DR. WATSON

Computer Learning Series

Beginners Assembly Language Programming for the CBM 64

Derek Bush & Peter Holmes

Assembly Language Programming

for the

CBM 64

Derek Bush & Peter Holmes

Other Books in this Series

The Dr Watson Book of Beginners Assembly Language Programming for the VIC 20 (4th Edition)
ISBN 0 907792 10 3

The Dr Watson Book of Beginners Assembly Language Programming for the Commodore PET 2/3/4/8000 (2nd Edition)
ISBN 0 907792 02 2

The Dr Watson Book of Beginners BASIC for the SPECTRUM (2nd Edition)
ISBN 0 907792 07 3

The Dr Watson Book of Beginners Assembly Language Programming for the BBC
ISBN 0 907792 08 1

The Dr Watson Book of Beginners BASIC for the CBM 64
ISBN 0 907792 14 6

The Dr Watson Book of Beginners BASIC for the ORIC
ISBN 0 907792 12 X

The Dr Watson Book of Beginners BASIC for the Dragon
ISBN 0 907792 13 8

The Dr Watson Book of Advanced BASIC for the BBC
ISBN 0 907792 15 4

The Dr Watson BASIC Adventure Part 1 for the CBM 64 (age 7 upwards)
ISBN 0 907792 16 2

NOVEMBER 1983

All programs in this book have been written expressly to illustrate specific teaching points. They are not warranted as being suitable for any particular application. Every care has been taken in the writing and presentation of this book but no responsibility is assumed by the author or publishers for any errors or omissions contained herein.

Series Editor : Peter Holmes.BSc,PhD,CEng,MIMechE.
Graphic Design : Sarah Stanbury, BA Hons.
Word Processing: Paula Papanicolaou

ISBN 0 90772 17 0

Published by: Glentop Publishers Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 1ED

Printed by Image Publicity 2 Northington Street London WC1N 2JJ Tel: 01-405 5333

CONTENTS

INTRODUCTION

This book is a self-paced course on machine-code/Assembly-language programming based on the 6510 microprocessor. Getting going in assembly-language is said to be a major problem for the beginner. However, it's a problem that we trust will disappear when the beginner's hands get on this book. By plotting a careful route through the instruction set, we have produced a text that will take the reader stage-by-stage through 6510 assembly language maintaining interest and understanding at all times. In order to overcome the usual difficulty associated with assembly languages, the Mnemonics used are totally uniform throughout the book and the instruction set used is that employed by the assembler that was written especially to accompany this text. Most of the examples are designed to be loaded via this assembler and all are designed to run on the Commodore 64.

In order to ensure that the vital first few machine code programs run, the opening chapter goes in great detail through the stages of entering, assembling, listing and running these programs. Those who bring to the book pre-existing skill are asked to bear with the text in these early stages. Notwithstanding the gentle pace of the opening, the reader should find himself/herself accelerating along with the text. By the end of chapter eight he/she will have covered the whole of the 6510 instruction set.

Just to test your understanding at each stage, the text is interspersed with numerous exercises. In all cases, the solutions to these are provided and, in many cases, detailed explanations are given to aid your understanding. As you've paid for these exercises, - don't waste your money - have a go at them. These too are carefully graded to expand your understanding of 6510.

As a further aid, the tape supplied with this book also contains a Binary BCD Hexadecimal Tutor. Appendix 1 provides a full explanation of how to use this program and should ensure that you understand binary, binary-coded-decimal and hexadecimal really well. A further feature of this program - the exercises - will enable you to test just how well you do understand. If you're shy, just try them when no-one's looking!

Well that's all the propoganda - we've done what we can to make learning assembler fun - the rest is up to you. Enjoy yourself.

Derek J Bush
Peter Holmes
London
May 1983

CHAPTER ONE

Having driven a computer before you are probably well aware that a 'something' exists called "machine-code" along with that other thing known as "assembler". Quite simply, machine-code is the language that the micro-processor chip in your computer understands. As an example take a simple addition sum - adding 1 and 2.

In English you would say:-

Add one to two, what's the answer?

In Basic you might say something like:-

```
10  LET A = 1
20  LET B = 2
30  C = A + B
40  PRINT C
```

In a 6510 machine-code you could say:-

```
A2  01
8E  84 03
A9  02
6D  84 03
8D  85 03
60
```

This is pretty-well unintelligible, isn't it? Well, that's why we use assembly language. The same problem is given below in assembly language along with a brief comment on each line.

```
LDXIM  1       Load X register with a '1'.
STX    900     Store the contents of X register in 900.
LDAIM  2       Load the Accumulator with a '2'.
ADC    900     Add to the accumulator the contents of
               memory location 900.
STA    901     Store contents of accumulator in memory
               location 901.
RTS            Return from machine-code sub-routine.
```

It's much easier to read that than the machine-code, isn't it? With an assembler you can enter your program in assembly language and be able to read through it and understand it readily. All the assembler does is to change the assembly language into machine-code. Thus, when it sees "LDXIM", it changes this command into an 'A2' and puts this into memory in the right place.

It is possible to enter machine-code directly into memory and this will be demonstrated later on in this book. However, until then we will concentrate mainly on assembly language programs.

The heart of the 6510 is the accumulator (A) through which almost all your data has to flow. It is basically an eight bit store that can store a number up to 255.

6510 instructions allow you to write directly into this store using the instruction "LoaD a number into the Accumulator using Immediate Mode". The Mnemonic for this is LDAIM.

LDAIM	LoaD Accumulator using Immediate Mode (i.e. with the value specified).

Another instruction allows you to transfer a number from this store to any specified memory location. If this memory location is between 1024 and 2023 then the number taken from the store will be displayed on the screen. The instruction is:-

STA	STore contents of Accumulator in the address specified.

Note that the TRANSFER commands you will meet should really be thought of as COPY commands as they take a copy of the data and create a SECOND copy of that data, leaving the original unaltered.

Let's have a go then at running a machine-code program!

We will create a program which will put a number into the accumulator and then transfer it to the top left position on the screen, i.e. location 1024.

A couple of points about the assembler: when you start to write a program the assembler needs to be told where you want the program to be placed in the CBM 64's memory. The CBM 64 has plenty of memory available where we can put a machine code program (in theory, we could use the whole 64k). However, for short programs the cassette buffer, which extends from 828 to 1019, provides a convenient 192 bytes. I say convenient as programs which are stored there are unlikely to get mixed up with any BASIC programs. So in our early programs we will make use of the cassette buffer and tell the assembler to start the programs there.

A second point about the assembler is that initially we will only use decimal format for numbers. Later on we shall see how other number formats may be used.

Lastly, the assembler must be told when we have finished entering the program. Thus, the first and last lines of the assembly program are:-

```
START ADDRESS? 828
..................
.... program .....
..................
END
```

N.B. The underlining of START ADDRESS? in the above is intended to indicate that this has been typed by the computer. This convention is used throughout this book and should make it easier for you to interpret the examples. Remember, if it is underlined, then the computer will type this; if it isn't underlined, then this is the bit that you will have to type in yourself.

The first and last lines shown above have nothing to do with the machine code program. They simply provide information to the assembler. (The word END is called a PSEUDO-CODE).

When you have put in your machine-code program you may run it by using the RUN facility provided by the assembler (or by using a SYS call to the start of the machine code program i.e. SYS 828 in this case). Either way you must tell the program to return from machine-code to BASIC or monitor. The command that does this is ReTurn from machine-code Subroutine or RTS.

Right, to put that into the program we must:

1. Tell the assembler that the START ADDRESS? is 828.

2. LoaD number '0' into accumulator A using Immediate Mode. The Mnemonic for this is LDAIM followed by the number to be loaded i.e. LDAIM 0

3. STore in a specific address the contents of the Accumulator, the Mnemonic is STA. After this we must tell the 6510 what the address is i.e. STA 1024

4. STore in another specific address the contents of the Accumulator i.e. STA 55296. I will explain why we do this later.

5. ReTurn from the Subroutine to BASIC i.e. RTS

6. Tell the assembler (not the 6510) to END

or

PROGRAM 1.1

```
START ADDRESS? 828
LDAIM 0
STA 1024
STA 55296
RTS
END
```

Now to enter this:-

a) Load the ASSEMBLER program into your machine.

b) Type in RUN <return>

c) Screen shows MENU

d) Select 'E' to Enter the program. Screen tells you to enter the assembly language program and prompts with
    START ADDRESS?

e) Type in "828" (without the quotes " of course) and press the <return> key. After the assembler has entered the code, it will type a "?" then:

f) Type in "LDAIM" <space> "0" press <return>

g) Type in "STA" <space> "1024" <return>

h) Type in "STA" <space> "55296" <return>

i) Type in "RTS" <return>

j) Type in "END"

At this stage your program should appear as follows:-

```
START ADDRESS? 828
? LDAIM 0
? STA 1024
? STA 55296
? RTS
? END
```

If it does, then O.K. press <return> and carry on to j). If it doesn't, press <return> and then go back to d).

Program returns to MENU.

k) Select "R" to Run the program. Screen asks for address of program start.

l) Type in "828" <return>. After this input the program first clears the screen and then runs the machine-code program which will print a black "@" in the top left-hand corner of the screen.

m) Press any key and continue. Program returns to MENU.

Now select 'L' to list the program.

The program then asks you for the <u>START ADDRESS?</u>

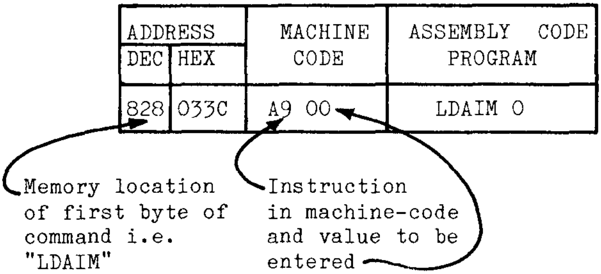
Type in "828" <return>

The screen displays:-

ADDRESS		MACHINE CODE	ASSEMBLY CODE PROGRAM
DEC	HEX		
828	033C	A9 00	LDAIM 0
830	033E	8D 00 04	STA 1024
833	0341	8D 00 D8	STA 55296
836	0344	60	RTS
etc		etc.	etc.

Your machine will show a whole screen of data but anything below the line beginning 836 will be ignored as the 6510 reads the RTS (ReTurn from machine-code Sub-routine) and returns to BASIC.

What the above shows is, taking the first line of program 1.1



ADDRESS		MACHINE CODE	ASSEMBLY CODE PROGRAM
DEC	HEX		
828	033C	A9 00	LDAIM 0

Memory location of first byte of command i.e. "LDAIM"

Instruction in machine-code and value to be entered

Thus the listing is both a check on what you entered and also gives you the full machine-code program or the object program. Reading this off from below the assembly language listing this is:-

```
A9  00
8D  00  04
8D  00  D8
60
```

This object code could be entered directly into memory and would yield the same results as the program you typed in. The assembly language only helps you to compile the program in the first place.

There is one point that might have been puzzling you which I will deal with before I introduce anything else. Why did we store a zero in 55296 ? The answer to this lies in the way that the CBM 64 puts colour on to the screen. If we store a value directly into the screen (as we did when we stored the zero in 1024), then we also need to tell the CBM 64 the colour to be used for the display. If we don't then the result is likely to be displayed in the same colour as the screen background, and a light grey character on a light grey background can't be seen at all. You would never know whether your very first program was a success or not.

Storing a zero in the first byte of the colour RAM (i.e. 55296) ensured that the character stored in the first byte of the screen (i.e. 1024) would be shown up as a black character. Similarly, if we had stored a value into 1025 (the second character of the screen) then we need to store a colour code into 55297 etc. We could have stored any value from 0 to 15 in 55296 and we would have got one of the sixteen colours that the CBM 64 can display. One value that we don't want to store in the colour RAM is the number representing the colour of the screen background, of course. In the case of the assembler, the screen background is light grey which is given by the value 15.

Now let us look at another instruction and use this in a program. As stated earlier, the accumulator is the repository of most "answers" and the new instruction ADC "does a sum" and loads the answer into the accumulator.

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

To do this, however, we must first add two lines to front of the program. These lines simply get the 6510 ready to do some adding. Don't worry what they mean for now - just type 'em in! - and follow the instructions.

One other point about the jargon! The term INSTRUCTION is used to describe an executable machine code statement. Thus it could consist of LDAIM or just RTS. However, the term is also used to refer to the mnemonic alone, as when one says the 6510 instruction set. In this book, the term COMMAND is used to refer to the mnemonic part of an instruction when this precision is required. For instance, in the instruction LDAIM 0, the LDAIM part may be referred to as the command.

Let's look at the stages:-

PROGRAM 1.2

```
START ADDRESS? 828   Gives address for beginning of
                     program.
CLD
CLC                  Gets the 6510 ready for adding.
LDAIM 1              LoaD "1" into the Accumulator in
                     IMmediate mode.
STA   1024           STore the contents of Accumulator
                     (1) in 1024.
STA   55296          STore the contents of Accumulator
                     (1) in 55296
LDAIM 2              LoaD "2" into the Accumulator in
                     IMmediate mode.
ADC   1024           ADd Contents of 1024 (1) to the
                     contents of the accumulator (2).
STA   1026           STore the contents of the
                     Accumulator (3) in 1026.
STA   55298          STore the contents of the
                     Accumulator (3) in 55298.
RTS                  ReTurn from machine-code
                     Subroutine.
END                  End assembly.
```

Right then, let's type it in!

If you make a mistake before pressing return, you may correct the mistake using the cursor keys with the INST/DEL key as normal. However, if you press return before you notice the mistake - just type in "END" and start again.

1. Run the Assembler program if not already running.
2. Select "E" to begin entering assembly program.
3. Tell assembler where to start, i.e. type in "828" <return>
4. Type in "CLD" <return> (press <return> after each entry).
5. Type in "CLC"
6. Type in "LDAIM 1"
7. Type in "STA 1024"
8. Type in "STA 55296"
9. Type in "LDAIM 2"
10. Type in "ADC 1024"
11. Type in "STA 1026"
12. type in "STA 55298"
13. Type in "RTS"
14. Type in "END"
15. Select "R" to run program and then give start address - 828. Screen displays a white "A" in 1024 and a red "C" in 1026.

Press any key to return to menu.

If you wish to list, select "L" and then give the location "828".

Another way of looking at the three lines

```
LDAIM 1
STA 1024
STA 55296
```

is as a way of putting a "1" into memory or of printing a CBM 64 ASCII "1" in white (or an "A") on the screen.

The 6510 has two index registers in addition to its accumulator and these are referred to as Index registers X and Y and each can store one 8-bit number. The arrangement of these or, as the jargon has it, the ARCHITECTURE of the 6510 is shown below (in part) in Fig. 1.1.

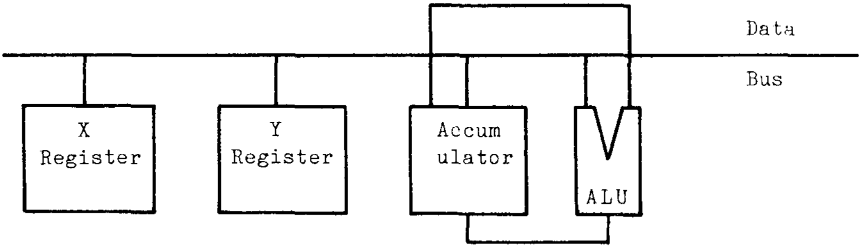


Fig. 1.1

In this figure the X and Y registers are shown identically, although they do differ slightly. Nevertheless, they are both index registers. The real advantage of index registers is that we can increment (increase by 1) or decrement (decrease by 1) the value which they contain and, in addition, we can use them to 'step through' memory, so they are very powerful as we shall see later. To the right of the figure is the 'ALU' or Arithmetic and Logic Unit which is used by the 6510 for all arithmetic and logical operations which it needs to carry out. The ALU has two inputs for the data that it manipulates and one output which feeds the result of the operation into the accumulator. Notice that almost all data flows through the accumulator and this makes the accumulator a key feature of the 6510. Data flows between the various registers along the 'Data Bus' which is a common pathway for communication **within** the 6510. For talking to devices beyond the chip this data bus is extended to access memory also.

In the remainder of this chapter, we will look at these registers and the ways that data can be fed in, out and between these.

First of all we'll have a go at using the X-register - so to load this we use the instruction

LDX	LoaD index register X with data from the specified address, i.e.
LDX 900	means LoaD index register X with the data in memory location 900.

LDX differs from the earlier "LDAIM" (apart from one loading the Accumulator and one X register) in that the LDAIM command is an Immediate Mode command. When the 6510 sees this it looks for what's immediately following the instruction and loads that - as data - into the Accumulator. With the new command above "LDX" the 6510 looks for what follows and this specifies the ADDRESS of the data. Thus with the instruction:-

LDX 900

the 6510 goes to memory location 900 to find the data which it loads into the X-register. This instruction (as are all the register instructions) is really a COPY as the data put into the X-register is COPIED from location 900. That is to say, the data originally stored in memory location 900 remains there.

To recover the data we may use the instruction:-

STX	STore the contents of register X in the specified address, i.e.
STX 1024	means STore contents of X register into memory location 1024.

Right here goes. Here's the program!

PROGRAM 1.3

```
START ADD? 828

LDAIM 1         Load '1' into the accumulator.
STA 1024        Store the contents of accumulator in
                1024.
STA 55296       Store the contents of accumulator in
                55296 to give the screen display a
                colour (1 gives white)
LDX 1024        Load into X-register, contents of
                memory location 1024 (i.e. "1").
STX 1026        Store contents of X-register in 1026
                (i.e. 1)
STX 55298       Store contents of X-register in 55298
                to make the character placed on the
                screen white
RTS             Return from machine-code subroutine.
END             End assembly.
```

At Menu select 'R' to run the program.

The screen should display "A", "space", "A" (both A's coloured white) at the top left-hand position.

By now you should be able to write simple programs so, as an exercise try the following:-

Exercise 1.1

Load the Accumulator directly with a '1', display this (a screen 'A') in 1024. Answer on page 9-1.

Remember to put a "1" into 55296 to give the display the colour white.

Don't forget to put in the "RTS" at the end. If you do forget then the 6510 will run on to see what it can find and try to execute this. If you are lucky, then the 6510 may simply return to BASIC. However, with your luck, it will probably find something that crashes the system. The crash may be recoverable, in which case pressing the RUN/STOP key together with the RESTORE key may return the CBM 64 to BASIC. If pressing the RUN/STOP and RESTORE keys together does not restore you to BASIC then the crash is not recoverable and it will be necessary to switch the CBM 64 off and on again, re-load the assembler and start again.

Exercise 1.2

Write your name in the top left hand corner of the screen. One possible answer on page 9-1.

Exercise 1.3

Put an 'X' in each of the four corners of the screen. Answer on page 9-2.

You will need to know that the screen of the CBM 64 normally occupies the 1000 memory locations from 1024 to 2023. The 1000 memory locations is needed to provided 25 rows each of 40 characters. As we have seen, 1024 is the top left hand position of the screen, and since 1026 was the third position ...

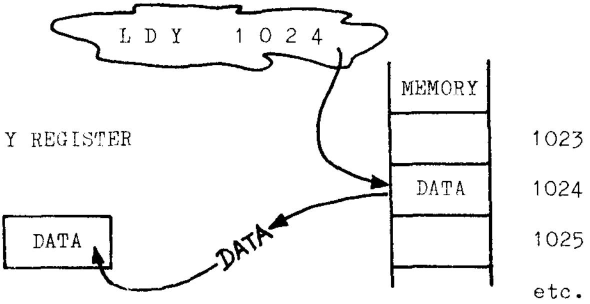
***

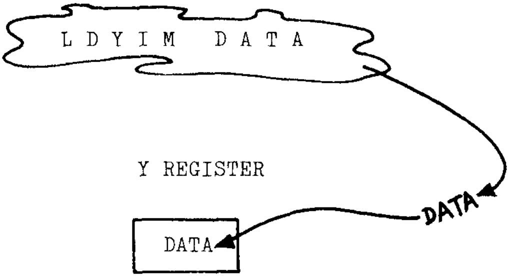
The Load and Store instructions that we have met so far are complemented by the corresponding Y-register instructions.

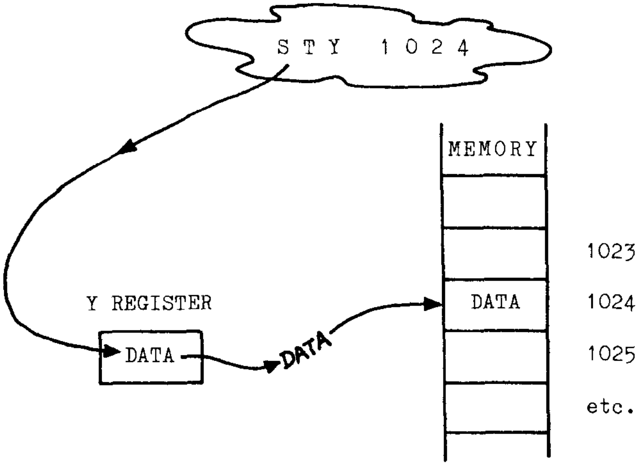
LDY LoaD register Y with data at specified address.

LDYIM	LoaD register Y with data specified in Immediate Mode.
STY	STore the data in the Y-register at specified address.

Diagrammatically, these instructions are shown below:-







You should now know, or be able to interpret the following:-

```
LDA      LDX      LDY      RTS
LDAIM    LDXIM    LDYIM
STA      STX      STY
ADC
```

For many operations, BUT NOT ALL, the X and Y registers can be treated interchangeably; for instance program 1.3 could be written:-

PROGRAM 1.3

```
START ADD? 828
LDAIM 1
STA 1024
STA 55296
LDX 1024
STX 1026
STX 55298
RTS
END
```

or

PROGRAM 1.3a

```
START ADD? 828
LDAIM 1
STA 1024
STA 55296
LDY 1024
STY 1026
STY 55298
RTS
END
```

Because of this interchangeability and the need to swap data rapidly between registers during the run of a program several instructions exist to do this automatically. They are typified by:-

TAX   Transfer the contents of the Accumulator into the index register X

Using this command in program 1.3 (to produce program 1.4), makes the program a little shorter but mangages to achieve the same result.

PROGRAM 1.4

```
START ADD? 828
LDAIM 1
STA 1024
STA 55296
TAX
STX 1026
STX 55298
RTS
END
```

When this program is run the screen should display two white "A"s, one in 1024 and one in 1026.

Descriptions given of the codes so far have been spelled out in detail. However, as you are getting more used to the jargon, it is reasonable to begin to abbreviate. From now on, instead of "the contents of the X register", we will just refer to X and similarly so with the Y-register (Y) and accumulator (A). Thus, a summary of the transfer instructions is:-

TAX	Transfer A into X.		
		TAY	Transfer A into Y.
TXA	Transfer X into A.		
		TYA	Transfer Y into A.

Exercise 1.4

Write a program that loads a "Z" into the accumulator and an "A" into the X-register. Then, without using any further Immediate Mode commands, swaps these over and prints the "Z" on the first screen memory location and the "A" on the last.
A possible answer on page 9-3.

Exercise 1.5

Write a program that: Loads a diamond into the accumulator, an asterisk into X and an "E" into Y. Then, without using any further Immediate Mode commands, moves the "E" into A, the diamond into X and the asterisk into Y. Print the diamond in the screen bottom-left, the asterisk in the bottom-right and two "E"s, one in each of the top two corners of the screen.

A possible answer on page 9-3.

Hint: There are 40 characters in the CBM 64 screen line, so the top right hand corner of the screen memory is located at (1024 + 39). I leave it to you to work out the address of the bottom left hand corner and the corresponding memory locations for the colours.

***

CHAPTER TWO

Jumping, Branching, the Program Counter

Few real life programs proceed along a smooth uninterrupted path without jumping or branching at some stage. This chapter looks at those commands and their uses and then examines the flags that enable the branches to be controlled.

Unconditional Jumps

These tell the program to jump willy-nilly - no conditions. Only two such 6510 instructions exist; the first to be considered is:-

JMP	JuMP to the specified address.

For instance, JMP 834 means jump to memory location 834.

Put in a program and it will look like this:-

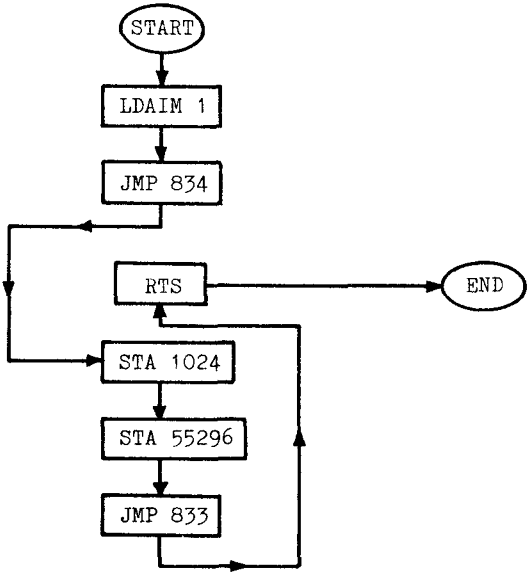


Fig. 2.1

Such a jump routine doesn't really achieve a lot but it could, for instance, be used to patch a piece of code into a program. In figure 2.1, for instance, the commands STA 1024 and STA 55296 have effectively been inserted into the program.

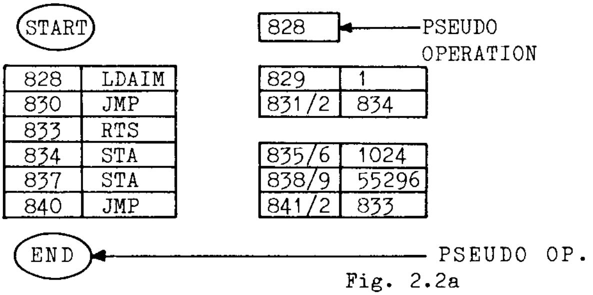
Now this can be typed in thus

PROGRAM 2.1

```
    START ADDRESS? 828
    LDAIM 1
    JMP 834
    RTS
    STA 1024
    STA 55296
    JMP 833
    END
```

Once again, it can be run by selecting 'R' on the menu and then starting the program at 828. When run, it should give a white "A" in the top left hand corner of the screen

When the jumps are used in this way it's necessary to tell the program exactly where to jump to, i.e. to give an ADDRESS, hence JMP 834. Calculating these addresses is quite straightforward as long as it is done systematically. For instance, all the commands or "SOURCE-CODES": RTS, LDAIM, JMP, etc., take up one byte of memory, thus, to jump over RTS in Fig. 2.2a we jump from 830 to 834, i.e. over 833 which contains RTS.



828	LDAIM
830	JMP
833	RTS
834	STA
837	STA
840	JMP

829	1
831/2	834
835/6	1024
838/9	55296
841/2	833

Fig. 2.2a

The part following the source-code is known as the OPERAND and calculating its length is a bit more complicated!

The easy way is simply to look it up in Appendix II. Here you will find a complete list of all the source codes available on the 6510 microprocessor. For instance, at the bottom of page A2-19 you will find the entry for the RTS instruction. Under the heading NO. BYTES OPER. (number of bytes in operand) you will find the value zero. Thus, as we know already, RTS doesn't have an operand. It doesn't need one as the address to which it points is determined by the point from which the sub-routine came originally. On page A2-13 you will find an entry for LDAIM which takes a single byte operand. The accumulator can only hold values up to 255, a single byte, in its eight bit register. There in the table for LDAIM is a

1 to confirm this. Other instructions that you have met, require two bytes as their operands are greater then 255. For instance, JMP has an entry on page A2-12 showing that it has a two byte operand, and so on. By the way, I know the list of source codes is rather frightening, but DON'T PANIC, take them as they come.

Figure 2.2a shows the location of the various instructions and operands for program 2.1.

LDAIM 1 — takes up two bytes - one for its object code A9 and one for the number to be loaded into the accumulator. Remember the accumulator is only one byte long, so it can only hold a number up to 255.

JMP 834 — takes three bytes - one for JMP (4C) and two for the address - here 834.

RTS — takes only one byte (60) - it has no operand as do TAX, TXA, etc.

STA 1024 — This takes three bytes - one for STA (8D) and two for 1024

STA 55296 — This takes three bytes - one for STA (8D) and two for 55296

JMP 833 — takes three bytes - one for JMP(4C) and two for the address - here 833.

This can also be seen by using the "LIST" command on the assembler MENU. Return to MENU and type "L" for LIST, then tell the assembler where to start listing, i.e. type in "828".

The screen will display:-

```
828 033C | A9 01    | LDAIM 1
830 033E | 4C 42 03 | JMP 834
833 0341 | 60       | RTS
834 0342 | 8D 00 04 | STA 1024
837 0345 | 8D 00 D8 | STA 55296
840 0348 | 4C 41 03 | JMP 833
```

Fig. 2.2b

As Fig 2.2b shows, the assembler breaks the instructions and operands down into one-byte chunks. We can calculate the total length of the program by counting the one-byte pieces of the machine code. Thus, program 2.1 is 15 bytes long:- A9 01 4C 42 03 60 8D 00 04 8D 00 D8 4C 41 03

***

JSR	Jump to Sub-Routine.

This is another jump command which is used along with RTS and together these are like GOSUB.....RETURN in BASIC.

```
        BASIC                   ASSEMBLER

  10    GOSUB 200                830  JSR 834
          "
          "
          "
          "
  200   REM***SUB-ROUTINE        834  STA 1024
          .    .    .          .    .    .
  300   RETURN                   840  RTS
```

We can modify program 2.1 to use this instruction instead of the straight JMP used there. The program then becomes:-

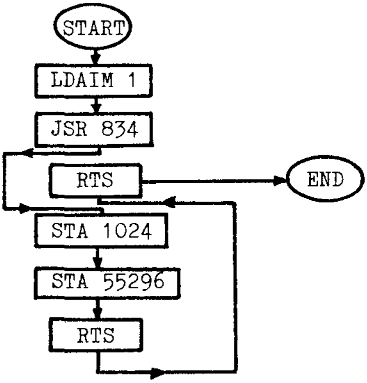


Fig. 2.3

PROGRAM 2.2

```
START ADDRESS? 828   Pseudo-code
LDAIM 1              Load Accumulator in immediate mode
                     with a '1'.
JSR 834              Jump to the subroutine at 834.
RTS                  Return from subroutine (i.e. back to
                     BASIC).
STA 1024             Store contents of accumulator in 1024.
STA 55296            Store accumulator in 55296.
RTS                  Return from subroutine.
END                  Pseudo-code
```

The advantage of RTS over JMP is shown in this program, as with RTS it is not necessary to calculate the address for the jump which organises the return to the main line of the program. In program

2.1 we had to put in the JMP 833 to have the same effect as RTS in this program. The 6510 does this trick by use of the

PROGRAM COUNTER (PC)

This is a 16-bit register which contains the address of the next command which is to be executed. In reality all it is is two 8-bit memories, one for each byte, which is built into the 6510 chip. When you select 'R' at the assembler MENU and then type in 828, this generates a command that sets PC to 828 and starts execution from there. As the PC fetches each byte from memory it is incremented by 1, thus always pointing to the next memory location containing the required data.

Take the first three lines of program 2.2 for instance:

```
START 828
LDAIM 1
JSR   834
```

A summary of the PC counter contents during execution of this is given in Fig. 2.3.

PROGRAM	PC BEFORE EXECUTION	PC AFTER EXECUTION
START 828	?	828
LDAIM 2	828	830
JSR 834	830	834

Fig. 2.3

This figure illustrates how the PC steps through the program until it comes to the JSR command. It then takes the jump command and sets the PC to the address specified, i.e. 834. As it is only two bytes long and can thus only store one address, the PC uses some external memory when it needs to remember more than one. This area of memory, the stack, is discussed below (page 7-8).

Exercise 2.1

Write a program to put a 3 in the accumulator. The program is to start at 828, then jump to a sub-routine at 900 which adds 3 to the 3 already in A, return to original routine and print the accumulator sum onto the top left-hand corner of the screen.

Answer on page 9-4.

Conditional Jumps

We have already looked at unconditional jumps but any program that needs to test for conditions needs CONDITIONAL JUMPS. In BASIC, the analogy is with the IF...THEN command,

```
i.e.    10  IF X=Y THEN 500
```

In this line the values X and Y, which have been stored in memory, are compared.

The 6510 carries out this operation in several different ways - one of these is by using a special register known as the STATUS REGISTER (SR), sometimes known as the PROCESSOR STATUS WORD. The SR is an eight bit register like the accumulator and X and Y register, but it is used quite differently from these. Whereas the other registers are used to store and manipulate bytes, the SR is treated as if it contained eight individual bits which are used as signals or flags. The 6510 normally only handles one status flag (as they are known) at a time, either setting the bit value to '0' or '1', or testing the status flag to determine whether it is set ('1') or cleared ('0').

One example of the status flags is the Z flag or the ZERO flag. Whenever an arithmetic process (or just a move) is carried out that produces a result of zero in the appropriate register (A, X or Y) then the Z flag is set to '1'. If, on the other hand the result of the process is non-zero then the Z flag is set to '0'. Look at it this way: if the flag is SET (to '1') then the condition (or status) is TRUE, if the flag is CLEAR (to '0') then the status is FALSE. The Z flag is concerned with a zero condition or status, so Z flag set means that it is true that the condition is zero.

Several different instructions can set this flag, one of these being:-

DEX	DEcrement the contents of the X register.

The segment of Program 2.3 below demonstrates this in use.

PROGRAM 2.3
(In Part)

```
START ADDRESS? 828
LDXIM 100
DEX
```

It loads X with '100' and then decrements this one down to 99. This facility for indexing both the X and Y registers accounts for their name - index registers. When the contents of X are zero, the zero flag is set to 1. If we wish to use the setting of this flag to control the program then we must use an instruction that tests the flag and brings about a branch dependent upon whether or not it is set. Such an instruction is:

BEQ Branch if result was EQual to zero. (i.e. if the ZERO flag is set.)

This checks on the status of the Z flag and if it is set (to 1), branches as specified. The operand in this case in only one byte long, so only 0 to 255 can be accommodated. As these 256 numbers are needed for branching in both directions 0 to 127 are assigned to forward jumps, for example '60' giving a forward jump of 60 steps while 128 to 256 are used for backward steps. In the case of the latter a branch instruction of, say, 200 gives a backward step of 256 - 200 or 56 steps.

The BEQ instruction is illustrated in Program 2.3 below, where it checks the condition of the Z flag, and, if set, branches forwards 3 bytes.

PROGRAM 2.3

```
START ADDRESS? 828
LDXIM 100
DEX
BEQ 3
JMP 830
STX 1024
STX 55296
RTS
END
```

When run this program prints a black "@" in 1024.

As with many X-register instructions, DEX has a corresponding Y register instruction:-

DEY DEcrement the contents of the Y register.

Exercise 2.2

Write a program to carry out the same operation as Program 2.3 but utilising the Y register. Answer on page 9-4.

A second instruction that also checks the Z flag is:

BNE Branch if Not Equal.

This does the reverse of the BEQ command and branches if the Z flag is NOT set. Program 2.4 below, is a modification of Program 2.3. Notice how the original program is shortened considerably by the use of BNE rather than BEQ.

PROGRAM 2.4

```
START ADDRESS? 828
LDXIM 100
DEX
BNE 253
STX 1024
STX 55296
RTS
END
```

When run, the program is identical in effect to Program 2.3 and puts a black "@" in 1024.

The index registers have been indexed downwards by the DEX and DEY commands. As you might expect, they can also be indexed upwards. This is done by means of:-

INX	INcrement the contents of X by one.
INY	INcrement the contents of Y by one.

Instructions to compare values

Naturally, when incrementing, a straight check for zero is not possible so the registers must be compared against a value previously set somewhere and the 6510 possesses three instructions to do this. The first of these instructions to be examined is:-

CPX	ComPare the contents of the specified memory address with the X register.

This is actually done by subtracting the memory contents from X and can thus give a positive, negative or zero value. Thus, the instruction CPX 900 does the following:-

1. Read contents of memory location 900.
2. Subtract these contents from those of the X register.
3. Set Z flag if answer=0. (Also sets other flags not yet considered.)

NOTE:

> Neither the contents of the memory location nor the X register are changed during this operation.

At the moment we are interested in the zero condition. To utilise this instruction we can set the X register at zero and store a value for comparison in memory somewhere. The flow diagram for this is given below, in Fig. 2.5.

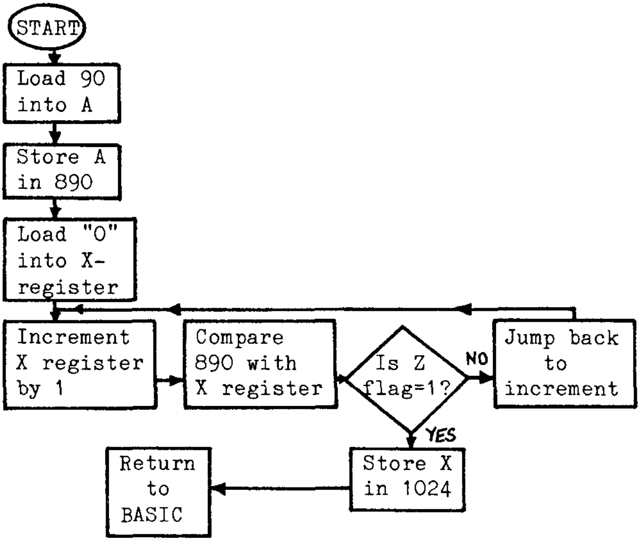


Fig. 2.5

Written into a program it looks like this:

PROGRAM 2.5

```
START ADDRESS? 828
LDAIM 90                    Load  90  (a  diamond)  into
                            Accumulator.
STA 890                     Store contents of Accumulator
                            in 890.
LDXIM 0                     Load 'O' into X register.
INX                         Increment X register.
CPX 890                     Compare  value  in  X  register
                            with that in 890 (i.e. 90).
BEQ 3                       Branch forward three bytes if
                            CPX answer=0.
JMP 835                     Jump to memory location 835.
STX 1024                    Store contents of X in 1024.
LDAIM 1                     Load a one into Accumulator.
STA 55296                   Store it in colour RAM to give
                            a white image on the screen
RTS                         Return  from  machine-code  to
                            BASIC.
END
```

Just to make that clearer, we'll step through some of the stages. Figure 2.6 below is numbered in stages, where stage 1 represents the first time that the program steps through INX, stage 2 the second time, and so on.

Note that although X is loaded with a '0', this is immediately incremented to a '1' at INX.

Loop Number	Accumulator contents	X-register contents	Z-flag
1	90	1	0
2	90	2	0
3	90	3	0
etc.			
88	90	88	0
89	90	89	0
90	90	90	1

Fig. 2.6

At stage 90, BEQ is activated and program jumps 3 bytes to STX 1024 and then RTS.

Now type in the program and run it. It should display a diamond in 1024.

The compare instruction CPX has a corresponding instruction for the Y register:

CPY   Compare the contents of specified location with those in the Y register.

Its operation corresponds exactly with that for CPX.

Exercise 2.3

Re-write Program 2.5 to use the Y-register rather than the X and on completion of the loop print out a purple heart at 1034 (you need a 4 in colour RAM to obtain a purple)

Answer on page 9-4.

The third compare instruction is:

CMP   CoMPare the contents of the specified memory with the Accumulator.

This is particularly useful as the results of all arithmetic operations are deposited in the Accumulator and CMP allows a direct comparison between a specified value and an 'answer'.

An example of its use is given in Program 2.6.

PROGRAM 2.6

```
START ADDRESS? 828
LDXIM 0                   Load a '0' into X.
LDAIM 83                  Load an '83' (a heart) into A.
INX                       Increment X.
STX 900                   Store X in 900.
CMP 900                   Compare A with 900.
BNE 247                   Branch if Not Equal.
STX 1024                  Store X in 1024.
LDAIM 1                   Load a '1' into the Accumulator.
STA 55296                 Store in colour RAM to get white
                          display.
RTS                       Return from Subroutine.
END
```

Let us have another look at program 2.5. It turns out that using CPX to put a '1' into the Z flag was a bit like using a sledgehammer to crack a rather small nut, as the Z flag is very readily set. In fact, the Z flag is set to '1' whenever a zero is passed from memory to the Accumulator, to the X or to the Y registers or when a zero answer is obtained to an arithmetic process. Thus, the program 2.5 could have been written omitting the CPX 890 instruction.

To demonstrate this, program 2.5 is re-written using the Y register and DEY and testing with the command BNE, as shown in figure 2.7 below:-

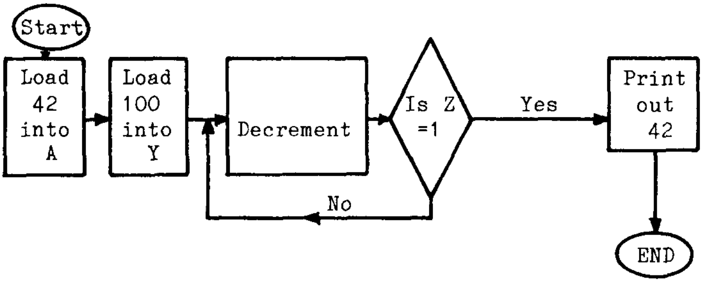


Fig. 2.7

And the program:-

PROGRAM 2.7

```
START ADDRESS? 828
LDAIM 42
LDYIM 100
DEY
BNE 253
STA 1024
LDAIM 1
STA 55296
RTS
END
```

The instruction BNE 253 gives a branch of 256 - 253, i.e. a backward branch of 3 taking the program back to DEY, as the step is counted from the beginning of the next instruction whether the branch is forward or backward.

If you haven't run program 2.7 by now, have a go, it should put a white asterisk in 1024!

Exercise 2.4

Now write the program for the flow diagram below, figure 2.8:-

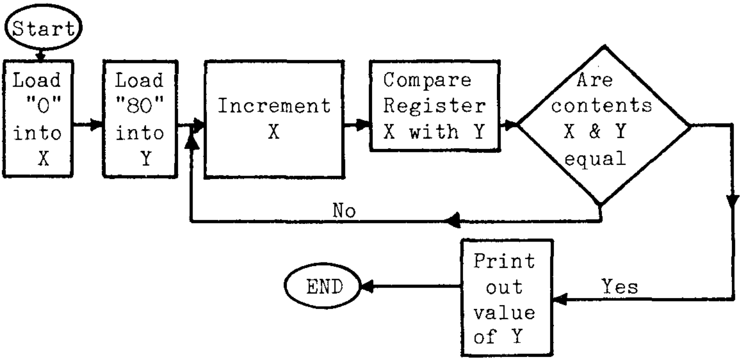


Fig. 2.8

If you run into difficulty with this, the more detailed flow-diagram on page 9-5 should help. PLEASE don't look at this until you've had a try, though.

The answer is given on page 9-5.

The 6510 Flags

Program 2.7 used the Z-flag which is only one of 7 flags available on the 6510. As these each only contain one BIT of data, i.e. a '0' or a '1', they can all be stored in one byte of memory - the Processor Status Register. Thus the flags are contained in the SR as shown below in Figure 2.8.

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

Fig. 2.8

It is not proposed to give a full description of all these now; they will, however, be described and illustrated when they are encountered. The Z-flag - the Zero flag - has, of course, already been met and used in programs and exercises.

The function of the flags is summarised below:-

N Negative flag. Set when an arithmetic operation results in a negative result. The flag is controlled by the instruction ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, TAX, TAY, TXA, TYA.

V Overflow flag. Set when an arithmetic operation results in an overflow from bit 6 to bit 7, tells that result will be wrong unless overflow allowed for. The flag is controlled by the instruction ADC, BIT, CLV, PLP, RTI, SBC.

B Break flag. Set when a programmed interrupt is brought about by a BRK instruction.

D Decimal flag. Set when arithmetic operations are to be carried out in decimal. The flag is controlled by the instructions CLD, PLP, RTI, SED.

I Interrupt flag. Set when an interrupt sequence is in operation. The flag is controlled by the instructions BRK, CLI, PLP, RTI, SEI.

Z Zero flag. Set when an arithmetic operation results in a zero answer. The flag is controlled by the instructions ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TXA, TYA.

C The Carry flag. Indicates the presence of a 'carry' or a 'borrow' during arithmetic operations. Also set during shift or rotate operations to indicate possible loss of a bit. The flag is controlled by ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC, SEC.

The N-Flag

The N-flag, bit 7 on Fig. 2.8 above, is the Negative flag which is set to '1' when the answer to an operation is negative. It can be tested by two instructions, one of which is:-

BMI	Branch on MIinus.

An instruction such as BMI 27 would test the N flag and if set branch the program forward 27 bytes.

An example of the use of BMI is given in program 2.8, where the contents of Y are incremented until a CPY command no longer gives a minus result and consequently, ceases to branch.

PROGRAM 2.8

```
START ADDRESS? 828
LDAIM 90                 Load 90 into A.
STA 900                  Store Accumulator contents
                         in 900.
LDYIM 0                  Load '0' into Y.
INY                      Increment Y.
CPY 900                  Compare contents of 900 with
                         contents of Y.
BMI 250                  Branch on minus, i.e. test N
                         flag.
STY 1024                 Store contents Y in 1024.
LDAIM 1                  Load Accumulator with '1'.
STA 55296                Store in colour RAM to get
                         white display.
RTS                      Return from subroutine.
END
```

On running, this program should put a white diamond in 1024.

—

In addition to the BMI command, the N flag is also tested by the command:-

BPL	Branch on PLus.

A command such as BPL 12 would test the N flag and if not set branch the program forward 12 bytes. Program 2.9 uses BPL to test the N flag and to branch when not set. In this program memory location 900 is loaded with 90 and Y loaded with a higher number, in this case 100. Thus, after a CPY instruction the negative flag is not set, i.e. a plus had occurred and the BPL instruction brought about a branch. As the Y register is decremented, however, the CPY test yields a minus at 90 - 91 and the Branch on PLus (BPL) test fails allowing the program to run through to the end.

PROGRAM 2.9

```
START ADDRESS? 828
LDAIM 91
STA 900
LDYIM 100
DEY
CPY 900
BPL 250
STY 1024
LDAIM 7
STA 55296
RTS
END
```

When run, this program will print 90 (a diamond) coloured yellow in 1024.

Exercise 2.5

Write a program using BPL to branch when the X register becomes negative having been decremented from 90, then print out the current value of X.

A possible answer on page 9-5.

CHAPTER THREE

More Instructions, Addressing, Screen Outputting

One of the advantages of machine code programs is their speed of operation and this naturally facilitates screen displays. Animation, for instance, can be achieved by means of commands such as:-

STAX	STore the contents of the Accumulator in the specified address indexed with the X register.

Thus, if X contains 100 and the accumulator 90, the instruction

```
STAX 1024
```

will put a diamond into (1024+100). When used along with the increment instruction, this enables the location on the screen to be indexed. Program 3.1 demonstrates this.

PROGRAM 3.1

```
START ADDRESS? 828
LDXIM 100               Load 100 into X.
LDAIM 90                Load 90 (a diamond) into A.
STAX 1023               Output diamond at (1023+X).
LDAIM 1                 Load 1 into A.
STAX 55295              Ensure colour is white.
DEX                     Decrement X value.
BNE 243                 Branch on not equal.
RTS
END
```

When run, this program puts a diamond in the first 100 locations on the screen. Once again, the 'X' command has a corresponding 'Y' command:

STAY	STore the contents of the Accumulator in the specified address indexed with the Y register.

Exercise 3.1

Modify program 3.1 to use the Y register rather than the X register, using only direct POKE commands.
Answer on page 9-6

Instead of the decrement instruction being used, the program could have used an increment but would then have had to do a compare with 100 in order to set the zero flag to a '1'. Try this as an exercise!

Exercise 3.2

Print an asterisk in the first 100 screen locations using an INX command to increment.
A possible answer on page 9-6.

—

In exercise 3.2 the branch instruction was activated by zero generated by a compare command. However, if the X or Y register is incremented from 255, it clocks back to zero and resets the Z flag. It can thus be used to branch without a compare if it is initially set to the appropriate value. Program 3.2 performs a similar function to 3.1 but uses INX rather than DEX. It increments X from 216 to 255 and on the 39 loops through, plus the next loop, prints out an asterisk. Once at 255 the 8 bit register is full of '1's and the addition of one more '1' ripples through each bit of the store, resetting them all to '0's. The 6510 does notice this switching over and sets a flag to remember the event (discussed above, page 2-13).

This program offers no advantage over 3.1 on its own but may in a particular context be advantageous.

PROGRAM 3.2

```
START ADD? 828
LDXIM 216
LDAIM 42
STAX 808
LDAIM 1
STAX 55080
INX
BNE 243
RTS
END
```

As with BASIC programs a character can be moved across the screen by filling the screen with the character while POKING a blank one space behind it. Program 3.3 demonstrates this type of routine.

PROGRAM 3.3

```
START ADDRESS? 828
LDXIM 0
LDYIM 32
STY 900
LDAIM 90
STA 901
STAX 1024
LDAIM 1
STAX 55296
TYA
STAX 1023
LDA 901
INX
BNE 238
RTS
END
```

When run the program runs a white diamond across the screen to 1279. You may be quite at home with this general technique of animation; if so you may ignore Figure 3.1 which steps through the program's instructions in the same order as does the CBM 64.

Command	Acc	X-Reg	Y-Reg	900	901	Screen mem.	Content	Z-flag
LDXIM 0	?	0	?	?	?	?	?	0
LDYIM 32	?	0	32	?	?	?	?	0
STY 900	?	0	32	32	?	?	?	0
LDAIM 90	90	0	32	32	?	?	?	0
STA 901	90	0	32	32	90	?	?	0
STAX 1024	90	0	32	32	90	1024	90	0
LDAIM 1	1	0	32	32	90	1024	90	0
STAX 55296	1	0	32	32	90	1024	90	0
TYA	32	0	32	32	90	1024	90	0
STAX 1023	32	0	32	32	90	1023	32	0
LDA 901	90	0	32	32	90	1023	32	0
INX	90	1	32	32	90	1023	32	0
BNE 238	90	1	32	32	90	1023	32	0
STAX 1024	90	1	32	32	90	1025	90	0
LDAIM 1	1	1	32	32	90	1025	90	0
STAX 55296	1	1	32	32	90	1025	90	0
TYA	32	1	32	32	90	1025	90	0
STAX 1023	32	1	32	32	90	1024	32	0
etc.	until	....						
STAX 1024	90	255	32	32	90	1279	90	0
LDAIM 1	1	255	32	32	90	1279	90	0
STAX 55296	1	255	32	32	90	1279	90	0
TYA	32	255	32	32	90	1279	90	0
STAX 1023	32	255	32	32	90	1278	32	0
LDA 901	90	255	32	32	90	1278	32	0
INX	90	0	32	32	90	1278	32	1
BNE 238	90	0	32	32	90	1278	32	1
RTS	90	0	32	32	90	1278	32	1

Fig. 3.1

As written, program 3.3. is by no means the only way of doing the job and as you no doubt observed, it is a long way from being the best; nevertheless it does flash the diamond to where it's required. Later on in this chapter we will develop a more acceptable version of the program, but in order to do this we must first look at the problems associated with...

The Timing of Programs

Program 3.3 highlights one of the problems of machine code - a few pages ago it was an advantage! - SPEED. Whereas in BASIC it's not often necessary to slow things down, that's not so in machine code. The 6510 chip takes its operating speed from an internal clock driven from a crystal oscillator, which in the case of the CBM 64 runs at 2 MHz (two MegaHertz) or two million cycles per second. Thus each cycle takes $^1/_2$ millionth of a second and speeds of operation of the various instructions will be referred to by the number of cycles that it takes for them to be carried out, or EXECUTED. Some of these operations take place entirely within the 6510 and are carried out much quicker than those which are required to retrieve data from memory. The instruction TAX, for instance, takes two cycles to execute while STAX takes six.

Clearly, a knowledge of the time taken for the instructions to be executed is important as it is this that determines the speed of operation of the program and also allows use to be made of the 2MHz clock for timing loops and delays.

Looking back at program 3.3, the time that a character remains on the screen prior to "moving on" can be calculated. Thus, the diamond appears at STAX 1024, and the next few stages are tabulated below:-

Command	Execution time (cycles)	Elapsed Time (cycles)
STAX 1024	-	0
LDAIM 1	2	2
STAX 55296	5	7
TYA	2	9
STAX 1023	5	14
LDA 901	4	18
INX	2	20
BNE 243	2	22
STAX 1024	5	27

LDAIM 1	2	29
STAX 55296	5	34
TYA	2	36
STAX 1023	5	41

So, from a diamond appearing to its being overwritten by a blank is 41 cycles or 20.5 micro-seconds. Overall, the 256 diamonds are written in about 5248 micro-seconds or 5.2 milli-seconds, much faster than the eye can follow. Indeed, since the televison screen is only scanned once every 20 milli-seconds (European PAL system) or 16.7 milli-seconds (USA NTSC system) then this is much faster than your television screen can follow.

To attain a more leisurely progress across the screen a delay could be programmed in to allow the diamond to stay in view longer. Such a device is simple in principle but may need some care when implementing in actual application. Program 3.4 below shows a simple delay loop.

PROGRAM 3.4

```
START ADDRESS? 828
LDXIM 250                 2 cycles.
DEX                       2 cycles.
BNE 253                   3 cycles.
RTS
```

This gives (ignoring LDXIM's 2 cycles) a delay of 5 cycles per loop, or 250x5=1250 cycles per execution. Even when run right through, a delay of only 625 micro-seconds is obtained and this must be augmented by nesting this loop within another one as program 3.5 shows.

PROGRAM 3.5

```
START ADDRESS? 828
LDYIM 200                 2 cycles
LDXIM 250                 2 cycles
DEX                       2 cycles
BNE 253                   3 cycles
DEY                       2 cycles
BNE 248                   3 cycles
RTS
```

A complete run of this program would run the DEX subroutine 200 times, i.e. achieving an overall delay of 200 x 625 micro-seconds, or $^1/8$ of a second.

If we wish to use the computer for precision timing, then we can clearly not ignore the odd two micro-seconds here and there and we must step very carefully through the program to ensure that we account for all parts of the program. In particular we must watch the branch instructions. For instance, the BNE in program 3.5 normally takes three cycles, i.e. when the branch succeeds. However, when the branch fails and the program runs past the instruction it takes only two cycles. Under other conditions, if the branch takes the program into another section of memory (i.e. another "page", see page 3-9 for discussion), the instruction takes an extra two cycles!

Another problem which is peculiar to the CBM 64 is that the VIC chip which controls the screen display can interfere with timing. What with its Sprites and all, the VIC chip has so much work to do that every so often it needs to take over control of the memory and when it does the 6510 simply has to wait. This means that if you really need to use the 6510 for accurate timing, it is necessary to stop the VIC chip from displaying while you run the timing code, and reinstate the display afterwards - not very difficult to do, especially in Assembler.

However, we were principally interested in delays in order to slow down our animation so let's try putting some delays into a program (3.3) to check that they really do work!

Program 3.6 uses 3.3 as a basis and inserts the delay loop illustrated in 3.4 putting 0.6 milliseconds between the appearance and disappearance of a diamond.

PROGRAM 3.6

```
START ADDRESS? 828
LDYIM       0
LDAIM       90         Set up a diamond.
STA         900
LDAIM       1          Set up colour white.
STA         901
LDAIM       32         Set up a blank.
STA         902
LDXIM       250        Set up for delay
LDA         900        Load and Display
STAY        1024       diamond
LDA         901        Load colour white
STAY        55296
DEX                    Delay
BNE         253        loop
LDA         902        Load blank into accumulator
STAY        1024       Display blank
INY                    Set up to process next
BNE         230        location on the screen.
RTS
```

Try entering this program and running it. Disappointing isn't it? The truth of the matter is that 0.6 milli-second simply isn't long enough. The television screen is only refreshed every 1/50th second (European PAL) or 1/60th second (USA NTSC) so it takes roughly 16-20 milliseconds for the screen to be scanned. If our little white diamond is only on the screen for approximately one thirtieth of that time, the chances are that only one in thirty of our diamonds is going to be seen (i.e. one or two a line). If you look carefully, and you have to look carefully, you will detect that the situation is a bit better than that, roughly three or four a line. My engineering friends tell me that this is because of something called 'interlace' and I have to believe them. In any case, it appears that the delay produced by program 3.6 is therefore about fifteen times too short.

How then can we increase the delay? The X register can only hold a maximum of 255 and we are already using a count of 250, so there isn't much scope for increasing the number of times around the loop unless we use the double loop illustrated in program 3.5. But the extent of the delay depends on the time used by the loop as well as the number of times round the loop. The DEX/BNE loop uses 5 cycles for each loop, could we make it use more? The answer is yes. If the BNE branch at the end of the delay loop went back to the LDA 900 instruction it would increase the number of cycles consumed by the loop to 23. This would mean that each of the diamond would appear on the screen for 3 milli-seconds, still not enough time for the 16-20 milli-second scan to see every diamond. Try modifying the program so that the BNE of the delay loop goes back to the LDA 900. When you run it again you should see roughly one in three of the diamonds.

We need to increase the delay by a factor of fifteen in the original program (or a factor of three in the modified version). One possible solution is to use the double loop of program 3.5. The problem is that program 3.6 uses the Y register for indexing the character across the screen and we cannot therefore use the Y register for the outer loop of the double loop, or can we? Well we can, of course. It is always possible to save the value in the Y register in memory prior to entering the delay loop and then retrieve it after the loop. When used in this way, the lack of more than two registers in the 6510 can be overcome quite readily - at the expense of a little more coding.

PROGRAM 3.6A

```
START ADDRESS? 828
LDYIM        0
LDAIM        90         Set up a diamond.
STA          900
LDAIM        1          Set up colour white.
STA          901
LDAIM        32         Set up a space.
STA          902
LDA          900        Load a diamond.
STAY         1024       Display it on the screen.
LDA          901        Load colour white.
STAY         55296      Store in colour RAM.
STY          903        Save Y register during...
LDYIM        15         Set outer loop.
LDXIM        250        Set inner loop.
DEX
BNE          253        Count down 250 times.
DEY
BNE          248        Count down 15
LDY          903        Restore Y register as screen index.
LDA          902        Load blank.
STAY         1024       Display blank.
INY                     Set up for next screen location.
BNE          219        Loop unless all done.
RTS
END
```

Enter this program and run it. Brilliant isn't it! It really demonstrates why computer games which are written in machine code are so much better than those written in BASIC. Bear in mind that we slowed the machine code down by adding 4000 delay cycles on to each 41 useful work cycles and you can imagine how much manipulation could be carried out using machine code.

Modes of Addressing

When moving a character across the screen we have used the instructions STAX and STAY as these are able to index along with the relevant register. In earlier work we used the STA instruction on its own and this is clearly a relative of STAX, STAY, etc. The difference between the two types of instruction lies in their modes of addressing and clearly the X and Y are part of this.

In fact, the STA instruction has seven varieties depending on the mode of addressing used. Thus, the address is a modifier of the command, designed to modify its function in a particular way.

The address essentially points the 6510 to a location in memory either directly or indirectly, the way it does this being determined by the particular mode of addressing used. Addressing is uniform throughout the 64K of possible memory except for the 256 locations from 0 to 255. To address these locations, only one byte is needed whereas all other locations need two. Hence, this area of memory is given a special name - ZERO PAGE - and a special mode of addressing. In fact, the whole of the memory that the 6510 can address is divided up into pages of 256 bytes and an instruction that causes operation over the boundary between pages takes an extra cycle to be executed.

In this book, addresses in Zero page memory are indicated by the use of a 'Z' on the instruction, e.g. STAZ. However, this is only a convention chosen here and is by no means a universal one. All that really matters is that the zero page STA command has the op-code of 85_{16} or 133_{10}. This is achieved with the assembler provided, by the position of STAZ in the Data statement of 133 (x 2).

Implied Addressing

This mode, sometimes also known as inherent addressing, is probably the easiest to use, as the 6510 does all the work for you!

Several instructions have already been used as TYA, TXA, RTS, in which the 6510 itself calculates the address. Basically they form two separate groups, one in which the whole instruction is executed within the 6510 itself, i.e. TYA, transfer Y to A and the other group where an external reference is necessary, e.g. RTS return from subroutine.

The members of the first group already considered are: DEX, DEY, INX, INY, TAX, TAY, TXA, TYA, while those yet to be discussed are: CLC, CLD, CLI, CLV, NOP, SEC, SED, SEI. Those in the second group are RTS (already used) and BRK, PHA, PHP, PLA, PLP and RTI.

Absolute Addressing

Instructions using this mode are among the easiest to understand as their operand is a two byte number that defines the address absolutely. Thus in program 3.6, page 3.6, the instruction STA 901 tells the X register exactly where to store its contents. Later in the same program LDA follows the same pattern.

Instructions that utilise this form of addressing are: those already met; ADC, CMP, CPX, CPY, JMP, JSR, LDA, LDX, LDY, STA, STX, STY, while those yet to be discussed are: AND, EOR, ORA and SBC.

Zero-Page Addressing

This form of addressing is really a sub-set of absolute addressing but is restricted in size of its operand to 255. The major advantage of this mode is that it executes in only three cycles compared with the four required for the absolute modes. In this book those instructions using zero-page modes are suffixed with a 'Z'. Because of the faster execution times when using page zero addresses, page zero of the CBM 64 is used by the BASIC interpreter and is, thus, not readily available for machine code programs. A few of these zero page locations are not used by BASIC and may be used by the machine code programmer, notably 251 to 254. When you get to know your way around the BASIC interpreter then you will discover that you can use a lot more. Finally, it is possible to use the zero page by relocating this page to another location in RAM but this is felt to be a procedure beyond the scope of this book.

However, even when you cannot write to a particular zero page location because it would interfere with the BASIC interpreter, you will often find it useful to read the information stored there by BASIC. Three of the locations in page zero that are useful are 160 to 162, which contain the jiffie clock that increments every 1/60th of a second. Program 3.7 is a very simple program that loads one byte of this into the accumulator and then prints it onto the screen.

PROGRAM 3.7

```
START ADDRESS? 828
LDAZ 160           Load accumulator in Zero-page
                   mode with contents of 160.
STA 1024           Store contents of A in 1024.
LDAIM 1            Load accumulator with 1.
STA 55296          Store in colour RAM.
RTS                Return from machine code
                   subroutine.
```

Because of their limited range of use, no zero-page instructions have been used to date in this book. Those that are available for use are: ADCZ, ANDZ, ASLZ, BITZ, CMPZ, CPXZ, CPYZ, DECZ, EORZ, INCZ, LDAZ, LDXZ, LDYZ, LSRZ, ORAZ, ROLZ, RORZ, SBCZ, STAZ, STXZ, STYZ.

Immediate Addressing

This mode of addressing allows a number to be loaded immediately, i.e. directly as specified, into a register or to be used directly as a means of comparison. All the immediate commands in this book are recognisable by the 'IM' in their source code. This format is not universal and some other assemblers use the format:

```
          Normal-instruction #operand

i.e.      instead of    LDAIM 90
                         LDA  #90
```

Many examples of immediate mode addresses have been seen already, for example in program 3.6, page 3.6. In this program, the Accumulator was loaded directly using LDAIM 32, and in other programs both the X register and the Y register have been loaded in the same way. However, many other instructions can be used in the immediate mode, resulting in neater programs. Program 3.8 demonstrates the use of:-

CPYIM	ComPare Y with value specified in Immediate Mode.

PROGRAM 3.8

```
START ADDRESS? 828
LDYIM 0                 Load Y with '0'.
TYA                     Transfer Y to A.
INY                     Increment Y.
STAY 1023               Store contents of A in 1023+Y.
LDAIM 1                 Load A with 1.
STAY 55295              Store in 55295+Y.
CPYIM 100               Compare Y with 100
BNE 242                 Branch if Z flag not set.
RTS                     Return   from   machine   code
                        subroutine.
```

On running, the program prints the first 100 characters of the character set in the first 100 screen locations.

Indexed Addressing

In this mode, an address is calculated using the contents of a register added to a specified address. It has been used frequently to print characters across the screen in the form STAX, STAY. In program 3.8, STAY was used in this way with the instruction STAY 1024. In some assemblers, the register which indexes the command is appended after the operand; thus the command

```
STAX 1024
```

would be written:

```
STAX 1024,X
```

When using indexed addressing, the X and Y registers behave differently and care must be taken over their use.

Both registers can be used with absolute indexed instructions, i.e. operands with two bytes.

Those indexed with X are: ADCX, ANDX, ASLX, CMPX, DECX, EORX, INCX, LDAX, LDYX, LSRX, ORAX, ROLX, RORX, SBCX, STAX.

Those indexed with Y are: ADCY, ANDY, CMPY, EORY, LDAY, LDXY, ORAY, SBCY, STAY.

N.B. Exceptions to the generality are that STY CANNOT be indexed with X.

ASL, DEC, LSR, ROL, ROR CANNOT be indexed with Y.

The source codes that can be used in zero-page mode have the suffix 'Z' added to this to identify the register used. Thus the instruction ADC becomes ADCZX. The total list of zero-page, X-indexed commands is: ADCZX, ANDZX, ASLZX, CMPZX, DECZX, EORZX, INCZX, LDAZX, LDYZX, LSRZX, ORAZX, ROLZX, RORZX, SBCZX, STAZX, STYZX.

It is, of course, NOT possible to use the two X register index commands STX and LDX with reference to X itself!

Relative Addressing

Many programs used so far have utilised relative addressing, in which a branch has been defined relative to the current position of the program, i.e. the operand expresses the desired displacement. In program 3.8 the instruction BNE 242 was used to test the setting of the Z flag and to branch when it was not set. All the branch instructions used so far utilise relative addressing and it is this group as a whole that uses this mode. The group consists of:- BCC, BCS, BEQ, BMI, BNE, BPL, BVC, BVS.

Indirect Addressing

This is by far the most complicated - and the most versatile of all the addressing modes. It gets the "indirect" in the name from the fact that the operand is a pointer and not an address. It is this pointer that directs the 6510 towards the memory location that contains the address.

However, once again, the X and Y indexing mechanisms differ considerably in operation and give rise to different sub-modes of addressing. All the instructions that utilise this kind of addressing are recognisable in the assembler as they contain either an 'IX' or an 'IY' and have a one byte operand. Because of this they can only point to locations in the zero-page and hence suffer from the same restrictions as the other zero-page commands.

Using the X-Register

With indirect addressing using the X register, the operand is indexed with (i.e. added to) the contents of the X register to produce a pointer. This location and the one after it are then examined and their contents provide addresses for the required data in the order least significant byte, most significant byte.

The technique is useful for examining one particular element in a table, the actual position in the table being set by the value in the X register. With the lack of availability of spare zero-page space on the CBM 64 the addressing mode is of limited use. However, just to demonstrate one instruction in use, program 3.9 uses:-

LDAIX   LoaD A Indirectly indexed with X.

This is used to retrieve four bytes stored in page zero from 84 - 88. This is an area of RAM used for numeric storage in program 3.9.

PROGRAM 3.9

```
START ADDRESS? 828
LDXIM 0                 Load X immediate with '0'.
LDAIX 84                Load A indirect 84+X.
STAX 1024               Store A in 1024+X.
LDAIM 1                 Load A with 1
STAX 55296              Store in colour RAM
INX                     Increment X.
CPXIM 4                 Compare X immediate with '4'.
BNE 241                 Branch if not equal.
RTS
```

When run, the program will print four characters in the first four screen locations. What these are will differ every time depending upon what BASIC had been doing earlier. When used in a program these would form two addresses in the order:

character 1	Address 1	least significant byte	(LSB)
character 2	Address 1	most significant byte	(MSB)
character 3	Address 2	least significant byte	(LSB)
character 4	Address 2	most significant byte	(MSB)

This type of addressing is known as Indexed Indirect Addressing, or perhaps more clearly as PRE-INDEXED INDIRECT ADDRESSING. As the latter name implies, the addressing is pre-indexed as the X value is added on before the 6510 picks up the address.

Instructions that use this type of addressing are:- ADCIX, ANDIX, CMPIX, EORIX, LDAIX, ORAIX, SBCIX, STAIX.

Using the Y-Register

Indirect addressing using the Y register operates somewhat differently, as the operand instruction points directly to a zero-page memory location. This contains the least significant byte of the address, the next memory location containing the most significant byte. Finally, the index register contents are added to this address to form the final indexed address. Not surprisingly, this form of addressing is referred to as POST-INDEXED INDIRECT ADDRESSING as the indexing is carried out after the address is retrieved. The BASIC interpreter and the CBM 64 operating system make extensive use of this instruction. When you have become an experienced Assembler user you will no doubt wish to look at the way the BASIC interpreter works and you will see how useful these instructions are.

Commands that use this type of addressing are:- ADCIY, CMPIY, EORIY, LDAIY, ORAIY, SBCIY, STAIY.

Indirect Absolute Addressing

This mode of addressing is used by one instruction only:

JMPIA JuMP Indirectly Addressed.

It is an absolute instruction in that the operand is a two byte address and can thus address any location in memory. However, it is indirect in that, at that location and the subsequent one it finds the address (LSB first then MSB) for the jump instruction. In the Dr Watson assembler it is recognised by the mnemonic JMPIA but some other assemblers may represent this with JMP followed by the operand in brackets, e.g. JMP (844), i.e. Jump to 844 to find LSB and 845 to find MSB of jump address.

Putting this into a program using JMPIA 844 yields:

PROGRAM 3.10 (Part)

```
DEC HEX
840 0348 JMPIA 844
 .    .   .
 .    .   .
844 034C 81
845 034D 3
 .    .   .
 .    .   .
849 0351 STA 1024
852 0354 LDAIM 1
854 0356 STA 55296
857 0259 RTS
```

This program, on meeting JMPIA 844 looks in location 844 and finds 81 and then in 845 and finds 3 and assembles these as an address 849 (3 x 256 + 81). This address is placed in the program counter which will cause the CBM 64 to jump to 849 to execute the program stored there. As program 3.10 (Part) stands it is not possible to use the assembler to put 81 and 3 into 844 and 845 as these numbers do not represent valid op-codes. The assembler will simply reject them. They can be most easily put in by loading them into a register and then transferring into memory - as is shown in program 3.10 (Whole).

PROGRAM 3.10 (Whole)

```
START ADDRESS? 828
LDAIX 90                Load A with diamond.
LDXIM 81                Load X with low byte of
STX   844               address, store in 844.
LDXIM 3                 Load X with high byte of
STX   845               address, store in 845.
JMPIA 844               Indirect jump to address.
END

START ADDRESS? 849
STA   1024              Store diamond to screen.
LDAIM 1                 Load A with 1 (white)
STA   55296             Store in colour RAM.
RTS
END
```

When run, this first loads the address 849 into locations 844 and 845. The JMPIA then retrieves these and jumps to 849 where it executes the routine stored there. This, however, is a routine and not a subroutine as it was entered with a JMPIA and not a JSR, so the RTS at 857 returns the program to the BASIC program. Once run, this program will have modified itself and put the 3 and 81 into 844 and 845.

Addressing Generally

The whole subject of addressing is clearly a complex one and one to be approached only with care. A basic rule must be to check carefully in Appendix II before using any addressing of which you are not absolutely certain. To some extent, the assembler will assist in weeding out instructions that don't exist but it can't write your programs!

CHAPTER FOUR

Adding it up, Logical Operators

On page 1-1, as early as program 1.1, a program was written to add together two simple numbers and display these. The "simplicity" of the numbers arises from the fact that they were only single digits and that their answer gave no carries. When larger numbers are added and carries arise, the 6510 handles these by use of its carry or C flag.

This is set automatically when an addition operation is carried out that brings about a carry between the two bytes of a two byte number - clear, eh? If not, please read on!

Using one byte, it's only possible to count up to 255, thus if we wish to count beyond this we have to use two bytes. These 16 bits then allow us to count up to 65536. If you have not yet read the section on binary and hexadecimal (page A1-1) then it would be well to do it now! Naturally it would be possible to hold as large a number as there are bytes free; however, we will now consider the use of two bytes which is rather fancily described as DOUBLE PRECISION operation.

If two or more bytes are to be successfully utilised to represent a single number then they must be linked from the first to the second byte by some mechanism. This is the function that the carry flag performs and its operation is tested by the instruction:-

BCC   Branch on Carry Clear.

This tests for the carry flag being clear, i.e. set to 0, and executes a branch if this is so. One precaution is always best observed when testing for this flag, however. That is to ensure that the flag is in the expected state prior to the operation that may modify it. The instruction which performs this task is:-

CLC   CLear the Carry flag.

This sets the carry flag to 'clear' or '0' and is used prior to the process that may reset this in program 4.1:

PROGRAM 4.1

```
        CLC
        LDAIM 0
        ADCIM 1
        BCC 252
        STA 1024
        LDAIM 1
        STA 55296
        RTS
```

NOTE:
As you are well aware by now, the assembler must be given a start address, i.e."START ADDRESS? From this chapter onwards this first line is omitted in the listing but must be added in.

When run, this program progressively increases the accumulator contents by 1 until at 255 the ADCIM 1 instruction flips the eight '1's over to eight '0's and sets the carry flag to 1. Thus when the accumulator is displayed with the instruction STA 1024, it is seen to contain '0' (i.e. a white @ on the screen).

—

The 6510 has a second test instruction for the carry flag, this being:-

BCS	Branch on Carry Set.

This tests for the carry flag being 'set', i.e. containing a '1', and if the test is positive, executes a branch. Program 4.2 illustrates this instruction in use:

PROGRAM 4.2

```
CLC
LDAIM 0
ADCIM 1
BCS   3
JMP   831
STA   1024
LDAIM 1
STA   55296
RTS
```

Once again, this program progressively fills all eight bits of the accumulator with '1's and finally, on flipping these over to '0's sets the carry flag and terminates the program. At the end, the accumulator contains all '0's and therefore a white '@' is displayed on the screen.

Let's have a go at adding together two numbers larger than 256, we'll take 1157. Firstly we have to calculate the MSB and LSB and to do this, of course, we have to convert the number to a hexadecimal format. Thus:-

INT (1157/4096)	= 0	therefore Right-most character	= 0
INT (1157/256)	= 4	therefore 2nd character	= 4
INT (1157-4x256)/16)	= 8	therefore 3rd character	= 8
(1157-4x256-8x16)	= 5	therefore 4th character	= 5

Thus:

1157_{10} = 0485_{16}

and its two parts

MOST SIGNIFICANT BYTE	LEAST SIGNIFICANT BYTE
04_{16}	85_{16}

To add together two 1157_{10}'s we must first add the LSB's, check if there is a carry and then add the MSB's taking into account the necessity (or otherwise) of a carry, i.e. add LSB's.

```
                85
               +85
plus carry      0A
```

$$\begin{array}{cc} 8 & 5 \\ +8 & 5 \\ \hline 16_{10}, & 10_{10} \end{array} = \text{carry} + 0A_{16}$$

Then add the MSB's

```
                04
               +04
                08
```

Next add in the carry

08 + carry = 09

In the explanation we have glibly said "+carry" and it is this operation that the C-flag does for the programmer. The flag is set to a '1' when an operation is carried out that leads to a "carry". The next operation that is carried out then takes account of this carry and adds 1 on to the next "add" carried out.

With C flag = "0"	With C flag = "1"
04 + 04 = 08	04 + 04 = 09

Thus the answer to the example is, in Hex 090A, or in decimal $9 \times 256+10 = 2314_{10}$.

Now let's try doing that the long way - using the computer!

We can rely on the 6510 handling the carry but we can't rely on it knowing when to execute a carry! All double precision work is carried out least significant byte (LSB) first as it is during this addition operation that the carry arises and it is then stored up ready for the most significant byte (MSB) addition when this is done. You may remember that when the 6510 is using indirect addressing commands it stores the LSB of the address first and the MSB second - this is the order that they are used when the index is added on to the 'pointer' address. We could, if we so wished, stick to this organisation ourselves when we are storing 'numbers' (as distinct from addresses). But since we are doing the organising of the way in which the two bytes are added (not the 6510), there is no real advantage in storing 'numbers' either way (BASIC stores it's integers MSB followed by LSB).

In order to ensure that the LSB addition is not upset by a carry, it is most important that we preface the addition of the LSB's by the clear carry (CLC) instruction.

Firstly we must work out the value of MSB and LSB in decimal, as both methods of putting data into memory that we have discussed to date require this. For the LSB, its decimal value is

$$8 \times 16 + 5 = 133_{10}$$

and for MSB it is

$$0 \times 16 + 4 = 4_{10}$$

Now to write the program - but before we do that, let's just introduce a new instruction:-

NOP   No OPeration

When the 6510 meets it, it does nothing for two cycles of its operation.

We'll see why we put it in shortly; for now, type in the program.

PROGRAM 4.3

```
CLC                     Clear carry flag.
CLD                     Explained later.
LDAIM 133               Load A with LSB.
ADCIM 133               Add with carry 2nd LSB.
STA   1026              Store sum of LSB's in 1026.
LDXIM 1                 Load 1 for white display.
STX   55298             Store it in colour RAM.
NOP                     Do nothing.
LDAIM 4                 Load A with MSB.
ADCIM 4                 Add with carry 2nd MSB.
STA   1024              Store sum of MSB's in 1024.
STX   55296             Store 1 in colour RAM.
RTS                     Return from machine code
                        subroutine.
END
```

Having done this you may run the program, it should put I J on the screen.

The output means: I or 9_{10} and J or 10_{10}

i.e. $90A_{16} = 2314_{10}$

Stepping through that program, the various stages are:

(? signifies Random value.)

Stage	Accumulator	X Reg	1026	1024	C Flag
START 828	?	?	0	0	?
CLC	?	?	0	0	0
CLD	?	?	0	0	0
LDAIM 133	133	?	0	0	0
ADCIM 133	133	?	0	0	1
STA 1026	133	?	10	0	1
LDXIM 1	133	1	10	0	1
STX 55298	133	1	10	0	1
NOP	133	1	10	0	1
LDAIM 4	4	1	10	0	1
ADCIM 4	9	1	10	0	0
STA 1024	9	1	10	9	0
STX 55296	9	1	10	9	0
RTS	9	1	10	9	0

Fig. 4.1

As the above chart shows, at the ADCIM 133, a carry is generated and the C flag is set to '1' and this carry affects the subsequent ADC. The other thing to notice is that at the ADCIM 4 a further carry is not generated and the C flag is therefore cleared to '0'. If you want to check this you could replace the NOP command with CLC which would clear the flag after it has been set and notice that the answer that you get would be 'incorrect'.

This can be done by POKEing into 842 the code for CLC, i.e. 24:

PROGRAM 4.4

```
          POKE 842,24
```

Now, when run, program 4.3 modified by 4.4 will give a display of:-

```
          H J
```

In this run, the value of J has been computed and when its value of 266 overflowed, 256 was carried, the carry bit set and 10 was stored in the accumulator. However, this 256 was lost when the carry bit was later cleared by CLC (Program 4.4).

Using Hexadecimal Inputs

I'm willing to bet that you were somewhat horrified by the messy conversions into and out of hexadecimal which were carried out when preparing to write program 4.3. However, the assembler possesses a few lines that save at least one of these tasks, the calculation from hexadecimal back to decimal. Thus, any operand can be entered in hexadecimal, providing it is preceded by a '$' sign. Utilising this in program 4.3, this becomes:

PROGRAM 4.3a

```
          CLC
          CLD
          LDAIM $85
          ADCIM $85
          STA   1026
          LDXIM 1
          STX   55298
          NOP
          LDAIM $04
          ADCIM $04
          STA   1024
          STX   55296
          RTS
          END
```

When program 4.3a is run, it gives exactly the same result as program 4.3, i.e. I J. Select the list option and you will see that the assembler has converted the $85 into a decimal value and POKEd this into memory.

Exercise 4.1

Using hex inputs, add together 1807_{16} and $2AFA_{16}$. Verify your program afterwards in base 10. Answer page 9-7.

To recap the use of the C flag; it stores the fact that an addition operation has occurred that yielded a carry. If the C flag is left set by mistake then this carry will be added on to the next ADC operation whether you want it or not. For the carry to be passed from the LSB to the MSB you have to arrange the ADC's to be done in the appropriate order.

The 6510, of course, has an instruction that carries out subtraction with carry; this is:-

> SBC   SuBtract from the accumulator with Carry, the data at the specified memory location.

e.g.   SBC 891

means look in memory location 891 and subtract the number that you find there from the number in the accumulator.

However, in the same way as it was necessary to prepare the carry flag for addition by clearing it to '0', it is also necessary to prepare it for subtraction. Not unexpectedly, though, it is necessary to set it, rather than clear it i.e. to '1' rather than to '0'. This instruction which sets the carry flag is:-

SEC	SEt the Carry bit to '1'.

Let's look at that in a simple program to take 2 from 4. For a change we will use the immediate mode to load the data.

PROGRAM 4.5

```
        SEC
        LDAIM 4
        SBCIM 2
        STA 1024
        LDAIM 1
        STA 55296
        RTS
        END
```

By now you should be well enough equipped to try a double precision subtraction on you own so have a go at the exercises below - in case you have a problem exercise 4.1 is explained in fair detail to demonstrate the carry operation. (page 9-?)

Exercise 4.2

Write a program to subtract 600 from 800 using ABSOLUTE ADDRESSING. Store data in memory locations 890 onwards. Display the answer in 1034.
Answer on page 9-8.

Exercise 4.3

Write a program to subtract 500_{10} from the sum of 300_{10} and 400_{10}.

Display the answer in 1040/1 in the order LSB/MSB.

Answer on page 9-8.

Multiplication

The arithmetic instructions available for the 6510 allow additions and subtractions to be carried out but no provision is made for multiplication. This has to be carried out, therefore, by a series of repeated additions. For example, the sum 2x3 can be expressed as 2+2+2, and is thus relatively simple to evaluate. The process is thus one of adding 2 to the accumulator three times and requires the three to be set up in a loop to define the number of passes through this. Of course, the accumulator should contain zero before we do the adding.

The process is illustrated in Fig. 4.2 below, where Y = number of passes through the loop (in this case 3), and A = current sum.

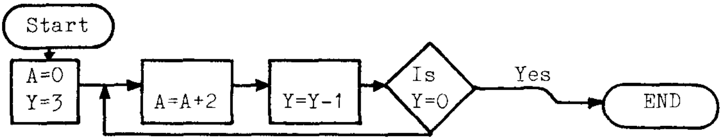


Fig. 4.2

It is put into program 4.5 below:

PROGRAM 4.5

```
          CLC
          LDYIM 3
          LDAIM 0
          ADCIM 2
          DEY
          BNE   251
          STA   1024
          LDAIM 1
          STA   55296
          RTS
```

When run, the program gives an 'F' (a CBM 64 '6') in 1024. Within this program, the key segment is

PROGRAM 4.5a

```
          ADCIM 2
          DEY
          BNE 251
```

This small loop that does the work is known as an ALGORITHM. One limitation of this simple algorithm is that it can only handle an answer up to 255 - after that, it generates a carry and clocks the accumulator back to zero. if no account is taken of this carry, then the answer has lost 256 for each loop! When elaborated somewhat, this algorithm can handle double precision multiplication. This can be achieved by checking for a carry after each addition and if a carry is generated, adding one onto the MSB. One way of carrying out this incrementing is by means of the instruction:-

INC   INCrement the contents of the specified memory location.

Program 4.6 shows the algorithm 4.5a elaborated to record the number of carries generated and to increment the MSB to record these.

PROGRAM 4.6

```
          LDYIM 0      } Set up 905 to receive carry bits.
          STY   905    }
          LDYIM 17       Set up number of times around loop.
          LDAIM 0        Clear accumulator to receive sum of LSB's.
     ---->CLC            Clear carry prior to adding.
    |     ADCIM 16       Add 16 to A once.
    |  ---BCC   3        If no carry generated then skip...
    | |   INC   905      Increment carry record.
    |  -->DEY
     -----BNE   245      Check if Y decremented to zero.
          STA   1026     Display LSB sum.
          LDXIM 1        Load 1 in for white display.
          STX   55298    Store it in colour RAM.
          LDA   905    }
          STA   1024   } Display MSB.
          STX   55296  }        in white.
          RTS
```

When run, this should display white A P or 256+16, i.e. 16x17.

In the example used - 16x17 - it made little difference whether the algorithm was arranged to add 16 seventeen times or to add 17 sixteen times. If the sum 2x100 was attempted, however, then it would obviously be much quicker to add 100 twice rather than to add 2 one hundred times. This could be tackled by writing a short subroutine that ensures that the multiplier is provided with the smallest of the two values, and thus the algorithm is cycled the least possible number of times. However, on page 4-?? another method of multiplication will be introduced - binary multiplication - and this keeps the number or iterations down to the minimum.

Division

Division using the 6510 is a process of repeated subtraction in the same way that multiplication is one of repeated addition. It is illustrated in program 4.6a in which 30 is divided by 2. In this the accumulator is used to store the running remainder, i.e. starts with 30 and progressively declines to zero (30,28,26...4,2,0). The X register is used to load the divisor into memory while the Y register counts the number of times that the subtraction can be made.

PROGRAM 4.6a

```
LDYIM 0
LDXIM 2
STX   900
LDAIM 30
SEC
SBCIM 2
INY
CMP 900
BCS 247
STY 1024
LDXIM 1
STX 55296
STA 1026
STX 55298
RTS
```

When run, this displays the quotient 15 (as a white letter O) in 1024 and the remainder 0 (as an @) in 1026.

Binary Coded Decimal Arithmetic

In addition to numbers being represented in binary and decimal notation, a hybrid or mixed notation, binary coded decimal or BCD, also exists. The usage and format of this is described in Appendix 1, page A1-7. BCD forms a bridge between the two notations and in many cases greatly facilitates output. Fortunately for us, the 6510 chip can handle BCD arithmetic and is turned on to the BCD handling mode by the instruction:-

SED	SEt Decimal mode of operation.

This instruction sets the D flag automatically to a '1' and thereafter, arithmetic is done in BCD. When the decimal mode of operation is no longer required it is cleared with the instruction:-

CLD	CLear the Decimal flag.

This sets the flag back to a '0' and thereafter, arithmetic is done in binary. You may remember, that we have used the CLD instruction on a number of occasions, prior to using the ADC and SBC instructions. It should be clear to you now, that we were ensuring that the 6510 carried out binary arithmetic.

A simple program to add together 1 and 2 using BCD is given in program 4.7.

When run, program 4.7 puts a 'C' in 1024. It is usually considered to be good practise to clear the decimal flag after doing any BCD arithmetic, since most arithmetic is done in binary - hence the CLD just before the RTS.

The example given in program 4.7 is identical in effect to the arithmetic that we have done to date, with the exception that in BCD the carry occurs after each half-byte exceeds 9. This is demonstrated below in program 4.8, which adds together two 6's. If program 4.7 is still in 828 then 4.8 can be POKEd in by:

```
POKE 831,6
POKE 836,6
```

```
PROGRAM 4.7                PROGRAM 4.8

          SED                      SED
          CLC                      CLC
          LDAIM 2                  LDAIM 6
          STA   900                STA   900
          LDAIM 1                  LDAIM 6
          ADC   900                ADC   900
          STA   1024               STA   1024
          LDXIM 1                  LDXIM 1
          STX   55296              STX   55296
          CLD                      CLD
          RTS                      RTS
```

When run, program 4.8 puts a white R in 1024, which is its way of saying 12! This comes from the way that BCD is stored in memory as NYBBLES (a nybble is half a byte - ouch!! - don't blame me, I didn't invent the word). The 'R' itself comes from the CBM 64's poke code of 18, which in binary is:

$$18_{10} = 00010010_2$$

However, the memory location is storing two nybbles rather than one byte and hence:

0	0	0	1	0	0	1	0	really means	0	0	0	1		0	0	1	0
										$=1_{10}$					$=2_{10}$		

Fig. 4.3

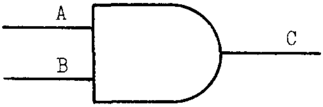
i.e. the number represents 1x10+2x1, or 12_{10}.

The above example emphasises the problems that arise when thinking in decimal and working in binary. As the 6510 itself works in binary and won't easily be persuaded to change, it is necessary to master the techniques of binary number processing.

If we look at the answer to program 4.8, in which the answer consisted of two nybbles held within one byte, we see one of, the problems of bit manipulation. What was needed in this case was a technique for modifying individual bits within a byte. In order to extract the lower order nybble from the binary number it is only necessary to erase the higher order nybble, i.e. to fill it with 'O's. This can be done with an instruction:-

AND	Perform a logical AND between the accumulator

An AND is a logical operator that compares two logic states and produces an output based on the comparison. If we examine a logic AND gate such as is used in electronic circuitry, it makes the AND function clearer.



An Electronic AND

Fig. 4.4

Figure 4.4 shows an AND gate with two inputs A and B, and an output C. Its function is such that if both inputs, A AND B, are set at '1', then its output, C, is a '1'. However, if either or both of its inputs A AND B are 'O' then its ouput is 'O'.

This is normally expressed in what is known as a TRUTH TABLE, that for the A, B, C AND gate in fig. 4.4 being shown in fig. 4.5.

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

Truth Table for Electronic AND

Fig. 4.5

To use the table, the value of C - the output - is read off for the appropriate inputs of A and B. Thus, taking an A input of 'O' and a B input of 'O' the output C is O.

Exercise 4.4

Using the truth table, fig. 4.5, work out the logic output (C) obtained for the following inputs:-

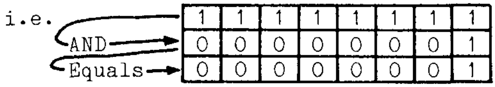
A=1 AND B=O
A=O AND B=1
A=1 AND B=1

Answer on page 9-9.

When an AND is performed by the 6510, it operates on all eight bits in the accumulator simultaneously. Thus if 255 is ANDed with 1 then:

$$255_{10} = 11111111_2$$

$$1_{10} = 00000001_2$$



i.e.								
	1	1	1	1	1	1	1	1
AND	0	0	0	0	0	0	0	1
Equals	0	0	0	0	0	0	0	1

Fig. 4.6

The result of the operation is that all the bits ANDed with a '0' have been stripped off leaving only the first bit.

Exercise 4.5

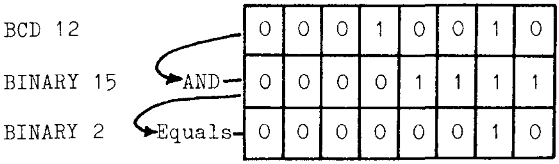
What is the result when 149_{10} is ANDed with 52_{10}?

If you have problems, work it out bit by bit using the truth table.

Answer on page 9-9.

As the above exercise shows, the AND instruction can be used to strip BITS from a number and could be used to convert part of the BCD 12 from program 4.8. This BCD '12' was stored as two nybbles in one byte. If the Most Significant Nybble (MSN) could be changed into four zeros then the byte would read out directly as the value of the Least Significant Nybble. Such a masking out of bits can be done by using an AND command, as any '1' in the ANDing number will leave '1's in the number ANDed as they were while '0's in the ANDing number will switch any '1's in the ANDed number to '0's.

Let's try that with the BCD 12.



BCD 12		0	0	0	1	0	0	1	0
BINARY 15	AND	0	0	0	0	1	1	1	1
BINARY 2	Equals	0	0	0	0	0	0	1	0

Fig. 4.7

By ANDing the BCD number with 00001111_2 (binary 15_{10}), the four most significant bits have been erased and the number converted into the LSN (in this case 2_{10}).

In a program the AND instruction may be used with several different addressing modes, the absolute mode being illustrated in:

PROGRAM 4.9

```
        LDXIM 15      Load X with '15'.
        STX   900     Store X in 900.
        LDAIM 18      Load A with '18'.
        AND   900     AND A with 900.
        STA   1024    Store A in 1024.
        LDXIM 1       Load 1 for white display.
        STX   55296   Store in colour RAM.
        RTS
```

When run, this will display a white 'B' in 1024.

Using immediate addressing, ANDIM, program 4.9 can be re-written as below:

PROGRAM 4.9a

```
        LDAIM 18
        ANDIM 15
        STA   1024
        LDXIM 1
        STX   55296
        RTS
```

When defining the truth table of the AND instruction it is written somewhat differently from that used in electronics. This is because the result of the AND operation is deposited back in the accumulator, i.e. this forms both part of the input and the output. The truth table for AND is given in figure 4.8 below:

Data for ANDing

Accumulator contents

D A	0	1
0	0	0
1	0	1

Truth table for AND

Fig. 4.8

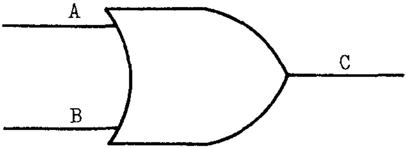
The 6510 chip also uses two other logical operators, one of these being an OR function. With the Mnemonics used in this book it is referred to as ORA.

i.e. ORA(ccumulator)

It is defined formally as:-

ORA	Perform a logical inclusive OR between the Accumulator and the data specified.

In electronic circuitry, the OR is depicted as shown in Fig. 4.9.



An Electronic OR

Fig. 4.9

Its mode of operation is that if a '1' is present on A OR B then the output C is set to a '1'. This is a bit like an AND in reverse - the AND gives a '1' only if both inputs are '1', while the OR gives a '0' only if both inputs are '0'. The truth table for the 6510's ORA command is given below on figure 4.10.

D A	0	1
0	0	1
1	1	1

Truth table for OR

Fig. 4.10

In action, the ORA command has the following effect:

Binary of 149_{10}	1	0	0	1	0	1	0	1
Binary of 52_{10}	0	0	1	1	0	1	0	0
Binary of 181_{10}	1	0	1	1	0	1	0	1

ORA

GIVES

Fig. 4.11

Putting that into a program gives:

PROGRAM 4.10

```
          LDAIM 149
          ORAIM 52
          STA   1024
          LDXIM 1
          STX   55296
          RTS
```

When run, this program will put a 181 (a white reverse-field 5) in 1024.

As with AND, ORA has several modes of addressing to suit different applications.

Third among the logical operators is:-

EOR	Perform a logical Exclusive OR between the accumulator and the data specified.

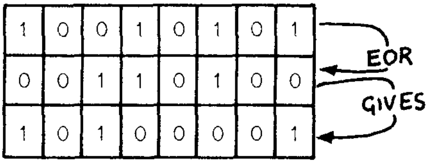
This operation is probably the least easy to understand and is best illustrated by means of the truth table, fig. 4.10.

A \ D	0	1
0	0	1
1	1	0

Truth table for EOR

Fig. 4.12

One way of expressing the function is that the output will be '1' if either of the inputs is '1' but not both. Using this instruction with the above example, i.e. 14961_0 EORed with 52:



Binary of 149_{10}	1	0	0	1	0	1	0	1
Binary of 52_{10}	0	0	1	1	0	1	0	0
Binary of 161_{10}	1	0	1	0	0	0	0	1

Fig. 4.13

The program to demonstrate that is below:

PROGRAM 4.11

```
LDAIM 149
EORIM 52
STA   1024
LDXIM 1
STX   55296
RTS
```

When run, the program will put a 161 (a white reverse field "!") in 1024.

EOR, like the other logical operators, has several modes of address to facilitate its use in programs.

Exercise 4.6

Calculate the result of the following logical operations:-

(i) 100_{10} ANDed with 87_{10}.

(ii) 75_{10} ORed with 27_{10}.

(iii) 99_{10} EORed with 57_{10}.

(iv) 94_{10} EORed with the result of 100_{10} ANDed with 87_{10}.

Write a program to verify each operation.

Answers on page 9-9.

Other Forms of Bit Manipulation

Other 6510 instructions exist that enable one to manipulate bits within a byte and as a group these lead to the movement of bits to the right or left within the byte itself.

In the earlier example using BCD arithmetic, the AND instruction was able to isolate the LSN from the byte. It was not possible, though, to extract the MSN using the available logic. This does become possible, however, using one of the bit manipulation commands:-

LSR	Logical Shift of the specified contents one bit to the Right.

When this is done, the bits are mo ed along one place to the right with the right-most bit falling off into the carry and a '0' being put into the left-most bit. Thus the LSR command operating on 149_{10} gives the following:

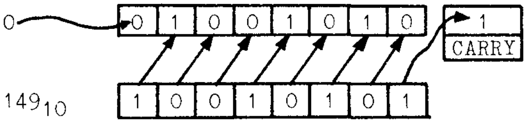


Fig. 4.14

As with most 6510 commands, LSR has several addressing modes and the particular address used informs the 6510 where the data which is to be shifted resides. Thus:

```
LSRA      means logical shift right of the data in A.
LSR 900   means logical shift right of the data in 900.
```

Using the Accumulator mode LSRA to prove the above example is:

PROGRAM 4.12

```
          LDAIM 149
          LSRA
          STA   1024
          LDXIM 1
          STX   55296
          RTS
```

When run, this will put a 74_{10} (a white ╮ in 1024)

By using four such right shifts the MSN nybble in a byte can be moved into the place of the least significant nybble and the left--most four bits filled with 0's. This enables one to isolate the MSN in a BCD calculation.

This is demonstrated in program 4.13, this time using LSR in its absolute mode.

PROGRAM 4.13

```
          LDYIM 18     } Load '18' into 900.
          STY   900    }
          LDYIM 4        Set loop counter.
    ┌────►LSR   900    } Shift 900 four
    │     DEY          } places to
    └─────BNE   250    } right.
          LDA   900    } Print out contents
          STA   1024   } of 900
          LDXIM 1      } in white.
          STX   55296
          RTS
```

When run, the program will print a 1 (a white 'A') in 1024.

Exercise 4.7

Suppose that the answer to a problem in BCD is 86_{10}.

Write a machine-code program to decode this and display the answer in decimal (POKE form, i.e. characters) in 1024 and 1025.

A possible answer on page 9-11.

A further 6510 instruction mirrors LSR in that it moves the bits to the left; it is:-

ASL	Arithmetic Shift Left: Shift the specified contents one bit to the left.

The left-most bit is shifted into the carry bit and bit 0 is loaded with a '0'. An ASL instruction, operating on 149 gives:-

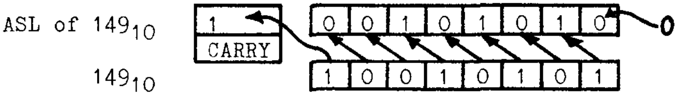


Fig. 4.15

Using the various modes of addressing, ASL can operate on data in different locations, e.g.

```
ASLA      means Arithmetic Shift Left on data in A.
ASL  900  means Arithmetic Shift Left on data in 900.
```

Using the accumulator mode to test the above example is:

PROGRAM 4.14

```
LDAIM 149
ASLA
STA   1024
LDXIM 1
STX   55296
RTS
```

When run, this puts a 42 (a white asterisk) in 1024.

Binary Multiplication

We have seen from programs 4.5 and 4.6 that multiplication can be carried out using a repetitive, or RE-ITERATIVE, process but we have also seen that this is a lengthy process, and at times very tricky. However, we tackled the problem very much from a point of view of conventional arithmetical processes and binary has its own way of doing these things! As the 6510 itself thinks in binary and has a number of instructions for manipulating the bits within its bytes, binary arithmetic has a lot to offer.

Firstly, let's examine our way of doing decimal (conventional) arithmetic. Take the sum 13x14. We define 13 as the MULTIPLICAND and 14 as the MULTIPLIER and lay the multiplication out as below:-

```
  13    Multiplicand
X 14    Multiplier
----
  52
 130
----
 182
----
```

Fig. 4.16

In this conventional format, we first multiply the multiplicand by the lowest digit of the multiplier and store this as the first partial product, i.e. 4x13=52. Next, we multiply the multiplicand by the second digit of the multiplier, i.e. 1x13, and then multiply this by 10 to obtain the second partial product, i.e. 13x10=130. Thus the total answer is the sum of the two parts, i.e. 52+130=182. Actually, in the second stage it would have been more correct to say that we multiplied the multiplicand by the second digit of the multiplier and then multiplied the result by the BASE (which happened to be 10). The total answer was then the sum of the two partial products.

It is quite possible to perform the same multiplication process using numbers in binary format.

For example, to multiply 5 x 7 in binary,

5_{10} = 0101 and 7_{10} = 0111 (working only to 4 bits)

$$\text{i.e.}\quad \begin{array}{r} 7 \\ \underline{\times\ 5} \end{array} \quad = \quad \begin{array}{r} 0111 \\ \underline{\times\ 0101} \end{array}$$

Partial Product 1	0111	Current total 1 =	0111
Partial Product 2	00000	Current total 2 =	0111
Partial Product 3	011100	Current total 3 =	011011
Partial Product 4	0000000	Current total 4 =	011011

ANS = 100011 = 32 + 2 + 1 + 35_{10}

Fig. 4.17

With all digits in the multiplier equal to one, all the significant bits in the partial products parts have the same pattern of digits as the multipliand, i.e. "111". Thus the multiplication process in binary reduces to one of successive addition following movement to the left of multiplicand.

8-Bit Multiplication

The block diagram for this process is given in figure 4.18, where Answer=ANS, D=Multiplicand, R=Multiplier, N=current bit number, LSB=least significant bit of multiplier.

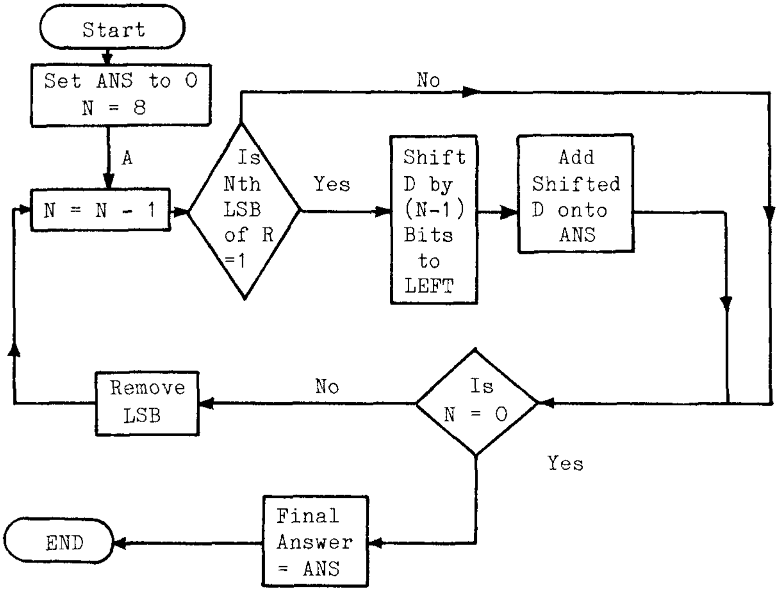


Fig. 4.18

Working through this flow diagram for the example of $2_{10} \times 2_{10}$

$$D = R = 00000010_2$$

Successive passes past A are referred to as A1, A2, etc. For this very simple example with no carries, only four passes through the loop will be made (two would suffice as only the two right-most digits are significant.

START	ANS = 0	N = 0	
A1	N = 1	1st LSB = 0	N = 8
A2	N = 2	2nd LSB = 1	
	Shift R (N-1) bits (i.e. 1) to left, i.e. $0010_2 \longrightarrow 0100_2$		
	add R to ANS, i.e. ANS=$0+0010_2=0010_2$		
	N = 8		
A3	N = 3	3rd LSB = 0	N = 8
A4	N = 4	4th LSB = 0	N = 8

The process then carries on uneventfully for the remaining four zero bits, adding nothing to the ANS.

Thus the answer = 0010_2 = 4

Putting this into a program gives:

PROGRAM 4.15

```
        LDXIM 2
        LDYIM 8
        STX   901      Initialisation
        STX   902
        LDAIM 0

        CLC

        LSR            Check if multiplication necessary.
        BCC   4

        CLC             'Multiply' if necessary.
        ADC   901

        ASL   901
        DEY            Check if all bits checked.
        BNE   240

        STA   1024
        LDXIM 1
        STX   55296
        RTS
```

When run, the program will print a white D (4) in 1024. It may be checked by changing the LDXIM instruction to input a different multiplicand and multiplier.

Exercise 4.8

Re-write program 4.15 so as to multiply two different numbers together.

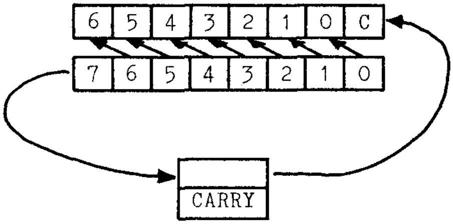
One possible answer on page 9-12.

Unfortunately, program 4.15 is ony really half true as an eight bit multiplication routine and works only over a range of small numbers for the multiplier and multiplicand. In the full routine, the ASL instruction multiplies the multiplicand by the base eight times. Thus, by the eighth shift the right-most bit would have fallen off the left-hand end of the register. In fact, with the number 2_{10} worked through in the example, the second bit (representing 2^1, or 2_{10}) would be lost after seven shifts. However, this did not affect the overall result because after two LSR's of the 00000010, all the "1"s had been cleared and all subsequent partial sums were equal to zero.

Were program 4.15 to be used with a larger number where the final answer involved a carry (i.e. was greater than 255), then this loss of left-most bits would be significant and the answer obtained would be wrong. Fortunately, when the left-most bit falls out of a register during an ASL it does not just fall into space but is caught in the carry. Thus the problem is one of retrieving this and transferring it to the MSB of the answer. This can be achieved by means of the command:-

ROL	ROtate Left the contents of the specified address.

In this operation, all the bits of the specified address are rotated left, with the carry bit being loaded into the right-most bit and the left-most bit being transferred to the carry.



The ROL Operation

Fig. 4.19

As the rotation involves the carry bit, it is known as a 9-bit rotation and provides a means of picking the carry bit back up again. Using ROL enables one to write an eight bit multiplication accommodating the carry but produces a rather complex program. Such a program is listed in chapter 5 (page 5-7) as, to make it clearer to follow, both labels and memory labels have been used.

The instruction ROL has a right-handed colleague:-

ROR	ROtate Right the contents of the specified address.

Both these instructions can be addressed in several ways and take forms as:-

ROLA	ROtate Left the contents of the Accumulator.
RORA	ROtate Right the contents of the Accumulator.

Such forms will be described when utilised.

One other instruction is available for bit manipulation:-

BIT	AND specified content's BITs with accumulator.

Thus, BIT 900 performs a logical AND between the bits in the specified memory location, i.e. 900, and the accumulator. It also has a zero-page version, BITZ.

While BIT performs the same logical function as AND, it differs in that it leaves both accumulator and memory as they are. It does, however, modify the relevant flags in the PSW in the following way:

The Z flag is set if the result of the ANDing is a zero (and cleared, of course, if the result is not zero).

The N flag is affected as follows:- bit 7 of the location being tested is copied to the Processor Status register (bit 7 of the PSR being the N flag). This is a very convenient way of testing whether the contents of a particular location are positive·or negative without the necessity of loading the value into one of the registers.

The V flag (which we haven't really discussed yet) is bit 6 of the Processor Status register. The BIT instruction also copies bit 6 of the location being tested to bit 6 of the PSR. This isn't quite so useful as the N flag as bit 6 doesn't normally signify anything very special. However, if you look at some of the clever machine code programming used in the BASIC interpreter and operating system, you will occasionly find some very neat uses of the BIT instruction operating on the V flag.

Using these binary instructions, a process analogous to binary multiplication can be carried out.

8-Bit Binary Division

This process is analogous to the binary multiplication routine, needing only 8 re-iterations to handle an 8-bit number. It is illustrated in program 4.15A, where the dividend (in this case 31) is stored in location 900 and the divisor (2), in 901. The Y register is used as the loop counter to ensure that 8 passes are made through the algorithm. By means of an ASL and a ROLA instruction, the remainder is built up in the accumulator.

PROGRAM 4.15A

```
LDXIM 31
STX   900
LDXIM 2
STX   901
LDYIM 8
LDAIM 0
ASL   900
ROLA
CMP   901
BCC   6
SBC   901
INC   900
DEY
BNE   238
LDX   900
STX   1024
LDYIM 1
STY   55296
STA   1026
STY   55298
RTS
```

When run, this will display the quotient 15 (as a O) in 1024 and the remainder 1 (as an A) in 1026.

CHAPTER FIVE

Advanced Functions of the Assembler

Labels

The use of labels enables a program to be directed to a named instruction without the necessity of calculating branches or jump addresses. A fancier term for label is SYMBOLIC LABEL as the label itself is symbolic of a location in memory. For instance, the instruction

```
BNE LOOP1
```

instructs the assembler to create the machine code that tells the 6510 to branch to an instruction labelled LOOP1. Thus, LOOP1 STAX 1024 creates a label called LOOP1, whose address is the same as the "STAX" in STAX 1024. In order to tell the assembler that a label is a label and not an instruction, it is preceded by an asterisk (*). This rule is only a convention that has been chosen when writing this particular assembler. Thus, the beginning of LOOP1 would be entered:

```
*LOOP1 STAX 1024
```

Further conventions must be observed when using labels, particularly those concerning spaces. The asterisk, for instance, must be followed immediately by the label (no space between). It may be as long as required but must NOT contain any spaces. There is no particularly technical reason for this, it is simply that the assembler looks for a space in order to work out how long the label is. For this reason the label must be followed by a space and then a normal instruction. These rules may sound a little formidable but don't worry, the assembler will pick up any errors and let you know what is wrong with any particular line. When referring to a label in an instruction it is only necessary to replace the operand with the label itself.

To summarise: LABELS

(i) A label is defined by the asterisk (*) that precedes it.

(ii) The label may be of any length but it is as well to stick to about six characters.

(iii) There must be no gap between the asterisk and the label.

(iv) The label must be followed by a gap prior to the instruction.

It all sounds a little complicated so let's see it illustrated in a program. This uses two loops, called "LOOP1" and "LOOP2" and does some unnecessary branching and jumping by way of illustration.

PROGRAM 5.2 (In Assembly language)

```
         LDXIM 160
         LDYIM 1
         JMP   LOOP2
  *LOOP1 LDAIM 83
         STAX  1183
         TYA
         STAX  55455
         DEX
         BNE   LOOP1
         JMP   END
  *LOOP2 LDAIM 90
         STAX  1023
         TYA
         STAX  55295
         DEX
         BNE   LOOP2
         LDXIM 120
         DEY
         JMP   LOOP1
  *END   RTS
         END
```

Although this program jumps about somewhat, it is still relatively easy to follow. It starts by initialising X and Y then jumps to loop 2 and on completion reinitialises X and Y and then jumps back to loop 1, and from there to the END. Note that a *END will not indicate the end of assembly; the "*END" is a label whereas "END" on its own (without the asterisk) is the pseudo-code that terminates the assembly process. Once this process is complete, the program will reside in memory in exactly the same format as any other program that has been entered. To check this, list the program - from 828 - and the following should appear:

PROGRAM 5.2 (In disassembled assembly language)

```
         In Assembly          In disassembled
         Language             Assembly Language

         LDXIM 160            LDXIM 160
         LDYIM 1              LDYIM 1
         JMP   LOOP2          JMP   850
  *LOOP1 LDAIM 83             LDAIM 83
         STAX  1183           STAX  1183
         TYA                  TYA
         STAX 55455           STAX 55455
         DEX                  DEX
         BNE   LOOP1          BNE   244
         JMP   END            JMP   868
```

```
*LOOP2 LDAIM 90             LDAIM 90
       STAX  1023           STAX  1023
       TYA                  TYA
       STAX 55293           STAX 55295
       DEX                  DEX
       BNE   LOOP2          BNE   244
       LDXIM 120            LDXIM 120
       DEY                  DEY
       JMP   LOOP1          JMP   835
*END   RTS                  RTS
       END
```

Program 5.2 can exist in a variety of forms, three of which are readily available. In its original form it was written in assembly language with labels and this was converted by the assembler into machine code and stored in memory in this form. When the assembler is then asked to list this program it reads the machine code from memory and changes this back into assembly language. Were this process to be carried out immediately after assembly then it would be possible to re-label the label points by editing the assembler but as currently written this is not so, in common with other assemblers. Moreover, once the BASIC program has been re-run, the variables, i.e. the LABELS and LABEL REFERENCES, will have been lost. Re-creating the assembly program from machine code is known as DISASSEMBLY, i.e. the 'list' command could be re-titled as 'disassemble' and this process cannot re-create labels.

When run, this program will print four rows of white diamonds and three of black hearts in the top seven screen rows.

Exercise 5.1

Add a further loop - loop 3 - after loop 2 in program 5.2. Re-write the program to run loop 3 first, followed by loop 1 and then loop 2. Loop 3 should put two rows of red asterisks on the screen below the diamonds.

A possible answer is given on page 9-12.

Memory Labels

In addition to labelling instructions, the assembler also allows memory locations to be given labels. Once again, the assembler needs to be told what to expect and the presence of a memory label is indicated by an "@" at the beginning of line. It is followed immediately by the name assigned to that location and then, after a space, by the location itself in decimal. Thus the instruction

```
@LSB 900
```

informs the assembler that memory location 900 may, in the rest of the program, be referred to as "LSB".

Program 5.3 illustrates the use of memory labels in double precision addition - it adds together two 16 bit numbers:-

$$\text{Number 1} = 2760_{10}$$
$$\text{Number 2} = 948_{10}$$

made up of LSB1 and MSB1, and LSB2 and MSB2; the answers are stored in ANSLSB1 and ANSMSB2.

PROGRAM 5.3

```
@LSB1 900          Define memory location for LSB1.
@MSB1 901          Define memory location for MSB1.
@LSB2 902          Define memory location for LSB2.
@MSB2 903          Define memory location for MSB2.
@ANSLSB 904        Define memory location for ANS LSB.
@ANSMSB 905        Define memory location for ANS MSB.

LDAIM    10     }  Store MSB1 in memory.
STA      MSB1   }
LDAIM    200    }  Store LSB1 in memory.
STA      LSB1   }
LDAIM    3      }  Store MSB2 in memory.
STA      MSB2   }
LDAIM    180    }  Store LSB2 in memory.
STA      LSB2   }

CLC                Clear carry prior to addition.

ADC     LSB1    }  Add LSB's together, store answer
STA     ANSLSB  }  in ANSLSB and print on screen.
STA     1025    }

LDA     MSB1    }  Load MSB1 and add with carry to
ADC     MSB2    }  MSB2, store answer in ANSMSB and
STA     ANSMSB  }  print on screen.
STA     1024    }

LDXIM   2       }  Load 2 into colour RAM to print
STX     55296   }  in red.
STX     55297   }
RTS
END
```

When run, program 5.3 will display a red N (14) in 1024 and a red [(124) in 1025.

As promised (!) in chapter 4, a listing is given below of an eight bit binary multiplication using labels and memory labels. In order to illustrate the carry operation the first few loops through this program are illustrated on fig. 5.1. The numbers 255 were chosen as multiplier and multiplicand as their binary pattern of eight ones is easy to follow. They also give an early carry, although any multiplier over 128 would have given the carry when ASLed.

Abbreviations used in this program are:

```
MPR      =  Multiplier    =  255
MPD      =  Multiplicand  =  255
TEMP     =  Temporary location to store carry bit.
RESLSB   =  Result - Least significant bit.
RESMSB   =  Result - Most significant bit.
ALGO     =  Start of multiplication algorithm.
NOCARRY  =  Jump to point if no carry arises.
```

PROGRAM 5.4

```
@MPR       900               ]  Define memory location for
           LDAIM  255        }  multiplier, load and store
           STA    MPD        ]  multiplier.
@MPD       904               ]  Define memory location for
           LDAIM  255        }  multiplicand, load and store
           STA    MPD        ]  multiplicand.
@TEMP      902               ]  Define memory locations for
@RESLSB    906               }  temporary store, and LSB/MSB
@RESMSB    907               ]  of result.
           LDAIM   0         ]  Initialise (by loading in
           STA     TEMP      }  zero) the temporary and
           STA     RESLSB    }  result store.
           STA     RESMSB    ]
           LDYIM   8            Set loop counter to 8.
*ALGO      LSR     MPR       }  Check if right-most bit of
           BCC     NOCARRY   ]  multiplier=0; branch if so.
           LDA     RESLSB    ]  Calculate current partial
           CLC               }  product and add in to
           ADC     MPD       }  current partial sum.
           STA     RESLSB    ]
           LDA     RESMSB    }  Add current carry into MSB
           ADC     TEMP      ]  sum.
           STA     RESMSB
*NOCARRY   ASL     MPD       ]  Current Partial=0, set up
           ROL     TEMP      }  next loop.
           DEY               }
           BNE     ALGO      ]
           LDA     RESLSB    ]
           STA     1025      }  Display result of LSB and
           LDXIM   1         }  MSB.
           STX     55297     }
           LDA     RESMSB    }
           STA     1024      }
           STX     55296     ]
           RTS
```

When run, program 5.4 will print a ▟ in 1024 and an A in 1025, i.e. answer of 254,1 or $FE01 = 65025.

Fig. 5.1, page 5-10, steps through the first few stages of this program, once all the registers are set up. The contents of each address are shown only when they change.

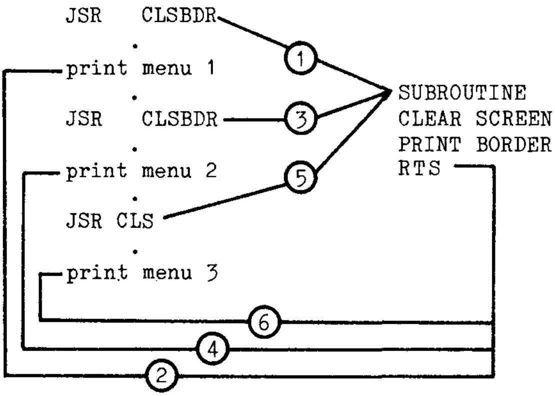
LABEL	INSTRUCTION	Y-REGISTER	ACCUMULATOR	MPR MULTIPLIER	C	TEMP	MPD	RESLSB	RESMSB
		0000 1000	0000 0000	1111 1111		0000 0000	1111 1111	0000 0000	0000 0000
ALGO	LSR MPR			0111 1111	1				
	BCC NOCARRY								
	LDA RESLSB		0000 0000						
	CLC				0				
	ADC MPD		1111 1111						
	STA RESLSB							1111 1111	
	LDA RESMSB		0000 0000						
	ADC TEMP		0000 0000						
	STA RESMSB								0000 0000
NOCARRY	ASL MPD				1		1111 1110		
	ROL TEMP				0	0000 0001			
	DEY	0000 0111							
	BNE ALGO								
ALGO	LSR MPR			0011 1111	1				
	BCC NOCARRY								
	LDA RESLSB		1111 1111						
	CLC								
	ADC MPD		0000 0001		1				
	STA RESLSB							0000 0001	
	LDA RESMSB		0000 0000						
	ADC TEMP		0000 0001						
	STA RESMSB								0000 0001
NOCARRY	ASL MPD						1111 1100		
	ROL TEMP				0	0000 0011			
	DEY	0000 0110							

Fig.5.1

Macro Instructions

One further feature of your assembler is that it enables you to use MACRO instructions. These are blocks of code that you wish to repeat and are thus given a label. When you wish to insert these into your program you simply need to type in the label and this automatically inserts the whole macro. Thus a macro is very similar to a subroutine except that the assembler writes it in every time it is called rather than using JSR instructions. As an example, take a short routine that clears the screen and puts in a border called CLSBDR. Assuming that the program has a hierarchical menu structure and calls this routine 3 times, it could be written in as (i) below of Fig. 5.2 in which the routine is called as a subroutine, or as in (ii) in which the routine is written in 3 times as a macro.

(i)



(ii)

```
MACRO CLSBDR
      .
 MENU 1
      .
MACRO CLSBDR
      .
 MENU 2
      .
MACRO CLSBDR
      .
 MENU 3
```

Fig. 5.2

The above figure is somewhat unfair on subroutines in that they are not so messy as the picture suggests, the 6510 doing much of the organisation. Where macros score is in producing a program that reads more logically and is easier to follow. They take up more space than a subroutine, which is only written once, but as they don't use up processor time jumping about, they run more quickly. Just whether a macro or a subroutine is more preferable in any particular case is left to the individual programmer.

To identify a macro on this particular assembler it is preceded by a "+" sign (as the macro is later 'added' into the program). As with other features, the + sign must immediate precede the macro's name which may be any length and this should be the only entry on that particular line, e.g. "+MACRO1", say. The first occurrence of the macro is then typed in. The end of the macro is signalled by a "+" sign followed by "END". To include a further copy of the macro in the program at a later point, another MACRO1, say, just the line +MACRO1 needs to be entered into the program at the appropriate point - see Fig. 5.3.

```
   .
   .
 +MACRO1

  LDAIM 90    }  First occurrance of MACRO1
  LDXIM 40    }  inserted into program at
  STAX  1024  }  this point. It is defined
  DEX         }  here at the first occurrance.
  BNE   250   }

 +END         }  Signals the end of the macro
   .          }  definition.
   .
 +MACRO1         Assembler inserts a second
                 copy of the macro here.

   .
  RTS
  END
```

Fig 5.3

When using labels, label references and macros, it should be borne in mind that the assembler needs to store the names and locations during the assembly process. The version of assembler supplied on the tape will accept up to a maximum of 21 labels or memory labels, 21 label references and 11 macros. Macros which contain labels or references create new labels and references and these must also be taken into account. Thus, repeating a macro with 1 label and 2 references creates an extra 2 labels and references + one macro.

1. If the numbers of labels, references or macros likely to be used is going to exceed the numbers provided for in the program then it may be necessary to re-dimension the arrays holding the labels, label references or macros:
   a) Labels and memory labels are stored in F$(X), F(X) which

are dimensioned in line 75 of the assembler program i.e. to enable 32 labels to be used, line 75 should be altered to contain:
75 DIM F$(31), F(31).

b) Label references are stored in K$(X), K(X) and K1(X) which are dimensioned in line 80 i.e. to enable 40 label references to be used line 80 should contain:
80 DIM K$(39), K(39), K1(39).

c) Macros are stored in M$(X), M(X) and M1(X) which are dimensioned in line 85 i.e. to enable 16 macros to be used, line 85 should contain:
85 DIM M$(15), M(15), M1(15).

Warning: The Dr Watson Assembler for the CBM 64 includes a section written in machine code which is initially loaded in memory immediately following the BASIC program. This machine code provides the facilities for the Machine Language Monitor (MLM) which are discussed in the next chapter. The first time that the assembler is run, this machine code section is moved into the C block of RAM and the space following the BASIC program is released.

There is a risk that editing the BASIC Assembler program can disturb the machine code section. If you do make any changes to the dimensions in lines 75, 80 or 85 (or any other changes for that matter) be sure to carry out the following procedure:

(i) switch the CBM 64 off and then on, to start with a clean memory,
(ii) load the assembler into memory, but do not run,
(iii) make the required changes, but do not run,
(iv) if you wish to retain a copy of the new (changed) assembler, save it now, but don't overwrite the old assembler, just in case,
(v) do a test run of the new assembler.

If the test fails, check your amendments carefully. If you are confident that that these have been entered correctly, try again, from (i) above.

2. As you are aware, the CBM 64 is blessed with a large memory, and, as a consequence it is extremely unlikely that space problems are likely to occur with the assembler. However, for the record:
   a) Each occurrence of the label suite of variables F$(X) and F(X) requires 8 bytes.
   b) Each occurrence of the label reference suite K$(X), K(X) and K1(X) requires 7 bytes.
   c) Each occurrence of the macro suite M$(X), M(X) and M1(X) requires 7 bytes.

A reasonable rule of thumb is that each new label, reference or macro requires about 10 more bytes.

Further Options of the Assembler

One of the options which is offered on the main MENU is 'Other functions - O'. If this is selected, the screen will display an alternative MENU which provides you with additional functions for inserting or moving code (I), or accessing the Machine Language Monitor (M), or converting a machine code program to BASIC DATA statements (D), or obtaining an assembly listing on the Printer, instead of the screen (P), or, finally, the option to return to the main MENU (X). The 'Monitor - M' option will be dealt with in the next chapter, the other functions are described below.

Inserting or Moving Code

The INSERT function is accessed by selecting an 'O' at the main MENU, an 'I' at the second MENU, and responding 'I' when asked "Insert..I : Move..M".

This facility will be found to be of most use when you need to insert a new section of code into an existing program. The need for this can arise through design or accident (planned expansion, or missed lines). The INSERT facility allows you to open up a gap in the program, into which you can enter new code (using the E option on the main MENU).

In the example shown below, a six byte gap is opened up in an existing program which extends from 828 to 842.

i) Enter start address of code to be entered, i.e. the address of the start of the gap, when asked:

   Start Address for Insertion? 832

ii) Enter the address of the end of the current program when asked:

   End of Current Program? 842

iii) Enter length of the code to be inserted, in bytes, when asked:

   Length of Insertion? 6

iv) When space has been created for the new code, the screen will say:

   OK Space Inserted

v) Press any key to return to the main MENU and then use the normal ENTER procedure to enter the insert by selecting 'E'.

To access the MOVE function, much the same path is followed as for the INSERT function, but selecting 'M' when asked "Insert..I : Move.. M".

In the example below, a machine code program originally written into 828 to 842 is to be moved to the start of the C block of RAM, $C000 onwards.

i) Enter the start address of the program to be moved when asked:

   Old Start Address of Block? 828

ii) Enter the end address of the program when asked:

   Old End Address of Block? 842

iii) Enter the address where the program is to start, after moving, when asked:
   New Start Address of Block
   ....i.e. after move ? $C000

iv) When the block has been moved, the screen will display:

   OK-Block Moved

v) Press any key to return to the main MENU.

Converting a Program to BASIC DATA Statements

A convenient method of attaching a machine code program to a BASIC program is to convert the program to a series of numbers in a DATA statement. The BASIC program also has to include a simple loop which will poke the numbers in the DATA statement back into the appropriate memory locations. This latter technique is described in program 6.3 of the next chapter. The first problem, that of converting the program to DATA statements, is most easily handled by letting DR. Watson's assembler do it for you.

Because of the limitations brought about by the way that the assembler uses the screen in order to carry out this function, a maximum of 128 bytes may be converted to DATA statements at any one time. The DATA statements which are created, contain a maximum of 16 values (representing 16 bytes of the program), hence a maximum of eight DATA statements are created each time. This restriction isn't really very serious, if you wanted to convert a 200 byte program, say, then you simply use the 'D' option twice.

The DATA statements that are created, are added to the assembler program. From here, they can be copied into the BASIC program that will load and make use of the machine code program that they represent.

In the example shown below, a machine code program created in the cassette buffer between 828 and 977 (150 bytes long) is to be converted into BASIC DATA statements.

'O' has been selected at the main MENU and 'D' at the secondary MENU.

i) Enter a suitable line number for the first BASIC DATA statement to be created when asked:

<u>Line No for 1st DATA statement?</u> 20000

The number 20000 has been selected to ensure that the DATA statements will not get mixed up with the assembler. The DATA staements will be numbered 20000 20010 20020 etc.

ii) Enter the start address of the program which is to be converted to DATA staements when asked:

<u>Start Address of Data?</u> 828

iii) Enter the address of the end of the program, unless it is bigger than 128 bytes. In this case, the program is 150 bytes so we will have to have two bites at the bytes (sorry!). So, when asked:

<u>End Address of Data?</u> 955

iv) The assembler will immediately convert the program to DATA statements which will appear briefly on your screen. In this case there will be eight DATA statements which will be added to the assembler program as 20000 - 20070. The screen will then display the message:

<u>DATA Statements now Entered</u>

and you may press any key to return to the main MENU.

v) As there are a further 22 bytes to convert, select the 'O' option on the main MENU, 'D' on the secondary MENU and repeat the process. Obviously, we must specify a line number greater than 20070, when asked, otherwise the program's bytes will be out of order, 20080 will do nicely. The start address and the end address this time will be 956 and 977, which are the bytes remaining to be converted.

Saving the newly acquired DATA statements can be a problem. Simple enough, if you have one of the BASIC Aid/Toolkit programs or cartridges which provide a DELETE command. This will allow you to remove the assembler up to, but excluding the DATA statements e.g. DELETE -19999, say. The DATA statement program can be saved and later MERGED (another BASIC aid command?) with its BASIC program.

However, if you haven't access to such useful tools, then here is one easy method of saving the DATA statements. This technique works well, provided all the DATA statements will fit on to the screen:

a) Exit from the assembler, and list the DATA statements on the screen (e.g. LIST 20000-).

b) Type NEW to remove the assembler from memory. OK, I know that at this moment you have lost, not only the assembler, but also the DATA statements - but worry ye not.

c) Move the cursor to the first of the DATA statements displayed on the screen and press the RETURN key. The first DATA statement has been re-entered into memory. Press the RETURN key again and the next DATA statement has been recovered. Repeat for each of the DATA statements.

d) You can now, either save the program in memory, which consists of the DATA statements only, or you could start to type the BASIC program around them.

Printing the Assembly Listing on a Printer

The 'L' option on the main MENU allows you to display the assembly listing on the screen. If you own a CBM printer however, you will want to print your listings. The 'P' option on the secondary MENU provides this facility. The printed output will be exactly the same as the screen listing and will be printed in double width characters.

At the main MENU select 'O', then select 'P' at the secondary MENU. Make sure that your printer is connected and switched on, of course. Now provide the start address of the program and the end address when asked to do so, and the printer will produce your listings. While the printer is listing, the screen of the CBM 64 will be blanked. This is to avoid the chance of the printer hanging up, which can occasionally happen if you are using a printer with the old VIC 20 ROM. Don't panic, when the listing is finished, the screen will return bearing the message:

OK-Program Listed

and you can press any key to return to the main MENU.

CHAPTER SIX

Without the Assembler!

So far all the machine code programs considered have been entered via the assember. However, as discussed in the last chapter, all the assembler does is to make it easy to POKE data into memory. The assembler is, of course, not the only way of putting a machine code program into memory, one other way is to POKE it in directly. Program 6.1 is expressed in this direct POKE form.

PROGRAM 6.1

```
POKE 828,160
POKE 829,1
POKE 830,162
POKE 831,0
POKE 832,169
POKE 833,90
POKE 834,157
POKE 835,0
POKE 836,4
POKE 837,152
POKE 838,157
POKE 839,0
POKE 840,216
POKE 841,232
POKE 842,208
POKE 843,244
POKE 844,96
```

We cannot enter this into memory, of course, while we are running the assembler. So first of all, select the X option on the main menu to exit. The CBM 64 will report READY and we can enter the direct POKE commands. One easy way to do this to clear the screen first and type in POKE 828,162 from 1024 onwards. After each <return>, a HOME will put the cursor back on the P in POKE. It is then a simple matter to edit the line, changing the necessary digits. When the program is in, it may be run by the SYS 828 command and it will put 256 white diamonds onto the screen. Can you imagine the program that these POKEs have created? Fortunately, it's not necessary to imagine it, as the assembler will disassemble the code from memory for you and display it. First RUN the assembler by typing RUN, then select 'L' at the MENU and type in the address 828.

The assembler will then reveal the source program for 6.1a which was:

PROGRAM 6.1a (In Assembly Language)

```
LDYIM 1
LDXIM 0
LDAIM 90
STAX  1024
TYA
STAX  55296
INX
BNE   244
RTS
```

As well as entering programs without the assembler it is also possible to run them directly or from a BASIC program. For instance, to run the program that currently starts at memory location 828 it is only necessary to put an 828 into the program counter. If program 6.1 is still in memory this can be used for a demonstration of this - if it is not still in memory type it in. Next exit from the assembler and get back to BASIC with the READY cursor.

Now type in SYS 828 and the program will run, putting in the diamonds.

Now run this from BASIC. Put in this following program:

PROGRAM 6.2

```
20000 PRINT"clear"      Clear screen.
20010 SYS 828           Direct PC to 828.
20020 PRINT "ALL OVER"
```

Now type in "RUN 20000" <return> and the BASIC program should run in the following way:-

Line 20000	The BASIC program clears the screen.
Line 20010	Hands control to the machine code program at 828. Control is handed back to BASIC when the final RTS is encountered.
Line 20020	BASIC prints out the message "ALL OVER".

Running a program directly is relatively easy but the direct mode of entry of a program is obviously a tedious way of entering a long program so a further, more easily entered option is offered by the storing of the machine code in DATA statements. Program 6.3 shows a machine code program loaded in via BASIC.

PROGRAM 6.3

```
1 FOR X = 0 TO 16
2 READ A
3 POKE (828+X),A
4 NEXT X:RESTORE
5 DATA 160,1,162,0,169,90,157,0,4
6 DATA 152,157,0,216,232,208,244,96
7 END
```

This is run as for a normal BASIC program with a RUN command. Once run, the DATA will have been loaded into memory and lines 1 to 7 must be looped out and the statements REMed out.

```
1.  GO TO 10:FOR X= 0 TO 16
5.  REM DATA 160,1,162....
6.  REM DATA 152,157,0....
```

Once this is done, the main assembler can be run and the program RUN from the MENU - it's located at 828. Of course, once the data is loaded into memory, the program can also be run by a SYS 828 command.

The Dr. Watson Assembler for the CBM 64 offers a further method of entering machine code via:

THE MONITOR : Option M (of 'other functions')

This feature offers a ready facility for examining and modifying memory. To enter Machine Language Monitor (MLM) type "M" at the MENU. The computer will then display:

```
C*
   PC    SR AC XR YR SP
.;E416   B1 FF 28 06 EA
.
```

and the cursor will be flashing merrily away, in the position immediately following the last dot ("."). The MLM, on the Dr. Watson Assembler is a full implementation of the Commodore Machine Language Monitor as used on their PET range of microcomputers. If you have used the Machine Language Monitor on a PET computer, then you should be immediately at home with the Dr. Watson Monitor. However, if you are not familiar with MLM, then you are probably wondering what on earth the display means. Have patience, all will be revealed below.

The cursor following the dot indicates that Monitor is waiting for a single letter command from you to indicate which facility is required. To investigate this type in the command:

```
.M 033C 034C <return>
```

Take care when doing this to put in all the spaces, otherwise Monitor will be confused. This command should, assuming program 6.1 is still loaded, result in the display below:

```
.M 033C 034C
.:033C A0 01 A2 00 A9 5A 90 00
.:0344 04 98 9D 00 D8 E8 D0 F4
.:034C 60 00 00 00 00 00 00 00
.
```

Fig. 6.1

The M command requests the monitor to list, in hexadecimal, the block of memory defined, i.e. in this case from 033C to 034C (addresses in hex). The block which is listed above is, in fact, the program loaded by program 6.1 and the listing tells that 033C contains A0, 033D contains 01, and so on to 034C which is loaded with 60. The values that you will see displayed in 034D to 0353 may not be 00 as shown in figure 6.1 since these may contain value which were not changed when you loaded program 6.1. You can list any part of memory using the M command, of course.

Look at the block listed by monitor again. Notice that each line starts with the dot (indicating that monitor is awaiting a command) followed by a colon (":"). This colon is, in fact, one of monitor's commands: the command to change memory. One of the most useful facilities offered by the monitor is the ability to edit machine code programs as if they were BASIC programs. For instance, while program 6.1 is loaded, go into monitor and list 033C to 034C.

Type: .M 033C 034C<return>

At this stage the cursor should be flashing immediately below the colon of the 034C line of the display. Now move the cursor over the first '1' of 01 at 033D and replace this with a '6' using the normal editing that you use in BASIC. After changing this, enter it into memory by pressing RETURN. You will be able to see the change that you made quite easily since monitor originally displayed the line in orange and your change will be entered in light blue. Now move the cursor down again and type:

```
.M 033C 034C<return>
```

Monitor will show you the block again and you will be able to confirm that the contents of 033D have indeed been changed to 06. The : command to monitor has been obeyed. Now repeat the process to replace the 5A in 0341 with a 58 followed by a RETURN to action the : command which enters the new value. This time however, to display the change, move the cursor up to the .M 033C 034C line, and press RETURN.

The display should now appear as in fig. 6.2:

```
.M 033C 034C
.:033C A0 02 A2 00 A9 58 9D 00
.:0344 04 98 9D 00 D8 E8 D0 F4
.:034C 60 00 00 00 00 00 00 00
.
```

Fig. 6.2

You can of course make as many changes as you like in one line. When you press the RETURN key, all the changes will be made. You could if you wished to build the change line yourself, type in the colon followed by the address followed by up to eight hex values. However, most users find it much easier to use the M command to display the line and then use the : command which monitor has set up for them in the display.

In order to exit from the monitor the X command is used followed by return. Try this and the machine will return to the MENU.

The Assembler program may then be RUN and on LISTing the LDYIM 1 and LDAIM 90 instructions will be seen to have been changed to LDAIM 6 and LDAIM 88. On running, the white diamonds will be replaced by blue clubs.

Not only can programs be edited in monitor, they can also be entered. Try this with the following exercise, using the monitor to enter the following code:

i) Enter monitor.
ii) List memory contents
Type M 033C 034E return.
iii) Replace contents with:

PROGRAM 6.5

```
.M 033C 0358
.:033C A0 50 A9 4D 99 FF 03 99
.:0344 9f 04 A9 4E 99 4F 04 88
.:034C D0 F0 A9 09 A0 F0 99 FF
.:0354 D7 88 D0 FA 60 88 88 88
```

Note that the final "88"'s in this program are really irrelevant as they occur after the 60 or RTS. Thus any values could be put into these locations.

The machine code program is now entered into memory and may be RUN either from the Assembler or by SYS 828. It can also be dis-assembled using the assembler to reveal the Assembly language version.

A very useful built-in feature of the monitor is its ability to move whole blocks of code from one location to another. This is very simply achieved by changing the address displayed on the monitor listing and pressing return. For instance, the program in 033C onwards can be copied to start at 900_{10} (0384_{16}) or by the following procedure:

i) Enter MONITOR
ii) List 033C to 0358 by
iii) .M 033C 0358 <return>
iv) Re-position cursor to the 033C display line
v) Replace the 033C by 0384 and press <return>
vi) Repeat for the 0344, 034C and 0354 lines, as below:

```
0344 goes to 038C
034C goes to 0394
0354 goes to 039C
```

The procedure has made a copy of the 033C program in 0384 onwards. It can be run from the assembler or by a SYS 900.

The program can also be run by a monitor command "G", which is short for "GO!" It runs a machine-code program directly from memory. Naturally the CBM 64 has to be told where the program starts and so the total command reads:

```
.G 0384<return>
```

Protecting Machine – Code in Memory

Two monitor commands enable machine-code programs to be SAVEd onto tape and LOADed back from this. However, one slight problem arises in doing this as all machine-code programs so far have been stored in the cassette buffer and this is used when LOADing and SAVEing programs.

In order to protect programs from being over-written, they must be stored elsewhere in memory.

On the CBM 64, BASIC programs are normally stored between 2048 (0800_{16}) and 40959 ($9FFF_{16}$). However, strings generated by a program are written from the top of memory downwards. The CBM 64 knows where to start storing this information, as when the machine is turned on it checks what the highest available memory locations are and stores this information in the zero-page locations 55 and 56. If a machine-code program is stored at the top of memory, then it will be written over when any BASIC program is run - such as the assembler - when this generates strings. However, before it stores the strings, the BASIC program checks in memory locations 55 and 56 to find where to start. This allows one to protect the top part of memory by loading an address into 55 and 56 that leaves some clear space.

Immediately after switching on the CBM 64, the address 40960_{10} ($A000_{16}$) is to be found in 55 - LSB, and 56 - MSB. In order to protect 100 bytes, this address would be lowered to 40860_{10} ($9F9C_{16}$). There are a number of ways of achieving this. For instance, by means of direct programming:

```
POKE 55,156(9C16)
POKE 56,159(9F16)
```

followed by,

```
CLR
```

The CLR command is required to persuade the CBM 64 to adjust all the other pointers that it maintains. They point to the end of the space in memory which is used to store the variables, the arrays and bottom of the string space. These need to be cleared following an adjustment of the top of memory pointer, otherwise the 64 could get it's line crossed somewhat, consider what would happen if an attempt were made to access the strings which are now in the protected part of memory.

Lowering the top of memory could also be achieved by including a line in the BASIC program which is associated with the machine code program. The line should read:

```
10 POKE 55,156 : POKE 56,159 : CLR
```

Another method is to make use of the Monitor. The locations associated with the top of memory are 37_{16} and 38_{16}. In the same way that we altered the program i.e. using the M command to display the locations, editing the : line and typing return, similarly, we can alter the high memory values in 37_{16} and 38_{16}.

However, there may be a problem associated with lowering the top of memory. Memory may have been lowered already by another program, perhaps one of yours. Although A000 will be found in 37_{16} and 38_{16} at switch on time, the address which will be found later on may well have been lowered. In general, the safest way to lower memory is to calculate the values from the current values stored in 55_{10} (37_{16}) and 56_{10} (38_{16}). The following BASIC programming lowers the top of memory by 200 bytes.

```
100 MT = 256*PEEK(56)+PEEK(55) : REM Get OLD address
110 MN = MT - 200 : REM Calculate NEW address
120 MH = INT(MN/256) : REM Calculate NEW hi-byte
130 ML = MN - 256*MH : REM Calculate NEW lo-byte
140 POKE 56,MH : POKE 55,ML : CLR : REM Lower memory
```

However, having said all that, I must now reveal that there is a portion of memory on the CBM 64 which is never used by BASIC. This region extends from 49152_{10} ($C000_{16}$) to 53247_{10} ($CFFF_{16}$). Machine code which is placed in this region will not be overwritten by BASIC. However, the previous explanation of how to lower memory is not really a waste of time, since C000 to CFFF might be used as a working space by cartridges which are plugged into the CBM 64. Although this is not very likely, it is as well to know how to *protect your program if you have to place it in BASIC memory.*

The Dr. Watson Assembler for the CBM 64 does use a few hundred of the 4096 bytes available in the C000 to CFFF area at the top of the area, but there is a great deal of space which you may safely use. The address of the first byte in this area which is used by the Assembler may be found in 53246 (CFFE$_{16}$) and 53247 (CFFF$_{16}$) if you need to know this. As ever, the format is low byte followed by high byte

The 033C program can be copied to C000 onwards, using Monitor's : command, and listed by means of the M command. The lines displayed will start with the addresses C000, C008, C010 and C018. Once stored at height the LOAD and SAVE routines of the Monitor can be used.

Save

For a program to be SAVEd, the monitor needs to know where it starts and where it ends and, as with other programs, it needs a name. Thus the full command becomes

```
.S "program-name",01,start-address,end-address+1<return>
```

if saving to tape, and

```
.S "program-name",08,start-address,end-address+1<return>
```

if saving to disk.

Notice the peculiarity that the end address is one larger than the highest address of the program being saved.

Translating this to save the check program to tape, it becomes:

```
.S "CHECK",01,C000,C01D <return>
```

After the RETURN, the CBM 64 will respond with a "PRESS RECORD & PLAY ON TAPE" (if you are saving to tape) and when this has been done, an "OK". As you will expect, the screen will go blank during the saving process. Finally, when the recording is complete, the CBM 64 will respond with a "." and the flashing cursor of the MONITOR as it awaits further instructions. Should you have made a mistake in your typing, monitor will display a red question mark immediately following the portion of the line that it did not understand - just re-type the line (or edit the bad line).

Load

Provision is made on the monitor for re-loading a machine code program from tape and storing this back into the location from which it was originally saved. It does this by saving the start address on to the tape or disk so that it knows where to load the program back. As this address is known, it doesn't need to be specified (and should not be specified) in the LOAD command i.e.

```
    .L "program-name",01<return>
```

to load from tape, and

```
    .L "program-name",08<return>
```

from disk.

Thus

```
    .L "CHECK",01<return>
```

will recover the CHECK program from tape.

In fact, the L command can be greatly simplified if using cassette tape. The 01 can be ommitted, and Monitor will assume you mean tape, and if the name is ommitted also, then Monitor will load the first program from tape that it finds. Thus, the simplest form of the load command is:

```
                .L<return>
```

Following the return, the CBM 64 will respond with a "PRESS PLAY ON TAPE" and an "OK" when pressed. As is usual with the CBM 64, the screen will go blank during the loading process, clearing when a program is found etc. When the program is loaded the CBM 64 will respond with a "." and the flashing cursor of the Monitor as its awaits further instructions.

Register Display & Debugging

You might remember that I promised to explain the display presented by Monitor when it is first entered. I shall now keep my promise. The display looked rather like the following:

```
*C
   PC   SR AC XR YR SP
.;E146  31 10 00 06 F8
.
```

The *C tells us that monitor was entered by a CALL from a program. The other way that it could have been entered is by a BREAK, in which case it would have displayed *B - more of BREAK entries later. The next line is simply a heading for the third line, namely: Program Counter (PC), Status Register (SR), Accumulator

(AC), X-Register (XR), Y-Register (YR) and finally Stack Pointer (SP). In practice, these values are of academic interest only for a CALL entry, they really come into their own if the entry is made as a consequence of a BREAK.

One of the 6510 instructions that has not yet been dealt with is BRK which has a hex value of 00_{16}. If the 6510 encounters a BRK instruction then the normal flow of the program is 'interrupted' (a full description of the BRK command and of machine 'interrupts is given in chapter 8 but a brief treatment is given here). If a Machine Language Monitor has not been implemented, then the result will simply be that the screen will be cleared and you will return to the BASIC READY state. If a MLM has been implemented, however, then all sorts of nice things happen. The values stored in the Program Counter and the Registers have been saved during the BRK interrupt and one of the functions of the Monitor is to recover these and present them to the programmer. Hence the display.

Sometimes, the reason for the BREAK is simply that the program has gone beserk and is executing data as if they were instructions, there are always a large number of zeros hanging around in a computer's memory and these will be interpreted as BRK instructions. This is good, since you will (a) have regained control of your runaway program and (b) know what part of memory it was running away in.

However, a program behaving like a rogue elephant is not the only reason for a BREAK entry to Monitor. A much better reason is that you yourself organised the BRK. Suppose that you have a machine code program which is not doing what you expect, and you are unable to determine what is going wrong in spite of bringing all your intellectual power to bear on the problem. Don't give up, there is a way forward.

You can insert a BRK instruction into your program and run it. When (if) the BRK instruction is executed you will arrive in Monitor with the values of all the registers shown to you. You can now use the M command to look at various values stored in those parts of memory which you have been messing about, and this, hopefully, may give you the clue that you need. If you don't arrive in Monitor, by the way, that in itself is vital information, since presumably, you placed the BRK instruction in a part of the program that you expected to be executed so either the program can't get that far or it is taking an unplanned route.

You can seed your program with as many BRK instructions as you wish. There is no problem of identification as Monitor tells you the value of the Program Counter when it is entered, so you will be able to identify which of your many BRK's caused the entry.

When you try your first BRK program, you will discover a curious thing. The Program Counter which is shown is not the address of the BRK instruction but is one byte bigger. In other words, it is the address of the instruction which follows the BRK. This is useful, since a G command to Monitor without an address (G followed by RETURN) will make Monitor automatically pick up the value of the PC and registers as shown and recommence execution.

Let us have a look at a display, this time from a BREAK entry.

```
*B
   PC   SR AC XR YR SP
.;035C  30 00 02 03 F4
.
```

Notice the semicolon (";") at the start of the third line. It is, in fact, another command to Monitor. This one says 'change the registers' to these values. You can edit the semicolon line, (in the same way as you edited the colon line resulting from a M command) and typing a return will cause any changes which you have made to be remembered by Monitor. If you re-enter your program using the G command, then the program will restart with the new values stored in its registers. Naturally, if you change the value of the PC, then you will re-enter the program at a different point from the BRK exit.

There is one other command to Monitor which we have not yet dealt with. The R command (R followed by RETURN) will cause Monitor to display the registers, just like the display you see when you enter Monitor, but without the *C or *B line. The main use of this command is when you are debugging and wish to re-enter the program, but you want to change a register, say. The chances are that the original display of the registers has long since scrolled off your screen. The R command will put them back there, and you can change the register before hitting the G.

You may feel that putting extra BRK commands into your program is a bit of a nuisance, especially if the program is long. In this case, you could use the M and : commands to change the value of an existing instruction to 00_{16}, thereby changing it to a BRK command. If you do this, you must remember before re-entering the program to:

(a) decrement the Program Counter by 1 (it is probably pointing to the first byte of the address part of the instruction that you BRKed, and
(b) restore the value of the instruction that was replaced (the trick here is to make a note of the hex value BEFORE you change it to zero).

I am sure that you can see what a powerful tool this gives you for debugging your machine code program. You will be surprised how useful this can be.

Finally, there is one more benefit that is yours. Because the Monitor loaded by Dr. Watson's assembler is loaded into the C block of memory, out of the reach of BASIC, it remains in memory after you have stopped using the assembler. It will only be destroyed by switching the CBM 64 off, or if you overwrite it with a machine code program, of course. Therefore, it can be used whenever required.

To enter Monitor, you simply type:

```
SYS 10
```

and you will enter Monitor via a BRK command. The reasoning behind this is that location 10 on the CBM 64 always contains a zero, and as we know this is the opcode for the BRK. So, a SYS 10 tells the 64 to execute the BRK command stored at 10, bingo, we enter Monitor. Why, I hear you ask, should I want to enter Monitor, if I am not working with machine code programs? You will be surprised how often you will find this convenient. For example, to adjust the pointers to bottom and top of memory, to change colours of the screen (it is easier to remember the hex addresses of the registers D020 and D021 than their decimal equivalents) and so on and so on.

One word of warning however. If you need to press STOP/RESTORE at any time, then the BRK link to Monitor will be broken (no pun intended), and SYS 10 will simply take you back to the BASIC READY state. The trick is to PEEK at the two locations 790 and 791, immediately after leaving assembler, and make a note of the values that you find there. These locations contain the link address to Monitor for the BRK command. You will find that they form the LO/HI address of somewhere in the C block. If you need to hit STOP/RESTORE at any time, poking these values back into 790 and 791 will restore access to Monitor.

Summary of Monitor Commands

M <lo-address> <hi-address>	Display Memory
: <address> <byte1> <byte2>...	Change Memory
R	Display Registers
; <prog-counter> <status-reg> etc	Change Registers
S <"filename">,<device>,<lo-add>,<hi-add + 1>	Save block of memory
L <"filename">,<device>	Load from tape/disk
G <address>	Go run
X	Exit from Monitor

Colour Displays on the CBM 64

One of the major features of your CBM 64 computer is its ability to produce colour displays. This facility is just as readily used from machine-code as from BASIC.

Controlling the Colour

Program 6.6 shows how the screen/border combination can be demonstrated using a short machine code program.

PROGRAM 6.6

```
         LDYIM 15
         LDXIM 15
 *SCREEN STY $D021
 *BORDER STX $D020
         DEX
         BPL BORDER
         DEY
         BPL SCREEN
         RTS
```

Unfortunately, when this is run as it stands there is little to see as the whole program runs through in about 2600 cycles, or 1300 microseconds! In order to enable the changes to be visible, a delay must be inserted to hold each change on the screen long enough for the eye and brain to perceive it.

In previous chapters we used counting loops to produce the delay, sometimes nested loops. The next program does things the easy way by making use of the three byte jiffy clock at 160-162 (A0-$A2_{16}$). This is a binary counter which counts jiffies (1/60^{ths} of a second), 162 ($A2_{16}$) is incremented by 1 every jiffy, and rolls over into 161 ($A1_{16}$) every 256 jiffies, 161($A1_{16}$) rolls over into 160($A2_{16}$) every 65536 jiffies. This gives ample scope for delays. Program 6.2 loads 246 into 162 and waits for it to become positive which will happen in approximately 1/6 second.

PROGRAM 6.7

```
         LDYIM 15
 *BORDER STY   $D020
         LDXIM 15
 *SCREEN STX   $D021
         LDAIM 246
         STA   162
 *LOOP   LDA   162
         BMI   LOOP
         DEX
         BNE   SCREEN
         DEY
         BNE   BORDER
         RTS
```

Individual characters on the screen can also be controlled by use of the screen colour codes (in fact we have been using this from our earliest programs). If memory location 55296 ($D800_{16}$) contains a "2", then the top left memory location, i.e. 1024 (0400_{16}) will print out on the screen in red. The complete colour set is given below in fig. 6.3.

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

Fig. 6.3

Program 6.8 shows how the colour of blocks of screen can be defined by means of the screen colour codes.

PROGRAM 6.8

```
        LDXIM 200
 *LOOP  LDAIM 8
        STAX  $D7FF
        LDAIM 160
        STAX  1023
        DEX
        BNE   LOOP
        RTS
```

This process is taken a stage further in program 6.9 where the sixteen different colours are cycled through, with a delay between each.

PROGRAM 6.9

```
         LDAIM 0     } Set up first colour and save
         STA   1019  } at top of tape buffer.

*LOOP1   LDXIM 200     Set counter for 10 rows.
*LOOP2   LDAIM 160     Load reversed space for screen.

         STAX  1023  ]
         LDA   1019  |
         STAX  $D7FF } Print block onto screen.
         DEX         |
         BNE   LOOP2 ]

         LDAIM 186   ]
         STA   162   } Delay of 60
*DELAY   LDA   162   | jiffies (1 sec).
         BMI   DELAY ]

         LDA   1019  ]
         ADCIM 1     |
         CMPIM 16    } Move to next colour
         STA   1019  | and check if all done.
         BNE  LOOP1  |
         RTS         ]
```

CHAPTER SEVEN

Using the CBM 64's Built-in Subroutines

As the CBM 64 itself uses the 6510 chip, it has stored inside it, on ROM (Read Only Memory), machine code routines that control the 6510. These enable the CBM 64 to deal with the BASIC commands that are put into programs, all input and output and all the standard routines which are needed to keep the CBM 64 alive and well. The ROM which handles all the BASIC statements is located in memory between $A000 and $BFFF. The ROM which looks after all the non-BASIC routines is called the Kernel by Commodore, and lies between $E000 and $FFFF.

In addition to this use of memory, both BASIC and the Kernel make use of the RAM memory in the bottom four pages between $0000 and $03FF, the most frequently used locations being in the zero page $00 to $FF. Some of this usage is for the storage of transient data such as, for instance, the jiffy clock ($A0 to $A2) that ticks away all the time the computer is turned on. Some of the RAM is used for more permanent data, such as the pointers stored in 43 to 56 ($2B to $38) which indicate the area of memory used by the BASIC program, and its data areas. Some of these uses take up only one, two or three bytes, others use much more, such as the cassette tape buffer which uses 192 bytes when transferring data between memory and the cassette tape.

The most difficult aspect in using the built-in subroutines is to know where they obtain their data from and where they deposit the data that they have generated. This is especially true of the routines which make up the BASIC ROM. Fortunately, Commodore have been very helpful in the design of the Kernel routines. The starting points and the sources of the data for the major Kernel routines are very well defined. Another helpful feature is that the starting points of the Kernel routines will be kept the same when new/better versions of the ROM are developed and this will enable machine code programs which use only these routines to retain their usefulness for the forseeable future.

First let's have a look at the contents of the accumulator using a CBM 64 Kernel subroutine. We have already displayed the accumulator by using a STA command to move a copy of the accumulator to a screen location, e.g. STA 1024. A better/easier method, however is to use the Kernel subroutine which is called CHROUT and is located at 65490 ($FFD2). This will output the accumulator to the screen, starting from the current cursor position. This is illustrated in:

PROGRAM 7.1

```
LDAIM 42       Load accumulator with '42'
               (an asterisk).
STA   1524     Store accumulator in the middle of
               the screen (1524).
LDXIM 1        Load X with 1 for colour white.
STX   55796    Store it in colour RAM.
JSR   $FFD2    Jump to CBM 64 CHROUT subroutine.
RTS
END
```

When run, this displays two asterisks. In the middle of the screen at 1524 a white asterisk which we placed directly. However, we also have another asterisk, probably in the top left hand corner of the screen and probably coloured light blue. This extra asterisk was placed there by the ROM subroutine. Notice that the subroutine didn't have to be told where the asterisk was to be placed, nor did we specify which colour. The subroutine placed the asterisk in the current cursor position with the current colour.

Unfortunately, though, outputting is not quite as strightforward as this program suggests! To demonstrate this, try replacing the LDAIM 42 with an LDAIM 1. You can do this with a POKE 829,1. Now, when you run the program, although the expected white 'A' appears in the middle of the screen, nothing appears in the top top left hand corner of the screen. The problem is that the CBM 64 uses the ASCII character set when using subroutine 65490, rather than the screen set. The ASCII code for asterisk happens to be the same as the screen code, so the program worked properly in the first run. However, the ASCII code for 'A' is 65, not 1 as it is in screen code. The ASCII code set is listed in Table 4 (Appendix 2), page A2-25.

The two major advantages of this subroutine are that, firstly, it locates the cursor automatically and will increment this automatically, each time the subroutine is called; secondly it will automatically store the current colour in the appropriate position in the colour RAM. If you ran the program from the assembler then the current cursor position would have been 1024, because the assembler does a 'clear screen' and sets the current colour to light blue before running your program.

If we wish to, we can set the current cursor position and the current colour from our machine code program. Setting the current colour is quite easy. The current colour is stored in 646 ($0286) 0 black, 1 white, 2 red as usual. So, loading the appropriate value (0-15) in this location is all that is needed.

Positioning the cursor could be quite complicated, but for the presence of another built- in subroutine called appropriately PLOT which is located at 65520 ($FFF0). This does all the hard work for us. PLOT will either read or set the current position of the cursor using the X and Y registers. If entered with the carry flag set, then PLOT will return the current position of the cursor in the X and Y registers, X containing the number of the row (0 to 24) and Y containg the column (0 to 79). If entered with carry clear, then the values that we have stored in the X and Y registers will be used to position the cursor.

Let us use the PLOT subroutine to place a yellow asterisk at the beginning of the tenth line of the screen.

PROGRAM 7.2.

```
        CLC               Set up for move cursor (not read it).
        LDXIM 9           Load 9 (gives tenth line).
        LDYIM 0           Load 0 (gives first column).
        JSR   $FFFO       Call PLOT to position cursor.
        LDAIM 7           Load 7 for yellow.
        STA   646         Make this the current colour.
        LDAIM 42          Load asterisk.
        JSR   $FFD2       Output it.
        RTS
```

If we had wished to put the asterisk at the eighteenth position of the tenth line then we would have loaded Y with 17 in the line beginning at 831. Thus program 7.2 can be modified by program 7.2a.

PROGRAM 7.2a

```
        831 LDYIM 17
            END
```

..to yield program 7.3:

PROGRAM 7.3

```
        CLC
        LDXIM 9
        LDYIM 17
        JSR   $FFFO
        LDAIM 7
        STA   646
        LDAIM 42
        JSR   $FFD2
        RTS
        END
```

When run, this program moves the cursor down 10 and across 18, and prints the yellow asterisk at this location.

To illustrate how the CHROUT routine updates the cursor so that it will move to the next cursor position following each call, modify program 7.3 by:

PROGRAM 7.3a

```
0843       LDYIM 4
*LOOP      JSR   $FFD2
           DEY
           BNE   LOOP
           RTS
           END
```

to yield:

PROGRAM 7.3b

```
           CLC
           LDXIM 9
           LDYIM 17
           JSR   $FFFO
           LDAIM 7
           STA   646
           LDAIM 42
           LDYIM 4
*LOOP      JSR   $FFD2
           DEY
           BNE   LOOP
           RTS
           END
```

When run, program 7.3b will print four yellow asterisks in line 10 in columns 18, 19, 20 and 21. Notice that we didn't have to reload the accumulator with 42 each time round the loop, so the CHROUT subroutine did not alter the value of the accumulator. It is also clear that the Y register is not altered, otherwise the counting of the four asterisks would not have worked. In fact, CHROUT doesn't alter either A, X or Y. This is one of the good features of this particular routine. Not all the built-in subroutines are so kind, so it is important to bear in mind the possibility of one or more of the registers being altered by any subroutine that we choose to use from the kernel or BASIC ROMs.

Many BASIC programs use the GET command, which accepts a single byte input into the keyboard buffer. GET uses one of the Kernel subroutines to carry out this operation called GETIN which is located at 65508 ($FFE4). When called, GETIN retrieves a character from the keyboard queue and returns it as an ASCII value in the accumulator. If the queue is empty, GETIN does not wait, but returns a value of zero. Program 7.4 shows GETIN in operation.

PROGRAM 7.4

```
           LDAIM O         Set accumulator clear.
*LOOP      JSR   $FFE4     Jump to GETIN subroutine.
           BEQ   LOOP      Branch back if no input.
           JSR   $FFD2     Display accumulator on screen.
           RTS
```

When run, this program sets up the loop

```
*LOOP      JSR   $FFE4
           BEQ   LOOP
```

which waits for an input. Once the input occurs the program runs through the BEQ command and executes the remainder of the program, i.e. displaying the character obtained by GETIN. Notice that we choose to use the CHROUT subroutine to place the result on the screen rather than place it there directly. Both GETIN and CHROUT work with ASCII code, rather than the CBM 64 screen code. Direct placement on the screen would cause an input 'Z'to appear as diamond on the screen. In the case of numbers, the CBM 64 screen and ASCII codes coincide, so a numeric input will result in the same character being displayed on the screen either way.

CHRIN at 65487 ($FFCF) is an alternative input Kernel subroutine to GETIN. When inputting from the keyboard, its action is similar to the BASIC INPUT statement i.e. the first time that GETIN is called, the cusor will be switched on and will remain on until a RETURN (CHR$(13)) is typed. The characters which are typed in are stored in the BASIC buffer which starts at 512 ($0200), any editing which is done during the typing such as deletes and inserts will be applied. However, we do not need to organise the retrieval of this characters from this buffer, as the characters will be returned in sequence for each call on CHRIN. There is no need to organise the display of the characters as this also is organised by CHRIN. This produces the shortest program yet in 7.5!

PROGRAM 7.5

```
           JSR $FFCF
           RTS
```

The GETIN subroutine can be used to design your own INPUT routine. For instance, you could use GETIN to enter one character at a time, checking, say, for a certain number of characters or for the inputting of a certain terminating character which need not necessarily be a RETURN. You could also set the routine to check each character in the actual INPUT itself and give a warning if it is an invalid character.

Program 7.6 shows an arrangement with a check built in to look for a comma (ASCII 44) to be input.

PROGRAM 7.6

```
        LDXIM 44      }  Store terminator in
        STX   900     }  900.

*LOOP   JSR   $FFE4   }  Use GETIN subroutine.
        BEQ   LOOP    }  Wait for input.

        JSR   $FFD2      Output A onto screen.

        CMP   900     }  Look for comma, if not present
        BNE   LOOP    }  branch back.

        LDAIM 13      }  Output a <return> to
        JSR   $FFD2   }  screen to be tidy.

        RTS
```

This program simulates an INPUT routine that is terminated by a comma instead of a RETURN. To use a terminator other than comma, change the operand of the first instruction so that 900 will be loaded with the correct value. Try this for:

Exercise 7.1

Modify program 7.6 to accept an input that is terminated by a space, use a POKE command to make the change.

Answer on page 9.10.

As was mentioned earlier, one of the major problems encountered when using built-in subroutines is that they too use the 6510. That means that they put things into A, X and Y and they also modify the SR - the Z, N and other flags. Hence when returning from any JSR it's not reasonable to assume that everything is just as it was left before the JSR, unless, of course we know (as with the CHROUT routine) what is left undisturbed.

As an example of this, look at the following program that is designed to input a four-character string from the keyboard. Firstly it sets a loop counter in X at 4, then it uses the GETIN and CHROUT subroutines. On return from these it decrements X and checks for Z flag set - all very straightforward!

PROGRAM 7.8

```
        LDXIM 4                 Set up loop counter.
*LOOP   JSR   $FFE4   }         GETIN subroutine.
        BEQ   LOOP    }
        JSR   $FFD2             CHROUT subroutine.
        DEX
        BNE   LOOP              Check for end of loop.
        RTS
```

However, when run, this program RTS's after only one character has been input! This suggests that one of the subroutines is using the X register. To verify this the program can be made to print out the X register at various stages just to check. This is done in program 7.9, where the contents of the X register are examined immediately after returning from the subroutine.

PROGRAM 7.9

```
         LDXIM 4
*LOOP    JSR   $FFE4
         BEQ   LOOP
         STX   1024
         LDYIM 1
         STY   55296
         JSR   $FFD2
         STX   1026
         LDYIM 1
         STY   55298
         DEX
         BNE   LOOP
         RTS
```

When run, this program prints the one character input as before but also displays two A's at 1024 and 1026. Since the first A is output immediately after the exit from the GETIN routine (JSR $FFE4) then it is clear that GETIN does modify the X register. As we have already discovered, CHROUT is innocent in this case. Since the X register was changed to 1 by GETIN then the DEX/BNE at the end of the program caused the program to leave the loop on the first pass.

To overcome this problem, the X register value must be stored somewhere prior to entering the subroutine and then retrieved prior to being decremented. Program 7.10 shows this process, where X is temporarily stored in 900 during the subroutine.

PROGRAM 7.10

```
                                     Assembled Version
@LOOPCONST 900
           LDXIM 4                    LDXIM 4
*STORE     STX   LOOPCONST            STX 900
*GET       JSR   $FFE4                JSR 65508
           BEQ   GET                  BEQ 251
           JSR   $FFD2                JSR 65490
           LDX   LOOPCONST            LDX 900
           DEX                        DEX
           BNE   STORE                BNE 239
           RTS                        RTS
```

When run, this program allows four entries from the keyboard, displays these and then passes out of the loop.

As a technique, the use of memory storage in this way does work, but it does call for some care in storing the data safely and retrieving it when needed. Fortunately the 6510 has a device for doing this operation automatically. It is

THE STACK

The stack (S) is a block of memory, on the CBM 64 machines located from 511 ($01FF) down to 256 ($0100) capable of holding 255 bytes. It is used for the rapid transfer of data and is filled downwards from 511, its next vacant location being recorded by a STACK POINTER (or SP as it is known; this was shown as the last value in the line which was displayed by Monitor in response to the R command). When anything is to be retrieved from the STACK only the last item put in is accessible. The usual analogy is with a stack of plates, only the top one being accessible as this was the last one put there. However, the 6510 stack is filled DOWNWARDS, i.e. from 511 towards 256, so plates are put at the bottom and retrieved from there, antipodean fashion! This mode of filling and emptying the stack is known as last-in, first-out, i.e. the stack is a last-in, first-out (LIFO) store.

One function of the stack is to record addresses during subroutine jumps, which it does automatically. When the 6510 sees an instruction such as JSR 50000 it must first of all record where the next instruction is so that it can find its new location after the subroutine has been executed and then place the "50000" into the program counter (PC). The process is examined below with the program segment 7.11 (from 7.10).

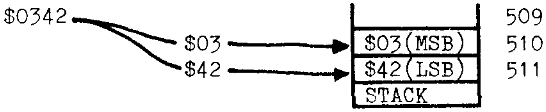
PROGRAM 7.11

```
STEP 1.  828 ($033C)  STX 900    ($0384)
STEP 2.  831 ($033F)  JSR $FFE4  (65508)
STEP 3.  834 ($0342)  BEQ 251
```

INSTRUCTION   STX 900

STEP 1.
- i) Calculate address of next instruction, i.e. 831 or $033F.
- ii) Put next address into program counter.
- iii) Execute instruction STX 900.
- iv) Retrieve address for next instruction from PC, i.e. $033F.

STEP 2.
- v) Fetch next instruction, i.e. at $033F. This is JSR $FFE4.
- vi) Retrieve address for next instruction in program, i.e. $0342, and place this on the stack.



vii) Record the next vacant location in the stack pointer, i.e. 509 (SP=509).
viii) Load $FFE4 into program counter.
ix) Jump to subroutine at $FFE4.
x) Execute subroutine until RTS is encountered.
xi) Look at stack pointer to find last data. SP=509, therefore Data stored at 510 and 511.
xii) Extract data from 510 and 511 (i.e. $0342) and load this into PC, reset SP to 511.
xiii) Jump to $0342.

STEP 3. xiv) Fetch next instruction and carry on with program.

In this example, the first subroutine could have met nested subroutines, and each time a JSR was executed the return address would have been piled on top. Then, as the program returned successively through these subroutines, the return addresses would have been stripped off to steer the 6510 back to the original point of departure. Fig. 7.2 illustrates this for subroutines nested 3 deep starting from a program with a JSR $0384 instructi.on in $033D.

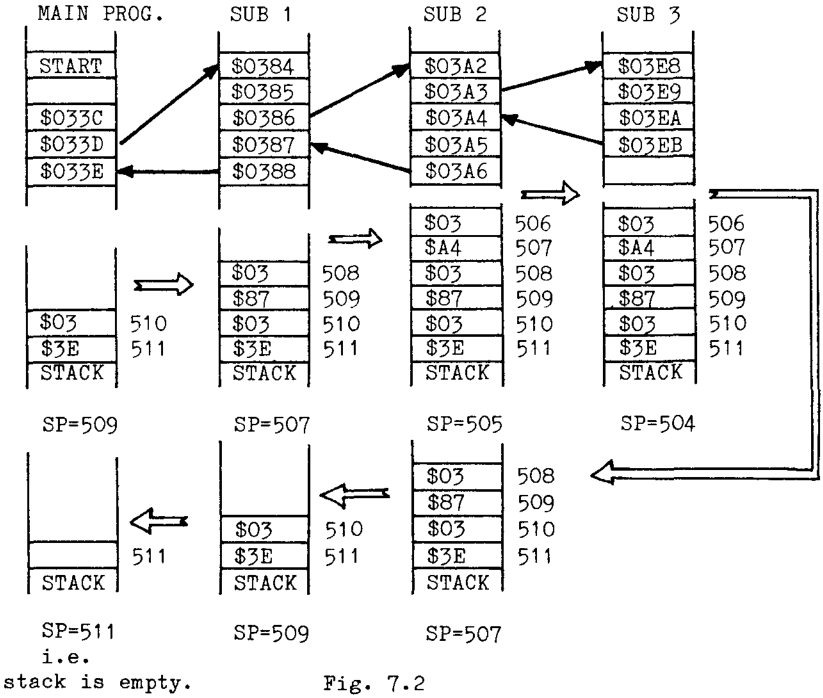


Fig. 7.2

In the above program, the instructions at $03EB, $03A6 and $0388 would all be RTS. Thus when the 6510 finds an RTS at $03EB, it takes the top address off the stack, which is $03A4. Reading downwards at SUB 3, the stack gives

i) address for return to subroutine 2.
ii) address for return to subroutine 1.
iii) address for return to the main program.

Fortunately, the operation of the stack in recording addresses when executing subroutines is automatic and so the programmer can allow the 6510 to do the job. However, as was seen when using built-in subroutines, the stack does not automatically store register contents but must be programmed to do so. Only two instructions exist for storing registers, neither of these operating on the X and Y registers. These must be handled via the accumulator which is stored using:-

PHA	PusH contents of Accumulator onto stack.

The contents may then be retrieved by means of:-

PLA	PuLl top of stack into Accumulator.

In both cases, the stack pointer is updated appropriately so that it continues to point to the next empty location in the stack.

Using these instructions, program 7.8 can be rewritten to transfer the X register into the stack and retrieve this when needed. The stack pointer will take care of the order of the data, providing that the LIFO structure of the stack is borne in mind and data is entered and retrieved in the right order.

Writing these in yields:

PROGRAM 7.12

```
LDXIM 4          Set up loop constant.
TXA            } Transfer X into stack.
PHA            }
JSR   $FFE4
BEQ   251
JSR   $FFD2
PLA            }
TAX            } Recover X from stack,
DEX            } decrement and check for end of
BNE   241      } loop.
RTS
```

When run, the program accepts a four character input and prints this on the screen.

As well as affecting the accumulator, a subroutine is most likely to reset one or more flags in the SR. On returning to the main program, the reset flags are then certain to upset the course of this. To overcome this problem, the 6510 has built-in provision for saving the SR (i.e. the condition of all the flags on the stack). This is brought about by the instruction:-

PHP	PusH Processor status register on stack.

Retrieval of the data is achieved by:-

PLP	PuLl stack to Processor status register.

In program 7.12 the SR was not saved on the stack as, prior to testing the Z flag with a BNE, the DEX instruction reset this. However, under other circumstances it may have been necessary to preserve the state of the SR so this should be written into 7.12 as an exercise.

Exercise 7.2

Rewrite program 7.12 so as to save the condition of SR in the stack prior to the subroutines and retrieve this after these.

A possible answer on page 9.11

When using the stack, the main precaution to take is to check the order of entry anad retrieval several times - always LIFO. For instance, a possible routine for saving the accumulator, X and Y registers and the SR is given in Fig. 7.3.

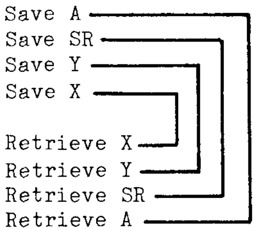


Fig. 7.3

CHAPTER EIGHT

Interrupting the 6510, Signed & Floating-Point Variables

When doing a job that has to be done, no-one likes interruptions until the job's finished. The 6510 is just like that too! During a piece of program it has its Accumulator and X and Y registers under control and all the flags set appropriately. Thus, when an interrupt, comes all these registers have to be stored, usually in the stack, and after the interrupt they must all be retrieved. An interrupt is, in fact, only a subroutine that comes when IT is ready and not the 6510! This means that the interrupt is generated from outside the 6510's cosy system, either from an external device or the keyboard.

The handling of an interrupt has to be prepared for in the program and at certain times the program may well not be able to allow an interruption. If, for instance, another device is sending a stream of data into memory, then a HAND-SHAKING procedure is carried out between the two machines. Quite simply, this is an exchange of messages along the line: "I am ready to send data, are you ready to receive it?" "Yes." "Here's the data ..... end of data." "Thanks!". If such an exchange is interrupted, then the data is likely to become garbled, and hence worthless. During such periods when no interrupt is allowable, the program can block most interrupts - not all - to allow a particular process to be completed. The instruction that allows this blocking is:-

SEI	SEt Interrupt Disable Flag (prevent interrupts).

Ironically, the first action that usually needs to be taken in an interrupt is to set the I (interrupt disable) flag by use of SEI. We need to prevent any more interruptions, at least for a while, so that we can check up to see if the interrupt which has occurred is our interrupt (there may be other potential interrupts lurking around). When we have made sure that the interrupt which has taken place is the one we are interested in, then the interrupt disable flag (or simply I flag) may be reset to '0' or cleared by the instruction:-

CLI	CLear Interrupt Disable Flag (allow interrupts).

If you think about the above, you will realise that we may allow interrupts to be interrupted. That is perfectly true. It is rather like someone starting to peel the potatoes when the kettle interrupts by coming to the boil, so the potato cleaning task is suspended for a while and a start is made on making a pot of tea. At this point the milkman, who wants his milk bill to be settled, interrupts by ringing the doorbell. So the tea making task is suspended and the milkman is attended to. In the middle of this, the telephone rings ... After dealing with the telephone call, then the milkman paying task may be completed, then we must finish making the pot of tea, at which time we can go back to peeling the potatoes. The trick is to remember how far you have got with each task when it is restarted. Unlike human beings, the 6510 is very good at remembering where it has got to, and has little difficulty.

Not all interrupts can be blocked by setting the I (interrupt disable) flag, as some are crucial and must get through at all costs. Many such circumstances may arise during the control of plant or when a power failure calls for immediate action. To enable the 6510 to distinguish between the two conditions, it has two different input pins, one for each type of interrupt. One of these is the NMI or Non Maskable Input pin which cannot be blocked, and the other the IRQ or Interrupt ReQuest pin which is masked off by the I flag.

When the 6510 receives an interrupt signal it always completes its current instruction before doing anything else. In the case of an IRQ interrupt it would then check the I flag and if not clear, continue until the program cleared this. Next, before going into the interrupt procedure, the contents of PC are saved on the stack (telling the 6510 where to return to when it has finished processing the interrupt), and then the SR (status word) is saved. This saving of data is really only a half-measure as, almost certainly, A, X and Y would be changed during the interrupt procedure.

Next the I flag is set to '1' to prevent further interrupts and *then the appropriate address for the appropriate interrupt routine* is loaded into the PC. These addresses are found at 65530 and 65531 ($FFFA and $FFFB) for an NMI interrupt and 65534 and 65535 ($FFFE and $FFFF) for an IRQ interrupt. These interrupt routines are terminated with an instruction:-

RTI ReTurn from Interrupt.

On meeting this, the 6510 does three things:

(a) restores the SR so that the status flags are the same as when the interrut occurred,
(b) resets the I flag - an automatic CLI, and
(c) looks in the stack for the return address and stores it in the SR - like an automatic RTS.

Unfortunately, the 6510 only does half the job at an interrupt and, as discussed in chapter 7 (page 7-10), A, X and Y should be saved on the stack. In the case of both X and Y this must be transferred into A before being pushed onto the stack (by a PHA), and when pulled off the stack (by a PLA) it will need to be restored to the appropriate register by a TAX or TAY.

The 6510 possesses one other interrupt instruction:-

BRK	BReaK

When the BRK instruction is encountered, the 6510 first resets the PC by indexing this by one place (so that the PC points to the byte following the BRK instruction), and stores this address on the stack then it sets the Break Flag (B flag) which is bit 4 of the SR and stores it on the stack also. Following this, the 6510 then does a normal IRQ interrupt using the IRQ VECTOR at $FFFE (LSB) and $FFFF (MSB). The IRQ routine will check whether this interrupt was caused by a true IRQ or a BRK instruction. Normally, on the CBM 64, if the IRQ routine discovers that it was a BRK entry, then the routine will jump into BASIC, the screen is cleared and READY will be displayed. As you will have discovered in chapter 6, however, nothing so mundane and useless will occur when you are the owner of a Dr. Watson assembler. A BRK instruction on the Watsonised CBM 64 will cause an entry to Monitor, great stuff for debugging!

To test this, run the following:-

PROGRAM 8.1

```
LDAIM 90
STA   1024
LDAIM 1
STA 55296
BRK
```

When run, this will print a white diamond in 1024 and then go into the interrupt routine, which on the Watsonised CBM 64 enters the Monitor.

By using the assembler to disassemble the code at $FFFE and $FFFF, you will be able to find this address and from there to follow through the rest of this routine. By following this code through you will discover how the assembler traps the BRK command and enters Monitor instead of the normal boring old reset, clear screen and READY routine. The vital element is the JMPIA 790 instruction which you will find in 65368 ($FF58). Before Watsonisation, 790 and 791 normally contain the address of the routines which take us back to the BASIC READY state. Now they contain the address of the Monitor Break entry point, this address having been planted there by the assembler.

When the 6510 chip is used as an element in a system, it is then that BRK comes into its own as the interrupt vectors, as they are called (addresses pointing to routines) at $FFFA and $FFFE can point to whatever routine the designers wish. On a computer like the CBM 64, the Commodore designers have made the decision and wisely pointed the 6510 along a path that will do least harm. However, by routing this routine through a vector in RAM which is accessible and changeable by the user, they give us the option of doing something different, like running Monitor. This use of RAM vectors is a feature of the CBM 64 which enables us to trap into many of the routines used by BASIC or the Kernel operating system.

Signed Numbers

In all the mathematical exercises done so far, the numbers used have been treated as simple positive numbers. Thus, any arithmetical processes have dealt with these numbers as strings of eight bits. However, if negative integers are to be used in arithmetic, one of the bits must be used to indicate that a given byte represents a negative number. Bit 7, the left-most bit, is used to do this, being set to a zero if the number is positive and a one if it is negative. By using one bit in this way, the magnitude of the number stored in the remaining seven bits is restricted to +127 to -128. Conventionally a zero is used on bit 7 to indicate the presence of a positive number and a 1 for a minus number. One of the problems that arises from this usage is that, in theory, two forms can exist for the number zero, i.e. +zero and -zero:

$$+0 = 00000000$$

$$-0 = 10000000$$

In order to overcome the problem, the negative number is normally represented in what appears to be a wierd form - TWO'S COMPLEMENT! Wierd it may be, but it works!! In order to work out a two's complement representation, take the number 38_{10} which in binary is 00100110. To convert it to its two's complement negative form, first of all switch each bit of the positive number from a 0 to 1 or vice versa, i.e. to its COMPLEMENT:

$$00100110 \longrightarrow 11011001$$

Next 1 is added to this switched form or ONE'S COMPLEMENT, i.e.

$$\begin{array}{rr} & 11011001 \\ + & 1 \\ \hline & 11011010 \\ \hline \end{array}$$

This yields the negative two's complement representation, i.e.

$$-38_{10} = 11011010_2$$

To understand the significance of this representation, three sums using it are illustrated below.

i) 38 - 38, which should of course yield zero.

$$\begin{array}{rcl} -38 & = & 11011010 \\ +38 & = & 00100110 \\ \hline 00 & = & 00000000 \end{array}$$

ii) $43_{10} - 38_{10}$

$$43_{10} = 00101011_2$$

$$-38_{10} = 11011010_2$$

$$\text{i.e.}\quad \begin{array}{rcll} -38 & = & 11011010 & \\ \underline{+43} & = & \underline{00101011} & \\ & = & \underline{00000101} & = \underline{5}_{10} \end{array}$$

iii) $24_{10} - 38_{10}$

$$24_{10} = 00011000_2$$

$$\text{i.e.}\quad \begin{array}{rcr} 24 & = & 00011000 \\ -38 & = & \underline{+11011010} \\ & & \underline{11110010} \end{array}$$

As this answer has a 1 in bit 7, it is a negative two's complement representation. To convert this, first find the one's complement:

11110010   00001101

Next add on 1

$$\text{i.e.}\quad \begin{array}{rcl} 00001101 & & \\ +\underline{\qquad\quad 1} & & \\ \underline{00001110} & = & \underline{-14}_{10} \end{array}$$

Overflows

In signed number arithmetic the seven "magnitude" bits (i.e. 0 to 6) can only store a number up to +127, so any attempt to store a number greater than this will result in a 'carry' into bit 7, or as it is known in this case, an OVERFLOW. Consider the sum 100_{10}+30.

$$\text{i.e.}\quad \begin{array}{rcr} 100_{10} & = & 01100100_2 \\ +\ \underline{30}_{10} & & +\ \underline{00011110}_2 \\ \underline{130_{10}} & & \underline{10000010_2} \end{array}$$

Thus 10000010 is a negative number, as signified by the '1' in bit 7, and is therefore in two's complement form. To convert this, first find the one's complement

$$01100100_2 \longrightarrow 10011011_2$$

and then add 1

$$10011011_2 \longrightarrow 10011100_2 = \underline{156}_{10}$$

The 6510 handles this situation by monitoring the accumulator and, when an overflow occurs, setting the overflow or V-flag. This flag can be tested by the instructions:-

BVC	Branch on oVerflow Clear.
BVS	Branch on oVerflow Set.

BVC tests the overflow flag and if it is clear, or not set (V=0), then a branch is executed.

BVS tests the overflow flag and if it is set (V=1), then a branch is executed.

When carrying out multiple precision arithmetic processes with signed integers, bit 7 must be treated as an internal carry and when an overflow occurs this must be transferred to the most significant byte.

Program 8.2 illustrates the use of BVC to test for an overflow as the accumulator is indexed with ones.

PROGRAM 8.2

```
CLC
LDAIM 0
ADCIM 1
BVC   252
STA   1024
LDXIM 1
STA   55296
RTS
END
```

When run, the content of the accumulator is increased progressively until the right-most seven bits are filled with ones. At the next increment, the seven ones reset to zeroes and a carry is generated that pops into bit seven. This sets the carry bit and stops the branching, allowing the program to run through to the RTS. The STA 7900 then prints the value at overflow as an white reverse @ (i.e. 128_{10}).

As with other flags, provision exists for the control of the overflow flag and it can be cleared by the instruction:-

CLV	CLear the oVerflow flag.

Unlike the other flags, however, the overflow flag cannot be set (as on the carry flag). It isn't really something that a programmer wants to do, in the normal way of things anyway, so the 6510 designers left it out of the instruction repertoire.

Numerical Screen Output

In all the numerical examples to date, screen output has been in CBM 64 display code. While it is possible to interpret this using a table, it is obviously necessary in a program to display numbers as numbers to the base 10. The major complication in this procedure lies in the fact that CBM 64 display code is effectively a base 256 representation and can thus display 0_{10} to 255_{10} using only one character, where base 10 displays would require up to three characters to represent the same value.

Program 8.3 tackles this conversion task and firstly sets out to find if the number is greater than 200 - i.e. first digit is '2' - or if it is less than 200 and greater than 100 - i.e. first digit '1'. It then checks in a similar way for the tens and units and uses the stack to store the remainder (i.e. so far unprocessed portion) of the number while adding the conversion constant (48) to the accumulator to change the binary value to the display value equivalent. In the example given, the number to be displayed - 152 - is loaded into the accumulator at the start of the program.

PROGRAM 8.3

```
          LDXIM 1        Load colour white into X.
          LDAIM 152      Put in number to be O/P.
          CMPIM 200      Compare A with 200.
          BCC   ONEHUND  Branch if number less than 200.
          SBCIM 200      Remove left-most digit
                         (carry already set).
          PHA            Store remainder on stack.
          LDAIM 50       Load A with '2' for 2 x 100.
          STA   1024     Print A on screen.
          STX   55296    Store white in colour RAM
          PLA            Retrieve A from stack.
          JMP   TENS     Jump to tens routine.
 *ONEHUND CLC            Clear carry.
          CMPIM 100      Compare A with 100.
          BCC   TENS     Branch if less than 100.
          SBCIM 100      Remove left-most digit
                         (carry already set).
          PHA            Store remainder on stack.
          LDAIM 49       Load A with '1' for 1 x 100.
          STA   1024     Print A on screen.
          STX   55296    Store white in colour RAM.
          PLA            Retrieve A from stack.
    *TENS CLC            Clear carry.
          LDYIM 0        Set Y to Zero.
          CMPIM 9        Compare A with 9.
          BCC ZEROTENS   Branch if A less than 9.
    *LOOP INY            Increment Y.
          SBCIM 10       Subtract 10 from A
                         (carry already set).
          CMPIM 9        Compare A with 9.
          BCS   LOOP     Branch if A greater than 9.
*ZEROTENS PHA            Store A on stack.
          TYA            Transfer Y to A.
          ADCIM 48       Add conversion constant to A
                         (carry already clear).
          STA   1025     Print A on screen.
          STX   55297    Store white in colour RAM.
          PLA            Retrieve A from stack.
          ADCIM 48       Add conversion constant to A
                         (carry still clear).
          STA   1026     Print A on screen.
          STX   55298    Store white in colour RAM.
          RTS
```

When assembled the program looks as in program 8.3a.

PROGRAM 8.3a

```
LDXIM 1          Load white  into X.

LDAIM 152        Check if number less than 200,
CMPIM 200        if so branch.
BCC   15

SBCIM 200
PHA              Subtract 200 from number and
LDAIM 50         print out digit '2' for 200.
STA   1024
STX   55296
PLA
JMP   868

CLC              Check if number less than 100,
CMPIM 100        if so branch.
BCC   12

SBCIM 100
PHA              Subtract 100 from number and
LDAIM 49         print out digit '1' for 100.
STA   1024
STX   55296
PLA

CLC
LDYIM 0          Check if remainder of number less
CMPIM 9          than or equal to 9; if so, branch.
BCC   7

INY
SBCIM 10
CMPIM  9
BCS    249       Count number of 10's left in
PHA              remainder and print out 10's
TYA              digit.
ADCIM 48
STA   1025
STX   55297
PLA
ADCIM 48         Print out unit's digit.
STA   1026
STX   55298
RTS
```

When run, this program will print '152' in white at the top left-hand corner of the screen. In general terms it can be used as a subroutine which prints out the contents of the accumulator.

Floating-Point Numbers

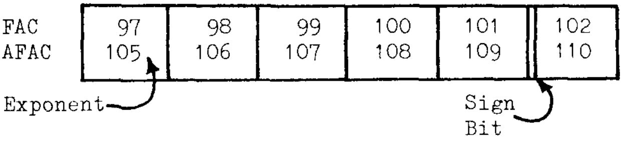
With the integers used so far, the scope of the arithmetical processes carried out has been somewhat restricted. When working in BASIC, however, the binary floating-point constants have 10 digit precision and are displayed with 9 digits. Their exponents have a range of -38 to +37. Each value is stored in six consecutive bytes of memory and, to ease the manipulation of these, two 'accumulators' are provided in memory locations 61_{16} to 66_{16} (97_{10} to 102_{10}) and 69_{16} to $6E_{16}$ (105_{10} to 110_{10}). These are known as the Floating-Point Accumulator (FAC) and Alternative Floating--Point Accumulator (AFAC).

To store a number which could contain up to 10 digits when displayed in base 10 form, the floating point accumulators use only six bytes. How is this trick managed?

When running BASIC programs, you will have seen that very large or very small numbers are expressed in exponential form or scientific notation. Thus, 4079.013 can be expressed as 0.4079013E+4 or 0.4079013 x 10^4, while 0.0000417 can be expressed as 0.417E-4 or 0.417 x 10^{-4}. This representation contains two parts: the first part (the 0.4079013 in the first example) is called the MANTISSA, and the second part (the +4 of the 10^4) is called the exponent. These two parts are stored in binary in the floating point accumulators in the following manner:

(a) BINARY MANTISSA is stored in the middle four bytes of FAC and AFAC. The sign of the mantissa is stored in the sixth byte where a '1' on bit 1 signifies a negative mantissa and a '0' signifies a positive mantissa.

(b) BINARY EXPONENT, is stored in the first byte of FAC and AFAC. As this has to store both positive and negative exponents, the exponent needs to be converted to a positive form and this is done by adding 128. Thus, an exponent of 10 would be stored as 128+20, i.e. 48, while an exponent of -20 would yield 128-20, i.e. 108.



FAC	97	98	99	100	101	102
AFAC	105	106	107	108	109	110

When loading a floating-point number, BASIC normalises its binary representation and thus its left-most (most significant) digit is always 1. Take, for instance, the number $+1400_{10}$ (0578_{16}). Expressed in binary this is $0000\ 0101\ 0111\ 1000_2$. In this form, the binary number has an exponent of 2 (i.e. 1) and has an implied binary point (a binary number has a binary point instead of a decimal point, also known as the RADIX) to the right of the least-significant digit, i.e.

$0000\ 0101\ 0111\ 1000_2.$

Binary point or
RADIX

Putting the exponent in gives a representation:

$0000\ 0101\ 0111\ 1000_2.\ *2^0$

and when NORMALISED, i.e. the binary point moved just past the left-most significant digit:

$.1010\ 1111\ 0000\ 0000 \quad *2^{11}$

i.e. the binary point has been moved 11 places to the left and so the number needs to be multiplied by 2^{11} to return to its original value.

Finally this is padded up with 0's in its least-significant bytes to form a four byte mantissa.

i.e.	Byte 2	Byte 3	Byte 4	Byte 5
	1010 1111	0000 0000	0000 0000	0000 0000

The exponent of 11_{10} is now converted to the correct format by adding 128_{10} to its raw form, i.e. it becomes 139_{10} or $1000\ 1011_2$ and this byte is added to the first location in the accumulator, i.e. the representation becomes:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
1000 1011	1010 1111	0000 0000	0000 0000	0000 0000

The sign of the mantissa is expressed in byte 6 by the condition of its most-significant bit, this being set to a '1' for a negative number and a '0' for a positive one. Thus, to represent $+1400_{10}$, bit 7 of byte 6 of FAC or AFAC is set at '0'. The other bits of this byte may be set at some other number. Thus the final floating-point accumulator representation of $+1400_{10}$ is:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
1000 1011	1010 1111	0000 0000	0000 0000	0000 0000	0??? ????

To use this accumulator directly from machine-code is clearly none to easy, as the data has to be loaded into the memory locations in the correct format. However, some built-in subroutines are accessible to enable this to be done. For instance, a two byte integer can be converted into floating-point and loaded into the floating-point accumulator (FAC) by means of the subroutine at $B391_{16}$ (45969).

To accomplish this, the least-significant byte is loaded into the Y register and the MSB into the accumulator. Then a JSR $B391 will convert this integer into floating-point and load it into FAC. This subroutine is demonstrated below in program 8.4, where the LSB and MSB of an integer are converted into a floating-point number.

PROGRAM 8.4

```
LDY LSB
LDA MSB
JSR $B391
RTS
```

These two floating-point accumulators - FAC and AFAC - are used for the manipulation of floating-point data and a means is provided for transferring floating-point data into them.

Taking AFAC, for instance, the subroutine for loading floating--point data from memory resides at $BA8C (47756) and it moves the contents pointed to by the Accumulator (LSB) and the Y register (MSB) into AFAC. Thus, if the data is stored at $0350 onwards, program 8.5 carries out this transfer:

PROGRAM 8.5

```
LDAIM 80
LDYIM 3
JSR   $BA8C
RTS
```

A similar subroutine re-locates data into FAC, this being located at $BBA2 (48034).

Unfortunately, as with the other routines using the floating-point accumulators, a measure of faith has been called for as in no case has the content of either FAC or AFAC been visible. To rectify this a hybrid BASIC/machine-code command is utilised:

The USR Command

This command allows the transfer of data between FAC and a machine-code program. For instance, the line B=USR(Q) in a BASIC program causes the system to place the value of Q into the floating-point accumulator. It then jumps to a machine code routine whose address it finds at $0311 (785) LSB and $0312 (786) MSB. The presumption is that you have placed a machine code routine into memory starting at that address and set the values of $0311 and $0312 to point to the routine. When your routine makes use of FAC it will, of course be using the value that was in Q when the USR function was called. When the particular arithmetic process has been carried out, your routine should leave the answer in FAC and, because of the assignment, this will then be placed into B by BASIC. As you would expect, we can use the USR function in the same way as any other function, for instance PRINT USR(P+2) will print out the result of the machine code routine which will have started with FAC containing the value stored in P incremented by 2.

To test this, a number can be placed into the floating-point accumulator by means of the USR function and then printed out. This is done by means of the two programs, the first written in BASIC and the second in Assembler - program 8.6a and program 8.6b.

PROGRAM 8.6a

```
20 000 PRINT"<CLR>"        i.e. the key marked CLR
                           which is displayed as a
                           reversed heart.
20 010 POKE 785,60         60=$3C }Address=$033C
20 020 POKE 786,3           3=$03 }
20 030 B=1400
20 040 Q=USR(B)
20 050 PRINT"Q=";Q
```

PROGRAM 8.6b

```
828 RTS
```

When this program is run, an address - $033C (828) - is loaded into locations 785 and 786 by lines 20 010 and 20 020.

When line 20 040 is run, the argument B (1400) is loaded into the floating-point accumulator. Control is then handed over to the machine code program at $033C (828). At this point, the machine code routine doesn't actually modify the value of FAC, it simply obeys the RTS which will return control to BASIC. BASIC will place the contents of FAC into Q. Line 20 050 prints out the value stored in Q, which has not been modified by the machine code routine.

This routine offers a way of loading any number, which is valid in BASIC, into the floating-point accumulator. It also offers a way - albeit rather round-about - of examining the contents of FAC. This is not as straight-forward as might be imagined, as most BASIC commands use the FAC when operating; thus even a PEEK command changes its contents. However, if the contents are examined in machine-code, immediately after being set they can be seen before BASIC gets its hands on them again. To do this, program 8.6b should be amended to examine memory locations $61 to $66 (97 to 102) and then to print these on the screen. This is done in program 8.7.

PROGRAM 8.7

```
LDXIM 6       Set up loop counter.
LDYIM 1       Load Y with 1 for white.
LDAX  96      Load one byte of FAC.
STAX  1424    Print byte on screen.
TYA           Transfer 1 to A.
STAX  55696   Load white in colour RAM.
DEX           Decrement loop counter.
BNE   243     Branch if more to do.
RTS
```

PRINT CONTENTS OF FAC ON SCREEN

As this program prints the contents in CBM 64 display code, program 8.6a should also be modified to decode this - as below.

PROGRAM 8.8

```
20 000 PRINT "<home>"
20 010 POKE785,60
20 020 POKE786,3
20 030 B=1400
20 040 A=USR(B)
20 050 PRINT"A=";A
20 060 FORX=0TO5
20 070 PRINT PEEK(1425+X);"  ";
20 080 NEXT X
```

This program simply PEEKs the locations that display the contents and prints the answer. One thing to note when you are typing this program in is that the '<home>' in line 20 000 indicates that you type the unshifted CLR/HOME key (easier for you to interpret than a reversed heart in the listing).

When run, this will display:

```
A = 1400
```

and the contents:

```
139  175   0   0   0   47
```

When compared with the calculated contents on page 8-11 it will be seen that bytes 1-5 are identical, while byte 6 contains something different. In fact it is only the most-significant bit that is of consequence and as this is set to a '0', a positive mantissa is indicated. Just to verify this, change 20 030 in program 8.8, to read

```
20 030  B = -1400
```

On re-running, byte 6 of the FAC will now read 128+47, i.e. its MSB has been reset to a '1'.

Those earlier subroutines that required faith can now be tested using programs 8.8 and 8.9. Take program 8.4. This, it was said, converted an integer into floating-point and loaded it into FAC. Program 8.9 below tests this by printing out the contents of FAC; it uses 1400 ($0578) as the integer, i.e. LSB=$78=120 and MSB=$05=5.

They are loaded into Y (LDYIM 120) and A (LDAIM 5).

PROGRAM 8.9

```
LDYIM 120
LDAIM 5
JSR   $B391
LDXIM 6
LDYIM 1
LDAX  96
STAX  1424
TYA
STAX  55696
DEX
BNE   243
RTS
```

CONVERT INTEGER TO FLOATING-POINT

Program 8.8 should then be modified so as not to put 1400 into FAC, i.e. line 20 030 should read

```
20 030 B=1         (or any number other than 1400)
```

The suite of programs can then be executed using a RUN 20000 and will display the value of A as 1400.

Doubters will also, most probably, wish to see program 8.5 run, i.e. to load floating-point data from memory. This is done in programs 8.10 and 8.11 where the number +2000 is used. However, a slight complication arises here, as floating-point numbers are stored in a slightly different format in memory from those in FAC and AFAC. Taking +2000, for instance, this is stored in FAC and AFAC as 139 250 0 0 0 0, or:

FAC	97	98	99	100
AFAC location	105	106	107	108
	139 1000 1011	250 1111 1010	0 0000 0000	0 0000 0000

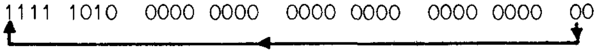
FAC	101	102
AFAC location	109	110
	0 0000 0000	0 0000 0000

However, when storing large quantities of data, a more economical format is used which requires five bytes only. This is known as MFLPT (Memory FloaTing PoinT) format; the six byte format used in FAC and AFAC is known as FLPT. Byte 6 of FLPT is very wasteful as only bit 7 of byte 6 (memory location 102 or 110) is really needed. Obviously, to be able to dispense with byte 6 we have to find a way of storing bit 7 as it gives the sign of the mantissa, so one spare bit must be found elsewhere. Such a spare location exists in byte 2 on its left-most bit, as during normalisation of the mantissa the decimal point is moved until this bit is filled by a '1'. As it should always be set to '1' then the BASIC interpreter doesn't really need to read it to discover that it is '1' and therefore its location can be used for another purpose. The BASIC interpreter can be set up to assume that this is a '1' and automatically make an allowance made for this by adding a 1 to the left of the mantissa when using this. This redundant bit, therefore, can be and is used to store the sign bit, i.e. a '0' for positive and a '1' for negative, and byte 6 can be dispensed with, giving us the MFLPT format. The representation of +2000 in the alternative floating-point formats then becomes:

a) in FLPT format (as used in FAC and AFAC):

```
    1          2          3          4          5          6

1000 1011  1111 1010  0000 0000  0000 0000  0000 0000  0000 0000
```



b) in MFLPT format (as used for storing variables):

```
    1          2          3          4          5

1000 1011  0111 1010  0000 0000  0000 0000  0000 0000
   139        122         0          0          0
```

To use the "load FAC from memory" subroutine the variable data has to be loaded into memory, and this is most easily done using a direct POKE program - program 8.10 below:

PROGRAM 8.10

```
POKE 853,139
POKE 854,122
POKE 855,0
POKE 856,0
POKE 857,0
```

The machine code program to use this subroutine sets a pointer to the data in 853_{10} (0385_{16}) onwards and then transfers it with a JSR $BBA2.

Program 8.11 should be executed by means of the BASIC program 8.8.

PROGRAM 8.11

```
LDAIM 85
LDYIM 3
JSR   $BBA2
LDXIM 6
LDYIM 1
LDAX  96
STAX  1424
TYA
STAX  55696
DEX
BNE   243
RTS
```

TRANSFER DATA FROM MEMORY TO FAC

When run, program 8.11 will load the floating-point representation of +2000 into FAC and print this onto the screen. BASIC program 8.8 will then decode the graphic data and print this out in numbers.

Naturally AFAC has a similar facility for transferring data, this being located at $BA8C (47756).

Once the data is loaded into FAC or AFAC a facility exists for copying from one to the other, that for copying AFAC into FAC residing at $BBF6 (48118).

A similar subroutine copies the other way, i.e. from AFAC to FAC, this being located at $BCOC (48140).

EXERCISE 8.1

Write a suite of programs to load 2000 into AFAC from memory and then print out the contents of this from 1424 onwards. Then transfer AFAC into FAC and again print out the contents, this time from 1624 onwards.

One possible answer give on page 9-11.

Floating-Point Subroutines

Having got numbers into FAC and AFAC, use can be made of the built-in COMMODORE subroutines that offer the opportunity to handle these six-byte monsters with reasonable ease. A word of warning here, though. When using these subroutines one must first know where they are - obvious, eh? Obvious perhaps! However, to date, many of these have already had four addresses - old ROM, 2.0 BASIC and 4.0 BASIC on the PET, BASIC V2 on the VIC 20 and they are now relocated again for BASIC V2 on the CBM 64. The addresses quoted in the book are for the CBM 64 and are correct for the models of CBM 64 used in writing this book. However, it could happen that COMMODORE might decide to issue a new version of BASIC for the CBM 64 some time in the future, and these addresses might therefore change again.

A comprehensive list of these addresses is provided in Appendix III. In addition to the addresses for the CBM 64, the equivalent addresses are provided for the VIC 20, PET BASIC 2.0, and BASIC 4.0. The equivalent addresses will be found most useful when setting out to convert for the CBM 64 a machine code program which was written for one of the other machines.

A further caveat must be given along with the advice to try these. One must know where these subroutines pick up their data from and where they deposit the result, if they are to be used safely. Many of them start with a short initialisation section that prepares the data and deposits it in the right place for action. The subroutine that moves data from memory into AFAC, for instance, starts by transferring its data address into $1F (31) and $23 (35) from the accumulator and Y register, hence when started at $BA8C it expects to find the data address in A and Y. However, at $BA90 (47760) the subroutine proper begins and then retrieves its data address from $22 (34) and $23 (35). Thus it can be entered early with the data address in A and Y, or a few bytes later with its address in $22 and $23. As an interesting exercise it may be of use to disassemble this subroutine and try and follow through the various stages. This can be done by selecting 'L' for List at the MENU and then listing from 47756.

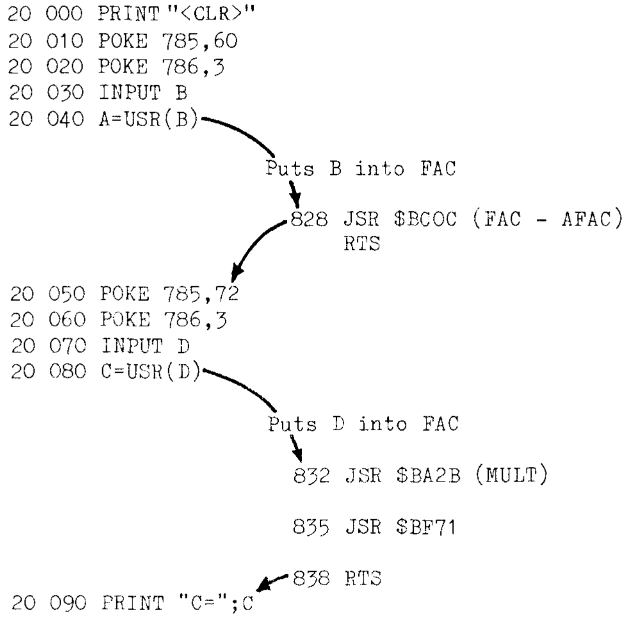
These subroutines are just collections of 6510 instructions, much as you should be writing by now. However, they very cleverly written and use bits of other subroutines to save space, and it is these JSR and JMP jaunts that you may find difficult to follow.

EXERCISE 8.2

Write a program suite to input the numbers 1.047 and 4038.22 into a machine code program. Multiply these together and then take the square-root of the sum. Print out this answer on the screen from BASIC.

Second thoughts!!...

This is not as easy as it seems; the original plan looked like this:



```
20 000 PRINT"<CLR>"
20 010 POKE 785,60
20 020 POKE 786,3
20 030 INPUT B
20 040 A=USR(B)
                      Puts B into FAC

                        828 JSR $BC0C (FAC - AFAC)
                            RTS

20 050 POKE 785,72
20 060 POKE 786,3
20 070 INPUT D
20 080 C=USR(D)
                      Puts D into FAC

                        832 JSR $BA2B (MULT)

                        835 JSR $BF71

                        838 RTS
20 090 PRINT "C=";C
```

That was the plan; however, it won't work! It fails because the program suite assumes that the FAC's contents stay put while they are, in fact, constantly changing as BASIC runs a program. Following line 20040 FAC contains B and after the JSR (FAC- AFAC) both FAC and AFAC contain B. However, in executing lines 20050 to 20080 the operating system utilises FAC and AFAC, and thus changes the contents of these. Thus when JSR (MULT) is called, the contents of FAC and AFAC that are multiplied together are not those expected.

The problem can only be overcome by saving AFAC in memory while returning to BASIC. This is done by means of the subroutine at $BBC7. This subroutine copies the contents of AFAC into the five bytes of memory starting at the address stored in $49 and $4A. Program 8.15 illustrates this, putting AFAC into $0384 onwards.

PROGRAM 8.15

```
LDAIM 132  ⎫ Load LSB of address - 132
STAZ  $49  ⎭ ($84) into $49.

LDAIM 3    ⎫ Load MSB of address - 3
STAZ  $4A  ⎭ ($03) into $4A.

JSR   $BBC7
RTS
```

COPY AFAC INTO MEMORY

The action of this can be checked by running the direct program:
FOR X=0 TO 5 : PRINT PEEK(900+X);: NEXT X

To use this subroutine in Exercise 8.2, it is necessary to reload the data into AFAC. This is done by the subroutine at $BA8C (47756).

In order to operate, the subroutine needs to be told where to find the data and this is done by loading the address of the first byte of data into the accumulator (LSB) and the Y register (MSB). Thus a "reload AFAC" program to recover data from 900 onwards would look like:

PROGRAM 8.16

```
LDAIM 132     Load LSB of address.
LDYIM 3       Load MSB of address.
JSR $BA8C     Load AFAC from memory.
RTS
```

Perhaps now is a suitable time to try Exercise 8.2. One possible answer is given on page 9-12.

Other Available Subroutines

Appendix III lists altogether 17 subroutines that have been found most useful when dealing with floating-point numbers from machine code. Those not yet discussed are covered below:

Addition

Using the subroutine at $B86A (47210), the floating-point numbers in FAC and AFAC are added together and the sum loaded into FAC. This is demonstrated in programs 8.17 and 8.18.

PROGRAM 8.17

```
20 000 PRINT "<CLR>"
20 010 POKE 1,60
20 020 POKE 2,3
20 030 INPUT B
20 040 A=USR(B)
20 050 RUN 20 060
20 060 POKE 1,72
20 070 POKE 2,3
20 080 INPUT D
20 090 C=USR(D)
20 100 PRINT"C=";C
```

PROGRAM 8.18

```
828 LDAIM 132
    STAZ  72
    LDAIM 3
    STAZ  74       }      Store FAC in memory.
    JSR   $BBC7
    RTS

840 LDAIM 132             Retrieve data from
    LDYIM 3        }      memory, store in AFAC.
    JSR   $BA8C

847 JSR   $B86A    }      Addition subroutine.
    RTS
```

ADD TWO FLOATING-POINT NUMBERS

When the program at 20000 is run, it requests two inputs and then prints out the sum of these.

Subtraction

Programs 8.17 and 8.18 can be used to demonstrate this by inserting the subroutine $B853 (47187) at 848 and 849, i.e.

PROGRAM 8.19 (Part)

```
847 JSR $D853
```

This can be run by a RUN 20000 and will print out the answer. It subtracts the second input from the first.

Division

Once again we may use programs 8.17 and 8.18 to demonstrate the use of the Division routine at $BB12 (47890) - replace 848 and 849 as before.

PROGRAM 8.20 (Part)

```
847 JSR $DB12
```

and RUN 20000. The program divides the first input by the second.

Exponentiation

The exponentiation routine is at $BF7B (49019) - so replace 848 and 849 once again.

PROGRAM 8.21 (Part)

```
847 JSR $DF7B
```

and RUN 20000. The program raises the first inputted number to the power of the second input.

Other routines need only one input and operate on this alone; these are:

Log

The subroutine is at $B9EA (47594) and computes the natural logarithm or $\log_e$ (log to the base e). Program 8.22 demonstrates this subroutine in use.

PROGRAM 8.22

```
828 JSR $B9EA
831 RTS
```

It is called by program 8.23.

PROGRAM 8.23

```
20 000 PRINT"<CLR>"
20 010 POKE 785,60
20 020 POKE 786,3
20 030 INPUT B
20 040 A=USR(B)
20 050 PRINT"A=";A
```

A RUN 20000 activates both routines and prints out $\log_e$ of the input value.

```
The three functions SIN $E26B (57963)
                    COS $E264 (57956)
                and TAN $E2B4 (58036)
```

can be incorporated into a program suite such as 8.22/23. In each case, the input value is entered in radians and the computed function is returned in FAC.

Appendix III contains a very comprehensive list of the many subroutines which exist within the CBM 64 ROM, and I suspect that you find the list somewhat bewildering at first sight. As you develop more experience, however, I feel sure that you will find the list more and more useful, so don't be put off.

However, when using these built-in subroutines, the major problem lies in knowing exactly how to integrate these into any particular program as the subroutines' start-points are well documented. These problems broadly revolve around where the subroutine gets its data from and where it deposits its results. On the latter point, as will have been seen, the FAC is a common place in which to place results.

As to the linking in of data, the first few lines of a subroutine should give the clue. First, these lines should be disassembled using the assembler LIST function. Next these should be examined for the first use of A, X and Y as once these are re-loaded in the subroutine, any data originally in them will have been destroyed. If addresses are involved certain zero-page locations are popular such as $22/$23 (34/35) as well as $49/$4A (73/74). Appendix III also contains a list of the locations in the zero-page and following three pages together with a brief description of their uses.

Similarly, the deposition of processed data will be revealed by the last few instructions and if FAC and AFAC are involved, their memory locations should feature. The end of the subroutine is usually an RTS instruction but it might be a JMP instruction to another of the ROM subroutines, the actual return being the result of the 'called' routines RTS.

Finally, if logic (and all else!) fails there's always the 'shoot or bust' technique left, i.e. just trying it. If this is attempted, keep good records as constant reloading after crashes can be time-consuming. Of course, after a crash the machine-code program is not lost provided control can be regained by use of the STOP/RESTORE keys. However, if you have to resort to turning the CBM 64 off and then on again to regain control however, then the chances are that you will lose the machine code program in memory. The moral is to use the T=TAPE option of the Dr. Watson Assembler to save any lengthy routine before risking a run.

Remember that the subroutines in ROM are only machine-code like that that you have produced and once disassembled should be quite comprehendable, if at times somewhat involved. Don't be afraid, therefore, to use things not detailed here! The 'experts' gained their exalted status by being inquisitive and trying it out!

CHAPTER NINE

Solution to Exercises

Chapter One

Exercise 1.1 (Page 1-10)

Since the accumulator contains a '1' it seems reasonable to use this to print a white 'A'.

```
START ADDRESS? 828
LDAIM 1
STA   1024
STA   55296
RTS
END
```

Exercise 1.2 (Page 1-10)

Assuming that your name is FRED and you wish to print this in black, then your solution might look like the following:

```
START ADDRESS? 828
LDAIM 6
STA   1024
LDAIM 0
STA   55296
LDAIM 18
STA   1025
LDAIM 0
STA   55297
LDAIM 5
STA   1026
LDAIM 0
STA   55298
LDAIM 4
STA   1027
LDAIM 0
STA   55299
RTS
END
```

This is a perfectly good solution and does what it sets out to do. However, you may think that there is scope for improvement, and there is. The LDAIM 0 occurs four times in the solution above, and only once in the following:

Exercise 1.2 (alternative solution).

```
START ADDRESS? 828
LDAIM 6
STA   1024
LDAIM 18
STA   1025
LDAIM 5
STA   1026
LDAIM 4
STA   1027
LDAIM 0
STA   55296
STA   55297
STA   55298
STA   55299
RTS
END
```

It doesn't matter that we have separated the setting of the colour from the insertion of the character. Machine code operates so fast, that you haven't the slightest chance of seeing the small interval between the two.

Exercise 1.3 (Page 1-10)

```
START ADDRESS? 828
LDAIM 24
STA   1024
STA   1063
STA   1984
STA   2023
LDAIM 2
STA   55296
STA   55335
STA   56256
STA   56295
RTS
END
```

Exercise 1.4 (Page 1-13)

```
START ADDRESS? 826
LDAIM 26
LDXIM 1
STX   900
TAX
LDA   900
STA   1024
STX   2023
STA   55296
STA   56295
RTS
END
```

The characters had to be printed in white if the solution wasn't to have a third Immediate Mode instruction.

Exercise 1.5 (Page 1-13)

```
START ADDRESS? 828
LDAIM 90
LDXIM 42
LDYIM 5
STX   900
TAX
TYA
LDY   900
STA   1024
STA   1063
STX   1984
STY   2023
STA   55296
STA   55335
STA   56256
STA   56295
RTS
END
```

Once again, in order to comply with the condition that only three LD?IM instructions were to be used, this solution uses the 5 ('E') to provide the colour, which turns out to be green.

Chapter Two

Exercise 2.1 (Page 2-5)

i)

```
START ADD? 828
LDAIM 3
JSR   900
STA   1024
LDAIM 1
STA   55296
RTS
END
```

ii)

```
START ADD? 900
STA   950
ADC   950
RTS
END
```

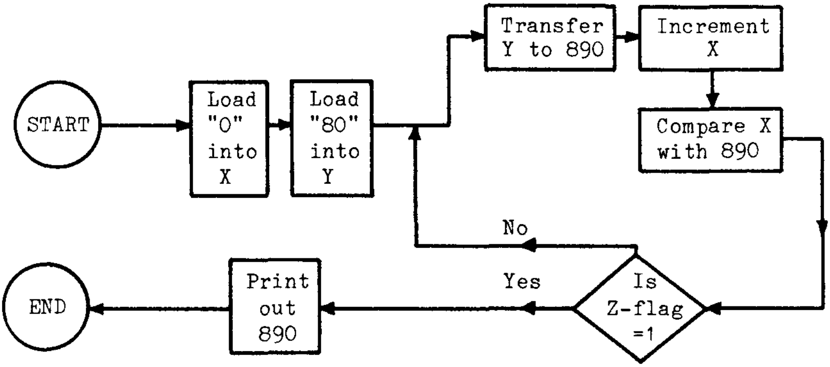
Exercise 2.2 (Page 2-7)

```
START ADD? 828
LDYIM 100
DEY
BEQ   3
JMP   830
STY   1024
STY   55296
RTS
END
```

Exercise 2.3 (Page 2-10)

```
START ADD? 828
LDAIM 83
STA   890
LDYIM 0
INY
CPY   890
BEQ   3
JMP   835
STY   1034
LDAIM 4
STA   55396
RTS
END
```

Exercise 2.4 (Page 2-12)



```
START ADD? 828
LDXIM 0
LDYIM 80
STY   890
INX
CPX   890
BNE   250
STX   1024
LDAIM 1      } This program prints the result in white
STA   55296  } - you could use any other colour.
RTS
END
```

Exercise 2.5 (Page 2-15)

```
START ADDRESS? 828
LDXIM 90
DEX
CPX   900
BPL   250
STX   1024
STX   55296    Prints in black as X contains 0.
RTS
END
```

Chapter Three

Exercise 3.1 (Page 3-1)

The LDXIM instruction at 828 must be changed to a LDYIM. From the Table 1 (Appendix I) the object code for this is 160, so a

```
        POKE 828,160
```

does the trick.

The STAX at 832 has to be changed to an STAY (object code 153)

```
i.e.    POKE 832,153
```

The STAX at 837 has to be changed to a STAY (object code 153)

```
i.e.    POKE 837,153
```

Lastly, the DEX at 840 needs to be replaced by a DEY (object code 136)

```
i.e.    POKE 840,136
```

Thus the final program reads:-

```
        LDYIM 100
        LDAIM 90
        STAY  1023
        LDAIM 1
        STAY  55295
        DEY
        BNE   243
        RTS
        END
```

Exercise 3.2 (Page 3-2)

```
        LDXIM 100      Load X with '100'.
        STX   900      Store X in 900.
        LDXIM 0        Load X with '0'.
        LDAIM 42       Load A with 42 (asterisk).
        STAX  1023     Store A in (1023 + X).
        LDAIM 1        Load A with 1 (white).
        STAX  55295    Store A in (55295 + X).
        INX            Increment X.
        CPX   900      Compare X with 900.
        BNE   242      Branch if not equal.
        RTS            Return from subroutine.
        END
```

Chapter Four

Exercise 4.1 (Page 4-6)

$$\begin{aligned} 1807_{16} &= 1 \times 4096 + 8 \times 256 + 0 \times 16 + 7 \times 1 \\ &= 4096 + 2048 + 07 \\ &= 6151_{10} \end{aligned}$$

$$\begin{aligned} 2AFA_{16} &= 2 \times 4096 + 10 \times 256 + 15 \times 16 + 10 \times 1 \\ &= 11002_{10} \end{aligned}$$

i.e. $6151_{10} + 11002_{10} = 17153_{10}$

Program to add

```
START ADD? 828
CLC
CLD
LDAIM $07
ADCIM $FA
STA   1026
LDXIM 1        } Using X for the colour saves
STX   55298    } an instruction later on.
LDAIM $18
ADCIM $2A
STA   1024
STX   55296
RTS
END
```

Screen display gives answer of - and A, i.e. 67_{10} and 01_{10}.

Converting this to hex gives:

$67_{10} = 4 \times 16 + 3 \times 1 = \$43$

$01_{10} = 0 \times 16 + 1 \times 1 = \$01$

Thus 67_{10} and 01_{10} $= \$4301$
$= 4x4096+3+256+0x16+1$

$= 17153_{10}$

Cross-checking this by hex addition:

```
  1807
+ 2AFA
= 4301
```

Exercise 4.2 (Page 4-7)

```
START ADDRESS? 828
CLD
SEC
LDAIM 32
SBCIM 88
STA   1036
LDXIM 1
STX   55308
LDAIM 3
SBCIM 2
STA   1034
STX   55306
RTS
END

Ans = @ and █
    = 0 and 200
```

Exercise 4.3 (Page 4-8)

```
CLD            Clear decimal to make sure in
               binary mode.
CLC            Clear carry flag.

LDAIM $2C      Load LSB 1.          ⎫ Add LSB's
ADCIM $90      Add LSB 2 to A.      ⎬ together.
STA   900      Sum of LSB's.        ⎭

LDAIM $01      Load MSB 1.          ⎫ Add MSB's
ADCIM $01      Add MSB 2.           ⎬ together incl.
STA   901      Sum of MSB's.        ⎪ incl. carry
                                    ⎭ (if there).
SEC            Set carry in preparation
               for subtraction.

LDA   900      Load LSB 3 (of 500)  ⎫ Subtract
SBCIM $F4      Subtract LSB 1+2     ⎪ LSB's
               from LSB3.           ⎬ and
STA   1041     Display sum of LSB's ⎪ display.
LDXIM 1        in white.            ⎪
STX   55313                         ⎭

LDA   901      Load MSB 3 (of 500)  ⎫ Subtract
SBCIM $01      Subtract MSB 1+2     ⎪ MSB's
               from MSB 3.          ⎬ and
STA   1040     Display sum MSB's    ⎪ display.
STX   55312    in white.            ⎭


RTS
END
```

When run this should display @ and █ or 0 and 200.

Exercise 4.4 (Page 4-12)

A = 1 B = 0 C (output) = 0
A = 0 B = 1 C = 0
A = 1 B = 1 C = 1

Exercise 4.5 (Page 4-13)

$$\begin{array}{rcl} 149_{10} & = & 10010101_2 \\ 52_{10} & = & \underline{00110100_2} \\ \text{149 AND 52} & = & \underline{\underline{00010100_2}} \\ & = & 16 + 4 \\ & = & 20_{10} \end{array}$$

Exercise 4.6 (Page 4-17)

	128	64	32	16	8	4	2	1
100 =	0	1	1	0	0	1	0	0
87 =	0	1	0	1	0	1	1	1
75 =	0	1	0	0	1	0	1	1
99 =	0	1	1	0	0	0	1	1
57 =	0	0	1	1	1	0	0	1
94 =	0	1	0	1	1	1	1	0
27 =	0	0	0	1	1	0	1	1

Thus:

i)

100	0	1	1	0	0	1	0	0
AND 87	0	1	0	1	0	1	1	1
=	0	1	0	0	0	1	0	0

$= 01000100_2$

$= 0 + 64 + 0 + 0 + 0 + 4 + 0 + 0 = 68_{10}$

```
LDAIM 100
ANDIM 87
STA   1024
LDXIM 1
STX   55296
RTS
END
```

When run gives a white - (68_{10})

ii)

75	0	1	0	0	1	0	1	1
OR 27	0	0	0	1	1	0	1	1
=	0	1	0	1	1	0	1	1

= 01011011

= 0 + 64 + 0 + 16 + 8 + 0 + 2 + 1 = 91_{10}

```
LDAIM 75
ORAIM 27
STA   1024
LDXIM 1
STX   55296
RTS
END
```

When run gives a white + (91_{10})

iii)

99	0	1	1	0	0	0	1	1
EOR 57	0	0	1	1	1	0	0	1
=	0	1	0	1	1	0	1	0

= 01011010

= 0 + 64 + 0 + 16 + 8 + 0 + 2 + 0 = 90_{10}

```
LDAIM 99
EORIM 57
STA   1024
LDXIM 1
STX   55296
RTS
END
```

When run gives a white diamond (90_{10}).

iv)

100	0	1	1	0	0	1	0	0
AND 87	0	1	0	1	0	1	1	1
=	0	1	0	0	0	1	0	0
EOR 94	0	1	0	1	1	1	1	0
=	0	0	0	1	1	0	1	0

= 00011010

= 0 + 0 + 0 + 16 + 8 + 0 + 2 + 0 = 26_{10}

```
LDAIM 100
ANDIM 87
EORIM 94
STA   1024
LDXIM 1
STX   55296
RTS
END
```

When run gives a white Z (26_{10}).

Exercise 4.7 (Page 4-18)

86_{10} in BCD = 10000110

which in binary = 134_{10}

Thus a program is:

```
LDAIM 134
ANDIM 15
STA   1025
LDXIM 1
STX   55297
LDYIM 4
LDAIM 134
LSRA
DEY
BNE   252
STA   1024
STX   55296
RTS
END
```

When run this gives an output of a white H and F (i.e. 86).

Exercise 4.8 (Page 4-22)

Written to multiply together the numbers 3 and 4.

```
         LDXIM 3
         STX   901
         LDXIM 4
         STX   902
         LDYIM 8

         below here as program 4.12

         LDAIM 0
         CLC
         LSR   902
         BCC   4
         CLC
         ADC   901
         ASL   901
         DEY
         BNE   240
         STA   1024
         LDXIM 1
         STX   55296
         RTS
         END
```

When run this should print a white L (12) in 1024.

Chapter Five

Exercise 5.1 (Page 5-3)

```
                                  OR
         LDXIM 80                           LDXIM 80
         LDYIM 2                            LDYIM 2
         JMP   LOOP3                        JMP   865

 *LOOP1  LDAIM 83                           LDAIM 83
         STAX  1183                         STAX  1183
         TYA                                TYA
         STAX  55455                        STAX  55455
         DEX                                DEX
         BNE   LOOP1                        BNE   244
         JMP   END                          JMP   883

 *LOOP2  LDAIM 90                           LDAIM 90
         STAX  1023                         STAX  1023
         TYA                                TYA
         STAX  55295                        STAX  55295
         DEX                                DEX
         BNE   LOOP2                        BNE   244
         JMP   LOOP1                        JMP   835
```

```
*LOOP3  LDAIM 42                 LDAIM 42
        STAX  1223               STAX  1223
        TYA                      TYA
        STAX  55495              STAX  55495
        DEX                      DEX
        BNE   LOOP3              BNE   244
        LDXIM 120                LDXIM 120
        DEY                      DEY
        JMP   LOOP2              JMP   850

*END    RTS                      RTS
        END
```

Chapter Seven

Exercise 7.1 (Page 7-6)

```
POKE 829,32  (If program loaded from 828)
```

The program should then read:-

```
LDXIM 32
STX   900
JSR   65508
BEQ   251
JSR   65490
CMP   900
BNE   243
LDAIM 13
JSR   65490
RTS
END
```

Exercise 7.2 (Page 7-11)

```
LDXIM 4
TXA
PHA
PHP
JSR $FFE4
BEQ 251
JSR $FFD2
PLP
PLA
TAX
DEX
BNE 239
RTS
END
```

In the above program, SR could have been loaded into stack before the X register (via the accumulator), as long as it was then retrieved first, i.e. LIFO operation.

Chapter Eight

Exercise 8.1 (Page 8-18)

The data, +2000 is to be stored from 900_{10} (0384_{16}) onwards. This data is:

```
        139   122    0    0    0
```

It can be entered using:

PROGRAM 8.12

```
POKE 900,139
POKE 901,122
POKE 902,0
POKE 903,0
POKE 904,0
```

The machine code routine is:

PROGRAM 8.13

```
LDAIM 132       Set up address for subroutine.
LDYIM 3         03₁₆=03₁₀ : 84₁₆=132₁₀
JSR   $DA8C     Put data into AFAC from memory.
LDXIM 6
LDAX  96
STAX  1423      Print FAC onto screen.
LDAIM 1
STAX  55695
DEX
BNE   242
JSR   $DBFC     Transfer data AFAC→FAC.
LDXIM 6
LDAX  96
STAX  1623
LDAIM 1
STAX  55895
DEX
BNE   242
RTS
END
```

TRANSFER AFAC to FAC

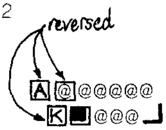
The BASIC program is:

PROGRAM 8.14

```
20 000 PRINT "<clear>"
20 010 POKE 785,60
20 020 POKE 786,3
20 030 B=1               (for example)
20 040 A=USR(B)
20 050 PRINT "A=";A
20 060 RUN 20070
20 070 FOR X=0 TO 5
20 080 PRINT PEEK(1424+X);" ";
20 090 NEXT X
20 100 PRINT
20 110 FOR X=0 TO 5
20 120 PRINT PEEK(1624+X);" ";
20 130 NEXT X
```

When run this should print out:

```
A= 2000
  129  128  0  0  0  0
  139  250  0  0  0  122
READY
```



Exercise 8.2 (Page 8-19)

```
20 000 PRINT "<clear>"
20 010 POKE 785,60
20 020 POKE 786,3
20 030 INPUT B
20 040 A=USR(B)
20 050 RUN 20060
20 060 POKE 785,70
20 070 POKE 786,3
20 080 INPUT D
20 090 C=USR(D)
20 100 PRINT "C=";C
```

```
LDAIM 132
STAZ  73
LDAIM 3
STAZ  74
JSR   $ABC7
RTS
LDAIM 132
LDYIM 3
JSR   $AA8C
JSR   $A82B
JSR   $AF71
RTS
END
```

Appendix One

Exercise A1.1 (Page A1-2)

i) $0000\ 0011_2 = 0+0+0+0+0+0+2+1$
$= 3_{10}$

ii) $0000\ 0100_2 = 0+0+0+0+0+4+0+0$
$= 4_{10}$

iii) $1000\ 0000_2 = 128+0+0+0+0+0+0+0$
$= 128_{10}$

iv) $1000\ 0011_2 = 128+0+0+0+0+0+2+1$
$= 131_{10}$

v) $1011\ 0111_2 = 128+0+32+16+0+4+2+1$
$= 183_{10}$

vi) $0111\ 0011_2 = 0+64+32+16+0+0+2+1$
$= 115_{10}$

Exercise A1.2 (Page A1-5)

i) $0009_{16} = 0x409+0x256+0x16+9x1$
$= 0+0+0+9$
$= 9_{10}$

ii) $0013_{16} = 0x4096+0x256+1x16+3x1$
$= 0+0+16+3$
$= 19_{10}$

iii) $00A5_{16} = 0+0+10x16+5x1$
$= 160+5$
$= 165_{10}$

iv) $0AAE_{16} = 0+10x256+10x16+14x1$
$= 2560+160+14$
$= 2734_{10}$

v) $000E_{16} = 0+0+0+14$
$= 14_{10}$

vi) $011A_{16} = 0+256+16+10$

$= 282_{10}$

vii) $00EA_{16} = 0+0+14\text{x}16+10$

$= 224+10$

$= 234_{10}$

viii) $F0A3_{16} = 15\text{x}4096+0+10\text{x}16+3$

$= 61440+160+3$

$= 61603_{10}$

Exercise A1.3 (Page A1-7)

i) $4_{10} = 0100_2\text{(BCD)}$

ii) $10_{10} = 1\text{x}10+0$

iii) $77_{10} = 7\text{x}10+7$

$= 0111\ 0111_2\text{(BCD)}$

iv) $97_{10} = 9\text{x}10+7$

$= 1001\ 0111_2\text{(BCD)}$

v) $53_{10} = 5\text{x}10+3$

$= 0101\ 0011_2\text{(BCD)}$

vi) $102_{10} = 1\text{x}100+0\text{x}10+2\text{x}1$

$= 0001\ 0000\ 0010_2\text{(BCD)}$

vii) $953_{10} = 9\text{x}100+5\text{x}10+3\text{x}1$

$= 1001\ 0101\ 0011_2\text{(BCD)}$

viii) $2579_{10} = 2\text{x}1000+5\text{x}100+7\text{x}10+9\text{x}1$

$= 0010\ 0101\ 0111\ 1001_2\text{(BCD)}$

Exercise A1.4 (Page A1-8)

i) $0000\ 0001_2\text{(BCD)} = 0\text{x}10+1\text{x}1$

$= 1_{10}$

ii) $0000\ 1001_2\text{(BCD)} = 0\text{x}10+9\text{x}1$

$= 9_{10}$

iii) $0001\ 0101_2\text{(BCD)} = 1\text{x}10+5\text{x}1$

$= 15_{10}$

iv) $0010\ 0000_2 = 2\text{x}10+0\text{x}1$

$= 20_{10}$

v) $0100\ 1001_2\text{(BCD)} = 4\text{x}10+9\text{x}1$

$= 49_{10}$

vi) $1010\ 0011_2$(BCD)

*** This is not a valid BCD number as the first nybble, $1010_2 = 10_{10}$, i.e. is greater than allowed in BCD.

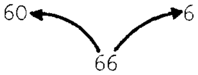
vii) $1001\ 0111_2$(BCD) = 0x10+7x1
= 97_{10}

viii) $1000\ 1000_2$(BCD) = 8x10+8x1
= 88_{10}

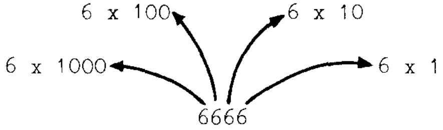
APPENDIX ONE

Binary, Binary-Coded Decimal & Hexadecimal Notation

Counting systems in general use throughout the world use the decimal system and this has been developed to count up to and beyond 10 and also below the value 1. In this standard the digits to the left of a number are of greater value than those to the right. For instance, in the number 66, the first 6 has a value 10 times the second, i.e.

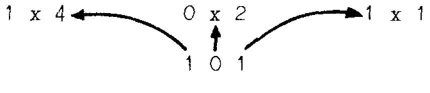


This is extended in larger numbers where digits to the left are successively greater by a multiple of ten, i.e.



A system where the position or place of a digit in a number affects its value is known as a PLACE-VALUE numbering system. In the decimal system, the values of digits increase in multiples of 10 and this is known as the BASE for that system. Other systems use different bases but follow the same pattern as the decimal system, i.e. the place to the left is greater by being multiplied by the base.

The computer, being basically electronic in operation, works better if it is told to only recognise two states, on or off or '0' and '1', and thus uses the Binary system - base 2. Thus, any number in binary consists simply of 0's and 1's, or electronically, zero volts (off) and some volts (on). To count past one, the binary system must resort to place-value notation and, as with other cases, the multiplying factor is the base, i.e. 2. Thus, the number 101 in base 2 or binary represents:



i.e. 4+0+1=5. Clearly the plethora of bases presents a problem when representing numbers as in base 10, '101' represents one hundred and one while in binary (base 2) '101' represents 5. To overcome this ambiguity, a convention exists when representing numbers in that the base is written to the right of the number, just below the line. Thus, the two numbers discussed above become:

101_{10} = One hundred and one in base ten.

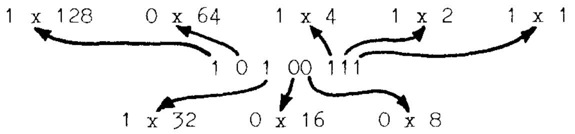
101_2 = Five in base two.

The present-day generation of personal computers (1983-style) use eight bit registers or memories and can thus represent numbers up to 11111111_2, i.e. in base 10:

128	+64	+32	+16	+8	+4	+2	+1	$=255_{10}$
1	1	1	1	1	1	1	1	Digit
128	64	32	16	8	4	2	1	Equivalent in base 10

Fig. A1.1

By way of example, let's take one more conversion - say, 101010111_2.



Thus 10100111 = 1x128 +0x64 +1x32 +0x16 +0x8 +1x4 +1x2 +1x1
= 128+32+4+2+1
= 167_{10}

Just to check your understanding, have a go at the following:

EXERCISE A1.1

Calculate the value of the following in base 10:-

i) 00000011_2
ii) 00000100_2
iii) 10000000_2
iv) 10000011_2
v) 10110111_2
vi) 01110011_2 Answers on page 9-16.

If you remain unclear on this, or simply want to see it demonstrated, load and run the Binary/Hex tutor program which is included on the assembler tape. At the menu select 'H' for "Decimal, Binary and Hexadecimal". Then, when asked "At what number do you wish to start?", press "1" <return>. The screen will then display three rows of boxes, of which the top two are currently of interest. These represent a decimal number (marked "DEC") and a binary number (marked "BIN"). At this stage, they should contain the numbers 1_{10} and 1_2. The decimal number has three digits and thus has a capacity of 999_{10}, and the binary, with its eight binary digits (bits) will hold up to 255_{10}.

From this point, the program will simply count, every time that you press the space bar, both the decimal and the binary boxes will index one. Try pressing the space bar once and the boxes should contain a 2_{10} and a 10_2. If you carry on indexing then you will see how binary counts. When you get to the stage where the decimal shows 15_{10} the binary should read 1111_2. Now index one further and the binary will change to 10000_2. One way of looking at this is to lay out the addition:-

```
  1111   A
+    1   B
```

On adding the 1 (A) to the 1 (B) this gives '2' i.e. 0, carry 1. This carry then produces another '0' plus another carry, and so on.

If you continued to press the space bar long enough, then eventually the binary register would become full. However, this would take an awful lot of pressing, so we will take a short cut to this state of affairs. Instead of pushing the space bar, press the RETURN key instead. This will return you to the menu where you can select the 'H' option again. This time, when asked "at what number do you want to start?" type a fairly high value which is less than 255, say 240. Off you go again until the binary register is full i.e. 1111111111_2. The addition of a further one, now, will clock all the binary register back to zeros and 256 will be lost. However, with the 6510, all is not lost as the 6510 has a carry flag that stores the fact that a carry has occurred. Clocking past 255_{10} with the Binary/Hex tutor will show this happening. This is a handy feature of the 6510 but it must not be relied on as more than a temporary store of the carry. The carry flag is just as easily reset as it is set to 1!

In order to make sure that you really understand the binary notation, you may wish to try some of the exercises which are provided by the BIN/HEX exercises. Select 'E' at the main menu. This will provide you with a menu of exercises and you can select '2' to try the exercises converting decimal numbers into binary or '5' to try converting them back again. When you are running the exercises, by the way, typing a space (instead of a digit) will delete the last entry that you made, thus providing you with a correction facility. When you are satisfied that you have done enough, pressing the <return> key will take you back to the main menu.

While the 0's and 1's are convenient for the computer, they are much less so for the mere human so a compromise is sought. Decimal notation is of little use as, apart from 1_2 and 1_{10} there is no other correspondence. A further idea would be to take the whole eight binary bits as a digit (i.e. up to 255_{10}) and use a base of 256! What would you see as the objection to this? That's apart from the idea itself being a bit mind-bending! Time to think ... The answer comes from an examination of the base 10 case in which ten digits (0 to 9) are needed to represent the ten steps up to 10. In the base 2 system, two digits are needed so base 256 would need 256 digits!

A compromise system adopted splits the eight bits up into two parts and represents these separately. Thus, the largest number to be represented is 1111_2 or 15_{10} and this requires, along with the 0, sixteen different symbols. The ones adopted for this job are:

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

Fig. A1.2

Using this notation, any eight bit number can be represented by two symbols, one for the most significant four bits and one for the least significant four bits. To avoid the rather long description of these two halves of a byte, they are given the term NYBBLES. Thus a byte consists of two nybbles, a most significant nybble (MSN) and a least significant nybble (LSN) - see Fig. A1.3.

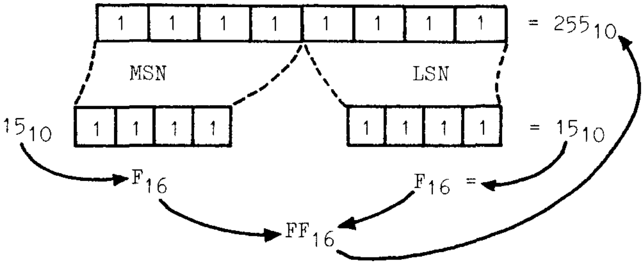


Fig. A1.3

The system described, which uses sixteen symbols is, of course, given the name HEXADECIMAL - usually abbreviated to HEX. Its major advantage, as far as computers are concerned, is that it is compatible with binary. Any eight bit binary number can be represented by two hexadecimal characters.

You are now in a position to look at the Binary/Hex tutor program again. The third row of boxes, which we ignored last time round, contains the Hex numbers. While the counting is going on in the binary boxes, so it is in the Hex boxes also. The comparability between binary and HEX shows wherever a major carry occurs - take for instance 1111_2, 15_{10} or F_{16}: one index past this clocks the binary ones to zeroes and adds a one to the left, i.e. to 10000_2 or 10_{16}. These major points of correspondence occur at

$$1_2 = 1_{16} = 1_{10}$$

$$0001\ 0000_2 = 10_{16} = 16_{10}$$

$$0000\ 0001\ 0000\ 0000_2 = 100_{16} = 256_{10}$$

$$0001\ 0000\ 0000\ 0000_2 = 1000_{16} = 4096_{10}$$

Up to 9, the hex characters coincide with the decimal ones and between 10 and 15 the single letters correspond to the decimal numbers. After 15, Hex to decimal conversion becomes a little more tricky, as the use of two numbers together, e.g. $FF_{16}=255$, once again calls for place-value notation. This time, as the base is 16 the ratio between any place and its neighbour is 16.

The values, in base 10 of the places in hexadecimal are:

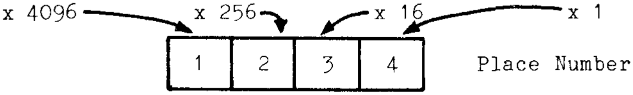


Fig. A1.4

Using Fig. A1.4 the way that $E92F_{16}$ makes up 59695_{10} is explained below in figure A1.5.

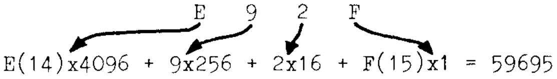


Fig. A1.5

Now that hex is totally mastered! try the following; the first two are explained fully on page 9.16.

EXERCISE A1.2

Calculate the value in decimal of the following:-

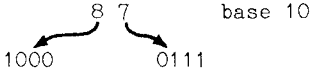
i) 0009_{16}
ii) 0013_{16}
iii) $00A5_{16}$
iv) $0AAE_{16}$
v) $000E_{16}$
vi) $011A_{16}$
vii) $00EA_{16}$
viii) $F0A3_{16}$

Answers on page 9-16.

Binary Coded Decimal

As well as decimal, binary and hexadecimal notations, one other system is used in computing - binary-coded decimal. As its name suggests it is a hybrid form with elements from binary and decimal. It is commonly used where an output is required in digital format, e.g. a digital clock, or when great precision is required and no bits can be dropped.

In BCD the normal decimal base is retained, i.e. one place is a factor of 10 times its neighbour but each individual digit is represented in binary. Thus the number 87_{10} would be represented:



i.e. BCD = 1000 0111 (or in eight bits 10000111)

Fig. A1.6

As the largest digit required in decimal notation is 9, only four bits of binary are needed to represent this, i.e. $9_{10}=1001_2$, thus a BCD digit can be represented by a nybble and two digits by a byte. Figure A1.6 shows this, where 87_{10} is represented in BCD as 10000111_2. This can give rise to ambiguity in that 10000111_2 in binary is 135_{10}. To overcome this, BCD representations will be given the notation 10000111_2(BCD).

Using four bits of binary, it is possible to count up to 15_{10} (i.e. $1111_2=15_{10}$) but in BCD the largest digit used is 9, so inevitably BCD is less economical in its use of space. Its largest digit, 9, is 1001_2 and when one is added to this it clocks over to 0000_2 and carries the 1 to the next nybble, i.e.

8_{10}	=	0000 1000	(base 2 BCD)
9_{10}	=	0000 1001	" " "
10_{10}	=	0001 0000	" " "
11_{10}	=	0001 0001	" " "

Fig. A1.7

It would probably be helpful at this point if you load and run the Binary/Hex tutor program again. This time, select 'B' at the main menu, and when asked "At what number..." enter a 1 <return>.

The display will then show three rows of boxes again but this time they will contain decimal, binary and BCD. If you press the space bar as before, to index from '1', you will notice that up to 9_{10}, binary and BCD are identical.. However, as you index from 9_{10} to 10_{10} keep an eye on the BCD box and you will see the 1 carried over to the most significant nybble. From 10_{10} upwards BCD becomes a true hybrid representing the decimal number in a binary form.

As the number increases, the uneconomical nature of BCD will become apparent as 99_{10} changes to 100_{10}. (As before, typing <return> instead of the space bar will get you back to the main menu, which will allow you to restart at a value nearer to 99_{10}.) When 99_{10} indexes to 100_{10} you will see the BCD generate a carry from its most significant nybble to the carry flag.

As mentioned above, this carry is only a short term expedient and must be picked up at the earliest possible moment if it is not to be lost. The carry is generated on the BCD boxes at 99_{10} while the binary boxes will store up to 255_{10}. BCD is therefore fairly uneconomical in memory usage, but it has its uses in particular situations. In the past, microcomputers have always been dogged by their lack of memory and consequently BCD has been little used. However, the new generation of microcomputers have much larger memories and it is quite likely that BCD will be used much more frequently than it was in the past. Perhaps it is a sign of the times, that, although all COMMODORE computers have had BCD capability, your CBM 64 is the first to make use of BCD, albeit in a very small way. The 24 hour time of day clocks which are built into the 64's two input/output chips (6526's) do make use of BCD.

As you know all about BCD now! try the following:-

EXERCISE A1.3

Convert the following decimal numbers into BCD:

i) 4
ii) 10
iii) 77
iv) 97
v) 53
vi) 102
vii) 953
viii) 2579

Answers on page 9-17.

EXERCISE A1.4

Convert the following BCD numbers into decimal:-

i) 0000 0001
ii) 0000 1001
iii) 0001 0101
iv) 0010 0000
v) 0100 1001
vi) 1010 0011
vii) 1001 0111
viii) 1000 1000

Answers on page 9-17.

In the explanations given of the value of places in place-value notation a simplification was adopted in order to make these explanations clearer for our less mathematically-inclined brethren. However, if you wish to see a slightly more mathematical explanation, please read on. Otherwise - END OF APPENDIX I.

With binary numbers it was said that the places increase their value in multiples of 2, but the least significant bit of the binary number was equivalent to the same symbol base 10 (or for that matter base 3, or whatever). In actual fact the multiplying factor is the base raised to the power of its place starting with zero at the left. i.e. in binary:

7	6	5	4	3	2	1	0	Place
128	64	32	16	8	4	2	1	Previously stated multiplication factor
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Mathematically more precise factor.

Thus the least significant bit is multiplied by 2^0 or 1. (If you are not sure of this try the direct program PRINT 2 0.) The next bit is multiplied by 2^1, and so on.

This rule holds for ANY base; let's apply it for hex, i.e. base 16:

Least significant bit factor = 16^0 = 1

2nd most significant bit factor = 16^1 = 16

3rd most significant bit factor = 16^2 = 256

Most significant bit factor = 16^3 = 4096

APPENDIX TWO

TABLE 1

The 6510 Instruction Set with Mnemonics used in this Book

Abbreviations used in this table:

A = Accumulator
X = X register
Y = Y register

S = Set (to 1)
C = Clear (to 0)
? = Condition according to data.

ADC ADd specified contents to accumulator with Carry: store answer in accumulator; Condition negative, overflow, zero and carry flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ADC	109	6D	2	4	Absolute
ADCIM	105	69	1	2	Immediate
ADCIX	97	61	1	6	Indirect with X
ADCIY	113	71	1	5*	Indirect with Y
ADCX	125	7D	2	4*	Indexed with X
ADCY	121	79	2	4*	Indexed with Y
ADCZ	101	65	1	3	Zero-page
ADCZX	117	75	1	4	Zero-page indexed with X

* Plus 1 cycle if page boundary crossed.
To add without carry, clear carry flag (CLC) before ADC.
ADC operates in decimal or binary mode according to D-flag setting.

N	V	B	D	I	Z	C
?	?	-	-	-	?	?

AND AND specified contents with accumulator: store answer in accumulator: condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
AND	45	2D	2	4	Absolute
ANDIM	41	29	1	2	Immediate
ANDIX	33	21	1	6	Indirect with X
ANDIY	49	31	1	5*	Indirect with Y
ANDX	61	3D	2	4*	Indexed with X
ANDY	57	39	2	4*	Indexed with Y
ANDZ	37	25	1	3	Zero-page
ANDZX	53	35	1	4	Zero-page indexed with X

* Plus 1 cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

A \ D	0	1
0	0	0
1	0	1

ASL Arithmetic Shift Left of specified contents: bit 7 put into carry, '0' into bit zero; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ASL	14	03	1	6	Absolute
ASLA	10	0A	0	2	Accumulator
ASLX	30	1E	2	7	Indexed with X
ASLZ	6	06	2	5	Zero-page
ASLZX	22	16	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

BCC Branch on Carry Clear: Test carry flag, if clear (C=0) branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BCC	144	90	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: No effect.

BCS Branch on Carry Set: test carry flag, if set (C=1), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BCS	176	BO	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BEQ Branch on result EQual to zero: test Z-flag, if set, (Z=1), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BEQ	240	FO	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BIT AND specified BITs with accumulator: A remains unaltered; set Z (=1) if bits match, transfer bits 6 and 7 of specified data to V and N flags respectively; condition zero flag according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BIT BITZ	44 36	2C 24	2 1	4 3	Absolute Zero-page

N	V	B	D	I	Z	C
Bit 7	Bit 6	-	-	-	?	-

BMI Branch on result MInus: test N flag if set (N=1), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BMI	48	30	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BNE Branch on result Not Equal to zero: test Z flag if not set (Z=0), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BNE	208	DO	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BPL Branch on result PLus: test N flag, if not set (N=0), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BPL	16	10	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BRK BReaK into interrupt: initiate interrupt sequence; Save PC+2 on stack; set B flag (B=1); save PSW on stack; load interrupt vectors (FFFE and FFFF) into PC. Set I flag (I=1).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BRK	0	0	0	7	Implied

N	V	B	D	I	Z	C
-	-	S	-	S	-	-

BVC Branch on oVerflow Clear: test overflow flag, if not set (V=0), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BVC	80	50	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

BVS Branch on oVerflow Set: test overflow flag, if set (V=1), branch relative.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
BVS	112	70	1	2*	Relative

* Plus one cycle if branch implemented to same page.
Plus two cycles if branch implemented to different page.
FLAGS: no effect.

CLC CLear Carry flag: load '0' into carry flag (C=0).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLC	24	18	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	-	-	C

CLD CLear Decimal flag: load '0' into decimal flag.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MOE
	DEC	HEX			
CLD	216	D8	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	C	-	-	-

CLI CLear Interrupt disable flag: load '0' into interrupt flag (I=0).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLI	88	58	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	C	-	-

CLV CLear oVerflow flag: load '0' into overflow flag (V=0).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CLV	184	B8	0	2	Implied

N	V	B	D	I	Z	C
-	0	-	-	-	-	-

CMP CoMPare specified data with accumulator: subtract data from accumulator, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CMP	205	CD	2	4	Absolute
CMPIM	201	C9	1	2	Immediate
CMPIX	193	C1	1	6	Indirect with X
CMPIY	209	D1	1	5*	Indirect with Y
CMPX	221	DD	2	4*	Indexed with X
CMPY	217	D9	2	4*	Indexed with Y
CMPZ	197	C5	1	3	Zero-page
CMPZX	213	D5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

CPX ComPare specified data with X register: subtract data from X register, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CPX	236	EC	2	4	Absolute
CPXIM	224	E0	1	2	Immediate
CPXZ	228	E4	1	3	Zero-page

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

CPY ComPare specified data with Y-register: subtract data from Y-register, do not store result; set Z if equal, otherwise reset; condition N by bit 7 and C by result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
CPY	204	CC	2	4	Absolute
CPYIM	192	CO	1	2	Immediate
CPYX	196	C4	1	3	Zero-page

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

DEC DECrement specified memory contents by one, store result in specified memory location; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEC	206	CE	2	6	Absolute
DECX	222	DE	2	7	Indexed with X
DECZ	198	C6	1	5	Zero-page
DECZX	214	D6	1	6	Zero-page, indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

DEX DEcrement contents of X-register: store result in X-register; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEX	202	CA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

DEY DEcrement contents of Y-register: store result in Y-register; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
DEY	135	88	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

EOR Perform Exclusive OR between accumulator and specified contents: store result in accumulator. Condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
EOR	77	4D	2	4	Absolute
EORIM	73	49	1	2	Immediate
EORIX	65	41	1	6	Indirect with X
EORIY	81	51	1	5*	Indirect with Y
EORX	93	5D	2	4*	Indexed with X
EORY	89	59	2	4*	Indexed with Y
EORZ	69	45	1	3	Zero-page
EORZX	85	55	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

D A	0	1
0	0	1
1	1	0

INC INCrement specified contents by one: store result in specified location; condition negative and zero page flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
INC	238	EE	2	6	Absolute
INCX	254	FE	2	7	Indexed with X
INCZ	230	E6	1	5	Zero-page
INCZX	246	F6	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

INX INcrement X-register by one: store result in X-register; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
INX	232	E8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

INY INcrement Y-register by one: store result in Y-register; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
INY	200	C8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

JMP JuMP to specified address (load specified address into program counter).

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
JMP	76	4C	2	3	Absolute
JMPIA	108	6C	2	5	Indirect

FLAGS: no effect.

JSR Jump to SubRoutine at specified address: store program counter contents +2 on stack; load specified contents into program counter.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
JSR	32	20	2	6	Absolute

FLAGS: no effect.

LDA LoaD Accumulator with specified contents: condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LDA	173	AD	2	4	Absolute
LDAIM	169	A9	1	2	Immediate
LDAIX	161	A1	1	6	Indirect with X
LDAIY	177	B1	1	5*	Indirect with Y
LDAX	189	BD	2	4*	Indexed with X
LDAY	185	B9	2	4*	Indexed with Y
LDAZ	165	A5	1	3	Zero-page
LDAZX	181	B5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LDX LoaD X-register with specified contents: condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LDX	174	AE	2	4	Absolute
LDXIM	162	A2	1	2	Immediate
LDXY	190	BE	2	4*	Indexed with Y
LDXZ	166	A6	1	3	Zero-page
LDXZY	182	B6	1	4	Zero-page indexed with Y

* Plus one cycle if page boundary crossed.

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LDY Load Y-register with specified contents: condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LDY	172	AC	2	4	Absolute
LDYIM	160	AO	1	2	Immediate
LDYX	188	BC	2	4*	Indexed with X
LDYZ	164	A4	1	3	Zero-page
LDYZX	180	B4	1	4	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

LSR Perform Logical Shift Right of specified contents: load bit 0 into carry bit and a '0' into bit 7; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
LSR	78	4E	2	3	Absolute
LSRA	74	4A	0	2	Accumulator
LSRX	94	5E	2	3	Indexed with X
LSRZ	70	46	1	6	Zero-page
LSRZX	86	56	1	6	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	?

NOP No OPeration; wait two cycles then continue.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
NOP	234	EA	0	2	Implied

FLAGS: no effect.

ORA Perform logical OR between Accumulator and specified contents: store result in accumulator; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ORA	13	OD	2	4	Absolute
ORAIM	9	09	1	2	Immediate
ORAIX	1	01	1	6	Indirect with X
ORAIY	17	11	1	5*	Indirect with Y
ORAX	29	1D	2	4*	Indexed with X
ORAY	25	19	2	4*	Indexed with Y
ORAZ	5	05	1	3	Zero-page
ORAZX	21	15	1	4	Zero-page indexed with X

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TRUTH TABLE

D A	0	1
0	0	1
1	1	1

PHA Push Accumulator onto stack: update stack pointer; A remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PHA	72	48	0	3	Implied

FLAGS: no effect.

PHP PusH Processor status word onto stack: update stack pointer; PSW remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PHP	8	08	0	3	Implied

FLAGS: no effect.

PLA PuLl Accumulator from stack: update pointer; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PLA	104	68	0	4	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

PLP PuLl Processor status word from stack: update stack pointer; condition ALL flags according to PSW retrieved.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
PLP	40	28	0	4	Implied

N	V	B	D	I	Z	C
?	?	?	?	?	?	?

ROL ROtate Left one place, specified contents: load carry bit into bit 0 of specified data and bit 7 of these into carry flag; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ROL	46	2E	2	6	Absolute
ROLA	42	2A	0	2	Accumulator
ROLX	62	3E	2	7	Indexed with X
ROLZ	38	26	1	5	Zero-page
ROLZX	54	36	1	6	Zero-page indexed with X

ROL is a 9-bit rotation.

N	V	B	D	I	Z	C
?	-	-	-	-	?	bit 7

ROR ROtate Right one place, specified contents: load carry bit into bit 7 of specified data and bit 0 of these into carry flag; condition negative and zero flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
ROR	110	6E	2	6	Absolute
RORA	106	6A	0	2	Accumulator
RORX	126	7E	2	7	Indexed with X
RORZ	102	66	1	5	Zero-page
RORZX	118	76	1	6	Zero-page indexed with X

ROR is a 9-bit rotation.

N	V	B	D	I	Z	C
?	-	-	-	-	?	bit 0

RTI ReTurn from Interrupt: retrieve PSW and PC from stack, update stack pointer.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
RTI	64	40	0	6	Implied

N	V	B	D	I	Z	C
?	?	?	?	?	?	?

RTS ReTurn from Subroutine: retrieve PC from stack and increment by one, update stack pointer.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
RTS	96	60	0	6	Implied

FLAGS: no effect.

SBC SuBtract with Carry specified contents from accumulator: store answer in accumulator; condition negative, overflow, zero and carry flags according to result.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SBC	237	ED	2	4	Absolute
SBCIM	233	E9	1	2	Immediate
SBCIX	225	E1	1	6	Indirect with X
SBCIY	241	F1	1	5*	Indirect with Y
SBCX	253	FD	2	4*	Indexed with X
SBCY	249	F9	2	4*	Indexed with Y
SBCZ	229	E5	1	3	Zero-page
SBCZX	245	F5	1	4	Zero-page indexed with X

* Plus one cycle if page boundary crossed.
To subtract without carry, set carry flag (SEC) before SBC.
SBC operates in decimal or binary mode according to D-flag setting.

N	V	B	D	I	Z	C
?	?	-	-	-	?	?

SEC SEt Carry flag: C=1

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SEC	56	38	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	-	-	S

SED SEt Decimal flag: D=1

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SED	248	F8	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	S	-	-	-

SEI SEt Interrupt disable flag: I=1

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
SEI	120	76	0	2	Implied

N	V	B	D	I	Z	C
-	-	-	-	S	-	-

STA STore Accumulator contents at specified address: A remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STA	141	8D	2	4	Absolute
STAIX	129	81	1	6	Indirect with X
STAIY	145	91	1	6	Indirect with Y
STAX	157	9D	2	5	Indexed with X
STAY	153	99	2	5	Indexed with Y
STAZ	133	85	1	3	Zero-page
STAZX	149	95	1	4	Zero-page indexed with X

FLAGS: no effect.

STX STore contents of X-register at specified address: X remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STX	142	8E	2	4	Absolute
STXZ	134	86	1	3	Zero-page
STXZY	150	96	1	4	Zero-page indexed with Y

FLAGS: no effect.

STY STore contents of Y-register at specified address: Y remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
STY	140	8C	2	4	Absolute
STYZ	132	84	1	3	Zero-page
STYZX	148	94	1	4	Zero-page indexed with X

FLAGS: no effect.

TAX Transfer contents of Accumulator to X-register: A remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TAX	170	AA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TAY Transfer contents of Accumulator to Y-register: A remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TAY	168	A8	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TSX Transfer contents of Stack-pointer to X-register: SP remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TSX	186	BA	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TXA Transfer contents of X-register into Accumulator: X remains unalted; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TXA	138	8A	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TXS   Transfer contents of X-register into Stack: X remains unaltered.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TXS	154	9A	0	2	Implied

FLAGS: no effect.

TYA   Transfer contents of Y-register into Accumulator: Y remains unaltered; condition negative and zero flags according to data.

MNEM.	OP-CODE		NO. BYTES OPER.	NO. CYCLES	ADDRESSING MODE
	DEC	HEX			
TYA	152	98	0	2	Implied

N	V	B	D	I	Z	C
?	-	-	-	-	?	-

TABLE 2

6510 FLAG GUIDE

This guide lists all the 6510 flags, along with the families of instructions that set them and the branch instructions that test their condition. As all members of the family condition the flags in the same way the whole family is referenced by the first three letters of the mnemonic, i.e. ADC refers to the Add with Carry family or ADC, ADCIM, ADCIX, ADCIY, ADCX, ADCY, ADCZ and ADCZX.

N - NEGATIVE FLAG

Instruction to Condition				Instruction to Test
ADC	DEX	LDY	SBC	BMI
AND	DEY	LSR	TAX	BPL
ASL	EOR	ORA	TAY	
BIT	INC	PLA	TXA	
CMP	INX	PLP	TYA	
CPY	INY	ROL		
CPX	LDA	ROR		
DEC	LDX	RTI		

BIT instruction loads N-flag with bit 7 of the specified memory location.

V - OVERFLOW FLAG

Instruction to Condition			Instruction to Test
ADC	CLV	RTI	BVC
BIT	PLP	SBC	BVS

BIT loads V with bit 6 of the specified memory location.

B - BREAK FLAG

Instruction to Condition			Instruction to Test
BRK	PLP	RTI	

D - DECIMAL FLAG

Instruction to Condition				Instruction to Test
CLD	PLP	RTI	SED	

I - INTERRUPT FLAG

Instruction to Condition				Instruction to Test
BRK	CLI	PLP	RTI	
SEI				

Z - ZERO FLAG

Instruction to Condition

ADC
AND
ASL
BIT
CMP
CPY
CPX
DEC
DEX
DEY
EOR
INC
INX
INY
LDA
LDX
LDY
LSR
ORA
PLA
PLP
ROL
ROR
RTI
SBC
TAX
TAY
TXA
TYA

Instruction to Test

BEQ
BNE

C - CARRY FLAG

Instruction to Condition

ADC
ASL
CLC
CMP
CPX
CPY
LSR
PLP
ROL
ROR
RTI
SBI
SEC

Instruction to Test

BCC
BCS

TABLE 3

Hexadecimal to Decimal Conversion Chart

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

TABLE 4

ASCII CHARACTER SET

HEX	MSB	0	1	2	3	4	5	6	7
LSB	BIN	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	1	|
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	0	–	o	DEL

TABLE 5
CBM 64 Screen Codes

poke	set1	set2	poke	set1	set2	poke	set1	set2	poke	set1	set2
0	@		33	!		65	♠	A	97	▌	
1	A	a	34	"		66	│	B	98	▄	
2	B	b	35	#		67	─	C	99	▔	
3	C	c	36	$		68	─	D	100	▁	
4	D	d	37	%		69	─	E	101	▏	
5	E	e	38	&		70	─	F	102	▒	
6	F	f	39	'		71	│	G	103	▕	
7	G	g	40	(		72	│	H	104	▄	
8	H	h	41	)		73	╮	I	105	◤	▨
9	I	i	42	*		74	╰	J	106	▐	
10	J	j	43	+		75	╯	K	107	├	
11	K	k	44	,		76	└	L	108	▗	
12	L	l	45	-		77	╲	M	109	└	
13	M	m	46	.		78	╱	N	110	┐	
14	N	n	47	/		79	┌	O	111	▂	
15	O	o	48	0		80	┐	P	112	┌	
16	P	p	49	1		81	●	Q	113	┴	
17	Q	q	50	2		82	▁	R	114	┬	
18	R	r	51	3		83	♥	S	115	┤	
19	S	s	52	4		84	▏	T	116	▎	
20	T	t	53	5		85	╭	U	117	▍	
21	U	u	54	6		86	╳	V	118	▐	
22	V	v	55	7		87	○	W	119	▔	
23	W	w	56	8		88	♣	X	120	▀	
24	X	x	57	9		89	▕	Y	121	▃	
25	Y	y	58	:		90	♦	Z	122	▁	✓
26	Z	z	59	;		91	┼		123	▖	
27	[		60	<		92	▒		124	▝	
28	£		61	=		93	│		125	┘	
29	]		62	>		94	π	▒	126	▘	
30	↑		63	?		95	◥	▧	127	▚	
31	←		64	─		96	<space>				
32	<space>										

Codes from 128 to 255 are reversed images of codes 0-127.

TABLE 6

PRINTS	CHR$	PRINTS	CHR$	PRINT$	CHR$	PRINTS	CHR$
	0	.	46	#	92	f4	138
	1	/	47	]	93	f6	139
	2	0	48	↑	94	f8	140
	3	1	49	←	95	SHIFT<RET'N>	141
	4	2	50	⊟	96	<UCASE>	142
<WHT>	5	3	51	♠	97		143
	6	4	52	◫	98	<BLK>	144
	7	5	53	⊟	99	<UCRSR>	145
<NOCBM>	8	6	54	▔	100	<RVSOFF>	146
<OKCBM>	9	7	55	▔	101	<CLR>	147
	10	8	56	▁	102	<INST>	148
	11	9	57	▏	103	<BRWN>	149
<RETURi	12	:	58	▕	104	<LTRED>	150
<RET'N>	13	;	59	╮	105	<GRY1>	151
<LCASE>	14	<	60	╰	106	<GRY2>	152
	15	=	61	╯	107	<LTGRN>	153
	16	>	62	□	108	<LTBLU>	154
<DCRSR>	17	?	63	╲	109	<GRY3>	155
<RVSON>	18	@	64	╱	110	<PUR>	156
<HOME>	19	A	65	□	111	<LCRSR>	157
<DEL>	20	B	66	□	112	<YEL>	158
	21	C	67	●	113	<CYN>	159
	22	D	68	▁	114	<SPACE>	160
	23	E	69	♥	115	▌	161
	24	F	70	▏	116	▄	162
	25	G	71	╭	117	▔	163
	26	H	72	╳	118	▁	164
	27	I	73	○	119	▏	165
<RED>	28	J	74	♣	120	▒	166
<RCRSR>	29	K	75	▕	121	▕	167
<GRN>	30	L	76	♦	122	▒	168
<BLU>	31	M	77	┼	123	◤	169
<SPACE>	32	N	78	▒	124	▐	170
!	33	O	79	│	125	├	171
"	34	P	80	π	126	▗	172
#	35	Q	81	◥	127	└	173
$	36	R	82		128	┐	174
%	37	S	83	<ORNG>	129	▂	175
&	38	T	84		130	┌	176
'	39	U	85		131	┴	177
(	40	V	86		132	┬	178
)	41	W	87	f1	133	┤	179
*	42	X	88	f3	134	▎	180
+	43	Y	89	f5	135	▍	181
,	44	Z	90	f7	136	▐	182
-	45	[	91	f2	137	▔	183

PRINTS	CHR$	PRINTS	CHR$	PRINT$	CHR$	PRINTS	CHR$
▔	184	□	186	▝	188	▘	190
▁	185	▖	187	◰	189	▚	191

CODES	192-223	SAME AS	96-127
CODES	224-254	SAME AS	160-190
CODES	255	SAME AS	126

TABLE 7

VIC II (Video Interface Controller) Registers

No	Address		Bits	Description
	Hex	Decimal		
0	D000	53248		Sprite 0 X position
1	D001	53249		Sprite 0 Y position
2	D002	53250		Sprite 1 X position
3	D003	53251		Sprite 1 Y position
4	D004	53252		Sprite 2 X position
5	D005	53253		Sprite 2 Y position
6	D006	53254		Sprite 3 X position
7	D007	53255		Sprite 3 Y position
8	D008	53256		Sprite 4 X position
9	D009	53257		Sprite 4 Y position
10	D00A	53258		Sprite 5 X position
11	D00B	53259		Sprite 5 Y position
12	D00C	53260		Sprite 6 X position
13	D00D	53261		Sprite 6 Y position
14	D00E	53262		Sprite 7 X position
15	D00F	53263		Sprite 7 Y position
16	D010	53264		Sprites 0-7 X position (ms bit of X coordinate)
17	D011	53265		VIC Control Register:
			7	Raster compare (ms bit)
			6	Extended colour text mode, 0 = disabled, 1 = enabled
			5	Bit -map mode, 0 = disabled, 1 = enabled
			4	Blank screen to border colour, 0 = blanked, 1 = normal
			3	Display 24/25 row text, 0 = 24 rows, 1 = 25 rows
			2-0	Smooth scroll to Y dot position
18	D012	53266		Read/write raster value
19	D013	53267		Light pen X position
20	D014	53268		Light pen Y position
21	D015	53269		Sprite display enable/disable
			7-0	0 = disable, 1 = enable
22	D016	53270		VIC Control Register
			7-6	Unused
			5	Reset bit, 0 = normal status, 1 = stopped
			4	Multi-colour mode, 0 = disabled, 1 = enabled
			3	Display 38/40 column text, 0 = 38 columns, 1 = 40 columns
			2-0	Smooth scroll to X dot position

No	Address		Bits	Description
	Hex	Decimal		
23	D017	53271		Sprite vertical expansion
			7-0	0 = normal, 1 = expanded
24	D018	53272		VIC Memory Control Register
			7-4	Video matrix base address
			3-0	Character dot data base address
25	D019	53273		VIC Interrupt Flag Register
			7	VIC interrupt flag, 0 = no interrupt, 1 = interrupt
			6-4	Unused
			3	Light-pen interrupt, 0 = none, 1 ≠ interrupt detected
			2	Sprite to sprite collision, 0 = none, 1 = collision detected
			1	Sprite to background collision, 0 = none, 1 = collision detected
			0	Raster compare, 0 = not matched, 1 = matched
26	D01A	53274		VIC Interrupt Enable Register
			7	1 = set interrupt enable for..
			6-4	Unused
			3	light-pen, 0 = disable, 1 = enable
			2	sprite to sprite collision, 0 = disable, 1 = enable
			1	sprite to background collision, 0 = disable, 1 = enable
			0	raster match with D012, 0 = disable, 1 = enable
27	D01B	53275		Sprite to background priority
			7-0	0 = background, 1 = sprite
28	D01C	53276		Sprite multi-colour mode
			7-0	0 = standard mode, 1 = M.C.M.
29	D01D	53277		Sprite Y expansion
			7-0	0 = normal, 1 = expanded
30	D01E	53278		Sprite to sprite collision detect
			7-0	0 = do not detect, 1 = detect
31	D01F	53279		Sprite to background collision detect
			7-0	0 = do not detect, 1 = detect
32	D020	53280		Border colour
33	D021	53281		Background colour (text)
34	D022	53282		Background colour 1
35	D023	53283		Background colour 2
36	D024	53284		Background colour 3

No	Address		Bits	Description
	Hex	Decimal		
37	D025	53285		Sprite multi-colour register 0
38	D026	53286		Sprite multi-colour register 1
39	D027	53287		Sprite 0 colour
40	D028	53288		Sprite 1 colour
41	D029	53289		Sprite 2 colour
42	D02A	53290		Sprite 3 colour
43	D02B	53291		Sprite 4 colour
44	D02C	53292		Sprite 5 colour
45	D02D	53293		Sprite 6 colour
46	D02E	53294		Sprite 7 colour

TABLE 8

SID (Sound Interface Device) Registers

Voice 1 Registers

No	Address		Bits	Description
	Hex	Decimal		
0	D400	54272		Frequency control - low byte
1	D401	54273		Frequency control - high byte
2	D402	54274		Pulse waveform width - low byte
3	D403	54275		Pulse waveform width - high byte
			7-4	Unused
			3-0	Bits 11-8 of waveform width
4	D404	54276		Voice Control Register
			7	Random noise waveform, 0 = not selected, 1 = selected
			6	Pulse noise waveform, 0 = not selected, 1 = selected
			5	Sawtooth noise waveform, 0 = not selected, 1 = selected
			4	Triangle noise waveform 0 = not selected, 1 = selected
			3	Test bit for oscillator, 0 = normal, 1 = disable
			2	Ring modulate with oscillator 3 0 = don't modulate, 1 = do
			1	Synchronise with oscillator 3 0 = don't synchronise, 1 = do
			0	Gate bit, 0 = start release, 1 = start attack
5	D405	54277		Envelope Attack/Decay control
			7-4	Attack duration
			3-0	Decay duration
6	D406	54278		Envelope Sustain/Release control
			7-4	Sustain level
			3-0	Release duration

Voice 2 Registers

7	D407	54279		Frequency control - low byte
8	D408	54280		Frequency control - high byte
9	D409	54281		Pulse waveform width - low byte
10	D40A	54282		Pulse waveform width - high byte
			7-4	Unused
			3-0	Bits 11-8 of waveform width

No	Address		Bits	Description
	Hex	Decimal		
11	D40B	54283		Voice Control Register
			7	Random noise waveform, 0 = not selected,
				1 = selecte d
			6	Pulse noise waveform, 0 = not selected,
				1 = selected
			5	Sawtooth noise waveform,
				0 = not selected, 1 = selected
			4	Triangle noise waveform
				0 = not selected, 1 = selected
			3	Test bit for oscillator, 0 = normal,
				1 = disable
			2	Ring modulate with oscillator 1
				0 = don't modulate, 1 = do
			1	Synchronise with oscillator 1
				0 = don't synchronise, 1 = do
			0	Gate bit, 0 = start release
				1 = start attack
12	D40C	54284		Envelope Attack/Decay control
			7-4	Attack duration
			3-0	Decay duration
13	D40D	54285		Envelope Sustain/Release control
			7-4	Sustain level
			3-0	Release duration

Voice 3 Registers

14	D40E	54286		Frequency control - low byte
15	D40F	54287		Frequency control - high byte
16	D410	54288		Pulse waveform width - low byte
17	D411	54289		Pulse waveform width - high byte
			7-4	Unused
			3-0	Bits 11-8 of waveform width

No	Address		Bits	Description
	Hex	Decimal		
18	D412	54290		Voice Control Register
			7	Random noise waveform, 0 = not selected, 1 = selected
			6	Pulse noise waveform, 0 = not selected, 1 = selected
			5	Sawtooth noise waveform, 0 = not selected, 1 = selected
			4	Triangle noise waveform, 0 = not selected, 1 = selected
			3	Test bit for oscillator, 0 = normal, 1 = disable
			2	Ring modulate with oscillator 2 0 = don't modulate, 1 = do
			1	Synchronise with oscillator 2 0 = don't synchronise, 1 = do
			0	Gate bit, 0 = start release 1 = start attack
19	D413	54291		Envelope Attack/Decay control
			7-4	Attack duration
			3-0	Decay duration
20	D414	54292		Envelope Sustain/Release control
			7-4	Sustain level
			3-0	Release duration

SID General Registers

21	D415	54293		Filter cutoff frequency low byte
			7-3	Unused
			2-0	Used
22	D416	54294		Filter cutoff frequency high byte
23	D417	54295		Filter resonance control
			7-4	Resonance level
			3	External input, 0 = don't filter, 1 = filter
			2	Voice 3 output, 0 = don't filter, 1 = filter
			1	Voice 2 output, 0 = don't filter, 1 = filter
			0	Voice 1 output, 0 = don't filter, 1 = filter

No	Address		Bits	Description
	Hex	Decimal		
23	D418	54296		Filter mode/Volume
			7	Cut-off voice 3, 0 = normal, 1 = cut-off
			6	Select high-pass mode, 0 = don't select, 1 = select
			5	Select band-pass mode, 0 = don't select, 1 = select
			4	Select low-pass mode, 0 = don't select, 1 = select
			3-0	Output volume
25	D419	54297		Analogue to digital conversion 1 (game paddle 1)
26	D41A	54298		Analogue to digital conversion 2 (game paddle 2)
27	D41B	54299		Oscillator 3 Random Number Generator Output
28	D41C	54300		Envelope Generator 3 Output

Note: Registers 0 to 24 (54272 to 54296, hex D400 to D418) are write-only, 25 to 28 (54297 to 54300, hex D419 to D41C) are read-only.

APPENDIX THREE

CBM 64 Memory Maps

The serious machine code programmer of the CBM 64 is writing programs which, of necessity, must co-exist peacefully with the other software which has been provided in ROM by Commodore.

Firstly, many of the most powerful 6510 instructions require use of zero page locations. The BASIC and Kernel routines make extensive use of the zero page, however, and knowledge of the locations that they use, is necessary to avoid conflicts.

Secondly, considerable benefits may be obtained by making use of the comprehensive routines which are included in the ROM. Considerable expertise was used in the writing of these routines and it is unlikely that we ordinary folk are going to invent a better 'wheel', not just yet anyway.

Lastly, there will be many occasions where the CBM 64 programmer will wish to convert for use on the CBM 64, programs which were originally written for other microcomputers in the Commodore range. Unfortunately, the use made of RAM and the disposition of the various routines in ROM has changed as the different machines have been developed, and some knowledge of these changes is required. Many of the programs written for the PET computer range, particularly the 40 column screen machines are potential targets for conversion. The VIC 20 shares many of the charactersistics of the CBM 64 and, provided the different screen width can be allowed for, there is ample scope for conversion here also.

The intrepid traveller without a map is likely to get lost. The memory maps which follow, should go some way to avoiding the dreaded journey up that well known creek. In addition to providing a comprehensive guide to the CBM 64 ROM, however, they also provide essential cross references to other CBM micros: the VIC 20, and the PET range using either BASIC 2 or BASIC 4. In the tables, these references are identified as VIC, B2 or B4 respectively. An entry of -- in the table means that no meaningful counterpart exists.

Wherever possible, I have used the labels which are used by Commodore for the CBM 64. In cases where it has not been possible to discover the 'genuine' labels, I have adopted the labels created by Mike Todd of the Independent Commodore Products Users Group (ICPUG). Where all else has failed, however, I have invented my own labels.

Certain conventions have been used in the descriptions. A, X, Y refer to the 6510 accumulator, X-register and the Y-register. References which are enclosed in parentheses should be read as 'the contents of' that reference. Thus, (TXTPTR) is the two byte address in the usual low/high byte form, (A/Y) is the value or address formed from the contents of the accumulator (low byte), and the Y-register (high byte).

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
D6510	0000	0	--	--	6510 On-chip Data Direction Register.
R6510	0001	1	--	--	6510 On-chip 8-bit Input/Output register.
	0002	2			Unused.
ADRAY1	0003-0004	3	0003	--	Jump vector: Convert FAC to integer in (A/Y).
ADRAY2	0005-0006	5	0005	--	Jump vector: Convert integer in (A/Y) to floating point in (FAC).
CHARAC	0007	7	0007	0003	Search character / temporary integer during INT.
ENDCHR	0008	8	0008	0004	Flag: Scan for Quote at end of string.
INTEGR	0007-0008	7	0007	0003	Temporary integer during OR/AND.
TRMPOS	0009	9	0009	--	Screen column from last TAB.
VERCK	000A	10	000A	009D	Flag: 0 = load, 1 = verify.
COUNT	000B	11	000B	0005	Input buffer pointer/Number of subscripts.
DIMFLG	000C	12	000C	0006	Flag: Default Array dimension.
VALTYP	000D	13	000D	0007	Data type flag: $00 = numeric, $FF = string.
INTFLG	000E	14	000E	0008	Data type flag: $00= floating point, $80 = integer.
GARBFL	000F	15	000F	0009	Flag: DATA scan / List quote / Garbage collection.
SUBFLG	0010	16	0010	000A	Flag: Subscript reference / User function call.
INPFLG	0011	17	0011	000B	Input flag: $00 = INPUT, $40 = GET, $98 = READ.
TANSGN	0012	18	0012	000C	Flag: TAN sign / Comparitive result.
CHANNL	0013	19	0013	000E	File number of current input device.
LINNUM	0014-0015	20	0014	0011	Temporary: Integer value.
TEMPPT	0016	22	0016	0013	Pointer: Temporary string stack.
LASTPT	0017-0018	23	0017	0014	Last temporary string address.
TEMPST	0019-0021	25	0019	0016	Stack for temporary strings.
INDEX	0022-0025	34	0022	001F	Utility pointer area.
INDEX1	0022-0023	34	0022	001F	First utility pointer.
INDEX2	0024-0025	36	0024	0021	Second utility pointer.
RESHO	0026-002A	38	0026	0023	Floating point product of multiply and divide.
TXTTAB	002B-002C	43	002B	0028	Pointer: Start of BASIC text area.
VARTAB	002D-002E	45	002D	002A	Pointer: Start of BASIC variables.
ARYTAB	002F0030	47	002D	002C	Pointer: Start of BASIC arrays.
STREND	0031-0032	49	0031	002E	Pointer: End of BASIC arrays +1.
FRETOP	0033-0034	51	0033	0030	Pointer: Bottom of string space.
FRESPC	0035-0036	53	0035	0032	Utility string pointer.
MEMSIZ	0037-0038	55	0037	0034	Pointer: Highest address available to BASIC.
CURLIN	0039-003A	57	0039	0036	Current BASIC line number.
OLDLIN	003B-003C	59	003B	0038	Previous BASIC line number.
OLDTXT	003D-003E	61	003D	003A	Pointer: BASIC statement for CONT.
DATLIN	003F-0040	63	003F	003C	Current DATA line number.
DATPTR	0041-0042	65	0041	003E	Pointer: Used by READ - current DATA item address.
INPPTR	0043-0044	67	0043	0040	Pointer: temporary storage of pointer during INPUT and READ routines.

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
VARNAM	0045-0046	69	0045	0042	Name of variable being sought in variable table.
VARPNT	0047-0048	71	0047	0044	Pointer: to value of (VARNAM) if integer, to descriptor if string.
FORPNT	0049-004A	73	0049	0046	Pointer: index variable for FOR/NEXT loop.
VARTXT	004B-004C	75	004B	0048	Temporary storage for TXTPTR during READ, INPUT and GET.
OPMASK	004D	77	004D	004A	Mask used durin FRMEVL.
TEMPF3	004E-0052	78	004E	004B	Temporary storage for FLPT value.
FOUR6	0053	83	0053	0050	Length of string variable during garbage collection.
JMPER	0054-0056	84	0054	0051	Jump vector used in function evaluation - JMP ($4C) followed by address.
TEMPF1	0057-005B	87	0057	0054	Temporary storage for FLPT value.
TEMPF2	005C-0060	92	005C	0059	Temporary storage for FLPT value.
FAC	0061-0066	97	0061	005E	Main floating point accumulator.
FACEXP	0061	97	0061	005E	FAC exponent.
FACHO	0062-0065	98	0062	005F	FAC mantissa.
FACSGN	0066	102	0066	0063	FAC sign.
SGNFLG	0067	103	0067	0064	Pointer: series evaluation constant.
BITS	0068	104	0068	0065	Bit overflow area during normalisation routine.
AFAC	0069-006E	105	0069	0066	Auxiliary floating point accumulator.
ARGEXP	0069	105	0069	006	AFAC exponent.
ARGHO	006A-006D	106	006A	0067	AFAC mantissa.
ARGSGN	006E	110	006E	006A	AFAC sign.
ARISGN	006F	111	006F	006C	Sign of result of arithmetic evaluation.
FACOV	0070	112	0070	006D	FAC low-order rounding.
FBUFPT	0071-0072	113	0071	006E	Pointer: used during CRUNCH / ASCII conversion.
CHRGET	0073-008A	115	0073	0070	Subroutine: Get next byte of BASIC text.
CHRGOT	0079	121	0079	0076	Entry to get same byte again.
TXTPTR	007A-007B	122	007A	0079	Pointer: current byte of BASIC text.
RNDX	008B-008F	139	008B	0088	Floating RND function seed value.
STATUS	0090	144	0090	0096	Kernal I/O status word ST.
STKEY	0091	145	0091	009B	Flag: $7F = STOP key (64).
SVXT	0092	146	0092	009C	Timing constant for tape.
VERCKK	0093	147	0093	009D	Flag: 0 = load, 1 = verify.
C3PO	0094	148	0094	00A0	Flag: Serial bus - output character buffered.
BSOUR	0095	149	0095	00A5	Buffered character for serial bus.
SYNO	0096	150	0096	00AB	Cassette sync. number.
TEMPX	0097	151	0097	00AD	Temporary storage of X register during CHRIN.
TEMPY	0097	151	0097	00AD	Temporary storage of Y register during RS232 fetch.
LDTND	0098	152	0098	00AE	Number of open files / index to file table.
DFLTN	0099	153	0099	00AF	Default input device (0).
DFLTO	009A	154	009A	00B0	Default output device (3).

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
PRTY	009B	155	009B	00B1	Parity of byte output to tape.
DPSW	009C	156	009C	00B2	Flag: byte received from tape.
MSGFLG	009D	157	009D	--	Flag: $00 = program mode: suppress error messages, $40 = kernal error messages only, $80 = direct mode: full error messages.
FNMIDX	009E	158	009E	00B4	Index to cassette file name / Header ID for tape write.
PTR1	009E	158	009E	00C0	Tape error log pass 1.
PTR2	009F	159	009F	00C1	Tape error log pass 2.
TIME	00A0-00A2	160	00A0	008D	Real-time jiffy clock.
TSFCNT	00A3	163	0A3	00B7	Bit counter tape read or write / Serial bus EOI flag.
TBTCNT	00A4	164	00A4	00B9	Pulse counter tape read or write / Serial bus shift counter.
CNTDN	00A5	165	00A5	00BA	Tape synchronising count down.
BUFPNT	00A6	166	00A6	00BB	Pointer: Tape I/O buffer.
				00BC	" " " second buffer PET.
INBIT	00A7	167	00A7	--	RS232 temporary for received bit (64/VIC) / Tape temporary (all).
BITC1	00A8	168	00A8	00BE	RS232 input bit count (64/VIC) / Tape temporary(all).
RINONE	00A9	169	00A9	00BF	RS232 flag: Start bit check (64/VIC) / Tape temporary (all).
RIDATA	00AA	170	00AA	00C2	RS232 input byte buffer (64/VIC) / Tape temporary (all).
RIPRTY	00AB	171	00AB	00C3	RS232 Input parity (64/VIC) / Tape temporary (all).
SAL	00AC-00AD	172	00AC	00C7	Pointer: Tape buffer / Screen scrolling.
EAL	00AE-00AF	174	00AE	00C9	Tape end addresses / End of program.
CMP0	00B0-00B1	176	00B0	00CB	Tape timing constants.
TAPE1	00B2-00B3	178	00B2	00D6	Pointer: Start address of tape buffer.
BITTS	00B4	180	00B4	00CE	RS232 write bit count (64/VIC) / Tape read timing flag (all)..
NXTBIT	00B5	181	00B5	00CF	RS232 next bit to send (64/VIC) / tape read - end of tape (all).
RODATA	00B6	182	00B6	00D0	RS232 output byte buffer (64/VIC) / tape read error flag (all).
FNLEN	00B7	183	00B7	00D1	Number of characters in filename.
LA	00B8	184	00B8	00D2	Current file - logical address (number).
SA	00B9	185	00B9	00D3	Current file - secondary address.
FA	00BA	186	00BA	00D4	Current file - first address.
FNADR	00BB-00BC	187	00BB	00D5	Pointer: Current file name address.
ROPRTY	00BD	189	00BD	00DD	RS232 output parity (64/VIC) / Tape byte to be input or output (all).
FSBLK	00BE	190	00BE	00DE	Tape input/output block count.
MYCH	00BF	191	00BF	00DF	Serial word buffer.
CAS1	00C0	192	00C0	00F9	Tape motorswitch.
STAL	00C1-00C2	193	00C1	00FB	Start address for LOAD and cassette write.
MEMUSS	00C3-00C4	195	00C3	--	Pointer: Type 3 tape LOAD and general use.

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
LSTX	00C5	197	00C5	0097	Matrix value of last key pressed; no key = $40 (64/VIC), = $FF (PET).
NDX	00C6	198	00C6	009E	Number of characters in keyboard buffer queue.
RVS	00C7	199	00C7	009F	Flag: Reverse - on = $01, off = $00.
INDX	00C8	200	00C8	00A1	Pointer: End of line for input (used to suppress trailing spaces).
LXSP	00C9-00CA	201	00C9	00A3	Cursor X-Y (line-column) position at start of input.
SFDX	00CB	203	--	--	Flag: Print shifted characters.
KEYVAL	--	--	00CB	00A6	Matrix value of key pressed during last keyboard scan.
BLNSW	00CC	204	00CC	00A7	Flag: Cursor blink - $00 = enabled, $01 = disabled.
BLNCT	00CD	20	00CD	00A8	Timer: Count down for cursor blink toggle.
GDBLN	00CE	206	00CE	00A9	Character under cursor while cursor inverted.
BLNON	00CF	207	00CF	00AA	Flag: Cursor status - $00 = off, $01 = on.
CRSW	00D0	208	00D0	00AC	Flag: Input from screen = $03, or keyboard = $00.
PNT	00D1-00D2	209	00D1	00C4	Pointer: Current screen line address.
PNTR	00D3	211	00D3	00C6	Cursor column on current line, including wrap-round line(s), if any.
QTSW	00D4	212	00D4	00CD	Flag: Editor in quote mode - $00 = not.
LNMX	00D5	213	00D5	00D5	Current logical line length: 39 or 79 (64/PET); 21,43,65 or 87 (VIC).
TBLX	00D6	214	00D6	00D8	Current screen line number of cursor.
SCHAR	00D7	215	00D7	00D9	Screen value of current input character/last character output.
INSRT	00D8	216	00D8	00DC	Count of number of inserts outstanding.
LDTB1	00D9-00F2	217	00D9	00E0	Screen line link table / Editor temporaries.
USER	00F3-00F4	243	00F3	--	Pointer: Current colour RAM location.
KEYTAB	00F5-00F6	245	00F5	--	Vector: Current keyboard decoding table.
RIBUF	00F7-00F8	247	00F7	--	RS232 input buffer pointer.
ROBUF	00F9-00FA	249	00F9	--	RS232 output buffer pointer.
FREKZP	00FB-00FE	251	00FB	--	Free zero page space for user programs.
BASZPT	00FF	255	00FF	--	BASIC temporary data area.
ASCWRK	00FF-10A	255	00FF	00FF	Assembly area for floating point to ASCII conversion.
BAD	0100-013E	256	0100	0100	Tape input error log.
STACK	0100-01FF	256	0100	0100	6510 hardware stack area.
BSTACK	013F-01FF	319	013F	013F	BASIC stack area.
BUF	0200-0258	512	0200	--	BASIC input buffer - 64/VIC.
			0200		" " " 0200-0250 - PET.
LAT	0259-0262	601	0259	0251	Kernal Table: active logical file numbers.
FAT	0263-026C	611	0263	025B	Kernal table: active file first addresses (device numbers).
SAT	026D-0276	621	026D	0265	Kernal table: active file secondary addresses.

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
KEYD	0277-0280	631	0277	026F	Keyboard buffer queue (FIFO).
MEMSTR	0281-0282	641	0281	--	Pointer: bottom of memory for operating system.
MEMSIZ	0283-0284	643	0283	--	Pointer: top of memory for operating system.
TIMOUT	0285	645	0285	--	Serial IEEE bus timeout defeat flag (not used by VIC).
COLOR	0286	646	0286	--	Current character colour code.
GDCOL	0287	647	0287	--	Background colour under cursor.
HIBASE	0288	648	0288	--	High byte of screen memory address.
XMAX	0289	649	0289	--	Maximum number of bytes in keyboard buffer.
RPTFLG	028A	650	028A	--	Flag: Repeat keys - $00 = cursors, INST/DEL & space repeat, $40 = no keys repeat, $80 = all keys repeat.
KOUNT	028B	651	028B	--	Repeat key - speed counter.
DELAY	028C	652	028C	--	Repeat key - first repeat delay counter.
SHFLAG	028D	653	028D	0098	Flag: Shift key - $00 = none, $01 = SHIFT, $02 = CBM, $04 = CTRL (note shifts are additive i.e. $07 = SHIFT, CBM & CTRL).
LSTSHF	028E	654	028E	---	Last shift key used for debouncing.
KEYLOG	028F-0290	655	028F	--	Vector: Routine to determine keyboard table to use based on shift key pattern, $EB48 for 64, $EBDC for VIC.
MODE	0291	657	0291	--	Flag: Case change - $00 = disabled, $80 = enabled.
AUTODN	0292	658	0292	--	Flag: Auto scroll down - $00 = disabled.
M51CTR	0293	659	0293	--	RS232 - Pseudo 6551 control register image.
M51CDR	0294	660	0294	--	RS232 - Pseudo 6551 command register image.
M51AJB	0295-0296	661	0295	--	RS232 non-standard bits/second, not used on VIC.
RSSTAT	0297	663	0297	--	RS232 - Pseudo 6551 status register image.
BITNUM	0298	664	0298	--	RS232 number of bits left to send.
BAUDOF	0299-029A	665	0299	--	RS232 baud rate - full bit time microseconds.
RIDBE	029B	667	029B	--	RS232 index to end of input buffer.
RIDBS	029C	668	029C	--	RS232 pointer: High byte of address of input buffer.
RODBS	029D	669	029D	--	RS232 pointer: High byte of address of output buffer.
RODBE	029E	670	029E	--	RS232 index to end of output buffer.
IRQTMP	029F-02A0	671	029F	--	Temporary store for IRQ vector during tape opeations.
ENABL	02A1	673	02A1	--	RS232 enables.
TODSNS	02A2	674	--	--	TOD sense during tape I/O.
TRDTMP	02A3	675	--	--	Temporary storage during tape read.
TD1IRQ	02A4	676	--	--	Temporary D1IRQ indicator during tape read.
TLNIDX	02A5	677	--	--	Temporary for line index.
TVSFLG	02A6	678	--	--	Flag: TV standard - $00 = NTSC, $01 = PAL.

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
	02A7-02FF	679	02A1	--	Unused.
IERROR	0300-0301	768	0300	--	Vector: Indirect entry to BASIC error message, (X) points to message. Normal contents $E38B (64).
IMAIN	0302-0303	770	0302	--	Vector: Indirect entry to BASIC input line and decode. Normal contents $A483 (64).
ICRNCH	0304-0305	772	0304	--	Vector: Indirect entry to BASIC tokenise routine. Normal contents $A57C (64).
IQPLOP	0306-0307	774	0306	--	Vctor: Indirect entry to BASIC LIST routine. Normal contents $A71A (64).
IGONE	0308-0309	776	0308	--	Vector: Indirect entry to BASIC character dispatch routine. Normal contents $A7E4 (64).
IEVAL	030A-030B	778	030A	--	Vector: Indirect entry to BASIC token evaluation. Normal contents $AE86 (64).
SAREG	030C	780	030C	--	Storage for 6510 accumulator during SYS.
SXREG	030D	781	030D	--	Storage for 6510 X-register during SYS.
SYREG	030E	782	030E	--	Storage for 6510 Y-register during SYS.
SPREG	030F	783	030F	--	Storage for 6510 S-register during SYS.
USRPOK	0310	784	0000	0000	USR function JMP instruction ($4C).
USRADD	0311-0312	785	0001	0001	USR address low byte/high byte.
	0313	787			Unused.
CINV	0314-0315	788	0314	0090	Vector: Hardware IRQ interrupt address. Normal contents $EA31 (64).
CNBINV	0316-0317	790	0316	0092	Vector: BRK instruction interrupt address. Normal contents $FE66 (64).
NMINV	0318-0319	792	0318	0094	Vector: Hardware NMI interrupt address. Normal contents $FE47 (64).
IOPEN	031A-031B	794	031A	--	Vector: Indirect entry to kernal OPEN routine. Normal contents $F34A (64).
ICLOSE	031C-031D	796	031C	--	Vector: Indirect entry to kernal CLOSE routine. Normal contents $F291 (64).
ICHKIN	031E-031F	798	031E	--	Vector: Indirect entry to kernal CHKIN routine. Normal contents $F20E (64).
ICKOUT	0320-0321	800	0320	--	Vector: Indirect entry to kernal CHKOUT routine. Normal contents $F250 (64).
ICLRCH	0322-0323	802	0322	--	Vector: Indirect entry to kernal CLRCHN routine. Normal contents $F333 (64).
IBASIN	0324-0325	804	0324	--	Vector: Indirect entry to kernal CHRIN routine. Normal contents $F157 (64).
IBSOUT	0326-0327	806	0326	--	Vector: Indirect entry to kernal CHROUT routine. Normal contents $F1CA (64).
ISTOP	0328-0329	808	0328	--	Vector: Indirect entry to kernal STOP routine. Normal contents $F6ED (64).
IGETIN	032A-032B	810	032A	--	Vector: Indirect entry to kernal GETIN routine. Normal contents $F13E (64).

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
ICLALL	032C-032D	812	032C	--	Vector: Indirect entry to kernal CLALL routine. Normal contents $F32F (64).
USRCMD	032E-032F	814	032E	03FA	User defined vector. Normal contents $FE66 (64) i.e as CBINV.
ILOAD	0330-0331	816	0330	--	Vector: Indirect entry to kernal LOAD routine. Normal contents $F4A5 (64).
ISAVE	0332-0333	818	0332	--	Vector: Indirect entry to kernal SAVE routine. Normal contents $F5ED (64).
	0334-033B	820	0334	--	Unused.
TBUFFR	033C-03FB	828	033C	027A	Tape I/O buffer.
	03FC-03FF	1020	03FC	--	Unused.
VICSCN	0400-07F7	1024	1E00	8000	Default screen video matrix.
SPNTRS	07F8-07FF	2040	--	--	Default Sprite data pointers.
	0800-9FFF	2048	0400	0400	Normal BASIC program space.
	8000-9FFF	32768	A000	--	Optional cartridge ROM space.
	A000-BFFF	40960	C000	B000	BASIC ROM (part) or 8k RAM.
	C000-CFFF	49152	--	--	4k RAM.
	D000-DFFF	53248	9000	E800	Input/output devices and colour RAM or 4k RA.
	D000-D02E	53248	9000	E880	6566 Video Interface Chip - VIC-II (64), 6561 VIC (VIC), 6845 CRT (80-col PETs).
	D400-D41C	54272	9000	--	6581 Sound Interface Device - SID (64), 6561 VIC (VIC).
	D500-D7FF	54528	--	--	SID images.
	D800-DBFF	55296	9400	--	Colour RAM (nybbles).
	DC00-DC0F	56320	9110	E810	6526 Complex Interface Adaptor - CIA (64), 6522 VIA (VIC), 6521 PIA (PET).
	DD00-DD0F	56576	9120	E820	6526 Complex Interface Adaptor - CIA (64), 6522 VIA (VIC), 6521 PIA (PET).
				E840	6522 VIA (PET).
	E000-FFFF	57344	E000	B000	BASIC (part)/Kernal ROM or 8k RAM.
	E000-E4FF	57344	E000	--	BASIC ROM or RAM.
	E500-FFFF	58624	E500	E000	Kernal ROM or RAM.

BASIC ROM

Label	64	VIC	B2	B4	Description
BCOLD	A000	C000	--	--	BASIC cold start vector.
BWARM	A002	C002	--	--	BASIC warm start vector.
	A004	C004	--	--	Authorship note: "CBMBASIC"
STMDSP	A00C	C00C	C000	B000	BASIC Command vector table.
FUNDSP	A052	C052	C046	B066	BASIC Function vector table.
OPTAB	A080	C080	C074	B094	BASIC Operators vector & priority table - 2 byte address & one priority.
RESLST	A09E	C09E	C092	B0B2	Command keyword table.
MSCLST	A129	C129	C11D	B13D	Miscellaneous keyword table.
OPLIST	A140	C140	C134	B161	Operator keyword table.
FUNLST	A14D	C14D	C141	B16E	Function keyword table.
ERRTAB	A19E	C19E	C192	B20D	Error messages.
ERRPTR	A328	C328			Error message pointers.
OKK	A364	C364			Non-error messages - "OK", "ERROR", "IN", "READY", "BREAK".
FNDFOR	A38A	C38A	C2AA	B322	Find FOR entry on stack or skip them to find GOSUB entry when called by RETURN.
BLTU	A3B8	C3B8	C2D8	B350	Move block of memory up - check suffiient memory then ...
BLTUC	A3BF	C3BF	C2DF	B357	Move block (LOWTR) to (HIGHTR)-1 up to new block ending at (HIGHDS)-1.
GETSTK	A3FB	C3FB	C31B	B393	Check stack for space to accommodate (A)x2 entries - error "OUT OF MEMORY" if not.
REASON	A408	C408	C328	B3A0	Check address (A/Y) is lower than bottom of string space - if not ...
OMERR	A435	C435	C355	B3CD	Print "OUT OF MEMORY" error message.
ERROR	A437*	C437*	C357	B3CF	[*0300 to +3] Print error message indicated by (X) then ...
ERRFIN	A469	C469	C37A	B3F0	Print "ERROR" or "BREAK" (if entered from STPEND).
READY	A474	C474	C389	B3FF	BASIC restart - print "READY" then ...
MAIN	A480*	C480*	C392	B406	[*0302 to +3] Input line - identify BASIC line or command.
MAINI	A49C	C49C	C3AB	B41F	If BASIC line then get line number and convert keywords in line to tokens.
INSLIN	A4A2	C4A2	C3B1	B470	Insert text from BASIC buffer into program - line number in (LINNUM) on entry - line must have keywords changed to tokens and length of line in (Y) - if (BBUFF) = 00 then line will be deleted - routine exits to MAIN.
FINI	A52A	C52A	C439	B4AD	After inserting new line into BASIC text - do RUNC, LNKPRG and reenter at MAIN.
LNKPRG	A533	C533	C442	B4B6	Rebuild BASIC text link pointers.
INLIN	A560	C560	C46F	B4E2	Input line into BASIC buffer - place 00 at end.
RDCHR	--	--	C481	--	Input character to Acc - if charcter is 0F (line feed, invert flag in 0D to suppress output.

Label	64	VIC	B2	B4	Description
CRUNCH	A579*	C579*	C495	B4FB	[*0304 to +3] Change keywords to tokens - line in BBUFF - set (TXTPTR) to BBUFF - (Y) to line length and (TXTPTR) to BBUFF-1 on exit.
FNDLIN	A613	C613	C52C	B5A3	Search BASIC text from start for line number in (LINNUM) ... or...
FNDLNC	A17	C617	C530	B5A7	Search BASIC text from (A/X) for line number in (LINNUM) - if found: set C and (LINPTR) points to start of line - else clear C.
SCRATH	A642	C642	C55B	B5D2	NEW enters here - check syntax then ...
SCRTCH	A644	C644	C55D	B5D4	Reset first byte of text to 00 - set (VARTAB) to (TXTTAB)+2 then ...
RUNC	A659	C659	C572	B5E9	Reset execution to start of program (STXPTR) and then CLEARC.
CLEAR	A65E	C65E	C577	B5EE	CLR enters here - check syntax then ...
CLEARC	A660	C660	C579	B5FC	Set (FRETOP) to (MEMSIZ) - abort I/O - set (ARYTAB) to (VARTAB) then ...
LDCLR	A677	C677	C590	B60B	Do RESTOR - reset (TEMPPT) - reset stack.
STXPT	A68E	C68E	C5A7	B622	Set (TXTPTR) to (TXTTAB)-1 to reset execution to start of program.
LIST	A69C	C69C	C5B5	B630	Entry point for LIST command.
OPLOP	A717*	C717*	C63A	B6B5	[*0306 to +3] Handle LIST character - if non-token or token in quotes print it, else expand token and prit.
FOR	A742	C472	C658	B6DE	Entry point for FOR - saves (TXTPTR), (CURLIN) and final value on stack then ...
NEWSTT	A7AE	C7AE	C6C4	B74A	Check for STOP key then handle next BASIC statement from text.
CKEOL	A7C4	C7C4	C6DA	B7F5	Check end of line is also end of text - else get next line parameters.
GONE	A7E1*	C7E1*	C6F7	B77C	[*0308 to +3] Execute statement within line.
GONE3	A7ED	C7ED	C700	B785	Interpret BASIC command and execute it.
RESTOR	A81D	C81D	C730	B7B7	Entry point for RESTORE command - reset (DATPTR) to start of BASIC.
STOP	A82C	C82C	C73F	B7C6	Entry point for STOP command - clear carry (for 'BREAK' message) then jump into END routine.
END	A82F	C82F	C741	B7C8	Entry point for END - set carry then ....
FINID	A834	C834	C744	B7CB	If not direct - save (TXTPTR) in (OLDTXT) then...
STPEND	A841	C841	C751	B7D8	Save (CURLIN) in (OLDLIN) and exit to READY (if carry set = END) or ERRFIN (if carry clear = STOP).
CONT	A857	C857	C76B	B7EE	Entry point for CONT - restore (TXTPTR) and (CURLIN) unless (OLDTXT) is zero (CAN'T CONTINUE).
RUN	A871	C871	C785	B808	Entry point for RUN - do CLR then GOTO.
GOSUB	A883	C883	C790	B813	Entry point for GOSUB - save (TXTPTR), (CURLIN) and GOSUB flag (8D) on stack then GOTO.
GOTO	A8A0	C8A0	C7AD	B830	Entry point for GOTO - read number from BASIC text into (LINNUM) then ...
GOTOC	A8A3	C8A3	C7B0	B833	Scan for end of current line - search for (LINNUM) line and set (TXTPTR) when found.

Label	64	VIC	B2	B4	Description
RETURN	A8D2	C8D2	C7DA	B85D	Entry point for RETURN - check syntax then ...
RTC	A8D4	C8D4	C7DC	B85F	Clear stack up to first GOSUB entry - then set (TXTPTR) and (CURLIN) from stack.
RTNOGS	A8E0	C8E0	C7E8	B86B	Display "RETURN WITHOUT GOSUB" then exit to MAIN.
NOSTMT	A8E3	C8E3	C7EA	B86E	Display "UNDEFINED STATEMENT" then exit to MAIN.
DATA	A8F8	C8F8	C800	B883	Entry point for DATA - scan text for end of statement update (TXTPTR) to ignore.
DATAN	A906	C906	C80E	B891	Set scan for colon statement delimiter then do SERCHX ...
REMN	A909	C909	C80E	B891	Set scan for zero byte (end of statement) then do SERCHX.
SERCHX	A90B	C90B	C811	B894	Search text for (X) or zero byte - exit with (Y) set to number of bytes to delimiter.
IF	A92B	C92B	C830	B8B3	Entry point for IF - evaluate expression, perform REM if zero (FALSE).
REM	A93B	C93B	C843	B8C6	Entry for REM - scan for zero byte and update (TXTPTR).
DOCOND	A940	C940	C848	B8CB	If IF condition non zero (TRUE) then do command or GOTO as appropriate.
ONGOTO	A94B	C94B	C853	B8D6	Entry point for ON - get number from text and scan for line number - do GOTO or GOSUB.
LINGET	A96B	C96B	C873	B8F6	Read integer from text into (LINNUM) - error if value not in range 0 - 63999.
LET	A9A5	C9A5	C8AD	B930	Entry point for LET - find target variable in variable space and set (FORPNT) to point at it - evaluate expression then PUTINT, PTFLPT, PUTTIM or GETSPT as appropriate.
PTFLPT	A9C4	C9C4	C8CC	B94F	Put (FAC) into variable pointed to by (FORPNT).
PUTINT	A9D6	C9D6	C8DE	B961	Put integer in (FAC+3) into variable pointed to by (FORPNT).
PUTTIM	A9E3	C9E3	C8EB	B972	et TI$ from string - set (INDEX1) to point to string and (A) to six (string length).
GETSPT	AA2C	CA2C	C937	B9BA	Put string descriptor pointed to by (FAC+3) into string variable pointed to by (FORPNT).
PRINTN	AA80	CA80	C98B	BA88	Entry point for PRINT£ - do CMD then restore default I/O (unlisten IEEE if device number ≥ 3).
CMD	AA86	CA86	C991	BA8E	Entry point for CMD - set CMD output device from table then call PRINT.
STRDON	AA9A	CA9A	C9A5	BAA2	Print routine - print string and check for end of print statement.
PRINT	AAA0	CAA0	C9AB	BAA8	Entry point for PRINT - identify PRINT parameters (SPC, TAB etc) - evaluate expression.
VAROP	AAB8	CAB8	C9C3	BAC0	Output variable - (if number convert to string) output string.
NUMDON	AABC	CABC	C9C7	BAC4	Print routine - print numeric.
CRDO	AAD7	CAD7	C9E2	BADF	Output CR/LF - if VIC/64 and (CHANNL) ≥ 127 then output CR only.
COMPRT	AAE8	CAE8	C9EF	BAF0	Print tabs or spaces for comma delimitor, SPC or TAB functions.

Label	64	VIC	B2	B4	Description
STROUT	AB1E	CB1E	CA1C	BB1D	Print string pointed to by (A/Y) until zero byte or quote (") found. (string < 256 bytes).
STRPRT	AB21	CB21	CA1F	BB20	Print string pointed to by (FAC+3) until zero byte or quote (") found (string < 256 bytes).
OUTSTR	AB24	CB24	CA22	BB23	Print string pointed to by (INDEX1) of length (A).
OUTSPC	AB3B	CB3B	CA39	BB3A	Print space (cursor right if to screen).
PRTSPC	AB3F	CB3F	CA3D	BB3E	Print space always.
OUTSKP	AB42	CB42	CA40	BB41	Print cursor right always.
OUTQST	AB45	CB45	CA43	BB44	Print question mark.
OUTDO	AB47	CB47	CA45	BB46	Print (A).
TRMNOK	AB4D	CB4D	CA4F	BB4C	Handle error messages for GET, INPUT and READ.
GET	AB7B	CB7B	CA7D	BB7A	Entry point for GET - check not direct (illegal), identify GET£, get one character.
INPUTN	ABA5	CBA5	CAA7	BBA4	Entry point for INPUT£ - set input device, do input then unlisten IEEE if device > 3.
INPUT	ABBF	CBBF	CAC1	BBBE	Entry point for INPUT - output prompt message if any, do input.
BUFFUL	ABEA	CBAE	CAED	BBE8	Read input - if (BBUFF) zero (no input string) VIC/64 skip, BASIC 2/4 abort.
QINLIN	ABF9	CBF9	CAFA	BBF5	Print ? and input data into BBUFF buffer.
READ	AC06	CC06	CB07	BC02	Entry point for READ - set READ flag (98) in INPFLG, set (X/Y) = DATPTR.
INPCON	AC0D	CC0D	CB0E	BC09	Entry point to READ for INPUT - set INPUT flag (00) in INPFLG, set (X/Y) = BUF.
INPCO1	AC0F	CC0F	CB10	BC0B	Entry point to READ for GET - set GET flag (40) in INPFLG, set (X/Y) = BUF.
RDGET	AC35	CC35	CB36	BC31	Part of READ routine which GETs a byte.
RDINP	AC43	CC43	CB44	BC3F	Part of READ routine which INPUTs, uses RDGET.
DATLOP	ACB8	CCB8	CBB9	BCB4	Part of READ routine which READs DATA values, uses RDGET.
	ACFC	CCFC	CBFC	BCF7	ASCII string - "?EXTRA IGNORED<CR>".
	AD0C	CD0C	CC0D	BD07	ASCII string - "?REDO FROM START<CR>".
NEXT	AD1E	CD1E	CC20	BD19	Entry point for NEXT - get NEXT's variable and confirm that corresponding FOR is on stack, calculate next loop vriable value.
DONEXT	AD61	CD61	CC62	BD5B	If loop counter valid, sets (CURLIN) and (TXTPTR) from stack and reenters FOR loop.
FRMNUM	AD8A	CD8A	CC8B	BD84	Evaluate numeric expression from BASIC text, enters FRMEVL (see below) then enters CHKNUM.
CHKNUM	AD8D	CD8D	CC8E	BD87	Tests VALTYP for numeric result from FRMEVL (see below), exits to READY with "TYPE MISMATCH ERROR" if string found.
CHKSTR	AD8F	CD8F	CC90	BD89	Test VALTYP for string result from FRMEVL (see below), exits to READY with "TYPE MISMATCH ERROR" if numeric found.

Label	64	VIC	B2	B4	Description
FRMEVL	AD9E	CD9E	CC9F	BD98	Input and evaluate any expression in BASIC text. Set VALTYP (00 if numeric, FF if string) and INTFLG (00 if floating point, 80 if integer). If expression is numeric floating point, result is returned in FAC. If expression is numeric integer, result is returned in (FAC+3) in HI/LO format. If expression is string, then a pointer to the string descriptor is returned in (FAC+3), this is usually a copy of VARPNT. In addition, if expression is a simple variable, then VARNAM will be set to point to the first byte of the name. Finally, if an error is found in the expression then exits to READY with "SYNTAX ERROR".
EVAL	AE83	CE83	CD84	BE81	Evaluate single term in expression. Identify functions, pi, TI, TI$ etc.
	AEA8	CEA8	CDA3	BEA0	Floating point value of pi in MFLPT format - 3.1415965.
QDOT	AEAD	CEAD	CDAB	BEA5	Evaluate non-variable term in expression.
PARCHK	AEF1	CEF1	CDEC	BEE9	Evaluate expression within parenthesis in expression.
CHKCLS	AEF7	CEF7	CDF2	BEEF	Check that character pointed to by (TXTPTR) is a right parenthesis. If not - "SYNTAX ERROR".
CHKOPN	AEFA	CEFA	CDF5	BEF2	Check that character pointed to by (TXTPTR is a left parenthesis. If not - "SYNTAX ERROR".
CHKCOM	AEFD	CEFD	CDF8	BEF5	Check that character pointed to by (TXTPTR) is a comma. If not - "SYNTAX ERROR".
SYNCHR	AEFF	CEFF	CDFA	BEF7	Check that character pointed to by (TXTPTR) is the same as contained in the accumulator. If not - "SYNTAX ERROR".
SYNERR	AF08	CF08	CE03	BF00	Print error message "SYNTAX ERROR" and return to BASIC READY.
DOMIN	AF0D	CF0D	CE08	B505	Create monadic minus or NOT for use in evaluation.
RSVVAR	AF14	CF14	--	--	Set carry if variable pointed to by (FAC+3) is a 'reserved' i.e. ST, TI, TI$.
ISVAR	AF28	CF28	CE0F	BF04	Find variable named in BASIC text. Set (VARNAM) to point at name in tables if found. Place numeric values in FAC, string pointer in (FAC+3).
TISASC	AF48	CF48	CE2E	BFAD	Convert TI to ASCII string and set (FAC+3) to point to string.
ISFUN	AFA7	CFA7	CE89	C037	Evaluate function. Return numeric value in FAC, strig value as pointer in (FAC+3).
STRFUN	AFB1	CFB1	CE93	C051	Save string descriptor of string function on stack and evaluate.
NUMFUN	AFD1	CFD1	CEB3	C071	Evaluate argument of numeric function and compute function value.

Label	64	VIC	B2	B4	Description
OROP	AFE6	CFE6	CEC8	C086	Perform OR. Set OR flag and use ANDOP to evaluate.
ANDOP	AFE9	CFE9	CECB	C089	Perform AND. Set AND flag then convert floating point values to fixed point, do AND (or OR if OR flag set) then convert back to floating point.
DOREL	B016	D016	CEF8	C0B6	Perform relations < > or =. If numeric expression uses NUMREL, string expression uses STRREL.
NUMREL	B01B	D01B	CEFD	C0BB	Perform numeric comparison.
STRREL	B02E	D02E	CF10	C0CE	Perform string comparison.
DIM	B081	D081	CF63	C121	Perform DIM.
PTRGET	B08B	D08B	CF6D	C12B	Identify variable named in BASIC text and place name, not pointer to name, in (VARNAM).
ORDVAR	B0E7	D0E7	CFC9	C187	Find variable whose name is in (VARNAM) and set (VARPNT) to point at it. If necessary, use NOTFNS to create new variable.
ISLETC	B113	D113	CFF7	C1B6	Set carry if character in accumulator is a letter.
NOTFNS	B11D	D11D	D001	C1C0	Create new variable with name as in (VARNAM), unless PTRGET called by ISVAR.
NOTEVL	B128	D128	D00C	C1CB	Create new variable with name as in (VARNAM) and set (VARPNT) to point at it.
FMAPTR	B194	D194	D078	C2C8	Set (ARYPNT) to start of array and place number of array dimensions in COUNT.
	B1A5	D1A5	D089	C2D9	Floating point value of 32768 (10000_{16}) in FLPT format.
FACINX	B1AA	D1AA	--		Convert (FAC) to integer in (A/Y).
INTIDX	B1B2	D1B2	D08D	C2DD	Evaluate expression in BASIC text as positive integer (0 to 32767) and place result in (FAC+1).
AYINT	B1BF	D1BF	D09A	C2EA	Check that FLPT value in (FAC) is within integer range (-32768 to 32767) then convert to four byte integer in FAC+1 through FAC+4.
ISARY	B1D1	D1D1	D0AC	C2FC	Get array parameters from BASIC text and push ontack.
FNDARY	B218	D218	D0F3	C343	Find array - name in (VARNAM), parameters read by ISARY.
BDSBSC	B245	D245	D020	C370	Display "BAD SUBSCRIPT" then exit to MAIN.
ILLQNT	B248	D248	D023	C373	Display "ILLEGAL QUANTITY" then exit to MAIN.
NOTFDD	B261	D261	D13C	C38C	Create array from parameters on stack.
INLPN2	B30E	D30E	D1EA	C439	Set (VARPNT) to point at element within array.
UMULT	B34C	D34C	D228	C477	Compute number of bytes in subscript (Y) of array starting at(VARPNT).
FRE	B37D	D37D	D259	C4A8	Entry point for FRE function - do garbage collection and set function value to (FRETOP) - (STREND).
GIVAYF	B391	D391	D26D	C4BC	Convert integer in (A/Y) to FLPT in (FAC) within range 0 to 32767.
POS	B39E	D39E	D27A	C4C9	Entry point for POS function - returns value of (CPOS) in FAC.
SNGFT	B3A2	D3A2	D27C	C4CB	Convert (Y) to FLPT format in (FAC) within range 0 to 255.
ERRDIR	B3A6	D3A6	D280	C4CF	Print "ILLEGAL DIRECT" if in direct mode - i.e. (CURLIN) = $FF.

Label	64	VIC	B2	B4	Description
DEF	B3B3	D3B3	D28D	C4DC	Entry point for DEF - create FN function.
GETFNM	B3E1	D3E1	D2BB	C50A	Check syntax of FN and locate FN descriptor and set (DEFPNT) to point at it.
FNDOER	B3F4	D3F4	D2CE	C51D	Entry point for FN function - get FN descriptor then ...
SETFNV	B423	D423	D2FD	C54C	Set (TXTPTR) to start at FN in text, evaluate expression, reset (TXTPTR).
STRD	B465	D465	D33F	C58E	Entry point for STR$ function, evaluate expression and convert to ASCII string.
STRINI	B475	D475	D34F	C59E	Create space for string whose descriptor is in (FAC+3) and length in (A), exit with new descriptor in (DSCTMP) and pointer to old descriptor in (DSCPNT).
STRLIT	B487	D487	D361	C5B0	Scan string starting at (A/Y) and create descriptor, exit with (FAC+3) pointing to descriptor. String expected to end with null byte or ".
PUTNW1	B4D5	D4D5	D3AF		Set descriptor on descriptor stack and update pointer.
GETSPA	B4F4	D4F4	D3CE	C61D	Set (FRETOP) and (FRESPC) for new string whose length is in (A).
GARBA2	B526	D526	D400	C66A	Do garbage collection - close up space in string space used by discarded strings.
DVARS	B5BD	D5BD	D497	--	Search variable and array tables for next string descriptor to be saved by garbage collection.
GRBPAS	B606	D606	D40E	--	Move string up to overwrite unwanted strings in garbage collection.
CAT	B63D	D63D	D517	C74F	Concatenate two strings in expression then continue to evaluate expression.
MOVINS	B67A	D67A	D554	C78C	Transfer string whose descriptor is pointed to by (STRNG1).
FRESTR	B6A3	D6A3	D57D	C7B5	Confirm string mode then ...
FREFAC	B6A6	D6A6	D580	--	Perform string housekeeping, enter with pointer to string descriptor in (FAC+3) and exi with length in (A) and (INDEX1) pointing to start of string.
FRETMS	B6DB	D6DB	D5B5	C811	Update string descriptor stack pointer.
CHRD	B6EC	D6EC	D5C6	C822	Entry point for CHR$ function.
LEFTD	B700	D700	D5DA	C836	Entry point for LEFT$ function.
RIGHTD	B72C	D72C	D600	C862	Entry point for RIGHT$ function.
MIDD	B737	D737	D611	C86D	Entry point for MID$ function.
PREAM	B761	D761	D63B	C897	Pull from stack string descriptor pointer, store in (DSCPNT), pull string parameter to (A).
LEN	B77C	D77C	D656	C8B2	Entry point for LEN function.
LEN1	B782	D782	D65C	C8B8	Do string housekeeping then force numeric mode, exit with string length in (Y).
ASC	B78B	D78B	D665	C8C1	Entry point for ASC function, get first character in string and convert to FLPT format.
GTBYTC	B79B	D79B	D675	C8D1	Evaluate expression in text. Validate that in range o to 255, else "ILLEGAL QUANTITY" error. Return value in (X).

Label	64	VIC	B2	B4	Description
VAL	B7AD	D7AD	D687	C8E3	Entry point for VAL function. Confirm argument is string then ...
STRVAL	B7B5	D7B5	D68F	C8F5	Convert string starting at (INDEX1) of length (A) to FLPT value in (FAC).
GETNUM	B7EB	D7EB	D6C6	C921	Read parameters from BASIC text for POKE or WAIT, store 1st integer in (LINNUM), 2nd integer in (X).
GETADR	B7F7	D7F7	D6D2	C92D	Convert FLPT value in (FAC) to an unsigned integer (0 to 65535) and place result in (LINNUM) and (Y/A).
PEEK	B80D	D80D	D6EB	C943	Entry point for PEEK.
POKE	B824	D824	D707	C95A	Entry point for POKE.
WAIT	B82D	D82D	D710	C963	Entry point for WAIT.
FADDH	B849	D849	D72C	C97F	Add 0.5 to (FAC).
FSUB	B850	D850	D733	C986	Floating point subtraction: (FAC) = MFLPT at (A/Y) - (FAC).
FSUBT	B853	D853	D736	C989	Entry point for subtraction: (FAC) = (ARG) - (FAC).
FADD5	B862	D862	D76E	C998	Part of addition normalisation routine.
FADD	B867	D867	D773	C99D	Floating point addition: (FAC) = MFLPT at (A/Y) + (FAC).
FADDT	B86A	D86A	D776	C9A0	Entry point for addition: (FAC) = (ARG) + (FAC).
OVERR	B97E	D97E	D88A	CAB4	Output "OVERFLOW ERROR" message.
MULSHF	B983	D983	D88F	CAB9	Multiply by a byte.
	B9BC	D9BC	D8C8	CAF2	Constant 1.0 in MFLPT format.
	B9C1	D9C1	D8CD	CAF7	Various constants used for series evaluation of functions.
LOG	B9EA	D9E	D8F6	CB20	Perform LOG function - check argument is positive, then series evaluation of LOG_e.
FMULT	BA28	DA28	D934	CB5E	Multiply (FAC) by MPLPT pointed to by (A/Y), answer in FAC.
FMULTT	BA30	DA30	D934	CB5E	Perform floating point multiply routine. Multiply (FAC) by (AFAC), answer in FAC.
MLTPLY	BA59	DA59	D965	CB8F	Multiply (FAC) by a byte, result to RESHO.
CONUPK	BA8C	DA8C	D998	CBC2	Load AFAC with MFLPT value pointed to by (A/Y).
MULDIV	BAB7	DAB7	D9C3	CBED	Multiplication subroutine to test (FAC) and (AFAC) for underflow/overflow.
MLDVEX	BAD4	DAD4	D9E0	CC0A	Handle overflow ("?OVERFLOW ERROR") or underflow error (zero FAC).
MUL10	BAE2	DAE2	D9EE	CC18	Multiply (FAC) by 10, answer in FAC.
TENC	BAF9	DAF9	DA05	CC2F	Constant 10 in MFLPT format.
DIV10	BAFE	DAFE	DA0A	CC34	Divide (FAC) by 10, answer in FAC.
FDIVF	BB07	DB07	DA13	CC3D	Divide (AFAC) by MFLPT value pointed at by (A/Y) (sign in X), answer to FAC.
FDIV	BB0F	DB0F	DA1B	CC45	Divide (AFAC) by MFLPT pointed to by (A/Y), answer to FAC.
FDIVT	BB12	DB12	DA1E	CC48	Perform floating point division routine - (AFAC) divided by (FAC), answer to FAC. On entry (A) = (FACEXP).

Label	64	VIC	B2	B4	Description
MOVFM	BBA2	DBA2	DAAE	CCD8	Load FAC with MFLPT pointed to by (A/Y).
MOV2F	BBC7	DBC7	DAD3	CCFD	Store (FAC) into TEMPF2.
MOV1F	BBCA	DBCA	DAD6	CD00	Store (FAC) into TEMPF1.
MOVXF	BBD0	DBD0	DADC	CD06	Store (FAC) into location pointed to by (FORPNT).
MOVMF	BBD4	DBD4	DAE0	CD06	Store (FAC) into location pointed to by (X/Y).
MOVFA	BBFC	DBFC	DB08	CD32	Load (FAC) from (AFAC).
MOVAF	BC0C	DC0C	DB18	CD42	Load (AFAC) from (FAC).
ROUND	BC1B	DC1B	DB27	CD51	Round off (FAC) in FAC.
SIGN	BC2B	DC2B	DB37	CD61	Find sign of (FAC), result in A - $01 = positive, $00 = zero, $FF = negative.
SGN	BC39	DC39	DB45	CD6	Perform SGN function.
ACTOFC	BC3C	DC3C	DB48	CD72	Store (A) in (FAC).
INTOFC	BC44	DC44	DB50	CD7A	Store integer in (FAC+1) as FLPT in FAC, on entry X should contain $90.
ABS	BC58	DC58	DB64	CD8E	Perform ABS function.
FCOMP	BC5B	DC5B	DB67	CD91	Compare (FAC) with MFLPT pointed to by (A/Y). Result returned in A: $01 = (FAC) > MFLTP value, $00 = equal, $FF = (FAC) < MFLPT.
QINT	BC9B	DC9B	DBA7	CDD1	Convert FLPT value in (FAC) to four byte integer in (FAC+1) in higher/lower form.
INT	BCCC	DCCC	DBD8	CE02	Perform INT - convert (FAC) to integer then convert back to FLPT in FAC.
FIN	BCF3	DCF3	DBFF	CE29	Convert ASCII string, pointed at by (TXTPTR) in BASIC text, to FLPT in FAC.
	BDB3	DDB3	DCBF	CEE9	MFLPT constants used in ASCII string conversion.
INPRT	BDC2	DDC2	DCCE	CF78	Print "IN" followed by current line number i.e. (CURLIN).
LINPRT	BDC9	DDC9	DCD5	CF7F	Print current line number from (CURLIN).
NUMPRT	BDCD	DDCD	DDC9	CF83	Display integer stored in (X/A).
INTPRT	BDD1	DDD1	DDCD	CF87	Display integer stored in (FACHO).
FACOUT	BDD7	DDD7	DCE3	CF8D	Print (FAC) as ASCII string.
FOUT	BDDD	DDDD	DCE9	CF93	Convert (FAC) to ASCII string starting at STACK and ending with null byte. Note this routine corrupts $FF, which otherwise would have been a spare zero page location.
FYOUT	BDDF	DDDF	DCEB	CF95	Convert (FAC) to ASCII string starting at STACK-1+(Y).
FOUTIM	BE68	DE68	DD74	D01E	Convert TI to ASCII string starting at STACK and ending with null byte.
	BF11	DF11	DE1D	D067	MFLPT constants used in ASCII conversion.
SQR	BF71	DF71	DE5E	D108	Perform SQR function.
FPWRT	BF7B	DF7B	DE68	D112	Perform exponentiation (raise to power). (AFAC) to the power of (FAC), answer in (FAC).
NEGOP	BFB4	DFB4	DEA1	D14B	Negate (FAC), answer in (FAC).
	BFBF	DFBF	DEAC	D156	MFLPT constants for EXP routine.
EXP	BFED	DFED	DEDA	D184	Evaluate EXP function.

Label	64	VIC	B2	B4	Description
POLYX	E043	E040	DF2D	D1D7	Evaluate series for functions. On entry (A/Y) points to single byte integer which is one less than the number of constants which follow. Routine initially converts argument to range 0 to 0.999999999.
	E08D	E08A	DF77	D221	MFLPT constants for RND evaluation.
RND	E097	E094	DF7F	D229	Perform RND evaluation.
BIOERR	E0F9	E0F6	--	--	Handle input/output error within BASIC.
BCHOUT	E10C	E109	--	-	BASIC output character routine - uses kernal CHROUT routine.
BCHIN	E112	E10F	--	--	BASIC input character routine - uses kernal CHRIN routine.
BCKOUT	E118	E115	--	--	BASIC open output channel routine - uses kernal CHKOUT routine.
BCKIN	E11E	E11B	--	--	BASIC open channel for input routine - uses kernal CHKIN routine.
BGETIN	E124	E121	--	--	BASIC get character routine - uses kernal GETIN routine.
SYS	E12A	E127	FFDE	FFDE	Perform SYS - 64/VIC set A, X, Y and SR from (SYSA), (SYSX), (SYSY), (SYSS) before entering machine code routine, and restores new values on return.
SAVET	E156	E153	FFDB	FFDB	Perform SAVE - 64/VIC fetch parameters from BASIC text before calling kernal routine, PET kernal routine reads parameters also.
SAVER	E15F	E15C	F6A7	F6E3	Save RAM to specified device - 64/VIC jump to kernal SAVE routine.
VERFYT	E165	E162	FFDB	FFDB	Perform VERIFY - 64/VIC fetch parameters from BASIC text before calling kernal routine.
LOADT	E168	E165	FFD5	FFD5	Perform LOAD - 64/VIC fetch parameters from BASIC text before calling kernal routine.
LOADR	E175	E172	F322	F356	Load RAM from specified device - 64/VIC jump to kernal routine.
OPENT	E1BE	E1BB	FFC0	FFC0	Perform OPEN - 64/VIC fetch parameters from BASIC text before calling kernal routine.
OPENR	E1C1	E1BE	F524	F563	Open specified file - 64/VIC jump to kernal routine.
CLOSET	E1C7	E1C4	FFC3	FFC3	Perform CLOSE - 64/VIC fetch parameters from BASIC text before calling kernal routine.
CLOSER	E1CA	E1C7	F2AC	F2E0	Close specified file - 64/VIC jump to kernal routine.
SLPARA	E1D4	E1D1	F43E	F47D	Get parameters from BASIC text for LOAD/SAVE/VERIFY. 64/VIC call this routine before calling kernal routine.
COMBYT	E200	E1FD	F460	F49F	If (TXTPTR) points to comma, read byte from BASIC text.
DEFLT	E206	E203	F50E	F54D	If end of statement found, unstack calling routine and exit with default parameters set.
CMMERR	E20E	E20B	F516	F555	Verify (TXTPTR) pointing to comma not followed by colon or null byte - "SYNTAX ERROR" if not.

Label	64	VIC	B2	B4	Description
OCPARA	E219	E216	F4CE	F50D	Fetch parameters from BASIC text for OPEN and CLOSE routines - set defaults.
COS	E264	E261	DFD8	D282	Evaluate COS function - add pi/2 to (FAC) then...
SIN	E26B	E268	DFDF	D289	Evaluate SIN function.
TAN	E2B4	E2B1	E028	D2D2	Evaluate TAN function by computing SIN/COS.
PI2	E2E0	E2DD	E054	D2FE	MFLPT constant pi/2.
TWOPI	E2E5	E2E2	E059	D303	MFLPT constant 2*pi.
FR4	E2EA	E2E7	E05E	D308	MFLPT constant 0.25.
	E2EF	E2EC	E063	D30D	MFLPT constants for SIN function evaluation.
ATN	E30E	E30B	E08C	D32C	Evaluate ATN function.
	E33E	E33B	E0BC	D35C	MFLPT constants for ATN function evaluation.
BASSFT	E37B	E467	--	--	BASIC warm restart routine called by BREAK if BRK istruction encountered or STOP/RESTORE pressed. Closes channels, restores default I/O, resets stack, and exits through IERROR with (X) = $80.
INITV	E453	E45B	--	--	Copies BVTRS to RAM block 0.
INIT	E394	E378	--	--	Initialise BASIC on reset (cold start) - if 64/VIC, call INITV to set BASIC vectors in $0300 - $030B, then..
INITNV	E397	E37B	E116	D3B6	Call INITCZ to set up BASIC variable in block 0 of RAM, (PETs do destructive RAM test above $0400, 64/VIC don't), call INTMS, then exit to BASIC READY.
INITAT	E3A2	E387	E0F9	D399	CHRGET routine master copy - copied down to zero page by INITCZ.
RNDSED	E3BA	E39F	E111	D3B1	MFLPT constant 0.811635157 used as initial seed for random number generation.
INITCZ	E3BF	E3A4	--	--	Initialise BASIC RAM - set USRPOK, ADRAY1, ADRAY2, copy INITAT and RNDSED to CHRGET and RNDX, set TXTTAB and FRETOP to (LORAM), HIRAM), set first byte in BASIC text area to zero.
WORDS	E460	E429	E1B7	D44B	Text "BYTES FREE".
FREMES	E473	E436	E1C4	D458	Text "??? COMMODORE BASIC ???" etc.
BVTRS	E447	E44F	--	--	ROM copies of BASIC vectors.
INITV	E453	E45B	--	--	Copies BVTRS to RAM block 0.

Label	64	VIC	B2	B4	Description
IOBASK	E500	E500	--	--	Returns in (X/Y) the address of 6526 Complex Interface Adaptor (CIA) chip used by IRQ routines (and hence keyboard routines). This is the IOBASE Kernal routine.
SCRORG	E505	E505	--	--	Returns screen organisation columns (X), rows (Y). Entry through SCREEN Kernal vector.
PLOTK	E50A	E50A	--	--	Sets/returns cursor position: screen row through (X), column through (Y). Sets cursor if carry clear, returns position if carry set on entry. Entry through PLOT Kernal vector.
CINT	E518	E518	E1DE	E60F	Initialise input/output. This routine is called by the IOBASE Kernal vector.
NXTD	E566	E581	E257	E05F	Home screen, reset screen line link table.
STUPT	E56C	E587	E25D	E06F	Move cursor to (TBLX), (PNTR).
VPAN	E59A	E5B5	--	--	Reset default I/O, including VIC-II chip registers.
PANIC	E5A0	E5BB	--	--	Rest default I/O channels then ...
INITV	E5A8	E5C3	--	--	Restore default values of 6567 (VIC-II) chip registers.
KBGET	E5B4	E5CF	E285	E0A7	Get characters from keyboard buffer. GETIN comes here if DFLTN = 0.
KBINP	E5CA	E5E5	E29A	E0BC	Input character (as distinct from GET). KSINP comes here if CRSW=0.
KSINP	E632	E64F	E2F4	E116	Input character from keyboard or screen. CHRIN comes here if DFLTN = 0.
SCINP	E63A	E657	E2FF	E121	Input character from screen. KSINP comes here if CRSW = 3.
QTSWC	E684	E6B8	E33F	E16A	Toggle quote flag (QTSW). During input stops tokenisation of keywords within quotes.
SCPUT	E691	E6C5	E34C	E177	Prints (A) to screen. Used by SCNPNT.
SCNPNT	E716	E742	E3D8	E202	Prints character to screen interprets cursor controls, colour changes, case changes etc.
CKDECL	E8A1	E8E8	--	--	Check for decrement of line counter.
CKINL	E8B3	E8FA	--	--	Check for increment of line counter.
CKCOLR	E8CB	E912	--	--	Check colour.
COLTAB	E8DA	E921	--	--	Table of colour codes.
SCROL	E8EA	E975	E53F	E3C8	Screen scrolling routines.
CLRLN	E9FF	EA8D	--	E396	Clear the screen line pointed to by (x) ie. 0 to 24.
DSPP	EA13	EAA1	--	--	Display (A) on screen.
KEY	EA31*	EABF*	E61B*	E442*	The main IRQ interrupt handling routine. [*CINV vector points herel.
SCNKYK	EA87	EB1E	E68E	E4CD	Keyboard scan routine, check for key depression and place characters in keyboard queue. This is the routine pointed to by the Kernal vector SCNKEY.
KEYCOD	EB79	EC46	E6F8	E6D1	Keyboard matrix tables. Used by SCNKYK to convert key depression to ASCII character. Tables exist for the various shft modes.

Label	------64--------		VIC	B2/B4	
	Hex	Dcml	Hex	Hex	Description
TALKK	ED09	EE14	F0B6	F0D2	ORs (A) to convert device number to a TALK address for the IEEE bus and transmits this as a command. This is the Kernal routine pointed to by TALK.
LISTNK	ED0C	EE17	F0BA	F0D5	ORs (A) to convert device number to a LISTEN address for the IEEE bus and transmits this as a command. This is the Kernal routine pointed to by LISTEN.
SECNDK	EDB9	EEC0	--	--	Converts (A) and transmits it as a LISTEN secondary address on the IEEE bus. This is the Kernal routine accessed by SECOND.
TKSAK	EDC7	EECE	--	--	Converts (A) and transmits it as a TALK secondary address on the IEEE bus. This is the kernel routine actioned by TKSA.
CIOUTK	EDDD	EEE4	F16F	F19E	Transmits a byte on to the IEEE bus. The character is buffered so that the 'hand shaking' can be carried out. This is the CIOUT Kernal routine.
UNTLKK	EDEF	EEF6	F17F	F1AE	Transmits an UNTALK command on to the IEEE bus. This is the Kernal routine which is addressed by the UNTALK vector.
UNLSNK	EDFE	EF04	F183	F1B9	Transmits an UNLISTEN command on the IEEE bus. The UNLSN Kernal vector comes here.
ACPTRK	EE13	EF19	F18C	F1C0	A byte is 'hand shaken' off the IEEE bus and placed in A. This is the ACPTR Kernal routine.
RSTRAB	EEBB	EFA3	--	--	Continuation of the main NMI interrupt routine used for RS232 devices.
RSTBGN	EF06	EFEE	--	--	Outputs a byte to the RS232 channel (2).
RSRCVR	EF59	F036	--	--	Part of the NMI interrupt routine which builds the individual bit coming from the RS232 channel into a byte.
KMSGTX	F0BD	F174	F000	F000	Text of the Kernal error and control messages is stored here.
KMSSG	F12B	F1E2	F156	F185	Print Kernal message to the screen if output is enabled i.e. (MSGFLG) has bit 7 set.
NGETIN	F13E	F1F5	F1D1	F205	Get character from channel and return in A. If no character has been sent, return 0. This is the Kernal GETIN routine.
NBASIN	F157	F20E	F1E1	F215	Input character from buffer into A. This is the CHRIN Kernal routine.
NBSOUT	F1CA	F27A	F232	F266	Output the byte in A to the output channel. This is the CHROUT Kernal routine.
NCHKIN	F20E	F2C7	F7BC	F7FE	Allocates the file specified by (X) as the input channel. This is the routine used by the CHKIN Kernal vector.
NCKOUT	F250	F309	F770	F71F	Allocates the file specified by (X) as the output channel. This is the routine used by the CHKOUT Kernal vector.

Label	64	VIC	B2	B4	Description
NCLOSE	F291	F34A	F2AC	F2E0	(A) specifies the file to be closed. The details are removed from the device tables (LAT, FAT and SAT). Output files are 'tidied up'. This is the CLOSE Kernal routine.
NCLALL	F32F	F3EF	F26E	F2A2	This routine aborts all current I/O. The number of open files (LNTND) is set to zero and any IEEE files are UNTALKed or UNLISTENed. The routine does not close 'output' files tidily, so mayonly be safely used with input files (use CLOSE for output files). This is the CLALL Kernal routine.
NCLRCH	F333	F3F3	F272	F2A6	Deallocates the input/output channels and restores the default devices (DFLTN = 0, DFLTO = 3). This is the CLRCHN Kernal routine.
NOPEN	F34A	F40A	F524	F563	Opens the file whose specification is stored in FNLEN, LA, FA, SA and FNADR by inserting the details in the LAT, FAT and SAT tables and carrying out appropriate procedures for files on tape or disk. This is the Kernal OPEN routine.
LOADSP	F49E	F542	F3CC	F40B	Load RAM (if A contains 0) or verify (if A contains 1) the file specified by FNADR, FNLEN from device specified by FA. 64/VIC will, if SA contains 0, load file into RAM starting at (X/Y). Otherwise, the address stored with the file is used. This is the LOAD Kernal routine.
LUKING	F5AF	F647	--	--	Subroutine to display "SEARCHING FOR" then....
OUTFN	F5C1	F659	--	--	Display file name specified by FNADR, FNLEN.
LODING	F5D2	F66A	--	--	Subroutine to display "LOADING" or "VERIFYING" (depending on A) then does OUTFN.
SAVESP	F5DD	F675	F6A4	F6E3	Save the RAM specified by A,X and Y as a file specified by FNADR,FNLEN, on device FA. If SA contains 2 a file saved to tape will have and end of tape marker written after program. 64/VIC, if SA contains 1 or 3, program saved to tape will automatically reload to memory from whence it was saved.
SAVING	F68F	F728	--	--	Subroutine to display "SAVING" file name.
UDTIMK	F69B	F734	F729	F768	Part of the IRQ interrupt servicing routine which updates the real time jiffy clock. In addition it stores the current keyboard matrix value in STKEY, which enables STOP to function. This is the UDTIM Kernal routine.
RDTIMK	F6DD	F760	--	--	Subroutine to read jiffy clock into (A/X/Y). This is the Kernal RDTIM routine.
SETTMK	F6E4	F767	--	--	Subroutine to load (A/X/Y) into the jiffy clock. This is the Kernal SETTIM routine.
NSTOP	F6ED	F770	F30F	F343	Suboutine which checks the value stored in STKEY and returns with Z flag set if the value represents a depression of the STOP key. The routine also closes active channels (using CLRCHN) and flushes the keyboard queue. This is the Kernal STOP routine.

Label	64	VIC	B2	B4	Description
KERROR	F6EB	F77E	F315	F349	Errors detected by the Kernal routines enter this routine to issue the appropriate error message.
FAH	F72C	F7AF	F5A6	F5E5	Find and read the header block on tape.
TCNTL	F80D	F88A	F806	F84B	The tape control routines reside here. They undertake functions such as switching cassette motors on and off, timing etc.
READ	F92C	F98A	F855	F89A	The tape reading routines.
RDI5	FA70	FABD	F931	F976	Byte handling routine for tape reading.
WRITE	FBA6	FBEA	FB93	FBD8	Tape writing routies.
START	FCE2	FD22	FCD1	FD16	Cold start routine normally accessed when the system is initially switched on. It is the routine which is pointed to by the harware vector at $FFFC. Memory is initialised, and all input/output devices are set up. The first part of the 64/VIC routine checks if a cartridge is loaded in block 8 (64) OR A (VIC) and if so, jumps to the cartridge for initialisation. MEMCHK -- FE91 -- -- The routine on the VIC checks on what memory enhancements are fitted and sets up the screen and start of BASIC text accordingly.
AOINT	FD02	FD3F	--	--	Subroutine checks for cartridge in 8 block (64) or A block (VIC). Returns with Z flag set if cartridge found.
RESTRK	FD15	FD52	--	--	Restores Kernal indirect vectors to standard. This is the Kernal RESTOR routine.
VECTRK	FD1A	FD57	--	--	Loads Kernal indirector vectors with user's values. This is the Kernal VECTOR routine.
RAMTSK	FD50	FD8D	--	--	Subroutine sets zero page up and does non-destructure test of whole of RAM until check fils. Address of check failure stored in MEMSIZ using MEMTOP. This is the Kernal RAMTAS routine.
IOINTK	FDA3	FDF9	--	--	Initialise 6526 CIA's (64), 6522 VIA's (VIC) or 6521 PIA's and 6522 VIA (PET) to standard values. This is the Kernal IOINIT routine.
SETNMK	FDF9	FE49	--	--	Stores X,Y in FNADR and A in FNLEN. This is the Kernal SETNANE routine.
STLFSK	FE00	FE50	--	--	Stores A, X and Y in LA, FA, SA. This is the Kernal SETLFS routine.
READSS	FE07	FE57	--	--	Reads status flag into A. If FA contains 2, reads RSSTAT (RS232 status) else reads STATUS. This is the Kernal READST routine.
STMSGK	FE18	FE66	--	--	Sets the Kernals message flag MSGFLG to value of A. On exit A contains value of STATUS (never RSSTAT). This is the Kernal SETMSG routine.
STTMOK	FE21	FE6F	--	--	Stores A into TIMOUT flag to enable or defeat parallel EEE bus time outs. This is the Kernal SETTMO routine.
MEMTPK	FE25	FE73	--	--	Reads/sets (X/Y) from/into MEMSIZ depending on carry flag. This is the Kernal MEMTOP routine.

Label	64	VIC	B2	B4	Description
MEMBTK	FE25	FE73	--	--	Reads/sets (X/Y) from/into MEMSTR depending on the carry flag. This is the Kernal MEMBOT routine.
NMI	FE43	FEA9	--	--	On the 64/VIC NMI interrupts are mainly used to handle RS232 devices. PET's made no use of NMI at all.
BAUDTB	FEC2	FF5C	--	--	Baud rate tables for the 64/VIC.
T2NMI	FED6	--	--	--	Subroutine to handle RS232 bit input.
PULS	FF48	FF72	E61B	E442	This routine is entered when an interrupt occurs. Registers are saved and the source of the interrupt is determind, IRQ or BRK instruction. Appropriate actions are then taken.
PCINT	FF58	--	--	--	Patch to CINIT routine which determines whether PAL or NTSC standard is being used by checking number of raster lines.
RLSNUM	FF80	--	--	--	Release/version numbers of Kernal operating system. If your value differs from 0 then some of the addresses listed for the 64 above may be slightly different.
KVCTRS	FF81	FF8A	FFC0	FFC0	Kernal jump vector table.
NMIVEC	FFFA	FFFA	FFFA	FFFA	NMI interrupt vector address.
RSTVEC	FFFC	FFFC	FFFC	FFFC	Initialisation vector address.
IRQVEC	FFFE	FFFE	FFFE	FFFE	IRQ interrupt vector address.

The Kernal Vectors - Easy Access to Operating System Routines.

The prudent programmer makes as much use as possible of the ROM routines provided with his/her computer; after all if one wants to re-invent wheels then there are plenty of hobbies which seem to be designed to cater for this. However, there is a problem with this approach. What happens to your clever programs when you change your computer for the super new Commodore computer with three times the power at a quarter of the price next year ? All those JSR's to the ROM routines are now going to the wrong address, and a massive re-write is required.

The purpose of the Kernal is to avoid this problem and to make it easier to write programs which will transfer. The Kernal is simply a 'jump table' in the ROM which provides the link to the routine. The address of the jump stays in the same place, although the routine to which it jumps may be moved to different places in the new machines. All the programmer has to do is JSR to the address which gives access to the routine that he/she requires, this address always remaining the same.

The CBM 64 has a Kernal jump table providing access to 39 of the routines in the operating system ROM. These routines have been chosen as the ones most likely to be of use to the programmer. It is part of the understanding that you have with Commodore, that these Kernal entries will be preserved on any upgrades of the operating system or computer. Thus the CBM 64 Kernal includes all the VIC 20 Kernal, so any machine code program which was written for the VIC 20 which depended on use of the VIC 20 Kernal, will run on the CBM 64.

Using the Kernal routines is quite straightforward. Firstly, if you need to provide data to the routine, then you must provide this in the form that the routine expects. This usually means loading the values into one or more of the registers. Then you simply JSR to the Kernal address. If the routine returns any values, then these too are normally returned in one or more of the registers.

The thirty nine CBM 64 Kernal routines are described below:

ACPTR: Read a byte from an IEEE Serial Device.
Call address: $FFA5 (65445)
Communicating registers: A.
Registers affected: A, X.
Preparatory routines: TALK, TKSA.
Associated routines: TALK, TKSA, UNTLK.
Error returns: See READST.

Reads a byte from a serial device (e.g. disk) with full handshaking. The device must have previously been commanded to TALK. The byte is returned in A. Beginners will find that using SETLFS, SETNAM, OPEN and CHRIN provides an easier approach to reading data from the serial bus.

CHKIN Open a channel for input.
Call address: $FFC6 (65478)
Communicating registers: X.
Registers affected: A, X.
Preparatory routines: OPEN.
Associated routines: SETLFS, SETNAM, OPEN, CHRIN, CLRCHN, CLOSE, CLALL.
Error returns: 3, 5 or 6 (see READST).

A logical file previously opened by OPEN is defined as an input channel in order that it may be read. Load X with logical number of file.

CHKOUT Open a channel for output.
Call address: $FFC9 (65481)
Communicating registers: X.
Registers affected: A, X.
Preparatory routines: OPEN.
Associated routines: SETLFS, SETNAM, OPEN, CHROUT, CLRCHN, CLOSE CLALL.
Error returns: 0, 3, 5, 7 (see READST).

A logical file previously opened by OPEN is defined as an output channel in order that it may be written to. Load X with logical number of file.

CHRIN Input a byte from the input channel.
Call address: $FFCF (65487).
Communicating registers: A.
Registers affected: A, X.
Preparatory routines: OPEN, CHKIN.
Associated routines: SETLFS, SETNAM, OPEN, CHKIN, CLRCHN, CLOSE, CLALL.
Error returns: (see READST).

Reads a byte of data from the channel opened for input. Byte returned in A.

CHROUT Output byte to the output channel.
Call address: $FFD2 (65490).
Communicating registers: A.
Registers affected: A.
Preparatory routines: OPEN, CHKOUT.
Associated routines: SETLFS, SETNAM, OPEN, CHKOUT, CLRCHN, CLOSE, CLALL.
Error returns: 0 (see READST).

Output the byte in A to the channel opened for output.

CINT Initialise screen editor and 6567 VIC-II chip.
Call address: $FF81 (65409).
Communicating registers: none.
Registers affected: A, X, Y.
Preparatory routines: none.
Error returns: none

This routine may be used to restore the VIC-II chip to its normal status, for instance, following hi-resolution graphics.

CIOUT Output a byte to a device on the IEEE serial bus.
Call address: $FFA8 (65448).
Communicating registers: A.
Registers affected: none.
Preparatory routines: LISTEN, SECOND.
Associated routines: LISTEN, SECOND, UNLSN.
Error returns: (see READST).

Outputs the byte in A to the serial bus. Beginners will find that using SETLFS, SETNAM, OPEN and CHROUT provides an easier approach to writing data to the serial bus.

CLALL Close all files currently open.
Call address: $FFE7 (65511).
Communicating registers: none.
Registers affected: A, X.
Preparatory routines: none.
Associated routines: SETLFS, SETNAM, OPEN, CHKIN, CHRIN, CHKOUT, CHROUT, CLRCHN, CLOSE.
Error returns: none.

This routine may be called to close all currently open files.

CLOSE Close the specifed logical file.
Call address: $FFC3 (65475).
Communicating registers: A.
Registers affected: A, X, Y.
Preparatory routines: none.
Associated routines: SETLFS, SETNAM, OPEN, CHKIN, CHRIN, CHKOUT, CHROUT, CLRCHN, CLALL.
Error returns: 0, 240 (see READST).

Closes the file whose logical number is supplied in A.

CLRCHN Close (clear) all Input/output channels.
Call address: $FFCC (65484).
Communicating registers: none
Registers affected: A, X.
Preparatory routines: none.
Associated routines: SETLFS, SETNAM, OPEN, CHKIN, CHRIN, CHKOUT, CHROUT, CLOSE, CLALL.
Error returns: none.

Closes the current input/output channels and resets default channels (keyboard and screen).

GETIN Get a character from the keyboard or RS232 device.
Call address: $FFE4 (65508).
Communicating registers: A.
Registers affected: A, X, Y.
Preparatory routines: OPEN, CHKIN.
Error returns: (see READST).

Reads a byte into A from the input channel. Should only be used with the keyboard or an RS232 device. If no character is available, then a zero is returned in A. CHRIN is the preferred routine for all devices.

IOBASE Returns the address of the 6526 CIA.
Call address: $FFF3 (65523).
Communicating registers: X, Y.
Registers affected: X, Y.
Preparatory routines: none.
Error returns: none.

Returns in (X/Y) the address of the 6526 Complex Interface Adaptor (CIA) used for IRQ (and hence keyboard processing). If programs which handle the CIA registers use an offset from the address obtained via this routine, then interchangability of programs will be considerably improved. For the CBM 64, the address returned is $DC00, VIC 20 returns $9100.

IOINIT Initialise all input/output devices and routines.
Call address: $FF84 (65412).
Communicating registers: none.
Registers affected: A, X, Y.
Preparatory routines: none.
Error returns: none.

May be used to restore all input/output devices to their normal condition. Devices include 6567 VIC-II, the two 6526 CIAs as well as the 6581 SID.

LISTEN Command the specified device on the serial bus to listen.
Call address: $FFB1 (65457).
Communicating registers: A.
Registers affected: A.
Preparatory routines: none.
Associated routines: SECOND, CIOUT, UNLSN.
Error returns: (see READST).

This routine will command the device specified by (A), on the IEEE serial bus to listen. CIOUT may then be used to transmit bytes to the device.

LOAD Load (or verify) RAM from device.
Call address: $FFD5 (65493).
Communicating registers: A, X, Y
Registers affected: A, X, Y.
Preparatory routines: SETLFS, SETNAM.
Error returns: 0, 4, 5, 8, 9 (see READST).

Loads (or verifies) RAM from the device defined by the SETLFS and SETNAM routines. (A) specifies the action: 0 = load, 1 = verify. If the secondary address specified in the corresponding SETLFS routine was 0, then (X/Y) specify the load address. A secondary address of 1 or 2 causes the address stored in the header of the file to be used as a load address.

MEMBOT Return or set the address of the bottom of available memory.
Call address: $FF9C (65436).
Communicating registers: X, Y.
Registers affected: X, Y.
Preparatory routines: none.
Error returns: none.

If carry is set on entry, routine returns the address of the bottom of memory in (X/Y). If carry is clear, the (X/Y) is used to set the bottom of memory.

MEMTOP Return or set top of memory.
Call address: $FF99 (65433).
Communicating registers: X, Y.
Registers affected: X, Y.
Preparatory routines: none.
Error returns: none.

Similar in action to MEMBOT, but used to return or set the top of memory.

OPEN Open a logical file.
Call address: $FFC0 (65472).
Communicating registers: none.
Registers affected: A, X, Y.
Preparatory routines: SETLFS, SETNAM.
Associated routines: SETLFS, SETNAM, CHKIN, CHRIN, CHKOUT, CHROUT, CLRCHN, CLOSE, CLALL.
Error returns: 1, 2, 4, 5, 6, 240 (see READST).

Used to open the file which has previously been specified by the SETLFS and SETNAM routines.

PLOT Return or set current cursor position.
Call address: $FFF0 (65520).
Communicating registers: A, X, Y.
Registers affected: A, X, Y.
Prepatatory routines: none.
Error returns: none.

This routine may be used to obtain the current position (row, column) of the cursor, or to move the cursor to a specified position. Entry with carry set obtains the current row (X), and column (Y). Entry with carry clear move the cursor to the row and column specified by X and Y.

RAMTAS Initialise memory, including non-destructive test of RAM above $03FF.
Call address: $FF87 (65415).
Communicating registers: A, X, Y.
Registers affected: A, X, Y.
Preparatory routines: none.
Error returns: none.

Normally used during the initialisation of the CBM 64 after switching on. Clears and resets $0000 to $0101, and $0200 to $03FF, carries out a non-destructive test of RAM above $0400, resets the screen base to $0400.

RDTIM Read system clock.
Call address: $FFDE (65502).
Communicating registers: A, X, Y.
Registers affected: A, X, Y.
Preparatory routines: none.
Error returns: none.

Returns the system (jiffy) clock into A (msb), X, Y (lsb). The only advantage of this routine is that the user doesn't need to know where the clock is maintained. It is located in $A2, $A1, $A0 on the CBM 64, but it could be elsewhere on another machine.

READST Read input/output status word.
Call address: $FFB7 (65463).
Communicating registers: A.
Registers affected: A.
Preparatory routines: none.
Error returns: none.

This routine returns the current status of the input/output devices in A. It is usual to call the routine following any input/output operation which might result in an error. The bits returned in the accumulator may be interpreted as shown in the table below:

Bit position	Numeric value	Tape read	Serial I/O	Tape load/verify
0	1		Time out write	
1	2		Time out read	
2	4	Short block		Short block.
3	8	Long block.		Long block.
4	16	Read error.		Mismatch.
5	32	Checksum.		Checksum.
6	64	End of file.	EOI read.	
7	-128	End of tape.	Device not present.	End of tape.

RESTOR Restore default system and interrupt vectors.
Call address: $FF8A (65418).
Communicating registers: none.
Registers affected: A, X, Y.
Preparatory routines: none.
Associated routines: VECTOR.
Error returns: none.

This routine restores the defaults addresses in the page 3 vectors to BASIC and the operating system, and the three interrupts.

SAVE Save RAM to the device.
Call address: $FFD8 (65496).
Communicating registers: A, X, Y.
Registers affected: A, X, Y.
Preparatory routines: SETLFS, SETNAM.
Associated routines: SETLFS, SETNAM.
Error returns: 5, 8, 9 (see READST).

The routine saves a section of memory to a device previously specified by SETLFS and SETNAM routines. The addresses of the start and the end of the RAM to be saved is specified in A, X and Y. As there are only three registers, and there are four bytes of data in the two addresses to be passed, (A) is used to specify the low byte of a zero page address which is the load address. (X/Y) should specify an address one greater than the last byte to be saved.

SCNKEY Scans the keyboard and returns the ASCII value of any key currently being pressed.
Call address: $FF9F (65439).
Communicating registers: none.
Registers affected: A, X, Y.
Preparatory routines: none.
Associated routines: STOP, UDTIM.
Error returns: none.

In the normal course of events, the keyboard is scanned once every fiftieth of a second and any keys being pressed are transferred to the keyboard buffer. The SCNKEY routine is the routine which undertakes this task. The scanning is part of the normal IRQ interrupt servicing procedure. It may sometimes occur that the normal IRQ interrupt is disallowed, either by a SEI, or, perhaps by trapping the IRQ and not allowing normal keyboard scanning. In this case, the user may call the SCNKEY routine. This should always be done prior to any attempt to read the keyboard using GETIN ot CHRIN, when normal interrupt is suspended.

SCREEN Returns the format of the screen.
Call address: $FFED (65517).
Communicating registers: X, Y.
Registers affected: X, Y.
Preparatory routines: none.
Error returns: none.

Returns the format of the screen: columns in X and rows in Y. It is therefore possible for a machine code program to discover the width of the screen and adjust itself accordingly.

SECOND Send secondary address for LISTEN to a device on the serial bus.
Call address: $FF93 (65427).
Communicating registers: A.
Registers affected: A.
Preparatory routines: LISTEN.
Associated routines: ACPTR.
Associated routines: LISTEN, ACPTR, UNLSN.
Error returns: (see READST).

This routine is used to transmit a secondary address on the serial bus following a LISTEN routine, in order that the device which is to listen may be identified.

SETLFS Set up a logical file.
Call address: $FFBA (65466).
Communicating registers: A, X, Y.
Registers affected: none.
Preparatory routines: although SETNAM and SETLFS may be called in any order, both must be called.
Associated routines: SETNAM, OPEN, SAVE, LOAD.
Error returns: none.

This routine is an essential preliminary for many of the input/output routines. The logical file number, first address (or device number as it is better known) and secondary address (if any) are transmitted to the routine through A, X an Y respectively. Thus to open the printer with a logical file number of 2, and transmit a secondary address of 7 for lower case, the following code would be written:

```
LDAIM 2 / LDXIM 4 /  LDYIM 7 / JSR $FFBA.
```

SETMSG Controls the format of the system messages output by the operating system.
Call address: $FF90 (65424).
Communicating registers: A
Registers affected: A.
Preparatory routines: none.
Error returns: none.

This little used routine controls the printing of error and control messages. Bits 7 and 6 of the value supplied in A determine the printing. If bit 7 is set then error messages are printed, bit 6 set permits control messages to be printed.

SETNAM Set up file name.
Call address: $FFBD (65469).
Communicating registers: A, X, Y.
Registers affected: none.
Preparatory routines: although SETNAM and SETLFS may be called in any order, both are required.
Associated routines: SETLFS, OPEN, LOAD, SAVE.
Error returns: none.

This routine is used prior to opening, saving or loading. The length of the file name in bytes is loaded into A, and the address where the file name string is stored is supplied in (X/Y). If no file name is desired then the routine is called with a zero loaded into A.

SETTIM Set the system (jiffy) clock.
Call address: $FFDB (65499).
Communicating registers: A, X, Y.
Registers affected: none.
Preparatory routines: none.
Error returns: none.

The three bytes of the jiffy clock are loaded with (A) msb, (X) and (Y) lsb by this routine.

SETTMO Set IEEE bus timeout flag.
Call address: $FFA2 (65442).
Communicating registers: A
Registers affected: none.
Preparatory routines: none.
Error returns: none.

This routine may be used to enable or disable timeout trapping on the IEEE bus. CBM micro-computers controlling a parallel IEEE bus will normally abandon a transaction if no response is received within 64 milliseconds. This timeout may be defeated by calling SETTMO with bit 7 of (A) set, and enabled if bit 7 is clear. Note, the CBM 64 serial bus is not affected by the timeout flag, however, this routine may be required if it is anticipated that the program may be used to handle the PET range of IEEE devices (using an appropriate hardware interface).

STOP Check if STOP key has been pressed.
Call address: $FFE1 (65505).
Communicating registers: A
Registers affected: A, X
Preparatory routines: none.
Associated routines: UDTIM, SCNKEY.
Error returns: none.

If the STOP was pressed during the last keyboard scan, this routine returns with the Z flag set. In the event that the normal keyboard interrupt has been disabled, it is necessary to issue a call to UDTIM prior to STOP.

TALK Command a device on the IEEE serial bus to TALK.
Call address: $FFB4 (65460)
Communicating registers: A
Registers affected: A
Preparatory routines: none.
Associated routines: TKSA, ACPTR, UNTLK
Error returns: (see READST).

The accumulator should contain the number of the device on the serial bus which is commanded to TALK.

TKSA Send a secondary address to the device which is commanded to TALK.
Call address: $FF96 (65430).
Communicating registers: A
Registers affected: A
Preparatory routines: TALK
Associated routines: TALK, ACPTR, UNTLK
Error returns: (see READST).

The routine transmits a secondary address on the IEEE serial bus to a device which is commanded to TALK.

UDTIM Update the system clock.
Call address: $FFEA (65514).
Communicating registers: none.
Registers affected: A, X.
Preparatory routines: none.
Associated routines: SCNKEY, STOP.
Error returns: none.

The name of this routine is somewhat deceptive. In addition to updating the system clock it also scans the keyboard and stores the current key matrix in STKEY. As a consequence it is possible for the STOP Kernal routine to determine whether the STOP key has been depressed, even when the normal IRQ interrupt routine has been disabled. Provided the user issues a call to UDTIM (or SCNKEY) prior to any call to the STOP routine, then STOP key detection will take place.

UNLSN Send an UNLISTEN command to all devices on the IEEE serial bus.
Call address: $FFAE (65454).
Communicating registers: none.
Registers affected: A
Preparatory routines: none.
Associated routines: LISTEN, SECOND, CIOUT.
Error returns: (see READST).

This routine commands all devices currently LISTENing on the IEEE serial bus to stop LISTENing.

UNTLK Send an UNTLK command to all devices on the IEEE serial bus.
Call address: $FFAB (65451)
Communicating registers: none.
Registers affected: A.
Preparatory routines: none.
Associated routines: TALK, TKSA, ACPTR.
Error returns: (see READST).

All devices currently TALKing on the IEEE serial bus will stop sending data following the issue of this command.

VECTOR Read/set the system vector jump addresses stored in RAM.
Call address: $FF8D (65421).
Communicating registers: X, Y.
Registers affected: A, X, Y.
Preparatory routines: none.
Associated routines: RESTOR.
Error returns: none.

Calling this routine with carry set will cause the current contents of the system vectors to be stored in a list whose address is pointed to by (X/Y). If called with carry clear the vectors will be restored from this list.

INDEX

E = Explained

P = First Program

NOTES

beginners Assembly for the CBM 64

A complete course for the Commodore 64 covering the whole instruction set for the new Commodore micro-processor.

Based on the earlier successful assembly language books for the Pet and the VIC 20 this book follows the highly-praised format of these to provide a carefully structured self-paced course. This structure allows you the reader to proceed at your own pace, the important thing being that you get there - quickly or slowly - depending on your inclination.

To make really sure that you get going immediately the tape supplied with the book contains a full 6510 assembler, a full machine language monitor and a binary/hexadecimal/BCD self tutoring program.

Just to let you know how well you are doing, each chapter has exercises - along with solutions and explanations. These re-enforce your reading and allow you to test yourself and monitor your progress - in private.

The book will serve as a complete reference work, the appendices describing the machine in great detail. Appendix II gives complete detail of the instruction set. Appendix III, in particular, provides a comprehensive guide to the CBM 64, VIC 20 and Pet ROMS never previously published.