains a reference number, an address (of the opcode byte), the code in hex, and its assembly language version.

Some parts of the ROM contain nothing but data bytes, with no opcodes, and if these are not numbers which could be taken as opcodes, they appear as BYT instructions. If the disassembly extends to more than 25 lines, pressing any key will display another 25 lines, until all of the requested memory has been disassembled. The use of the disassembler is particularly useful if you want to make use of ROM routines.

When you have made use of the TIM monitor to your satisfaction, you can then press the X key to return to BASIC. All in all, it's a most satisfactory piece of software, and all the more remarkable because it's only part of a complete package which features the MIKRO assembler – and that's what we are going to deal with next.

## Using the MIKRO assembler

At the time when this book was being written, there were several assembler programs for the C 64, but the MIKRO assembler was by a long way the most useful product in my estimation. Even though this is an introductory book for readers who may never go as far in machine code as to use an assembler, a description is necessary. Dispensing with a description of this assembler would be like writing a history of aviation and omitting the names of Alcock and Brown – even though not many readers would have direct experience of their flight.

Any assembler worthy of the name will be written in machine code. MIKRO goes one step further by being in cartridge form. This is a great advantage, because an assembler that has to be loaded from tape can be a nuisance. The reason is that inevitably when you are developing a machine code program with an assembler, the program will run away from you at some stage in testing. When it does so, it usually manages to corrupt memory (change stored values), and it will be as likely to corrupt the assembler as anything else if the assembler is in RAM. With the assembler in cartridge ROM, its code is safe, and you can return to it at any time.

All assemblers are different, and it's likely that my description of how to proceed with MIKRO will not exactly match the action of any other assembler. The principles, however, are the same, and it's a matter of learning a different sequence of commands. The differences between assemblers are rather like the differences between computers – but if you know the language, the differences become less important. The language in this case is 6502 assembly

language. What you have to learn is how you type a program in such a form that the MIKRO assembler will deal with it, process it into machine code, and store the code so that you can run the program.

## Driving the MIKRO

Before you can get to grips with MIKRO, you need to know how it copes with the problem of assembling your instructions. The principle is that you type your assembly language program in numbered lines, just as you would write a BASIC program. This is not coincidental, because your program can be written even if the MIKRO cartridge is not installed! You are simply using the facilities of the C 64 to write lines of instructions. Provided you don't call on the C 64 to run them, you don't need the MIKRO present – not until you assemble the program, that is. Your lines of assembly language can be saved on tape in exactly the same way as you save any BASIC program. What distinguishes this program from BASIC is that it uses assembly language, and that it contains directions to the assembler program. It is, however, an advantage to have the MIKRO assembler cartridge present when you type the program. One good reason is that you can type AUTO (then RETURN), to get auto line numbering. When you type AUTO and press RETURN, the number 1ØØ appears on the screen. This is the starting line number. Each time you press RETURN after having typed a line of BASIC or of assembly language, a new line number will appear. The numbers rise in tens in the conventional way, so that you will see the sequence 1ØØ, 11Ø, 12Ø ... and so on. This is such a useful facility that I always have the MIKRO cartridge installed when I use BASIC.

The AUTO command can be modified to change the start number and the increment number. For example, typing AUTO 1Ø,5 will give a starting number of 1Ø, going up by 5 each line. You can break out of the automatic sequence of line numbers by pressing the RETURN key.

Another very useful editing command that MIKRO offers is the DELETE command, sadly omitted from the C 64. DELETE can be followed by two line numbers, separated by a dash. Its effect will be to delete these line numbers and all the line numbers between them. For example, DELETE 1ØØ-2ØØ will delete 1ØØ, 11Ø, 12Ø ... all the way up to 2ØØ. You can also omit one of the numbers in the command, so that DELETE 2ØØ – will delete line 2ØØ and all the line

numbers higher than 2ØØ, and DELETE -3ØØ will delete all line numbers up to and including 3ØØ.

In addition to these editing commands which are equally useful when you are writing BASIC programs, MIKRO has two commands which are specifically intended for assembly language programming. FORMAT will cause any listing to be organised into neat columns, showing the address, label, opcode, operand and remarks separated, no matter how you typed them in the first place. This is particularly useful for assembly language programs, which can be hard to check unless they are neatly set out. The other very useful command is FIND. FIND has to be followed by a name, such as a label name, and it will list on the screen (or printer) each line in which this label name occurs. FIND will also work with a BASIC program, and it's very useful for tracing where you have used variable names. For example, FIND X will list each line in which X has been used. The line is not printed in the usual BASIC way, but rather in the style of an assembly language program. You can restrict the action of FIND to a set of lines by adding line numbers, separated by a dash, following a comma. For example, FIND LOOP, 15Ø-3ØØ would list each line between 15Ø and 3ØØ which contained the word LOOP. You can use the same variations as DELETE and LIST, so that FIND LOOP, 4ØØ- and FIND LOOP,-2ØØØ can be used.

Two other commands are also useful. The NUMBER command will carry out number conversions for you. For any number (in the normal range) following NUMBER, you will get a listing of the same number in hex, denary, octal (scale of eight, not used much nowadays) and binary. The different number types are identified by using the $ prefix for hex, @ for octal, and % for binary. If, for example, you type NUMBER123, you will see the display:

```
      HEX= $ØØ7B
  DECIMAL=  123
    OCTAL= @ØØØ1Ø5
   BINARY= %ØØØØØØØØØ1ØØØ1Ø1
```

You can type your number in any of these scales, so that NUMBER$45 or NUMBER@215312 or NUMBER%11Ø111Ø111 would all result in a display of the number in all its forms. The command DISASSEMBLE is also available, giving the same action as we noted for the .D command in the TIM monitor.

Now the assembly language that is used by MIKRO is fairly close to the standard that Mostek, the manufacturers of the 6502, have

laid down. It isn't exactly identical, but the differences are very small. One that you'll notice is that remarks are separated by an exclamation mark rather than a semicolon. We'll look at some of the differences here, but others are of interest only when you have had much more experience with machine code. The other way in which this program will differ from BASIC will be in the instructions to the assembler. It isn't enough to provide a set of assembly language instructions. You must, for example, specify in what addresses in memory you want the code to be assembled. As so often happens, though, you'll find that a standard set of these instructions will suffice for practically all of your uses. The MIKRO assembler will not place code in the addresses that we have been using for examples up till now. This is because it uses these addresses for its own purposes. 4K of memory, starting at address SCØØØ, can be used for your own routines. Alternatively, you can assemble your code at the address which is specified by poking 127 into address 56. This is the address $7FØØ, and to start assembly of code at the next address of $7FØ1, you would start your assembly language lines with:

```
1ØØ *=$7FØ1
```

We would normally start in this way by allocating memory. You still have to be sure that you have reserved this memory. This can be done either by using POKE56,127 in BASIC, or by a corresponding LDA #127 and STA 56 at the start of the machine code. If you are writing a program that is purely machine code, with no BASIC present, then all of the RAM is yours. In such a case, it makes sense to start placing code at the normal start-of-BASIC address. In this way, you can use the ordinary LOAD and SAVE commands if you want to. If you don't use the *=$7FØ1 or similar starting command, then your code will always be assembled starting at address $Ø33C. This is a piece of the RAM which is called the 'cassette buffer'. It is used only during LOAD and STORE operations, and it extends to $Ø3FC. It's not exactly an ideal place to put code, however, if you are going to use LOAD and STORE!

Following the starting address allocation, you can then allocate label names. These can be of addresses or of bytes. For example, you might have:

```
11Ø SCRNST=$Ø4ØØ
12Ø CARRET=$ØD
```

as the next two lines. These do not cause any code to be generated when you assemble. What they do is to ensure that wherever these

label words appear, the numbers are put into their place. Wherever the word SCRNST (Screenstart) appears, the number $Ø4ØØ will be placed. Wherever the word CARRET (Carriage Return) appears, the code $ØD will be placed. Using labels in this way makes the program much easier to follow and defining these words early in the program ensures that you know what they mean without having to search through the program. The label names should be of six letters or less. If you omit to define any label name, the assembly cannot proceed, and you will get an error message when you try to assemble the program. A label does not necessarily have to be defined in this way; it can be defined in the program as, for example, by:

```
22Ø LOOP:LDA #124
```

We write our assembly language in lines, then, as we might write BASIC, but with only one instruction in each line. Ending with the RTS instruction ensures that the program will return to BASIC, and the lines of the assembly language program can even be followed by lines of ordinary BASIC. You must, however, place an END line at the end of the assembly language section. It's not such a simple matter to assemble and then automatically go on to a BASIC program. If the END statement is not present, then MIKRO will try to read the lines of BASIC, and this will result in an error report. The main difference, in fact, is that the MIKRO assembler is rather more fussy about how you type assembly language. You must, for example, leave at least one space between each 'field' of assembly language. 'Field' in this sense means section, and the sections are label, opcode, operand and comments. You can't, for example, type:

```
LOOPLDA#$45!START IT
```

and hope that the assembler can cope in the same way as the BASIC of the C 64 can usually cope with commands run together in this way. What you normally find when you assemble is the error message: NO OPCODE, meaning that the assembler is unable to separate the parts of the command. If you type it as:

```
LOOP LDA #$45 !START IT
```

separating the fields, the assembler will be able to convert this perfectly into code for you.

Now as we go along, we'll look at refinements and useful additions to the stock of commands, but these are the fundamentals that you need to have firmly in your grasp in order to start making intelligent use of MIKRO. If each program you write starts with a *= to

allocate memory, and then defines as many labels as possible, then you are off to a flying start. Following these preliminaries, you can type your lines of assembly language. If you use a number with none of the distinguishing symbols in front of it (such as $), it will be taken as a denary number. Using $ means a hex number, and we have met the prefixes @ and % earlier in connection with NUMBER. The asterisk, *, is always taken to mean the 'current address'. This is the address of the start of the instruction in which the asterisk appears, so the asterisk is a useful way of indicating an address without actually having to know what it is! The exclamation mark is used to indicate a line which is a comment, like the use of REM in BASIC.

Any assembler also permits what are called 'assembler directives', or 'pseudo-ops', which are instructions to the assembler program itself. MIKRO is no exception, and some of these directives are worth looking at here because they (or versions of them) are used widely on assemblers for other microprocessors. We've already looked at *, which indicates the current address. Three more such pseudo-ops are WOR, BYT and TXT, all of which are used to place data, as distinct from instructions, into the memory. WOR will place a two-byte 'word' in the memory. It will separate the number into high and low bytes, and store the bytes in the correct low-then-high order. WOR$7F12, for example, would place into the memory the bytes $12 and $7F, in that order. These would be stored starting at the 'current address'. Suppose, for example, that you wanted a program to make use of a table of values which started at $9FFØ. You could include in your assembler program the lines:

```
2ØØ *=$7FF1
21Ø WOR $2E14,$115B,$513A
```

and this would store, starting at address $7FF1, the sequence 14,2E,5B,11,3A,51. Note that we can use several addresses following WOR, provided that they are separated by commas.

The BYT pseudo-op does the same for single bytes, so that BYT $ØD will place the carriage-return code at the current address. BYT can also place the ASCII code for letters, using the ' sign. If, for example, you use BYT ' A in the program, then the ASCII code for the letter A will be stored at the current address. TXT is a more convenient method of storing several letters. Following TXT, you can place a string of letters, using quotes at the start and end. The ASCII codes for all the letters, including spaces and punctuation marks, will be placed in memory starting at the current address. For example, using TXT "PRESS ANY KEY" will place the ASCII

codes for all the letters and spaces that are enclosed in the quotes.

### Making it work!

When an assembly language has been typed, it can be recorded just as if it were a BASIC program. This step is important, because this is your 'source code'. 'Source code' means that this is the set of instructions which will assemble up into machine code and, if you have a recording of it, you can reload it, edit it, and re-assemble it as much as you want. Because the source code can be recorded like BASIC, you have to be careful about labelling your cassettes or disks so as not to confuse the two. Source code will not run when you type RUN! To assemble your source code you must type ASSEMBLE, and then press RETURN. The assembler will then go over the source code three times. This has to be done because some of the label names may not be defined at the start of the program. Any errors which are found will usually be detected in either the first or second 'pass' through the source code, and assembly will then stop, with an error message. The error message will, as you might expect, point out what the error is, and in which line it occurs. If there are no errors, then you can expect to see on your screen a display something like this:

```
   ASSEMBLE
   * PASS (1) *
   * PASS (2) *
   * PASS (3) *
**** ASSEMBLY COMPLETE ****
   START ADDRESS - $9FØ1
   END ADDRESS   - $9F43
```

The addresses are useful if you are going to make use of the SAVE command of the TIM monitor to record the machine code on tape or on disk. The start address is also the one that you will need to use when you call the machine code into action by using SYS. Note that the MIKRO assembler does not, by itself, start the program running, unless the last line of the source code is SYS, followed by the correct start address.

This chapter, however, is not intended to give you a full description of the MIKRO assembler. My intent is to give you a taste of what the use of an assembler can be like. If and when you are ready to use MIKRO, you will now be able to make sense of the

rather brief instruction manual that comes with it. Even more important, you will be able to cope with other assemblers, and even with assemblers for other microprocessors, if you should ever change your machine.

# Chapter Nine
# Last Round-up

One of the main problems in writing a book about machine code for beginners is knowing where to stop. Volumes could be written about the machine code programming of the C 64 and still leave room for more, so that any finishing point has to be an arbitrary one. My aim has been to introduce the subject and take you to a level at which you can start to make progress on your own. Once you have reached this stage, you can make use of the other books that are available, which treat machine code at a more advanced level. This chapter is concerned with tying up loose ends, mentioning a few more instructions, and illustrating how to make use of some features of the C 64.

## The stack

You can't get much further in machine code programming without coming across the word *stack*. A 'stack' is a section of memory, and its special use is to preserve bytes that have been kept in registers. There's no special set of memory chips that we use as a stack – but we do set aside part of the RAM for this purpose. For a 6502 microprocessor, we have to use memory in the address range $1Ø0 to $1FF. What you probably find difficult to understand at the present time is why we should ever need to use memory in this way.

Let's take a simple example. Suppose you have a program in which the A register is being used to hold an ASCII code for a character. Suppose, now, in the middle of this program, that we want to create a time delay by making use of a countdown in the A register. Whenever we load the count value into the A register, we shall have replaced the ASCII code number that was stored there, and if we try to use the A register again in the rest of the program, we shall have to reload the address into it. This is what the stack is for.

By means of a single byte instruction, we can store the contents of the accumulator or the status register in the stack memory. Also, by using another similar instruction, we can get the values back into the correct registers again. The act of storing the register(s) on the stack is called 'pushing', and recovering the values is called 'pulling'.

We'll look in a moment at an example program which makes use of the stack but, for now, we're going to return to simpler matters. The rest of this chapter, in fact, will be devoted to examples of programs which will form a basis for developing into really useful routines for your C 64. I must emphasise at this stage that you have now got to the launch-pad as far as machine code is concerned. From now on, what you need is practice, and all the information that you can lay your hands on. Look closely at every program for the C 64 that contains machine code, for example. Even if the machine code is in the form of bytes that are poked into memory, you can disassemble these and find out what they do. By doing this, you can often discover addresses which will be very useful to you in your own programs. From now on, everything is potentially useful to you!

## The KEYBEEP routine

Now we'll look at a routine which illustrates a lot of the points that I have made earlier. It also serves to introduce you to rather more advanced programming. The intention this time is to come up with a program that will cause a short beep to sound each time you press a key. This should happen when you are working normally in BASIC, so what we need is a way of inserting a piece of extra machine code into BASIC. This is quite a different matter from creating a machine code program that runs and then returns to BASIC, and we have to know rather more about the C 64 to be able to carry this out.

To start with, we have the problem of 'breaking in' to the routines that BASIC uses. Fortunately, the C 64 uses one particularly handy 'junction box'. What I mean by a junction box is a piece of code that is placed in the RAM rather than in the ROM. Any code that is placed in the RAM can be changed, unlike code in the ROM. The purpose of this, in fact, is to allow changes to be made by 'patching'. Patching in this sense means inserting a piece of your own program into a routine which is used by the operating system of the C 64.

Looking for a place to patch is the most difficult part of any operation of this sort. There is one routine, located at $Ø324 and

$Ø325, which can be used in this way, but it is executed only when RETURN is pressed. This doesn't quite suit our needs because, for a KEYBEEP program, we want to break into a piece of program that is executed each time a key is pressed. It's almost impossible, working by yourself, to discover suitable addresses in a reasonable time, so we need to make use of the disassembled listing for this. A careful look at the listing reveals an interesting pair of addresses at $Ø28F and $Ø29Ø. These two contain another address, $EB48. This is the address of the keyboard decode routine, and that makes it interesting. What happens, you see, is that when a key is pressed on the keyboard, it causes electrical signals to arrive at a port. The 6502 then has to read these signals and convert them into a single-byte code, using a different code for each key. Obviously, the C 64 does this action in parts, checking first to find if a key is pressed, and then decoding it. A look at a disassembly of the keyboard routine (which starts at $EA87) shows that this address of $Ø28F is used indirectly. That means that the C 64 reads the bytes in $Ø28F and $Ø29Ø each time a key is pressed, and jumps to the address that is given by these two bytes. This looks like the sort of thing that we're looking for. This is where we can do our patching.

What we shall do is this. At the address $Ø28F, we shall replace the original byte of $48 with the low byte of a new address. At address $Ø29Ø, we shall place the high byte of this same new address. We can write a routine of our own which starts at this new address. At the end of this routine, we shall have a JMP instruction. The address following the JMP will be the correct keyboard decoding routine, $EB48. In this way, each time a key is pressed, our routine will be run before the machine gets to work on decoding the key. The routine that we're going to patch in is a simple one. It will poke the correct values for a sound into the sound chip addresses, just as a BASIC program would. It will have to include a delay routine in machine code, so that the sound lasts long enough to be heard, and it will end by returning to address $EB48. It all looks quite easy (when you read it), but it's going to introduce us to quite a lot of new ideas. Some of these are new coding instructions, others are the result of new information about the C 64 itself. At this stage, everything is useful!

Yes, we'll start with a flowchart. Figure 9.1 shows what we are after and, like any good flowchart, it does not specify very closely what we are going to do. Immediately after the start, the decision step 'is a key pressed?' is used. This has actually been taken care of by the operating system, so we don't have to worry any further about it. If a key is pressed, then we load the sound addresses. This is tedious

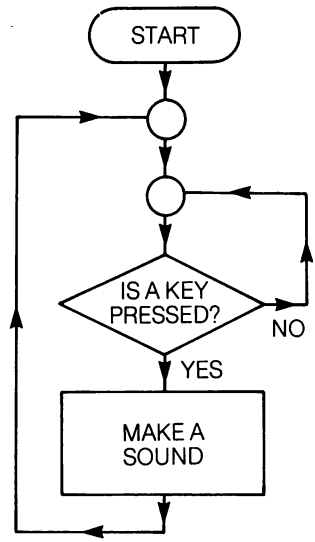



*Fig. 9.1.* The flowchart for a 'keybeep' program that sounds a note whenever a key is pressed.

rather than difficult - it's something that you should keep on tape ready for use after you have typed it once. This loading is followed by a delay, then the 'clear sound' action pokes zeros into some of the sound addresses. If we don't do this, then the sound will continue after the key has been pressed - which is not what we want. Finally, we jump back to BASIC, ready for the next key.

Figure 9.2 shows the assembly language version of what we have come up with. This has been printed out as a set of line numbers in the form that the MIKRO assembler uses. You can assemble this for yourself if you feel that you need the practice, but Fig. 9.3 shows the printout from a MIKRO assembly. This shows the addresses and the code bytes as well as the assembly language. Remember that the starting address has to be $7F00 when the MIKRO cartridge is installed. You can, if you want, rewrite this to fit into higher address numbers if you are not using MIKRO. If you want to use another starting address, part of the code must be altered. The critical part is the address $7F3C for the delay subroutine. If you change the address at which this program starts, then the address of the delay subroutine must also be changed.

## Details, details

Looking at the assembly language in detail, now, we start in the

```
100 *=$7F00
120 LDA #<POINT
130 STA $28F
140 LDA #>POINT
150 STA $290
160 RTS
170 POINT PHA
175 LDA #15
180 STA 54296
190 LDA #190
200 STA 54273
205 LDA #248
210 STA 54278
220 LDA #17
230 STA 54273
240 LDA #37
250 STA 54272
260 LDA #17
270 STA 54276
280 JSR DELAY
290 LDA #0
300 STA 54276
310 STA 54277
320 STA 54278
330 PLA
340 JMP $EB48
350 DELAY LDA #100
360 STA 251
370 LOOP1 STA 252
380 LOOP2 DEC 252
390 BNE LOOP2
400 DEC 251
410 BNE LOOP1
420 RTS
430 END
```

*Fig. 9.2.* The assembly language for a keybeep routine. This has been written in the form of numbered lines for the MIKRO assembler.

usual way by setting the address of $7FØØ. The next part of the program places a new address into $Ø28F and $Ø29Ø. This new address will be the address of the start of the sound routine. At the time when we write these first lines, we don't know what this address will be, so we use the label word POINT. The use of the symbols < and > is not confined to MIKRO; you will find these symbols used

```
100 7F00          *=$7F00
120 7F00 A90B                 LDA #<POINT
130 7F02 8D8F02               STA $28F
140 7F05 A97F                 LDA #>POINT
150 7F07 8D9002               STA $290
160 7F0A 60                   RTS
170 7F0B 48       POINT       PHA
175 7F0C A90F                 LDA #15
180 7F0E 8D18D4               STA 54296
190 7F11 A9BE                 LDA #190
200 7F13 8D01D4               STA 54273
205 7F16 A9F8                 LDA #248
210 7F18 8D06D4               STA 54278
220 7F1B A911                 LDA #17
230 7F1D 8D01D4               STA 54273
240 7F20 A925                 LDA #37
250 7F22 8D00D4               STA 54272
260 7F25 A911                 LDA #17
270 7F27 8D04D4               STA 54276
280 7F2A 203C7F               JSR DELAY
290 7F2D A900                 LDA #0
300 7F2F 8D04D4               STA 54276
310 7F32 8D05D4               STA 54277
320 7F35 8D06D4               STA 54278
330 7F38 68                   PLA
340 7F39 4C48EB               JMP $EB48
350 7F3C A964     DELAY       LDA #100
360 7F3E 85FB                 STA 251
370 7F40 85FC     LOOP1       STA 252
380 7F42 C6FC     LOOP2       DEC 252
390 7F44 D0FC                 BNE LOOP2
400 7F46 C6FB                 DEC 251
410 7F48 D0F6                 BNE LOOP1
420 7F4A 60                   RTS
```

*Fig. 9.3.* The printout from the MIKRO assembler, showing the hex codes as well as the assembly language.

in many 6502 programs in assembly language. The < sign means 'lower byte of' and the > sign means 'higher byte of', so that what we are doing in lines 12Ø to 15Ø is to load the two bytes of the address of POINT into the 'junction box' addresses of $Ø28F and $Ø29Ø. When this has been run, each press of a key will cause the machine to go to the address of POINT.

Now we want to run this 'set-up' part once only. Once we have set up these addresses, we want to return to BASIC, running the rest of

the program only when a key has been pressed. The next instruction in line 16Ø, then, is RTS, returning control to BASIC. The rest of the program is the bit which is used each time a key is pressed.

The beep part of the program starts in line 17Ø with the label word POINT. This ensures that when we assemble, the address of the first instruction will be placed in the 'junction box'. The first instruction is PHA. That's important. PHA means 'push the accumulator on to the stack'. When the machine wants the result of a key-press to be decoded, it will have the result in the accumulator. If we destroy that byte, then we can't expect the machine to operate normally. By using PHA, we preserve the content of the accumulator on the stack. We can then recover it at the end of our routine so that it is in the accumulator all ready when we jump to the decoding routine, just as if our program had not existed. Lines 175 to 27Ø then poke the bytes into the sound addresses to produce a low-pitched beep. Obviously, you can experiment with these values. The addresses are the same as those shown in the C 64 BASIC manual.

The next step is a delay. A subroutine has been used here, though the code could just as easily have been placed in the main part of the program. The subroutine is a familiar delay routine, using page Ø addresses. I avoided using the X or Y registers in case they were also used for the key-decoding routine. The reason for being careful is that there are no instructions for saving the X or Y registers on the stack. The only way this can be done is by transferring the bytes, in turn, to the accumulator, and then pushing the accumulator. When bytes are taken off the stack, they return to the accumulator (if PLA is used) and can then be transferred to other registers. This is so tedious that it is worthwhile trying to avoid having to do it. I've done it by using the page Ø addresses to store the count numbers. A value of 1ØØ (denary) in the stores produces a reasonably short beep.

Finally, the program ends by poking zero into three of the sound addresses to stop the sound. The value that was originally in the accumulator is now restored by the PLA command, and the routine ends with a jump to address $EB48 to decode the key.

When this program is assembled, and SYS32512 used to start it, you will find that you get a beep (and a delay) when each key is pressed. If you don't, perhaps you forgot that the sound comes from the loudspeaker of the TV, and so the volume control of the TV has to be turned up! This routine could set you off on several new tracks. To start with, could you make a program which would assist a blind user, by giving a different note for each key pressed? It would mean using the code that is in the A register at the time when the registers

are pushed to modify the 'pitch' byte that we put into the B register.

## Renumbering - an example of program design

As a grand finale, I'll go over the design of a more elaborate program from start to finish. Finish? No program is ever truly finished until you can hold your hand on your heart and swear that it couldn't possibly be improved. I'm not getting to that stage by any means. I hope, in fact, that you'll find much you can improve on, so that this program will serve as a guide to better things rather than something you just want to assemble and use. The program is a utility which the C 64 sadly lacks, a renumber routine. The idea is that you can poke into memory a starting number and an increment number, and have the lines of a BASIC program renumbered for you just by using a SYS address.

As usual, we had better start with some planning. The basis of the idea is that we can always get the address of the first byte of a BASIC program from addresses $2B and $2C. The next line starts at an address which is stored in the first two bytes of this line, and the line number is stored in the next two bytes. We should, therefore, be able to design a looping program which goes from one line to the next, changing the numbers, until it finds that the next address is ØØØØ. That's the end of the program, and time to stop. Sounds reasonable enough, and Fig. 9.4 summarises it.

The next step, as you should know, is to draw a flowchart, and this is shown in Fig. 9.5. The flowchart shows rather more of what we want to do, and I have added some extra information this time. We

1. Get address of first byte of BASIC program.
2. Save the first byte LB. This is the low byte of the address of the start of the next line.
3. Save the second byte HB. This is the high byte of the address of the start of the next line.
4. Put the line number low byte into the third address in the line.
5. Put the line number high byte into the fourth address in the line.
6. Add the line increment number to the (stored) line starting number.
7. Test the (stored) next line address. If this is ØØØØ, then stop.
8. Repeat, using the new line address taken from HB and LB.

*Fig. 9.4.* A summary in words of what the simple renumber program should do.

need to keep the 'current line address' in store, preferably at two zero page numbers. I have labelled these AD and AD+1. We should also keep the 'current new line number' at a couple of addresses, and these are labelled LIN and LIN+1. Finally, we need an increment. There's only one byte left, at 255, for this, so we'll specify that you can't renumber in increments of more than 255. That doesn't seem to be much of a hardship.

Now I want to emphasise that this is a simple program. It will only renumber the lines themselves, and it won't alter GOTO numbers or GOSUBs. As an exercise in designing machine language utilities, however, it's quite useful. It's also quite useful as an everyday program, provided you remember its limitations. It's ideal for renumbering the sort of programs that you need to write for the MIKRO assembler. In any case, let's give it a whirl.

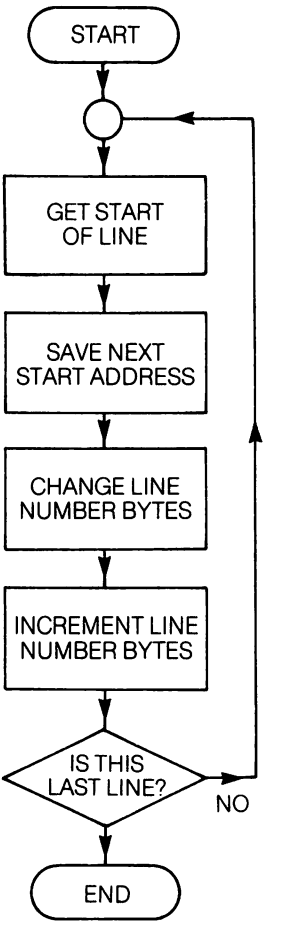



*Fig. 9.5.* A flowchart for the renumber program.

## The big renumber

The program is shown, in the form for MIKRO, in Fig. 9.6. Now this is considerably longer and more elaborate than anything we have done so far. I want, therefore, to go over it in detail, pointing out why each step has been taken. You very seldom get a detailed explanation of how a machine code program works, so take a good look. You'll soon find yourself trying to follow programs in the magazines, and these explanations will stand you in good stead for such work.

The program starts in the usual way by defining a starting address and values for some labels. The starting address has been chosen as $7F00 because I was using MIKRO to develop this program (when they get to this size, I prefer an assembler). Whatever starting value you use, you have to make sure that the memory is protected. When the program is loaded into protected memory, loading in a BASIC program will not disturb the machine code bytes. If you forget to set the top of memory by a poke to address 56, there will be trouble at some stage. The label AD is the zero page address which is used for the low byte of the address of the line that the program is currently working on. This address will change from line to line, so we will make a lot of use of AD. INC is the increment byte, the amount by which the line numbers will increase. This will usually be 10, but it makes the program more flexible if we can change this quantity. Anything from 1 to 255 can be used. LIN holds the low byte of the line number for each renumbered line.

Following this setting up part, we start on the program itself. The carry bit is cleared, just in case, and the accumulator is loaded from address 43. This is the low byte of the address of the first byte of BASIC, and we put it into the 'current line' address at AD. We then load the accumulator from 44, which contains the high byte of the start of BASIC address, and we put this into AD+1. In this way, AD and AD+1 carry a copy of 43 and 44, the bytes of the address of the first byte of your BASIC program.

Now the loop starts. The Y register is zeroed. The LDA (AD), Y will load the accumulator from the address that is held in AD and AD+1. This is the first byte of the BASIC program, and it's the low byte of the next line address. We store it on the stack by using PHA, and then increment the Y register. Because the Y register has been incremented, the next LDA (AD), Y will load from the address of the second byte in the BASIC line. This is the high byte of the address of the next line. This also is pushed on to the stack, next to

```
100 *=$7F00
110 AD=251
120 INC=255
130 LIN=253
140 CLC
150 LDA 43
160 STA AD
170 LDA 44
180 STA AD+1
190 LOOP LDY #0
200 LDA (AD),Y
210 PHA
220 INY
230 LDA (AD),Y
240 PHA
250 INY
260 LDA LIN
270 STA (AD),Y
280 INY
290 LDA LIN+1
300 STA (AD),Y
310 LDA LIN
320 ADC INC
330 STA LIN
340 BCC NXT
350 INC LIN+1
360 CLC
370 NXT PLA
380 STA AD+1
390 PLA
400 STA AD
410 BNE LOOP
420 LDA AD+1
430 BNE LOOP
440 RTS
450 END
```

*Fig. 9.6.* The assembly language listing for the renumber program. This is in the form that the MIKRO assembler uses.

the low byte. The Y register is then incremented again. The next step is to get our own first line number low byte from LIN. This is stored in the next byte of the line, again using the indirect Y addressing method. The Y register is incremented again, the high byte of the starting line number is loaded into the accumulator, and stored into

the line using indirect addressing. That completes the renumbering of the first line with the first of the line numbers that we want.

The next step is to add the increment to the line number. We load the accumulator from LIN, and add the increment from INC. The number in the accumulator (the low byte) is placed back in LIN, and if there is a carry, the high byte at LIN+1 is incremented. The carry is cleared – just to be sure! Now we have to put the next line number address into the AD and AD+1 addresses, so that we can repeat the processes. The high byte of the next line number was pushed last on the stack, so it comes off first, with the PLA command. It is stored in AD+1. The low byte is then pulled off and stored at AD. We can test at this point to see if the low byte of the next address is Ø. If it's not, then we haven't reached the end of the program yet, and we loop back. If this byte is zero, we have to test the high byte, by taking it from AD+1. If this byte is not zero, then we haven't reached the end and we loop back. If both bytes are zero, however, we have quite certainly reached the end of the program, and the RTS returns us to BASIC.

How do we use it? Reserve memory, and put the machine code in place. Load in your BASIC program and decide on a starting line number and an increment. Poke the start into 253 and 254, with the high byte in 254. For a starting number less than 255, like 1Ø or 1ØØ, you will only have to poke a number into 253, but it's safer to poke Ø into 254. If you want to start with line 1ØØØ, you will have to express this as two bytes – Fig. 9.7 shows how. Then poke the increment (usually Ø) into address 255. If you have assembled at $7FØØ, then

---

1ØØØ as a starting line number consists of two bytes. Divide 1ØØØ by 256, and the result is 3.9Ø625. The whole number part is 3. Now take 1ØØØ – 3*256, which is 232. 232 is the low byte of the line number.

---

*Fig. 9.7.* Splitting a line number like 1ØØØ into two bytes.

active the program by using SYS32512. In a flicker of an eyelid, your BASIC program is renumbered. If you assembled at any other address, of course, that's the address that you will have to use with SYS. Figure 9.8 shows the MIKRO assembly printout so that you can read the codes. The code can be assembled at any address, so you can transform this into a BASIC poke program if you want.

That's the end of this particular road, then. There's very little that's new to learn, except about the hidden surprises that the C 64

```
100 7F00          *=$7F00
110 7F00          AD          = 251
120 7F00          INC         = 255
130 7F00          LIN         = 253
140 7F00 18                   CLC
150 7F01 A52B                 LDA 43
160 7F03 85FB                 STA AD
170 7F05 A52C                 LDA 44
180 7F07 85FC                 STA AD+1
190 7F09 A000     LOOP        LDY #0
200 7F0B B1FB                 LDA (AD),Y
210 7F0D 48                   PHA
220 7F0E C8                   INY
230 7F0F B1FB                 LDA (AD),Y
240 7F11 48                   PHA
250 7F12 C8                   INY
260 7F13 A5FD                 LDA LIN
270 7F15 91FB                 STA (AD),Y
280 7F17 C8                   INY
290 7F18 A5FE                 LDA LIN+1
300 7F1A 91FB                 STA (AD),Y
310 7F1C A5FD                 LDA LIN
320 7F1E 65FF                 ADC INC
330 7F20 85FD                 STA LIN
340 7F22 9003                 BCC NXT
350 7F24 E6FE                 INC LIN+1
360 7F26 18                   CLC
370 7F27 68       NXT         PLA
380 7F28 85FC                 STA AD+1
390 7F2A 68                   PLA
400 7F2B 85FB                 STA AD
410 7F2D D0DA                 BNE LOOP
420 7F2F A5FC                 LDA AD+1
430 7F31 D0D6                 BNE LOOP
440 7F33 60                   RTS
```

*Fig. 9.8.* The MIKRO assembler printout for the renumber program. This has been renumbered by the program!

keeps for you. As you go on, though, you will accumulate experience, until you find that the surprises are fewer, and you can find ways round them more easily. When that time arrives, you're entitled to call yourself an expert, a real machine code programmer.

# Appendix A
# How Numbers are Stored

The C 64 uses five bytes of memory to store any number. This Appendix describes how numbers are stored, but if you have no head for mathematics, you may not be any the wiser!

To start with, floating point (not integer) numbers are stored in *mantissa-exponent* form. This is a form that is also used for denary numbers. For example, we can write the number 216ØØØ as $2.16 \times 10^5$, or the number .ØØØ12 as $1.2 \times 10^{-4}$. When this form of writing numbers is used, the power (of ten in this case) is called the *exponent*, and the multiplier (a number greater than 0 and less than 1) is called the *mantissa*. Binary numbers can also be written in this way, but with some differences. To start with, the mantissa of a binary number that is written in this form is always fractional, but no point is written. Secondly, the exponent is a power of two rather than a power of ten. We could therefore write the binary number 1Ø11ØØØØ as 1Ø11E1ØØØ. This means a mantissa of 1Ø11 (imagine it as .1Ø11) and exponent of 1ØØØ (2 to the power 8 in denary). There's no advantage in writing small numbers in this way, but for large numbers, it's a considerable advantage. The number:

11Ø1Ø1ØØØØØØØØØØØØØØØØ

for example can be written as 11Ø1Ø1E11ØØØ (think of it as .11Ø1Ø1 $\times 2^{24}$).

This scheme is adapted for the C 64, and other machines which use Microsoft BASIC. Since the most significant digit of the mantissa (the fractional part of the number) is always 1 when a number is converted to this form, it is converted to a Ø for storage purposes. The exponent then has (denary) 128 added to it before being stored. This allows numbers with negative exponents up to −128 to be stored without complications, since a negative exponent is then stored as a number whose value is less than 128 denary. The C 64 uses four bytes

to store the mantissa of a number, and one byte to store the exponent.

To take a simple example, consider how the number 2Ø (denary) would be coded. This converts to binary as 1Ø1ØØ, which is .1Ø1ØØØØ × $2^5$, writing it with the binary point shown, using eight bits, and with the exponent in denary form. The msb of the fraction is then changed to Ø, so that the number stored is ØØ1ØØØØ. Peeking this memory will therefore produce the number (denary) 32 in the mantissa lowest byte. Meantime, the exponent of 5 is in binary 1Ø1. Denary 128 is added to this, to make 1ØØØØ1Ø1. Peeking this memory will give you 133 (which is 128+5).

Integers need only two bytes for storage. The integer must have a value between −32768 and +32767. To convert an integer to the form in which the C 64 stores it, proceed as follows:

1. If the number is negative, subtract it from 65536 and use the result.
2. Divide the number by 256 and take the whole part. This is the most significant byte.
3. Subtract from the number 256 × (most significant byte). This gives the least significant byte.

*Example:* Convert 9438 into integer storage form.
The number is positive, so it can be used directly.
9438/256=36.867187.
The msb is therefore 36.
36*256=9216, and 9438−9216=222.
The low byte is 222.

# Appendix B
# Hex and Denary Conversions

### (a) Hex to Denary

*For single bytes (two hex digits) -*
Multiply the most significant digit by 16, and add the other digit.
*For example:*

$3D is 3*16 + 13 =61 denary.

*For double bytes (address numbers) -*
Write down the least significant digit. Now write under it the value of the next digit, multiplied by 16. Under that, write the next digit, multiplied by 256. Under that, write the next digit, multiplied by 4096.
*For example:*

$F3DB converts as follows:

| | |
|---|---|
| Write 1s digit | 11 |
| Next digit*16 is 13*16 | 208 |
| Next digit*256 is 3*256 | 768 |
| Next digit*4096 is 15*4096 | 61440 |
| Now take the total, which is | 62427 |

### Denary to hex

*For single bytes (less than 256 denary) -*
Divide by 16. The whole part of the number is the most significant digit. The least significant digit is the fractional part of the result multiplied by 16.

*For example:*

To convert 155 to hex:
153/16 = 9.6875, so 9 is the most significant digit.
The least significant digit is .6875*16, which is 11. This converts to hex B, so that the number is $9B.

*For double-byte numbers (numbers between 256 and 65535 denary)*
Divide by 16 as before. Note the whole number part of the result, and write down the fractional part, times 16, as a hex digit. Repeat the action with the whole number part, until only a single hex digit remains.
*For example:*

To convert 23815 to hex:
23815/16=1488.4375. The fraction .4375*16 gives 7, and this is the least significant digit.
Taking the whole number part, 1488/16=93.ØØ. Since there is no fraction, the next hex digit is Ø.
93/16=5.8125. The fraction .8125, multiplied by 16 gives 13, which is hex D. This is the third hex figure. Since the whole number part is less than 16 (it's 5), then this is the most significant digit, and the whole number is $5DØ7.

# Appendix C
# The Instruction Set

The instruction set of the 6502 is a relatively small one, but there will be some instructions in this list which you may never need to use unless you go in for very advanced programming indeed. A full description of the action of each instruction would take too much space, and so the action has been indicated by abbreviations. For a full description, see one of the books devoted to 6502 programming. In general, M means a byte at an address in memory, and the registers are referred to under their usual letter references. An arrow indicates where the result of an action is stored. For example, A+M+C→A+C means that the byte in the memory (addressed by the instruction) is added to the byte in the accumulator A, plus the carry C, and the result is placed in the accumulator A, with another possible carry in C.

The instruction codes are shown in columns graded by the addressing method. These methods are Immediate, Zero Page, Zero Page X Indexed, Absolute, Absolute X Indexed, Absolute Y Indexed, Indirect X, Indirect Y, and PC Relative. A few instructions use Implied Addressing. The implied addressing method means that no special addressing is needed. In the assembly language forms, ADDR means a full two-byte address and addr means a single (lower) byte address for zero page addressing. Disp is used to mean displacement in PC Relative addressing. Byte means the byte following an immediate addresses code. S has been used to mean the Processor Status register. Flags are referred to as C,N and V. All op codes are shown in hex.

| *Assembly language form* | *Addressing method* | *Opcode* | *Action* |
|---|---|---|---|
| ADC # Byte | Immediate | 69 | A+M+C→A+C |
| ADC addr | Zero page | 65 | |
| ADC addr, X | Zero page, X | 75 | |
| ADC ADDR | Absolute | 6D | |
| ADC ADDR,X | Absolute, X | 7D | |
| ADC ADDR, Y | Absolute, Y | 79 | |
| ADC (addr, X) | Indirect, X | 61 | |
| ADC (addr), Y | Indirect, Y | 71 | |
| AND # Byte | Immediate | 29 | A AND M→A |
| AND addr | Zero page | 25 | |
| AND addr, X | Zero page, X | 35 | |
| AND ADDR | Absolute | 2D | |
| AND ADDR,X | Absolute, X | 3D | |
| AND ADDR, Y | Absolute, Y | 39 | |
| AND (addr, X) | Indirect, X | 21 | |
| AND (addr), Y | Indirect, Y | 31 | |
| ASL A | Implied | ØA | Shift left |
| ASL addr | Zero page | Ø6 | |
| ASL addr, X | Zero page, X | 16 | |
| ASL ADDR | Absolute | ØE | |
| ASL ADDR, X | Absolute, X | 1E | |
| BCC Disp | Relative | 9Ø | Branch if C=Ø |
| BCS Disp | Relative | BØ | Branch if C=1 |
| BEQ Disp | Relative | FØ | Branch if Z=1 |
| BIT addr | Zero page | 24 | OR with M, |
| BIT ADDR | Absolute | 2C | Test N & V |
| BMI Disp | Relative | 3Ø | Branch if N=1 |
| BNE Disp | Relative | DØ | Branch if Z=Ø |
| BPL Disp | Relative | 1Ø | Branch if N=Ø |
| BRK | Implied | ØØ | Interrupt program |
| BVC Disp | Relative | 5Ø | Branch if V=Ø |
| BVS Disp | Relative | 7Ø | Branch if V=1 |
| CLC | Implied | 18 | Clear carry |
| CLD | Implied | D8 | Clear decimal mode |

| *Assembly language form* | *Addressing method* | *Opcode* | *Action* |
|---|---|---|---|
| CLI | Implied | 58 | Clear interrupt disable |
| CLV | Implied | B8 | Clear V flag |
| CMP #Byte | Immediate | C9 | A–M, set flags |
| CMP addr | Zero page | C5 | |
| CMP addr, X | Zero page, X | D5 | |
| CMP ADDR | Absolute | CD | |
| CMP ADDR, X | Absolute, X | DD | |
| CMP ADDR, Y | Absolute, Y | D9 | |
| CMP (addr, X) | Indirect, X | C1 | |
| CMP (addr), Y | Indirect, Y | D1 | |
| CPX # Byte | Immediate | Eø | X–M set flags |
| CPX addr | Zero page | E4 | |
| CPX ADDR | Absolute | EC | |
| CPY # Byte | Immediate | Cø | Y–M set flags |
| CPY addr | Zero page | C4 | |
| CPY ADDR | Absolute | CC | |
| DEC addr | Zero page | C6 | M–1→M |
| DEC addr, X | Zero page, X | D6 | |
| DEC ADDR | Absolute | CE | |
| DEC ADDR, X | Absolute, X | DE | |
| DEX | Implied | CA | X–1→X |
| DEY | Implied | 88 | Y–1→Y |
| EOR # Byte | Immediate | 49 | A EOR M>A |
| EOR addr | Zero page | 45 | |
| EOR addr, X | Zero page, X | 55 | |
| EOR ADDR | Absolute | 4D | |
| EOR ADDR, X | Absolute, X | 5D | |
| EOR ADDR, Y | Absolute, Y | 59 | |
| EOR (addr, X) | Indirect, X | 41 | |
| EOR (addr), Y | Indirect, Y | 51 | |
| INC addr | Zero page | E6 | M+1→M |
| INC addr, X | Zero page, X | F6 | |
| INC ADDR | Absolute | EE | |
| INC ADDR, X | Absolute, X | FE | |
| INX | Implied | E8 | X+1→X |
| INY | Implied | C8 | Y+1→Y |

| *Assembly language form* | *Addressing method* | *Opcode* | *Action* |
|---|---|---|---|
| JMP ADDR | Absolute | 4C | Jump to ADDR |
| JMP (ADDR) | Indirect | 6C | Jump to stored address |
| JSR ADDR | Absolute | 20 | Jump to subroutine |
| LDA # Byte | Immediate | A9 | M→A |
| LDA addr | Zero page | A5 | |
| LDA addr, X | Zero page, X | B5 | |
| LDA ADDR | Absolute | AD | |
| LDA ADDR, X | Absolute, X | BD | |
| LDA ADDR, Y | Absolute, Y | B9 | |
| LDA (ADDR, X) | Indirect, X | A1 | |
| LDA (ADDR), Y | Indirect, Y | B1 | |
| LDX # Byte | Immediate | A2 | M→X |
| LDX addr | Zero page | A6 | |
| LDX addr, Y | Zero page, Y | B6 | |
| LDX ADDR | Absolute | AE | |
| LDX ADDR, Y | Absolute, Y | BE | |
| LDY # Byte | Immediate | AØ | M→Y |
| LDY addr | Zero page | A4 | |
| LDY addr, X | Zero page, X | B4 | |
| LDY ADDR | Absolute | AC | |
| LDY ADDR, X | Absolute, X | BC | |
| LSR A | Implied | 4A | Shift right |
| LSR addr | Zero page | 46 | |
| LSR addr, X | Zero page, X | 56 | |
| LSR ADDR | Absolute | 4E | |
| LSR ADDR, X | Absolute, X | 5E | |
| NOP | Implied | EA | No operation |
| ORA # Byte | Immediate | Ø9 | A OR M→A |
| ORA addr | Zero page | Ø5 | |
| ORA addr, X | Zero page, X | 15 | |
| ORA ADDR | Absolute | ØD | |
| ORA ADDR, X | Absolute, X | 1D | |
| ORA ADDR, Y | Absolute, Y | 19 | |
| ORA (addr, X) | Indirect, X | Ø1 | |
| ORA (addr), Y | Indirect, Y | 11 | |
| PHA | Implied | 48 | Push A on stack |

| *Assembly language form* | *Addressing method* | *Opcode* | *Action* |
|---|---|---|---|
| PHP | Implied | ϕ8 | Push S on stack |
| PLA | Implied | 68 | Pull A from stack |
| PLP | Implied | 28 | Pull S from stack |
| ROL A | Implied | 2A | Rotate left |
| ROL addr | Zero page | 26 | |
| ROL addr, X | Zero page, X | 36 | |
| ROL ADDR | Absolute | 2E | |
| ROL ADDR, X | Absolute, X | 3E | |
| ROR A | Implied | 6A | Rotate right |
| ROR addr | Zero page | 66 | |
| ROR addr, X | Zero page, X | 76 | |
| ROR ADDR | Absolute | 6E | |
| ROR ADDR, X | Absolute, X | 7E | |
| RTI | Implied | 4ϕ | Return from interrupt |
| RTS | Implied | 6ϕ | Return from sub-Routine |
| SBC # Byte | Immediate | E9 | A-M-C*→M |
| SBC addr | Zero page | E5 | C* is a borrow |
| SBC addr, X | Zero page, X | F5 | |
| SBC ADDR | Absolute | ED | |
| SBC ADDR, X | Absolute, X | FD | |
| SBC ADDR, Y | Absolute, Y | F9 | |
| SBC (addr, X) | Indirect, X | E1 | |
| SBC (addr), Y | Indirect, Y | F1 | |
| SEC | Implied | 38 | Set carry flag |
| SED | Implied | F8 | Set decimal mode |
| SEI | Implied | 78 | Set interrupt disable |
| STA addr | Zero page | 85 | A→M |
| STA addr, X | Zero page, X | 95 | |
| STA ADDR | Absolute | 8D | |
| STA ADDR, X | Absolute, X | 9D | |
| STA ADDR, Y | Absolute, Y | 99 | |
| STA (addr, X) | Indirect, X | 81 | |
| STA (addr), Y | Indirect, Y | 91 | |
| STX addr | Zero page | 86 | X→M |
| STX addr, Y | Zero page, Y | 96 | |
| STX ADDR | Absolute | 8E | |

| *Assembly language form* | *Addressing method* | *Opcode* | *Action* |
|---|---|---|---|
| STY addr | Zero page | 84 | Y→M |
| STY addr, X | Zero page, X | 94 | |
| STY ADDR | Absolute | 8C | |
| TAX | Implied | AA | A→X |
| TAY | Implied | A8 | A→Y |
| TYA | Implied | 98 | Y→A |
| TSX | Implied | BA | S→X |
| TXA | Implied | 8A | X→A |
| TXS | Implied | 9A | X→S |

# Appendix D
# Addressing Methods of the 6502

Each addressing method has the effect of using a byte in the memory. The address at which this byte is stored is called the Effective Address (EA). The purpose of any addressing method is to make use of an effective address.

*Immediate Addressing:* The EA is the address that immediately follows the instruction byte.

*Zero Page Addressing:* Only the lower byte of the EA is given in the instruction. The upper byte is always $\emptyset\emptyset$, hence the name of zero page. For example, using zero page addressing with a byte of FB would make use of the address $\$\emptyset\emptyset$FB.

*Absolute Addressing:* The instruction is followed by two bytes, which form a complete address. For example, LDA $563F means that the accumulator is to be loaded from the address $563F.

*Indexed Addressing:* A number is stored in one of the index registers (X or Y). The effective address is this number plus any address that is specified in the instruction. For example, LDA 25, X means load the accumulator from the address which consists of the number stored in the X register, plus 25.

*Implied Addressing:* The address is implied by the instruction, and no special address reference is needed. For example, INX means increment the X register, and no EA is needed.

*Indirect Addressing:* There are two forms, both of which use zero page addresses. The X-indexed indirect adds the contents of the X register to the zero page address number, and fetches the byte at this address. This forms the low byte of the effective address. The high byte is then fetched from the next higher page zero address. The Y-indexed indirect fetches the byte from the page zero address, and

then adds the contents of the Y register to this byte. This is the lower byte of the effective address, and the higher byte is fetched from the next zero page address as before.

# Appendix E
# A Few ROM and RAM Addresses

Now that a full disassembly of the C 64 ROM is available, all of the ROM addresses will be widely known. A selection of the most useful addresses is shown below. Along with this, I have included some useful RAM locations, with brief notes on what is stored there.

### ROM addresses

INPUT routine $F157
OUTPUT routine $F1CA
Read keyboard $E112
Print to screen $E1ØC
Begin BASIC $AØØØ
READY message $A376
NEW $A644
Execute statement $A7E4
Video control chip $DØØØ to $DØ21
Port 1 $DCØØ to $DCØF
Port 2 $DDØØ to DDØF

### RAM addresses

$2B,$2C Start of BASIC
$2D,$2E Start of VLT
$2F,$3Ø Start of arrays
$31,$32 End of arrays
$33,$34 Start of string storage
$37,$38 Limit of memory
$9A Code for output device
$C5 Last key pressed
$CC Cursor enable (Ø=flash)

$D1, $D2 Screen line in use
$D3 Position of cursor
$D6 Cursor line number
$F3,$F4 Colour memory pointer

$Ø2ØØ BASIC input buffer
$Ø277 Keyboard buffer
$Ø28A Keyboard repeat ($8Ø makes all keys repeat)
$Ø28F,$Ø29Ø Address of keyboard decoding routine

# Appendix F
# Magazines and Books: Where to get the MIKRO 64 Assembler

The problem of where you go from here is solved by looking at the magazines and books that are available. The groundwork that this book has supplied should allow you to go on to any of the books that deal with 6502 programming, but which are not very useful to the complete beginner. A look at the books available in your local computer store, or from specialist mail order suppliers will show you what is available. I have already mentioned the very useful book by Milton Bathurst: *Inside the Commodore 64.*

Monthly magazines are also a fruitful source of ideas. *Personal Computer World's* series, *SUBSET*, is a very valuable source of ideas in machine code programming. Alan Tootill and David Barrow of *SUBSET* are writing a book: *6502 Machine Code for Humans* which should be published early in 1984, and this could be useful. *Your Computer* frequently prints articles on machine code topics for the C 64, and you should watch out for listings which may reveal the use of ROM routines that you haven't met before. Remember, too, that a lot of published information regarding the old PET models will be useful for the C 64.

### The MIKRO 64 Assembler

The MIKRO 64 Assembler is distributed in Europe and the rest of the world except USA and Canada by:

SUPERSOFT
Winchester House
Canning Road
Wealdstone
Harrow HA3 7SJ
UK

In the USA and Canada it is distributed by:

Skyles Electric Works
231E S Whisman Road
Mountain View
CA 94041
USA

# Index

Computers

When it comes to programming dazzling Commodore 64 effects like fast-moving graphics, games, key-beeps, and split screens, BASIC just isn't enough! You need the computer's own language at your command: machine code. And now, with this book, you can crack open a new world of possibilities for your computer.

INTRODUCING COMMODORE 64 MACHINE CODE's clear step-by-step guidelines will help you quickly master the fundamentals of machine code programming. As long as you know some BASIC, you need never have encountered it before! Here, you'll learn *what* machine code is, *how* it works, and how to *enter, run,* and *save* code. You'll be coached along with vital tips on how to write machine code most efficiently. Best of all, when you finish the book, you'll not only have access to many more Commodore 64 applications, but a more intricate knowledge of your micro itself.

So "break the code" and break into the exciting sights and sounds your Commodore 64 can give you.

IAN SINCLAIR is a well-known and regular contributor to such journals as *Personal Computer World, Computing Today, Electronics and Computing Monthly,* and *Electronics Today International.* He has written more than forty books on electronics and computing.

3
ISBN 0-13-477316-0

0 21898 47731 6

Cover design by Hal Siegel
PRENTICE-HALL, Inc.,
Englewood Cliffs, New Jersey 07632