An Introduction to 6502 Microprocessor Applications

DT102 Curriculum Manual

Lesson Module: 8.12/1
Issue: MP615/A

About this Manual

For tomorrows' engineers and technicians, training in the use of microprocessor systems and the design of control tasks will be very important.

We see microprocessors used in almost every area of modern life. They control domestic appliances, automated Teller machines, VCRs, automobile engine management and braking systems and so on - the applications are endless. In addition to these less obvious uses, microprocessors dominate today's working environment in the shape of the personal computer.

To gain a good working knowledge of microprocessor technology you will need to follow this manual carefully. It will lead you in a step by step manner through the following areas:

- Using the MAC III microcomputer.
- Introduction to 6502 programming.
- Writing Machine Code Programs.
- Program Debugging.
- Using the Merlin Text Editor.
- Introduction to Development Systems.
- Addressing Modes.
- Negative Binary Numbers.
- Programs with Loops.
- Further Programs with Loops.
- Indexed Addressing.
- Logical and Test Instructions.
- Input and Output Programming
- Programming the Applications Module.
- Stacks and Subroutines.
- Interrupts.

As you work through each chapter you will be guided by a series of student objectives and your progress will be continually assessed by questions in the Exercises, Practical Assignments and Student Assessments.

The Practical Assignments presented throughout the manual are graded in terms of complexity, starting with simple machine code programs and ending with more complex programming techniques in assembler code.

Your instructor has a copy of the Solutions book for this manual. It contains all the solutions to the assessment questions together with suggested solutions to all the programming tasks. Copies of these programs are provided on a disk supplied with the Solutions book.

What do I need to work through this manual?

To work through this manual you will need the following items:

1. MAC III 6502 microprocessor board.
2. Merlin Development System software pack (6502/Z80 version), including 6502 Cross Assembler Reference Manual and RS232 cable.
3. Microprocessor Applications board.
4. Personal Computer (PC) running Windows 95 or later, and fitted with RS232 serial communications (COM) port.
5. Two 0.1” shorting leads (supplied)
6. MAC III User Manual.
7. 6502 Instruction Set Reference Manual.
8. Note pad and pencil.

In addition, you will need a **power supply** and a **keypad/display unit**. The form that these items take will depend on whether you are using a *Digiac 2000* system or a *Digiac 3000* system:

	Power supply required	**Keypad/display unit required**
Digiac 2000 system	DT60 Power Supply unit	DT25 Keypad/display module
Digiac 3000 system	D3000 Experiment Platform or D3000 Virtual Instrument Platform	D3000-8.0 Microprocessor Master Board with built-in keypad/display

For further information, please refer to the MAC III 6502 User Manual.

Computerized Assessment of Student Performance

If your laboratory is equipped with the ***DIGIAC 3000*** Computer Based Training System, then the system may be used to automatically monitor your progress as you work through this manual.

If your instructor has asked you to use this facility, then you should key in your responses to the questions in this manual at your computer managed workstation. To remind you to do this, a 🖩 symbol is printed alongside questions that require a keyed-in response.

The following D3000 Lesson Module is available for use with this manual:

D3000 Lesson Module 8.12

Additional Teachware

If you are encountering microprocessors for the first time, it is recommended that you begin by reading the manual "An Introduction to Microprocessor Technology", which is available from LJ Technical Systems.

Other manuals available in this range are:

An Introduction to 6502 Microprocessor Troubleshooting.
An Introduction to Z80 Microprocessor Applications.
An Introduction to Z80 Microprocessor Troubleshooting.
68000 Microprocessor Concepts and Applications.
An Introduction to 68000 Microprocessor Applications.

Contents

Appendices **Pages**

Chapter 1 Using the MAC III Microcomputer

Objectives of this Chapter

Having studied this chapter you will be able to:

- Connect the MAC III Microcomputer, Keypad/display unit and Applications Module.
- Connect power to the MAC III Microcomputer and the Applications Module.
- Run the Applications Module demonstration programs.
- Select each section of the Applications Module demonstration program:
  - Analog to Digital Conversion.
  - Optical Link.
  - Proximity Detector.
  - Distance Measurement.
  - Constant Motor Speed Control.
  - Variable Motor Speed Control.
  - Beam Interruption.
  - Optical Feedback.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- MAC III 6502 User Manual.

1.1 Introduction

This chapter is designed to introduce you to running programs on the MAC III, and to familiarize you with the transducers available on the Applications Module.

- Connect the following items by referring to the MAC III User Manual:

  MAC III 6502 Microcomputer
  Power supply
  Keypad/display unit
  Applications Module

  If you are using a *Digiac 2000* system, refer to the User Manual chapter titled **Digiac 2000 Connections**.

  To connect a *Digiac 3000* system, refer to **Digiac 3000 Connections** chapter of the User Manual.

1.1a The Keypad/display is connected to the MAC III Microcomputer using:

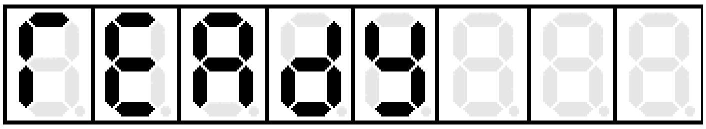
a one 9-wire cable.
b one 5-wire cable.
c one 16-wire cable.
d two 9-wire cables.

1.1b Power is connected to the Applications Module using:

a one cable terminated in a 9-pin connector.
b one cable terminated in a 5-pin connector.
c one cable terminated in a 16-pin connector.
d two cables, each terminated in a 9-pin connector.

- Switch **on** the power supply. If you are not sure how to do this, refer to the MAC III User Manual.

The MAC III display should now show:

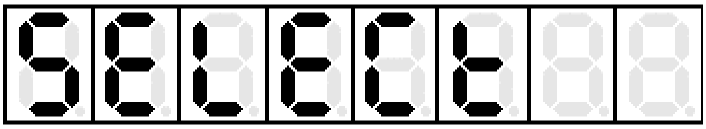


If this does not happen, switch the power off, check the connections and try again.

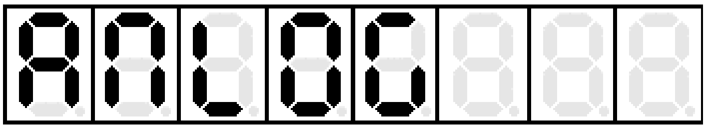
Press G followed by F 6 0 0

Press G again to run the program.

The message “APPLICAtIONS” will move quickly across the display, followed by the word "SELECt" for about one second thus:



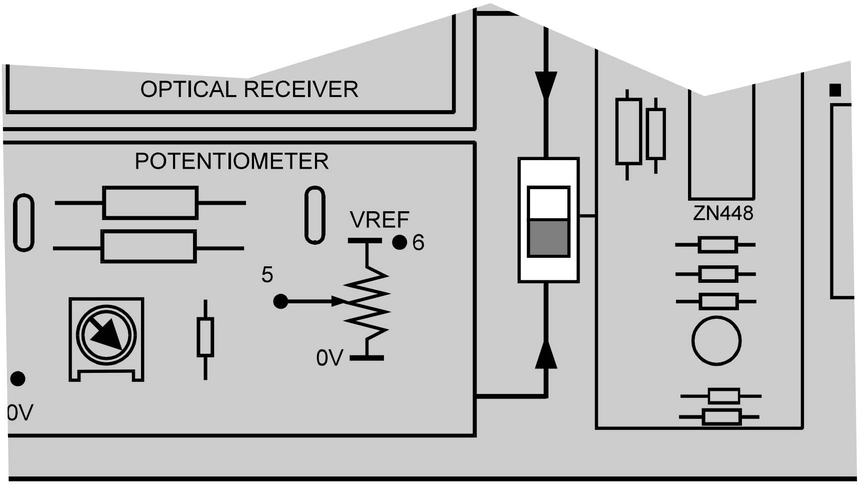
This is followed by the static display:



This indicates that the first of the demonstration programs has been selected. Other demonstration programs can be selected by using the + or - key.

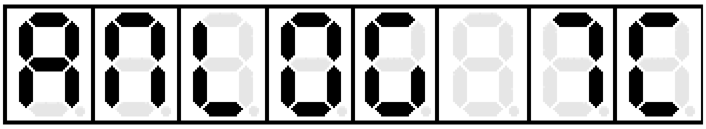
1.2 Analog to Digital Conversion

The Analog to Digital Conversion Demonstration Program will continually sample the potentiometer output, via the ADC and display a hexadecimal value between 00_H and FF_H, depending upon the position of the potentiometer wiper. It is important that the slider switch next to the ADC is set to its **lower** position so that the potentiometer is connected to the ADC.



Potentiometer and ADC Switch

Having set the slider switch the Analog to Digital Conversion Demonstration Program can be executed thus: Use the + or - keys to select "AnLoG" and press the G key once. Adjust the potentiometer over its full range. The display will vary between 00_H and FF_H. A typical display might be:



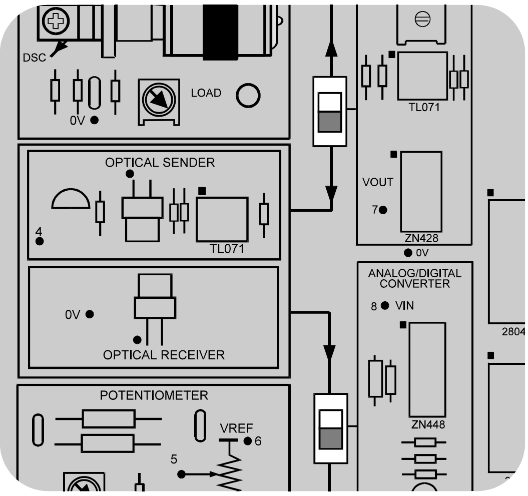
1.2a Turn the potentiometer fully clockwise. Enter the hexadecimal value shown on the display.

If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

The other demonstration programs are explained on the following pages.

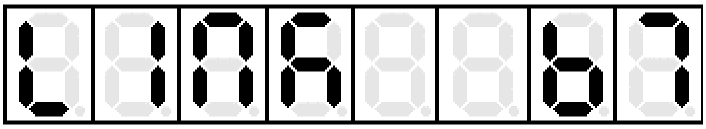
1.3 Optical Link

This demonstration program will continually sample the potentiometer output, via the ADC and then output the current value to the DAC. This analog output is then passed to the Optical Sender LED. The hexadecimal value output will also be displayed. Both the ADC and DAC slider switches should be set to their **lower** positions.



Optical Sensors

Use the + or - key to select “LInK” and press the G key. Adjust the potentiometer over its full range. The display will vary between 00_H and FF_H. A typical display might be:



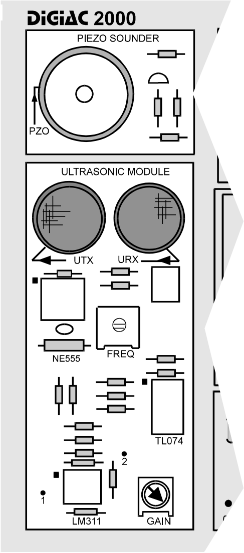
Note that the brightness of the optical sender LED will also vary correspondingly. The LEDs D0 to D7 show the data output from the MAC III to the optical sender. This is the binary equivalent of the hexadecimal value on the MAC III display.

1.3a Turn the potentiometer fully counter-clockwise. Enter the hexadecimal value shown on the display.

If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

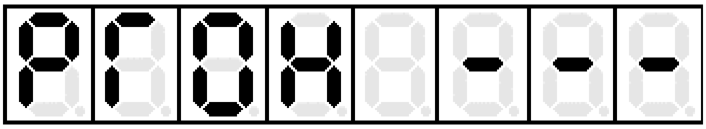
1.4 Proximity Detector

This demonstration program uses the ultrasonic transmitter and receiver as a proximity detector. The piezo sounder functions as an alarm and the display changes as an object is detected. The sensitivity of the detector can be adjusted using the “gain” control in the Ultrasonic Module block.

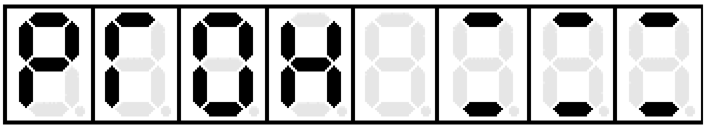


Ultrasonic Module and Piezo Sounder

Use the + or - key to select “ProX” and press the G key. Adjust the gain control **clockwise** until the alarm sounds, then turn it counter-clockwise until the alarm is just switched off. The display will read:



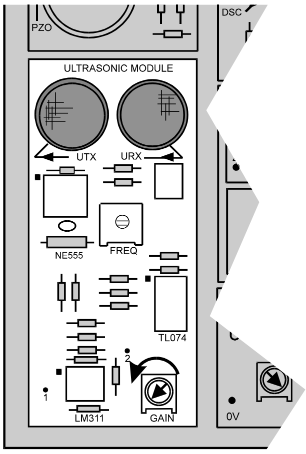
An object placed directly above the ultrasonic receiver and transmitter will be detected up to a distance of approximately 20 centimeters. When an object is detected, the alarm will sound and the display changes to:



If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

1.5 Distance Measurement

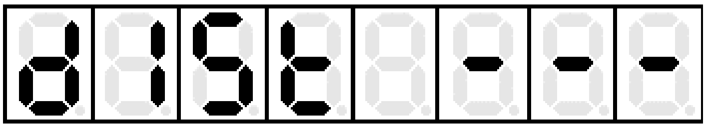
This demonstration program uses the ultrasonic transmitter and receiver to measure the distance of an object above the board. The program calculates the distance by measuring the time delay between the transmission of an ultrasonic pulse and its reflection being received. The “gain” control should initially be set fully **counter-clockwise**.



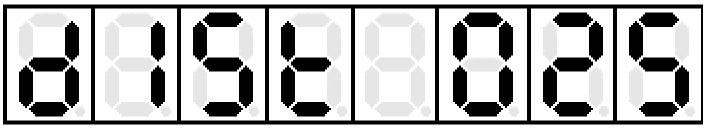
Ultrasonic Module with gain control fully counter-clockwise

Use the + or - key to select “dISt” and press the G key.

Initially, the display should show:



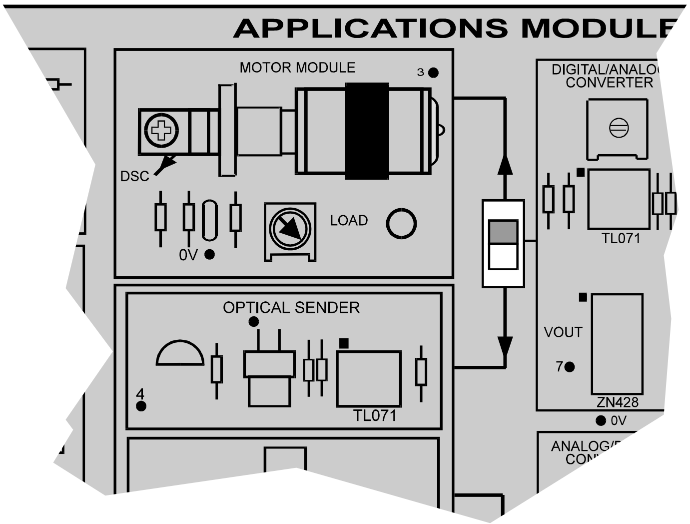
Turn the gain control clockwise from the fully counter-clockwise position until the display shows '000', then turn the control slowly counter-clockwise until '---' is once again displayed. The display will now show the height of an object above the board (in centimeters). For example:



If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

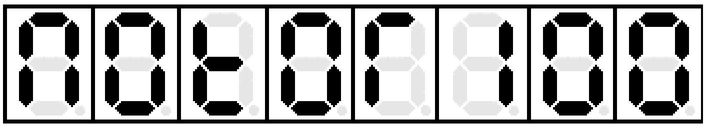
1.6 Constant Motor Speed Control

This demonstration program will cause the motor to run at a constant speed of 100 revolutions per second (rps). The motor "load" control can be used to vary the motor load. The program will compensate for these variations in load by changing the value sent to the DAC. This will allow the speed to be maintained at a constant 100 rps. The LED's D0 to D7 display the data being sent from the MAC III Microcomputer to the DAC. The DAC slider switch should be set to its **upper** position.



DAC Switch

Use the + or - key to select "Motor" and press the G key. Use the motor "load" control to vary the loading on the motor. Notice that the speed is kept constant at 100 rps although the input to the DAC (as indicated by D0 to D7) varies as the program compensates for load variations.

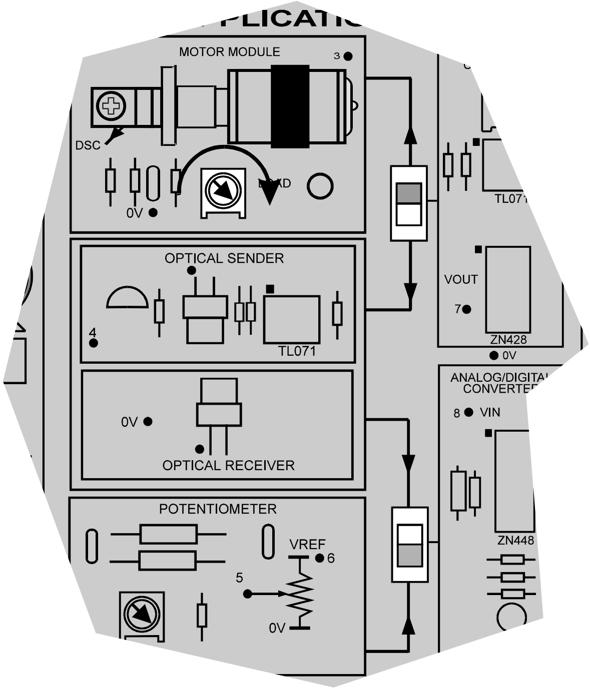


1.6a **With the constant speed control program running, turn the "LOAD" control fully clockwise. Wait for 5 seconds and then enter the motor speed value shown on the display.**

If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

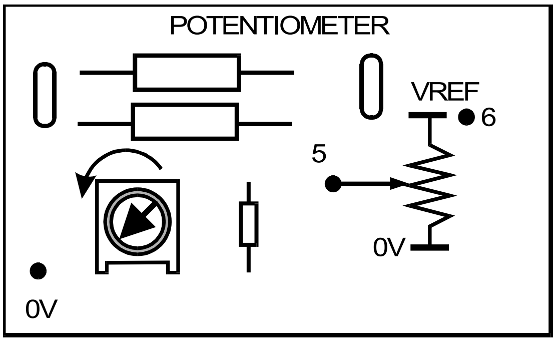
1.7 Variable Motor Speed Control

This demonstration program will cause the motor to run at a desired set speed, depending upon the setting of the potentiometer. The DAC slider switch should be set to its **upper** position and the ADC slider switch to its **lower** position. Also the 'Load' control in the Motor Module block should be turned to the **fully clockwise** (maximum load) position.



Correct positions of 'LOAD' control and ADC/DAC switches

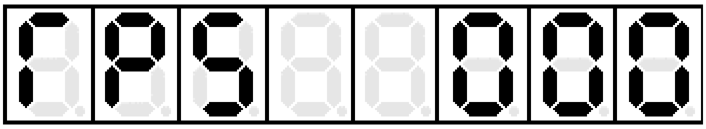
Turn the potentiometer fully counter-clockwise, as shown below.



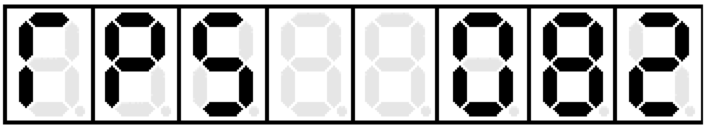
Potentiometer fully counter-clockwise

Use the + or - key to select “rPS” and press the G key.

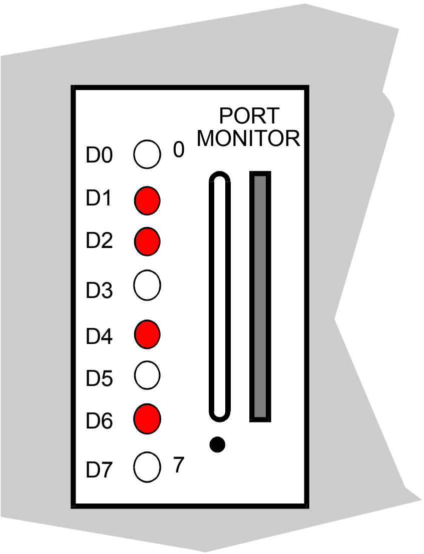
The display will show “000” and the motor will **not** rotate.



Gradually turn the potentiometer clockwise and the motor will rotate at the speed set by the potentiometer position.



The LED’s D0 to D7 show the data output from the MAC III to the DAC.



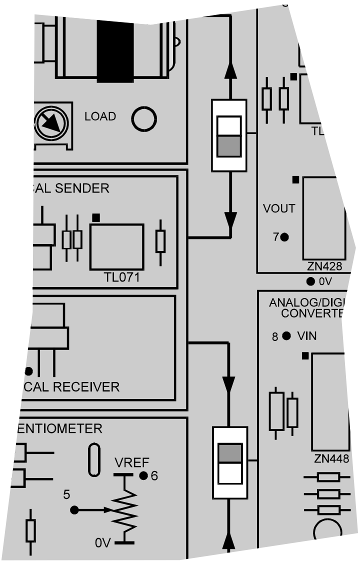
Port Monitor

1.7a **With the speed control program running, turn the potentiometer fully clockwise. Wait for 5 seconds and then enter the motor speed value shown on the display.**

If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

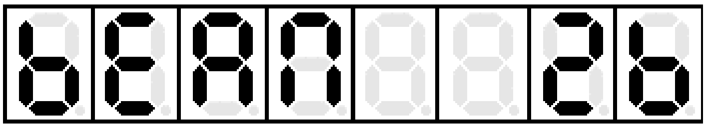
1.8 Beam Interruption

In this demonstration program the MAC III uses the DAC to fully turn on the optical sender LED. The optical receiver output is returned, via the ADC, to the MAC III. A hexadecimal value is displayed by the MAC III to indicate the intensity of light falling upon the optical receiver. The program will compare this hexadecimal value with the arbitrary value 15_H and use the piezo sounder as an alarm signal if the light level falls below this value. The DAC slider switch should be set to the **lower** position and the ADC slider switch to the **upper** position.

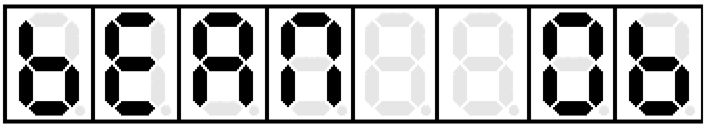


Switch Positions

Use the + or - key to select “bEAM” and press the G key. The display might typically look like the one below, although the number displayed may be different:



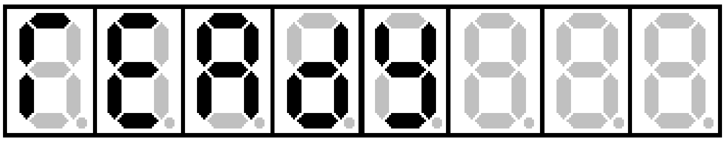
The alarm will sound if the optical link is broken (for example, by placing a piece of paper between the sender and receiver). The light intensity is displayed by the MAC III as a hexadecimal value.


bEAn 0b

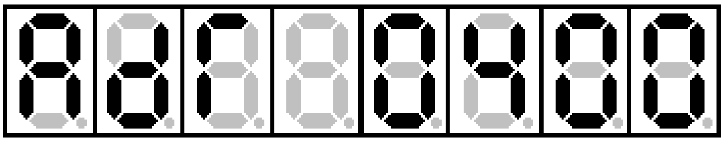
If the ambient lighting level is high, the light level at the receiver may exceed the threshold, even when the sender is blocked off. Fortunately the demonstration program allows the user to change the threshold level from its initial value of 15_H.

The procedure is as follows:

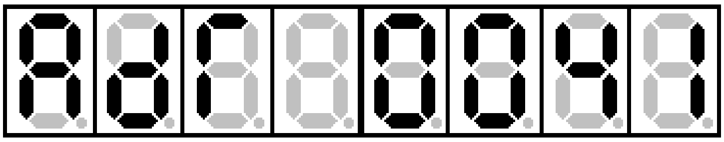
- Press and release the RESET key on the MAC III main board. The display will show:



- Press $\boxed{M}$ and the display will show:

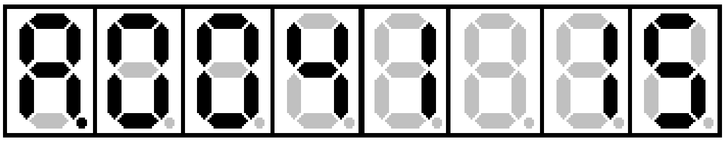


- Use the Hexadecimal Keypad to change the display to



by pressing the following keys in sequence: $\boxed{0}$ $\boxed{0}$ $\boxed{4}$ $\boxed{1}$

- Press $\boxed{M}$ again and the display will show:



The last two digits are the threshold value 15_H.

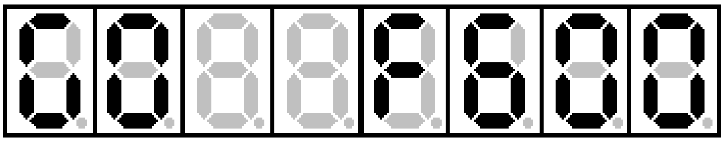
- Use the Hexadecimal Keypad to change the threshold value (15_H) to the desired level (higher or lower).

For example, to make the threshold value 25_H: Press $\boxed{2}$ $\boxed{5}$.

Similarly, to make the threshold value 50_H: Press $\boxed{5}$ $\boxed{0}$.

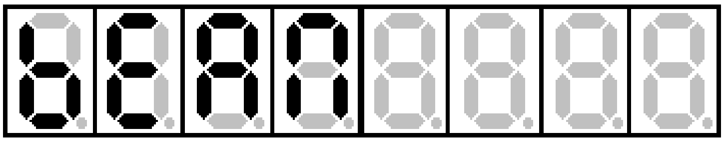
- To run the program again press G again followed by:

  F 6 0 0 and the display will show:



Press G once more to run the Applications Demonstration program

Use the + or - keys to select



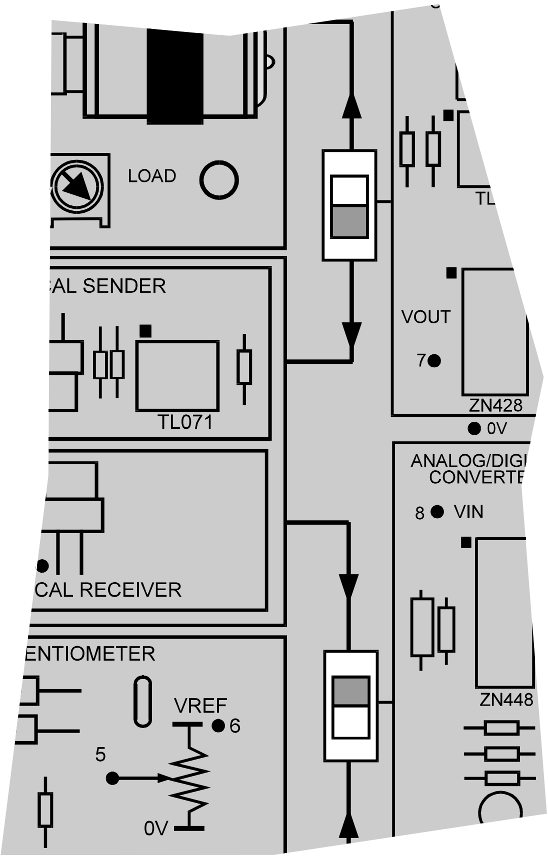
and press the G key again to run the program with the modified threshold value.

The significance of this procedure will be explained in subsequent chapters.

If the G key is held down then released, this program is halted. Another demonstration program can be selected, using the + and - keys.

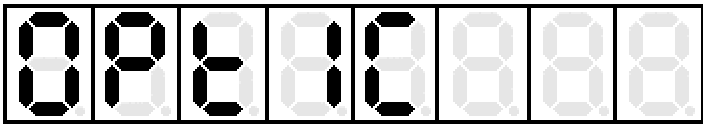
1.9 Optical Feedback

This demonstration program will use the optical sender LED to maintain the light level at the receiver at a preset value, under conditions of varying ambient lighting. The ADC slider switch should be set to the **upper** position and the DAC slider switch to the **lower** position.



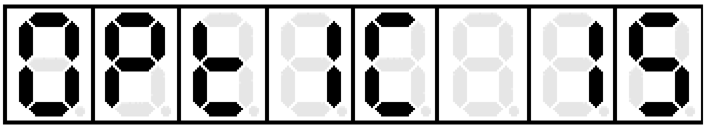
Switch positions

Use the $\boxed{+}$ or $\boxed{-}$ key to select:



and press the $\boxed{G}$ key.

The display will show the current light level at the receiver as a hexadecimal value. This will gradually increase or decrease to 15_H (the preset level to be maintained).



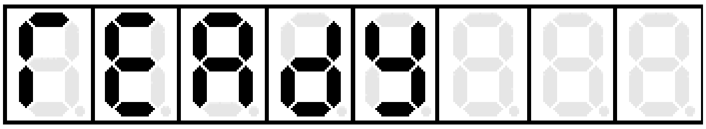
If the sender and receiver are covered so that the ambient light level falls, the program will increase the brightness of the sender LED to compensate and return the received value to 15_H. Conversely, if a bright light source is brought close to the sender and receiver, the brightness of the sender LED is reduced to return the received value to 15_H.

1.9a **With the optical feedback program running, place a piece of thin card or paper between the optical sender and the receiver. Enter the light intensity value shown on the display.**

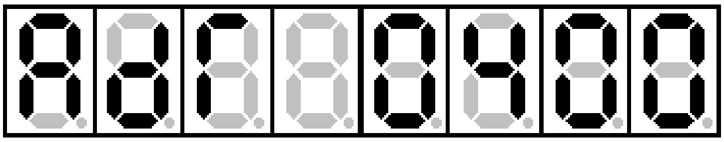
1.9b **With the optical feedback program running, remove any thin card or paper between the optical sender and the receiver. Enter the light intensity value shown on the display.**

The user can adjust the preset level in a similar way to the broken beam detector program:

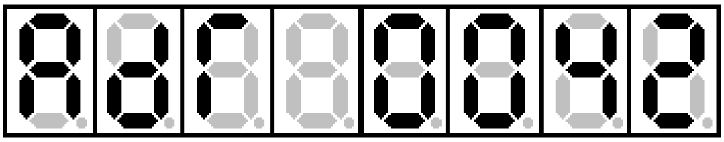
- Press and release the RESET key on the MAC III board. The display will show:



- Press M and the display will show:

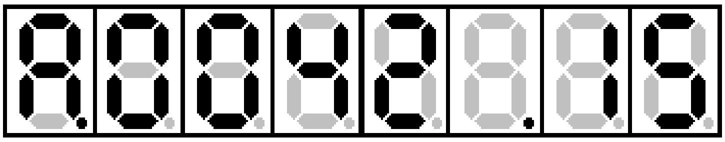


- Use the Hexadecimal Keypad to change the display to



thus: $\boxed{0}$ $\boxed{0}$ $\boxed{4}$ $\boxed{2}$

- Press $\boxed{M}$ again and the display will show:



This is the preset value 15_H.

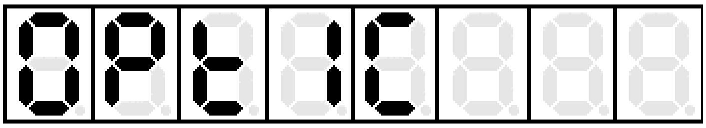
- Use the Hexadecimal Keypad to change the preset value (15_H) to the desired level.

For example, to make the threshold level 10_H: Press $\boxed{1}$ $\boxed{0}$.

Similarly, to make the threshold level 35_H: Press $\boxed{3}$ $\boxed{5}$.

- To run the program again press $\boxed{G}$ followed by $\boxed{F}$ $\boxed{6}$ $\boxed{0}$ $\boxed{0}$ and then press $\boxed{G}$ again.

- Use the $\boxed{+}$ or $\boxed{-}$ keys to select



and press $\boxed{G}$ once more to run the program with the modified threshold value.

If the G key is held down then released, this program is halted.

Another demonstration program can be selected, using the + and - keys.

Turn off the power supply before continuing to the next chapter.

Student Assessment 1

1. The MAC III Microcomputer is connected to the Applications Module using:

a one 9-wire cable.

b one 5-wire cable.

c one 16-wire cable.

d two 9-wire cables.

2. The Applications Module power cable is connected in the:

a bottom right hand corner of the Applications Module.

b bottom left hand corner of the Applications Module.

c top right hand corner of the Applications Module.

d top left hand corner of the Applications Module.

3. When power is applied to the MAC III Microcomputer, the display shows:

a '0400'.

b 'rEAdy'.

c 'SELECt'.

d 'WAItIng'.

4. The Applications Module demonstration program is executed by pressing:

a F 6 0 0 G

b G F 6 0 0 G

c 0 0 1 5 G

d G 0 0 1 5 G

Student Assessment 1 Continued ...

5. When the Applications Module demonstration program is run, the display sequence is:

a "APPLICAtIONS", "SELECt", then "AnLOG".

b "SELECt", "APPLICAtIONS", then "AnLOG".

c "RUNNING", "APPLICAtIONS", "SELECt", then "AnLOG".

d "RUNNING", "SELECt", "APPLICAtIONS", then "AnLOG".

6. The keys which are used to select different sections of the Applications Module demonstration software are:

a + and -

b G and R

c L and S

d M and P

7. When the Variable Motor Speed Control section of the Applications Module demonstration software is selected, the display will show:

a "LInk".

b "mOtOr".

c "PrOH".

d "rPS".

Chapter 2 Introduction to 6502 Programming

Objectives of this Chapter

Having studied this chapter you will be able to:

- Examine and modify the contents of MAC III memory.
- Explain the need for:
  - machine language.
  - machine code.
  - assembly language.
- Interpret the MAC III Memory Map.
- Use the keypad to key in and execute a machine code program.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- MAC III 6502 User Manual.

Introduction

- Connect the following items by referring to the MAC III User Manual:

  MAC III 6502 Microcomputer
  Power supply
  Keypad/display unit

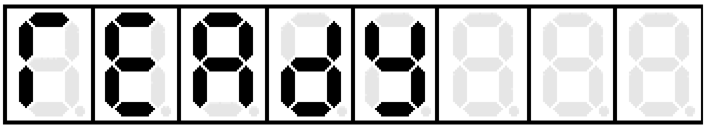
  If you are using a *Digiac 2000* system, refer to the User Manual section titled **Digiac 2000 Connections**.

  To connect a *Digiac 3000* system, refer to **Digiac 3000 Connections** section of the User Manual.

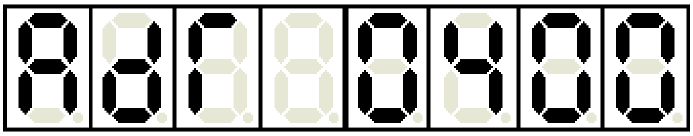
  Note that the Applications Module will not be required initially.

2.1 MAC III Memory

Switch the power **on**. The MAC III display will show:

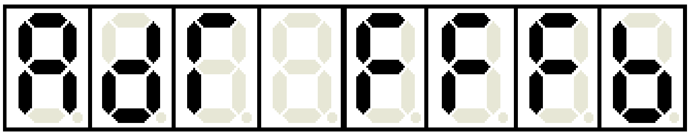


Press M and the display will show:



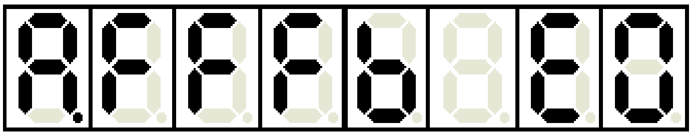
This means that location 0400_H is currently selected. Pressing any of the hexadecimal keys will change the currently selected memory location.

Press F F F B and the display shows:

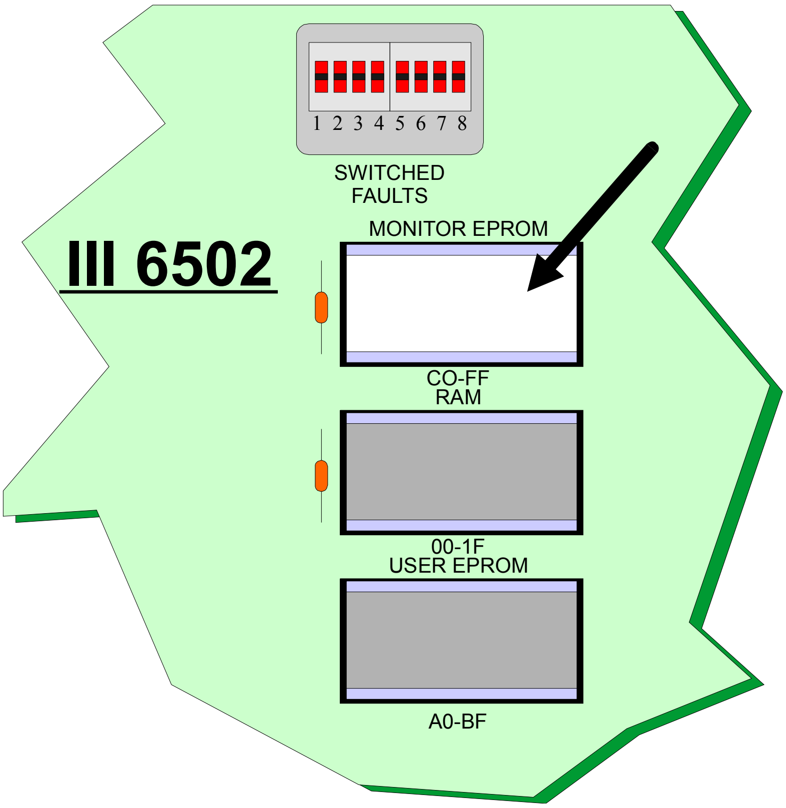


This means that location $FFFB_H$ has been selected.

Press M again and the display will show:



This indicates that the **contents** of location $FFFB_H$ are $E0_H$. This is actually a location within the Monitor EPROM and cannot be altered by the user.



MAC III board with Monitor EPROM arrowed.

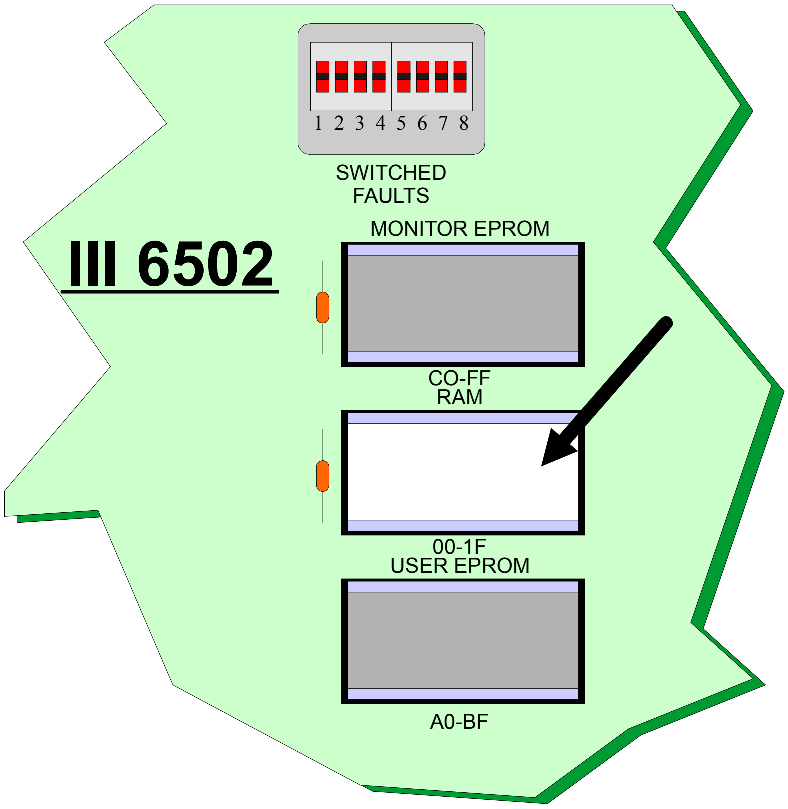
The $\boxed{+}$ and $\boxed{-}$ keys can be used to select the next or previous location respectively.

Use the $\boxed{+}$ key to step forward through a few locations. Notice that the contents will usually be different in each location. If the $\boxed{+}$ or $\boxed{-}$ key is **held** down, the function **repeats** until the key is released.

Switch off the power for a few seconds and then switch on again. Examine memory location $FFFB_H$ again. Notice that the contents of this location have **not changed** (they are still $E0_H$). Recall that ROM is **non-volatile**.

Press $\boxed{M}$ and use the hexadecimal keys to select location 0500_H. Use the $\boxed{M}$ key to discover the contents of this location. If the MAC III has just been switched on, the contents of 0500_H will probably be FF_H.

This location is within the RAM IC and so may be altered by the user as desired.



MAC III board with RAM arrowed.

The hexadecimal keys may now be used to change the contents of location 0500_H. Experiment with changing the contents of this and other RAM locations. Notice that if a location lies outside user RAM, the fourth decimal point display is lit as a warning.

Use the hexadecimal keypad to change the contents of location 0500_H to AB_H.

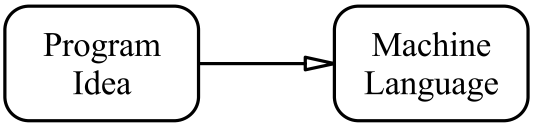
Switch off the power for a few seconds and then switch on again. Examine memory location 0500_H again. Notice that the contents of this location have **changed**. Recall that RAM is **volatile** and so its contents are **lost** when the power is switched off.

2.1a **Enter the hexadecimal contents of the MAC III memory location $FFFD_H$.**

2.2 Programming Levels

The microprocessor is only capable of interpreting data which is presented in **binary** form. This is called **machine language**. Programs written in machine language will be much more efficient in terms of memory space and execution time than those written in many high level languages (for example, BASIC).

Since machine language programs are written in the microprocessor's own "language" the programmer will require quite detailed knowledge of the microprocessor to be used.



Now, although the microprocessor uses **binary** data, programs written in binary are prone to error in transcription and are very time-consuming to write. **Hexadecimal** provides a convenient substitute, requiring very little in the way of conversion. Almost all microprocessors can be programmed using hexadecimal numbers to represent instructions and data. This type of programming is called **machine code programming**. The MAC III can be programmed in machine code by means of the keypad.

Even machine code is rather difficult for the programmer to remember accurately so it is usual for programs to be written on paper using **mnemonic codes**. These are an easily-remembered system of abbreviations for each microprocessor instruction. Programming using mnemonic codes is referred to as **assembly language** programming.

So, in order to write a program for a microprocessor:

- Assembly language program is written on paper.
- Assembly language program is **coded** into machine code.
- Machine code program is executed by microprocessor.

The **instruction set** is a listing of all the mnemonics and corresponding machine code for a given microprocessor. The instruction set for one type of microprocessor (for example, 6502) will **not** apply to another type of microprocessor (for example, Z80). There are however some exceptions to this rule.

2.2a **An easily-remembered abbreviation used when writing a microprocessor instruction is called a:**

a Binary Code.

b Hexadecimal Code.

c Machine Code.

d Mnemonic Code.

2.2b **Programming using mnemonic codes is called:**

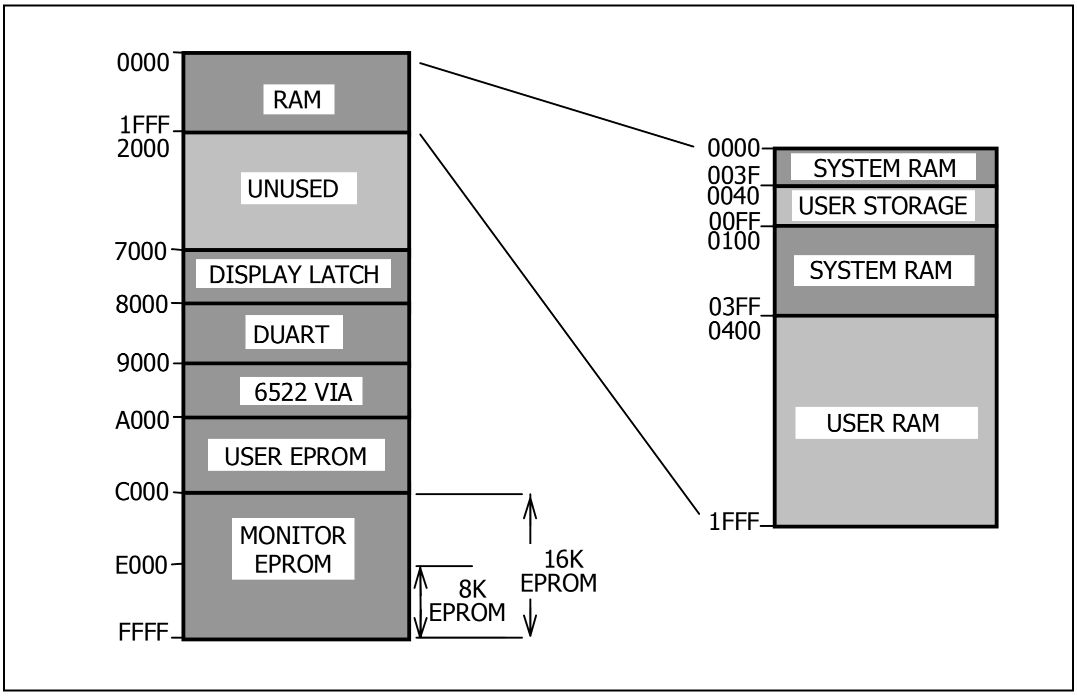
a Assembly Language Programming.

b High Level Language Programming.

c Machine Language Programming.

d Program Language Programming.

Before a program can be entered into a microcomputer system, it will be necessary to know which areas of RAM are available. A **memory map** will show memory usage in a diagrammatic form. The memory map for the MAC III is shown below:



The MAC III Memory Map shows, for example, that the monitor EPROM has an address range from C000_{H} to FFFF_{H} and that the MAC III RAM occupies addresses from 0000_{H} to 1FFF_{H}.

2.2c The function of the section of MAC III Memory that includes location 0800H is:

a Monitor EPROM.

b System RAM.

c User EPROM.

d User RAM.

2.3 Programming the Microprocessor

You will already have used the MAC III microcomputer for the Applications Module demonstration programs.

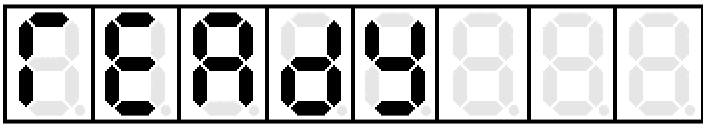
These programs were previously stored in the Monitor EPROM. Now **you** can key in a short program into RAM for the Applications Module.

- Connect the Applications Module by referring to the MAC III User Manual.

  If you are using a *Digiac 2000* system, refer to the User Manual section titled **Digiac 2000 Connections**.

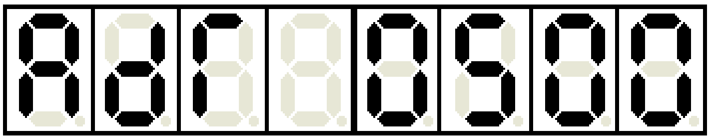
  To connect a *Digiac 3000* system, refer to **Digiac 3000 Connections** section of the User Manual.

Switch the power **on**. The display will show:

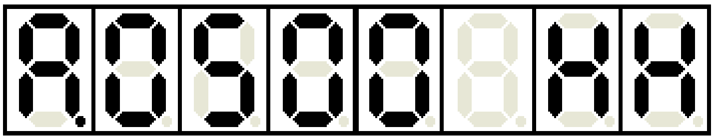


The MAC III may be programmed by placing the correct machine code in successive memory locations. You are now going to key in a machine code program. Select location 0500_H thus:

Press $\boxed{M}$ followed $\boxed{0}$ $\boxed{5}$ $\boxed{0}$ $\boxed{0}$. The display will show :



Press $\boxed{M}$ again and the display will show :



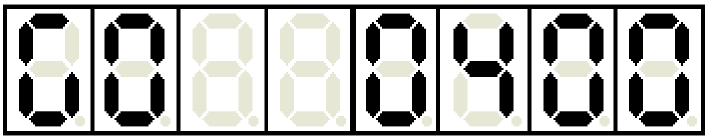
Where “HH” represents the current contents of location 0500_H. This will probably be FF_H if you have just switched on. Change the contents of 0500_H to $A9_H$ by pressing $\boxed{A}$ $\boxed{9}$. Now press the $\boxed{+}$ key to move on to location 0501_H. The whole program listed below can now be entered by repeating the above procedure for each location.

Location	Contents
0500	A9
0501	FF
0502	8D
0503	03
0504	90
0505	AD
0506	00
0507	10
0508	8D
0509	01
050A	90
050B	60
1000	88

It is **not** important at this stage to understand exactly how this program works. In fact, it will display the contents of memory location 1000_H as a binary pattern on the Applications Module Port Monitor (labeled D_0 to D_7).

You are now ready to run this program.

Press the G key once and the display will show:



This is the address from which program execution will begin.

Change this to 0500_H by keying in 0 5 0 0 .

Press the G key once again and the program will run.

The Applications Module Port Monitor should now show:

D7	D6	D5	D4	D3	D2	D1	D0
●	○	○	○	●	○	○	○

● lit
○ unlit

This is 1000 1000_2 (88_H) - the value which was programmed into location 1000_H. If you do not see this output on the port monitor, check the following:

- Has the machine code been correctly entered?
- Is the Applications Module connected to the MAC III?
- Is the power correctly connected to the Applications Module?

Change the value in location 1000_H and run the program again. Experiment with several other values in location 1000_H.

2.3a **Stop the program, change the data at location 1000_H to 72_H and run the program again. The pattern shown on the Applications Module Port Monitor LEDs (● = lit, ○ = unlit) is:**

☐ a

D7	D6	D5	D4	D3	D2	D1	D0
○	●	○	●	○	●	○	●

☐ b

D7	D6	D5	D4	D3	D2	D1	D0
○	○	●	○	○	●	●	●

☐ c

D7	D6	D5	D4	D3	D2	D1	D0
○	●	●	●	○	○	●	○

☐ d

D7	D6	D5	D4	D3	D2	D1	D0
○	●	●	●	●	●	○	●

Student Assessment 2

1. The data word at MAC III memory address E0DC_H is:

a 60_H

b 6C_H

c DC_H

d E0_H

2. The keystrokes required to change the contents of location 0407_H to B2_H are:

a B 2 M 0 4 0 7

b M B 2 M 0 4 0 7

c 0 4 0 7 M B 2

d M 0 4 0 7 M B 2

3. The form in which machine language is presented to the microprocessor is:

a Binary.

b Octal.

c Decimal.

d Hexadecimal.

4. Giving instructions to the microcomputer in hexadecimal form is called:

a Assembly Language Programming.

b Coding.

c High Level Programming.

d Machine Code Programming.

Continued ...

Student Assessment 2 Continued ...

5. **Programming using mnemonic codes is called:**
   - a Assembly Language Programming.
   - b Coding.
   - c High Level Programming.
   - d Machine Code Programming.

6. **The area of MAC III memory available for User Programs is:**
   - a 0000_H to $003F_H$
   - b 0100_H to $03FF_H$
   - c 0400_H to $1FFF_H$
   - d 2000_H to $6FFF_H$

7. **The function of the MAC III memory area $A000_H$ to $BFFF_H$ is:**
   - a RAM.
   - b Monitor EPROM.
   - c System RAM.
   - d User EPROM.

8. **The key used to enter the memory examination mode is:**
   - a $\boxed{+}$
   - b $\boxed{-}$
   - c $\boxed{L}$
   - d $\boxed{M}$

Student Assessment 2 Continued ...

9. The keystrokes required to run the program which starts at location 1000_H are:

$\boxed{a}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$ $\boxed{0}$ $\boxed{G}$

$\boxed{b}$ $\boxed{G}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$ $\boxed{0}$

$\boxed{c}$ $\boxed{G}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$ $\boxed{0}$ $\boxed{G}$

$\boxed{d}$ $\boxed{L}$ $\boxed{1}$ $\boxed{0}$ $\boxed{0}$ $\boxed{0}$ $\boxed{G}$

Chapter 3 Writing Machine Code Programs

Objectives of this Chapter

Having studied this chapter you will be able to:

- Explain the action of fundamental microprocessor instructions:
  - Load
  - Add
  - Decrement
  - Increment
  - Jump
- Describe the functions of operators and operands.
- Code an Assembly Language program.
- Write simple Assembly Language programs.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- 6502 Instruction Set Reference Manual.
- MAC III 6502 User Manual.

Introduction

The 6502 has a number of special and general purpose registers. All 6502 working registers are 8 bits (byte length). For the time being we shall only concern ourselves with the **Accumulator.**

The **Accumulator** is the primary CPU register. Most arithmetic and logical operations take data from the accumulator. The result of such operations is then returned to the accumulator.

3.1 Instruction Sets

Although instruction sets differ between manufacturers, certain fundamental types of instruction are common to almost all microprocessors:

Load	This will **duplicate** the contents of a memory location within the Accumulator.
Store	This will **duplicate** the contents of the Accumulator within a memory location.
Add	This will **add** the contents of one general purpose register or memory location with those of another general purpose register or memory location and place the result in the accumulator.
Decrement	This will **subtract one** from the contents of a specified register or memory location.
Increment	This will **add one** to the contents of a specified register or memory location.
Jump	This will **always** cause program execution to continue from a specified location **other than** the next location in sequence.

3.1a Enter the number of bits within the 6502 Accumulator.

3.1b A memory location initially contains the value 45_H. Enter the hexadecimal contents of this location after a 'Decrement' instruction has been executed.

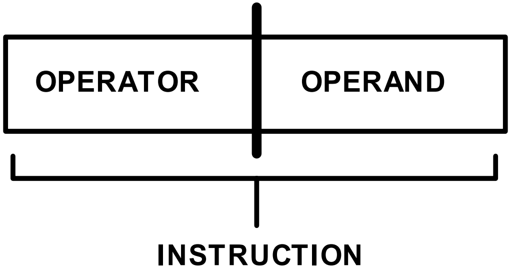
3.2 Instruction Mnemonics

It can be rather time-consuming and tedious to write each of these instructions out in full each time that they are required. Consequently, instructions are generally abbreviated to a 3-letter **mnemonic**. Some 6502 instruction mnemonics are given below:

Instruction	Mnemonic
Load	LDA
Add	ADC
Decrement	DEC
Increment	INC
Jump	JMP

3.3 Operators and Operands

Microprocessor instructions can be thought of as consisting of **two** distinct parts:



Operator

This is the part of an instruction that defines the operation which must take place. For example “Load”.

Operand

This provides any additional information necessary for the microprocessor to complete the instruction. For example if the operator is “Load...” then the operand might be “...the accumulator with the hexadecimal value $3B_H$”. Then the overall instruction would be: “Load the accumulator with the hexadecimal value $3B_H$”. Some instructions do **not** require an operand.

3.4 Simple Programs

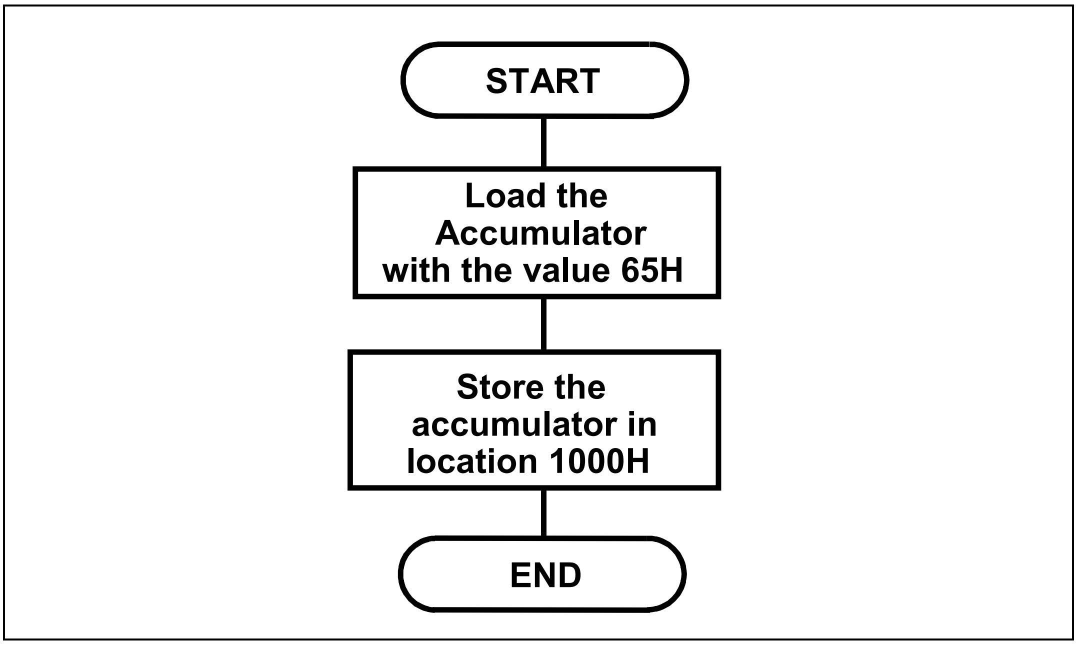
As mentioned in the previous chapter, the microprocessor can only interpret instructions given in binary form. These binary instruction codes are referred to as **opcodes**. The instruction set will include the opcode for every instruction. A list of instruction codes for the 6502 microprocessor is given in the 6502 Instruction Set Reference Manual. For convenience, these codes are expressed in hexadecimal form.

3.5 Worked Example

Write a program that will place the value 65_H in memory location 1000_H.

Solution:

Although this is a very simple program, it is good practice to first draw a flowchart:



The assembly language program will be:

```
LDA  #$65
STA  $1000
RTS
```

This program will be much easier to understand if **comments** are added. It is conventional for comments to be prefixed by a semi-colon thus:

```
LDA   #$65   ;Loads accumulator with 65H
STA   $1000  ;Saves the contents of the
             ;accumulator in location 1000H
RTS          ;Returns to the MAC III system
```

The last instruction (`RTS`) will return control to the MAC III monitor program after execution of a user program. The precise nature of this instruction is unimportant for the time being.

It is now necessary to look-up the opcodes or **code** the program. The opcodes will be found in the 6502 Instruction Set Reference Manual.

Take the first instruction (`LDA #$65`):

Turn to the 6502 instruction set and find the “load accumulator" instruction (`LDA`). The addressing mode here is **immediate**. More information concerning 6502 Addressing Modes is given in a later chapter. Notice from the instruction set that the correct opcode for `LDA` is **A9$_{\mathbf{H}}$.** Now, this is the **operator**. The CPU will interpret this code as "Load the accumulator with the hexadecimal value found in the **next** byte of memory". Clearly then, the following byte of memory must take the value **65$_{\mathbf{H}}$**. This is the **operand.**

The first instruction is now coded. The codes are usually written thus:

Machine Code	**Assembly Lang.**	**Comments**
`A9 65`	`LDA #$65`	`;Loads accumulator with` `;65H`
	`STA $1000`	`;Saves accumulator contents` `;in location 1000H`
	`RTS`	`;Returns to the MAC III system`

The next instruction is "store the contents of the accumulator in location 1000_H". This can now be coded, again by reference to the 6502 Instruction Set.

This time the required mnemonic is `STA`. You will again find this in the 6502 Instruction Set Reference Manual. Here the required addressing mode is **absolute**. Again, the topic of addressing modes will be studied in a subsequent chapter. The correct opcode for an absolute `STA` is $\mathbf{8D_H}$. Now, the CPU will interpret this code as "Save the contents of the Accumulator in the memory location specified by the **next two bytes** of memory". Clearly then this instruction will also require an operand but here it will be an **address** rather than **data**. The required address is 1000_H.

The 6502 expects address operands to be placed in memory **low byte first,** so this instruction can now be coded thus:

Machine Code	Assembly Lang.	Comments
`A9 65`	`LDA #$65`	`;Loads accumulator with` `;65H`
`8D 00 10`	`STA $1000`	`;Saves accumulator contents` `;in location 1000H`
	`RTS`	`;Returns to the MAC III system`

You will notice that in 6502 Assembly Language, a **hexadecimal** operand value is indicated by a dollar ($) symbol immediately **before** the value.

The last instruction may now be coded, again by reference to the 6502 Instruction Set Reference Manual. Find the "Return From Subroutine" (`RTS`) instruction. The topic of subroutines will be covered in one of the subsequent chapters. From the Instruction Set you should find that the correct opcode for `RTS` is $\mathbf{60_H}$. Notice that there is no choice of addressing modes in this case.

The coding is now complete:

Machine Code	Assembly Lang.	Comments
A9 65	LDA #$65	;Loads accumulator with ;65H
8D 00 10	STA $1000	;Saves accumulator contents ;in location 1000H
60	RTS	;Returns to the MAC III system

All that is required now is to specify the memory locations which this program will occupy. Anywhere in user RAM may be chosen. For example: starting at 0400_H:

Address	Machine Code	Assembly Lang.	Comments
0400	A9 65	LDA #$65	;Loads accumulator ;with 65H
0402	8D 00 10	STA $1000	;Saves the contents ;of accumulator in ;location 1000H
0405	60	RTS	;Returns to the ;MAC III System

This type of layout is a widely accepted convention. However, it may be modified slightly. A common variation is to explicitly state the contents of each location (as shown below).

This method of laying out programs is probably easier to understand in the initial stages of learning machine code programming.

Address	Machine Code	Assembly Lang.	Comments
0400	A9	LDA #$65	;Loads accumulator
0401	65		;with 65H
0402	8D	STA $1000	;Saves the contents
0403	00		;of accumulator in
0404	10		;location 1000H
0405	60	RTS	;Returns to the
			;MAC III System

Whichever convention you choose to adopt, it is recommended that you always write your programs under the **headings** shown in the previous tables. However, in order to save space in this manual, these headings will not be shown in subsequent program listings.

A Standard Programming Sheet is given in Appendix 1. This may be photocopied for use in writing machine code programs. Notice that there is an extra column marked "Label". It is useful in longer programs and particularly in those with loop structures, to "label" certain locations. This technique will be explained at a later stage.

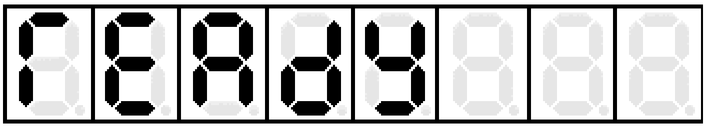
Having written the program on paper, it will be necessary to key it into the microcomputer.

3.5a Enter the hexadecimal byte that must be placed in location 0404_H.

3.5b In the instruction " `LDA #$65` "**, the operand is:**

- a `LDA`
- b `#$65`
- c 0400_H
- d 1000_H

Now enter the program into the MAC III, using the M and hexadecimal keys. Run this program, using the G and hexadecimal keys. Remember that the start address is 0400_H. Having run this program, the display should show:



Use the M and hexadecimal keys to read the contents of location 1000_H. The contents of this location should be **65_H** after the program has been executed. If this does not happen, check that the machine code been correctly entered.

If the correct machine code has not been entered, repeat the procedure, paying particular attention to the required keystrokes.

3.5c **The program in Worked Example 3.5 is to be modified so that the value 88_H is placed in location 1000_H. The memory location that must be changed is:**

a 0400_H

b 0401_H

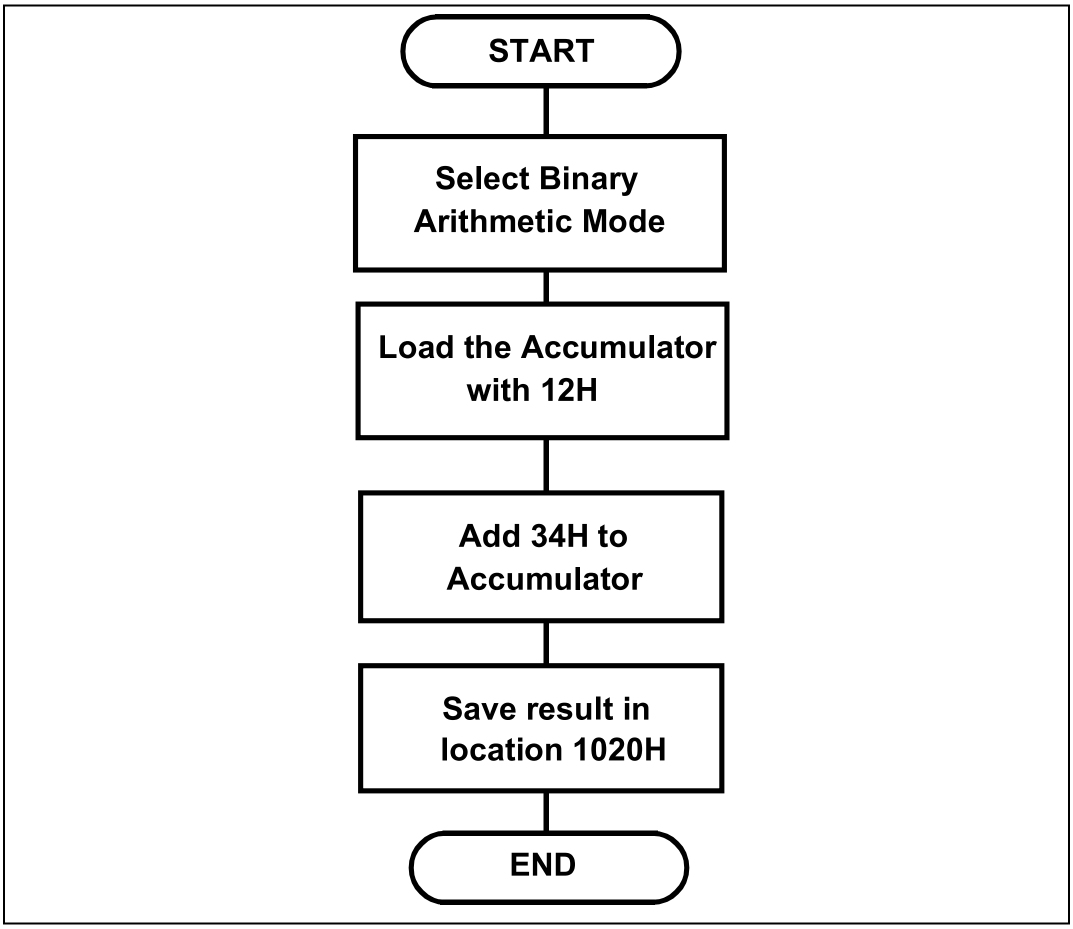
c 0402_H

d 0403_H

3.6 Worked Example

Write a program, starting at location 0600_H, which will add the values 12_H and 34_H and then save the result in location 1020_H.

Solution:



The 6502 Microprocessor is capable of performing addition in two ways, or modes. These are **binary mode** and **decimal mode**.

In binary mode, two binary numbers are added to give a binary result. Binary mode is the usual arithmetic mode for the 6502, and is the mode that will normally be used in this manual.

The other arithmetic mode, decimal mode, will be explained in Chapter 7.

Before the 6502 performs an addition, an instruction is required to select the required arithmetic mode. For binary mode, the instruction is:

```
CLD          ;Selects binary arithmetic mode
```

This instruction will be explained in more detail in Chapter 7. For now, you just need to remember to include it at the beginning of any program that performs binary addition or subtraction.

We will now consider the instructions required to perform the addition itself.

The 6502 instruction set will only allow addition to take place between the accumulator and a memory location. Consequently it will be necessary to **place** one value in the accumulator and then **add** the other value to the contents of the accumulator.

The resulting assembly language program will be:

```
CLD          ;Selects binary arithmetic mode

LDA #$12     ;Loads accumulator with 12H

ADC #$34     ;Adds 34H to accumulator

STA $1020    ;Saves accumulator in 1020H

RTS          ;Returns to MAC III System
```

Again, referring to the 6502 Instruction Set Reference Manual for the coding will give:

```
0600   D8   CLD            ;Selects binary arithmetic mode
0601   A9   LDA #$12       ;Loads accumulator with 12H
0602   12
0603   69   ADC #$34       ;Adds 34H to accumulator
0604   34
0605   8D   STA $1020      ;Saves accumulator in 1020H
0606   20
0607   10
0608   60   RTS            ;Returns to MAC III System
```

Having written this program, enter it into the MAC III and execute. Examine the contents of memory location 1020_H after execution of this program. Check that it contains 46_H (i.e. 12_H + 34_H). Now, location 1020_H will probably contain 46_H (i.e. the correct result). However, it may have given the result 47_H. This is because the 6502 Add instruction is actually an Add **With Carry**.

This means that the current state of the **Carry Flag** is added to the result. So, the Carry Flag must be **cleared** prior to addition. We shall examine the Carry Flag in more detail at a later stage. For the time being just remember that the Carry Flag should be cleared before the ADC instruction.

The Carry Flag is cleared by the "Clear the Carry Flag" (CLC) instruction. So this must be inserted into our assembly language program thus:

```
CLD                 ;Selects binary arithmetic mode

LDA #$12            ;Loads accumulator with 12H

CLC                 ;Clears the Carry Flag

ADC #$34            ;Adds 34H to accumulator

STA $1020           ;Saves accumulator in 1020H

RTS                 ;Returns to MAC III System
```

This program can be re-coded thus:

```
0600   D8   CLD                  ;Selects binary arithmetic mode

0601   A9   LDA #$12             ;Loads accumulator with 12H
0602   12

0603   18   CLC                  ;Clears the Carry Flag

0604   69   ADC #$34             ;Adds 34H to accumulator
0605   34

0606   8D   STA $1020            ;Saves accumulator in 1020H
0607   20
0608   10

0609   60   RTS                  ;Returns to MAC III System
```

Modify the program in the MAC III and place a known value in location 1020_H. Run the program and re-examine location 1020_H to verify correct operation.

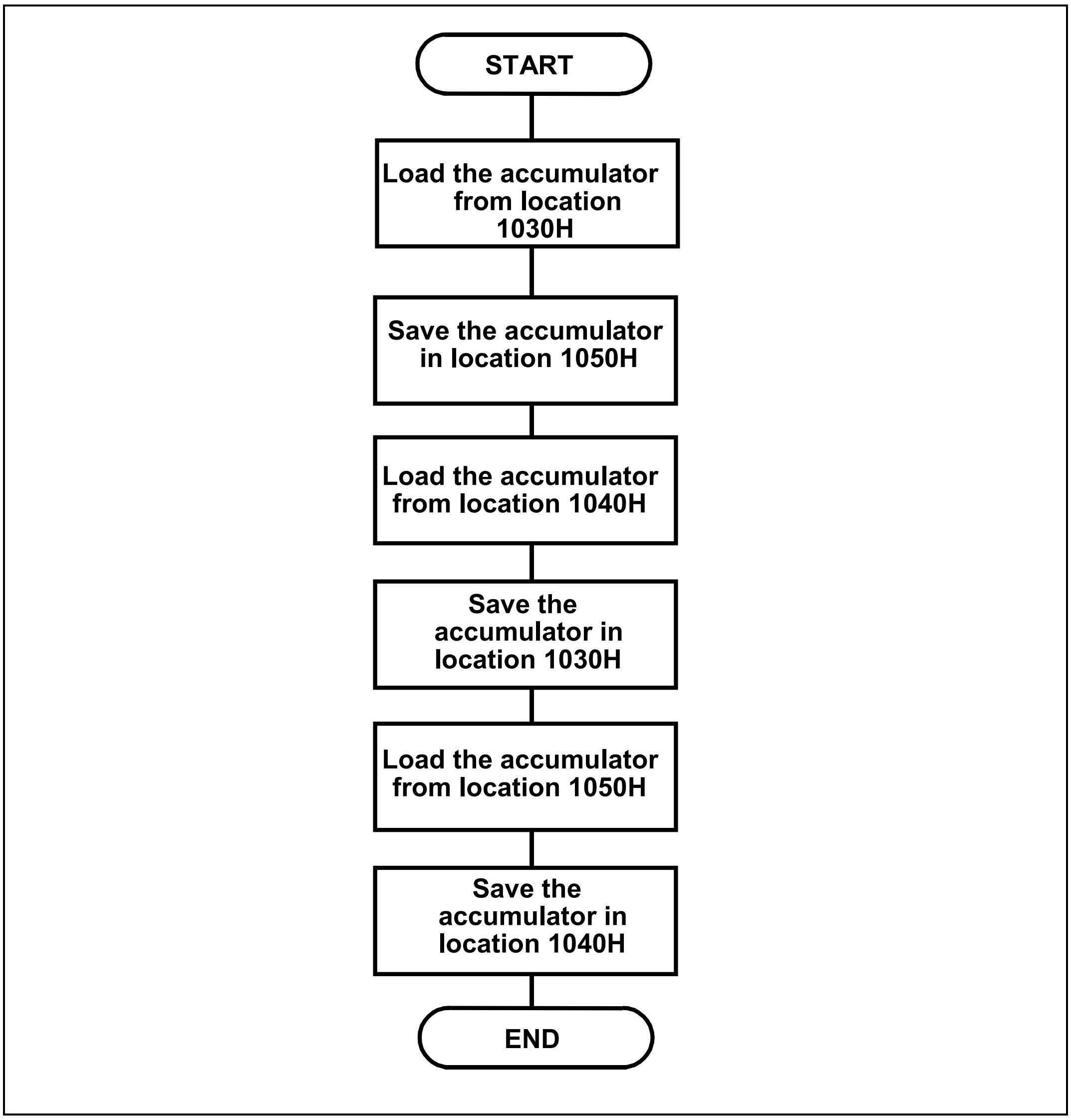
3.6a **The re-coded program in Worked Example 3.6 is to be modified so that the result is saved in location 1040_H. Enter the byte that must be placed in location 0607_H.**

3.7 Worked Example

Write a program, starting at location 0700_H, which will exchange the contents of locations 1030_H and 1040_H.

Solution:

This program will require the use of a **temporary store**. It is convenient to use another memory location, say location 1050_H for this purpose.



The assembly language program will be:

```
LDA $1030    ;Loads accumulator from 1030H
STA $1050    ;Saves accumulator in location 1050H
             ;- Temporary Store
LDA $1040    ;Loads accumulator from location 1040H
STA $1030    ;Saves accumulator in location 1030H
LDA $1050    ;Loads accumulator from 1050H
STA $1040    ;Saves accumulator in location 1040H
RTS          ;Returns to MAC III system
```

This program is coded using the Absolute Addressing modes for LDA and STA thus:

```
0700   AD   LDA $1030    ;Loads accumulator from 1030H
0701   30
0702   10
0703   8D   STA $1050    ;Saves accumulator in location 1050H
0704   50                ;- Temporary Store
0705   10
0706   AD   LDA $1040    ;Loads accumulator from location 1040H
0707   40
0708   10
0709   8D   STA $1030    ;Saves accumulator in location 1030H
070A   30
070B   10
070C   AD   LDA $1050    ;Loads accumulator from 1050H
070D   50
070E   10
070F   8D   STA $1040    ;Saves accumulator in location 1040H
0710   40
0711   10
0712   60   RTS          ;Returns to MAC III system
```

3.7a **Write a program, starting at memory location 0900_H, which will add the hexadecimal values 56_H and 78_H. The result should then be saved in memory location 1060_H. Run your program and then examine the contents of location 1060_H. Enter the byte that you find at this location.**

Student Assessment 3

1. **The primary 6502 Register is:**

   a the Accumulator

   b the Program Counter

   c the X Register

   d the Y Register

2. **The 6502 instruction which copies the Accumulator to a specified memory location is:**

   a Load.

   b Add.

   c Store.

   d Jump.

3. **The 6502 instruction which subtracts one from a specified register or memory location is:**

   a Load.

   b Add.

   c Increment.

   d Decrement.

4. **The function of the "Load" instruction is to:**

   a copy the Accumulator to a specified memory location.

   b copy a specified memory location to the Accumulator.

   c increase the contents of a specified register by one.

   d cause the program to continue from a specified address.

Student Assessment 3 Continued ...

5. The 6502 instruction which allows program execution to continue from some point other than the next location in sequence is:

a Load.

b Add.

c Store.

d Jump.

6. The part of an instruction which provides any additional information necessary to complete that instruction is called the:

a Address.

b Data.

c Operand.

d Operator.

7. The part of an instruction that defines the function to be carried out is called the:

a Address.

b Data.

c Operand.

d Operator.

8. The 6502 Assembly Language mnemonics for "copy the contents of memory location 1100_H into the Accumulator" are:

a `LDA #1100`

b `LDA $1100`

c `LDA #$1100`

d `LDA $#1100`

Continued ...

Student Assessment 3 Continued ...

9. If the carry flag has previously been cleared, the 6502 Assembly Language instruction `"ADC $1200"` **will add:**

- a the value 1200_H to the Accumulator
- b the contents of location 1200_H to the Accumulator
- c the value 1200_H to a specified memory location
- d the contents of location 1200_H to a specified memory location

10. The machine code for the instruction `"DEC $1020"` **is:**

- a CE 10 20
- b CE 20 10
- c DE 10 20
- d DE 20 10

11. The 6502 Assembly Language sequence which will place the hexadecimal value CC_H in location $10B0_H$ is:

a
```
LDA  #$CC
STA  $10B0
```

b
```
LDA  #$CC
STA  $B010
```

c
```
STA  $10B0
LDA  #$CC
```

d
```
STA  $B010
LDA  #$CC
```

Chapter 4 Program Debugging

Objectives of this Chapter

Having studied this chapter you will be able to:

- Explain the need for program debugging.
- Use the MAC III software debugging tools:

  Break Point
  Single Step

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- MAC III 6502 User Manual.

Introduction

Very often in machine code programming, a user-written program will not function correctly when first executed. Such a program will require **debugging** to ensure correct operation. Up to now the only debugging techniques you have used have been simple checks for incorrectly entered machine code or incorrect keystrokes. However, much more sophisticated debugging techniques are provided by many microcomputers.

4.1 Debugging Tools

The MAC III provides two basic software debugging tools which are found in many microcomputer systems:

- Break Point
- Single Step

4.2 Break Point

This allows the program under test to be halted at any desired point and the contents of registers and memory locations examined. Partial program results can thus be inspected.

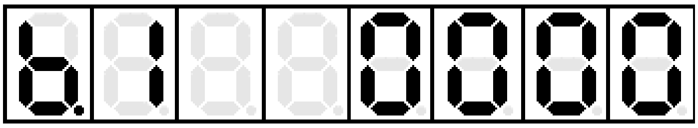
The best way to understand how break points work is to insert a break point into a 6502 program on the MAC III. Break points will only work on programs stored in RAM. This is because a special opcode is inserted at the break point.

Enter the simple program shown below:

```
0400   D8   CLD           ;Selects binary arithmetic mode
0401   A9   LDA #$25      ;Loads accumulator with 25H
0402   25
0403   18   CLC           ;Clears the Carry Flag prior to addition
0404   69   ADC #$35      ;Adds the value 35H to the accumulator
0405   35
0406   8D   STA $0500     ;Saves result in location 0500H
0407   00
0408   05
0409   60   RTS           ;Returns to MAC III System
```

You will now enter a break point at location 0406_H so that the result of the addition can be inspected in the accumulator.

Press the $\boxed{R}$ key **twice**. The display will show:

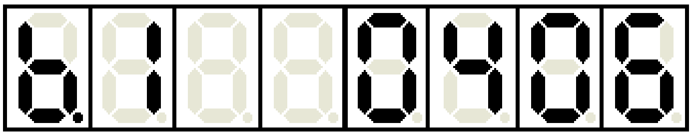


This indicates that break point 1 will occur at location 0000_H. The MAC III monitor allows up to 8 break points to be set or cleared. Other break points can be selected by pressing the $\boxed{+}$ or $\boxed{-}$ keys.

The location at which the break point is to be inserted can now be entered, from the hexadecimal keypad, thus:

Press $\boxed{0}$ $\boxed{4}$ $\boxed{0}$ $\boxed{6}$

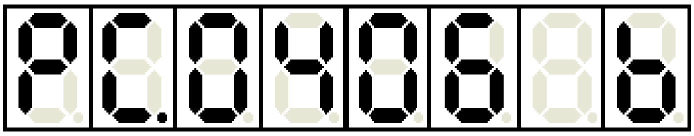
Break point 1 is now set at location 0406_H and the display shows:



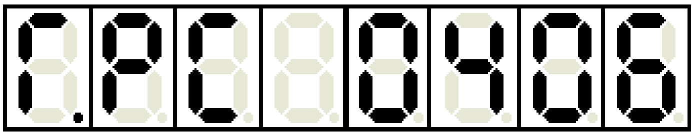
Now run the program in the usual way by pressing:

$\boxed{G}$ $\boxed{0}$ $\boxed{4}$ $\boxed{0}$ $\boxed{0}$

Press the $\boxed{G}$ key again and the program will run but stop at location 0406_H. The display will now show:



This indicates that a break point was found when the Program Counter reached 0406_H. Press $\boxed{R}$ once and the display will show:



This confirms that the program counter has reached location 0406_H. This is the address of the **next** instruction to be executed.

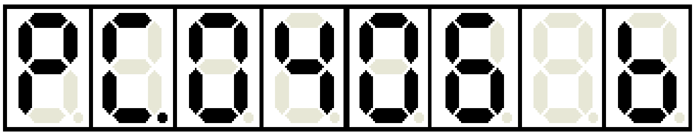
Program execution may be continued from the break point by pressing the G key **twice**.

Now, once program execution has halted at a break point, it is possible to examine the contents of the 6502 registers. This can be a useful aid in debugging programs. Use the R key to check that break point 1 is still set at 0406$_H$.

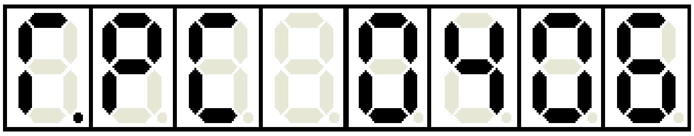
Run the program again by pressing:

G 0 4 0 0 and then by pressing the G key again.

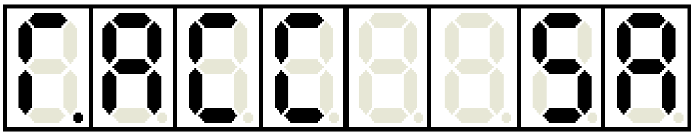
The display will once again show:



Now, press the R key once and the display will show:



This indicates that the Program Counter contents are 0406$_H$. Press the + key once and the display will show:



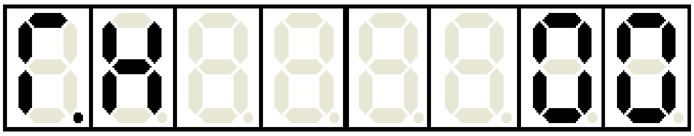
This indicates that the Accumulator holds 5A$_H$.

Now, refer back to the program:

```
0400   D8   CLD          ;Selects binary arithmetic mode

0401   A9   LDA #$25     ;Loads accumulator with 25H
0402   25

0403   18   CLC          ;Clears the Carry Flag prior to addition

0404   69   ADC #$35     ;Adds the value 35H to the accumulator
0405   35

0406   8D   STA $0500    ;Saves result in location 0500H
0407   00
0408   05

0409   60   RTS          ;Returns to MAC III System
```

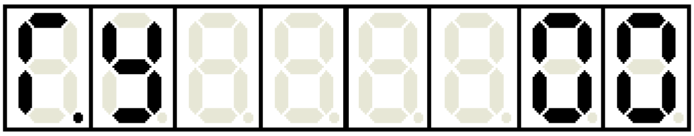
When the program has reached location 0406_H, the values 25_H and 35_H have been added. The sum of these values is $5A_H$, which is now in the accumulator.

Press the $\boxed{+}$ key again and the display will show:



This means that the X-register contains 00_H.

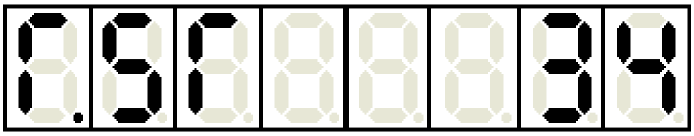
Pressing the $\boxed{+}$ key again will give the display:



This indicates that the Y-register also contains 00_H.

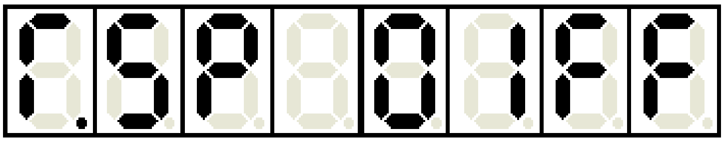
The uses of the X- and Y-registers will be explained in a subsequent chapter.

Press the $\boxed{+}$ key again and the display will show:



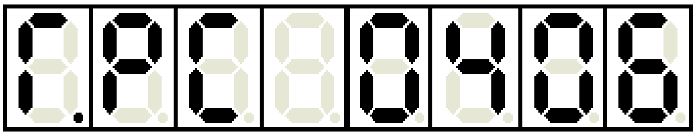
The hexadecimal value 34_H indicates the binary state of each bit within the status register (sometimes called the flag register). We have only seen the carry flag so far. This is the least significant bit of the status register (often referred to as "bit 0"). Now, $34_H = 0011\ 0100_2$ so the carry flag (i.e. least significant bit of the status register) is **clear**. The functions of the other bits within the status register will be explained as you progress through this manual.

Pressing the $\boxed{+}$ key again will produce the display:



This refers to the Stack Pointer register, which will be explained in a later chapter.

Press the + key once more and the display will again show:



This indicates that the current contents of the program counter register are 0406_H.

The 6502 registers can be checked again by pressing the + and - keys further. The contents of memory locations can also be checked at this stage by using the M key.

So, a break point will allow you to check two things:

1. that the program has actually reached the break point.
2. the contents of memory locations and 6502 registers at a given point in the program.

4.2a **Debugging is often necessary because user programs may:**

a require registers and memory locations to be specified.

b not be entirely correct when first executed.

c change the contents of ROM.

d use a break point.

4.2b **The keypad sequence "R R 0 4 1 7" will:**

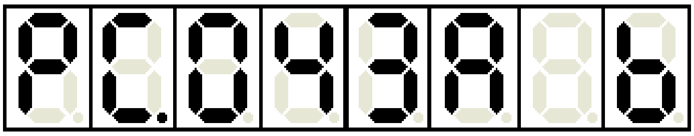
a allow the contents of location 0417_H to be modified.

b debug the program which starts at location 0417_H.

c set the Program Counter to 0417_H.

d insert a break point at location 0417_H.

4.2c The display



indicates that:

a the program start address is $043A_H$.

b a break point will be inserted at location $043A_H$.

c the contents of location $043A_H$ are 6_H.

d a break point has been reached at location $043A_H$.

4.3 Single Step

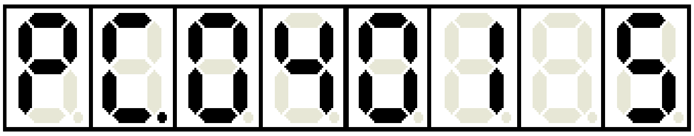
This allows the program under test to be halted at **every** instruction and the contents of registers and memory locations examined. This allows partial results to be inspected in the MAC III, in a similar way to break points.

The best way to understand how single step works is to step through a MAC III program. First enter the simple program shown in the previous section on break points.

Now press G and enter the start address of the program.

Press + and the first instruction **only** will be executed (In our example program, this is the "CLD" instruction).

The display will now show:



The “S” indicates that the MAC III is operating in a Single Step mode and the “PC.0401” that the address of the **next** instruction to be executed is 0401_H.

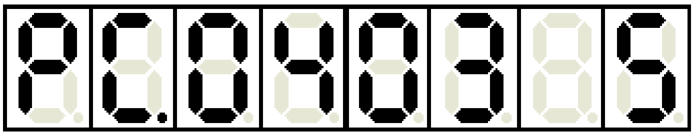
At this point you can press + again to execute the second instruction or press R then the + key to examine the contents of the CPU registers following the execution of the first instruction. Try examining the CPU registers at this point by pressing R. The display should show "r.PC 0401" to confirm the Program Counter contents.

The 6502 registers may now be cycled through, using the + and - keys. The hexadecimal keys can also be used to **alter** the contents of the CPU registers at this point if desired.

Note the accumulator contents, as these will be changed when we execute the second instruction of our program.

The second instruction can now be executed by pressing the G key, followed by the + key. Try this now.

The display should show:



Pressing + again will execute the next instruction. However, choose instead to examine the contents of the registers again, by pressing the R key and then using the + and - keys.

Notice that the accumulator now contains 25_H. This is the result of executing the second instruction ("LDA #$25").

The third instruction can be executed by pressing the G key, followed by the + key. Once again the address of the next instruction in sequence will be shown on the display.

4.3a **The keypad sequence required to start a program single stepping is:**

a G +
b R +
c G G
d G G R

4.4 Program Debugging

You will probably find that single stepping will prove the most useful program debugging technique in your first few programs. Try using the single step facility on some of the programs which you have already written.

You should be able to see the action of each instruction within the program by examining relevant registers and memory locations at each instruction.

When a break point is reached, it is possible to then single-step to the end of the program by pressing G +.

As your ability in writing machine code programs improves you will probably find that you are making more and more use of both single step and break points as the complexity of problems increases.

Student Assessment 4

1. **The process of finding and then correcting faults within a program is called:**

   a assembling.

   b compiling.

   c debugging.

   d linking.

2. **The key which is used at a break point to examine the contents of various registers is:**

   a G

   b R

   c +

   d -

3. **The key sequence required to set a break point at location 0428_H is:**

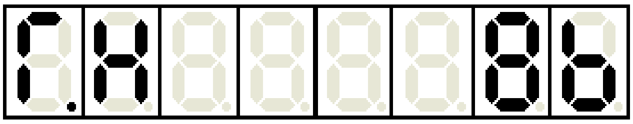
   a G G 0 4 2 8

   b M M 0 4 2 8

   c R R 0 4 2 8

   d S S 0 4 2 8

4. **The display**



indicates that the contents of:

a the Accumulator are $8B_H$

b the X Register are $8B_H$

c the Y Register are $8B_H$

d the Program Counter are $8B_H$

Chapter 5 The Merlin Text Editor

Objectives of this Chapter

Having studied this chapter you will be able to:

- Use the Merlin text editor to create, save and search text files.
- Access the Merlin command menus.
- Use the Merlin text editor to manipulate text
- Access the Merlin On-screen Help screens.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later. (Installation instructions are provided in the Getting Started book supplied with the software).
- MAC III 6502 User Manual.

Introduction

You will have already seen that hand assembly can be a time-consuming and error-prone process. A special computer program can be used to automatically **assemble** mnemonics into a machine-code program. Such a program is called an **Assembler**. The Assembler translates a **source program** written in mnemonics into an executable **object program** (machine code).

5.1 Text Editors

A text editor is a program that allows alphanumerical input (numbers and letters) to be entered into memory. These are almost always in the form of ASCII (American Standard Code for Information Interchange) codes.

The text editor will also allow alphanumeric input to be manipulated by means of a wide range of edit facilities. Programs written in this way are called **source code** and may be saved to disk as text files.

Merlin is a complete, Windows based, text editor for writing and editing any standard ASCII text file, including 6502 source code.

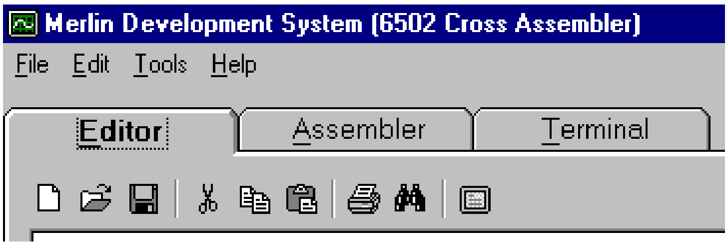
Merlin allows you to edit text using the PC keyboard, mouse or a combination of the keyboard and mouse.

5.2 Getting Started

Select **Programs** from the **Start menu,** then from the **LJ Merlin Development System** submenu click on the **Merlin** option. You should now have access to the Merlin text editing screen. If it is not already selected, click on the **Editor** tab.

If you are running the LJ Class*Act* Launcher software, you can also run the Merlin text editor using the **Merlin** application launch code.

The editor screen includes the standard windows menu titles and text editing command buttons, as shown opposite.



Place the mouse cursor over each of the toolbar buttons. A caption box will appear, this tells you the function of the button.

New – Creates an untitled blank text file. To save the file and give it a name, the **Save** function must be used.

Open – Opens an existing text file.

Save – Saves the open file. If the file has not previously been saved (that is, it has no name), then the **Save As** window will open, prompting you to name the file.

Cut – Removes selected text and places it on the clipboard ready for pasting. The selected text will remain on the clipboard until other text is copied or cut.

Copy – Copies the selected text and places it on the clipboard. The selection will remain available for pasting until other text is copied or cut.

Paste – Places the copied or cut text wherever the flashing cursor has been placed.

Print – Prints the currently open text file.

Find – Locates each occurrence of a given word.

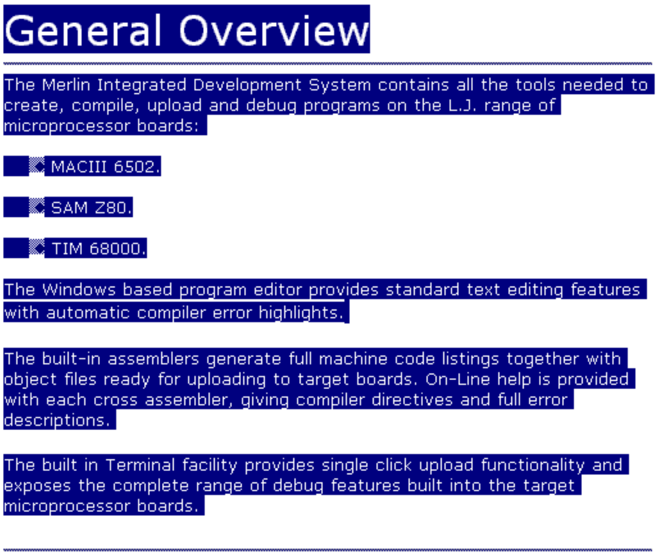
Options – Opens the **Options** window. This allows you to change the current settings.

These tools are also available from the menu bar at the top of the Merlin screen. The **File** menu contains the **New, Open, Print, Save** and **Options** commands, while the **Cut, Copy, Paste** and **Find** commands can be found in the **Edit** menu.

5.3 Use of the Merlin Editor

It is worthwhile spending a little time learning some of the basic features of the Merlin Editor, before going on to use it to create source files for the 6502 Cross Assembler.

From the **Help** menu select **Merlin On Screen Help,** the help screen will open. The screen is split into two sections. Copy the text in the right hand section by placing the mouse cursor to the left of the 'G' in 'General', hold the left mouse button down and drag the cursor downward until all the text is highlighted (see picture below). With the text selected, hold down the CTRL key on the keyboard and press the 'C' key (this is the keyboard shortcut CTRL + C). The selected text will be copied to the clipboard (Note there is not a Copy button available in the Help screen).



General Overview

The Merlin Integrated Development System contains all the tools needed to create, compile, upload and debug programs on the L.J. range of microprocessor boards:

- MACIII 6502.
- SAM Z80.
- TIM 68000.

The Windows based program editor provides standard text editing features with automatic compiler error highlights.

The built-in assemblers generate full machine code listings together with object files ready for uploading to target boards. On-Line help is provided with each cross assembler, giving compiler directives and full error descriptions.

The built in Terminal facility provides single click upload functionality and exposes the complete range of debug features built into the target microprocessor boards.

The selected text is now available for pasting. Close the help screen by clicking on the ☒ button in the top right corner of the window. This will bring you back to the main Merlin screen.

Left click inside the editing area once and then press the **Paste** button from the toolbar. You should now see the help text displayed in the editing screen. The font is different from the original, as Merlin does not support the help screen font style.

5.3a **Which of the following sequences is correct for copying text from one place to another?**

a Paste – Select – Save.

b Copy – Select – Cut.

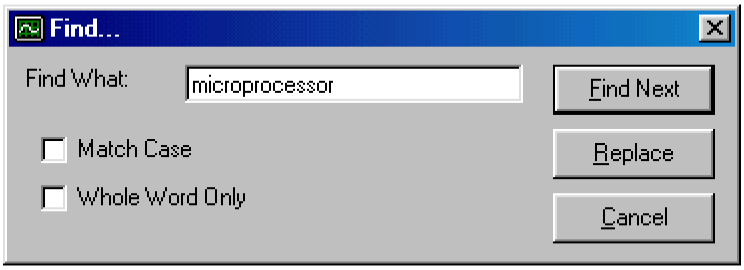
c Select – Copy – Paste.

d Print – Select – Copy.

Finding and Replacing Occurrences of Text

When dealing with large text files it is often beneficial to have a tool that locates specific text occurrences. This is especially true of large assembler source code files, as it is quite often necessary to change variable names and memory locations.

Click on the **Find** button on the toolbar. The **Find** window will open. If the **Find** window is covering the text, you can click on the blue title bar and drag it to a more convenient position. In the 'find what' field type 'microprocessor', then click the **Find Next** button. If the 'Finished Searching' text box appears then press 'OK' and you will be offered the option of starting again from the beginning, Do this. The first occurrence of the word will be found and selected (highlighted).



Find What: microprocessor

On the toolbar press the **Cut** button, the word will be deleted. Click the **Find Next** button on the **Find** window, the next occurrence will now be highlighted. From the toolbar press the **Copy** button then close the **Find** window by clicking cancel.

Place the flashing cursor between the words 'of' and 'boards' then press the **Paste** button. The original text should be restored.

In the **Edit** menu select the **Replace** option. In the ‘Find what’ field type ‘microprocessor’ (or use the paste facility). In the ‘Replace with’ field, type ‘mpcsr’ then click the **Replace All** button. All occurrences of the word microprocessor should now have been replaced with mpcsr. Swap the contents of the ‘Find what’ and ‘Replace with’ fields and repeat the process to restore the original text. Close the ‘Replace’ facility by clicking on the ‘Cancel’ button.

5.3b **In which menu are the Find and Replace commands located?**

a File
b Edit
c Tools
d Help

Saving Text Files

As you work through the exercises contained in this manual, you will create assembly language programs and save them to disk as source code files. You may save your files to floppy disk or to an area of a networked drive that has been made available by your instructor. For further guidance on saving your files, please consult your instructor.

Click on the **File** menu and select the **Save As** option. The ‘Save As’ window will open. Navigate to the drive and folder where your files will be saved. Press the **Create New Folder** button. A new folder will appear. Type ‘6502’ and press the Enter key to rename the folder. This will be the folder in which you save your source code files. It will be used only for 6502 programs as the source code for other microprocessors will be different.

The files you will create in subsequent chapters of this manual will be assembly language source code files. Although a source code file is essentially a text file, it must be saved as an *‘.ASM’* file as this allows the assembler to generate the object program.

Double click on the folder you have created and type *'Merlin'* in the 'file name field'. Check that the 'Save as type' option displays the *'.TXT'* extension. Press the save button. The window will close and the file will be saved in the 6502 folder.

5.3c **The file extension used for Assembly language source code is:**

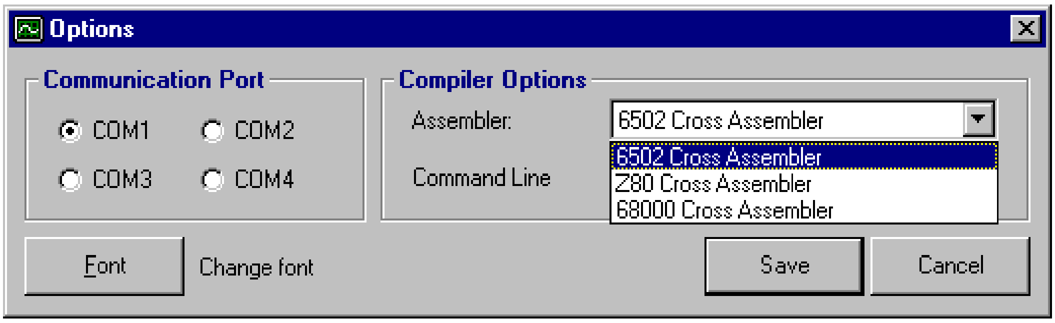
a .TXT
b .DOC
c .ASM
d .ASL

Printing Open Files

You can print the currently displayed file either by clicking the **Print** button on the toolbar or by choosing the **Print** option from the **File** menu.

5.4 Merlin Options

Ensure the serial communications cable (RS232) is fitted correctly between the MAC III board and the serial port of the PC. Make a note of the COM port number on the PC to which the cable is connected. From the **File** menu select **Options**. The Options window will open. Check the correct COM port is selected and change if necessary.



● COM1

Assembler: 6502 Cross Assembler

Under the **Compiler Options** ensure the 6502 Cross Assembler is selected from the 'Assembler' drop down menu.

A facility to change the font preferences is also available in the Options window. Click on the **Font** button, browse the different styles of fonts available then click **Cancel** to exit. On your return to the Options window, click on **Save** to save the current options.

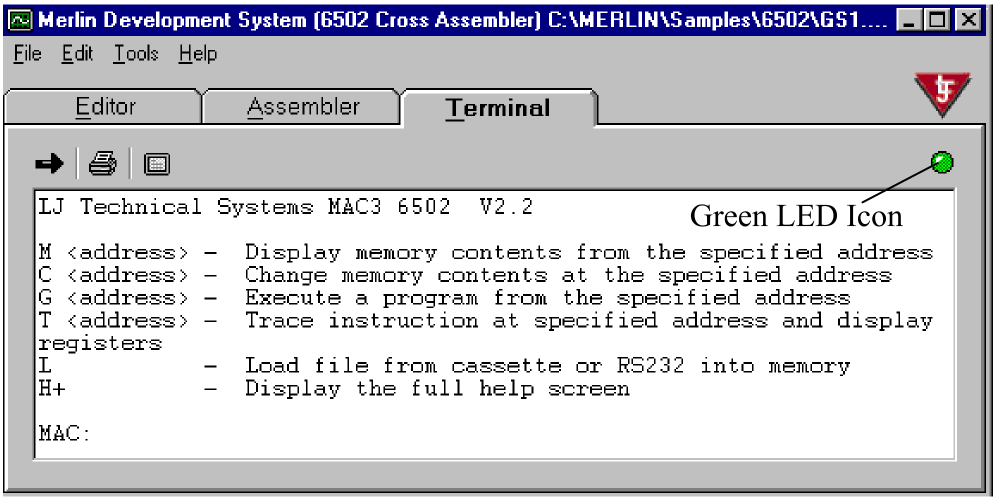
5.4a **Which of the following fonts is *not* available from the Fonts window?**

- a Courier.
- b Terminal.
- c Garamond.
- d LJ Terminal Display.

5.5 Checking Communication

In subsequent chapters of this manual you will be downloading programs from the PC to the MAC III board. In order to do this, there must be a working RS232 serial connection. To check that you have a working serial connection:

Click on the **Terminal** tab in the Merlin screen. In the top right corner there is an LED icon. With the MAC III board switched off this will appear red. Switch on the power supply to the MAC III board. The LED icon should now change to green and the screen will appear as below.



Green LED Icon

```
LJ Technical Systems MAC3 6502  V2.2

M <address> -  Display memory contents from the specified address
C <address> -  Change memory contents at the specified address
G <address> -  Execute a program from the specified address
T <address> -  Trace instruction at specified address and display
registers
L           -  Load file from cassette or RS232 into memory
H+          -  Display the full help screen

MAC:
```

You now have a working serial connection. Click on the Editor tab to return to the Merlin text editor.

Note: If there is no communication between the PC and the MAC III board, check that:

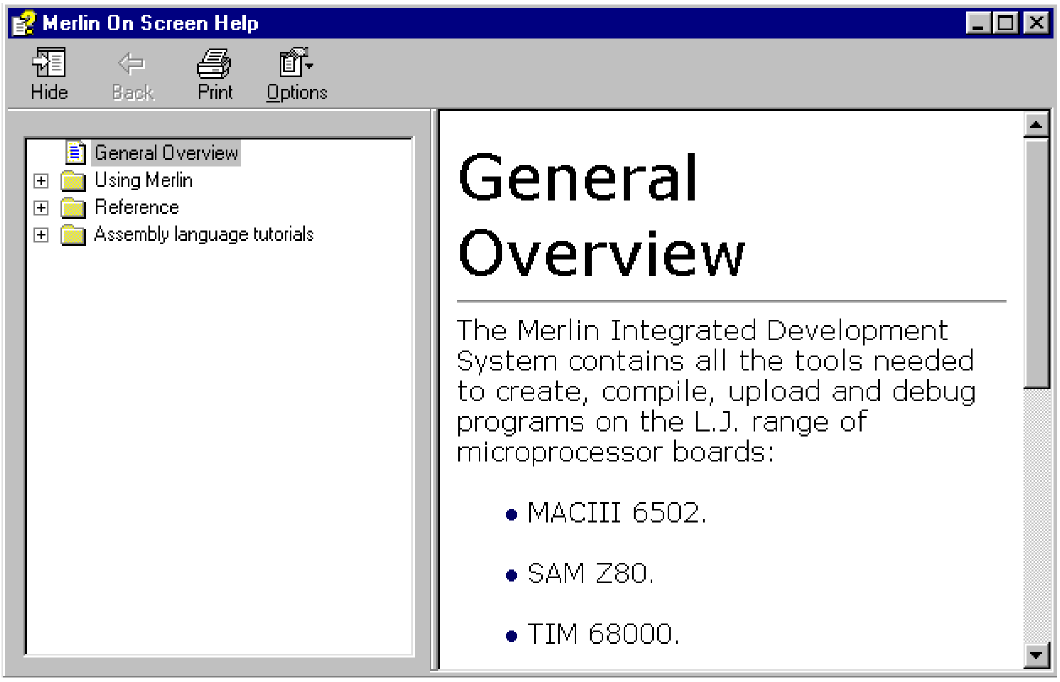
- the serial communication cable is connected correctly.
- the correct COM port is selected as described in Section 5.4.
- there is power supplied to the MAC III board (The power LED on the MAC III board should illuminate).

Then try to re-establish communication between the PC and the MAC III board.

5.6 Using the Merlin On Screen Help

The Merlin On Screen Help contains quick and easy access to help pages that cover all modes of Merlin operation.

From the **Help** menu select **Merlin On Screen Help.** The help screen will open as shown below. The screen is split into two separate windows.



General Overview

The Merlin Integrated Development System contains all the tools needed to create, compile, upload and debug programs on the L.J. range of microprocessor boards:

- MACIII 6502.
- SAM Z80.
- TIM 68000.

In the left window you will see the Navigation area. This allows you to select the help page required from the help folders. This window can be hidden using the **Hide** button and re-displayed using the **Show** button.

The pages are split into three sections as follows:

i. **Using Merlin** – covers the general functions of the different modes of Merlin.
ii. **Reference** – includes more specific details on subjects like error handling and transferring files.
iii. **Assembly language tutorials** - contains a sample program and notes for each Assembler you have installed.

The window on the right displays the help text that is selected. Each topic will contain links to other relevant topics. These appear as standard Windows navigation links (Colored blue and underlined).

In the left window expand the 'Using Merlin' folder by clicking on the ⊞ symbol. Click on the 'Using the source code editor' page. Read through this page using the scroll-bar on the right as required. Note how the link at the bottom of the page refers to the next page within the 'Using Merlin' folder.

5.6a **The link displayed at the bottom of the 'Using the source code editor' page is:**

a Assembling a program

b Compiling a program

c Error Handling

d Opening and Saving Files

Close the Help window by clicking on the ☒ button in the top right corner.

5.7 Exiting Merlin

You can exit from the Merlin Text Editor at any time, either by selecting the **Exit** command from the **File** menu or by clicking on the ☒ button in the top right corner of the window.

Try this now. You will be asked to save any unsaved work before you exit Merlin.

Student Assessment 5

1. **The three options that are contained in the Tools menu are:**

a Compiler, Terminal and Editor.

b Editor, Assembler and Terminal.

c Terminal, Assembler and Compiler.

d Print, Compiler and Terminal.

2. **This**  **button will:**

a Cut the currently selected text.

b Copy the currently selected text.

c Paste the text that is currently on the clipboard.

d Save the current file.

3. **The Merlin toolbar button that creates a new blank text file is:**

a 

b 

c 

d 

4. **The Merlin command used to place a duplicate of the selected text onto the clipboard is:**

a Paste.

b Copy.

c Cut.

d Print.

Continued ...

Student Assessment 5 Continued ...

5. The *Options* command can be found in which menu?

a File.

b Edit.

c Tools.

d Help.

6. The Merlin command that will locate each occurrence of a given word is:

a Select All.

b Edit.

c Find.

d Copy.

7. The links on the help pages are colored:

a red.

b green.

c yellow.

d blue.

8. The Merlin On Screen Help pages are split into how many sections?

a 1

b 2

c 3

d 4

Chapter 6 Introduction to Development Systems

Objectives of this Chapter

Having studied this chapter you will be able to:

- Use the Merlin Text Editor to enter 6502 assembly language programs.
- Recognize the operation of the 6502 Cross Assembler.
- Use Terminal commands to execute and debug an object program.
- Use Terminal Commands to examine and modify the contents of MAC III memory.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- 6502 Cross Assembler Reference Manual.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

An **Assembler** will examine the text of a source program and convert any 6502 instructions which it recognizes into 6502 machine code.

It will also alert you to any text it does not recognize and any instructions, which have incorrect form. Any text which follows a semi-colon (;) will be **ignored** by the assembler. This allows you to put **comments** in your programs.

6.1 Using the Text Editor to Create 6502 Source Programs

Many 6502 instructions require one or more **operands** to be specified. The operands specify the data, which is to be operated upon. These are listed after the instruction mnemonic. There are a number of possible operands (immediate data, absolute addresses, registers, etc.). For example:

```
LDA $0580
```

Here the operand is the absolute memory location 0580_H. . The '$' prefix indicates a hexadecimal number. Other types of numbers can be specified by the prefixes shown below:

Prefix	Number Type
%	Binary
@	Octal
None	Decimal
$	Hexadecimal

Another operand type is **immediate data**. This is a known **value**. Immediate data is specified by the '#' sign. This allows the assembler to distinguish between Immediate and other addressing modes. Immediate data can be expressed in binary, octal, decimal or hexadecimal.

For example:

```
LDA #%01010011    ;Loads the accumulator with the binary
                  ;value 01010011
LDA #@25          ;Loads the accumulator with the octal
                  ;value 25
LDA #43           ;Loads the accumulator with the decimal
                  ;value 43
LDA #$6E          ;Loads the accumulator with the hexadecimal
                  ;value 6E
```

The last major type of operand is a **label**. JUMP instructions require an operand to indicate the destination for the jump. In assembly language, you can specify locations, which may be jumped to by putting a label to them. You can then use the label as an operand for a jump instruction.

The last part of an assembly language line is a **comment**. Comments are totally ignored by the assembler, but are a very important aid to the programmer or another who wishes to understand the program. Assembly language programs tend to be quite difficult to follow if comments are omitted. Comments will help you to remember the function of a given section of code. Since the assembler ignores the comments, they do not cause the object program to become longer or reduce the speed of execution.

You will now use Merlin to generate a source program and save it in a file called PROG1.ASM.

Note: *The '.ASM' extension at the end of the filename is important, as it tells the Cross Assembler that this is an assembly language source file. If the filename does not have a '.ASM' extension, the Cross Assembler will not be able to generate any object code.*

Run the **Merlin** Cross Assembler as in Chapter 5, Section 5.2. You should now see the Merlin screen and the blank text editing area.

Note: *In order to carry out the work in this chapter and all subsequent chapters there will need to be a working serial connection between the MAC III board and the PC. Refer to Section 5.5 “Checking Communication” in Chapter 5 for further information.*

Enter the simple program below. Notice how the semi-colons are used to define the beginning of a comment:

```
CLD          ;Select binary arithmetic mode
LDA #$01     ;Loads accumulator with 01H
CLC
ADC #$02     ;Adds 02H to the accumulator
STA $0500    ;Saves result in 0500H
RTS          ;Returns to MAC III system
```

Now, the assembler will also need the required start address for the object code. A special instruction to the assembler (an assembler **directive**) is used for this purpose. The 'ORG' directive is used to tell the assembler where in memory to insert the object code.

Insert `ORG $0400` at the beginning of your program thus:

```
ORG $0400    ;Object code start address
CLD          ;Select binary arithmetic mode
LDA #$01     ;Loads accumulator with 01H
CLC
ADC #$02     ;Adds 02H to the accumulator
STA $0500    ;Saves result in 0500H
RTS          ;Returns to MAC III system
```

It is also good practice to give the program a title and a short description. These can be inserted as comments at the top of the screen thus:

```
;Program 1

;This program will add together 01H and 02H and save the
;result in location 0500H.

            ORG $0400     ;Object code start address
            CLD           ;Select binary arithmetic mode
            LDA #$01      ;Loads accumulator with 01H
            CLC
            ADC #$02      ;Adds 02H to the accumulator
            STA $0500     ;Saves result in 0500H
            RTS           ;Returns to MAC III system
```

Remember that each comment line must start with a semi-colon (;).

When you have completed this source program you can save your file by selecting the **Save As** command in the **File** menu. Save the file as “PROG1.ASM” in the 6502 folder previously created.

6.1a The character used to indicate binary data to the Cross Assembler is:

a %

b @

c $

d B

6.1b In a source program, the start address is specified using:

a a filename.

b an ORG directive.

c a LIST directive.

d a sub-directory.

6.2 Assembling an Object Program

It is now necessary to assemble the object code from your source program. There are two ways in which you can assemble your source code, either by selecting **Assembler** from the **Tools** menu or more simply by clicking on the **Assembler** tab. Click on the **Assembler** tab on the main screen. This assembles the source code and generates the object code. If the assembler finds no errors in your program, the message at the bottom of the assembler screen will show:

End of assembly: 0 errors found

When you see this message, an Object Code program has been assembled and saved in the 6502 folder. In the case of the PROG1.ASM file, an Object Code file was produced called PROG1.OBJ. The object code file can then be downloaded to the MAC III board, where its contents are stored in memory as machine code.

6.2a Assembly is the conversion of a source code program into:

a a development code program.

b a directive code program.

c an object code program.

d an operand code program.

If you do not see the message shown above, then your source program contains at least one error. In this case no Object Code will be saved; the line containing the error will be highlighted in red on the editor screen. Place the cursor on the line and a message indicating the nature of the error will appear at the bottom of the screen. You should make the necessary change to your source code and attempt to re-assemble.

The Assembler screen displays the source code program listing. This is quite a useful reference as this shows both the machine code and the corresponding program mnemonics. The physical address of each instruction is also shown.

The assembler screen will show the listing file as displayed below:

```
Line# Address  Object                    Source    AS65 6502 Cross Assembler V3.0
1     0400                              ;Program 1
2     0400
3     0400                              ;This program will add together 01H
                                        ;and 02H and save the result in
4     0400                              ;location 0500H
5     0400
6     0400               ORG $0400      ;Object code start address
7     0400    D8         CLD            ;Select binary arithmetic mode
8     0401    A9 01      LDA #$01       ;Loads accumulator with 01H
9     0403    18         CLC
10    0404    69 02      ADC #$02       ;Adds 02H to the accumulator
11    0406    8D 00 05   STA $0500      ;Saves result in 0500H
12    0409    60         RTS            ;Returns to MAC III system

End of Symbol Table

      0 labels declared
      12 sources lines read

18 bytes object code space used

End of assembly: 0 errors found
```

If you have a printer connected to your PC you can print this same listing by selecting the **Print** command either from the **File** menu or the toolbar.

On the editing screen remove the semi-colon in front of the line ‘Program 1’, and re-assemble the program. A message should appear as shown below:

End of assembly: 1 errors found

6.2b **The error displayed when placing the cursor on the line is:**

a ERROR 2: Missing space after label ‘1’.

b ERROR 4: Missing label after space ‘1’.

c ERROR 1: Fault at line ‘1’.

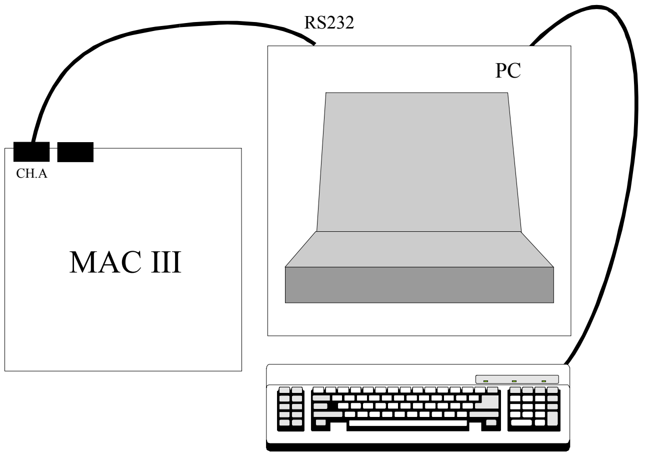
d ERROR 7: Missing label ‘10’.

Using the instructions described previously, remove the error and re-assemble the program.

6.3 Executing Assembled Programs

Although you can edit and assemble 6502 machine code using your PC, you cannot **run** 6502 programs. This is because the microprocessor within the PC is not a 6502 and so 6502 machine code is meaningless to it. It is therefore necessary to transfer your 6502 object code program from the PC to the MAC III.

To allow the PC to communicate with the MAC III, it is necessary to enter the ‘Terminal’ mode. Firstly, ensure that power to the MAC III is OFF, and that the PC and MAC III are connected via the serial communications cable supplied with the Merlin Development System.



PC connected to MAC III via the serial communications cable

Click on the **Terminal** tab to enter the Terminal mode. Switch on the power to the MAC III board. The Terminal screen will display as follows:

```
LJ Technical Systems MAC3 6502  V2.2

M <address> -  Display memory contents from the specified address
C <address> -  Change memory contents at the specified address
G <address> -  Execute a program from the specified address
T <address> -  Trace instruction at specified address and display registers
L           -  Load file from cassette or RS232 into memory
H+          -  Display the full help screen

MAC:_
```

You are now in **Terminal Mode**. In this mode the PC displays on its screen any character received through its serial port, from the target board. Also, any commands entered at the PC keyboard are transmitted to the target board. The light in the top right hand corner of the terminal screen indicates the state of the Terminal, that is a green light is displayed if connected and a red light if disconnected.

In effect, the PC is behaving as the keyboard and display of the MAC III.

Thus any key pressed on the PC keyboard is interpreted as a command by the MAC III. The MAC III will then respond by displaying information on the PC screen.

Note: *You can return to the* `MAC III:_` *prompt at any time, simply by pressing the RESET button on the MAC III board.*

To download the object code program to the MAC III, simply click the **Send File to Board** button. ➡ The PC will then download the most recently assembled object code file via the MAC III serial port.

If the file is successfully downloaded, the MAC III command prompt on the Terminal screen will respond `'Loaded'`.

Note: If the MAC III board does not respond correctly, refer to the Checking Communication section of Chapter 5.

You can check that the program has been entered into MAC III memory by pressing the M and the Enter keys in sequence. The display will then show:

```
0400:   D8   A9   01   18   69   02   8D   00   ....i...
0408:   05   60   FF   FF   FF   FF   FF   FF   .'......
0410:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0418:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0420:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0428:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0430:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0438:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0440:   FF   FF   FF   FF   FF   FF   FF   FF   ........
0448:   FF   FF   FF   FF   FF   FF   FF   FF   ........
```

This indicates that the contents of 0400_H are $D8_H$, the contents of 0401_H are $A9_H$ and so on. The right-hand column shows the ASCII equivalent of the contents of each memory location. The default value for memory display is 0400_H.

You can examine any area of MAC III memory by entering the start address after the ‘M’. For example, ‘M 0500’ will display the contents of MAC III locations 0500_H to $054F_H$.

The amount of memory shown on the screen can be altered by appending a semi-colon and the number of bytes to the command thus: ‘M 0400;8’ will display:

```
0400: D8 A9 01 18 69 02 8D 00  ....i...
```

Now examine the contents of location 0500_H by entering “M 0500”. The display should show the contents of 0500_H to be FF_H. When the program has been run we shall examine 0500_H again to confirm correct operation.

Press G then Enter to run the program. The default address for execution is 0400_H.

You can execute from any location by entering the start address after the “G”. For example, “G 0600” will execute from location 0600_H.

If you now use the “M” command to examine memory location 0500_H you should find that it has been modified to 03_H by the program ($01_H + 02_H = 03_H$).

So, you can now write and edit source programs, assemble these into 6502 Object Code and transfer programs to the MAC III.

6.3a **After entering "M 0480;8" the display screen shows:**

```
0480: 3D 06 E3 78 EF D2 10 05 >.......
```

This indicates that the contents of location 0486_H are:

a 05_H

b 10_H

c $3D_H$

d $D2_H$

6.3b **The Terminal Mode key sequence G 0 5 4 0 Enter will cause:**

a object code to be assembled, starting at location 0540_H.

b the contents of location 0540_H to be examined but *not* modified.

c program execution from location 0540_H.

d object code to be transferred to the MAC III, starting at location 0540_H.

6.4 Use of Labels

Labels can be used within a source file in place of hexadecimal values or addresses. The 'value' of the label is defined using an EQU assembler directive. For example:

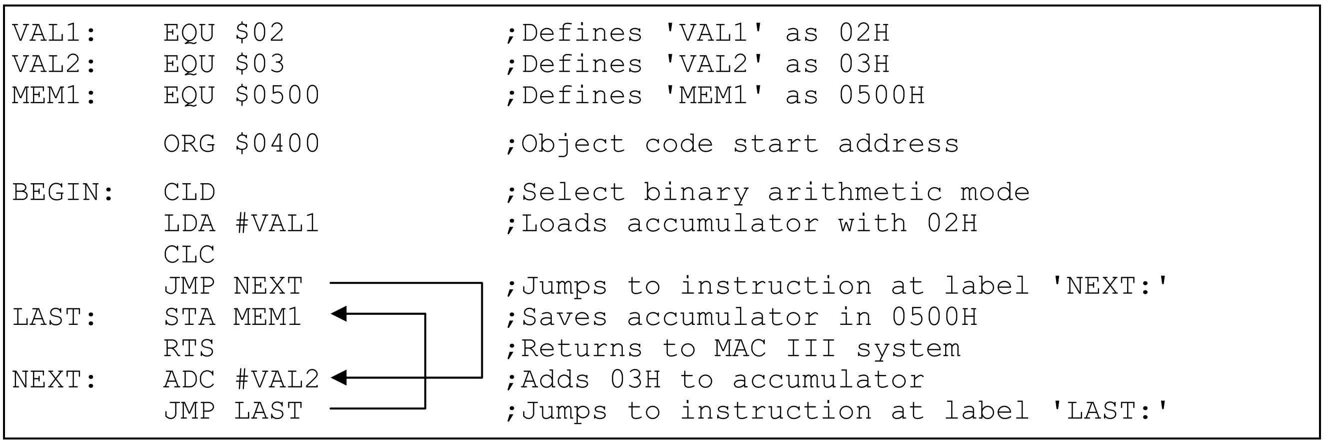
```
MEMORY:     EQU $0500        ;Defines "MEMORY" as 0500H
FIRST:      EQU $01          ;Defines "FIRST" as 01H
SECND:      EQU $02          ;Defines "SECND" as 02H

            ORG $0400        ;Object code start address
            CLD              ;Select binary arithmetic mode
            LDA #FIRST       ;Loads accumulator with 01H
            CLC
            ADC #SECND       ;Adds 02H to the accumulator
            STA MEMORY       ;Saves result in 0500H
            RTS              ;Return to MAC III System
```

The assembler will insert '01_H' in place of 'FIRST', '02_H' in place of 'SECND' and '0500_H' in place of 'MEMORY'.

There are a number of rules for the use of labels:

1. Labels must begin with a letter but may include numbers. So 'NUMB7' is acceptable, whereas '7NUMB' is not acceptable. Lower or upper case letters may be used.
2. Labels are limited to 8 characters.
3. A label must not be a reserved word or a 6502 mnemonic. A list of reserved words can be found in the 6502 Cross Assembler Reference Manual.
4. When a label is defined, it must appear in the left hand column of the assembly language and must be followed immediately by a colon (:). See the example above.

Return to the editing area by clicking on the **Editor** tab. Select the **New** command from the **File** menu or from the header bar icon. Enter the program shown on the next page. **Do not try to enter the arrows**. These are just to help you to understand how the program works.



```
VAL1:    EQU $02            ;Defines 'VAL1' as 02H
VAL2:    EQU $03            ;Defines 'VAL2' as 03H
MEM1:    EQU $0500          ;Defines 'MEM1' as 0500H

         ORG $0400          ;Object code start address

BEGIN:   CLD                ;Select binary arithmetic mode
         LDA #VAL1          ;Loads accumulator with 02H
         CLC
         JMP NEXT           ;Jumps to instruction at label 'NEXT:'
LAST:    STA MEM1           ;Saves accumulator in 0500H
         RTS                ;Returns to MAC III system
NEXT:    ADC #VAL2          ;Adds 03H to accumulator
         JMP LAST           ;Jumps to instruction at label 'LAST:'
```

JMP is a JUMP instruction. It will transfer program execution to a point other than the next location in sequence.

Your source program should be:

```
; Program 2

; This program adds 02H and 03H, saving the result in
; location 0500H, using labels.

VAL1:       EQU $02      ;Defines 'VAL1' as 02H
VAL2:       EQU $03      ;Defines 'VAL2' as 03H
MEM1:       EQU $0500    ;Defines 'MEM1' as 0500H

            ORG $0400    ;Object code start address

BEGIN:      CLD          ;Select binary arithmetic mode
            LDA #VAL1    ;Loads accumulator with 02H
            CLC
            JMP NEXT     ;Jumps to instruction at label 'NEXT:'
LAST:       STA MEM1     ;Saves accumulator in 0500H
            RTS          ;Returns to MAC III system
NEXT:       ADC #VAL2    ;Adds 03H to accumulator
            JMP LAST     ;Jumps to instruction at label 'LAST:'
```

Save the program as “PROG2.ASM”.

Assemble this program and produce a listing thus:

```
Line#  Address Object    Source AS65 6502 Cross Assembler V3.0

1      0400              ;Program 2
2      0400
3      0400              ;This program will add together 02H and 03H and
4      0400              ;save the result in location 0500H using labels
5      0400
6      0400              VAL1:   EQU  $02
7      0400              VAL2:   EQU  $03
8      0400              MEM1:   EQU  $0500
9      0400
10     0400                      ORG  $0400 ;Object code start address
11     0400
12     0400    D8        BEGIN:  CLD        ;Select binary arithmetic mode
13     0401    A9 02             LDA  #VAL1 ;Loads accumulator with 02H
14     0403    18                CLC
15     0404    4C 0B 04          JMP  NEXT  ;Jumps to instr. at label 'NEXT:'
16     0407    8D 00 05  LAST:   STA  MEM1  ;Saves result in 0500H
17     040A    60                RTS        ;Returns to MAC III system
18     040B    69 03     NEXT:   ADC  #VAL2 ;Adds 03H to accumulator
19     040D    4C 07 04          JMP  LAST  ;Jumps to instr. at label 'LAST:'
20     0410

Symbol Table

Symbol                           Value     Cross-reference (# is definition)

BEGIN . . . . . . . . . . . . .  0400      12#
LAST. . . . . . . . . . . . . .  0407      16#   19
MEM1. . . . . . . . . . . . . .  0500      8#    16
NEXT. . . . . . . . . . . . . .  040B      15    18#
VAL1. . . . . . . . . . . . . .  0001      6#    13
VAL2. . . . . . . . . . . . . .  0002      7#    18

End of Symbol Table

       6 labels declared
       20 sources lines read

24 bytes object code space used

End of assembly: 0 errors found
```

Notice that the assembler inserts hexadecimal values in place of labels. In the case of JUMP instructions the addresses are given low byte first. The listing file will also list the label definitions. This can be very useful, as a crosscheck.

Enter the terminal mode to download your program to the MAC III board. Run the program (G Enter) and examine the memory contents (M0500;1) to ensure that the program has been successful.

So, you have now seen two ways of defining an address label:

1. Using an EQU assembler directive (for example, MEM1 above).
2. Inserting the label just before an instruction (for example, LAST above).

There are other ways of defining labels. The program shown below loads locations 1100_H to 1102_H with the value 88_H. This can be used to demonstrate a third means of defining a label:

```
;Program 3

;This program uses labels which are modified to point to various
;locations

            ORG $0500         ;Object code start address

VAL1:       EQU $88           ;Defines "VAL1" as 88H
MEM1:       EQU $1100         ;Defines "MEM1" as 1100H

            LDA #VAL1         ;Loads accumulator with 88H
            STA MEM1          ;Saves accumulator in 1100H
            STA MEM1+1        ;Saves accumulator in 1101H
            STA MEM1+2        ;Saves accumulator in 1102H
            RTS               ;Returns to MAC III system
```

The value 1100_H is assigned to “MEM1”. It is however possible to change this within the program as shown above. So MEM1+1 is 1101_H and MEM1+2 is 1102_H.

Use Merlin to create the new source program on the previous page and save this program as “PROG3.ASM”. Assemble this program and produce a listing thus:

```
Line# Address  Object     Source AS65 6502 Cross Assembler V3.0

 1    0400                ;Program 3
 2    0400
 3    0400                ;This program uses labels which are modified to
 4    0400                ;point to various locations
 5    0400
 6    0400
 7    0400                       ORG   $0500    ;Object code start address
 8    0500
 9    0500                VAL1:  EQU   $88      ;Defines 'VAL1' as 88H
 10   0500                MEM1:  EQU   $1100    ;Defines 'MEM1' as 1100H
 11   0500
 12   0500    A9 88              LDA   #VAL1    ;Loads accumulator with 88H
 13   0502    8D 00 11           STA   MEM1     ;Saves accumulator in 1100H
 14   0505    8D 01 11           STA   MEM1+1   ;Saves accumulator in 1101H
 15   0508    8D 02 11           STA   MEM1+2   ;Saves accumulator in 1102H
 16   050B    60                 RTS            ;Returns to MAC III system
 17   050C

Symbol Table

Symbol                         Value Cross-reference (# is definition)

MEM1. . . . . . . . . . . . . . 1100  10#  13 14 15
VAL1. . . . . . . . . . . . . . 0088  9#   12

End of Symbol Table

      2 labels declared
      17 sources lines read

20 bytes object code space used

End of assembly: 0 errors found
```

Notice that the object code address references are modified according to the label modifier.

Download the program to the Mac III board and run from location 0500_H (G0500 Enter). Examine memory locations (M1100;3) to ensure correct operation.

6.4a **If the label `'VAL1'` is assigned the value $2D_H$, the 6502 Cross Assembler will interpret `'VAL1+2'` as:**

a 02_H

b 03_H

c $2D_H$

d $2F_H$

6.4b **The maximum number of characters for a label recognized by the 6502 Cross Assembler, is:**

a 6

b 8

c 12

d 26

6.5 Debugging Using the Terminal Software

An earlier chapter dealt with the debugging facilities available from the keypad/display unit - Break Point and Single Step. These and enhanced debugging facilities may also be used from Terminal.

Break Points

The break point facility allows a program to be halted at any desired point. The 6502 registers are then displayed on the screen, and memory locations may be examined. This allows partial program results to be inspected. A break point is easily set by entering ‘B’ followed by the required address. For example, to set a break point at location 0404_H, enter ‘B 0404’ at the ‘**`MAC III`**:’ prompt.

From Merlin file menu, open file PROG1.ASM and assemble the source code, using the commands described previously. Then select Terminal mode and download the object code file to the MAC III. Next, set a break point at 0406_H by entering 'B 0406'.

Run the program from the beginning by entering ‘G 0400’ and the display will show ‘**`*** At breakpoint ***`**’. The contents of the 6502 registers and the next instruction to be executed are also shown. To continue execution from a break point, simply press G followed by Enter .

A break point can be removed by entering ‘K’ followed by the required address. So, to clear the break point we have just set, enter ‘K 0404’. As with the MAC III monitor break point facility, up to 8 break points can be set. The command ‘K*’ will clear **all** break points.

Single Step

The Single Step or ‘Trace’ facility allows a program to be stepped through, instruction by instruction. At each step the contents of 6502 registers are displayed on the screen, and memory locations may be examined. To trace a program which starts at location 0400_H, enter ‘T 0400’ at the ‘**`MAC III`**:’ prompt. Use the ‘M’ command to check that PROG1 is still in MAC III memory. If this is not the case you will have to download the PROG1 object code program to the MAC III once again.

Now, single step from 0400_H by typing `'T 0400'` followed by Enter . The first instruction will be executed and the screen will display the contents of each register, and the next instruction to be executed. Press the Enter key and the next instruction is executed and the registers displayed. You can continue stepping through the program by pressing the Enter key.

Memory Edit

The 'M' command allows the contents of memory locations to be inspected. The contents of memory locations can be **modified** by using the 'C' command when in **Terminal** mode. This can work in a number of ways. Firstly, the contents of a single memory location can be changed by entering 'C' followed by the required address. Enter 'C 0600' and the screen will show:

```
0600:   FF
```

This shows the contents of 0600_H to be FF_H. you can modify these by entering the desired value, say 12_H and pressing the Enter key.

The screen will now show:

```
0600:     FF   12
0601:     FF
```

The contents of 0600_H are now 12_H and the contents of 0601_H can now be modified. If it is not necessary to modify 0601_H, press the Esc key and the 'MAC:' prompt will return. Alternatively, a colon after the new value will return to the 'MAC:' prompt thus:

```
0600:   FF  12:
MAC:_
```

A second way of using the 'C' command is to modify a number of consecutive locations. Enter 'C 0600' and the display will show:

```
C 0600
0600:   12  _
```

It is now possible to enter consecutive values with spaces between each thus:

```
C 0600
0600:   12  21 45 D4 22 E7:
```

Use the 'M' command to display thus 'M 0600;5' and the screen will show:

```
0600:   21  45  D4  22  E7
```

A third way of using the 'C' command is to enter ASCII codes directly. For example, to enter the ASCII codes for the message 'Hello' from location 0600_H: Enter 'C 0600' and type the ASCII characters in between quotation marks thus:

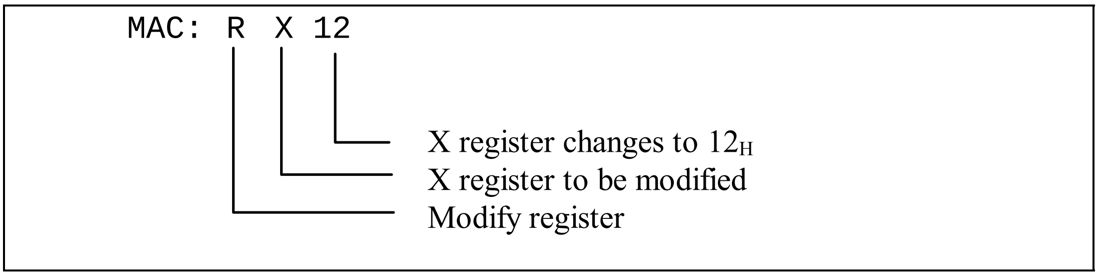
```
0600:  21  "Hello":
```

Use the 'M' command to display thus 'M 0600;5' and the screen will show:

```
0600:   48  65  6C  6C  6F      Hello
```

Display/Modify Registers

The ‘R’ command allows 6502 registers to be examined and modified. If you now enter ‘R’, the display will show the contents of all 6502 registers. The contents of a register can be changed by specifying the register and then the required value. For example:



Disassemble

Recall that assembling is the process of producing an object code program from a source program in 6502 Assembly Language mnemonics. Disassembly is the **reverse** process. In disassembly a 6502 Assembly Language mnemonic listing is produced from an object code program. The disassembler cannot, of course, reproduce comments but a source listing is far easier to understand than machine code!

So, a **disassembler** takes an object code program and presents it in assembly language form. This can be a very useful tool if source program files have been lost or are otherwise unavailable. The disassembler command is ‘D’. To disassemble a program in `MAC III` memory, enter ‘D’, followed by the start address.

For example, we can disassemble the start of the MAC III Monitor program in the Monitor EPROM.

To do this, type 'D F022' and press the Enter key.

The display will be similar to that shown below:

```
F022:   78          SEI
F023:   D8          CLD
F024:   A9 00       LDA  #00
F026:   8D 66 02    STA  0266
F029:   A2 80       LDX  #80
F02B:   9A          TXS
F02C:   AD 0D 80    LDA  800D
F02F:   29 10       AND  #10
F031:   F0 21       BEQ  F043
F033:   AD 67 02    LDA  0267
```

You should recognize some of the 6502 instruction mnemonics shown. By the end of this manual you will have used all of the instruction types above.

The length of the code to be disassembled can be specified thus:

D F022;20

The 20_H **instructions** from $F022_H$ will be disassembled

Keypad Restart

This facility allows control to be returned to the MAC III keypad. Simply type 'P' and press the Enter key.

To return control to Terminal Mode, press the RESET button on the MAC III board.

6.5a **The correct Terminal Mode key sequence to examine the contents from location 0480$_H$ is:**

- a 0 4 8 0 M Enter
- b 8 0 0 4 M Enter
- c M 0 4 8 0 Enter
- d M 8 0 0 4 Enter

6.5b **The Terminal Mode key sequence C 0 6 A 0 Enter will allow:**

- a object code to be assembled, starting at location 06A0$_H$.
- b the contents of location 06A0$_H$ to be examined but <u>not</u> modified.
- c the contents of location 06A0$_H$ to be examined <u>and</u> modified if required.
- d object code to be transferred to the MAC III, starting at location 06A0$_H$.

Student Assessment 6

1. **The `ORG` assembler directive is used to:**
   - a return to the MAC III system.
   - b assemble an object code program.
   - c define the start address for an object code program.
   - d generate error messages.

2. **Which of the following lines is a comment and will be ignored by the assembler?**
   - a # Program 1
   - b “Program 1”
   - c ; Program 1
   - d (Program 1)

3. **The instruction** “`LDA #$01`” **executes which operation?**
   - a Adds 01_H to the value stored in the accumulator.
   - b Stores the value held in the accumulator at address 01_H.
   - c Loads 01_H into the accumulator.
   - d Resets the accumulator to zero.

4. **The Terminal Mode key sequence** M 0 5 0 0 Enter **will allow:**
   - a the contents of location 0500_H to be examined.
   - b the contents of location 0500_H to be examined and modified.
   - c the execution of the object program which starts at location 0500_H.
   - d the saving to disk of the object program which starts at location 0500_H.

Continued ...

Student Assessment 6 Continued ...

5. **Assembling a file called "PROG6.ASM" will also create a file named:**

- a PROG6.OBJ
- b PROG6.TXT
- c PROG6.ORG
- d PROG6.TML

6. **The Terminal Mode key sequence $\boxed{\text{G}}$ $\boxed{\text{0}}$ $\boxed{\text{2}}$ $\boxed{\text{0}}$ $\boxed{\text{0}}$ $\boxed{\text{Enter}}$ will allow:**

- a the contents of location 0200_H to be examined but *not* modified.
- b the contents of location 0200_H to be examined *and* modified if required.
- c the execution of the object program which starts at location 0200_H.
- d the saving to disk of the object program which starts at location 0200_H.

7. **The contents of memory location 0380_H can be examined and modified using the Terminal Mode key sequence (followed by $\boxed{\text{Enter}}$):**

- a $\boxed{\text{A}}$ $\boxed{\text{0}}$ $\boxed{\text{3}}$ $\boxed{\text{8}}$ $\boxed{\text{0}}$
- b $\boxed{\text{C}}$ $\boxed{\text{0}}$ $\boxed{\text{3}}$ $\boxed{\text{8}}$ $\boxed{\text{0}}$
- c $\boxed{\text{G}}$ $\boxed{\text{0}}$ $\boxed{\text{3}}$ $\boxed{\text{8}}$ $\boxed{\text{0}}$
- d $\boxed{\text{M}}$ $\boxed{\text{0}}$ $\boxed{\text{3}}$ $\boxed{\text{8}}$ $\boxed{\text{0}}$

8. **The execution of a program starting at address 0600_H can be traced using the key sequence (followed by $\boxed{\text{Enter}}$):**

- a $\boxed{\text{L}}$ $\boxed{\text{0}}$ $\boxed{\text{6}}$ $\boxed{\text{0}}$ $\boxed{\text{0}}$
- b $\boxed{\text{G}}$ $\boxed{\text{0}}$ $\boxed{\text{6}}$ $\boxed{\text{0}}$ $\boxed{\text{0}}$
- c $\boxed{\text{C}}$ $\boxed{\text{0}}$ $\boxed{\text{6}}$ $\boxed{\text{0}}$ $\boxed{\text{0}}$
- d $\boxed{\text{T}}$ $\boxed{\text{0}}$ $\boxed{\text{6}}$ $\boxed{\text{0}}$ $\boxed{\text{0}}$

Chapter 7 Addressing Modes

Objectives of this Chapter

Having studied this chapter you will be able to:

- Describe the operation of the 6502 Addressing Modes:

  Implied
  Immediate
  Absolute
  Zero Page

- Write assembly language programs which use Implied, Immediate, Absolute and Zero Page addressing.
- Explain the use of Binary Coded Decimal (BCD) numbers to represent decimal values.
- Write programs that perform decimal arithmetic.
- Describe the operation of the 6502 Subtract instruction (SBC).
- Write assembly language programs which use the 6502 Subtract instruction.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

Addressing is concerned with the way in which operands are specified. So far we have seen the **Implied, Immediate** and **Absolute** addressing modes. This chapter will also introduce the **Zero Page** addressing mode. The 6502 actually has 13 different modes of addressing.

7.1 Implied Addressing

Recall that in Chapter 3 you learned that some instructions do not require an operand. However, this is not strictly true. There are instructions which **apparently** require no operand because the operand is **implicit** within the operator. For example, CLC and RTS. Implied addressing instructions are all **single-byte** instructions.

7.2 Immediate Addressing

In immediate addressing, the operand is itself contained in the byte of memory which **immediately** follows the opcode. This type of addressing can be used to load the accumulator with any 8-bit value. For example, consider the short section of program below:

```
0500  A9    LDA #$12    ; Loads accumulator with 12H
0501  12
0502  69    ADC #$34    ; Adds 34H to accumulator
0503  34
0504  60    RTS         ; Returns to ready
```

The instructions at locations 0500_H and 0502_H are examples of Immediate addressing. The actual operand is specified in each case by the following byte of memory.

Immediate addressing instructions will be made up of **two bytes**. The first byte is the opcode byte and the second the immediate data byte.

7.3 Absolute Addressing

In this mode of addressing, the **absolute** address of the operand is contained in the **two** bytes of memory immediately following the opcode, low byte first. This type of addressing can be used to load the accumulator with the contents of any location in memory or to save the contents of the accumulator in any memory location.

For example, consider the short section of program below:

```
0500  AD    LDA $1800   ; Loads accumulator from 1800H
0501  00
0502  18
0503  8D    STA $1880   ; Saves the accumulator in
0504  80                ; location 1880H
0505  18
0506  60    RTS         ; Returns to MAC III monitor
```

The instructions at locations 0500_H and 0503_H are examples of Absolute addressing. The address of the operand is specified in each case by the following two bytes of memory.

The address of the data to be acted upon may be anywhere within the full address range of 0000_H to $FFFF_H$.

Absolute addressing instructions will be made up of **three bytes**. The first byte is again the opcode byte. The second and third bytes specify the **address** of the operand.

7.3a The Accumulator initially contains the value $2C_H$ and location 0580_H initially contains $4D_H$. Enter the value which would be found in the Accumulator after the instruction "`LDA $0580`" **has been executed.**

7.4 Zero Page Addressing

A microcomputer memory system can be thought of in terms of **pages**, rather like the pages of a book. Each page has 256_{10} locations thus:

Page	Start Address	Final Address
00	0000	00FF
01	0100	01FF
02	0200	02FF
03	0300	03FF
|	|	|
|	|	|
FE	FE00	FEFF
FF	FF00	FFFF

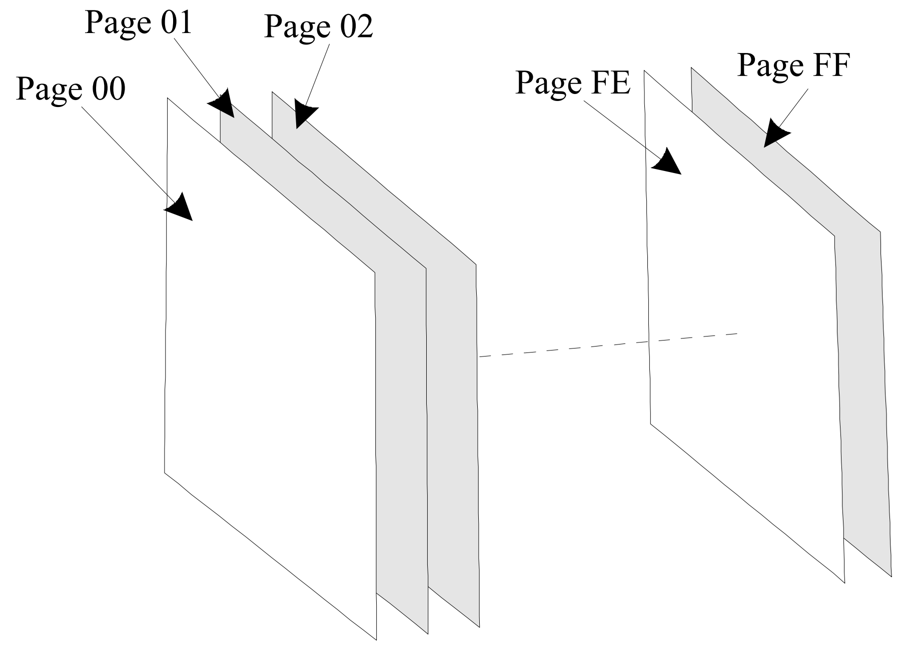


Diagram of Memory Pages

This mode is very similar to Absolute addressing except that operands may only occupy the address range 0000_H to $00FF_H$ - Page Zero of memory.

In this mode of addressing, the **Page Zero** address of the operand is contained in the byte of memory immediately following the opcode.

For example, consider the short program below:

```
0500  A9    LDA #$65     ; Loads accumulator with 12H
0501  65                 ; the value 65H
0502  85    STA $80      ; Saves the accumulator in
0503  80                 ; location 0080H
0504  60    RTS          ; Returns to MAC III monitor
```

The instruction at location 0502_H is an example of Zero Page addressing. The address of the operand is specified by its **page zero** address. Notice that in 6502 assembly language the "00" for page zero is ignored. Thus, location 0080_H is expressed as "$80".

Zero Page addressing instructions will be made up of **two bytes**. The first being the opcode byte and the second specifying the Page Zero address of the operand. Recall that immediate addressing instructions also comprise two bytes. Care must be taken not to confuse these two modes. Notice that the assembly language for these two modes is rather different:

```
LDA  #$73     ;Loads the accumulator with the
              ;immediate value 73H

LDA  $73      ;Loads the accumulator from memory
              ;location 0073H
```

This type of addressing has two advantages over Absolute addressing:

1. Fewer bytes are required.
2. Since fewer bytes are required, this mode is faster in operation.

Zero Page addressing has the disadvantage that it is restricted to the first 256_{10} locations in memory.

7.4a The 6502 Assembly Language program section:

```
LDA  #$42
STA  $70
```

a will add the values 42_H and 70_H

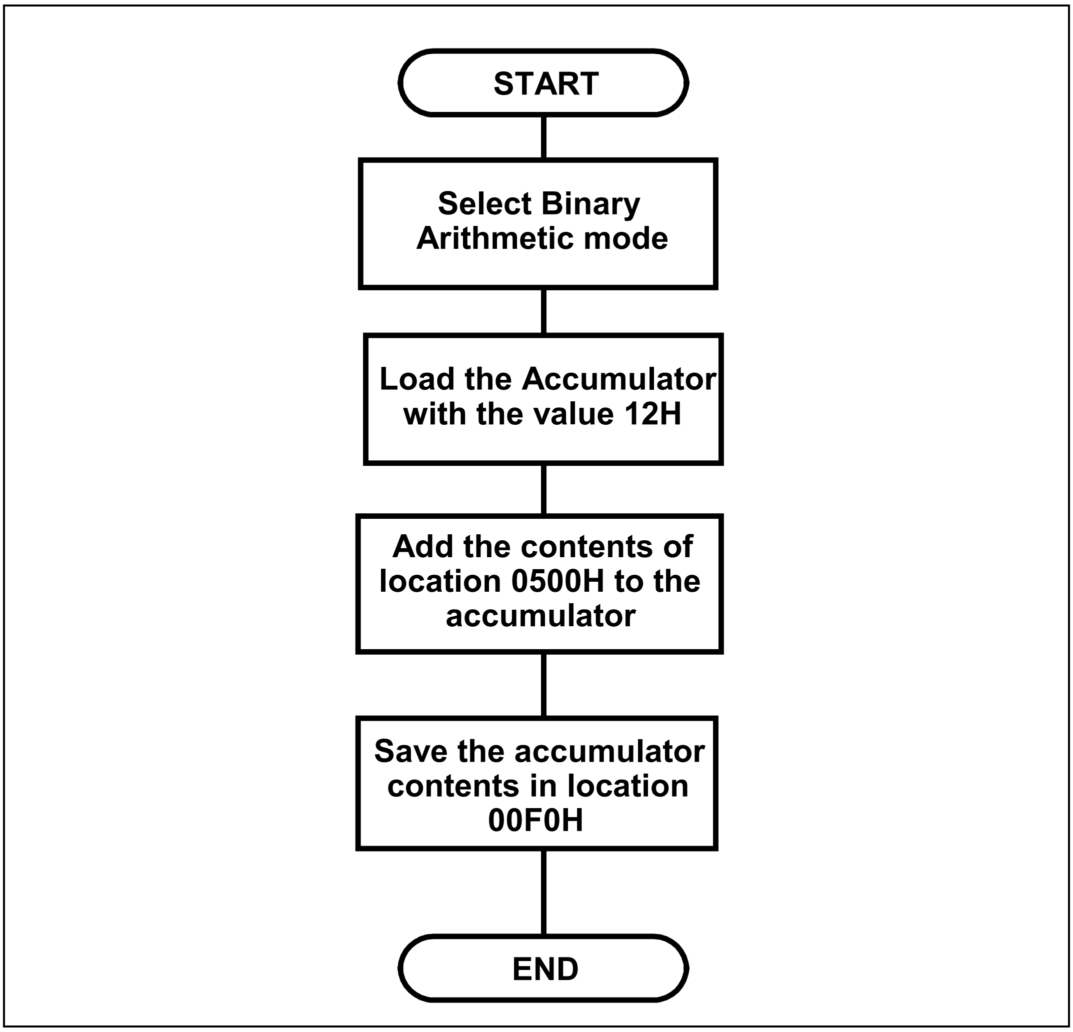
b will add the values 42_H and 70_H saving the result in location 0070_H

c will place the value 42_H in memory location 0070_H

d will place the value 42_H in memory location 7000_H

7.5 Worked Example

Write a program using zero page and other addressing modes which will add 12_H to the contents of memory location 0500_H and save the result in location $00F0_H$. The start address should be 0400_H.



The Assembly Language Program will be:

```
             ORG    $0400   ;Defines start address
0400   D8    CLD            ;Selects binary arithmetic mode
0401   A9    LDA    #$12    ;Loads accumulator with the value 12H
0402   12

0403   18    CLC            ;Clears the carry flag

0404   6D    ADC    $0500   ;Adds the contents of memory
0405   00                   ;location 0500H to the accumulator
0406   05

0407   85    STA    $F0     ;Saves the contents of the accumulator in
0408   F0                   ;location 00F0H

0409   60    RTS            ;Returns to MAC III monitor
```

Note that the "ORG" statement is an Assembler ***Directive*** *to define the memory address at which the assembled program is to start. In the absence of an "ORG" directive the default address 0400H will be assumed by the Cross Assembler.*

Examine the contents of locations 0500_H and $00F0_H$ before execution. Check these locations again after the program has been executed and verify that the contents of location $00F0_H$ are 12_H greater than location 0500_H.

7.5a **In the program for Worked Example 7.5, the addressing mode used by the instruction** "`ADC   $0500`" **is:**

a absolute

b immediate

c implied

d zero page

7.5b **Place the value $3A_H$ in location 0500_H. Run the program for Worked Example 7.5 and then examine the contents of memory location $00F0_H$. Enter the hexadecimal value which you find.**

7.6 Decimal Arithmetic

So far we have only considered programs which perform **binary** arithmetic. Such programs add or subtract binary (or hexadecimal) numbers, to give a binary (or hexadecimal) result. However, many problems involve the addition or subtraction of **decimal** numbers, and require a decimal result. This requires a considerable lengthening of programs for most microprocessors.

The 6502 is unlike many microprocessors in that it can perform decimal arithmetic **directly**, without the need for extra instructions.

A special flag within the Status Register is used to indicate to the Arithmetic and Logic Unit (ALU) the type of arithmetic that is required. This is the Decimal Flag (bit 3 of the status register).

When the Decimal Flag (D-Flag) is **set** (i.e. D=1), the ALU will perform **decimal** arithmetic. Conversely, when the D-Flag is **clear** the ALU performs **binary** (or hexadecimal) arithmetic.

There are two instructions which can be used to set/clear the D-Flag:

CLD	This instruction will **clear** the D-Flag, so the ALU will perform **binary** arithmetic. (You will recall that the 'CLD' instruction has been used previously in this manual, in programs which perform binary arithmetic.)
SED	This instruction will **set** the D-Flag, so the ALU will perform **decimal** arithmetic.

When operating in decimal mode, each 8-bit number is treated as two 4-bit codes, each code representing a decimal value between 0 and 9. A binary code of 0000_2 represents decimal value 0_{10}, a code of 0001_2 represents 1_{10}, and so on through to 1001_2 which represents 9_{10}.

Each byte therefore represents a two-digit decimal number, for example:

$0101\ 0100_2$ represents 54_{10}
$1001\ 1001_2$ represents 99_{10}

This way of representing decimal numbers is known as **Binary Coded Decimal** or **BCD**.

When working with BCD numbers, note that the 4-bit binary codes 1010_2 *through* 1111_2 *are invalid.*

7.6a **Enter the decimal value represented by the BCD number 01110010_2.**

7.6b **The BCD number which represents 42_{10} is:**

a 00101010_2
b 01000010_2
c 00100100_2
d 10100010_2

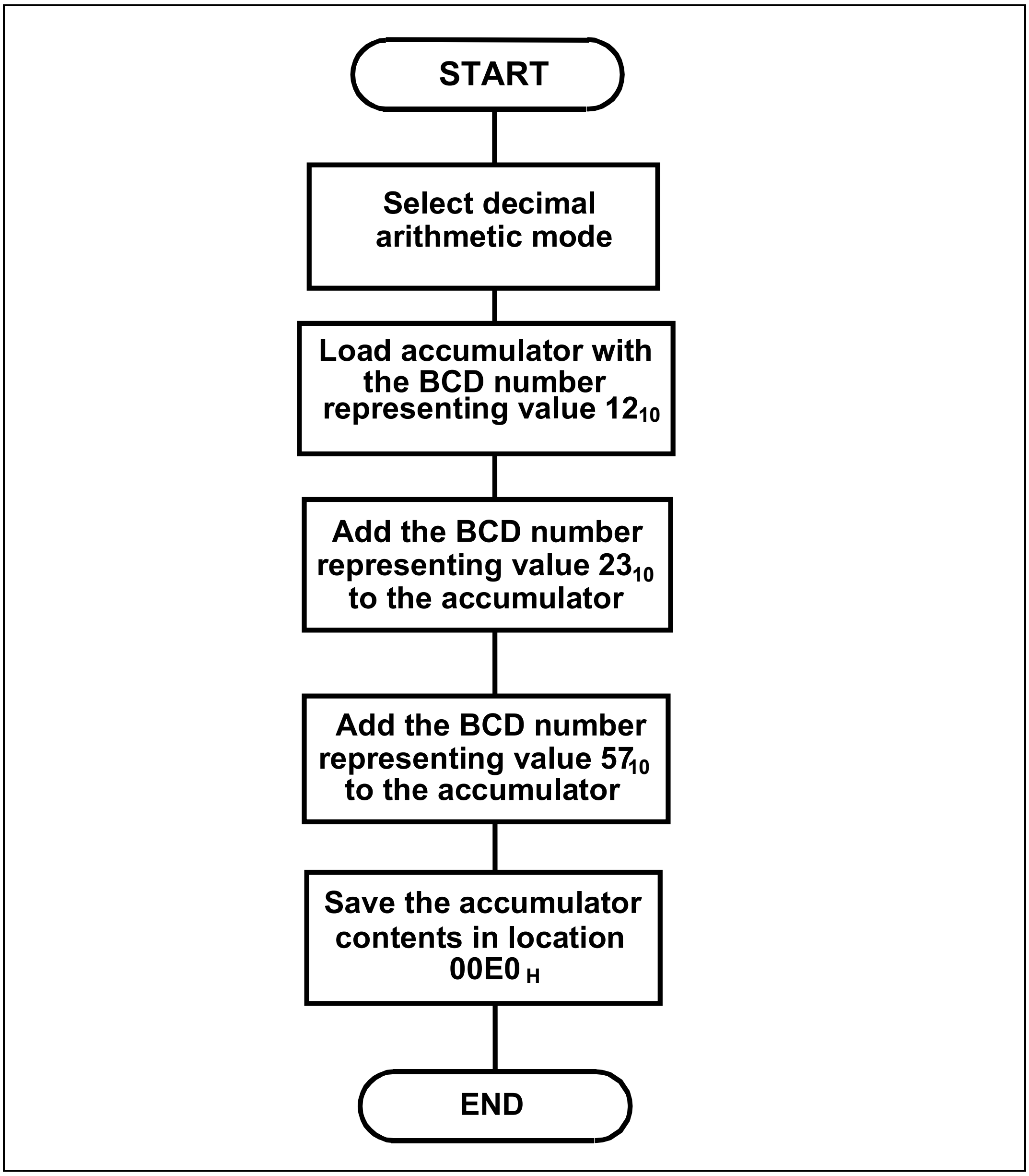
7.6c **The flag which is set to perform decimal arithmetic is the:**

a D-flag
b C-flag
c I-flag
d V-flag

7.7 Worked Example

Write a program which will perform the calculation below and place the result in location $00E0_H$:

$12_{10} + 23_{10} + 57_{10}$



The Assembly Language Program will be as opposite:

```
                ORG $0400     ;Defines the start address
0400    F8      SED           ;Sets the D-Flag so ALU will now perform
                              ;decimal arithmetic
0401    A9      LDA  #$12     ;Loads accumulator with the BCD number
0402    12                    ;representing the decimal value 12.

0403    18      CLC           ;Clears the carry flag
0404    69      ADC  #$23     ;Adds the BCD number representing the decimal
0405    23                    ;value 23, to the accumulator

0406    69      ADC  #$57     ;Adds the BCD number representing the decimal
0407    57                    ;value 57, to the accumulator

0408    85      STA  $E0      ;Saves the contents of the accumulator in
0409    E0                    ;location 00E0H
040A    60      RTS           ;Returns to MAC III monitor
```

Examine the contents of location $00E0_H$ before execution. Check this location again after the program has been executed and verify that it contains the BCD number representing 92_{10}.

Notice that when the 6502 performs decimal arithmetic with BCD numbers, the result itself is also a BCD number.

7.7a **In the program for Worked Example 7.7, the addressing mode used by the instruction** " `LDA   #$12` " **is:**

a absolute

b immediate

c implied

d zero page

7.7b **The program for Worked Example 7.7, is to be changed so that the result will be saved in location** $\mathbf{0500_H}$**. The instruction** " `STA   $E0`" **must be replaced by:**

a `STA   $05`

b `STA   $50`

c `STA   $0050`

d `STA   $0500`

7.8 Practical Assignment

Write a program, starting at location 0400_H, which will perform **binary** addition of the contents of memory locations 0050_H, 0051_H, and 0052_H. The result should be saved in memory location 1000_H.

Note: This requires binary arithmetic.

7.8a Place the value $2B_H$ in memory locations 0050_H, 0051_H, and 0052_H. Run your program for Practical Assignment 7.8 and enter the hexadecimal value you find in location 1000_H.

7.8b Modify your program for Practical Assignment 7.8 so that it will calculate the decimal sum of the contents of locations 0050_H, 0051_H, and 0052_H. Place the BCD number representing the decimal value 19_{10}, into memory locations 0050_H, 0051_H, and 0052_H. Run your modified program, then enter the decimal value represented by the BCD number which you find in location 1000_H.

Please Note: From now on, **binary arithmetic** will be used for all additions and subtractions in this manual unless otherwise stated.

7.9 The Subtract Instruction

The 6502 performs subtraction by means of the SBC (Subtract with Carry) instruction. This instruction will cause the contents of some specified memory location to be subtracted from the contents of the accumulator, **with borrow**. In arithmetic a "borrow" is the opposite of a "carry".

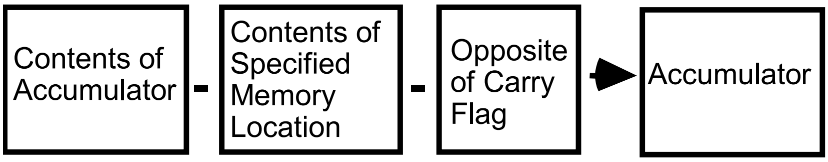
This means therefore that the **opposite** of the carry flag will be subtracted from the result.

This is not really as complicated as it sounds. Recall that the carry flag must be **cleared** prior to addition. In a similar way, the carry flag must be **set** prior to subtraction.

This is because it is the **opposite** of the carry flag which is subtracted, so to subtract 0 the carry flag must be at 1.

The reason for this structure in 6502 Addition and Subtraction instructions is that it is required for calculations which exceed 8-bits. The carry flag allows a carry (or a borrow) between one part of a multiple precision calculation and the next.

The Subtract instruction can be summarized thus:

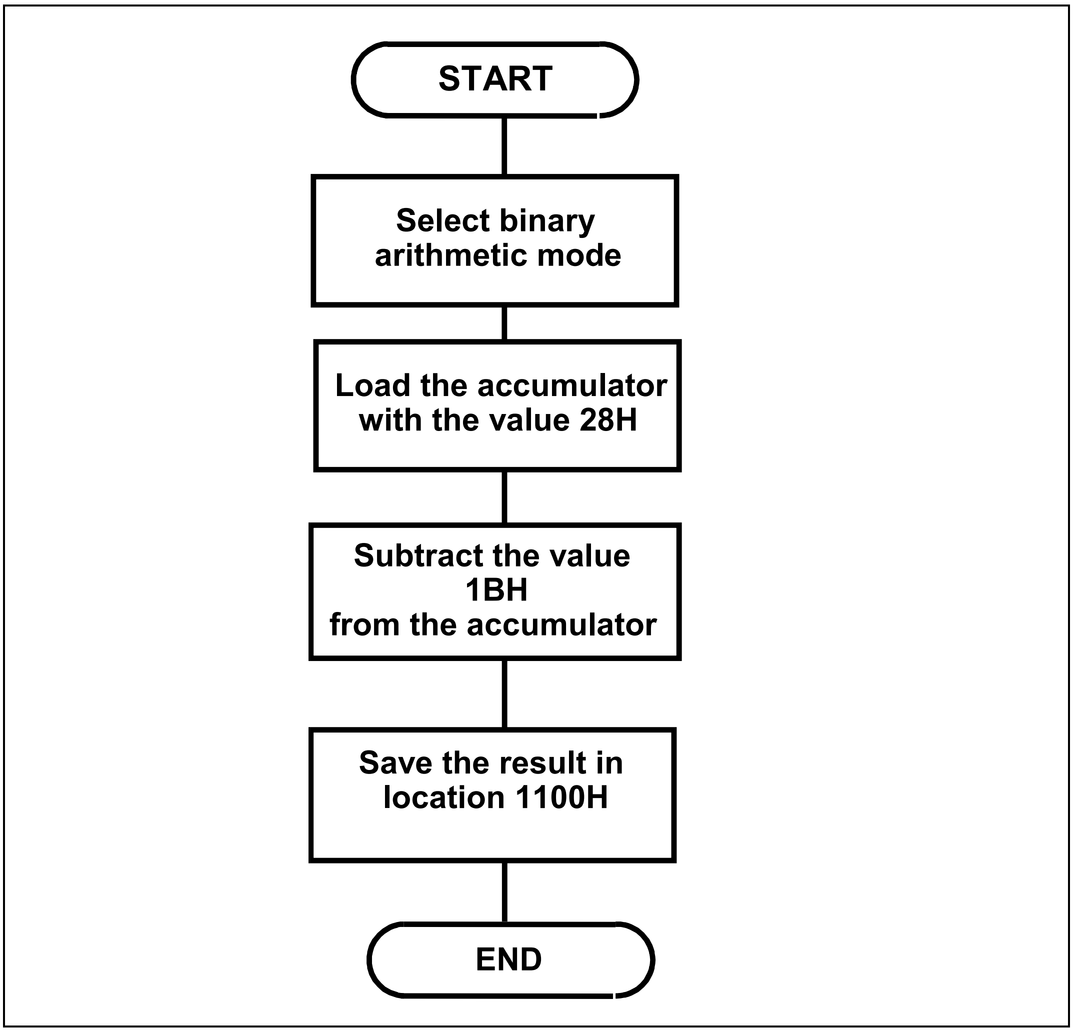


Now, you will have noticed that it is only necessary to clear the carry flag **once**, even if a number of ADC instructions follow.

This is because part of the action of the ADC instruction is to **clear** the Carry Flag - **unless the result exceeds 8-bits**. In a similar way, it is only necessary to **set** the Carry Flag **once** prior to a number of subtractions since the SBC instruction will itself **set** the Carry Flag - **unless a "borrow" is generated**.

7.10 Worked Example

Write a program which will subtract the value $1B_H$ from the value 28_H and save the result in location 1100_H.



The Assembly Language Program will be:

```
               ORG  $0400   ;Defines start address
0400   D8      CLD          ;Selects binary arithmetic mode
0401   A9      LDA  #$28    ;Loads accumulator with the
0402   28                   ;value 28H
0403   38      SEC          ;Sets the carry flag
0404   E9      SBC  #$1B    ;Subtracts the value 1BH
0405   1B                   ;from the accumulator
0406   8D     STA  $1100    ;Saves the result in location 1100H
0407   00
0408   11
0409   60      RTS          ;Returns to MAC III monitor
```

7.10a **Run the program for Worked Example 7.10. Examine the contents of location 1100_H. Enter the hexadecimal value you find at this location.**

7.10b **Modify the program for Worked Example 7.10 so that it will subtract $4D_H$ from 71_H. Run your program and then examine the contents of location 1100_H. Enter the hexadecimal value you find at this location.**

7.11 Practical Assignment

Write a program which will add the BCD number representing the value 21_{10} to the BCD number at location 0070_H and then subtract the BCD number at location 0510_H from the result. The final result must be stored as a BCD number in location 0520_H.

Note: This problem requires decimal arithmetic.

7.11a **Place the BCD number representing 32_{10} in memory location 0070_H and the BCD number representing 34_{10} in location 0510_H. Run your program for Practical Assignment 7.11 and enter the decimal value represented by the BCD number at location 0520_H.**

7.11b **Modify your program for Practical Assignment 7.11 so that it will perform binary arithmetic. Place the value $3E_H$ in memory location 0070_H and the value 42_H in location 0510_H. Run your modified program and enter the hexadecimal value you find in location 0520_H.**

Student Assessment 7

1. The 6502 addressing mode in which no operand bytes are required is called:

a Implied Addressing

b Immediate Addressing

c Absolute Addressing

d Zero Page Addressing

2. In Zero Page addressing, the number of operand bytes required is:

a 0

b 1

c 2

d 3

3. In Absolute addressing, the total number of bytes for an instruction is:

a 0

b 1

c 2

d 3

4. The 6502 Assembly Language instruction "`LDA   $60`" **will load the accumulator:**

a with the value 60_H

b with the value 60_{10}

c from location 0060_H

d from location 0060_{10}

Student Assessment 7 Continued ...

5. The 6502 Assembly Language instruction which causes the microprocessor to perform decimal arithmetic is:

a `CLC`

b `CLD`

c `SEC`

d `SED`

6. When a 6502 Subtract instruction is executed, the Carry Flag:

a is ignored

b shows any Borrow

c is subtracted from the result

d is saved in the Accumulator

7. The 6502 Assembly Language instruction "`SBC   $1200`" **will subtract :**

a the value 1200_H from the accumulator

b the value 12_{10} from the accumulator

c the contents of location 1200_H from the accumulator

d the contents of location 0012_{10} from the accumulator

8. The machine code for the 6502 Assembly Language instruction "`SBC   #$3C`" **is:**

a A9 3C

b A9 00 3C

c E9 3C

d E9 00 3C

Continued ...

Student Assessment 7 Continued ...

9. The 6502 Assembly Language instructions required to subtract the contents of location 0080_H from the Accumulator are:

a
```
CLC
SBC #80
```

b
```
CLC
SBC $80
```

c
```
SEC
SBC #80
```

d
```
SEC
SBC $80
```

10. The program section:

```
SED
LDA #$48
CLC
ADC #$22
```

a Performs decimal addition of 48_{10} and 22_{10}

b Performs binary addition of 48_H and 22_H

c Performs decimal subtraction of 22_{10} from 48_{10}

d Performs binary subtraction of 22_H from 48_H

Chapter 8 Negative Binary Numbers

Objectives of this Chapter

Having studied this chapter you will be able to:

- Form the 1's and 2's complements of binary numbers.
- Use complementary arithmetic to perform subtraction with binary and hexadecimal numbers.
- Use complementary arithmetic to represent negative binary and hexadecimal numbers.

Introduction

In the everyday decimal numbering system, a negative number is denoted by a **minus sign**. This is called "Sign Magnitude Form". This system can also be used for indicating negative binary numbers but unfortunately the microprocessor cannot understand a minus sign. Clearly then an alternative method is required. All microprocessors use **Complementary Arithmetic** to manipulate negative numbers.

8.1 Complementary Arithmetic

For any binary number there are **two** possible **complements**:

1's Complement: Found by simply **inverting** each bit.

For example: The 1's Complement of 1011_2 is 0100_2,
so -1011_2 is 0100_2 in 1's complement notation.

2's Complement: Found by adding 1 to the 1's complement.

For example: The 1's Complement of 1011_2 is 0100_2;
the 2's complement of 1011_2 is $0100_2 + 1_2 = 0101_2$
so -1011_2 is 0101_2 in 2's complement notation.

Almost all microprocessors use 2's complement notation. This is important in the understanding of relative addressing, which will be explained in a subsequent chapter.

Consider 0110_2:

1's complement of 0110_2 is 1001_2. Taking the 1's complement again gives 0110_2.

Similarly, the 2's complement of 0110_2 is $1001_2 + 1_2 = 1010_2$

Taking the 2's complement again gives $0101_2 + 1_2 = 0110_2$.

So a complementary number may be converted back to an ordinary number by simply taking the complement.

8.2 Worked example

Evaluate 1011101_2 - 101011_2 using 8-bit 2’s complements.

Solution:

Although 8-bit 2’s complements have been specified, neither of the values have 8 bits. It is most important in complementary arithmetic **not** to suppress leading zero’s, since these become 1’s when complemented. So the first step is to insert as many leading zeros as necessary in order to make both values 8-bits:

So the problem becomes **0**1011101$_2$ - **00**101011$_2$.

Now, - **00**101011$_2$ must be converted to 2’s complement form:

First convert to 1’s complements:

- 00101011_2 = + 11010100_2

Then add one : + 11010100_2 + 1_2 = + 11010101_2

So the problem now becomes 01011101_2 + 11010101_2

```
          01011101  +
          11010101
          --------
   1      00110010
          --------
   └ Carry out
```

So 1011101_2 - 101011_2 = **00110010_2**

The carry out can be **ignored**. It actually indicates the sign of the result. If the result had been negative then the carry out would have been zero.

8.2a **The 1's complement of 01001011_2 is:**

a 01001011_2

b 01001100_2

c 10110100_2

d 10110101_2

8.2b **The 2's complement of 01001011_2 is:**

a 01001011_2

b 01001100_2

c 10110100_2

d 10110101_2

8.2c **110001_2 - 11111_2 is:**

a 10010_2

b 10011_2

c 11001_2

d 11010_2

8.3 Worked Example

Evaluate $2B_H$ - 47_H using 8-bit 2's complements.

Solution:

In this case there are two new problems: the values are quoted in hexadecimal and the result will be negative. The problem can be simply converted to binary thus:

$$2B_H - 47_H = 0010\ 1011_2 - 0100\ 0111_2$$

Now, as before - $0100\ 0111_2$ must be converted to 2's complement form:

First convert to 1's complements:

$$-\ 0100\ 0111_2 = +\ 1011\ 1000_2$$

Then add one : + $1011\ 1000_2 + 1_2 = +\ 1011\ 1001_2$

So the problem now becomes $0010\ 1011_2 + 1011\ 1001_2$

```
     0010 1011
     1011 1001  +
     _________
0    1110 0100
     _________
└ Carry out
```

Now, the Carry Out is **zero** so this result is **negative**. It is expressed in 2's complement form and may be converted to sign magnitude form by simply taking the 2's complement:

The 1's complement is $0001\ 1011_2$ so the 2's complement is $0001\ 1011_2 + 1_2 = 0001\ 1100_2$.

Thus: $2B_H - 47_H = 0010\ 1011_2 - 0100\ 0111_2$

$$= -\ 0001\ 1100_2 = \mathbf{-\ 1C_H}$$

8.3a **The 1's complement of $3E_H$ is:**

a $3F_H$

b $C0_H$

c $C1_H$

d $C2_H$

8.3b **The 2's complement of 60_H is:**

a $9F_H$

b $A0_H$

c BF_H

d $C0_H$

8.3c **$3E_H$ - $0D_H$ is:**

a 31_H

b $6E_H$

c $E3_H$

d $F3_H$

8.4 Worked Example

Express -1_H using 8-bit 2’s complements.

Solution:

1’s complement: $-0000\ 0001_2 = +1111\ 1110_2$

2’s complement: $+1111\ 1110_2 + 1_2 = +1111\ 1111_2$

that is: FF_H

8.4a The 8-bit 2's complement form of -21_H is:

a $3E_H$
b $3F_H$
c DE_H
d DF_H

8.4b Enter the 8-bit 2's complement form of -55_H (in hexadecimal).

8.5 Worked Example

Express -2_H using 8-bit 2's complements.

Solution:

$-2_H = -0000\ 0010_2$

1's complement: $-0000\ 0010_2 = +1111\ 1101_2$

2's complement: $+1111\ 1101_2 + 1_2 = +\mathbf{1111\ 1110_2}$

that is: FE_H

If you continue to find -3_H, -4_H, -5_H and so on you will discover the values FD_H, FC_H and FB_H respectively. So, as the negative value increases, the count in hexadecimal decreases.

8.5a **Enter the 8-bit 2's complement form of $-B_H$ (in hexadecimal).**

8.5b **$39_H - 62_H$ is:**

$\boxed{a}$ -28_H

$\boxed{b}$ -29_H

$\boxed{c}$ $-D7_H$

$\boxed{d}$ $-D8_H$

Student Assessment 8

1. The 1's complement of 0010 1110_2 is:

- a 0010 1110_2
- b 1101 0001_2
- c 1101 0010_2
- d 1101 0011_2

2. The 2's complement of 0110 0111_2 is:

- a 0110 0111_2
- b 0110 1000_2
- c 1001 1000_2
- d 1001 1001_2

3. The 2's complement of a binary number is found by:

- a inverting each bit of the binary number.
- b adding 1 to the 1's complement.
- c adding 2 to the 1's complement.
- d subtracting 1 from the 1's complement.

4. The value -0011 0111_2 can be represented using 8-bit 2's complements as:

- a +0011 0111_2
- b +0011 1001_2
- c +1100 1000_2
- d +1100 1001_2

Continued ...

Student Assessment 8 Continued ...

5. The value -37_H can be represented using 8-bit 2's complements as:

a $+B8_H$

b $+B9_H$

c $+C8_H$

d $+C9_H$

6. The result of the subtraction $0100\ 1111_2$ - $0010\ 1101_2$ is:

a $0010\ 0010_2$

b $0010\ 0011_2$

c $1101\ 0010_2$

d $1101\ 0011_2$

7. The result of the subtraction 69_H - $4C_H$ is:

a $1C_H$

b $1D_H$

c $B5_H$

d $E3_H$

Chapter 9 Programs with Loops

Objectives of this Chapter

Having studied this chapter you will be able to:

- Describe the different types of program loop structure.
- Describe the use of the conditional and unconditional JUMP and BRANCH instructions.
- Explain the mechanism and use of 6502 relative addressing.
- Describe the function and operation of the following 6502 flags:
  - Carry Flag
  - Zero Flag
- Write programs which use the conditional and unconditional JUMP and BRANCH instructions.

Equipment Required for this Chapter

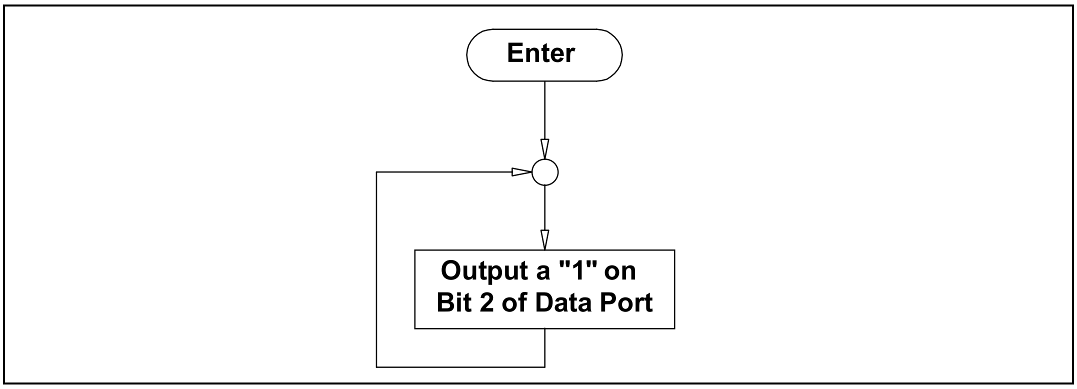
- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- 6502 Instruction Set Reference Manual.
- MAC III 6502 User Manual.

Introduction

Often it will be necessary to use a program **loop** to **repeat** a section of a program a number of times. There are three main types of program loop:

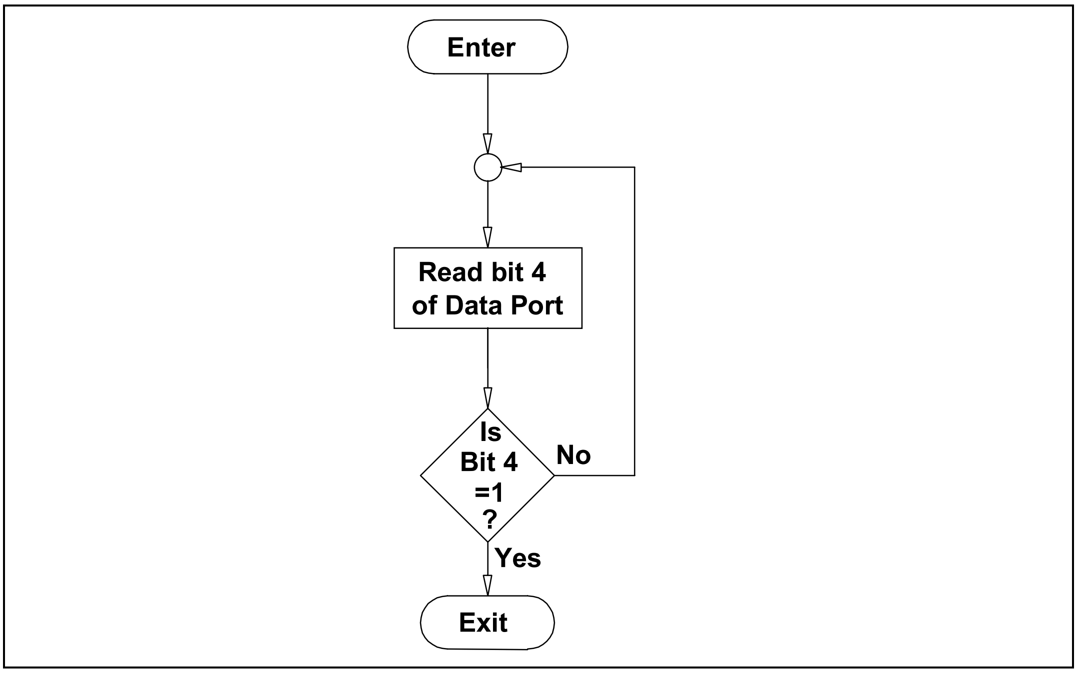
1. **Repeating a program section indefinitely**

For example: Output a “1” on bit 2 of a data port indefinitely.



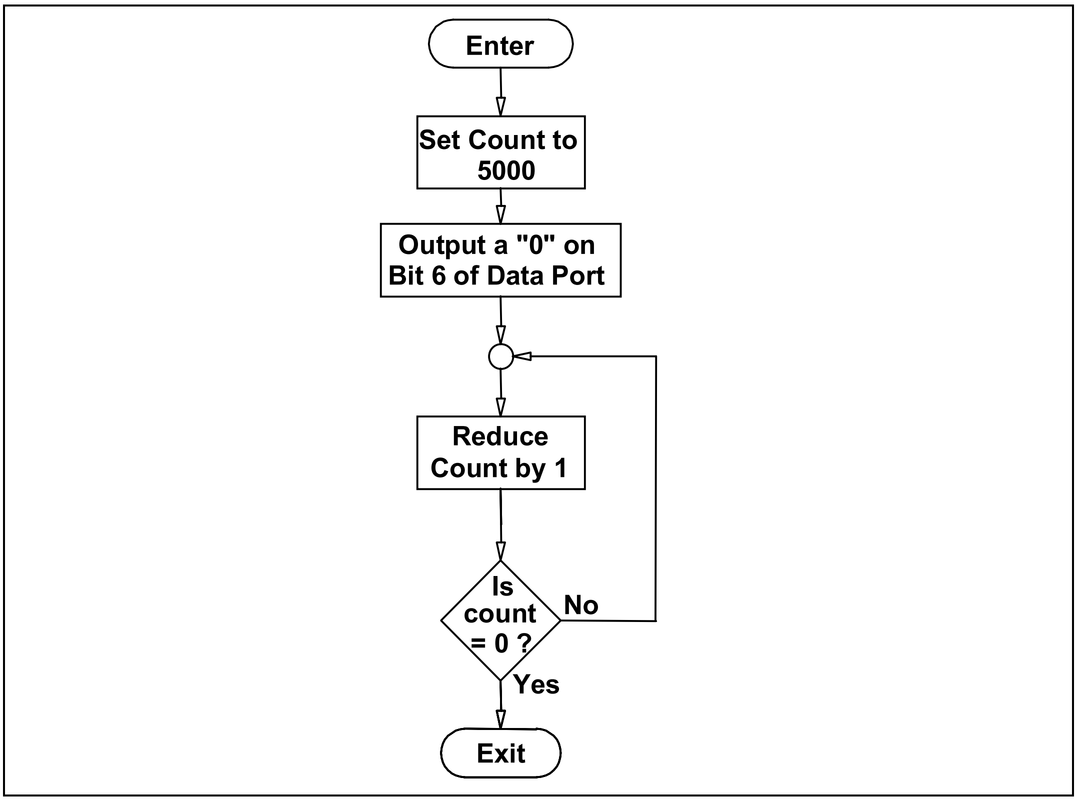
2. **Repeating a program section until some predetermined condition becomes true.**

For example: Waiting for a “1” to be input at bit 4 of a data port.



3. Repeating a program section for a predetermined number of passes.

For example: Output a “0” on bit 6 of a data port for the time it takes to repeat a loop 5000 times.



If, in this example, each pass through the loop were to take **1µs**, a "0" would be output on bit 6 of the data port for **5ms**.

In order to write assembly language programs with loops, it will be necessary to use JUMP and BRANCH instructions. These can be conditional or unconditional.

9.1 JUMP and BRANCH Instructions

These instructions cause program execution to be continued from some point other than the next location in sequence.

There are two types of JUMP/BRANCH instruction:

Unconditional JUMP/BRANCH- "*Always* JUMP/BRANCH "

Conditional JUMP/BRANCH - "*Only* JUMP/BRANCH *if* some condition is true"

In 6502 Assembly Language, a conditional jump is referred to as a BRANCH. An unconditional jump is simply referred to as a JUMP.

You have already used the Absolute JUMP instruction (JMP). Recall that the Absolute Address of the destination for the Jump is specified by the two bytes following the opcode byte, low byte first.

The 6502 BRANCH instructions all use **relative** addressing.

9.2 Relative addressing

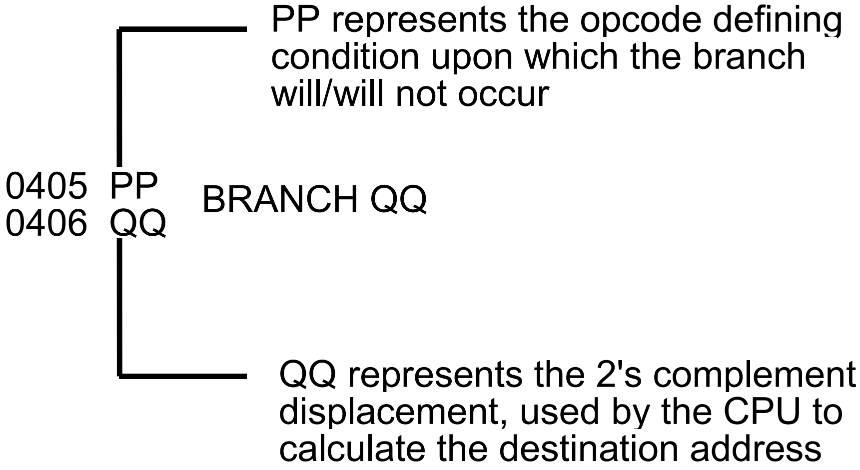
In this mode of addressing, the destination for the BRANCH is not specified absolutely (for example "location $021B_H$") but is expressed in terms of the number of locations further on (or back) in the program (for example "8 locations further on").

The 6502 Branch instructions have two parts:

1. The operator code which defines the condition on which branching will/will not occur.
2. A 2's complement displacement (or offset) which specifies the destination in terms of the number of bytes forward or backward.

The displacement is added to the Program Counter to produce the destination address.

For example:



```
0405  PP    BRANCH QQ
0406  QQ
```

PP represents the opcode defining condition upon which the branch will/will not occur

QQ represents the 2's complement displacement, used by the CPU to calculate the destination address

Forward Branching

Consider the following generalized instruction:

```
0400 PP   BRANCH  05
0401 05
```

If the branch is taken, then 05_H is added to the Program Counter to calculate the destination address. Now, it is important to note that the program counter will already be pointing to the next instruction in sequence (i.e. 0402_H). The Destination Address can therefore be calculated thus:

0402_H	+	05_H	=	0407_H
|		|		|
|		|		|
Program Counter	+	2's Complement Displacement	=	Destination Address

Backward Branching

Consider the following generalized instruction:

```
0400 PP   BRANCH  FA
0401 FA
```

If the branch is taken, then FA_H is added to the Program Counter to calculate the destination address. Again, the program counter will be pointing to the next instruction (i.e. 0402_H).

The Destination Address can therefore be calculated thus:

$FA_H = +1111\ 1010_2$

Taking the 1's complement: $+1111\ 1010_2 = -0000\ 0101_2$
Adding 1 to form the 2's complement:

$0000\ 0101_2 + 1_2 = -0000\ 0110_2$
$-0000\ 0110_2 = -06_H$

So the calculation becomes:

$$\begin{array}{ccccc} 0402_H & + & -06_H & = & 03FC_H \\ | & & | & & | \\ | & | & | & & \\ \text{Program} & + & \text{2's Complement} & = & \text{Destination} \\ \text{Counter} & & \text{Displacement} & & \text{Address} \end{array}$$

Range of Relative Addressing

The displacement for 6502 BRANCH instructions is always 8 bits in length. The largest possible positive offset will therefore be $7F_H$ ($0111\ 1111_2$) which is 127_{10}.

This means that it is not possible to perform a relative BRANCH more than 127_{10} locations in the forward direction.

Now, the largest possible negative offset will be 80_H ($1000\ 0000_2$).

$$\begin{aligned} 80_H &= +\ 1000\ 0000_2 \\ &= -\ 0111\ 1111_2 \text{ (1's complement)} \\ &= -\ 1000\ 0000_2 \text{ (2's complement)} \\ &= -\ 80_H \\ &= -\ 128_{10} \end{aligned}$$

So the limit of a backward relative BRANCH is 128_{10} locations.

In this Section, we have seen how the Destination Address for a BRANCH instruction can be calculated by adding the Program Counter contents to the 2's Complement Displacement. Later on we will calculate the 2's Complement Displacement for a BRANCH instruction, given the current Program Counter position and the required Destination Address.

9.2a **The types of 6502 instructions which allow program execution to continue from a point other than the next location in sequence are called:**

a Sequence or Over-ride instructions.
b Skip or Goto instructions.
c Mark Place instructions.
d Jump or Branch instructions.

9.2b **In relative addressing, the destination is specified by:**

a a 2's complement displacement.
b an absolute address.
c the contents of the status register.
d the contents of the X and Y registers.

9.3 Conditional Instructions

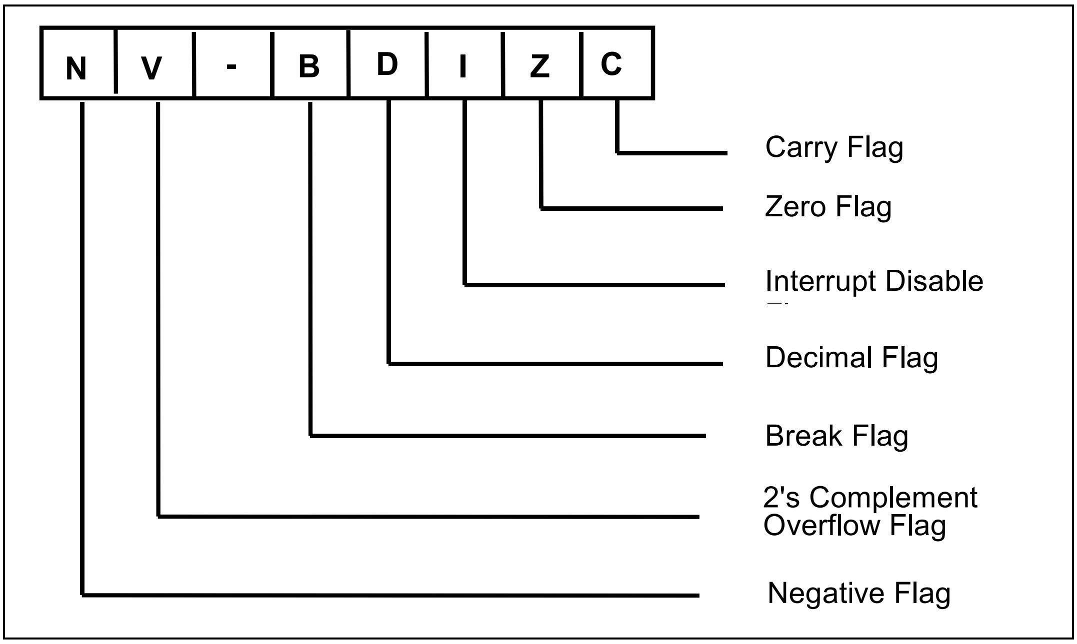
A conditional BRANCH is only taken if some predetermined condition is true. Otherwise the next instruction in sequence is executed. These instructions are very important since they allow the microprocessor to take decisions.

The conditions which these instructions test are the states of individual bits within the **status register** (or Flag register).

6502 Status Register

Each bit (or "flag") within the Status Register (or "Condition Code Register") is a single flip-flop which can store a 0 or a 1. These flags indicate the nature of the result of the last Arithmetic operation. Many instructions will affect various flags.

The 6502 Status Register has 7 flags thus:



We shall only consider two of these flags for the present - the Carry Flag and the Zero Flag.

Carry Flag

This flag is set (i.e. = 1) if the result of the last arithmetic operation is greater than 8 bits. For example:

If $3A_H$ is added to 47_H the result is 81_H and there is no carry out:

$$\begin{array}{rl} 3A_H & \\ 47_H & + \\ \hline 81_H & \\ \hline \end{array} \qquad \begin{array}{rl} 0011\ \ 1010_2 & \\ 0100\ \ 0111_2 & + \\ \hline 1000\ \ 0001_2 & \\ \hline \end{array}$$

So the carry flag is **cleared** (C = 0).

However, if $3A_H$ is added to $E7_H$ the result is **1**21_H. Thus a carry out is generated:

$$\begin{array}{rl} 3A_H & \\ E7_H & + \\ \hline 121_H & \\ \hline \end{array} \qquad \begin{array}{c|rl} & 0011\ \ 1010_2 & \\ & 1110\ \ 0111_2 & + \\ \hline \underset{\uparrow \text{ Carry Out}}{1} & 0010\ \ 0001_2 & \\ \hline \end{array}$$

and the carry flag is **set** (C = 1).

The carry flag is also used as a "borrow" flag when performing subtraction.

Zero Flag

This flag is set (i.e. = 1) if the result of the last operation was zero. For example, if the microprocessor subtracts 34_H from 34_H the result is 00_H and the zero flag is **set** (Z=1). If 34_H is added to 34_H the result is 68_H which is non-zero so the zero flag is **cleared** (Z=0).

The action of the Zero and Carry flags can be summarized thus:

EQ	Result Equal to Zero	(Z=1)
NE	Result Not Equal to Zero	(Z=0)
CS	Carry Flag Set	(C=1)
CC	Carry Flag Cleared	(C=0)

Now, refer to the 6502 Instruction Set Reference Manual for some of the instructions you have met so far. Note how the Zero and Carry Flags are affected by each instruction:

Instruction	Zero Flag	Carry Flag
LDA	Set if accumulator is loaded with zero, otherwise cleared	Not affected
STA	Not affected	Not affected
ADC	Set if result is zero, otherwise cleared	Set if a carry is generated, otherwise cleared
SBC	Set if result is zero, otherwise cleared	Cleared if a Borrow is generated, otherwise set
RTS	Not affected	Not affected
JMP	Not affected	Not affected

9.3a **After the 6502 has subtracted $4A_H$ from 67_H, the Zero (Z) and Carry (C) Flags will be:**

- a C=0, Z=0.
- b C=0, Z=1.
- c C=1, Z=0.
- d C=1, Z=1.

9.3b **After the 6502 has added 52_H to 67_H, the Zero (Z) and Carry (C) Flags will be:**

- a C=0, Z=0.
- b C=0, Z=1.
- c C=1, Z=0.
- d C=1, Z=1.

9.3c **After the 6502 has added 75_H to $8E_H$, the Zero (Z) and Carry (C) Flags will be:**

a C=0, Z=0.

b C=0, Z=1.

c C=1, Z=0.

d C=1, Z=1.

9.3d **After the 6502 has subtracted 72_H from 72_H, the Zero (Z) and Carry (C) Flags will be:**

a C=0, Z=0.

b C=0, Z=1.

c C=1, Z=0.

d C=1, Z=1.

9.4 Conditional Branch Instructions

Each of the Conditional Branch Instructions will test for a different flag set (=1) or clear (=0). If the condition is true, then the displacement is added to the Program Counter and execution continues from a point other than the next instruction. However, if the condition is **not** true execution will continue with the next instruction in memory.

The Conditional Branch Instructions which test the Carry and Zero Flags are:

BEQ	Branch if Result Equal to Zero	(Z=1)
BNE	Branch if Result Not Equal to Zero	(Z=0)
BCS	Branch if Carry Flag Set	(C=1)
BCC	Branch if Carry Flag Cleared	(C=0)

The Negative and 2's Complement Overflow flags may also be tested by corresponding branch instructions. However, we shall concentrate upon the Z- and C-Flags initially.

Note that for each Conditional Branch Instruction a **displacement** must be specified, which will be added to the Program Counter if the tested condition is true. This 2's complement displacement is calculated by counting forwards or backwards from the current Program Counter position to the Destination Address (the address you wish to branch to).

*Remember that the program counter will be pointing to the start of the **next instruction** in memory after the branch instruction.*

In 6502 assembly language, we use a **label** to identify the destination for a branch.

Example 1:

Address	Machine Code	Assembly Lang.	Comments
0400	F0	BEQ DEST	;Branches 3 locations forward from
0401	**03**		;current Program Counter value if
			;zero flag is set (ie if Z = 1)
0402	A9	LDA #$AA	
0403	AA		
0404	18	CLC	
0405	65	DEST: ADC $F0	;Destination for BEQ instruction
0406	F0		

In the above example, the destination for the branch is identified by label 'DEST' in the left-hand column of the assembly language. The displacement for the branch instruction is calculated by counting forwards from location 0402_H to the destination address (0405_H). Three bytes are counted, so the displacement required at address 0401_H is $\mathbf{03_H}$.

Example 2:

Address	Machine Code	Assembly Lang.	Comments
0500	18	LOOP: CLC	;Destination for BCC instruction
0501	6D	ADC $0600	
0502	00		
0503	06		
0504	90	BCC LOOP	;Branches 6 locations backwards
0505	**FA**		;(FA$_H$ = -6$_H$) from current Program
			;Counter value if carry flag is
			;clear (ie if C = 0)
0506	60	RTS	

In this second example, the label ‘LOOP’ in the left-hand column of the assembly language identifies the destination for the branch. The displacement for the branch instruction is calculated by counting backwards from location 0506$_H$ to the destination address (0500$_H$). Six bytes are counted, so the displacement required at location 0505$_H$ is the 2's complement value representing - 6$_H$. This value is **FA$_H$**.

*The 6502 Cross Assembler will calculate the 2's complement displacement for each branch instruction **automatically** from the labels in your assembly language program. Note the colon (:) which appears after each label in the left-hand column of the assembly language; this is a requirement of the Cross Assembler.*

9.4a The 6502 assembly language instruction "`BNE WAIT`" will branch to the location identified by the label '`WAIT`' if:

a the Carry Flag is set (C=1).

b the Carry Flag is clear (C=0).

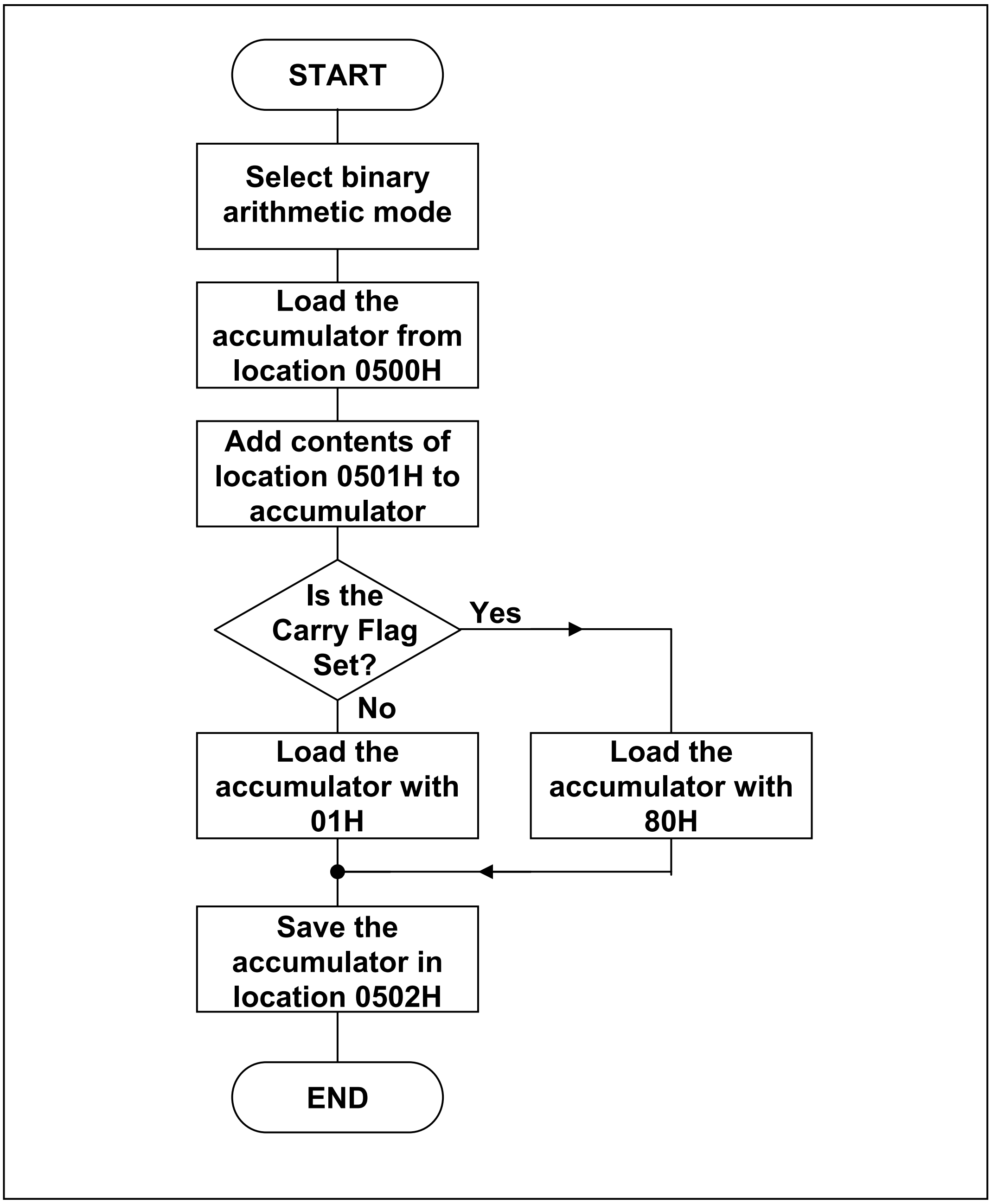
c the Zero Flag is set (Z=1).

d the Zero Flag is clear (Z=0).

9.5 Worked Example

Write a program which will add the contents of locations 0500_H and 0501_H. The value 80_H should be placed in location 0502_H if the result exceeds FF_H, otherwise 01_H should be placed in location 0502_H.

This problem requires the carry flag to be tested following the addition and then a marker value to be saved to indicate the status of the result.



The Assembly Language Program will be:

```
                  ORG $0400  ;Defines the start address
0400  D8          CLD        ;Selects binary arithmetic mode
0401  AD          LDA $0500  ;Loads the accumulator from location 0500H
0402  00
0403  05
0404  18          CLC
0405  6D          ADC $0501  ;Adds the contents of location 0501H to
0406  01                     ;the accumulator
0407  05
0408  B0          BCS CSET   ;Is the Carry Flag Set ?
0409  06
040A  A9          LDA #$01   ;C=0 so load accumulator with the marker
040B  01                     ;value 01H
040C  8D          STA  $0502 ;Save marker value in location 0502H
040D  02
040E  05
040F  60          RTS        ;Returns to MAC III system
0410  A9    CSET: LDA #$80   ;C=1 so load accumulator with the marker
0411  80                     ;value 80H
0412  8D          STA  $0502 ;Save marker value in location 0502H
0413  02
0414  05
0415  60          RTS        ;Returns to MAC III system
```

9.5a **Load the above program into the MAC III and then place the following values into MAC III memory:**

Location	**Contents**
0500_H	**12_H**
0501_H	**34_H**

Run the program and examine the contents of location 0502_H. Enter the hexadecimal value which you find.

9.5b **With the above program still loaded into MAC III memory, modify the following locations as indicated below:**

Location	**Contents**
0500_H	**AB_H**
0501_H	**CD_H**

Run the program again and examine the contents of location 0502_H. Enter the hexadecimal value which you now find.

The assembly language program for Worked Example 9.5 could have been written in a slightly different way, avoiding the need for repetition of the STA and RTS instructions thus:

```
                  ORG $0400  ;Defines the start address
0400  D8          CLD        ;Selects binary arithmetic mode
0401  AD          LDA $0500  ;Loads the accumulator from location 0500H
0402  00
0403  05
0404  18          CLC
0405  6D          ADC $0501  ;Adds the contents of location
0406  01                     ;0501H to the accumulator
0407  05
0408  B0          BCS CSET   ;Is the Carry Flag Set ?
0409  05
040A  A9          LDA #$01   ;C=0 so load accumulator with
040B  01                     ;the marker value 01H
040C  4C          JMP SAVE   ;Jump to save marker value instruction
040D  11
040E  04
040F  A9  CSET:  LDA #$80   ;C=1 so load accumulator with the marker
0410  80                     ;value 80H
0411  8D  SAVE:  STA $0502  ;Save marker value in location
0412  02                     ;0502H
0413  05
0414  60          RTS        ;Returns to MAC III system
```

9.5c **Load this modified program into the MAC III and place the following values in MAC III memory:**

Location	**Contents**
0500$_H$	**56$_H$**
0501$_H$	**78$_H$**

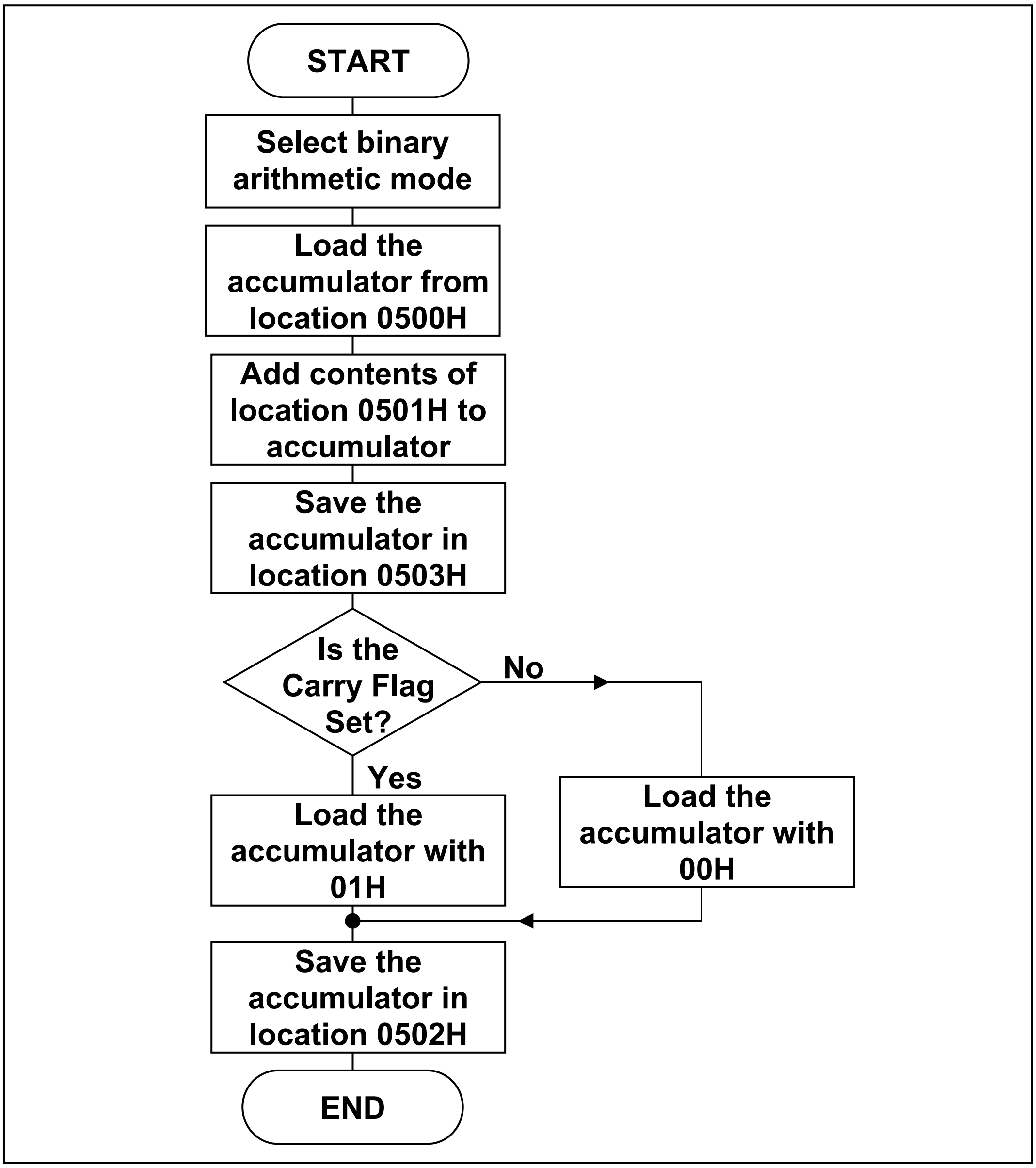
Run the program and examine the contents of location 0502$_H$. Enter the hexadecimal value which you find.

9.6 Worked Example

Write a program which will add the contents of locations 0500_H and 0501_H. The most significant byte of the result should be stored in location 0502_H and the least significant byte in location 0503_H. Now, consider the largest possible values:
$FF_H + FF_H = 01\ FE_H$

So the most significant byte can **only be 00_H or 01_H**.

The program must perform the addition, save the least significant byte and then test the carry flag to determine whether the most significant byte is 00_H or 01_H.



The Assembly Language program will be:

```
                ORG $0400  ;Defines the start address
 0400 D8        CLD        ;Selects binary arithmetic mode
 0401 AD        LDA $0500  ;Loads the accumulator from location 0500H
 0402 00
 0403 05
 0404 18        CLC
 0405 6D        ADC $0501  ;Adds the contents of location
 0406 01                   ;0501H to the accumulator
 0407 05
 0408 8D        STA $0503  ;Saves the least significant byte in
 0409 03                   ;location 0503H
 040A 05
 040B B0        BCS CSET   ;Is the Carry Flag Set ?
 040C 06

 040D A9        LDA #$00   ;C=0 so load accumulator with the value 00H
 040E 00
 040F 8D        STA $0502  ;Save most significant byte in location 0502H
 0410 02
 0411 05
 0412 60        RTS        ;Returns to MAC III system

 0413 A9  CSET: LDA #$01   ;C=1 so load accumulator with the value 01H
 0414 01
 0415 8D        STA $0502  ;Save most significant byte in location 0502H
 0416 02
 0417 05

 0418 60        RTS        ;Returns to MAC III system
```

9.6a **Use the program for Worked Example 9.6 to calculate $67_H + 89_H$. Enter the result you find.**

9.6b **Use the program for Worked Example 9.6 to calculate $CD_H + EF_H$. Enter the result you find.**

9.7 Practical Assignment

Write a program which will examine the contents of location 0500_H. If the contents are 00_H, location $00FF_H$ should be loaded with 80_H. If the contents are non-zero, location $00FF_H$ should be loaded with $7F_H$.

9.7a Load your program for Practical Assignment 9.7 into the MAC III. Place the value 56_H in memory location 0500_H. Run your program and then examine the contents of location $00FF_H$. Enter the hexadecimal value which you find.

9.7b With your program for Practical Assignment 9.7 still loaded in MAC III memory, now place the value 00_H in memory location 0500_H. Run your program again and examine the contents of location $00FF_H$. Enter the hexadecimal value which you find.

9.8 Loop Counters

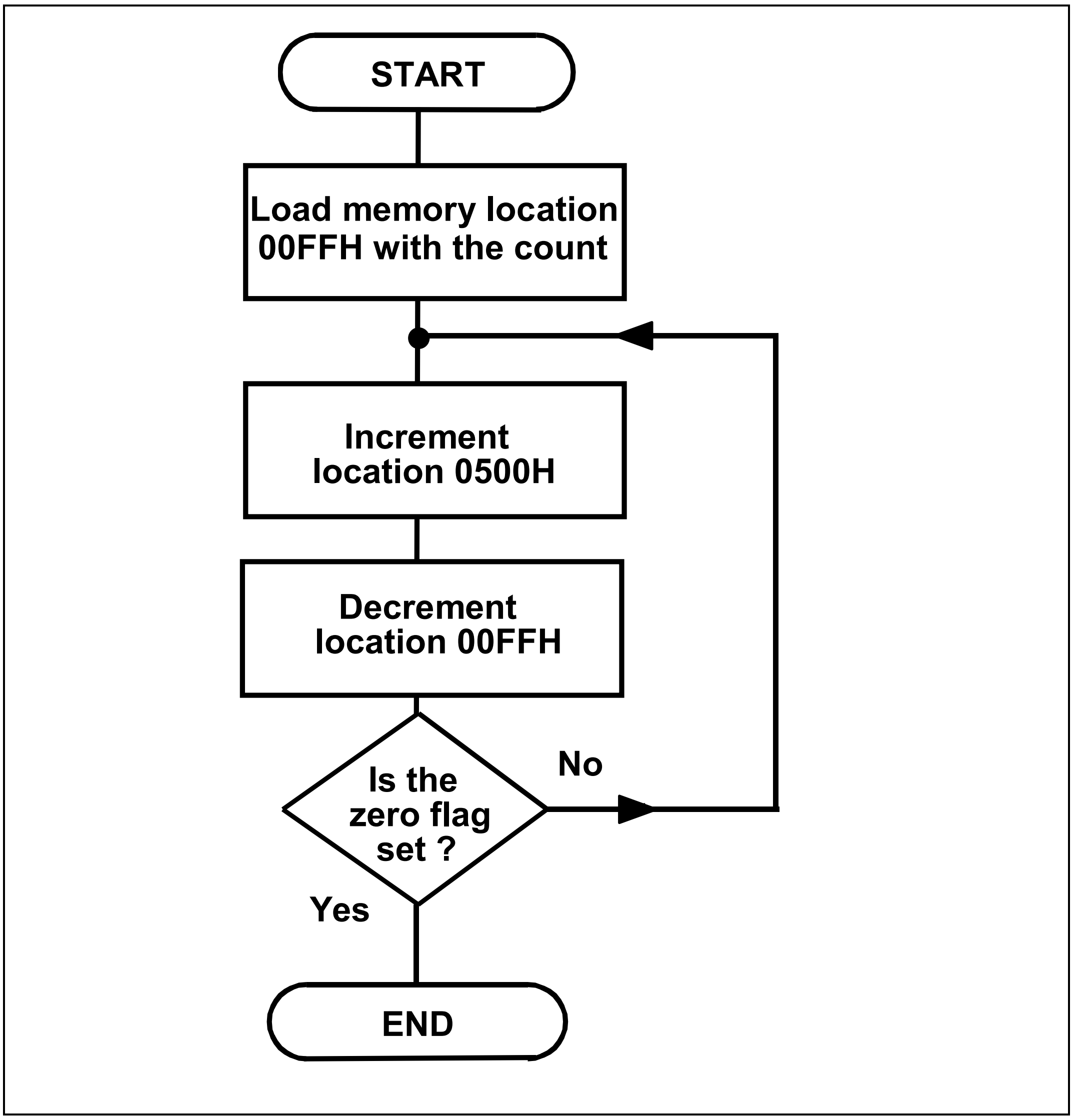
So far programs have been decision-making rather than repeated loops. Consider now the problem of repeating a section of a program a given number of times. These types of programs often use a register or memory location as a **loop counter**.

The loop counter is **decremented** (decreased by 01_H) on each pass through the loop and tested for zero. When the counter reaches zero the program exits from the loop and continues. This is a fundamental technique in assembly language programming.

9.9 Worked Example

Write a program which will increase the contents of location 0500_H, in steps of 01_H, by 07_H.

Memory location $00FF_H$ can be used as a convenient loop counter:



The Assembly Language program will be:

```
                 ORG  $0400  ;Defines the start address
0400 A9          LDA  #$07   ;Loads the accumulator with the
0401 07                      ;count value (07H)
0402 85          STA  $FF    ;Saves the count value in location 00FFH
0403 FF
0404 EE  LOOP:   INC  $0500  ;Adds 01H to the contents of location 0500H
0405 00
0406 05
0407 C6          DEC  $FF    ;Reduces the count value by 01H
0408 FF
0409 D0          BNE  LOOP   ;If the count value is NOT zero,
040A F9                      ;branch back to location 0404H
040B 60          RTS         ;Returns to MAC III system
```

9.9a **Load the above program into the MAC III. Place the value 28_{H} in memory location 0500_{H}. Run your program and then examine the contents of location 0500_{H}. Enter the hexadecimal value which you find.**

9.10 Practical Assignment

Location 0500_{H} contains a value between 00_{H} and 12_{H} which is to be multiplied by the value $0E_{H}$. Write a program which will perform this multiplication, saving the result in location $00F0_{H}$.

HINT: A simple means of achieving multiplication is to add a value to itself a given number of times.

9.10a **Use your program for Practical Assignment 9.10 to calculate $0A_{H}$ x $0E_{H}$. Enter the result you find.**

9.10b **Modify your program for Practical Assignment 9.10 to calculate 09_{H} x 08_{H}. Enter the result you find.**

Student Assessment 9

1. The type of structure used to repeat a section of program several times is called:

a an Echo

b a Go To

c a Loop

d a Repeat

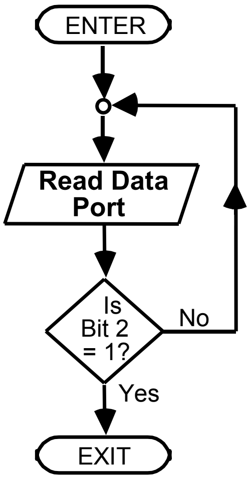
2. The program section described by the flowchart shown below will:

a repeat indefinitely

b repeat until a condition becomes true

c repeat for a given number of passes

d not repeat



3. The type of JUMP or BRANCH which is always taken is called a:

a Conditional Jump or Branch

b Direct Jump or Branch

c Indirect Jump or Branch

d Unconditional Jump or Branch

Student Assessment 9 Continued ...

4. The types of JUMPs or BRANCHes which allow the microprocessor to make decisions are called:

a Conditional Jumps or Branches

b Direct Jumps or Branches

c Indirect Jumps or Branches

d Unconditional Jumps or Branches

5. The type of addressing where the destination is expressed in terms of the number of bytes forward or backward from the present location is called:

a Conditional

b Direct

c Indirect

d Relative

6. The largest positive 8-bit offset for relative addressing is:

a 125_{10}

b 126_{10}

c 127_{10}

d 128_{10}

7. The assembly language instruction at location 0418_H is "`BCC INCPRT`". If the location identified by the label "`INCPRT`" is $041E_H$, the 2's complement displacement for the branch instruction will be:

a $F8_H$

b 04_H

c FA_H

d 06_H

Continued ...

Student Assessment 9 Continued ...

8. The Carry Flag is set when the result of the last arithmetic operation is:

a zero

b non-zero

c less than 8 bits

d greater than 8 bits

9. The Flag which is set when the result of the last arithmetic operation is zero is the:

a Carry Flag

b Negative Flag

c Overflow Flag

d Zero Flag

10. The program section which will repeatedly (and indefinitely) add 02_H to the Accumulator is:

a

```
HERE: ADC #$02
      JMP HERE
```

b

```
HERE: ADC #$02
      BEQ HERE
```

c

```
HERE: ADC #$02
      BCS HERE
```

d

```
HERE: ADC #$02
      BCC HERE
```

Student Assessment 9 Continued ...

11. The program section below will add the contents of location 2000_H to the Accumulator:

```
NEXT: ADC  $2000
      BCS  DONE
      JMP  NEXT
```

a indefinitely

b until the result is greater than 8 bits

c until the result is less than 8 bits

d until the result is equal to 2000_H

Chapter 10 Further Programs with Loops

Objectives of this Chapter

Having studied this chapter you will be able to:

- Describe the operation of the COMPARE instruction.
- Explain how the COMPARE instruction will affect the Carry and Zero flags.
- Write programs which use the COMPARE instruction.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

In the previous chapter you learned how to detect if the contents of the accumulator were zero or exceeded FF$_H$. In this chapter you will learn how to determine if the contents of the accumulator are **any** given value by using the COMPARE instruction.

10.1 The Compare Instruction

Consider the problem of testing the accumulator to see if it contains 21$_H$. Subtracting 21$_H$ from the accumulator would cause the zero flag to be **set** if the accumulator had contained 21$_H$. A simple example is shown below:

```
                   ORG  $0400 ;Defines the start address
0400  D8           CLD        ;Selects binary arithmetic mode
0401  AD           LDA  $0500 ;Loads the accumulator from location 0500H
0402  00
0403  05
0404  38           SEC
0405  E9           SBC  #$21  ;Subtracts 21H from the accumulator
0406  21
0407  F0           BEQ  ZERO  ;If the result is zero, branch to
0408  05                      ;location 040EH
0409  A9           LDA  #$55
040A  55
040B  85           STA  $F0   ;Saves the marker 55H in location 00F0H
040C  F0
040D  60           RTS        ;Returns to MAC III system
040E  A9    ZERO:  LDA  #$AA
040F  AA
0410  85           STA  $F0   ;Saves the marker AAH in location 00F0H
0411  F0
0412  60           RTS        ;Returns to MAC III system
```

This program will save the marker value AA$_H$ in location 00F0$_H$ if the contents of location 0500$_H$ are 21$_H$. If the contents of location 0500$_H$ are **not** 21$_H$, the marker value 55$_H$ is saved.

The only difficulty with this technique is that it destroys the contents of the accumulator. Now, since this is a very common problem in assembly language programming, all microprocessors provide a special instruction which is like subtraction but does **not** destroy the accumulator contents.

This is the COMPARE instruction. When a COMPARE is executed, the result of the subtraction is **lost** but the status register flags are conditioned to reflect the nature of the result (for example, zero flag set/clear).

The COMPARE instruction will condition 3 flags:

Zero Flag.
Carry Flag.
Negative Flag.

You have not yet met the Negative Flag but its operation is quite simple: The negative flag is **set** when the last ALU operation gives a **negative** 2's complement result.

The COMPARE instruction affects the Zero and Carry flags specifically thus:

Zero Flag:

Set if accumulator **equals** data.
Clear if accumulator does **not equal** data.

Carry Flag:

Set if accumulator is greater than or equal to data.
Clear if accumulator is smaller than data.

The loop counter programs in the last chapter all counted **down** to zero. You can now use COMPARE to allow counting **up** from zero to any desired value.

You can also use COMPARE to detect the greater of two values.

In 6502 assembly language there are a number of ways of using compare. These include:

compare immediate data with the accumulator.

compare the contents of a memory location with the accumulator.

So, the previous program section could be re-written thus:

```
                    ORG  $0400 ;Defines the start address
0400   AD           LDA  $0500 ;Loads the accumulator from
0401   00                      ;location 0500H
0402   05
0403   C9           CMP #$21   ;Compares the accumulator with 21H
0404   21
0405   F0           BEQ  ZERO  ;If the result is zero, branch to
0406   05                      ;location 040CH
0407   A9           LDA  #$55  ;Result is non-zero so saves the marker 55H
0408   55                      ;in location 00F0H
0409   85           STA  $F0
040A   F0
040B   60           RTS        ;Returns to MAC III system
040C   A9   ZERO:   LDA  #$AA  ;Result is zero so saves the marker AAH in
040D   AA                      ;location 00F0H
040E   85           STA  $F0
040F   F0
0410   60           RTS        ;Returns to MAC III system
```

Note that the operation of the COMPARE instruction is not affected by the state of the Decimal flag, so the "CLD" instruction is no longer required.

10.1a **If the Accumulator contains the value 49_H and then the instruction** "`CMP #$49`" **is executed, the status of the Carry (C) and Zero (Z) Flags will be:**

a C=0, Z=0

b C=0, Z=1

c C=1, Z=0

d C=1, Z=1

10.1b **The Accumulator initially contains the value $3A_H$. The instruction** "`CMP #$25`" **is then executed. Enter the new contents of the Accumulator (in hexadecimal).**

10.1c **The Accumulator initially contains the value 77_H. A COMPARE instruction is executed. This sets the Carry (C) Flag and clears the Zero (Z) Flag. The value which was compared with the Accumulator was:**

a less than 77_H

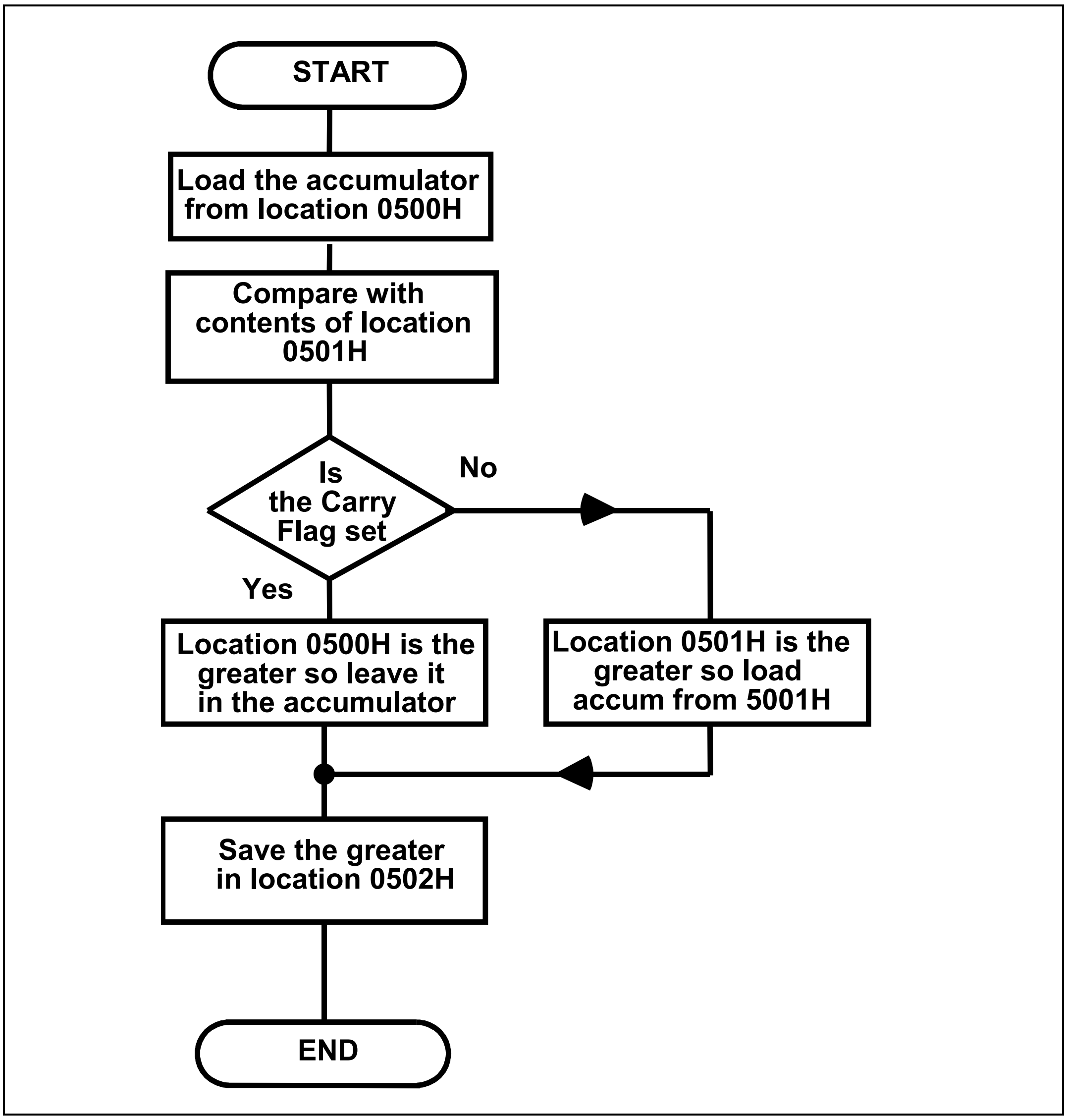
b equal to 77_H

c greater than 77_H

d the Status Register

10.2 Worked Example

Write a program that will inspect the contents of locations 0500_H and 0501_H. The greater of these two values should be saved in location 0502_H.



The Assembly Language program will be:

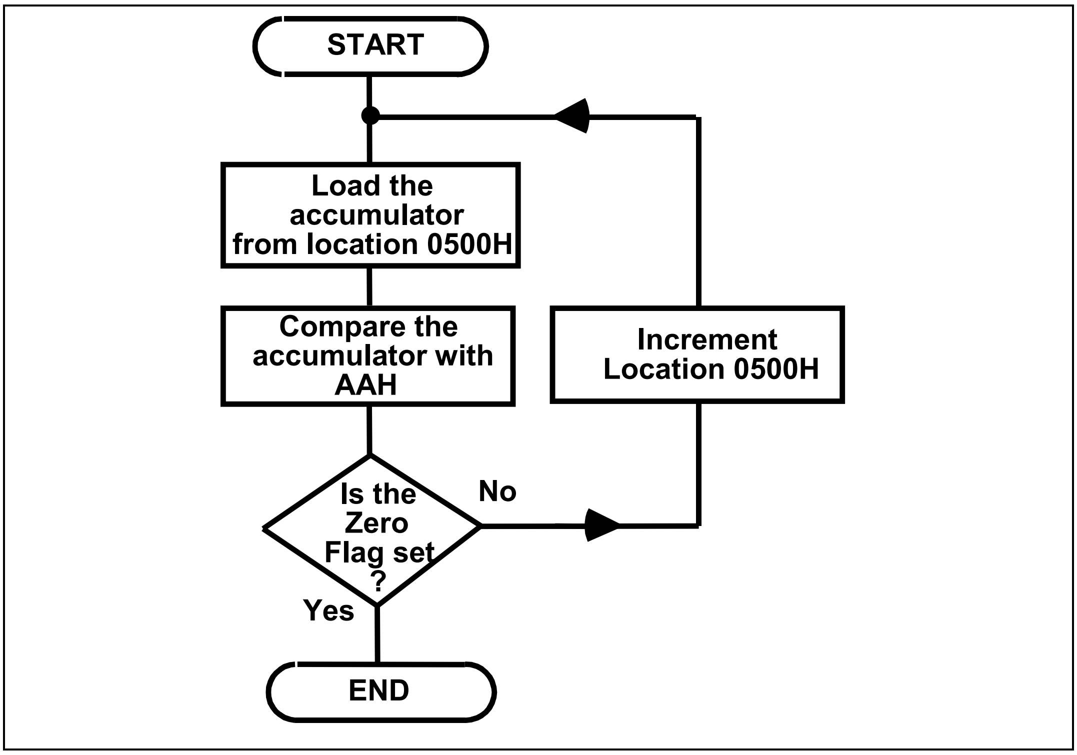
```
                   ORG  $0400   ;Defines the start address
0400   AD          LDA  $0500   ;Reads the contents of location
0401   00                       ;0500H
0402   05
0403   CD          CMP  $0501   ;Compares accumulator with the
0404   01                       ;contents of location 0501H
0405   05
0406   90          BCC  CCLR    ;If C=0, branch to location 040CH
0407   04
0408   8D          STA  $0502   ;C=1 so contents of 0500H are the
0409   02                       ;greater (or equal), so save in
040A   05                       ;location 0502H
040B   60          RTS          ;Returns to MAC III system
040C   AD    CCLR: LDA  $0501   ;C=0 so contents of 0501H are the
040D   01                       ;greater. Load accumulator from
040E   05                       ;location 0501H
040F   8D          STA  $0502   ;Saves greater value in location
0410   02                       ;0502H
0411   05
0412   60          RTS          ;Returns to MAC III system
```

Notice the instruction at location 0408_H: There is no need to load the accumulator again from 0500_H, since the accumulator will already contain this value and is unaffected by the CMP instruction.

10.2a Load the program for Worked Example 10.2 into MAC III memory. Place the value 46_H in location 0500_H and the value 71_H in location 0501_H. Run the program and examine the contents of location 0502_H. Enter the hexadecimal value which you find.

10.3 Worked Example

The contents of location 0500_H are AA_H or less. Write a program which will examine the contents of location 0500_H and then increase the contents in steps of 01_H until 0500_H contains AA_H.



The assembly language program will be:

```
                      ORG   $0400       ;Defines the start address
0400   A9             LDA   #$AA        ;Loads the value AAH into
0401   AA                               ;the accumulator
0402   CD     LOOP:   CMP   $0500       ;Compares accumulator
0403   00                               ;with the contents of
0404   05                               ;location 0500H
0405   F0             BEQ   SAME        ;If contents of location
0406   06                               ;0500H equal AAH, branch
                                        ;to 040DH
0407   EE             INC   $0500       ;Adds 01H to contents of
0408   00                               ;0500H
0409   05
040A   4C             JMP   LOOP6       ;Jump back to location
040B   02                               ;0402H
040C   04
040D   60     SAME:   RTS               ;Returns to MAC III system
```

10.3a **Load the program for Worked Example 10.3 into MAC III memory. Place the value 52_H in location 0500_H. Run the program and then examine the contents of location 0500_H. Enter the hexadecimal value which you find.**

10.4 Practical Assignment

Write a program which will examine the contents of location 0050_H. If this location contains 99_H, then location 0500_H should be loaded with 81_H. Otherwise location 0500_H should be loaded with $7E_H$.

10.4a **Load your program for Practical Assignment 10.4 into MAC III memory. Place the value $3B_H$ in location 0050_H. Run the program and then examine the contents of location 0500_H. Enter the hexadecimal value which you find.**

10.4b **The number of times that your program for Practical Assignment 10.4 uses a "`CMP`" instruction is:**

a once

b twice

c three times

d four times

10.5 Practical Assignment

Write a program which will inspect the contents of location 0580_H. Location $00FF_H$ should then be loaded with a marker value thus:

If the contents of location 0580_H are:

less than 37_H: load location $00FF_H$ with 80_H
equal to 37_H: load location $00FF_H$ with AA_H
greater than 37_H: load location $00FF_H$ with 01_H

10.5a **Load your program for Practical Assignment 10.5 into MAC III memory. Place the value 93_H in location 0580_H. Run the program and then examine the contents of location $00FF_H$. Enter the hexadecimal value which you find.**

10.5b **Enter the number of times that your program for Practical Assignment 10.5 uses a "`CMP`" instruction.**

10.6 Practical Assignment

Write a program which will inspect the contents of locations 0050_H, 0051_H and 0052_H. The largest of these should then be saved in location 0500_H.

10.6a **Load your program for Practical Assignment 10.6 into the MAC III. Place the values shown below in the memory locations indicated:**

Location	Contents
0050_H	$2D_H$
0051_H	71_H
0052_H	$5E_H$

Run your program and then examine the contents of location 0500_H. Enter the hexadecimal value which you find.

10.6b **With your program for Practical Assignment 10.6 still loaded in the MAC III, change the values stored in the memory locations below thus:**

Location	Contents
0050_H	52_H
0051_H	$4A_H$
0052_H	67_H

Run your program and then examine the contents of location 0500_H. Enter the hexadecimal value which you find.

Student Assessment 10

1. **The 6502 Assembly Language instruction which will subtract the contents of a memory location from the Accumulator and set or clear flags accordingly, without changing the contents of the memory location or the Accumulator is:**

   a Compare

   b Loop

   c Subtract

   d Test

2. **The 6502 Assembly Language instruction which can be used to check if the contents of the Accumulator are equal to 56_H is:**

   a `CHK $56`

   b `CHK #$56`

   c `CMP $56`

   d `CMP #$56`

3. **Following a COMPARE instruction, both the Zero and Carry Flags are clear (i.e. = 0). This indicates that:**

   a the accumulator contains zero.

   b the accumulator and operand are equal.

   c the accumulator is smaller than the operand.

   d the accumulator is greater than the operand.

4. **If the accumulator is greater than the operand for a COMPARE instruction, the Zero and Carry Flags will be:**

   a Z = 0 C = 0

   b Z = 0 C = 1

   c Z = 1 C = 0

   d Z = 1 C = 1

Continued ...

Student Assessment 10 Continued ...

5. Consider the program section:

```
        CMP  $1800
        BCC  DEST1
        LDA  #$11
        STA  $60
        RTS
DEST1:  LDA  #$88
        STA  $60
        RTS
```

The action of this program section will be to place the value:

a 11_H in location 0060_H if the Carry Flag is clear.

b 11_H in location 0060_H if the Zero Flag is set.

c 88_H in location 0060_H if the Carry Flag is clear.

d 88_H in location 0060_H if the Carry Flag is set.

6. For the program in Question 5 above; if the value in location 1800_H was equal to the contents of the Accumulator, the value placed in location 0060_H would be:

a 60_H

b 11_H

c 00_H

d 88_H

Chapter 11 Indexed Addressing

Objectives of this Chapter

Having studied this chapter you will be able to:

- ■ Explain how the following instructions operate on the Index Registers:

  Load Index Register
  Store Index Register
  Compare Index Register
  Increment/Decrement Index Register
  Transfer Index Register

- ■ Describe the operation of the following 6502 Addressing Modes:

  Absolute Indexed-X
  Absolute Indexed-Y
  Zero Page Indexed-X
  Zero Page Indexed-Y

- ■ Write programs which use the 6502 Indexed Addressing Modes.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

The 6502 has three General Purpose Registers:

Accumulator
X-Register
Y-Register

You will already be familiar with the Accumulator but we have not yet encountered the X- and Y-Registers. These are known as the **Index Registers**.

Index Registers may be used for general purpose applications (for example, as a counter or for temporary storage of data) but their main use is in **Indexed Addressing**.

The Indexed Addressing modes will be explained more fully later in this chapter. Before progressing to using Indexed Addressing, we must first examine how the Index Registers may be manipulated.

11.1 The Index Registers

There are a number of instructions which can be used to operate upon data within the Index Registers:

Load Index Register (LDX, LDY)

The Index Registers can be loaded with a value from memory, using Immediate, Absolute or Zero Page Addressing.

For example:

```
0412   A2   LDX  #$45    ;Loads the X-Register with the
0413   45                ;value 45H

046D   AC   LDY  $05E0   ;Loads the Y-Register from
046E   E0                ;location 05E0H
046F   05

0487   A6   LDX  $40     ;Loads the X-Register from
0488   40                ;location 0040H
```

Store Index Register (STX, STY)

The contents of Index Registers can be saved to a memory location, using Absolute or Zero Page Addressing.

For example:

```
0431   8E   STX  $0502   ;Saves the X-Register in location 0502H
0432   02
0433   05
04A3   84   STY  $90     ;Saves the Y-Register in location 0090H
04A4   90
```

Compare Index Register (CPX, CPY)

The contents of Index Registers can be compared with values in memory using Immediate, Absolute or Zero Page Addressing.

For example:

```
040A   E0   CPX  #$52    ;Compares the value 52H with
040B   52                ;the X-Register

0429   EC   CPX  $05B0   ;Compares the contents of
042A   B0                ;location 05B0H with the X-Register
042B   05

0492   C4   CPY  $38     ;Compares the contents of
0493   38                ;location 0038H with the Y-Register
```

Increment/ Decrement Index Register (INX, INY, DEX,DEY)

The contents of Index Registers can be Incremented or Decremented, using Implied Addressing.

For example:

```
0420   E8   INX          ;Increases the contents of the
                         ;X-Register by 01H

043C   C8   INY          ;Increases the contents of the
                         ;Y-Register by 01H

0482   CA   DEX          ;Decreases the contents of the
                         ;X-Register by 01H

04A2   88   DEY          ;Decreases the contents of the
                         ;Y-Register by 01H
```

Transfer Index Register (TAX, TXA TAY, TYA)

Data can be duplicated between Index Registers and the Accumulator by using the Transfer Instructions. These instructions use Implied Addressing.

For example:

```
0436   AA   TAX          ;Duplicates the contents of the
                         ;accumulator in the X-Register
0452   8A   TXA          ;Duplicates the contents of the
                         ;X-Register in the accumulator
0478   A8   TAY          ;Duplicates the contents of the
                         ;accumulator in the Y-Register
04A1   98   TYA          ;Duplicates the contents of the
                         ;Y-Register in the accumulator
```

It is not possible for the standard 6502 to transfer directly from one index register to another. Such transfers must be through the accumulator.

For example: To copy the contents of the X-Register into the Y-Register:

```
04C2   8A   TXA          ;Duplicates the contents of the
                         ;X-Register in the accumulator
04C3   A8   TAY          ;Duplicates the contents of the
                         ;accumulator in the Y-Register
```

11.1a **The 6502 instruction which will copy the contents of memory location 0527_H into the X Register is:**

- a `LDA $0527`
- b `LDX $0527`
- c `STA $0527`
- d `STX $0527`

11.1b **The 6502 instruction** "`CPY   $7A`" **will:**

- a copy the value $7A_H$ into the Y Register.
- b copy the contents of location $007A_H$ into the Y Register.
- c compare the value $7A_H$ with the Y Register.
- d compare the contents of location $007A_H$ with the Y Register.

11.2 Indexed Addressing

In an Indexed Addressing Mode, the contents of an Index Register are **added** to the operand to form the address of the data to be acted upon:

Base Address	+	Index Register Contents	=	Destination Address

The Base Address may be expressed in terms of Absolute or Zero Page addressing. Since the Index Register may be manipulated (for example Incremented, Decremented, etc.), a **range** of addresses may be specified. Indexed Addressing is frequently used in programs with a loop structure. The data source or destination can be changed at each pass through the loop by Incrementing or Decrementing the contents of the Index Register.

11.3 Absolute Indexed Addressing

Consider the Assembly Language instruction:

```
0423   BD   LDA $0520,X    ;Loads the accumulator from
0424   20                  ;the memory location
0425   05                  ;0520H + X
```

Suppose the X-Register contained 15_H, then the Destination Address would be formed thus:

Base Address	+	Index Register Contents	=	Destination Address
0520_H	+	15_H	=	0535_H

So this instruction would load the accumulator from location 0535_H.

Absolute Indexed-Y Addressing works in just the same way.

Consider the following program section:

```
045A   A0   LDY  #$62       ;Loads the Y-Register with
045B   62                   ;the value 62H
045C   99   STA  $0534,Y    ;Saves the accumulator in
045D   34                   ;the memory location
045E   05                   ;0534H + Y
```

Base Address	+	Index Register Contents	=	Destination Address
0534_H	+	62_H	=	0596_H

So this instruction would save the accumulator in location 0596_H.

11.4 Zero Page Indexed Addressing

These addressing modes are very similar to the Absolute Indexed Addressing modes, except that the Base Address may only be within the range 0000_H to $00FF_H$ (Page Zero).

Consider the Assembly Language instruction:

```
04C7   B5   LDA  $50,X    ;Loads the accumulator from
04C8   50                 ;memory location 0050H + X
```

Now, suppose the X-Register contained 20_H, then the Destination Address would be formed thus:

Base Address	+	Index Register Contents	=	Destination Address
0050_H	+	20_H	=	0070_H

So this instruction would load the accumulator from location 0070_H.

11.4a **The 6502 program section:**

```
LDX  #$2E
LDA  #$45
STA  $90,X
```

will place the value:

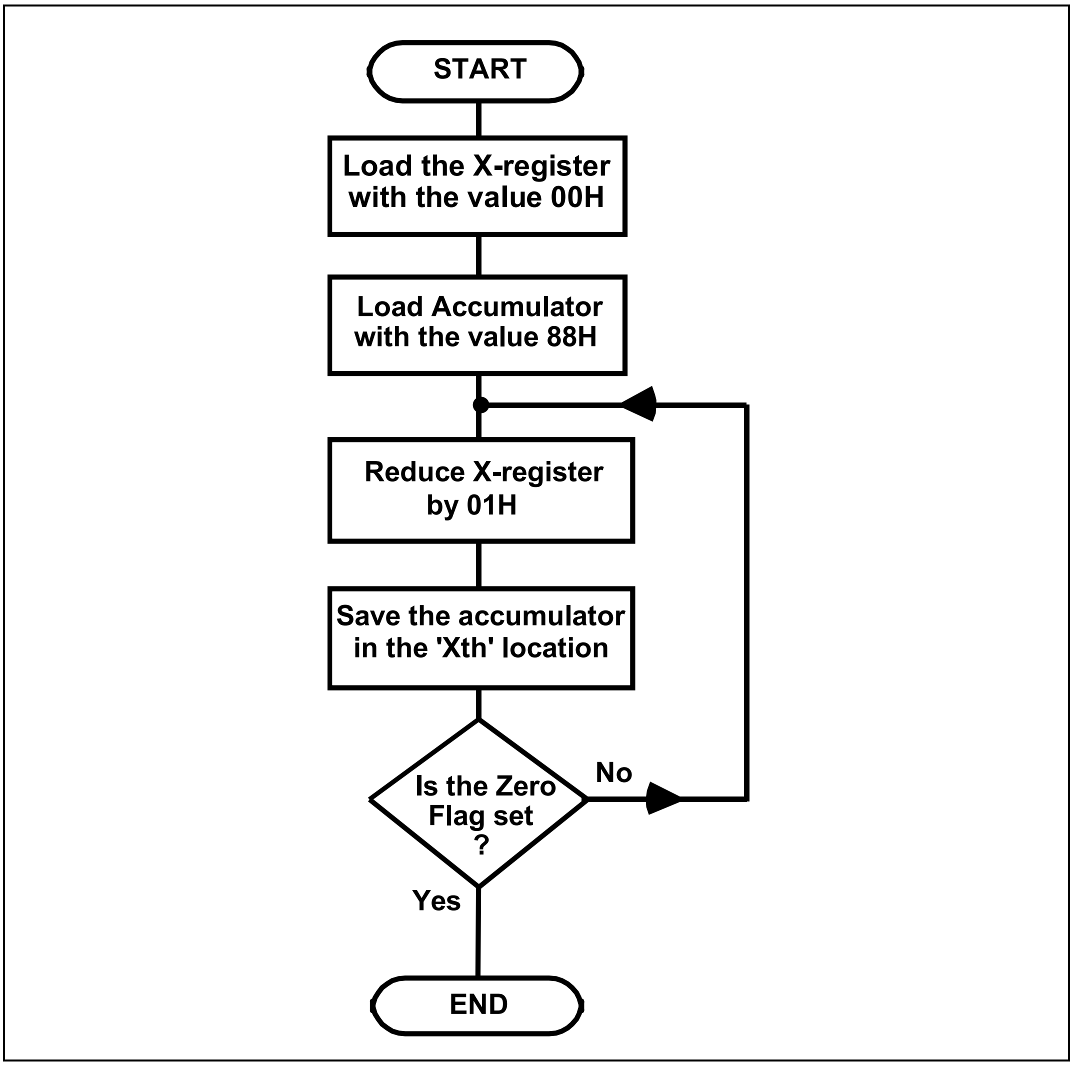
- a 2E$_H$ in location 00D5$_H$
- b 45$_H$ in location 00BE$_H$
- c 45$_H$ in location 0011E$_H$
- d 90$_H$ in location 0073$_H$

11.4b **The 6502 instruction which will copy the contents of the memory location in a data table starting at location 0200$_H$ and pointed to by the Y Register into the accumulator is:**

- a `LDA  $0200,Y`
- b `LDY  $0200,A`
- c `STA  $0200,Y`
- d `STY  $0200,A`

11.5 Worked Example

Write a program that will fill page 05_H of memory with the value 88_H.



Notice that the count is initially set to **zero** and that the count is decremented **before** each save instruction. On the first pass the X-register is at 00_H. This will then be decremented to FF_H (wrap around). On the final pass the X-register becomes zero again but the last location is filled **before** the X-register is tested by the BNE instruction.

The Assembly Language program will be:

```
                        ORG     $0400       ;Defines the start address
0400   A2               LDX     #$00        ;Sets the count to zero
0401   00
0402   A9               LDA     #$88        ;Loads the Accumulator
0403   88                                   ;with the 'fill' value
0404   CA      LOOP:    DEX                 ;Decrements the X-Register
0405   9D               STA     $0500,X     ;Saves the accumulator in
0406   00                                   ;the 'Xth' location
0407   05
0408   D0               BNE     LOOP        ;If the X-register is not
0409   FA                                   ;zero, branch back to
                                            ;location 0404H
040A   60               RTS                 ;Returns to MAC III system
```

11.5a **In the program above, the instruction which tests to see whether the next location is to be filled with 88_H is:**

a `DEX`

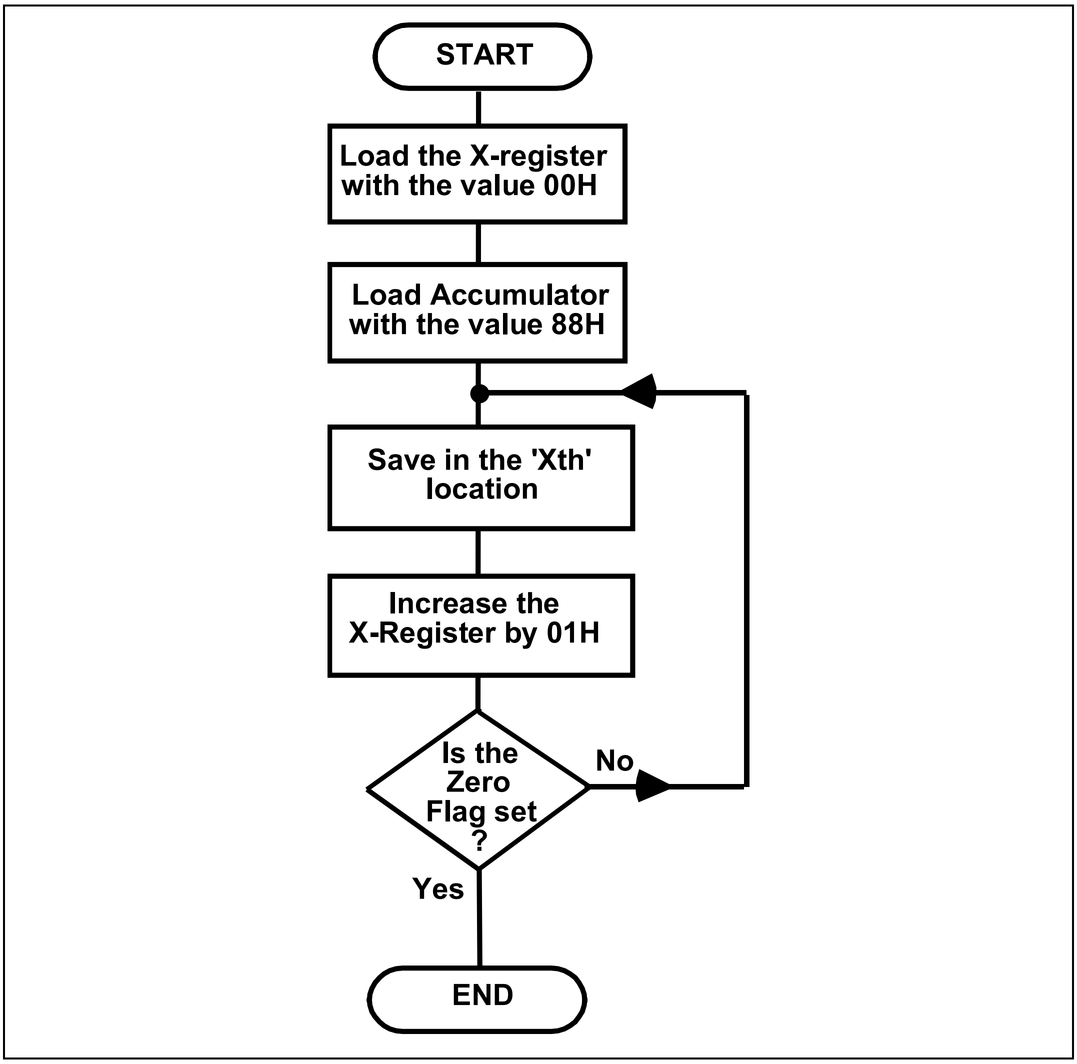
b `STA $0500,X`

c `BNE LOOP`

d `RTS`

11.5b **Load the program above into the MAC III and then execute from location 0400_H. Examine the contents of location 0500_H. Enter the hexadecimal value which you find at this location.**

This is not the only possible solution to this type of problem. An alternative strategy might have been:



Again the count is initially set to **zero** but is now **incremented after** each save instruction. On the final pass the X-register will be incremented from FF_H to 00_H and the program will exit from the loop.

The Assembly Language program will be:

```
                         ORG $0400        ;Defines the start address
0400   A2                LDX  #$00        ;Sets the count to zero
0401   00
0402   A9                LDA  #$88        ;Loads the Accumulator
0403   88                                 ;with the 'fill' value
0404   9D  LOOP:         STA  $0500,X     ;Saves the accumulator in
0405   00                                 ;the 'Xth' location
0406   05
0407   E8                INX              ;Increments the X-Register
0408   D0                BNE  LOOP        ;If the X-register is not
0409   FA                                 ;zero, branch back to
                                          ;location 0404H
040A   60                RTS              ;Returns to MAC III system
```

Modify the last program and run again to verify correct operation.

11.6 Practical Assignment

Write a program that will fill locations 0500_H to 0580_H with the value AA_H.

11.6a **Place the value 00_H in location 0580_H. Load your program for Practical Assignment 11.6 into the MAC III and execute. Examine the contents of location 0580_H. Enter the hexadecimal value which you find at this location.**

11.6b **Place the value 00_H in location 0581_H. Check that your program for Practical Assignment 11.6 is still loaded in the MAC III. Run the program and then examine the contents of location 0581_H. Enter the hexadecimal value which you find at this location.**

11.7 Practical Assignment

Write a program that will copy the block of data 0500_H - 0520_H to locations 0580_H - $05A0_H$.

11.7a **Place the value 68_H in location 0520_H. Load your program for Practical Assignment 11.7 into the MAC III and execute. Examine the contents of location $05A0_H$. Enter the hexadecimal value which you find at this location.**

11.7b **Place the value 22_H in location $05A1_H$. Check that your program for Practical Assignment 11.7 is still loaded in the MAC III. Run the program and then examine the contents of location $05A1_H$. Enter the hexadecimal value which you find at this location.**

11.8 Practical Assignment

Write a program that will examine the contents of each location from 0040_H to 0060_H and save the largest value found in location $00FF_H$.

11.8a **Place the value 00_H in every location from 0040_H to 0060_H. Now place the following values in the locations shown:**

Location	Contents
0040_H	45_H
0050_H	67_H
0060_H	32_H

Load your program for Practical Assignment 11.8 into the MAC III and execute. Examine the contents of location $00FF_H$. Enter the hexadecimal value which you find at this location.

Student Assessment 11

1. **The 6502 instruction that will save the contents of the X Register in location 0500_H is:**

a `STA  $0500`

b `STX  $0500`

c `STA  $0500,X`

d `STX  $0500,A`

2. **The Y Register initially holds the value $4F_H$. After the instruction " `DEY` " has been executed, the contents of the Y Register will be:**

a $4C_H$

b $4D_H$

c $4E_H$

d 50_H

3. **The instruction that copies the contents of the Accumulator into the X Register is:**

a `TAX`

b `TAY`

c `TXA`

d `TYA`

4. **The sequence of 6502 Assembly Language instructions required to transfer the contents of the X Register to the Y Register is:**

a
```
TXA
TAY
```

b
```
TXA
TYA
```

c
```
TAX
TAY
```

d
```
TAX
TYA
```

Continued ...

Student Assessment 11 Continued ...

5. **For the program section:**

```
LDX  #$16
LDA  $0515,X
```

The second instruction will load the accumulator from location:

a $04FF_H$

b 0515_H

c $052B_H$

d 0531_H

6. **The mode of addressing used by the 6502 instruction** "`STA  $0680,Y`" **is:**

a Absolute Indexed X

b Absolute Indexed Y

c Zero Page Indexed X

d Zero Page Indexed Y

7. **The 6502 instruction** "`LDA  $80,X`" **will load:**

a the Accumulator from location 0080_H

b the Accumulator from location 0800_H

c the Accumulator from location (0080_H - X)

d the Accumulator from location (0080_H + X)

8. **The 6502 program section:**

```
LDX  #$42
STA  $0800,X
```

will:

a Load the accumulator from location 0800_H

b Load the accumulator from location 0842_H

c Save the accumulator in location 0800_H

d Save the accumulator in location 0842_H

Student Assessment 11 Continued ...

9. After the 6502 instruction sequence below has been executed,

```
LDY  #$4D
STA  $0780,Y
DEY
STA  $0780,Y
DEY
STA  $0780,Y
```

the contents of the Y Register will be:

- a $4B_H$
- b $4C_H$
- c $4D_H$
- d $4E_H$

Chapter 12 Logical and Test Instructions

Objectives of this Chapter

Having studied this chapter you will be able to:

- Apply the AND logical operator to binary data.
- Describe the operation of the following 6502 instructions:

    AND
    BIT
    SHIFT
    ROTATE

- Use the AND and BIT instructions to test any bit(s) within the accumulator or a memory location.
- Write programs that use the instructions:

    AND
    BIT
    SHIFT
    ROTATE

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

You have learned in the previous chapter how to detect if the contents of a location or register are any given value. In this chapter you will learn how logical instructions can be used to test individual **bits** (or groups of bits) within a location or register.

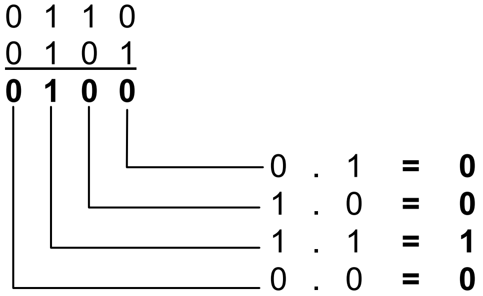
12.1 Logical Operators

You will already be familiar with the ways in which some **arithmetic** operators can be applied to data (for example, ADC, SBC, etc). It is also possible to apply **logical** operators to data (for example AND, OR, EXCLUSIVE-OR).

For example AND:

Recall 0 AND 0 = 0 $0 \,.\, 0 = 0$
0 AND 1 = 0 $0 \,.\, 1 = 0$
1 AND 0 = 0 $1 \,.\, 0 = 0$
1 AND 1 = 1 $1 \,.\, 1 = 1$

To AND together two binary numbers, the AND operator is applied bit by bit. For example:



So $0110_2 \,.\, 1101_2 = \mathbf{0100_2}$

Notice that any given bit in the result can only be 1 if **both** of the numbers have a 1 in that position. This property can be used to test for bits at 1 within a register or location.

Consider 99_H ANDed with a mask $F0_H$:

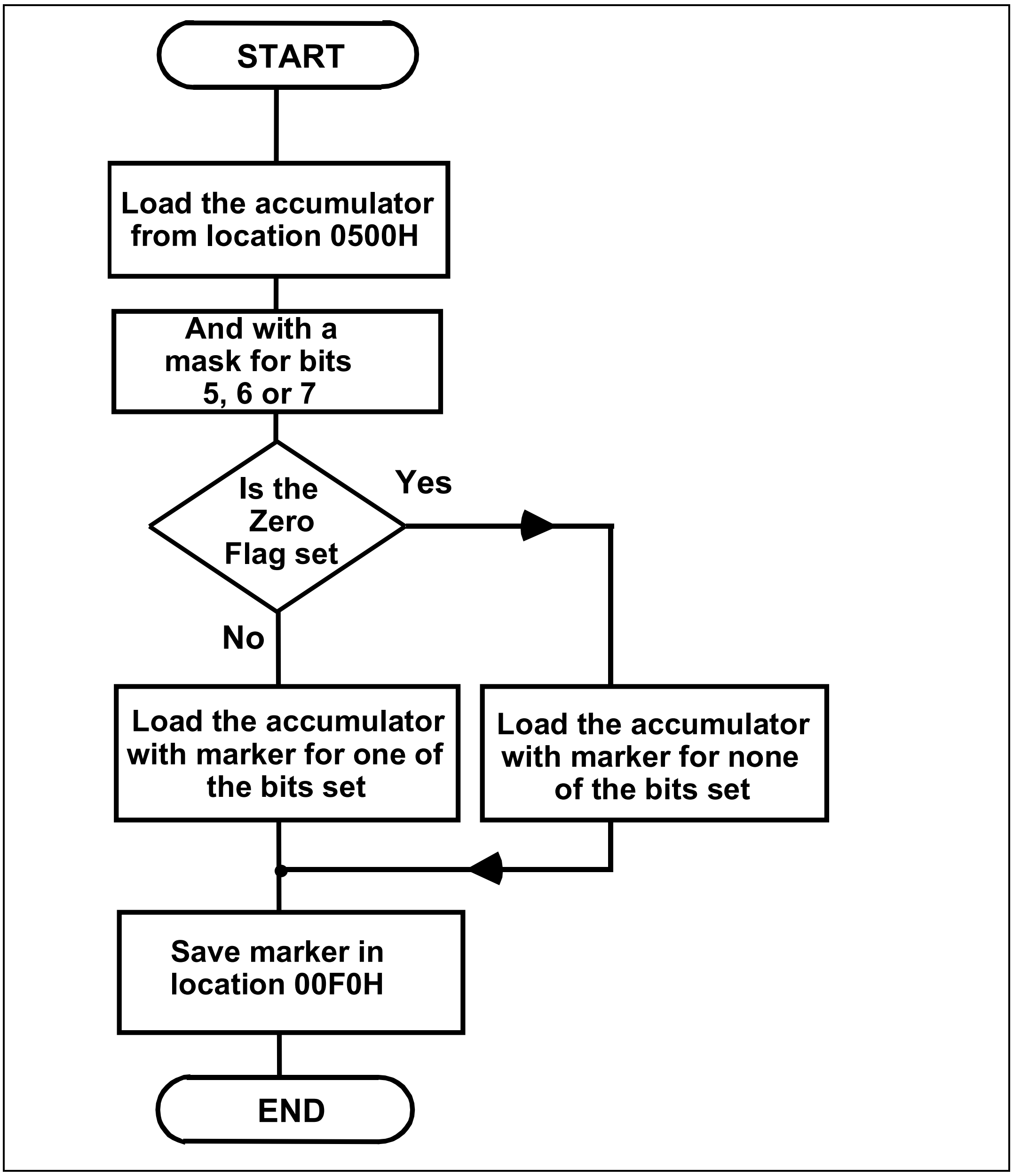
$$
\begin{array}{rl}
99_H = & 1001\ 1001_2\ . \\
F0_H = & 1111\ 0000_2 \\
\hline
 & \mathbf{1001\ 0000_2 = 90_H}
\end{array}
$$

This technique can be used to test for a **number** of bits within a register or memory location. Worked Example 12.2 shows how this may be achieved.

12.1a **The Accumulator initially contains the value $B7_H$. Enter the value found in the Accumulator after it has been ANDed with $C6_H$.**

12.2 Worked Example

Write a program that will examine the contents of location 0500_H. A marker value of $C0_H$ should be saved in location $00F0_H$ if **any** of bits 5, 6 and 7 of location 0500_H are **set**. Otherwise, a marker value of 03_H should be placed in location $00F0_H$.

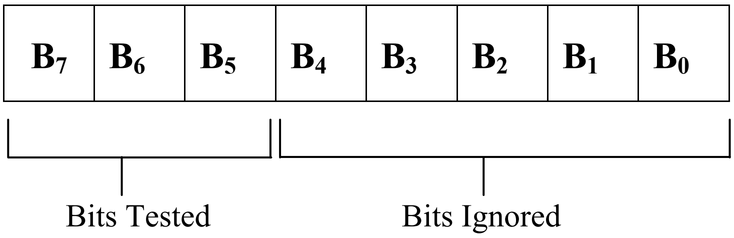


The Assembly Language program will be:

```
                        ORG  $0400       ;Defines the start address
0400   AD               LDA  $0500       ;Read contents of memory
0401   00                                ;location 0500H
0402   05
0403   29               AND  #$E0        ;Tests bits 5,6 and 7 of
0404   E0                                ;the accumulator
0405   F0               BEQ  NONE        ;If none of the tested
0406   05                                ;bits are set, branch to
                                         ;location 040CH
0407   A9               LDA  #$C0
0408   C0
0409   85               STA  $F0         ;One or more tested bits
040A   F0                                ;set, so save marker C0H
                                         ;in location 00F0H
040B   60               RTS              ;Returns to MAC III system

040C   A9       NONE:   LDA  #$03
040D   03
040E   85               STA  $F0         ;No tested bits set, so
040F   F0                                ;save marker 03H in
                                         ;location 00F0H
0410   60               RTS              ;Returns to MAC III system
```

The value with which location 0500_H is ANDed is called the **mask**. In this case the mask is $E0_H$.

This program tests bits 5, 6 and 7 of location 0500_H so any values above $1F_H$ should give a positive result. Conventionally, bits are numbered from 0 on the right thus:



12.2a **The 6502 instruction that can be used to test for several bits of a memory location set at the same time is:**

a `AND`

b `NOT`

c `ORA`

d `NOR`

12.2b **Load the program for Worked Example 12.2 into the MAC III. Place the value 16_H in location 0500_H. Run the program and then examine the contents of location $00F0_H$. Enter the hexadecimal value which you find.**

12.2c **The program for Worked Example 12.2 is to be modified to test for any of bits 2, 3 or 4 set in memory location 0500_H. Enter the required hexadecimal mask value.**

12.3 Other Logical Instructions

6502 assembly language also allows the OR and Exclusive-OR (XOR) operators to be applied to data. These are not so commonly used as the AND instructions.

The BIT Instruction

This instruction is analogous to the COMPARE instruction. It logically ANDs the contents of the accumulator with the contents of a specified memory location. The flags are conditioned accordingly but the contents of the accumulator are **unaffected** by this instruction.

For example:

```
0430   A9   LDA  #$0F    ;Loads accumulator with mask
0431   0F
0432   2C   BIT  $0500   ;ANDs location 0500H with 0FH
0433   00
0434   05
0435   60   RTS          ;Returns to MAC III system
```

The Zero Flag is set or cleared according to the result of the logical AND. However, the effect upon the N- and V-Flags is rather unusual: The N- and V-Flags take on the states of bits 7 and 6 respectively within the memory location which has been ANDed with the accumulator.

So, for the above example, suppose that location 0500_H contains $5D_H$:

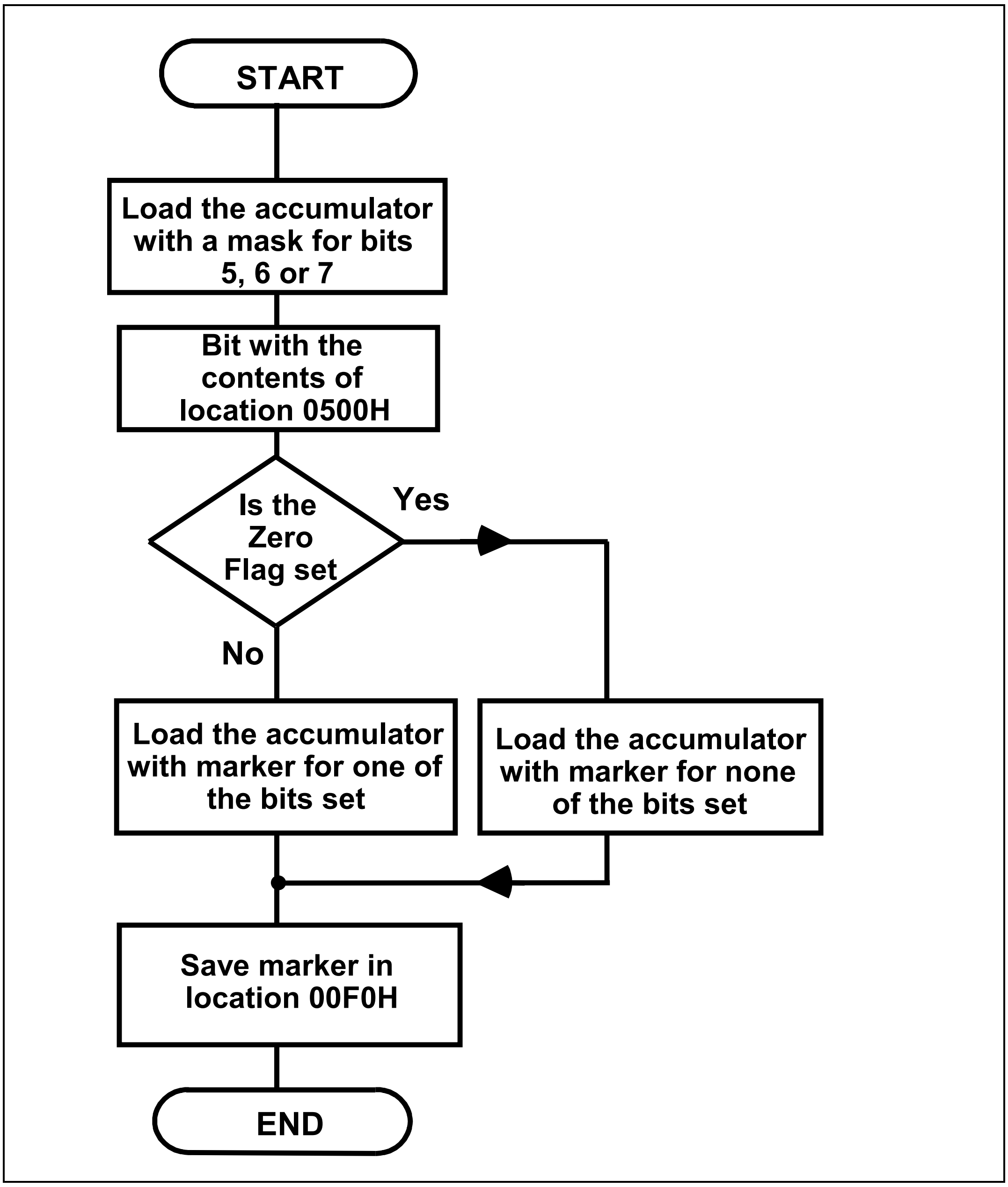
$5D_H$ is ANDed with $0F_H$ thus:

$$\begin{array}{rl} 5D_H = & 0101\ 1101_2 \\ & \bullet \\ 0F_H = & 0000\ 1111_2 \\ \hline & \mathbf{0000\ 1101_2 = 0D_H} \end{array}$$

This is a non-zero result so the zero flag will be **clear**.
The N- and V- flags follow bits 7 and 6 respectively of the value $5D_H$. Thus the N-flag will be **cleared** and the V-flag **set**.

12.4 Worked Example

Modify the solution to Worked Example 12.2 to make use of the BIT instruction.



The assembly language program will be:

```
                            ORG   $0400        ;Defines the start address
0400   A9                   LDA   #$E0         ;Loads the accumulator
0401   E0                                      ;with a mask for bits 5,6
                                               ;and 7
0402   2C                   BIT   $0500        ;ANDs contents of
0403   00                                      ;location 0500H with
0404   05                                      ;accumulator (but does
                                               ;not affect accumulator)
0405   F0                   BEQ   NONE         ;If none of tested bits
0406   05                                      ;are set, branch to
                                               ;location 040CH
0407   A9                   LDA   #$C0
0408   C0
0409   85                   STA   $F0          ;One or more tested bits
040A   F0                                      ;set, so save marker C0H
                                               ;in location 00F0H
040B   60                   RTS                ;Returns to MAC III system
040C   A9             NONE: LDA   #$03
040D   03
040E   85                   STA   $F0          ;No tested bits set, so
040F   F0                                      ;save marker 03H in
                                               ;location 00F0H
0410   60                   RTS                ;Returns to MAC III system
```

12.4a **The Accumulator initially contains the value A6$_{H}$. Enter the value found in the Accumulator after the instruction "`BIT $0500`" has been executed.**

12.4b **The program for Worked Example 12.2 is to be modified to test for any of bits 1, 2 or 3 set in memory location 0500$_{H}$. The instruction which must be changed is:**

a `LDA  #$E0`

b `BIT  $0500`

c `BEQ  NONE`

d `LDA  #$C0`

12.5 Shift and Rotate Instructions

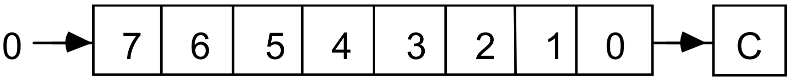
Shift Instructions

A logical shift involves each bit within a register moving one place to the left or right (depending upon the direction of the shift). Usually a zero is shifted into the register and the bit at the other end is lost.

For example: a register holding 1100 1010$_2$:

A shift right would cause the register to be changed to 0110 0101$_2$. Each bit moves one place to the right and a zero moves into the most significant bit position. There are two 6502 shift instructions. These both involve the Carry flag:

LSR: Shift right contents of the accumulator or a specified memory location.

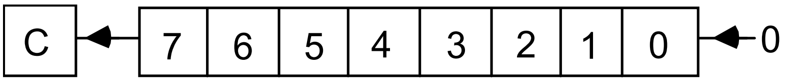


A zero is shifted into the most significant bit position and the least significant bit is shifted out into the carry flag.

For example:

```
0418   4E   LSR  $0500          ;Shifts the contents of
0419   00                       ;location 0500H RIGHT once
041A   05
```

ASL: Shift left contents of the accumulator or a specified memory location.



A zero is shifted into the least significant bit position and the most significant bit is shifted out into the carry flag.

For example:

```
0437   06   ASL  $E5            ;Shifts the contents of
0438   00                       ;location 00E5H LEFT once
```

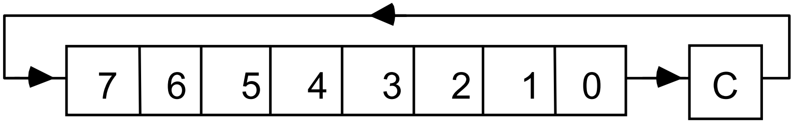
Rotate Instructions

These are similar to shift instructions, except that instead of one bit being lost and a zero shifting in, the last bit is shifted back in at the beginning.

For example: a register holding 1100 1010_2:
A rotate left would cause the register to be changed to
1001 0101_2. Each bit moves one place to the left and the most significant bit moves to the least significant position.

There are two 6502 rotate instructions. Like the shift instructions, these also involve the Carry flag:

ROR: Rotate **right** contents of the accumulator or a memory location.

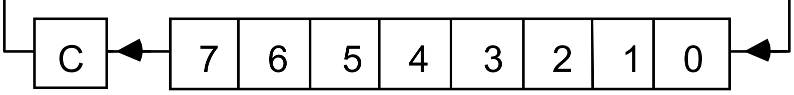


The least significant bit is shifted into the carry flag. The carry flag is also rotated into the most significant position.

For example:

```
044D   6A   ROR  A               ;Rotates the contents of
                                 ;the accumulator RIGHT
                                 ;once
```

ROL: Rotate **left** contents of the accumulator or a memory location.



The most significant bit is shifted into the carry flag. The carry flag is also rotated into the least significant position.

For example:

```
0473   3E   ROL $0500,X          ;Rotates the contents of
0474   00                        ;location (0500H + X)
041A   05                        ;LEFT once
```

Shift and Rotate instructions can be used for generating sequences for microprocessor control applications. They are also used in multiplication and division algorithms, since shifting left by one place gives multiplication by 2 (in the same way that adding a 0 to the right hand side of a denary number gives multiplication by 10). Similarly, shifting right gives division by 2.

12.5a **A register contains the byte $9C_H$. Enter the hexadecimal contents of this register after it has been shifted left 3 times.**

12.5b **The 6502 instruction which will shift the contents of location 0524_H once to the right is:**

a `ASL  #$0524`

b `ASL  $0524`

c `LSR  #$0524`

d `LSR  $0524`

12.5c **A register contains the byte 64_H. If the Carry Flag is clear, enter the hexadecimal contents of this register after it has been rotated right 4 times.**

12.5d **The 6502 instruction which will rotate the contents of the Accumulator once to the left is:**

a `ASL  A`

b `LSR  A`

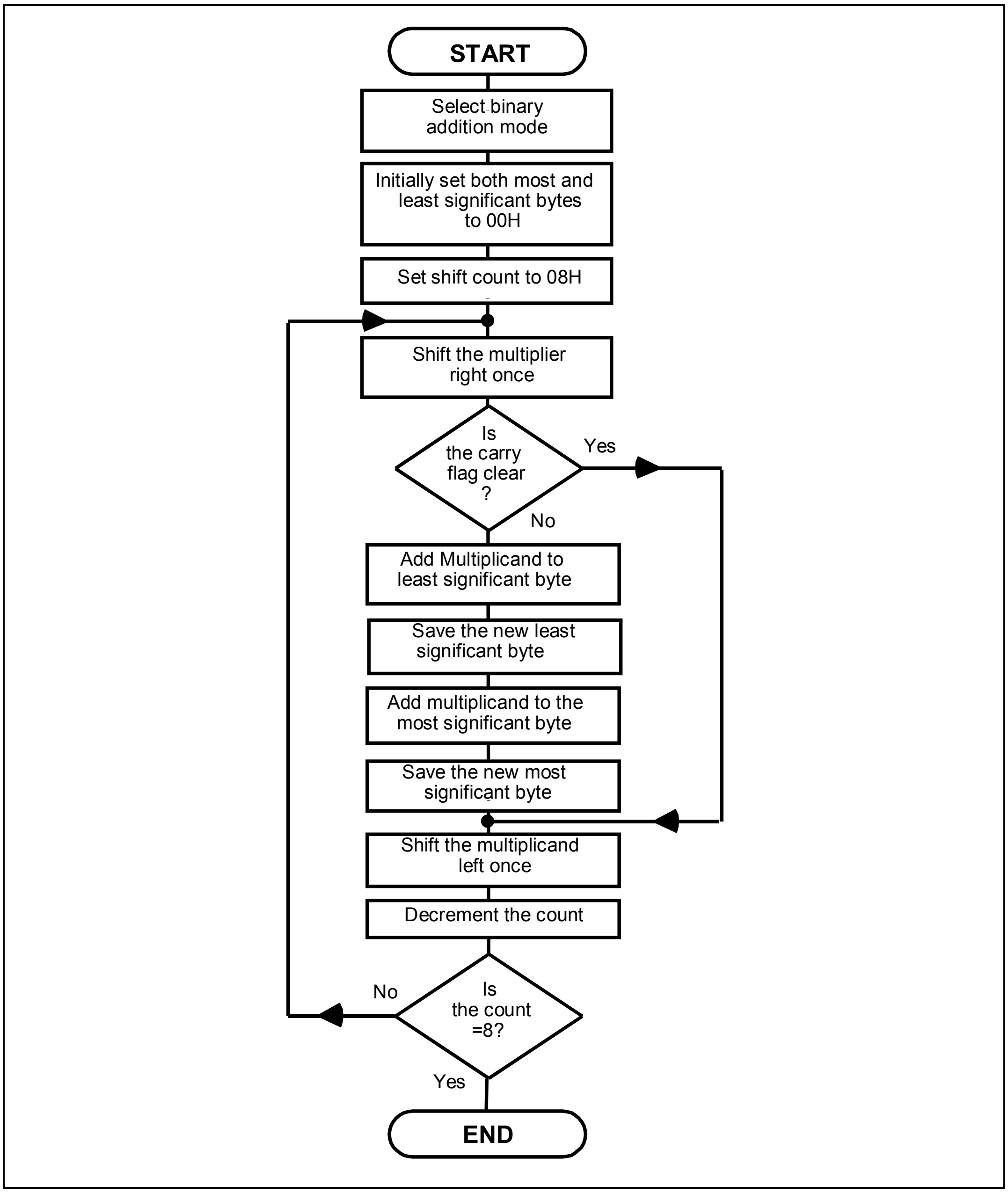
c `ROL  A`

d `ROR  A`

12.6 Worked Example

Write a program that will multiply together the contents of locations 0500_H and 0501_H, saving the most significant byte of the result in location $00F0_H$ and the least significant byte in location $00F1_H$.

Note: The two largest possible 8-bit values (FF_H and FF_H) will give a result $FE01_H$. So, although the result may well exceed 8 bits, it cannot exceed 16 bits.



The Assembly Language program will be:

```
                              ORG    $0400       ;Defines the start address
 0400   D8                    CLD                ;Selects binary addition mode
 0401   A9                    LDA    #$00
 0402   00
 0403   85                    STA    $F0         ;Clears Most Significant
 0404   F0                                       ;store
 0405   85                    STA    $F1         ;Clears Least Significant
 0406   F1                                       ;store
 0407   85                    STA    $FF         ;Clears Temporary Store
 0408   FF
 0409   A2                    LDX    #$08        ;Sets loop count to 8
 040A   08
 040B   4E    LOOP:           LSR    $0501       ;Shifts multiplier right
 040C   01                                       ;once
 040D   05
 040E   90                    BCC    CLEAR       ;If carry is clear, no
 040F   0E                                       ;addition so branch over
                                                 ;addition section
 0410   18                    CLC
 0411   A5                    LDA    $F1         ;Reads the current least
 0412   F1                                       ;significant byte
 0413   6D                    ADC    $0500       ;Adds multiplicand to
 0414   00                                       ;least significant byte
 0415   05
 0416   85                    STA    $F1         ;Saves new least
 0417   F1                                       ;significant byte
 0418   A5                    LDA    $F0         ;Reads the current most
 0419   F0                                       ;significant byte
 041A   65                    ADC    $FF         ;Adds multiplicand from
 041B   FF                                       ;temporary store
 041C   85                    STA    $F0         ;Saves new most
 041D   F0                                       ;significant byte
 041E   0E    CLEAR:          ASL    $0500       ;Shifts multiplicand left
 041F   00                                       ;once
 0420   05
 0421   26                    ROL    $FF         ;Rotates multiplicand
 0422   FF                                       ;left once
 0423   CA                    DEX                ;Reduces count by 1
 0424   DO                    BNE    LOOP        ;Repeat from location
 0425   E5                                       ;040AH until 8 shifts are
                                                 ;completed
 0426   60                    RTS                ;Returns to MAC III system
```

12.6a **Load the program for Worked Example 12.6 into the MAC III. Use this program to calculate $6A_H$ x 92_H. Enter the hexadecimal result that you obtain.**

Student Assessment 12

1. **When the binary number $1001\ 1001_2$ is logically ANDed with the mask $1111\ 0000_2$, the result is:**

a $0101\ 0111_2$

b $1000\ 1001_2$

c $1001\ 0000_2$

d $1111\ 1001_2$

2. **The mask required to test bits 6, 3 and 0 of the Accumulator is:**

a 24_H

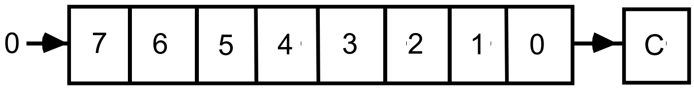
b 49_H

c 57_H

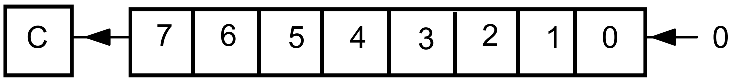
d 92_H

3. **The Shift Right instruction (LSR) can be represented as:**

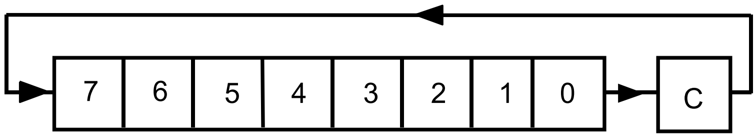
a



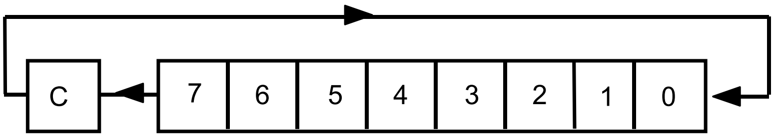
b



c



d



Continued ...

Student Assessment 12 Continued ...

4. The 6502 Assembly Language instruction that allows the Accumulator to be ANDed with a memory location but that does not change the contents of either is:

a `AND`

b `BIT`

c `LSR`

d `ROL`

5. Shifting a register one place to the left has the effect of:

a addition of 2

b subtraction of 2

c multiplication by 2

d division by 2

6. The Accumulator initially contains 34_H. After the instruction "`AND   #$EB`" has been executed, the contents of the Accumulator will be:

a 10_H

b 20_H

c 40_H

d 80_H

7. Initially, memory location 0600_H contains the value 70_H and the Accumulator contains $2D_H$. After the instruction "`BIT   $0600`" has been executed, the contents of the Accumulator will be:

a 07_H

b $0E_H$

c 20_H

d $2D_H$

Chapter 13 Input and Output Programming

Objectives of this Chapter

Having studied this chapter you will be able to:

- Describe the use of the LOAD and STORE instructions for data input and output respectively.
- Write programs which configure the 6522 VIA Data Ports as Inputs, Outputs or a mixture of both.
- Write programs which output and input data.
- Write programs to produce delays of given durations.

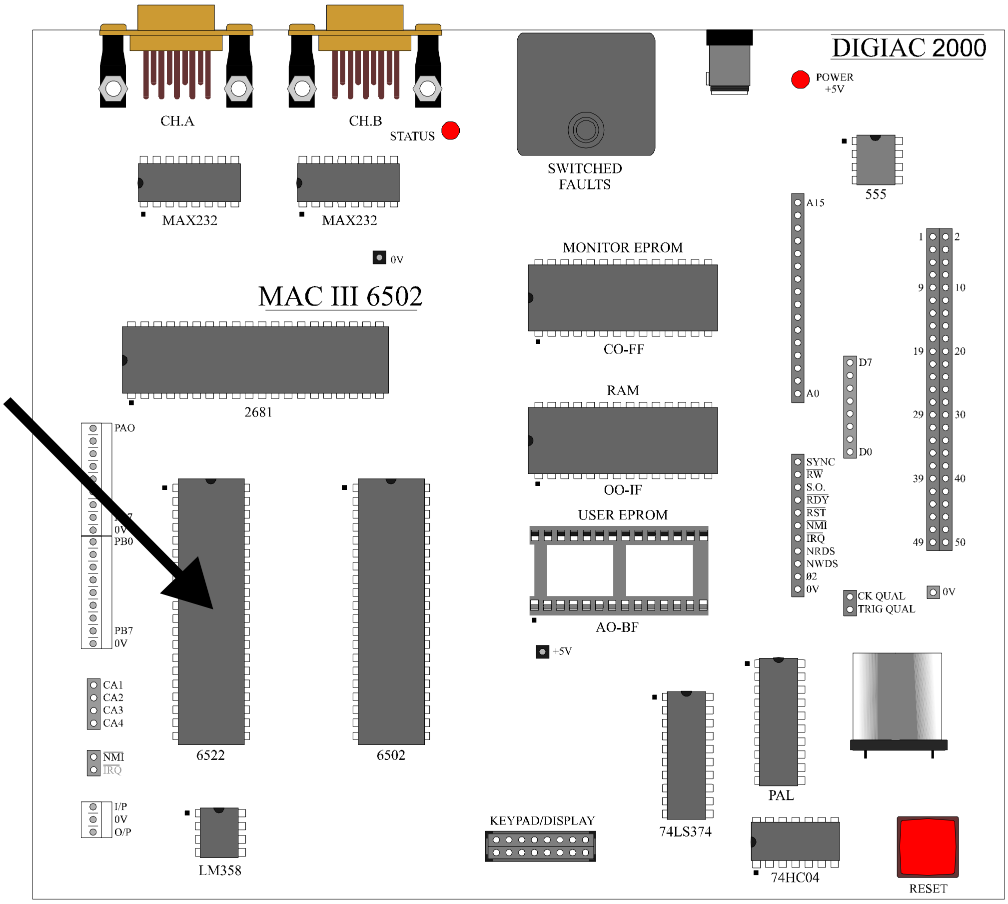
Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- 6502 Instruction Set Reference Manual.
- MAC III 6502 User Manual.

Introduction

Data enters and leaves the microcomputer via **Data Ports**. A port usually comprises 8 parallel connections to the external environment. These ports are often within programmable devices that allow the user to specify any desired combination of inputs **or** outputs. Ports may therefore be Output Ports, Input Ports or a mixture of both.

The MAC microcomputer uses a 6522 VIA (Versatile Interface Adapter).



MAC III Board showing 6522

Similar devices are also called PIO (Parallel Input/Output) and PIA (Programmable Interface Adapter). The 6522 has two 8-bit ports called Port A and Port B. Both Ports can be configured under program control to provide any desired combination of inputs and outputs.

13.1 Input and Output Instructions

Ports A and B are "memory mapped". This essentially means that they appear to be memory locations to the 6502. It follows therefore that the LOAD instruction can be used to read data in from an input port and the STORE instruction to send data out from an output port.

In the MAC III microcomputer, the addresses of the 6522 VIA registers that allow data to be read from, or written to, the data ports are:

Port A Data Register (PADR)	9001_H
Port B Data Register (PBDR)	9000_H

Examples:

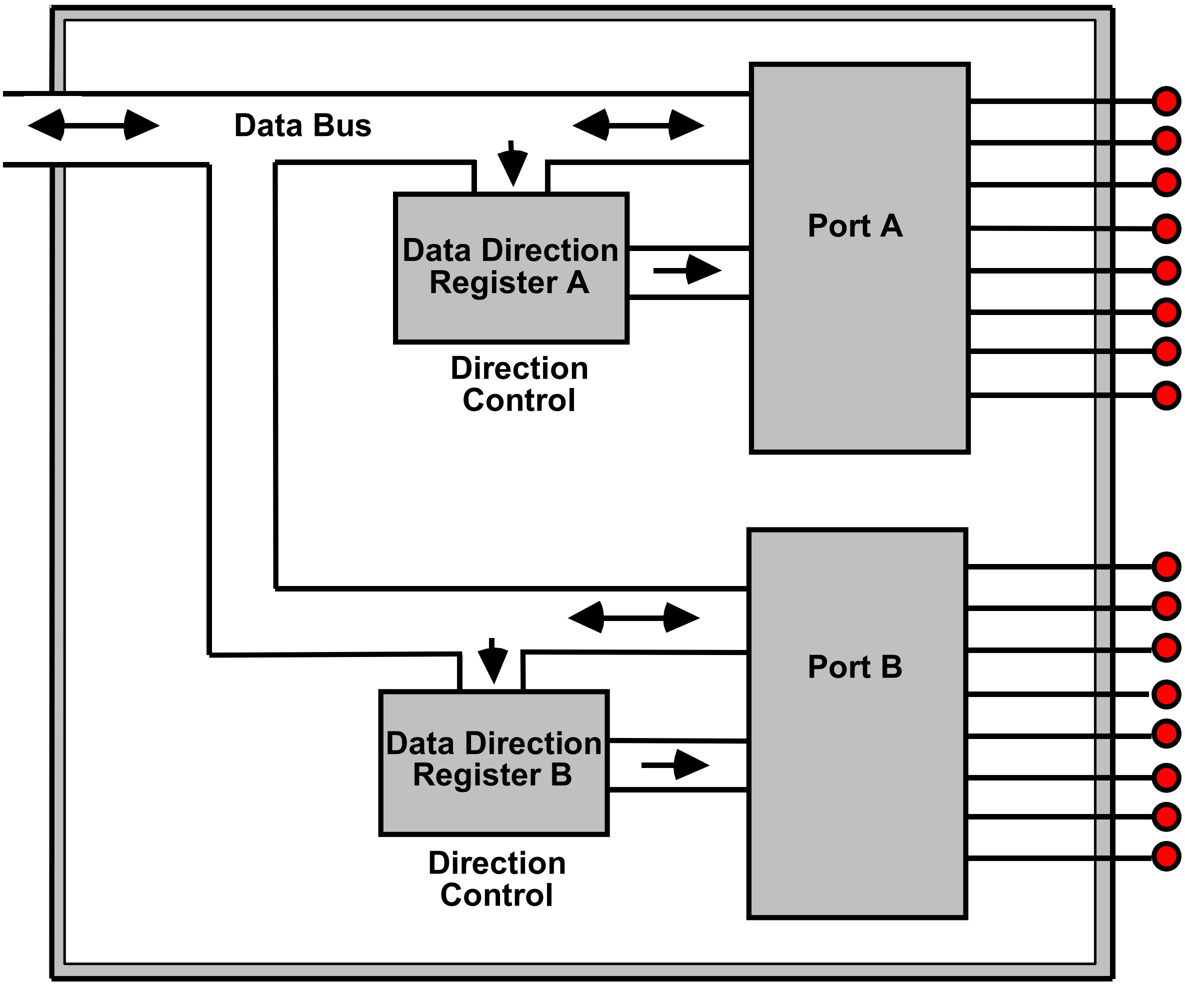
```
0456   8D   STA PADR       ;Outputs the accumulator
0457   01                  ;contents at Port A
0458   90
```

So, if the accumulator contained 99_H then the bit pattern $1001\ 1001_2$ would appear at Port A.

```
048C   AD   LDA PBDR       ;Inputs the contents of
048D   00                  ;Port B to the accumulator
048E   90
```

So, if the bit pattern $0001\ 0101_2$ is presented at Port A, the accumulator will be loaded with 15_H.

Each of the Port bits is **individually programmable** as an input **or** an output bit, using the **Data Direction Register** for Port A or Port B as appropriate.



Block diagram of 6522 Port Registers

The programming of the Data Direction Registers is quite simple: A "1" in any bit position of the Data Direction Register makes the corresponding Port bit an **output**.

For example, if Data Direction Register A holds the value 0011 0111_2:

Bits 7, 6 and 3 of Port A are **inputs**
Bits 5, 4, 2, 1 and 0 of Port A are **outputs**

In the MAC III microcomputer, the addresses of Port A and Port B Data Direction Registers (PADDR & PBDDR) are:

PADDR	9003_H
PBDDR	9002_H

Clearly then, the LOAD instruction can also be used to program the Data Direction Registers.

For example, to make Port A all output bits:

```
04A2   A9   LDA  #$FF     ;Loads accumulator with
04A3   FF                 ;1111 1111 binary
04A4   8D   STA  PADDR    ;Saves required Input/Output
04A5   03                 ;bit pattern in Port A Data Direction
04A6   90                 ;Register
```

Similarly, to make Port B all input bits:

```
04D5   A9   LDA  #$00    ;Loads accumulator with
04D6   00                ;0000 0000 binary
04D7   8D   STA  PBDDR   ;Saves required Input/Output
04D8   02                ;bit pattern in Port B Data Direction
04D9   90                ;Register
```

Now recall that each bit of a Data Port is **individually** programmable as an input or an output. So, for example, to make Port A bits 7, 6, 2 and 0 outputs and bits 5, 4, 3 and 1 inputs:

```
04F0   A9   LDA  #$C5    ;Loads accumulator with
04F1   C5                ;1100 0101 binary
04F2   8D   STA  PADDR   ;Saves required Input/Output
04F3   03                ;bit pattern in Port A Data Direction
04F4   90                ;Register
```

13.1a **The instruction that is used to output data from Port B of the MAC III 6522 VIA is:**

a `LDA PBDR`

b `LDA PADR`

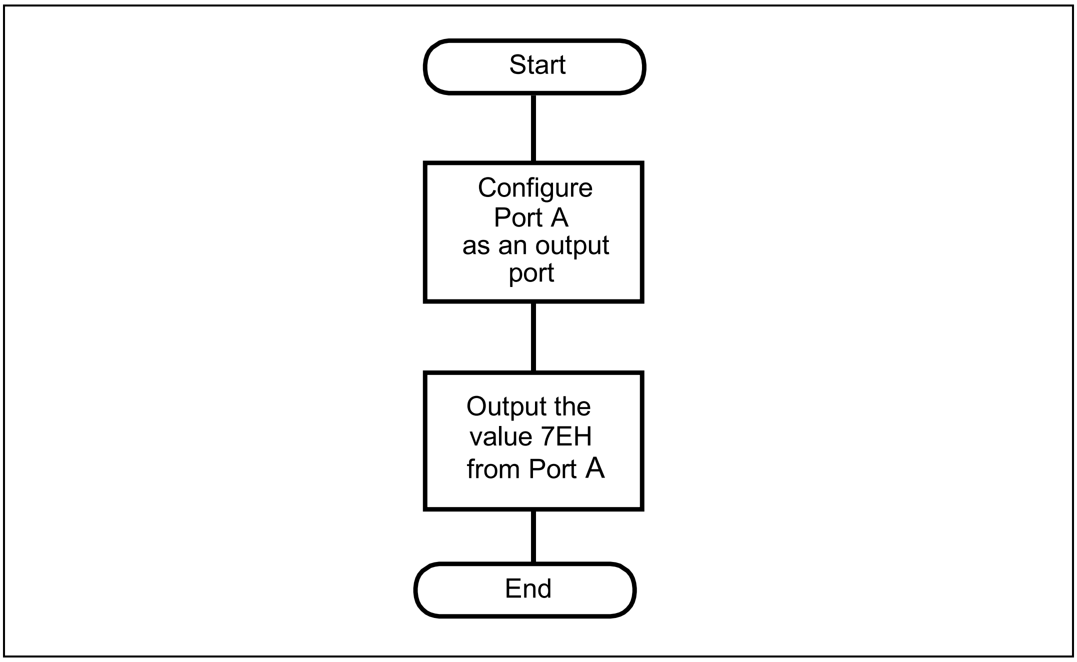
c `STA PBDR`

d `STA PADR`

13.1b **All bits of Port A are to be programmed as inputs. Enter the hexadecimal value that must be written to Port A Data Direction Register.**

13.2 Worked Example

Write a program that will output the value $7E_H$ from Port A.



The Assembly language program will be:

```
               PADR:   EQU    $9001
               PADDR:  EQU    $9003

                       ORG    $0400      ;Defines the start address
0400    A9             LDA    #$FF       ;Loads accumulator with
0401    FF                               ;1111 1111 binary
0402    8D             STA    PADDR      ;Makes Port A all output bits
0403    03
0404    90
0405    A9             LDA    #$7E       ;Loads accumulator with
0406    7E                               ;7EH
0407    8D             STA    PADR       ;Outputs accumulator contents
0408    01                               ;at Port A
0409    90
040A    60             RTS               ;Returns to MAC III system
```

Note that an assembly language 'EQU' directive is required by the 6502 Cross Assembler, in order to define the address represented by each label used in the program. If you are not using the 6502 Cross Assembler software, the EQU statements may be ignored.

Make sure that you have connected the Applications Module to the MAC III circuit board and to the power supply (refer to the MAC III User Manual for further guidance).

Load the program into MAC III memory and execute. You should see the bit pattern for $7E_H$ ($0111\ 1110_2$) on the Applications Module, thus:

D7	D6	D5	D4	D3	D2	D1	D0
○	●	●	●	●	●	●	○

● LED lit
○ LED unlit

13.2a **The program for Worked Example 13.2 is to be modified so that the byte which is output at Port A is 28_H. The instruction that must be changed is:**

- a `LDA #$FF`
- b `STA PADDR`
- c `LDA #$7E`
- d `STA PADR`

13.3 Practical Assignment

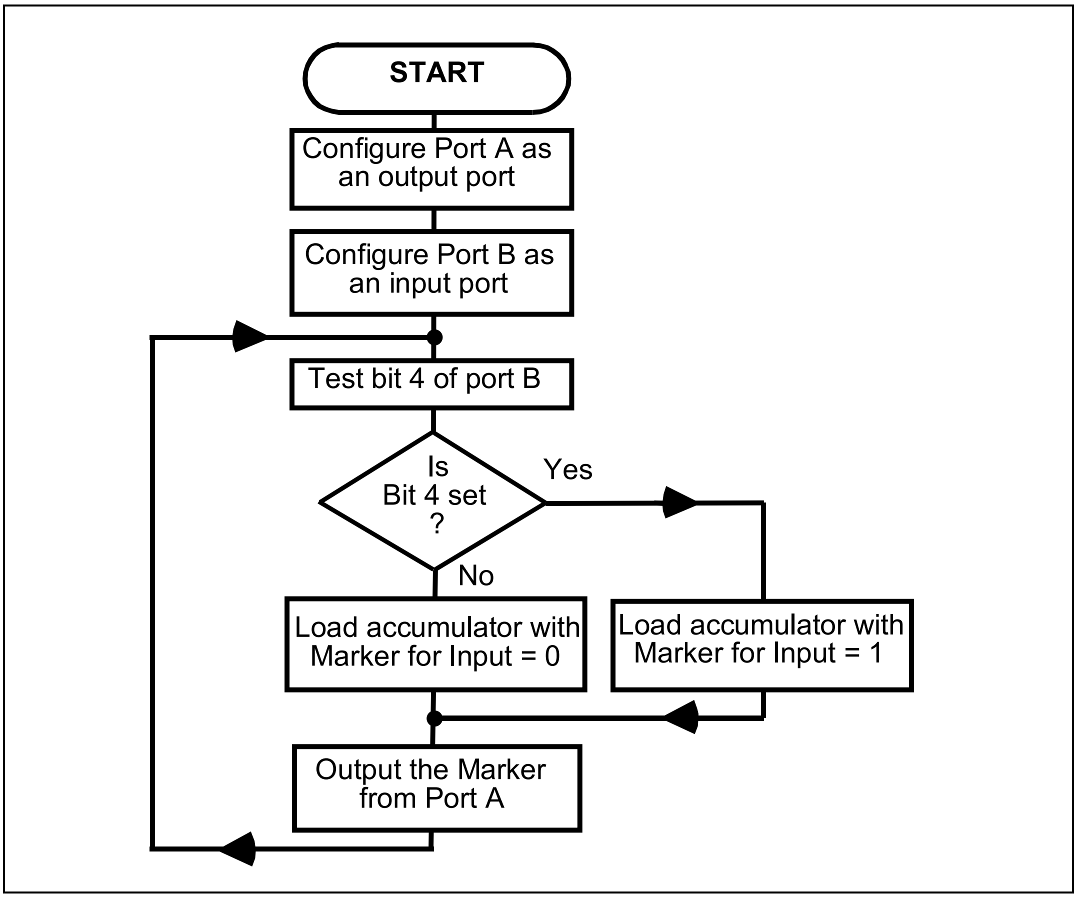
Write a program that will add the contents of memory locations 0040_H and 0041_H. The result should be output from Port A.

13.3a **Set the contents of memory location 0040_H to $1B_H$ and the contents of location 0041_H to $2F_H$. Run your program for Practical Assignment 13.3 and enter the hexadecimal value output at Port A.**

13.4 Worked Example

Write a program that will use the Applications Module motor disc detector as the input. If the input is a "1", output 07_H from Port A. If the input is "0", output 70_H from Port A. This is a very important exercise, since it is the first time that you will program the microcomputer to alter an **output** according to the state of an **input**. This is the basis of many microcomputer control programs. If you now rotate the motor disc on the applications module, you will see the Port B monitor LED for bit 4 change. If the LED is lit, a "1" is present. If the LED is unlit, a "0" is present. This can be used as the input for this exercise.

You will send an output marker value to Port A depending upon the state of this input. This program should loop back to keep checking the input and change the output as required. This is a fundamental process in continuous microcomputer control.



Notice how the program loops back. This will give a **continuous** loop. The output will change whenever the input changes.

The Assembly language program will be:

```
                PADR:         EQU    $9001
                PADDR:        EQU    $9003
                PBDR:         EQU    $9000
                PBDDR:        EQU    $9002

                              ORG    $0400       ;Defines the start address
0400    A9                    LDA    #$FF        ;Loads accumulator
0401    FF                                       ;with 1111 1111 binary
0402    8D                    STA    PADDR       ;Makes Port A all
0403    03                                       ;output bits
0404    90
0405    A9                    LDA    #$00        ;Loads accumulator
0406    00                                       ;with 0000 0000 binary
0407    8D                    STA  PBDDR         ;Makes Port B all
0408    02                                       ;output bits
0409    90
040A    A9      TESTB4:       LDA    #$10        ;Loads accumulator
040B    10                                       ;with mask for bit 4
040C    2C                    BIT    PBDR        ;Tests bit 4 of
040D    00                                       ;Port B
040E    90
040F    D0                    BNE    B4SET       ;Is bit 4 set ?
0410    08
0411    A9                    LDA    #$70
0412    70
0413    8D                    STA    PADR        ;Bit 4 not set so
0414    01                                       ;output 70H at
0415    90                                       ;Port A
0416    4C                    JMP    TESTB4      ;Loop back to test
0417    0A                                       ;bit 4 again
0418    04
0419    A9      B4SET:        LDA    #$07
041A    07
041B    8D                    STA    PADR        ;Bit 4 set so output
041C    01                                       ;07H at Port A
041D    90
041E    4C                    JMP    TESTB4      ;Loop back to test
041F    0A                                       ;bit 4 again
0420    04
```

Load this program into the MAC III and execute.

Rotate the motor disc and the input will switch between "1" and "0" (LED "on" and" off"). Check that the output LED's change between 07_H and 70_H as the input changes.

Note: Since this program contains a continuous loop, you will have to press the RESET button on the MAC III board to return control to the MAC III system.

13.4a **In the program for Worked Example 13.4, if the instruction** `" BNE  B4SET "` **is changed to** `"BEQ B4SET"`**, the program would:**

- a work in exactly the same way
- b always output 07_H
- c always output 70_H
- d output 07_H when the input is a '0' and 70_H when the input is a '1'

13.5 Time Delays

Often in Input/Output programs it will be necessary to provide a time delay. For example: to allow a peripheral device time to respond.

There are a number of ways of producing such delays. The simplest is to load a register or memory location with a value and then continually decrement the register or location until it reaches zero.

For example, using an Index Register:

```
0420   A2          LDX   #$60    ;Loads X-register with a count
0421   60
0422   CA   LOOP:  DEX           ;Reduce count by 01H
0423   D0          BNE   LOOP    ;If count is not yet zero,
0424   FD                        ;branch back to previous
                                 ;instruction
0425   60          RTS           ;Returns to MAC III system
```

Alternatively, using a memory location:

```
0430   A9          LDA   #$80
0431   80
0432   85          STA   $F0     ;Places count value in
0433   F0                        ;location 00F0H
0434   C6   LOOP:  DEC   $F0     ;Reduce count by 01H
0435   F0
0436   D0          BNE   LOOP    ;If count is not yet zero,
0437   FC                        ;branch back to previous
                                 ;instruction
0438   60          RTS           ;Returns to MAC III system
```

Now, the length of time delay produced will depend upon the initial value of the count. It will also depend upon the time each instruction within the loop takes to execute.

These times are given in the 6502 Instruction Set Reference Manual - the 6502 Instruction Set. Notice that times are expressed in terms of **clock cycles**, rather than in microseconds. This is to cater for a variety of clock frequencies.

The cycle time is related to the clock frequency thus:

$$\text{Cycle Time} = \frac{1}{\text{Clock Frequency}}$$

The MAC III microcomputer has a 1 MHz clock, giving a time for each cycle of 1µs. Consider the time delay example at the beginning of this section:

```
0420   A2          LDX   #$60      ;Loads X-register with a count
0421   60
0422   CA   LOOP:  DEX             ;Reduce count by 01H
0423   D0          BNE   LOOP      ;If count is not yet zero,
0424   FD                          ;branch back to previous
                                   ;instruction
0425   60          RTS             ;Returns to MAC III system
```

Now, the time taken to execute the first instruction (`LDX`) will be very small in comparison with the program loop and can almost always be ignored. From the 6502 Instruction Set Reference Manual you will see that " `DEX` " will have an execution time of 2 cycles and "`BNE LOOP`" has an execution time of 2 + 1 = 3 cycles (whenever the branch is taken - which will be every pass except the last).

Therefore each pass through the loop will take:

2 + 3 = 5 cycles.

Recall that each cycle is 1µS so each pass through the loop will take:

5 x 1 = 5µs.

Now, the count is initially set to 60_H which is 96_{10}. Hence the total delay will be:

96 x 5 = 480µs (0.48ms)

The maximum possible value for the initial count is FF_H (255_{10}). So the maximum possible delay using this structure is:

255 x 5 = 1275µs (1.275ms)

This technique can be extended to produce longer delays by **nesting** a second loop with the first.

So, for example:

```
0400   A2          LDX   CNT1       ;Loads X-register
0401   60                           ;with first count
0402   A0          LDY   CNT2       ;Loads Y-register
0403   FF                           ;with second count
0404   88   DCNT:  DEY              ;Reduce first count by 01H
0405   D0          BNE   DCNT       ;If first count is not yet zero,
0406   FD                           ;branch back to decrement first
                                    ;count again
0407               DEX              ;Reduce second count by 01H
0408   D0          BNE   DCNT       ;If second count is not yet
0409   FA                           ;zero, branch back to decrement
                                    ;first count again
040A   60          RTS              ;Returns to MAC III system
```

The action of this delay technique can be likened to a clock: The Y register represents seconds and the X register minutes. The least significant loop (based on the Y-Register) will produce a delay of:

255 x 5 = 1275µs (1.275ms)

The most significant loop (based on the X-Register) will, in this case, be executed 60_H (96_{10}) times. So the total delay will be:

96 x 1.275 = 122.4ms (0.1224s)

Note that the maximum delay which may be produced will be:

255 x 1.275 = 325.125ms (0.325125s)

The NOP Instruction

Using the dummy instruction NOP (No Operation) can produce very short delays. This instruction performs no function other than incrementing the program counter. The NOP instruction will produce a delay of 2 cycles (2µs for the MAC III system).

13.6 Worked Example

Write a program section that gives a delay of 1ms

Solution:

1ms = 1000µs

Time taken for one pass through simple loop = 5µs

$$\frac{1000\mu s}{5\mu s} = 200_{10}$$

So $\mathbf{200_{10}}$ ($C8_H$) is the value to be loaded into the register.

The program section will be:

```
0400   A2             LDX  #$C8      ;Loads X-register with a count
0401   C8
0402   CA   LOOP:     DEX            ;Reduce count by 01H
0403   D0             BNE  LOOP      ;If count is not yet zero,
0404   FD                            ;branch back to previous instruction
0405   60             RTS            ;Returns to MAC III system
```

13.6a The program for Worked Example 13.6 is to be modified to produce a delay of 800µs. Enter the hexadecimal value that the first instruction must load into the X Register.

13.7 Worked Example

Write a program section that gives a delay of 5ms

Solution:

5ms = 5000μs

Time taken for one pass through simple loop = 5μs

Maximum delay for a simple loop = 255 x 5 = 1275μs so nested loops must be used.

The first loop will give a delay of 1275μs so:

$$\frac{5000\mu s}{1275\mu s} = 3.9216_{10}$$

Now, since this is not a round number it must be rounded up to the nearest whole number: 4_{10}

Therefore **04_H** is the value to be loaded into the X-Register.

The program section will be:

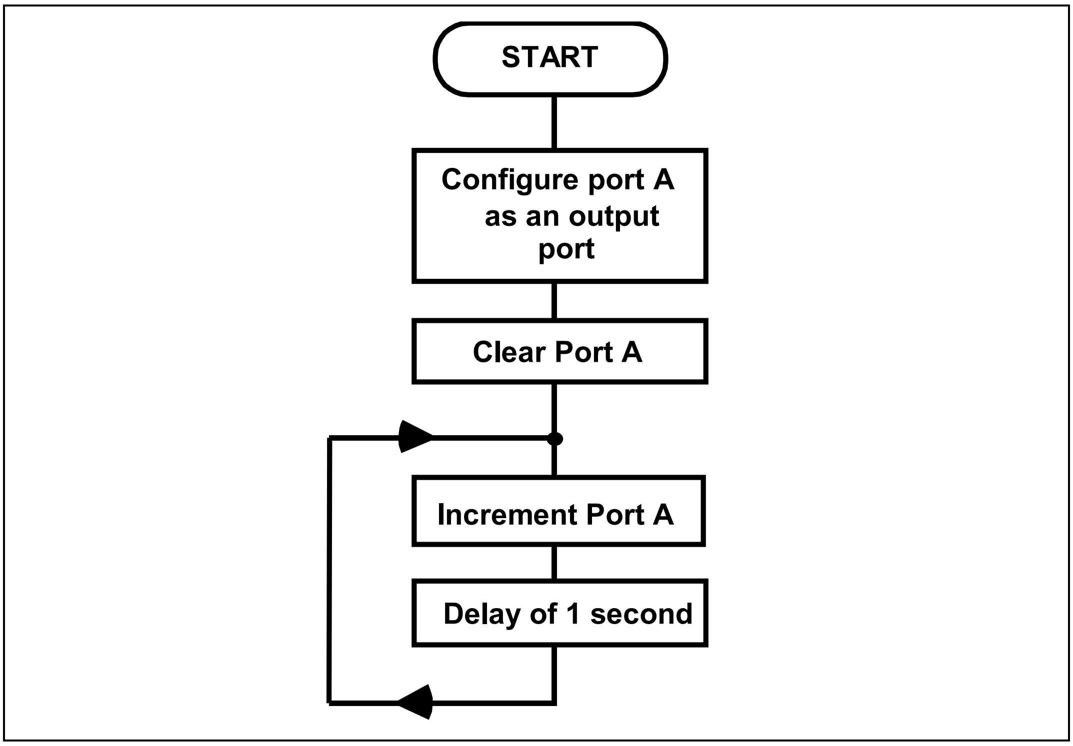
```
0400   A2               LDX    #$04       ;Loads X-register
0401   04                                 ;with first count
0402   A0               LDY    #$FF       ;Loads Y-register
0403   FF                                 ;with second count
0404   88   DCNT:       DEY               ;Reduce first count by 01H
0405   D0               BNE    DCNT       ;If first count is
0406   FD                                 ;not yet zero, branch back to
                                          ;decrement first count again
0407                    DEX               ;Reduce second count by 01H
0408   D0               BNE    DCNT       ;If second count is not yet
0409   FA                                 ;zero, branch back to decrement
                                          ;first count again
040A   60               RTS               ;Returns to MAC III system
```

13.7a **The program for Worked Example 13.7 is to be modified to produce a delay of 15.3ms. Enter the hexadecimal value that the first instruction must load into the X Register.**

13.8 Worked Example

Write a program that will produce an increasing binary count which changes about once per second.

This problem requires a delay of about 1 second between increments of Port A:



The maximum delay for a single nested loop is about 325ms. However, the least significant loop could be made longer by including several `NOP` instructions thus:

```
0500   A2               LDX    #$FF
0501   FF
0502   CA   LOOP:       DEX
0503   EA               NOP
0504   EA               NOP
0505   EA               NOP
0506   EA               NOP
0507   EA               NOP
0508   D0               BNE    LOOP
0509   F8
```

Each pass through this loop would give a delay of:

2 + 2 + 2 + 2 + 2 + 2 + 3 = 15 cycles = 15µs

The maximum delay which this loop could produce will be:

255 x 15 = 3825µs (3.825 ms)

If this is nested with another loop, the maximum overall delay will be:

255 x 3825 = 975375µs (0.98 s)

This is a quite acceptable approximation to the 1 second delay required.

So, the Assembly Language program will be:

```
            PADR:        EQU    $9001
            PADDR:       EQU    $9003

                         ORG    $0400      ;Defines the start address
0400   A9                LDA    #$FF
0401   FF
0402   8D                STA    PADDR      ;Makes Port A all output
0403   03                                  ;bits
0404   90
0405   A9                LDA    #$00
0406   00
0407   8D                STA    PADR       ;Clears Port A initially
0408   01
0409   90
040A   EE   INCNT:       INC    PADR       ;Increase count by 1
040B   01
040C   90
040D   A2                LDX    #$FF
040E   FF
040F   A0                LDY    #$FF       ;Initial values for delay
0410   FF
0411   CA   DCNT:        DEX
0412   EA                NOP
0413   EA                NOP
0414   EA                NOP
0415   EA                NOP
0416   EA                NOP
0417   DO                BNE    DCNT       ;Least significant delay
0418   F8                                  ;loop - 3.825 ms
0419   88                DEY
041A   D0                BNE    DCNT       ;Most significant delay
041B   F5                                  ;loop - 0.975 s
041C   4C                JMP    INCNT      ;Loop back to next
041D   0A                                  ;increment of Port A
041E   04
```

13.8a **Enter the delay in microseconds (µs) produced by a single** "`NOP`" **instruction.**

13.9 Practical Assignment

Write a program that will output a binary up-count, increasing by one about every 0.5 seconds at Port A. The Applications Module motor disc detector is to be used as an input. If the input is a "0", the binary count may continue. If the input is "1", the binary count should be suspended.

13.9a **Load your program for Practical Assignment 13.9 into the MAC III. Set the input to a logic "1" and run the program. Now set the input to logic "0" for 20 seconds and return it to logic "1". Enter the hexadecimal byte shown on the Port A monitor LED's.**

Student Assessment 13

1. Data enters and leaves the microcomputer by means of:

a a Data Direction Register

b a Data Port

c an Index Register

d the Status Register

2. The 6502 Assembly Language instruction that will read the data input at Port B is:

a INPUT

b LOAD

c READ

d STORE

3. The 6502 Assembly Language instruction "`STA  $9001`" **will:**

a copy the value input at Port A into the Accumulator

b copy the value input at Port B into the Accumulator

c output the contents of the Accumulator at Port A

d output the contents of the Accumulator at Port B

4. The bits of a 6522-VIA Port that are to be inputs have a logic 0 written into the:

a data input register

b data output register

c data direction register

d data port register

Student Assessment 13 Continued ...

5. The correct assembly language sequence required to output the value $D5_H$ from Port A on the MAC III is:

a
```
LDA  #$FF
STA  PADDR
LDA  #$D5
STA  PADR
```

b
```
LDA  #$00
STA  PADDR
LDA  #$D5
STA  PADR
```

c
```
LDA  #$D5
STA  PADDR
LDA  #$00
STA  PADR
```

d
```
LDA  #$D5
STA  PADDR
LDA  #$FF
STA  PADR
```

6. The 6502 Assembly Language instruction sequence:

```
LDA  #$0F
STA  $9002
```

will configure Port B:

a as all inputs

b as all outputs

c bits 0, 1, 2 and 3 as inputs and bits 4, 5, 6 and 7 as outputs

d bits 0, 1, 2 and 3 as outputs and bits 4, 5, 6 and 7 as inputs

Continued ...

Student Assessment 13 Continued ...

7. The time taken by the MAC III to execute a "`DEX`" instruction is:

a 1 µs

b 2 µs

c 3 µs

d 4 µs

8. The delay produced in the MAC III by the 6502 assembly Language program:

```
0400 A2       LDX  #$20
0401 20
0402 CA LOOP: DEX
0403 D0       BNE  LOOP
0404 FD
0405 60       RTS
```

will be:

a 160 µs

b 180 µs

c 240 µs

d 360 µs

Chapter 14 Programming the Applications Module

Objectives of this Chapter

Having studied this chapter you will be able to:

- Describe the operation of each section of the Applications Module:

  Piezo Sounder
  Ultrasonic Transmitter and Receiver
  Digital to Analog Converter
  Analog to Digital Converter
  Optical Sender and Receiver
  Optical Disc Encoder

- Write programs to control each section of the Applications Module.

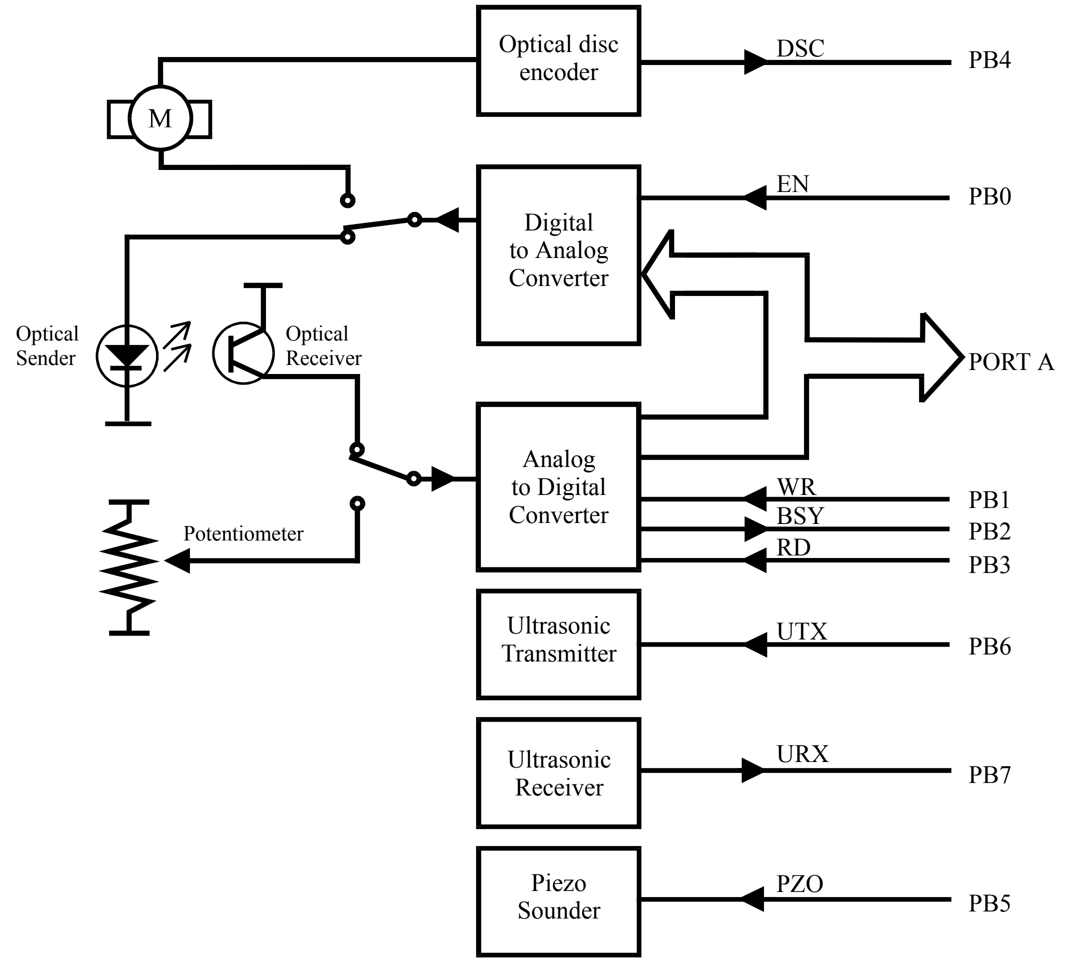
Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

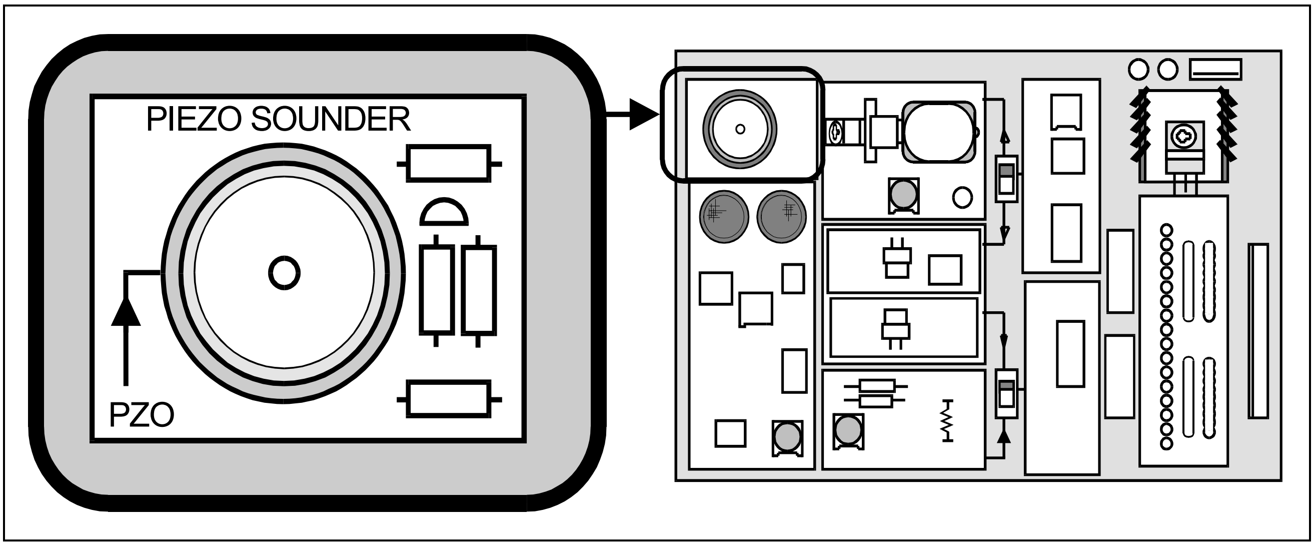
In the first chapter in this manual you learned how to run demonstration programs to control each section of the Applications Module. In other chapters you have learned how to program the microcomputer to make decisions and how to input and output data.

In this chapter you will combine these skills and write programs to control each individual section of the Applications Module.

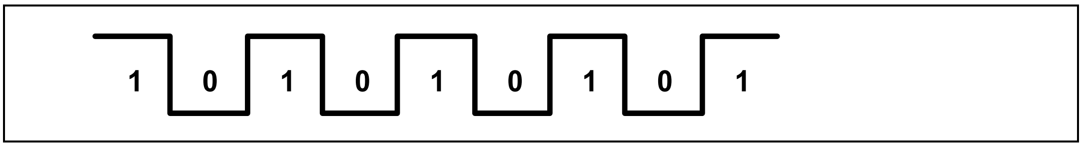


Applications Module Block Diagram

14.1 Piezo Sounder



The piezo sounder converts a TTL level waveform on Port B, bit 5 (PB5) into an audio signal of the same frequency. Changing the logic level on PB5 with respect to time will generate a TTL waveform thus:



14.2 Worked Example

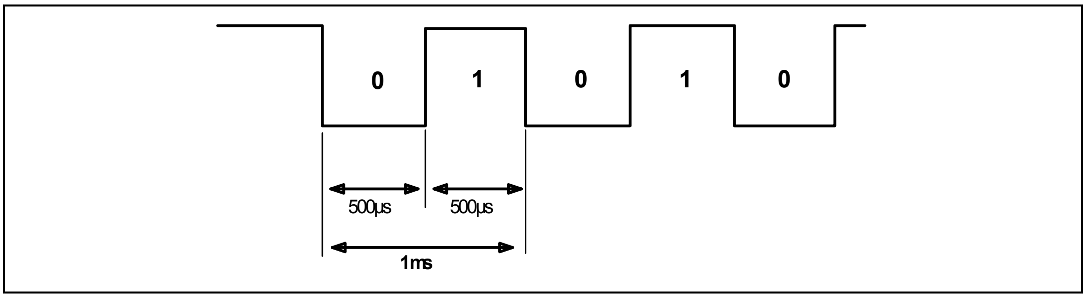
Write a program that will sound the Piezo Sounder at 1kHz.

Solution

This problem requires a square wave to be output to Port B, bit 5 (PB5). This is achieved by alternating the output between 0 and 1. Such a solution will, however not produce an audible output. It is necessary therefore to introduce a delay between changes of the output. For example, to produce a sound at 1kHz:

$$\text{Period} = \frac{1}{\text{Frequency}} = \frac{1}{1000} = \underline{1\text{ms}}$$

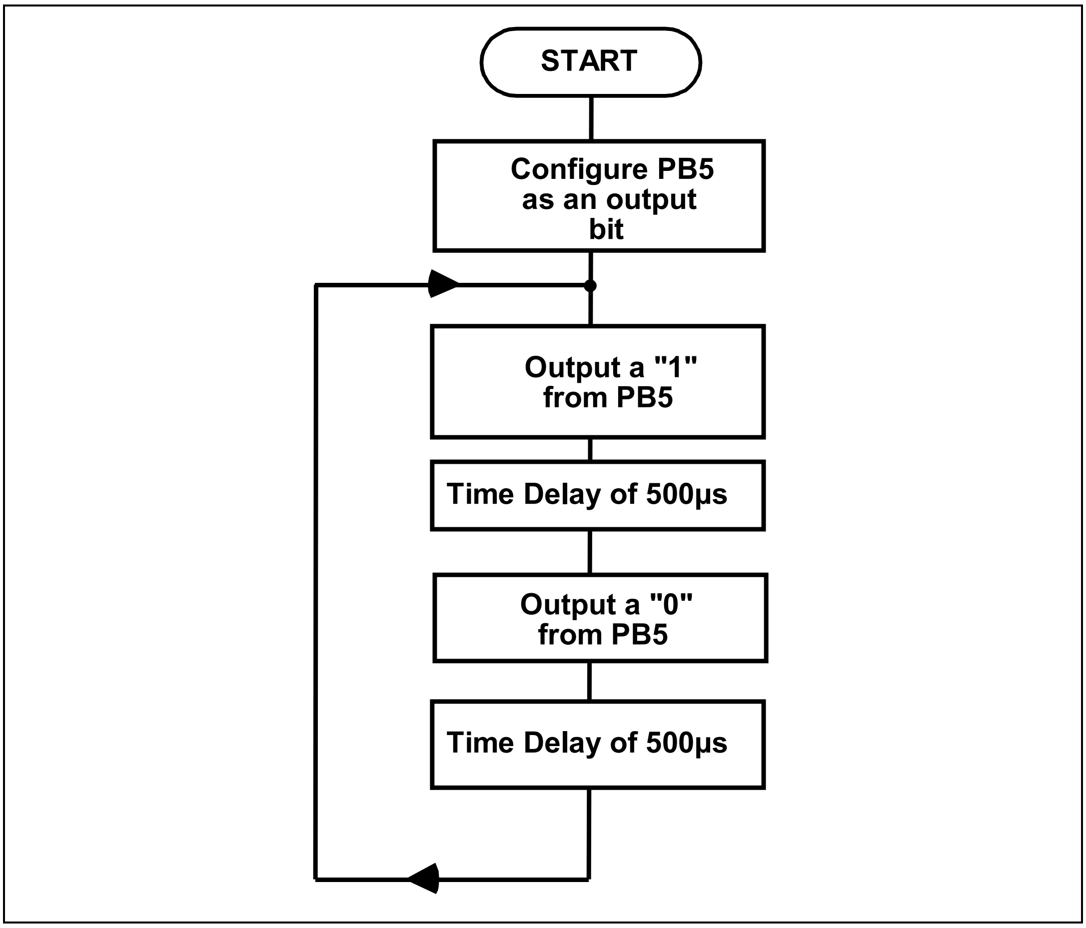
Now, the output must change **twice** during the period thus:



So a delay of 500µs will be required. The simple delay loop produces a delay of 5µs for each pass, so the number of passes required is:

$$\frac{500\mu s}{5\mu s} = 100_{10}\ (64_H)$$

Flowchart



The Assembly Language Program will be:

```
             PBDR:        EQU  $9000
             PBDDR:       EQU  $9002

                          ORG  $0400       ;Defines the start address
0400   A9                 LDA  #$20
0401   20
0402   8D                 STA  PBDDR       ;Makes Port B bit 5 (PB5)
0403   02                                  ;an output bit
0404   90
0405   A9    HIOUT:       LDA  #$20
0406   20
0407   8D                 STA  PBDR        ;Outputs a "1" on PB5
0408   00
0409   90
040A   A2                 LDX  #$64        ;Loads count for delay
040B   64
040C   CA    DELAY1:      DEX
040D   D0                 BNE  DELAY1      ;Delay of 500us
040E   FD
040F   A9                 LDA  #$00
0410   00
0411   8D    LOWOUT:      STA  PBDR        ;Outputs a "0" on PB5
0412   00
0413   90
0414   A2                 LDX  #$64        ;Loads count for delay
0415   64
0416   CA    DELAY2:      DEX
0417   D0                 BNE  DELAY2      ;Another delay of 500us
0418   FD
0419   4C                 JMP  HIOUT       ;Loop back to output a
041A   05                                  ;"1" on PB5
041B   04
```

14.2a **In Worked Example 14.2, the effect of reducing the delay between each change of the output state to 100µs will change the frequency of the sound emitted to:**

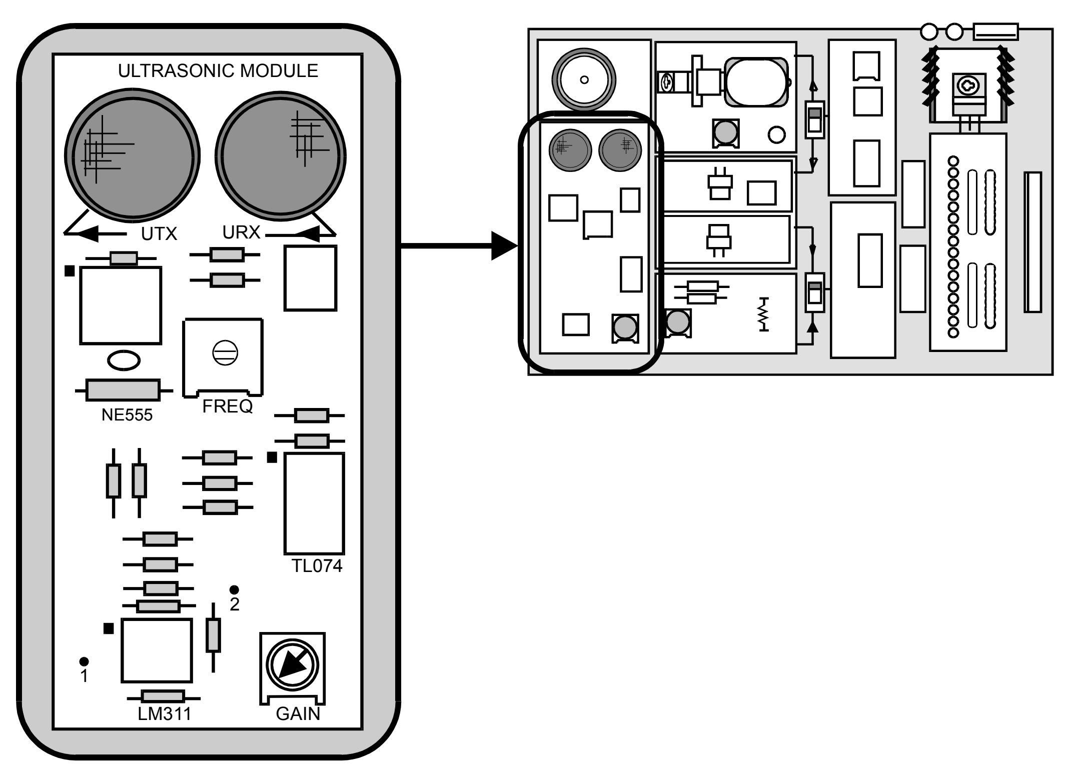
a 5kHz

b 10kHz

c 50kHz

d 100kHz

14.3 Ultrasonic Transmitter and Receiver



The Ultrasonic Transmitter is driven by a 40kHz oscillator within the Ultrasonic Module. The transmitter is switched on/off by the state of PB6 (labeled UTX):

PB6 = 1	**Transmitter ON**
PB6 = 0	**Transmitter OFF**

The Ultrasonic Receiver will detect the 40kHz ultrasound signal and pass an indication to PB7 (labeled URX) thus:

No 40kHz Detected:	**PB7 = 1**
40kHz Detected:	**PB7 = 40kHz TTL Squarewave**

The Transmitter and Receiver can be used together to detect reflections from an object placed directly above the module.

The sensitivity of the receiver circuit is set by the Gain control potentiometer. This allows the threshold at which signals are detected to be varied.

The Module can be used as a Proximity detector by generating an ultrasound signal and then monitoring the output from the receiver.

If any 40kHz signals appear on URX (PB7) then an object must be reflecting the ultrasound transmission.

The Applications Module User Manual also describes how to use this module for measuring distance. Recall that this was one of the demonstration programs.

14.3a **The Ultrasonic Transmitter is switched on by applying a:**

a logic "0" at PB6

b logic "1" at PB6

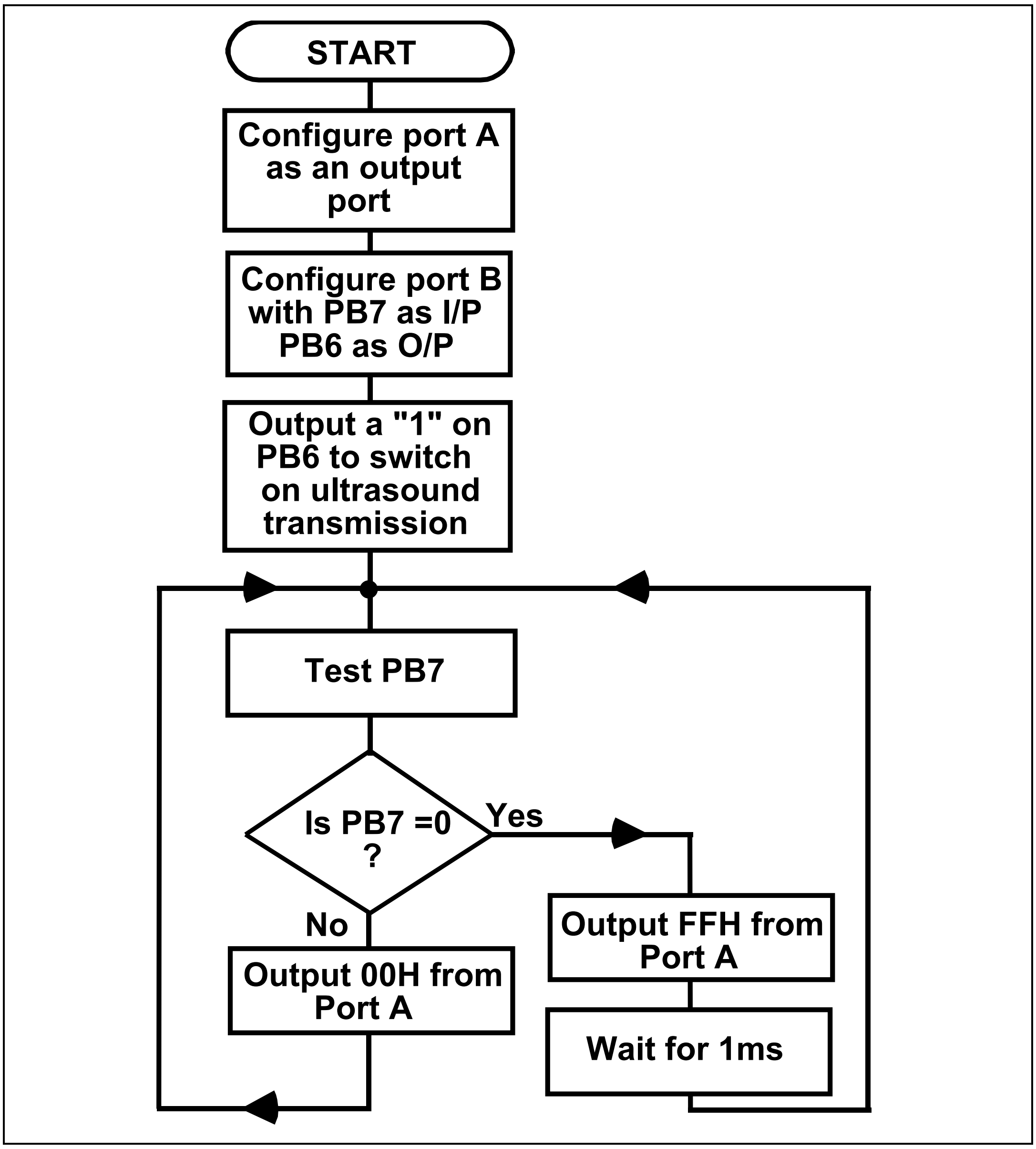
c logic "0" at PB7

d logic "1" at PB7

14.4 Worked Example

Write a program that will use the Ultrasonic Units within the Applications Module to act as a proximity detector. When an object is placed directly above the Ultrasonic Unit, all of the Port A monitor LED's should be lit.

Solution



The Assembly Language Program will be:

```
              PADR:         EQU    $9001
              PADDR:        EQU    $9003
              PBDR:         EQU    $9000
              PBDDR:        EQU    $9002

                            ORG    $0400         ;Defines the start address
0400    A9                  LDA    #$FF
0401    FF
0402    8D                  STA    PADDR         ;Sets Port A to all
0403    03                                       ;Output Bits
0404    90
0405    A9                  LDA    #$40
0406    40
0407    8D                  STA    PBDDR         ;Sets Port B: PB7=I/P,
0408    02                                       ;PB6=O/P, other bits
0409    90                                       ;don't care
040A    A9                  LDA    #$40
040B    40
040C    8D                  STA    PBDR          ;Outputs a "1" on PB6 to
040D    00                                       ;switch on Ultrasonic
040E    90                                       ;Transmitter
040F    A9    TSTPB7:       LDA    #$80          ;Mask for Bit 7
0410    80
0411    2C                  BIT    PBDR          ;Test PB7
0412    00
0413    90
0414    F0                  BEQ    LEDON         ;If PB7=0, branch to
0415    08                                       ;light LED's section
0416    A9                  LDA    #$00
0417    00
0418    8D                  STA    PADR          ;PB7=1 so switch off LED's
0419    01
041A    90
041B    4C                  JMP    TSTPB7        ;Jump back to test PB7
041C    0F                                       ;again
041D    04
041E    A9    LEDON:        LDA    #$FF
041F    FF
0420    8D                  STA    PADR          ;PB7=0 so switch on LED's
0421    01
0422    90
0423    A2                  LDX    #$C8          ;Sets initial value for
0424    C8                                       ;delay counter
0425    CA    WAIT:         DEX
0426    D0                  BNE    WAIT          ;Wait with Leeds on for about
0427    FD                                       ;1ms
0428    4C                  JMP    TSTPB7
0429    0F
042A    04
```

Load the program into the MAC III and execute.

Note: You will need to adjust the GAIN control in the Ultrasonic Module block to avoid false triggering.

This program could form the basis of an intruder alarm or automatic counter. Note the delay of approximately 1ms after the Port A LED's are lit - this delay ensures that the LED's are not switched off again when the detected 40kHz square wave next goes high.

14.4a **In Worked Example 14.4, the effect of changing the second** "`LDA  #$40`" **instruction to** "`LDA  #$00`" **would be to:**

a disable the Ultrasonic Transmitter

b enable the Ultrasonic Transmitter

c disable the Ultrasonic Receiver

d enable the Ultrasonic Receiver

14.5 Practical Assignment

Write a program that uses the Ultrasonic Units within the Applications Module to act as a proximity detector. When an object is placed directly above the Ultrasonic Unit, the Piezo Sounder should be activated.

14.5a **Run your program for Practical Assignment 14.5. The status of the "PZO" and "URX" LED's when the alarm is sounding are:**

a PZO LED off and URX LED off

b PZO LED off and URX LED on

c PZO LED on and URX LED off

d PZO LED on and URX LED on

14.5b **In your program for Practical Assignment 14.5, the data bits that were written to bit positions 7, 6 and 5 respectively of Data Direction Register B were:**

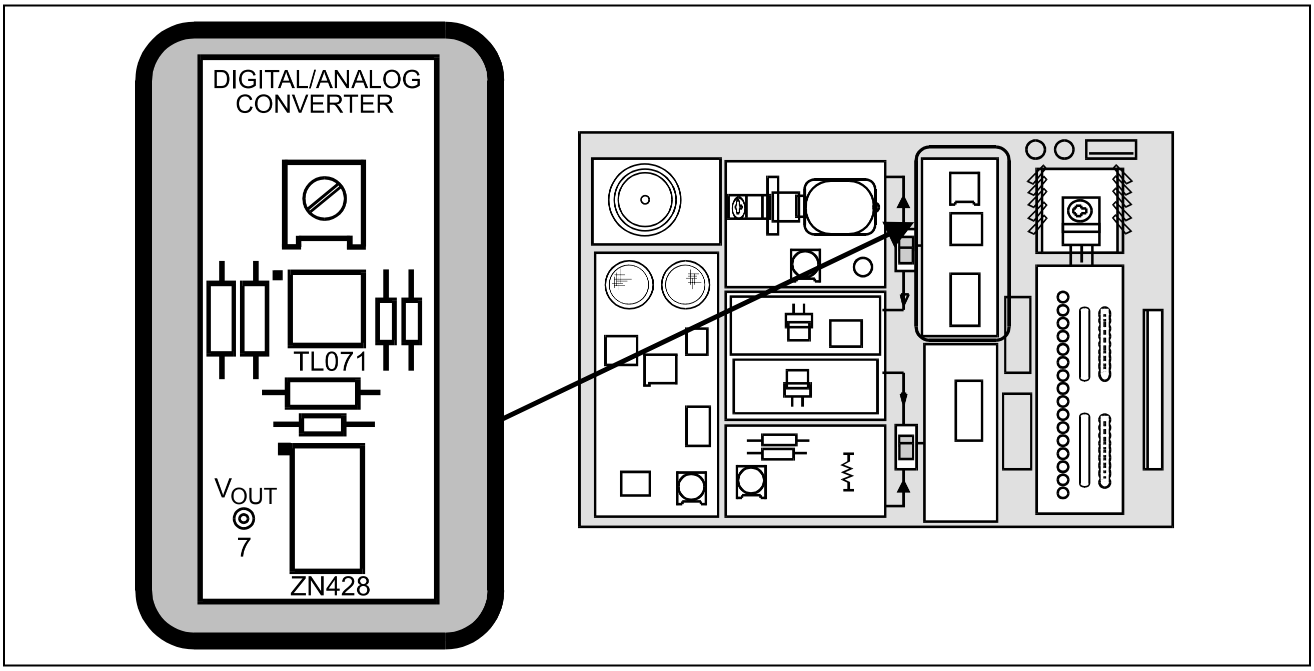
a 0, 0, 0

b 0, 1, 0

c 0, 1, 1

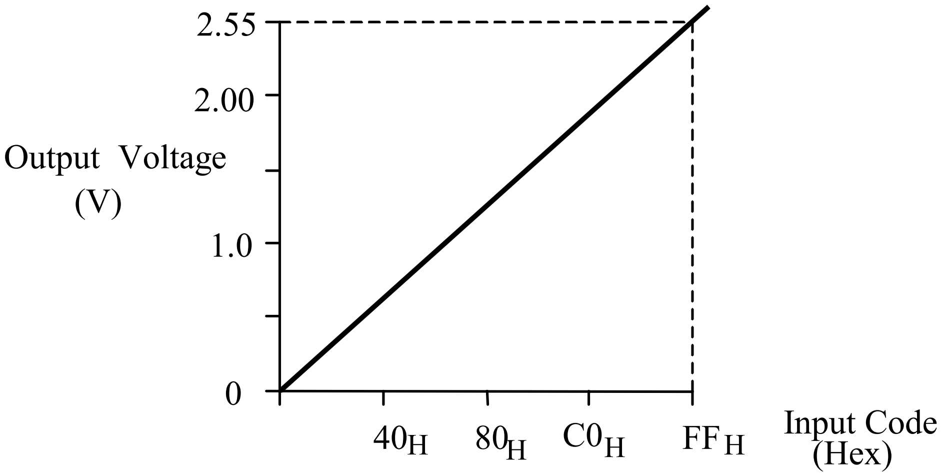
d 1, 0, 0

14.6 Digital to Analog Converter



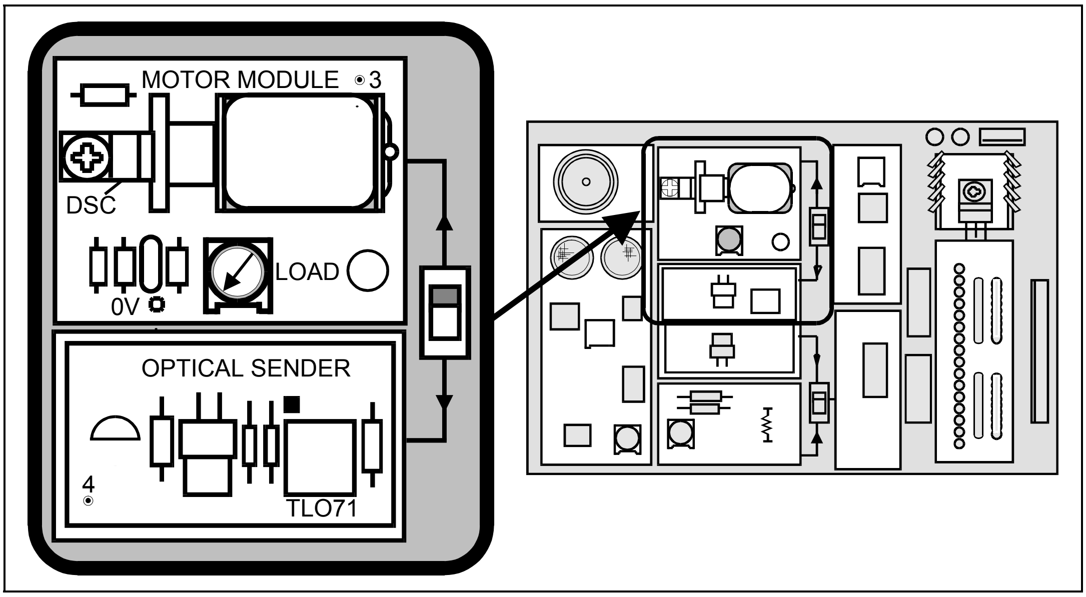
A Digital to Analog Converter (DAC) is necessary if a microprocessor-based control system is to produce an **analog** output. A DAC takes a digital value and **represents** it as a voltage level.

The Applications Module DAC has an 8-bit input. The output can range from 0V to 2.55V thus:



Notice that there are $FF_H = 255_{10}$ steps between 0V and 2.55V, so each increase in 1_H gives a voltage rise 0.01V (10mV).

The upper slider switch (DAC switch) on the Applications Module allows the output of the DAC to be applied to either the Optical Sender or the DC Motor:



The following sequence is required to initiate digital to analog conversion:

1. Output "0" on Port B, bit 0 (PB0) to enable the DAC
2. Output digital data from Port A

The voltage at the output will then be directly proportional to the input binary code.

14.6a **If an input code of 64_H is applied to the Applications Module Digital to Analog Converter (DAC), enter the output voltage (in volts).**

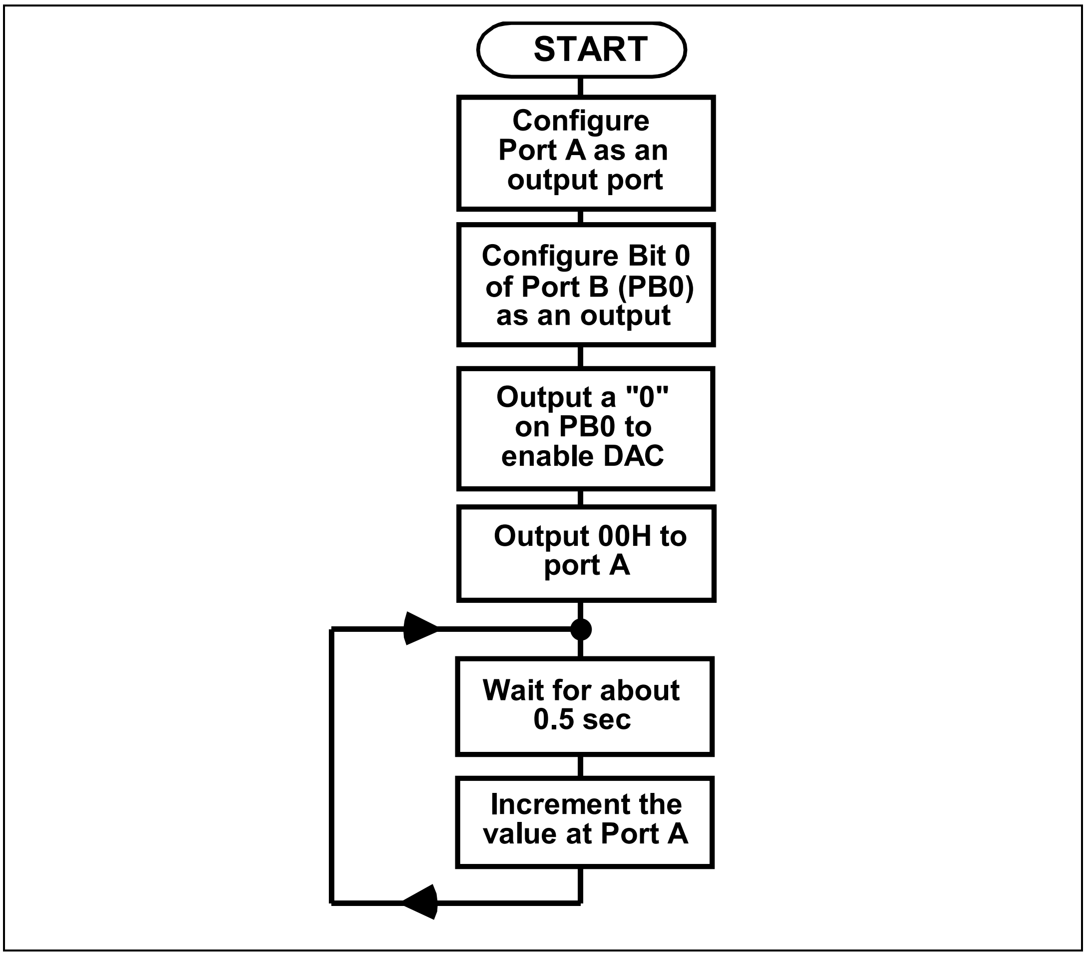
Note: *If PB0 is returned to logic "1" while the digital data is present at Port A, this data will become "latched" inside the DAC. The DAC output voltage will then remain held at a voltage proportional to the "latched" data, even if the data at Port A is subsequently changed.*

In order for the DAC output to respond to new data at Port A, PB0 must be taken to logic "0" again.

14.7 Worked Example

Write a program that will produce a slowly increasing binary count output (changing about every 0.5 seconds) at Port A which is passed, via the DAC to either the Optical Sender or the DC Motor.

Solution



The Assembly Language Program will be:

```
               PADR:          EQU    $9001
               PADDR:         EQU    $9003
               PBDR:          EQU    $9000
               PBDDR:         EQU    $9002

                              ORG    $0400        ;Defines the start address
0400    A9                    LDA    #$FF
0401    FF
0402    8D                    STA    PADDR        ;Configures Port A as an
0403    03                                        ;output port
0404    90
0405    A9                    LDA    #$01
0406    01
0407    8D                    STA    PBDDR        ;Configures PB0 as an
0408    02                                        ;output bit
0409    90
040A    A9                    LDA    #$00
040B    00
040C    8D                    STA    PBDR         ;Outputs a "0" on PB0 to
040D    00                                        ;enable DAC
040E    90
040F    A9                    LDA    #$00
0410    00
0411    8D                    STA    PADR         ;Sets Port A to 00H
0412    01                                        ;initially
0413    90
0414    A2     COUNTS:        LDX    #$FF
0415    FF
0416    A0                    LDY    #$FF         ;Sets count values for
0417    FF                                        ;0.5s delay
0418    CA     DELAY:         DEX
0419    EA                    NOP
041A    D0                    BNE    DELAY        ;Decrement X-register
041B    FD                                        ;until zero
041C    88                    DEY
041D    D0                    BNE    DELAY        ;Decrement Y-register
041E    FA                                        ;until zero to give a
                                                  ;delay of 0.5s
041F    EE                    INC    PADR         ;Increment the value
0420    01                                        ;output at Port A
0421    90
0422    4C                    JMP    COUNTS       ;Loop back to load delay
0423    14                                        ;count values again
0424    04
```

Ensure that the upper slider switch on the Applications Module is set to its upper position, so that the DC Motor Module is connected to the DAC. Set the motor LOAD control on the Applications Module to the fully counter-clockwise (minimum load) position.

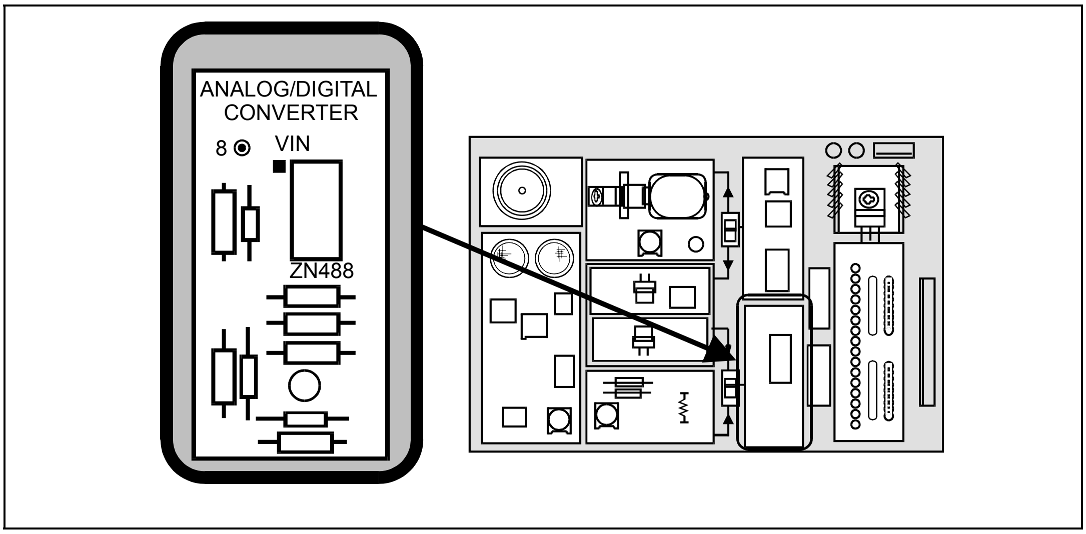
Load the above program into MAC III memory and execute.

Notice how a significant count is required before the motor begins to rotate. *Do not be tempted to rotate the disk yourself to start the motor, it will start by itself.*

This technique can be used, for example, to slowly run a DC Motor up to its operating speed.

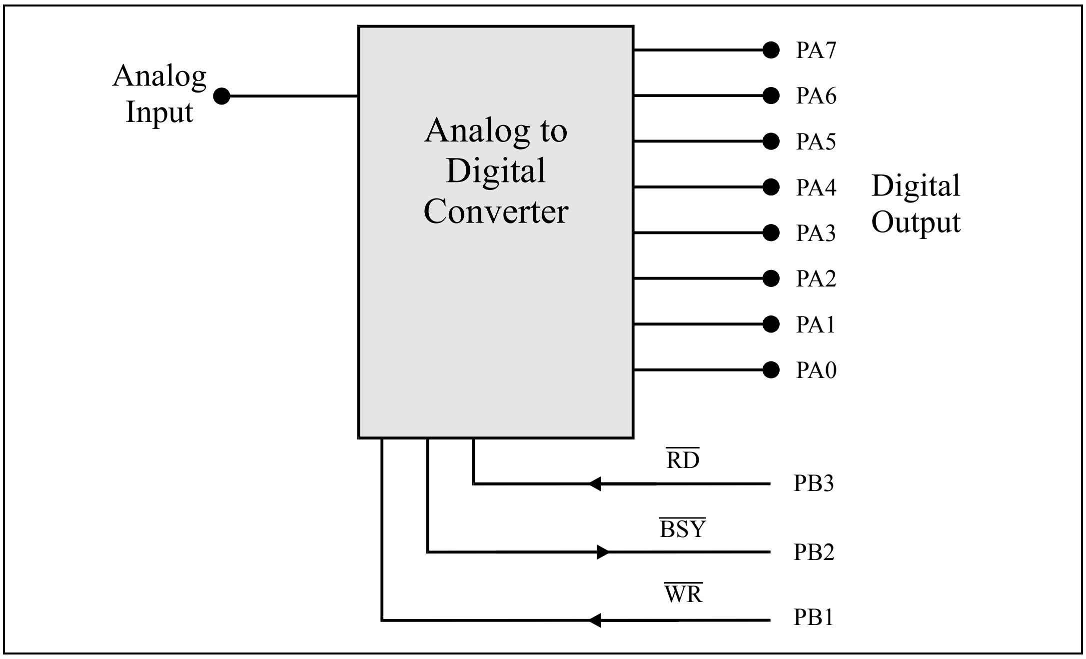
14.7a **Run the above program again and note the hexadecimal count at the monitor LED's when the motor just starts to rotate. Enter this hexadecimal byte.**

14.8 Analog to Digital Converter



An Analog to Digital Converter (ADC) is required where external analog inputs are to be applied to a digital system, for example a microcomputer. The ADC takes an input voltage and represents it as a binary value.

The Applications Module ADC has an 8-bit output connected to Port A and three control signals, connected to Port B thus:

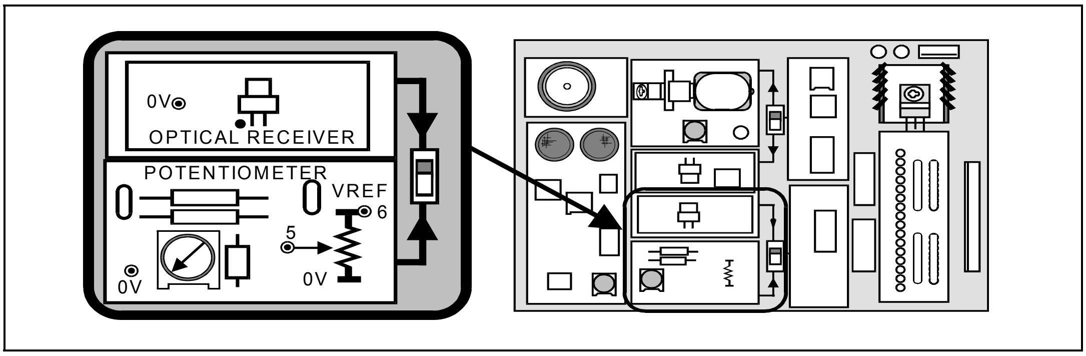


The input range is from 0V to 2.55V, like the DAC, so each step represents 10mV. The following sequence is required to perform conversion:

1. Output "1" on Port B, bits 1 and 3 (PB1 and PB3) initially.
2. Output a short negative-going pulse on Port B, bit 1 (PB1) to initiate conversion. This can be done by causing an output bit to change from 1 to 0 and back to 1 again.
3. Monitor Port B bit 2 (PB2) and wait for PB2=1 indicating that conversion is complete.
4. Output a "0" on Port B bit 3 (PB3) to enable ADC outputs.
5. Read the digital data on Port A.
6. Output a "1" on Port B bit 3 (PB3) to disable ADC outputs.

The value at Port A will now be directly proportional to the input voltage.

The ADC can be connected to either the Potentiometer or the Optical Receiver by means of the lower slider switch (ADC switch):

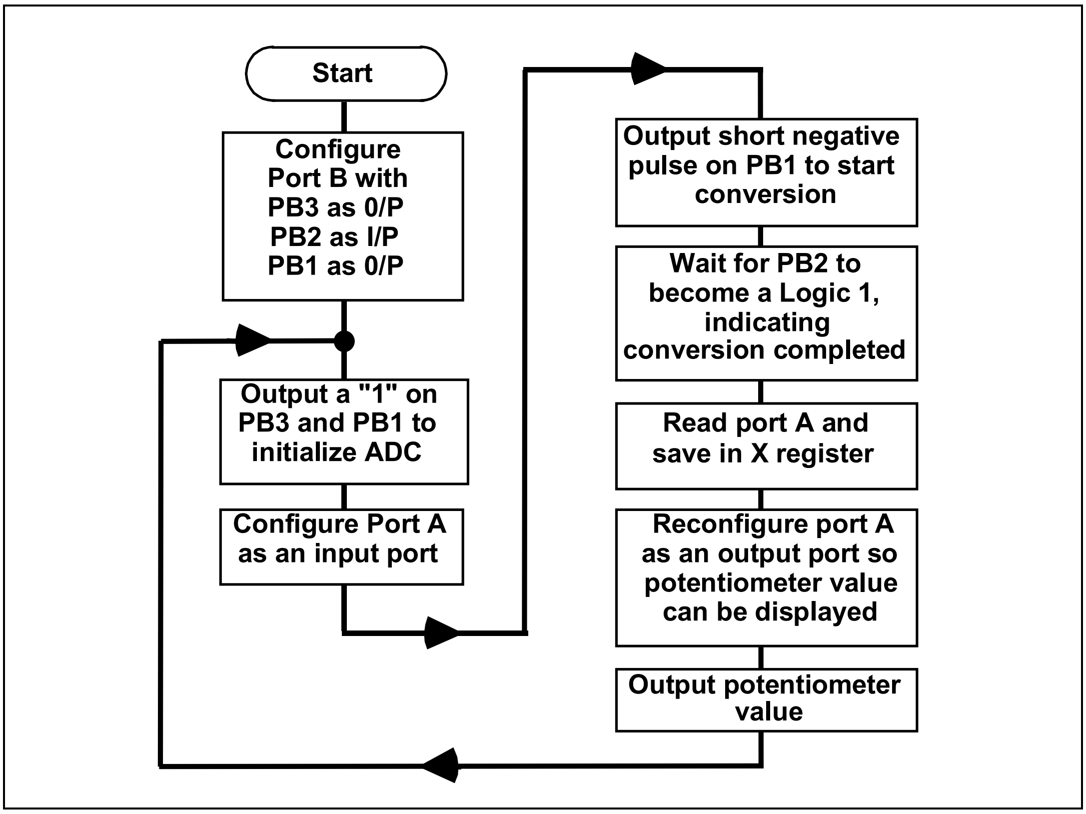


14.8a **If an input voltage of 1.5V is applied to the Applications Module Analog to Digital Converter (ADC), enter the output hexadecimal byte.**

14.9 Worked Example

Write a program which will output a binary value at Port A, dependent upon the setting of the Potentiometer.

Solution



The Assembly Language program will be:

```
            PADDR:  EQU   $9003
            PADR:   EQU   $9001
            PBDDR:  EQU   $9002
            PBDR:   EQU   $9000

                    ORG   $0400 ;Defines the start address
0400   A9   START:  LDA   #$0A
0401   0A
0402   8D           STA   PBDDR ;Configures Port B as
0403   02                       ;PB3=O/P, PB2=I/P AND PB1=O/P
0404   90
0405   A9   LOOP:   LDA   #$0A
0406   0A
0407   8D           STA   PBDR  ;Outputs a "1" on PB3 and PB1 to
0408   00                       ;initialize ADC
0409   90
040A   A9           LDA   #$00
040B   00
040C   8D           STA   PADDR ;Configures all of Port A as inputs
040D   03
040E   90
040F   A9           LDA   #$08
0410   08
0411   8D           STA   PBDR  ;Outputs a "0" on PB1
0412   00
0413   90
0414   A9           LDA   #$0A
0415   0A
0416   8D           STA   PBDR  ;Outputs a "1" on PB1 to generate a
0417   00                       ;short negative-going pulse on PB1
0418   90
0419   A9           LDA   #$04
041A   04
041B   2C   TSTB2:  BIT   PBDR  ;Tests PB2 for logic 1
041C   00
041D   90
041E   F0           BEQ   TSTB2 ;Repeat test of PB2 if not true
041F   FB
0420   A9           LDA   #$02
0421   02
0422   8D           STA   PBDR  ;PB2=1 so output a "0" on PB3
0423   00                       ;to enable ADC output
0424   90
0425   AD           LDA   PADR  ;Reads potentiometer input at Port A
0426   01
0427   90
```

Program continued:

```
0428  AA         TAX          ;Saves input value in the X-Register
0429  A9         LDA   #$FF
042A  FF
042B  8D         STA   PADDR ;Reconfigures Port A as all outputs
042C  03
042D  90
042E  8E         STX   PADR  ;Outputs potentiometer value at Port A
042F  01
0430  90
0431  A0         LDY   #$08  ;Loads X and Y registers with delay values
0432  08
0433  A2         LDX   #$FF
0434  FF
0435  CA  DELAY: DEX
0436  D0         BNE   DELAY
0437  FD
0438  88         DEY          ;Waits for about 10ms to allow output to
0439  D0         BNE   DELAY ;be displayed much more often than input
043A  FA
043B  4C         JMP   LOOP  ;Loop back
043C  05
043D  04
```

Load this program into MAC III memory and execute. Observe the effect of changing the Potentiometer setting.

This program can also act as an ambient light level indicator if the lower slider switch is moved to the upper position. The LED's now give an indication of the intensity of light falling on the Optical Receiver.

14.9a **Part of the program in Worked Example 14.9 generates a short negative going pulse on PB1. The purpose of this section of the program is to:**

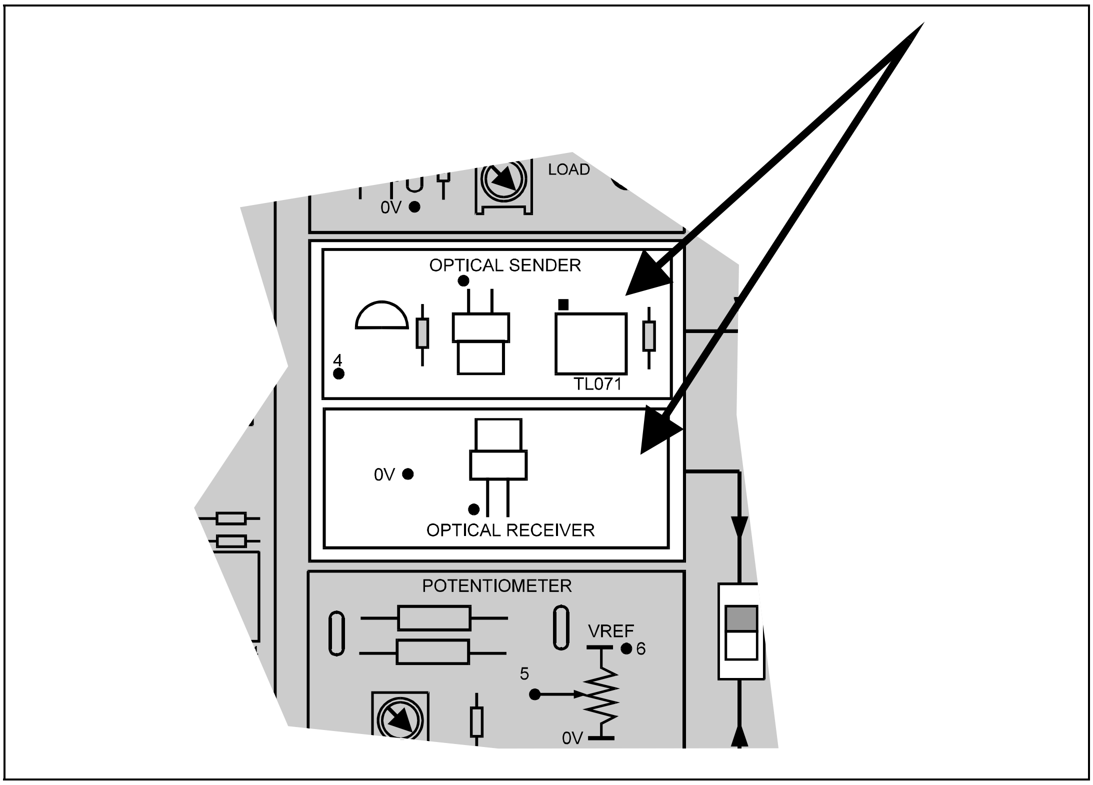
a initiate Analog to Digital Conversion

b enable the DAC

c disable the ADC outputs

d test the BSY signal line

14.10 Optical Sender and Optical Receiver



The Optical Sender and Optical Receiver units can be used in isolation, as LED and Light Detector respectively, or used together to form an Optical Link.

The output of the DAC can be switched to the Optical Sender by setting the upper slider switch to the lower position. The brightness of this LED then varies according to the code at the input to the DAC.

The Optical Receiver output can be switched to the input of the ADC by setting the lower slider switch to the upper position. The intensity of light falling on the Receiver can thus be converted into a binary value. The light intensity will depend upon the ambient lighting conditions and upon any light output from the Optical Sender unit.

14.11 Practical Assignment

Write a program which will sound the Piezo Sounder whenever the optical link between Optical Sender and Receiver is broken.

Note: *It can be assumed that if the optical link is unbroken, the ADC output will be greater than* 15_H*. When the link is broken, the ADC output will fall below* 15_H*.*

14.11a **In your solution to Practical Assignment 14.11, which bit position of Data Direction Register B was written with a logic "0"?**

a Bit 5

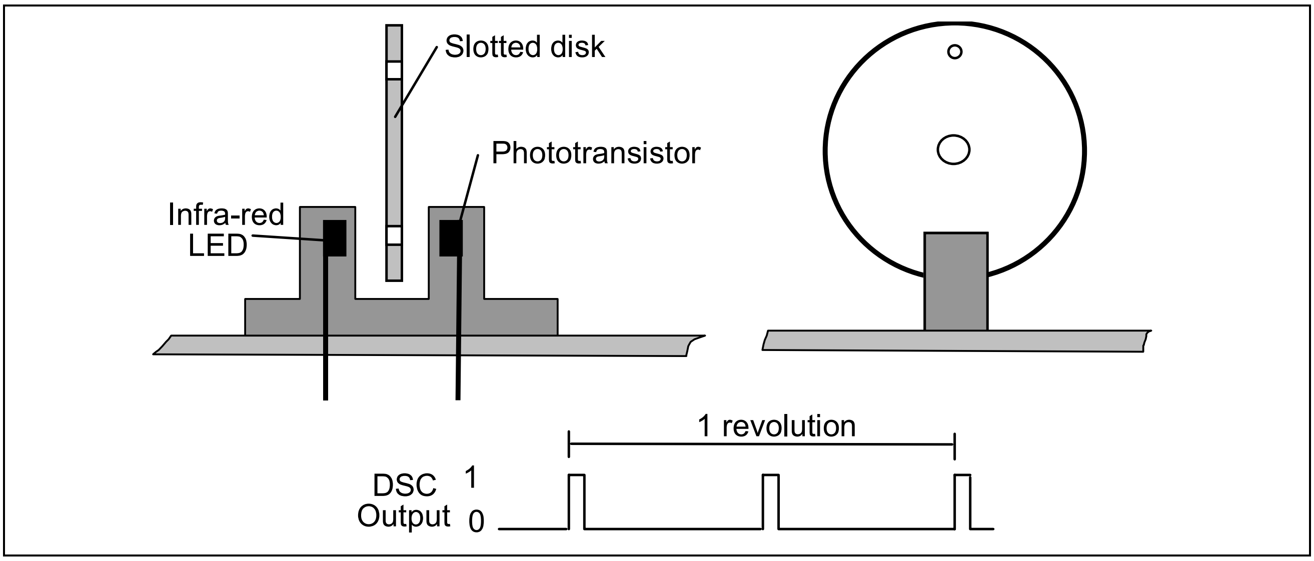
b Bit 3

c Bit 0

d Bit 2

14.12 Optical Disc Encoder

The motor disc passes between an optical transmitter and receiver. There are two holes in the disc, each one producing a short pulse as the shaft rotates. Clearly, the number of pulses per second is a measure of the speed of rotation of the motor shaft.



14.13 Practical Assignment

Write a program that will allow the speed of the DC Motor to be varied according to the setting of the Potentiometer.

14.13a **Run your program for Practical Assignment 14.13. Set the potentiometer to a point midway between the maximum and minimum settings. Enter the hexadecimal byte output at Port A.**

Student Assessment 14

1. **For the Piezo Sounder to produce an audio frequency, a TTL signal must be applied to:**

   a Port B, bit 5

   b Port B, bit 6

   c Port A, bit 5

   d Port A, bit 6

2. **The Ultrasonic Transmitter is switched on/off by the state of:**

   a Port B, bit 5

   b Port B, bit 6

   c Port A, bit 5

   d Port A, bit 6

3. **When the Ultrasonic Receiver detects a 40kHz ultrasound signal:**

   a PB6 is set to logic 1

   b PB6 is set to logic 0

   c PB7 is set to logic 1

   d PB7 has a 40kHz squarewave

4. **The section of the Applications Module that allows the microprocessor to produce an Analog output is the:**

   a ADC

   b DAC

   c Optical Disc Encoder

   d Potentiometer

Student Assessment 14 Continued ...

5. An increase of 01_H at the input of the Applications Module DAC produces a rise in output voltage of:

a 1mV

b 10mV

c 25.5mV

d 255mV

6. The section of the Applications Module that allows the microprocessor to read an Analog input is the:

a ADC

b DAC

c Optical Disc Encoder

d Piezo Sounder

7. The signal from the Applications Module ADC which indicates that conversion is complete is:

a $\overline{\text{RD}}$

b $\overline{\text{WR}}$

c $\overline{\text{BSY}}$

d $\overline{\text{EN}}$

8. The Applications Module units that could be used to form an ambient light measuring system are the:

a Optical Sender and the ADC

b Optical Sender and the DAC

c Optical Receiver and the ADC

d Optical Receiver and the DAC

Continued ...

Student Assessment 14 Continued ...

9. The number of pulses per revolution produced by the Applications Module Optical Disc Encoder is:

- a 0.5
- b 1
- c 2
- d 4

10. The effect of applying alternate logic '1' and logic '0' repeatedly at Port B, bit 5, with a delay of 0.1ms between each change, would be an output of:

- a 5 kHz at the Piezo Sounder
- b 10 kHz at the Piezo Sounder
- c 40 kHz at the Ultrasonic Transmitter
- d 80 kHz at the Ultrasonic Transmitter

11. The effect on the Applications Module of the program section:

```
LDA  #$40
STA  PBDDR
STA  PBDR
```

would be to:

- a Take the Piezo Sounder input high.
- b Take the Piezo Sounder input low.
- c Switch the Ultrasonic Transmitter on.
- d Switch the Ultrasonic Transmitter off.

Student Assessment 14 Continued ...

12. The program section required to enable the DAC is:

a
```
LDA  #$01
STA  PBDDR
LDA  #$00
STA  PBDR
```

b
```
LDA  #$01
STA  PBDDR
LDA  #$01
STA  PBDR
```

c
```
LDA  #$00
STA  PBDDR
LDA  #$00
STA  PBDR
```

d
```
LDA  #$00
STA  PBDDR
LDA  #$01
STA  PBDR
```

13. For the Applications Module ADC, conversion is initiated by applying an output of:

a logic '0' to Port B, bit 1

b logic '1' to Port B, bit 1

c a short negative-going pulse to Port B, bit 1

d a short positive-going pulse to Port B, bit 1

Chapter 15 Stack and Subroutines

Objectives of this Chapter

Having studied this chapter you will be able to:

- Explain the operation of a LIFO stack.
- Describe the stack save and restore instructions:
  Push
  Pull
- Explain the mechanism of subroutine calls.
- Describe the 6502 instructions:

  Jump to Subroutine
  Return from Subroutine
- Make use of MAC III Monitor Subroutines within your own programs which will:
  - write characters to the display.
  - read characters from the keyboard.
  - produce delays.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

Sometimes it is necessary to save data in a temporary store. This could be a partial result in a calculation, or because that register is required for another purpose. Clearly a memory location could be used to save data by direct addressing. However, the precise location of a temporary result is often relatively unimportant, provided the data can be retrieved reliably.

The stack is a special area of memory set aside for the storage of temporary data. It allows rapid storage and retrieval of data.

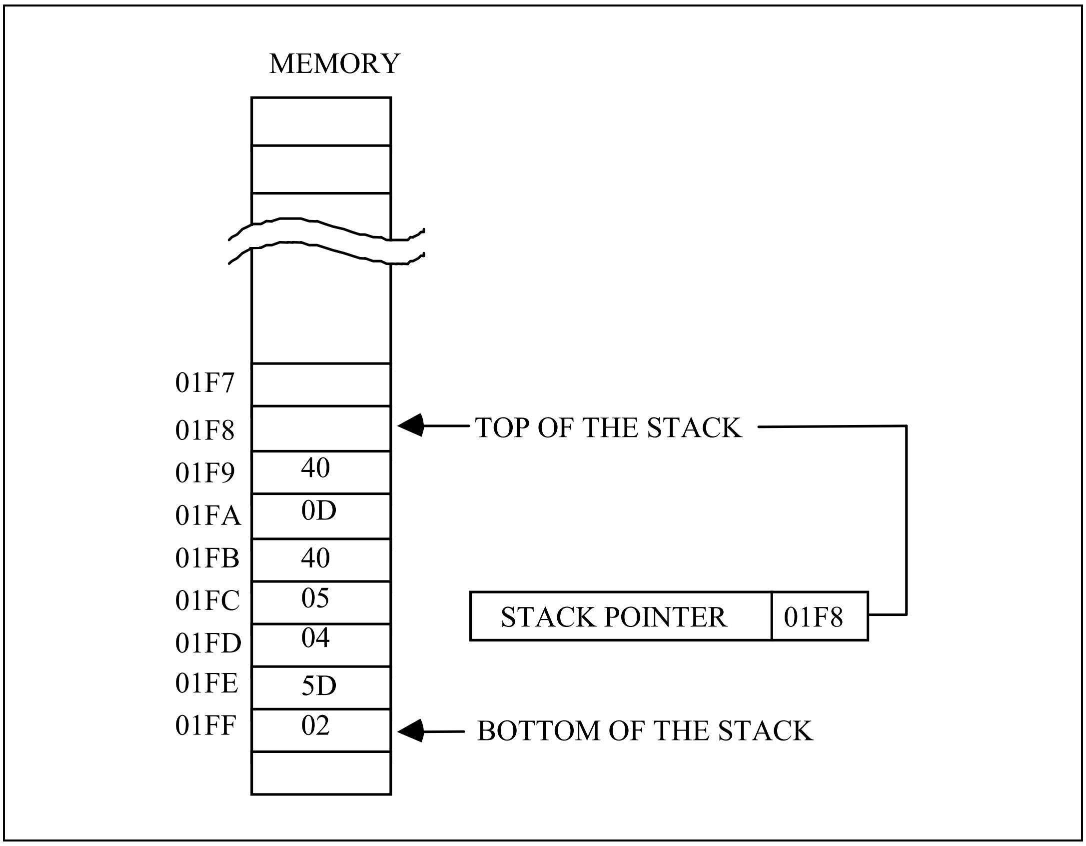
15.1 The Stack

The stack operates rather like a pile of documents in a tray. As sheets are placed in the tray, only the last document will be immediately accessible. The **last** document placed in the tray will be the **first** one removed.

In microcomputer stack terms this is called a “**Last In First Out**” or **LIFO** structure. In a LIFO stack the exact location of data is much less important than the **order** in which data words have been saved.

15.2 The Stack Pointer

A special register called the **Stack Pointer** is used to "point" to the next free location on the stack. The diagram below shows data saved on the stack and the Stack Pointer:



Clearly, the Stack Pointer will be **decremented** each time a new data word is stored. The 6502 reserves Page 01 of memory (i.e. locations 0100_H to $01FF_H$) for use as the stack.

15.3 Stack Save and Restore Instructions

Data is **saved** on the stack by using a **PUSH** instruction.

PUSH Accumulator (PHA)

This is used to save the accumulator contents on the stack. PHA performs the following actions:

1. Copies the accumulator contents into the stack location specified by the Stack Pointer.
2. Decrements the Stack Pointer, to point to the next free stack location.

For example:

Suppose the stack pointer contains 0180_H and the following is executed:

```
0400   A9   LDA  #$12      ;Loads the accumulator with 12H
0401   12
0402   48   PHA            ;Saves the accumulator on the stack
```

The PHA instruction will save the value 12_H at location 0180_H and then decrement the stack pointer to $017F_H$.

PUSH Status Register (PHP)

This instruction is very similar to PHA, except that it is the Status Register rather than the Accumulator which is saved on the stack.

This instruction allows the states of the flags at any point within a program to be saved and subsequently restored. You will see why this is important a little later in this chapter.

Data is **restored** from the stack by using a **PULL** instruction.

PULL Accumulator (PLA)

This instruction is used to restore the accumulator from the stack. PLA performs the following actions:

1. Increments the Stack Pointer, to point to the last byte saved on the stack.
2. Copies the contents of the stack location specified by the Stack Pointer into the accumulator.

For example:

Suppose the stack has the contents:

017C	9A
017D	78
017E	56
017F	34
0180	12

Stack Pointer = $017D_H$

If a PLA instruction is then executed:

Initially the Stack Pointer (SP) holds $017D_H$.
The SP is incremented to $017E_H$.
The contents of $017E_H$ are copied into the accumulator.

PULL Status Register (PLP)

Again, this instruction is very similar to PLA. The Status Register is restored from the stack.

Loading the Stack Pointer

The Stack Pointer register can only be loaded from the X-register. This requires the use of the Transfer X-Register to Stack Pointer instruction (TXS).

So, for example, to load the Stack Pointer with $019E_H$:

```
0433   A2   LDX  #$9E
0434   9E
0435   9A   TXS
```

Recall that the 6502 reserves page 01_H of memory for use as the stack. It is only necessary therefore to specify the **least significant byte** of the required stack pointer value ($9E_H$ in this case).

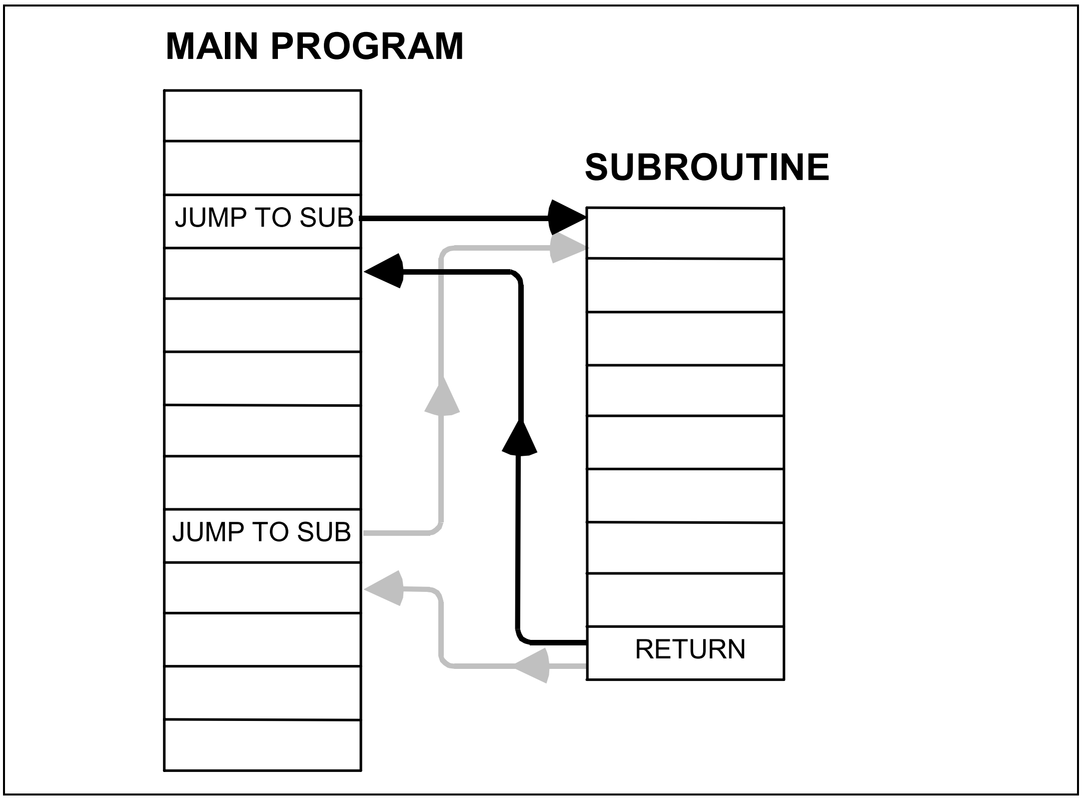
15.3a **The Stack Pointer is initially set to $01E0_H$. Enter the contents of the Stack Pointer after 5 bytes have been saved on the Stack.**

15.3b **The Stack Pointer is set to 0152_H. Enter the hexadecimal contents of the Stack Pointer after the instruction** "`PHA`" **has been executed.**

15.4 Subroutines

In many programs there will be sequences of instructions which are used several times within the program. For example the short time delay used in a number of the Applications Module programs in previous chapters. Such a repeated section of instructions is called a "**routine**". Now, rather than include the routine **every** time it is required, the microprocessor allows such sequences of object code to appear only **once** and then to be **called** upon several times within the program. A routine which can be used in this way is called a "**subroutine**".

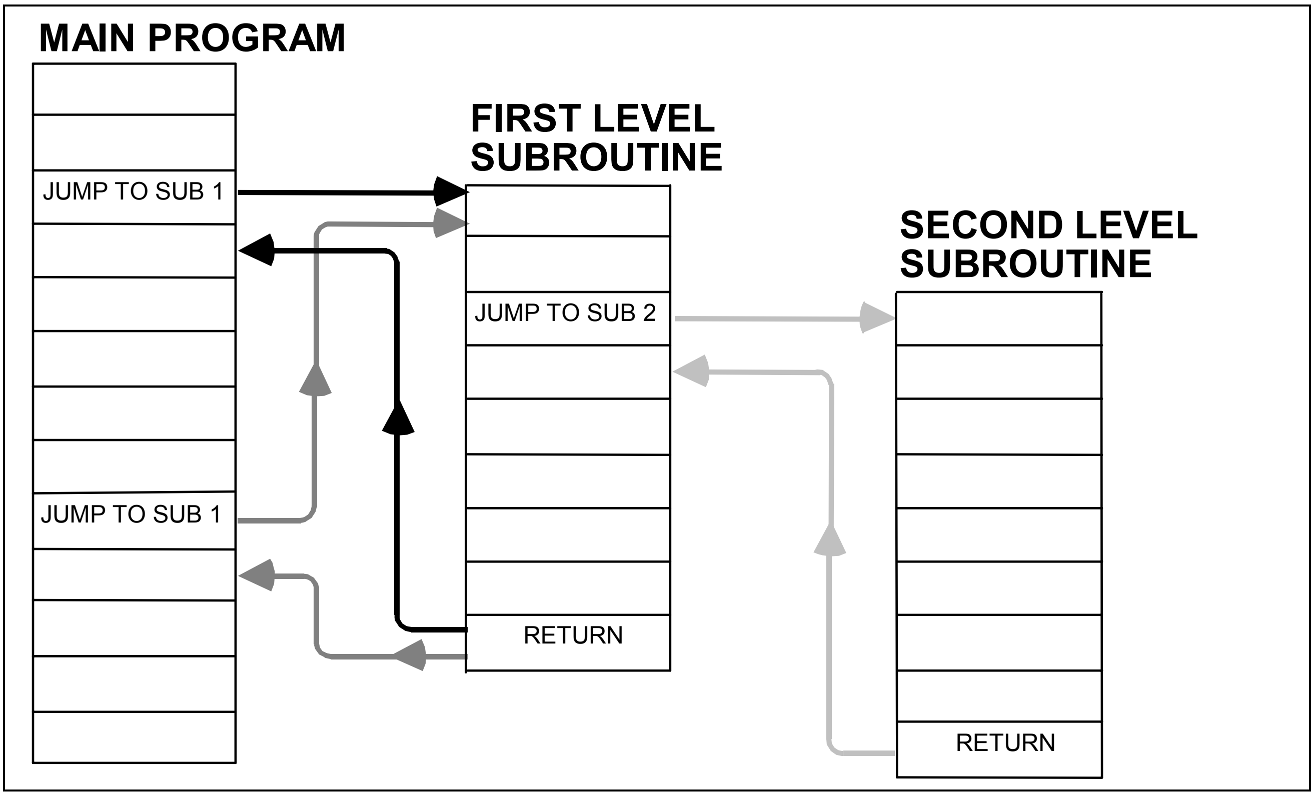
Subroutines are also often used by more than one program. Libraries of useful routines may be complied to reduce program development time. Programs which use subroutines are much easier to develop and understand.



A **Jump to Subroutine (JSR)** instruction transfers program execution to a subroutine.

A **Return from Subroutine (RTS)** instruction restores the program counter from the point at which it left the main program. You have been using RTS at the end of programs already. In the MAC III this allows you to return to the monitor program so that you can examine memory locations, etc.

When a subroutine is called, the return address is **automatically** saved on the **Stack**. At the end of a subroutine the return address is again **automatically** restored from the stack. This type of structure allows **multiple** levels of subroutines to be supported (sometimes called **nested** subroutines), where one subroutine calls another:



The first return address is saved on the stack and then the second. Since the stack has a LIFO action, each address will be restored as it is required (i.e. second return address **then** first).

Subroutines may use registers which the main program uses so it is good practice for a subroutine to save any registers which it uses.

15.4a **The function of a** "`JSR`" **instruction is to:**

a return to a main program from a subroutine.

b restore the Program Counter from the Stack.

c restore the General Purpose Registers from the Stack.

d transfer program execution to a subroutine.

We have seen how to save the Accumulator and the Status Register directly on the stack, using the PHA and PHP instructions respectively. Other registers must first be transferred to the Accumulator before PUSHing onto the stack.

So, to PUSH the X Register:

```
0421   8A   TXA        ;Copies X Register to the accumulator
0422   48   PHA        ;Copies accumulator to current top of the stack
```

Similarly, to PUSH the Y Register:

```
0434   98   TYA        ;Copies Y Register to the accumulator
0435   48   PHA        ;Copies accumulator to current top of the stack
```

The Stack Pointer itself can also be saved on the Stack by using the Transfer Stack Pointer to X Register (TSX) instruction:

```
045A   BA   TSX        ;Copies the Stack Pointer to the X Register
045B   8A   TXA        ;Copies X Register to the accumulator
045C   48   PHA        ;Copies accumulator to current top of the stack
```

It follows that these registers can all be restored from the stack. For example, to PULL the X Register:

```
0476   68   PLA        ;Copies contents of current top stack
                       ;location into the accumulator
0477   AA   TAX        ;Copies the accumulator into the X Register
```

Similarly, to PULL the Y Register:

```
0491   68   PLA        ;Copies contents of current top stack
                       ;location into the accumulator
0492   A8   TAY        ;Copies the accumulator into the Y Register
```

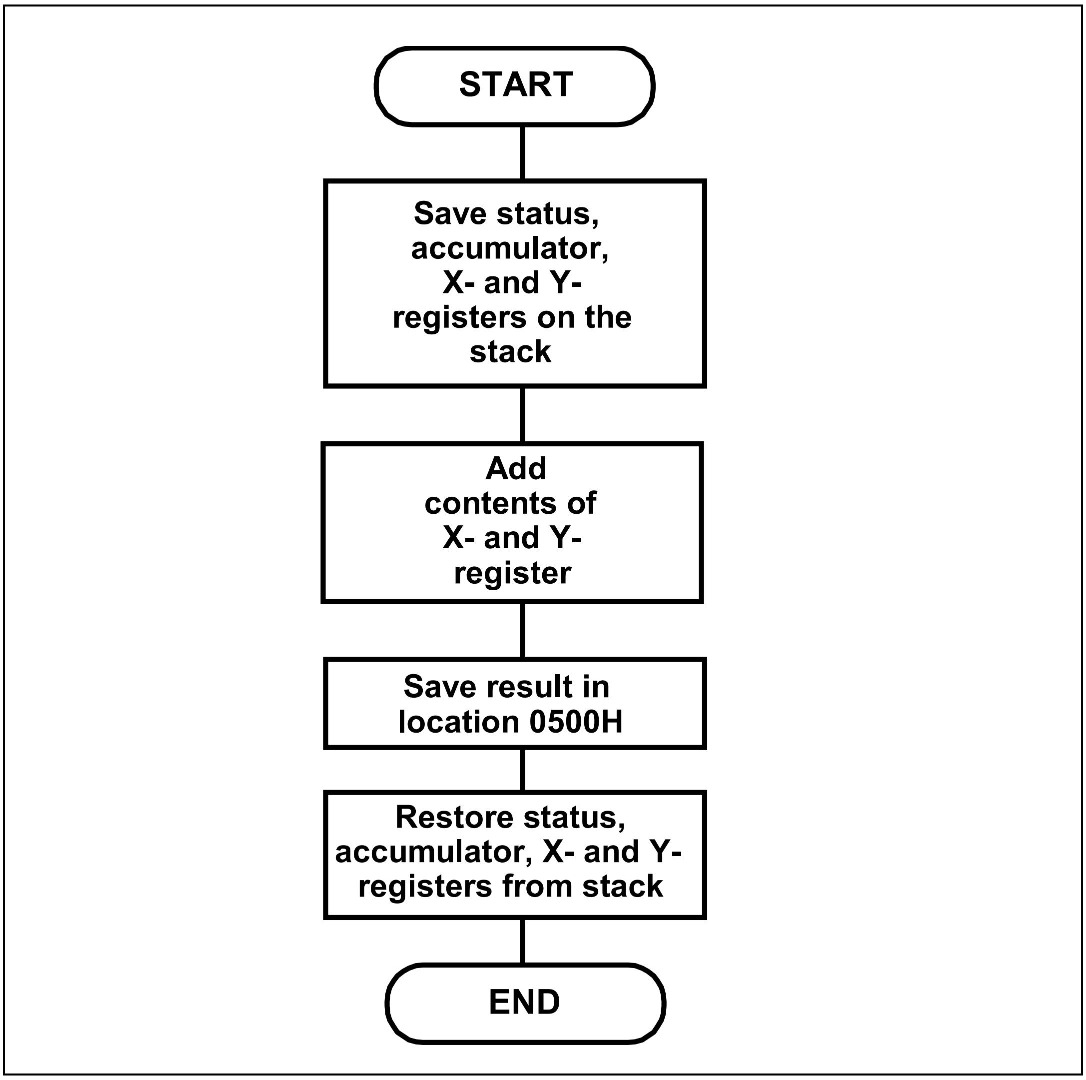
The stack pointer may also be restored from the stack, using the Transfer X Register to Stack Pointer instruction thus:

```
04B5   68   PLA          ;Copies contents of current top stack
                         ;location into the accumulator
04B6   AA   TAX          ;Copies the accumulator into the
                         ;X Register
04B7   9A   TXS          ;Copies the X Register into the Stack
                         ;Pointer
```

15.5 Worked Example

Write a subroutine which will add the contents of the X and Y registers, saving the result in location 0500_H. The previous contents of the accumulator, X Register and Y Register must be preserved.

Solution



The Assembly Language Program will be:

```
              ORG  $0400       ;Defines the start address
0400   08     PHP              ;Saves status register on stack
0401   48     PHA              ;Saves accumulator on stack
0402   8A     TXA
0403   48     PHA              ;Saves X-register on stack
0404   98     TYA
0405   48     PHA              ;Saves Y-register on stack
0406   86     STX  $40         ;Saves X-register in temporary
0407   40                      ;store, prior to addition
0408   D8     CLD
0409   18     CLC              ;Conditions D- and C-flags for addition
040A   65     ADC  $40         ;Accumulator already contains Y-register
040B   40                      ;contents, so add to temporary store
040C   8D     STA  $0500       ;Saves result in location 0500H
040D   00
040E   05
040F   68     PLA
0410   A8     TAY              ;Restores Y-register from stack
0411   68     PLA
0412   AA     TAX              ;Restores Y-register from stack
0413   68     PLA              ;Restores X-register from stack
0414   28     PLP              ;Restores accumulator from stack
0415   60     RTS              ;Returns from subroutine
```

Notice that:

1. The status register is saved and then restored at the end of the routine. This is because the CLC, CLD and ADC instructions may change the original status register contents.
2. Registers are PULLed in the opposite order in which they were PUSHed. This is due to the LIFO structure of the stack.

15.5a **The Stack Pointer register initially contains 0147_H. Enter the contents of the Stack Pointer after the program for Worked Example 15.5 has been executed.**

15.6 Practical Assignment

Write a subroutine that will use the stack to exchange the contents of the X Register and the Status Register. The Stack should be used to preserve the contents of any other registers used by the subroutine.

15.6a **The first two instructions in your program for Practical Assignment 15.6 are:**

a `PHA` and `PHP`

b `PHA` and `PLP`

c `PHA` and `TXA`

d `PHP` and `TXA`

15.7 Worked Example

Recall the program that you wrote to activate the Piezo Sounder:

```
              PBDR:        EQU    $9000
              PBDDR:       EQU    $9002

                           ORG    $0400   ;Defines the start address
0400   A9                  LDA    #$20
0401   20
0402   8D                  STA    PBDDR   ;Makes Port B bit 5 (PB5)
0403   02                                 ;an output bit
0404   90
0405   A9     HIOUT:       LDA    #$20
0406   20
0407   8D                  STA    PBDR    ;Outputs a "1" on PB5
0408   00
0409   90
040A   A2                  LDX    #$64    ;Loads count for delay
040B   64
040C   CA     DELAY1:      DEX
040D   D0                  BNE    DELAY1  ;Delay of 500 us
040E   FD
040F   A9                  LDA    #$00
0410   00
0411   8D     LOWOUT:      STA    PBDR    ;Outputs a "0" on PB5
0412   00
0413   90
0414   A2                  LDX    #$64    ;Loads count for delay
0415   64
0416   CA     DELAY2:      DEX
0417   D0                  BNE    DELAY2  ;Another delay of 500 us
0418   FD
0419   4C                  JMP    HIOUT   ;Loop back to output a
041A   05                                 ;"1" on PB5
041B   04
```

Notice that the same length of time delay has been used **twice**. Rewrite this program to make use of a single subroutine, which when called provides a 500µs time delay.

Solution

```
              PBDR:        EQU   $9000
              PBDDR:       EQU   $9002

                           ORG   $0400          ;Defines start address of
                                                ;main program
0400   A9                  LDA   #$20
0401   20
0402   8D                  STA   PBDDR          ;Makes Port B bit 5 (PB5)
0403   02                                       ;an output bit
0404   90
0405   A9     HIOUT:       LDA   #$20
0406   20
0407   8D                  STA   PBDR           ;Outputs a "1" on PB5
0408   00
0409   90
040A   20                  JSR   DELAY          ;Call delay of 500 us
040B   00
040C   05
040D   A9                  LDA   #$00
040E   00
040F   8D     LOWOUT:      STA   PBDR           ;Outputs a "0" on PB5
0410   00
0411   90
0412   20                  JSR   DELAY          ;Call delay of 500 us
0413   00
0414   05
0415   4C                  JMP   HIOUT          ;Loop back to output a
0416   05                                       ;"1" on PB5
0417   04
;Subroutine: 500µs delay
                           ORG   $0500          ;Defines start address of
                                                ;delay subroutine
0500   A2     DELAY:       LDX   #$64           ;Loads count for delay
0501   64
0502   CA     REDUCE:      DEX
0503   D0                  BNE   REDUCE         ;Delay of 500 us
0504   FD
0505   60                  RTS                  ;Return to main program
```

15.7a **Enter the number of times that the delay subroutine is called during each pass through the program of Worked Example 15.7.**

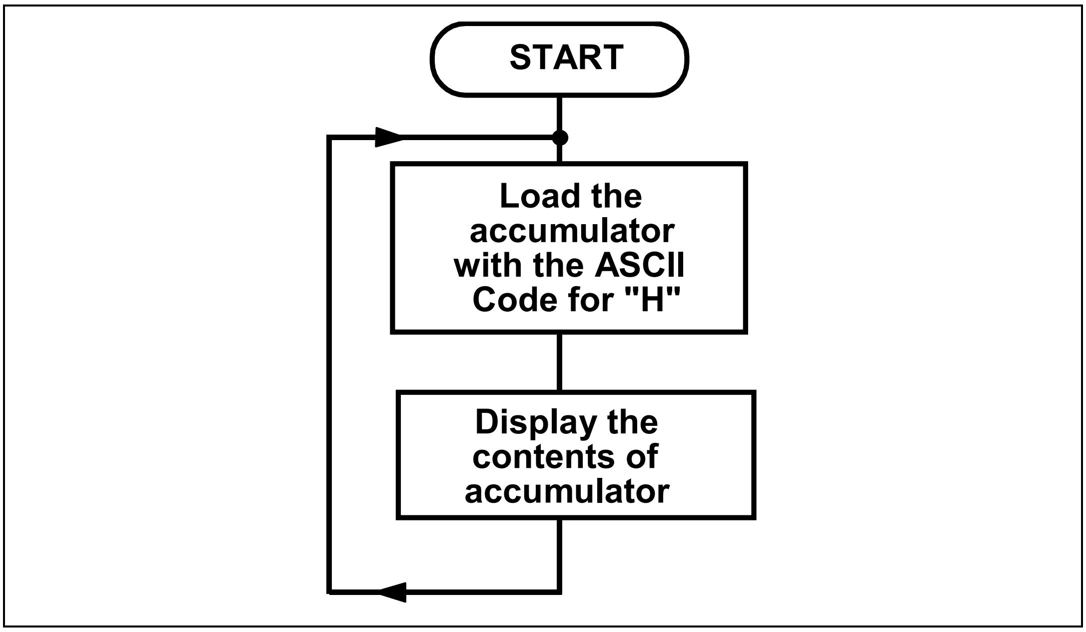
15.8 MAC III Monitor Subroutines

The MAC III Monitor program includes a number of subroutines. These are available for you to use in your own programs. A list and description of these subroutines can be found in Appendix 2.

One of these MAC III Monitor subroutines is "WRCHAR". This subroutine interprets the contents of the accumulator as an ASCII code and sends the corresponding character to the display. The following exercise will make use of this subroutine to write a given character to the display.

15.9 Worked Example

Write a program that will display the character "H".



Note: The ASCII code for "H" is 48_H. Other ASCII codes can be found in Appendix 3.

The Assembly Language Program will be:

```
            WRCHAR:      EQU   $C048

                         ORG   $0400      ;Defines the start address
0400   A9                LDA   #$48       ;Loads accumulator with
0401   48                                 ;the ASCII code for "H"
0402   20                JSR   WRCHAR     ;Call subroutine which
0403   48                                 ;displays the contents of
0404   C0                                 ;the accumulator as an
                                          ;ASCII character
0405   4C   HERE:        JMP   HERE       ;Wait forever - if RTS is
0406   05                                 ;used the MAC III monitor
0407   04                                 ;program will cause the
                                          ;display to be
                                          ;overwritten with "rEAdy"
```

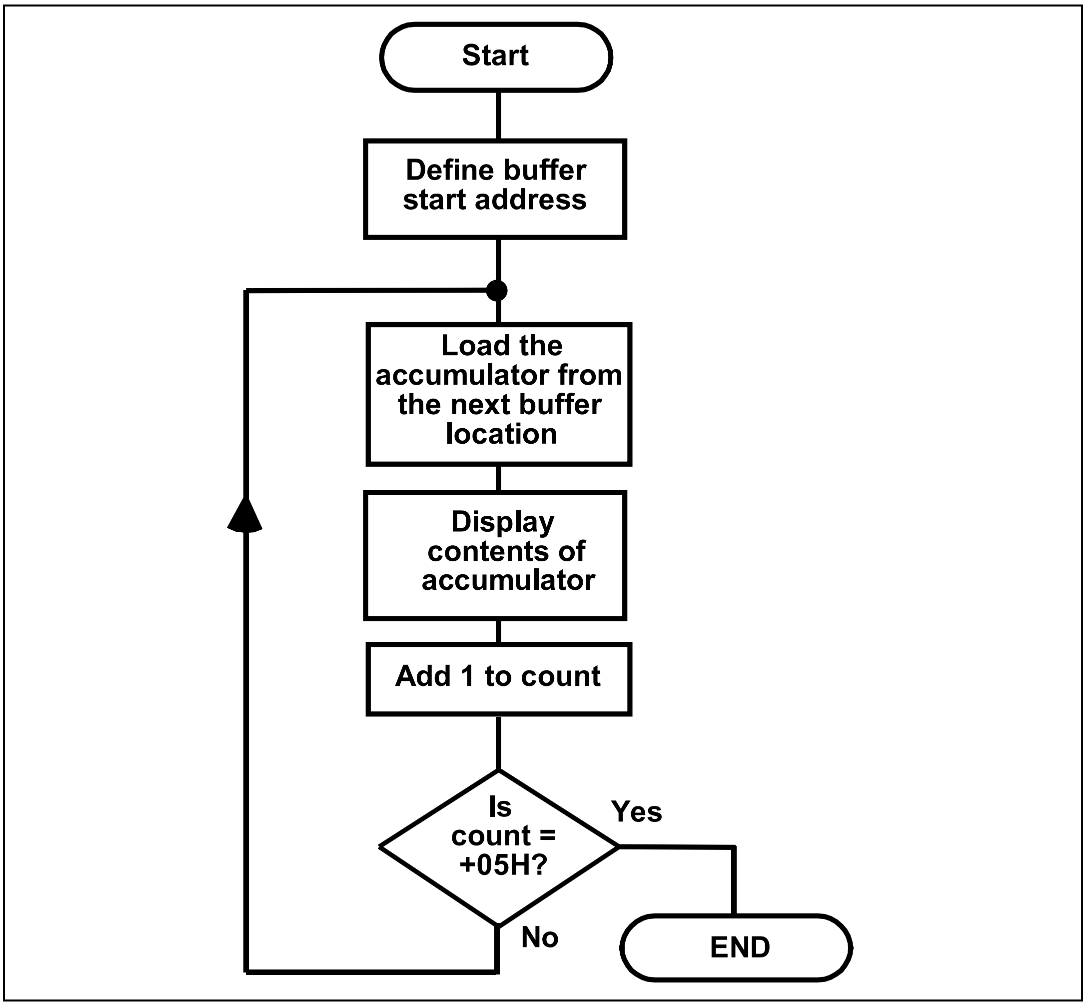
15.9a **The program for Worked Example 15.9 must be modified to display the character "*Z*". Enter the hexadecimal byte that the first instruction must write to the Accumulator.**

15.10 Worked Example

Write a program that will display the message "Hello" on the screen.

Clearly, this is an extension of the previous problem. One possible solution would be to effectively repeat the previous program a number of times but this would be a rather inelegant solution.

A more flexible approach is to set up a buffer in MAC III memory that contains the necessary codes and use a looped program to display each code in turn thus:



It is convenient to store the ASCII character codes in page zero memory, say from location 0040_H:

```
0040   48        ;Code for "H"
0041   45        ;Code for "E"
0042   4C        ;Code for "L"
0043   4C        ;Code for "L"
0044   4F        ;Code for "O"
```

The Assembly Language Program will be:

```
                WRCHAR:     EQU    $C048

                            ORG    $0400      ;Defines the start address
0400    A2                  LDX    #$00       ;Defines start of display
0401    00                                    ;buffer
0402    B5      NEXT:       LDA    $40,X      ;Read next value into the
0403    40                                    ;accumulator
0404    20                  JSR    WRCHAR     ;Call display subroutine
0405    48
0406    C0
0407    E8                  INX               ;Adds 1 to count
0408    E0                  CPX    #$05
0409    05
040A    D0                  BNE    NEXT       ;If count < 5, branch
040B    F6                                    ;back to read next value
040C    4C      HERE:       JMP    HERE       ;Wait forever to allow
040D    0C                                    ;steady display
040E    04
```

Execute the program and you should see the message "HELLO" on the display.

The program strategy outlined above could be used to display other words by changing the contents of locations 0040_H - 0044_H.

However, this would be limited to words with 5 or less letters. A more useful strategy is to continue to fetch codes for display from the buffer until the stop code 00_H is fetched.

An Assembly Language Program using this approach is shown below:

```
              WRCHAR:       EQU    $C048

                            ORG    $0400       ;Defines the start address
0400   A2                   LDX    #$00        ;Defines start of display
0401   00                                      ;buffer
0402   B5     NEXT:         LDA    $40,X       ;Read next value into the
0403   40                                      ;accumulator
0404   F0                   BEQ    HERE        ;If value = 0, finish
0405   07                                      ;display
0406   20                   JSR    WRCHAR      ;Call display subroutine
0407   48
0408   C0
0409   E8                   INX                ;Adds 1 to count
040A   4C                   JMP    NEXT        ;Loop back to display
040B   02                                      ;next character
040C   04
040D   4C     HERE:         JMP    HERE        ;Wait forever to allow
040E   0D                                      ;steady display
040F   04
```

Enter this program into MAC III memory and ensure that you have entered the ASCII codes for "HELLO", terminated by the stop code 00_H.

Execute the program and you should see the message "HELLO" on the display.

You can now easily experiment with other words by changing the contents of the buffer from location 0040_H.

The ASCII codes required are given in Appendix 3.

Make sure that you **end** your message with 00_H. If you are running this program via the MAC III keypad, check that your message does not exceed 8 characters, since this is the limit of the MAC III display.

The MAC III monitor uses exactly this technique to display a number of words (for example, "rEAdy", "APPLICAtIONS", "SELECt", etc.).

15.10a In the program above, the instruction that tests the next character to see if the end of the buffer has been reached is:

a `BEQ HERE`

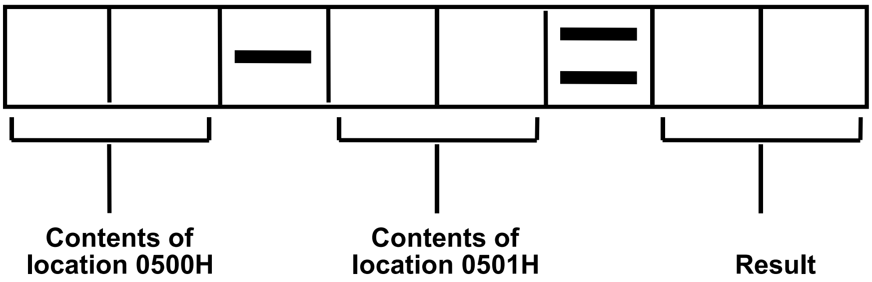
b `INX`

c `JMP HERE`

d `JSR WRCHAR`

15.11 Practical Assignment

Write a program that uses MAC III monitor subroutines to subtract the hexadecimal contents of location 0501_H from the contents of location 0500_H and display the following on the seven-segment displays:



15.11a **Load location 0500_H with 87_H and location 0501_H with 39_H. Run your program for Practical Assignment 15.11 and enter the byte shown as the result.**

15.12 Practical Assignment

Write a program, using MAC III monitor subroutines, that will produce an increasing binary count at Port A. The count should be incremented once per second.

15.12a **In your solution to Practical Assignment 15.12, the MAC III System subroutine that could be used to give a 1 second delay is called:**

a `CRLF`

b `GETIN`

c `WRCHAR`

d `WTNMS`

15.12b **Run your program for Practical Assignment 15.12. Wait for 25 seconds and read the binary count value that is displayed at Port A. Enter this count value as a hexadecimal number.**

15.13 Practical Assignment

Write a program that will sound the piezo sounder whenever the "S" key is held down on the MAC III keypad.

This program should be run from the MAC III keypad/display. If you are using the 6502 cross assembler Terminal software you should use the 'P' command to transfer control to the keypad/display, before running this program from the MAC III keypad.

15.13a In your solution to Practical Assignment 15.13, the instruction used to check if the S (and no other) key has been pressed is a:

- a Compare
- b Decrement
- c Increment
- d Store

15.14 Practical Assignment

Write a program, using MAC III monitor subroutines, that will allow the speed of the DC Motor to be controlled by the "+" and "-" keys. The motor should slowly accelerate when the "+" key is pressed, hold speed constant if no keys are pressed and decelerate when the "-" key is pressed.

15.14a Run your program for Practical Assignment 15.14. The effect of pressing the S key is that:

- a motor speed increases
- b motor speed decreases
- c motor stops
- d motor speed is unchanged

15.14b In your program for Practical Assignment 15.14, Port B is configured by writing a hexadecimal byte to Data Direction Register B. Enter the bit number of this register which must be at logic "1".

Student Assessment 15

1. **In a LIFO stack, the last data word stored will be restored:**

   a last

   b first

   c from the Stack Pointer Register

   d from the Status Register

2. **The last stack location used is defined by the contents of the:**

   a Data Register

   b Stack Pointer Register

   c X Register

   d Y Register

3. **The Stack Pointer contains 015D$_H$. After the instruction "`PLA`" has been executed, the Stack Pointer will contain:**

   a 015C$_H$

   b 015D$_H$

   c 015E$_H$

   d 015F$_H$

4. **The 6502 instruction that saves data on the stack is called:**

   a POP

   b PUSH

   c PULL

   d RTS

Continued ...

Student Assessment 15 Continued ...

5. A sequence of object code that appears once but which may be used several times is called a:

a Library

b Section

c Stack

d Subroutine

6. When a subroutine is called, the return address is saved:

a on the Stack

b in the Stack Pointer Register

c in the X Register

d in the Y Register

7. The 6502 instruction that transfers program execution to a subroutine is:

a CALL

b GOSUB

c JMP

d JSR

8. The 6502 instruction that usually occurs at the end of a subroutine is:

a RST

b RTS

c RET

d RETN

Student Assessment 15 Continued ...

9. The 6502 instruction sequence that will save the X Register on the Stack is:

a
```
PHA
TAX
```

b
```
PHA
TXA
```

c
```
TAX
PHA
```

d
```
TXA
PHA
```

10. The MAC III monitor subroutine that allows ASCII characters to be written to the display is:

a RDCHAR

b WRCHAR

c CLRSCR

d WTNMS

11. If a key is pressed, the MAC III monitor subroutine "GETIN" will place the corresponding value in the:

a Accumulator

b Status Register

c X Register

d Y Register

Chapter 16 Interrupts

Objectives of this Chapter

Having studied this chapter you should be able to:

- Describe the principles of interrupt and polled Input/Output.
- Explain the mechanisms of interrupts.
- Describe the 6502 Indirect Addressing mode.
- Use the 6502 interrupt system.
- Describe the 6502 Reset and Software Interrupt instruction.
- Use the auto-run feature of the MAC III system.

Equipment Required for this Chapter

- MAC III 6502 Microcomputer.
- Applications Module.
- Power supply.
- Keypad/display unit.
- Two shorting leads (supplied).
- Merlin Development System Software Pack, installed on a PC running Windows 95 or later.
- MAC III 6502 User Manual.

Introduction

An interrupt is a special **input** to a microprocessor that is examined as part of **every** instruction that the microprocessor executes. When an active transition occurs on this input, the current program is **suspended**.

The microprocessor will then start to execute an **interrupt service subroutine**. At the end of this routine the original program is usually resumed, from the point at which it was suspended.

The use of interrupts allows the microcomputer to respond quickly to external events.

16.1 Polling and Interrupts

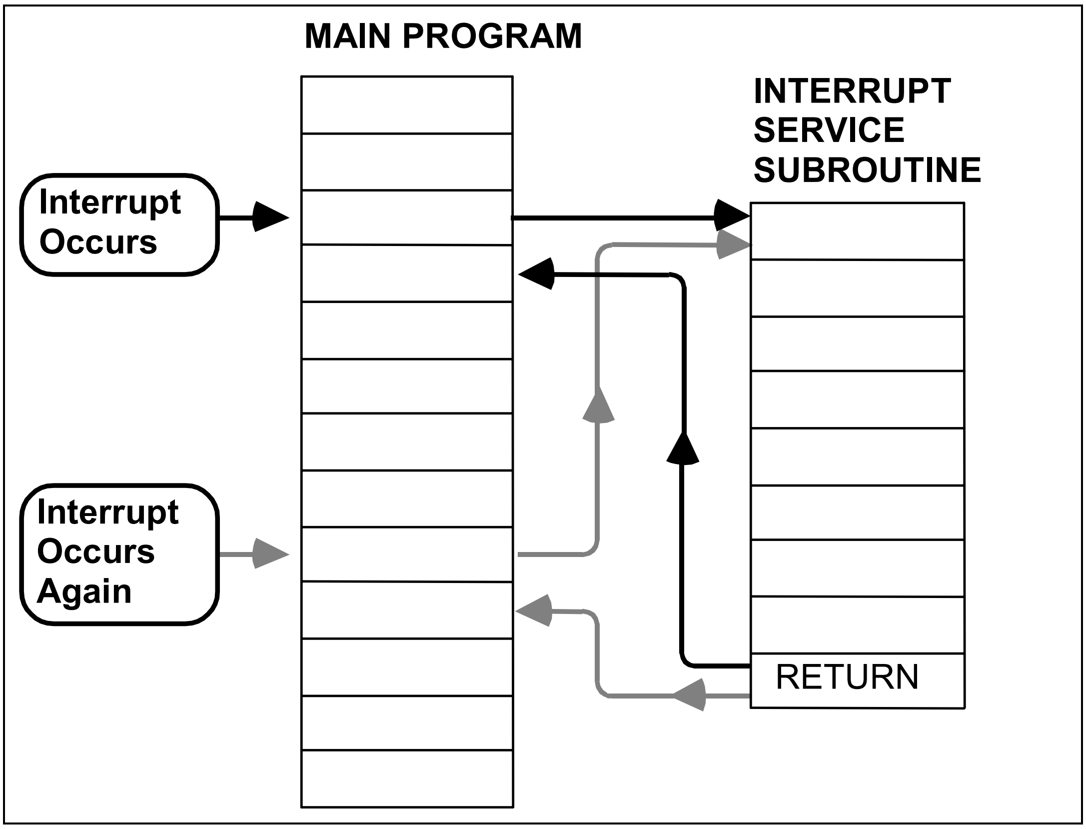
Many peripheral devices operate at a very much slower speed than the microprocessor. Consequently it will often be necessary for the microprocessor to wait while the peripheral responds. There are two basic techniques to achieve this synchronization:

Polled Input/Output

The microprocessor periodically checks the peripheral to see if it is ready for data transfer. This gives variable response times and also wastes microprocessor time in needless checking.

Interrupt Input/Output

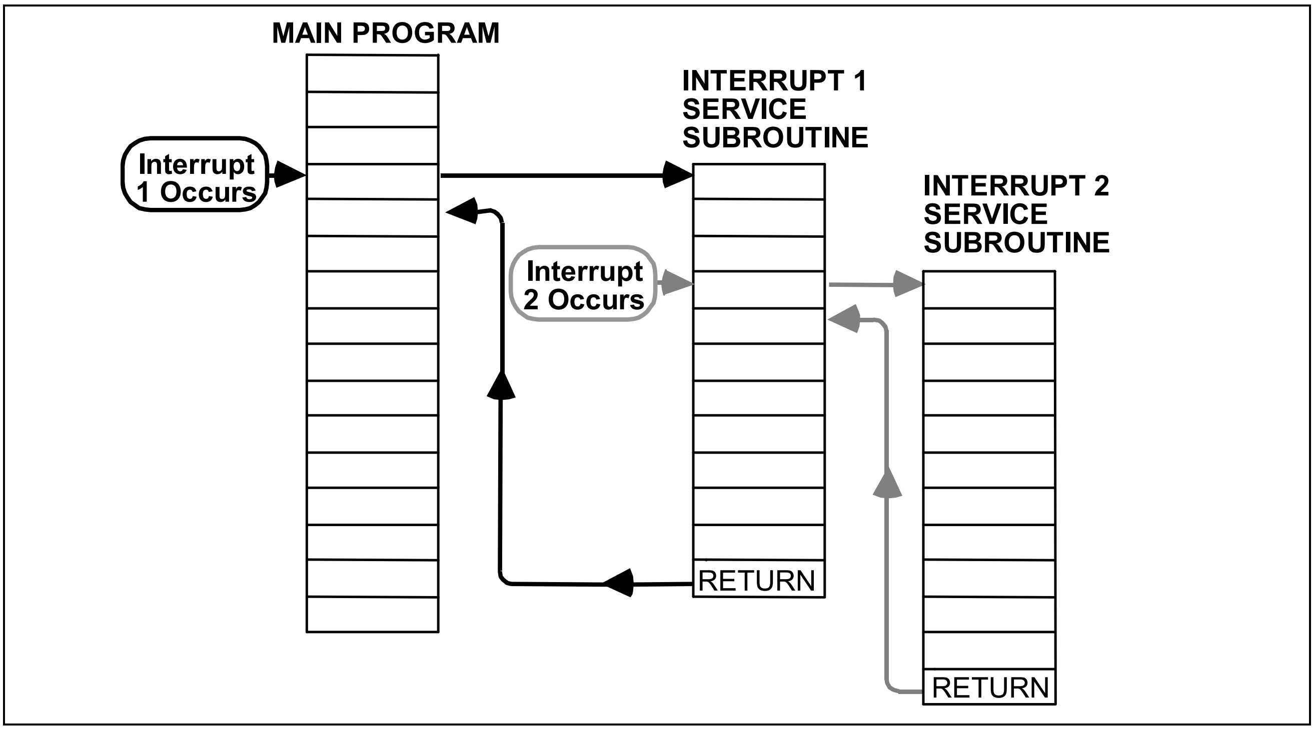
The peripheral signals it is ready for data transfer by **interrupting** the microprocessor. This has the advantage that the microprocessor does not waste time interrogating the peripheral over and over again. The microprocessor can actually be executing **another** program which is suspended when data transfer is required (sometimes called a background program).



Interrupts allow **external** events to cause a specific subroutine to be executed. So, an interrupt service subroutine can occur at **any** time during the execution of a program, unlike a normal subroutine that may only occur at a fixed position within a program sequence.

Since an interrupt may occur at **any** time within a program, the return address must be saved on the stack.

It is possible for microcomputer systems to have **multiple** interrupts, so **nesting** of interrupts may occur, where a second interrupt occurs while an interrupt service subroutine is already in progress:



The first return address is saved on the stack and then the second. Since the stack has a LIFO action, each address will be restored as it is required (i.e. second return address **then** first).

An interrupt service subroutine may use registers which the main program uses so it is good practice for interrupt service subroutines to save any registers they use.

The microprocessor will always complete the instruction in progress when an interrupt occurs before beginning the interrupt response sequence. When a particular interrupt occurs, the corresponding **interrupt vector** is loaded into the Program Counter. This vector defines the start of the appropriate interrupt service routine.

16.1a Usually, when an interrupt service routine has been completed:

a the microprocessor must be reset

b an interrupt will occur

c the interrupted program is resumed

d a Halt occurs

16.2 Interrupt Mechanisms

Interrupt Masking

Certain interrupts can be "disabled" so that the microprocessor will **not** respond when they occur. Interrupts that can be ignored in this way are called **maskable interrupts**.

Maskable interrupts can usually be enabled or disabled by setting or clearing an interrupt flag. In the 6502 this flag is called the Interrupt Disable Flag (I-Flag). The I-Flag is bit 2 of the Status Register.

Other interrupts are **non-maskable** and must **always** be serviced. Interrupts of this type are usually reserved for the most important tasks (for example, power failure routines).

Software Interrupts

Almost all microprocessors allow a special **instruction** to initiate an interrupt response (rather than an external **signal**). These are called **Software Interrupts**. It follows that software interrupts will be **synchronous** with the execution of the interrupted program.

The 6502 software interrupt instruction is called "Break" (BRK).

Reset

You will have already used this interrupt many times throughout this manual. When you press and release the "Reset" key on the MAC III board, the CPU will load the start address of the MAC III monitor program into the Program Counter and resume the fetch and execute process.

Return from Interrupt

Clearly, it will be necessary to terminate an interrupt service subroutine with a RETURN instruction, to restore the program counter from the stack. This allows the interrupted program to continue from the point at which it was interrupted.

The 6502 Return from Interrupt instruction has the mnemonic RTI.

Most interrupt service routines will terminate with an RTI instruction. One notable exception is the reset routine. Reset routines will **not** normally end with a RTI since a reset usually only occurs at initial start up or if there has been some catastrophic software failure.

Interrupt Priority

Some interrupts are considered to be more important than others. Consequently, a higher priority interrupt will interrupt a lower priority interrupt service subroutine but not vice versa. In such a case, the lower priority interrupt service subroutine will be resumed at the end of the higher priority routine.

Interrupt Response Time

Interrupts are used whenever it is necessary for the CPU to respond quickly to an event. For example: a machine tool micro-controller must respond quickly to an emergency stop condition.

However high the priority, **no** interrupt is serviced until the **current** instruction has been completed. This leads to very small variations in response times.

Interrupt Vectors

For each of the 6502 interrupt inputs there are **two** memory locations which hold the start address of the interrupt service routine for that interrupt. These are called **interrupt vectors**. Every microprocessor of the same model will have the **same** interrupt vectors. In the 6502 these are to be found at the **top** of memory ($FFFA_H$ - $FFFF_H$). When a valid interrupt occurs, the values contained within these locations are loaded into the Program Counter and the fetch/execute process is resumed.

Before we can progress further, you will need to understand a new addressing mode which is used in Interrupt processing. This is called **Absolute Indirect Addressing**:

16.3 Absolute Indirect Addressing

This mode of addressing is frequently used to **redirect** interrupt vectors. We shall meet this idea again, a little later in this chapter. The Absolute Indirect addressing mode uses the contents of **two** consecutive memory locations to form the address of the data to be acted upon. Only the JMP instruction can use this mode.

For example:

```
0423   6C   JMP ($0532)        ;Jumps to the address
0424   32                      ;specified by the contents of
0425   05                      ;locations 0532H and 0533H.
```

Location 0532_H holds the **low byte** of the final destination address and location 0533_H the **high byte**. Suppose the contents of these locations were:

Location	**Contents**
0532_H	$4B_H$ <------- Low byte of final destination address
0533_H	10_H <------- High byte of final destination address

Then " `JMP ($0532)` " will actually make a Jump to location **$104B_H$**. Interrupt vectors act in just the same way, defining the start address for an interrupt routine. Indirect addressing can also be used to **redirect** an interrupt, if the first instruction of an interrupt service routine is an indirect Jump.

16.3a **If location 0789_H contains 50_H and location $078A_H$ contains 00_H, the instruction** "`JMP ($0789)`" **will cause program execution to continue from location:**

a 0050_H

b 0789_H

c $078A_H$

d 5000_H

16.4 6502 Interrupt Flags

There are two 6502 flags associated with interrupts:

Interrupt Disable Flag (I-Flag)

This flag is used to enable or disable maskable interrupts. If the I-flag is **set** (i.e. if I=1) then maskable interrupts will **not** be acknowledged.

There are two 6502 instructions that can be used to enable or disable maskable interrupts:

CLI	Clears Interrupt Disable Flag to **allow** maskable interrupts to be acknowledged.
SEI	Sets Interrupt Disable Flag to **prevent** maskable interrupts from being acknowledged.

These instructions allow the user to define periods when maskable interrupts are to be acknowledged.

Break Flag (B-Flag)

This flag is set when a Software Interrupt instruction (BRK) occurs. We shall examine this instruction a little later in this chapter.

16.5 6502 Interrupt System

The 6502 has single maskable, non-maskable and software interrupts. There is also a reset input. All 6502 interrupt vectors are located at the **top** of memory (FFFA_H to FFFF_H).

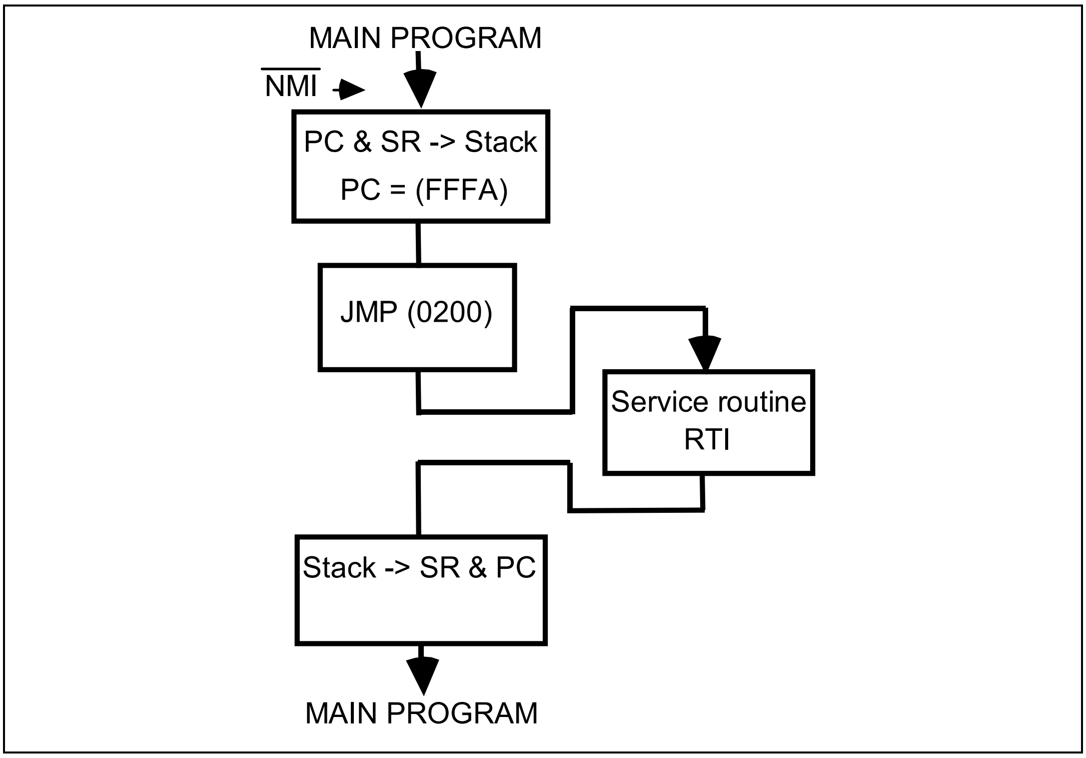
Non-Maskable Interrupt $\overline{\text{NMI}}$

This is an **active low**, **edge triggered** input. This means that it is activated by a transition from a logic "1" to a logic "0" on the $\overline{\text{NMI}}$ pin of the 6502.

The response to a $\overline{\text{NMI}}$ is listed below:

1. The **high byte** of the **program counter** is pushed onto the stack.
2. The **low byte** of the **program counter** is pushed onto the stack.
3. The **status register** is pushed onto the stack.
4. The **interrupt mask flag** is **set**, to prevent further interrupts from being serviced.
5. The contents of location FFFA_H are fetched and placed in the **low byte** of the program counter.
6. The contents of location FFFB_H are fetched and placed in the **high byte** of the program counter.
7. Program execution continues from the location pointed to by the program counter.

Note: The 6502 will **automatically** save the contents of the status register on the stack when a non-maskable interrupt occurs (step 3).



In the case of the MAC III, the NMI interrupt vectors have been **redirected** thus:

User NMI Vector: Location 0200_H: Low byte of interrupt vector
Location 0201_H: High byte of interrupt vector

These should be used, rather than $FFFA_H$ and $FFFB_H$, for user programs.

16.5a **In the 6502, maskable interrupts are prevented from interrupting the processor by:**

a applying a logic '0' to the IRQ pin

b applying a logic '1' to the IRQ pin

c clearing the I-flag

d setting the I-flag

16.6 Worked Example

Write a program that will load location 0040_H with the value 80_H. An $\overline{\text{NMI}}$ interrupt routine is also required which will reload location 0040_H with 01_H.

The main program and interrupt service routine will be trivial:

Main Program:

```
                                ORG   $0400       ;Main program start address
0400   A9                       LDA   #$80
0401   80
0402   85                       STA   $40         ;Saves marker to location
0403   40                                         ;0040H
0404   4C   HERE:               JMP   HERE        ;Wait forever - dummy
0405   04                                         ;program
0406   04
```

The NMI routine can be placed anywhere in memory:

```
                                ORG   $0500       ;NMI routine start address
0500   A9                       LDA   #$01
0501   01
0502   85                       STA   $40         ;Changes marker in
0503   40                                         ;location 0040H
0504   40                       RTI               ;Return to dummy program
```

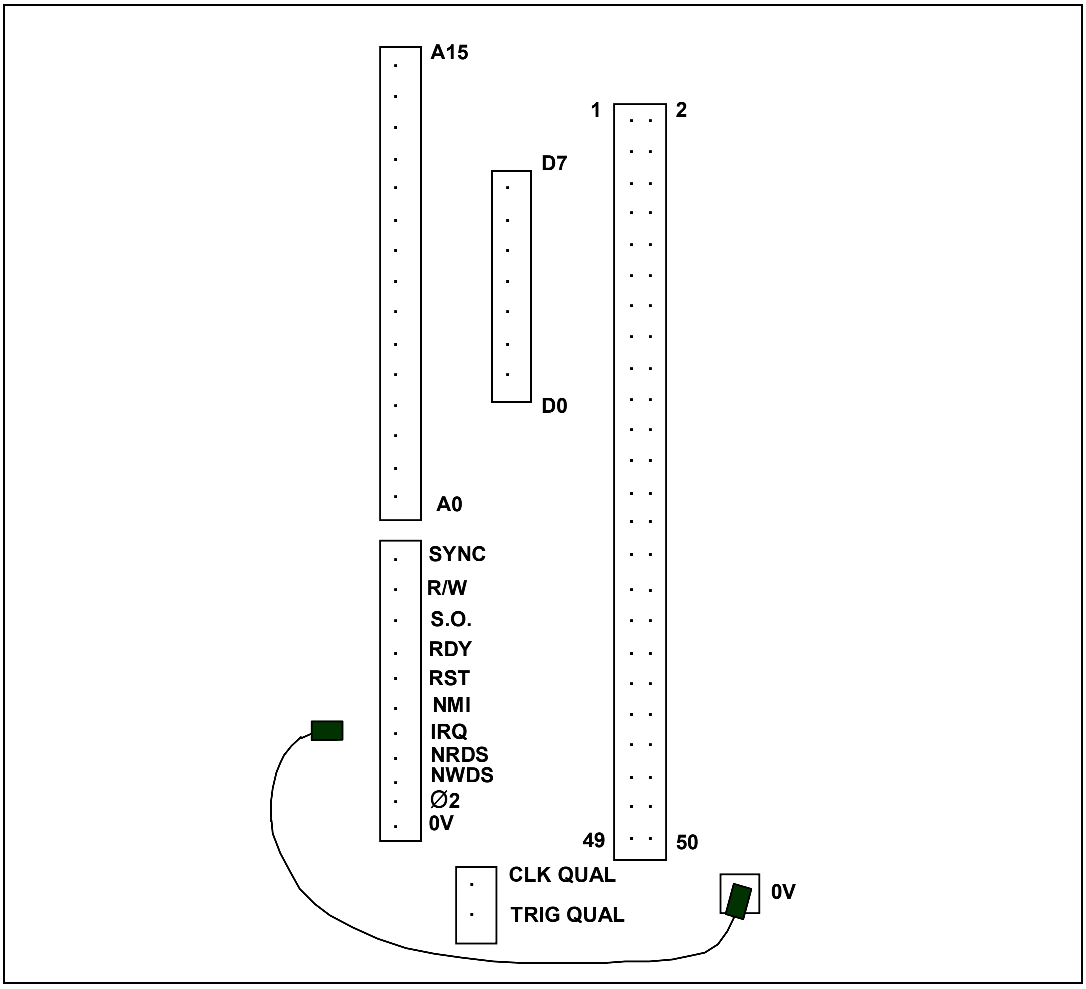
Interrupt Vectors:

```
                                ORG   $0200
0200   00                       WORD  $0500       ;Points to location
0201   05                                         ;0500H
```

Note that the 'WORD' statement is used by the 6502 Cross Assembler to store a two-byte value in memory (low byte first). If you are not using the 6502 Cross Assembler software, the contents of memory locations $0200 and $0201 must be entered using the memory edit facility at the MAC III keypad.

Load the above program, $\overline{\text{NMI}}$ routine and user interrupt vectors into MAC III memory.

You will be able to produce a non-maskable interrupt by using one of the short jumper leads supplied with this manual. Connect the lead to 0V but **do not** connect the other end to $\overline{\text{NMI}}$ for the moment:



Run the main program (from 0400_H). Press reset and examine the contents of location 0040_H. You will find this to be 80_H, since no interrupt has occurred.

Run the main program again and carefully **touch** the free end of the jumper lead on the NMI pin. Press the reset key and examine the contents of location 0040_H. You should find this to be 01_H, since a $\overline{\text{NMI}}$ **has** now occurred.

16.6a **The program for Worked Example 16.6 is to be modified so that the NMI routine starts at location 0580_H. Enter the address for the memory location that must be changed.**

16.7 Practical Assignment

Write a program that will activate the piezo sounder if a non-maskable interrupt occurs.

16.7a **In your program for Practical Assignment 16.7, the program section that produces an output on the piezo sounder is within the:**

a Main Program

b NMI Routine

c Software Interrupt

d Delay Subroutine

Maskable Interrupt ($\overline{\textbf{IRQ}}$)

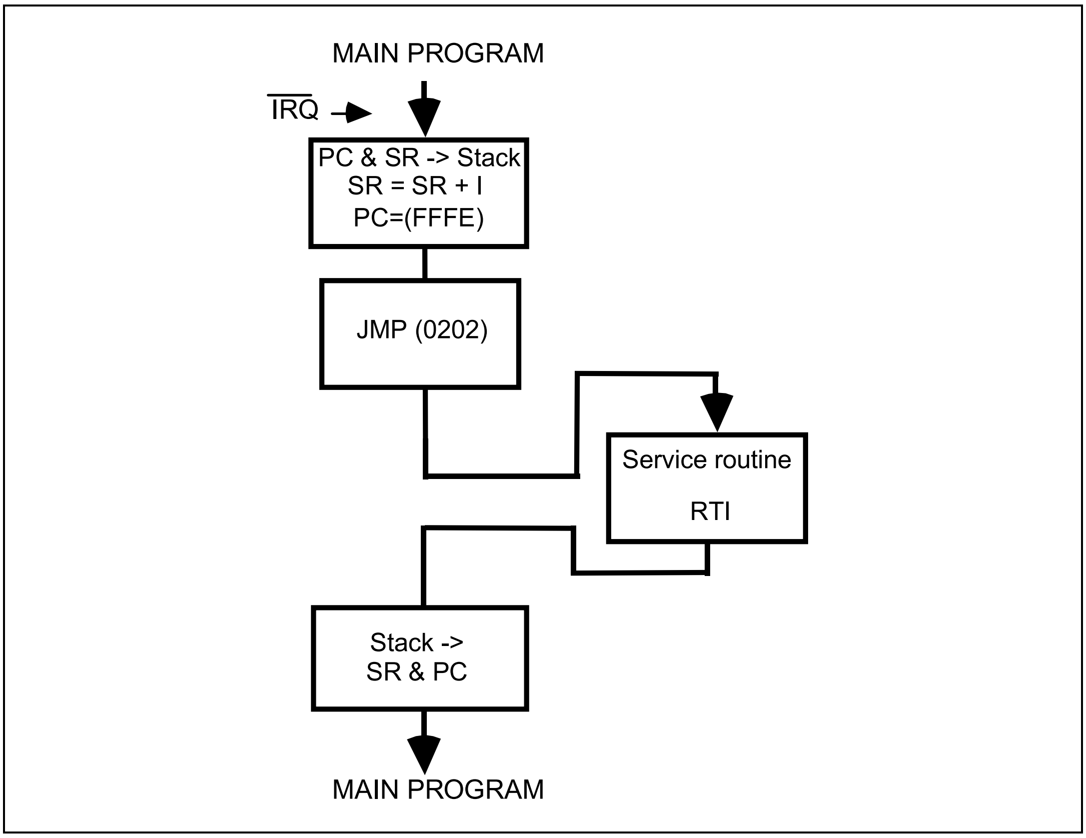
This is also called "Interrupt Request".

$\overline{\text{IRQ}}$ is an **active low**, **level triggered** input. This means that it is activated by a logic "0" on the $\overline{\text{IRQ}}$ pin of the 6502.

The response to an $\overline{\text{IRQ}}$ is listed below:

1. The **I-Flag** is tested: If this flag is **set**, the 6502 will effectively **ignore** the interrupt request and proceed with the next instruction in sequence. If the I-flag is **clear** then the interrupt request **must** be acknowledged and steps 2 to 8 below are carried out:
2. The **high byte** of the **program counter** is pushed onto the stack.
3. The **low byte** of the **program counter** is pushed onto the stack.
4. The **status register** is pushed onto the stack.
5. The **interrupt mask flag** is **set**, to prevent further interrupts from being serviced.
6. The contents of location FFFE_H are fetched and placed in the **low byte** of the program counter.
7. The contents of location FFFF_H are fetched and placed in the **high byte** of the program counter.
8. Program execution continues from the location pointed to by the program counter.

Note: Just like the $\overline{\text{NMI}}$, when an $\overline{\text{IRQ}}$ occurs, the 6502 will **automatically** save the contents of the status register on the stack.



Again, like the $\overline{\text{NMI}}$ vectors, the MAC III monitor program **redirects** the $\overline{\text{IRQ}}$ interrupt vectors thus:

User IRQ Vector:	Location 0202_H:	Low byte of interrupt vector
	Location 0203_H:	High byte of interrupt vector

So these are the vectors for user programs, rather than $FFFE_H$ and $FFFF_H$.

16.8 Worked Example

Write a program that will load location 0040$_H$ with the value 55$_H$. An $\overline{\text{IRQ}}$ interrupt routine is also required that will reload location 0040$_H$ with 88$_H$.

This is very similar to the previous Worked Example but with two important differences:

1. The user interrupt vectors are now at locations 0202$_H$ and 0203$_H$.
2. The main program must **clear** the interrupt disable flag so that the IRQ can be acknowledged.

Again, the main program and interrupt service routine are quite trivial:

Main Program:

```
                         ORG   $0400       ;Main program start address
0400   58                CLI               ;Clears interrupt disable
                                           ;flag to allow maskable
                                           ;interrupts
0401   A9                LDA   #$55
0402   55
0403   85                STA   $40         ;Saves marker to location
0404   40                                  ;0040H
0405   4C   HERE:        JMP   HERE        ;Wait forever - dummy
0406   05                                  ;program
0407   04
```

The IRQ routine can again be placed anywhere in memory:

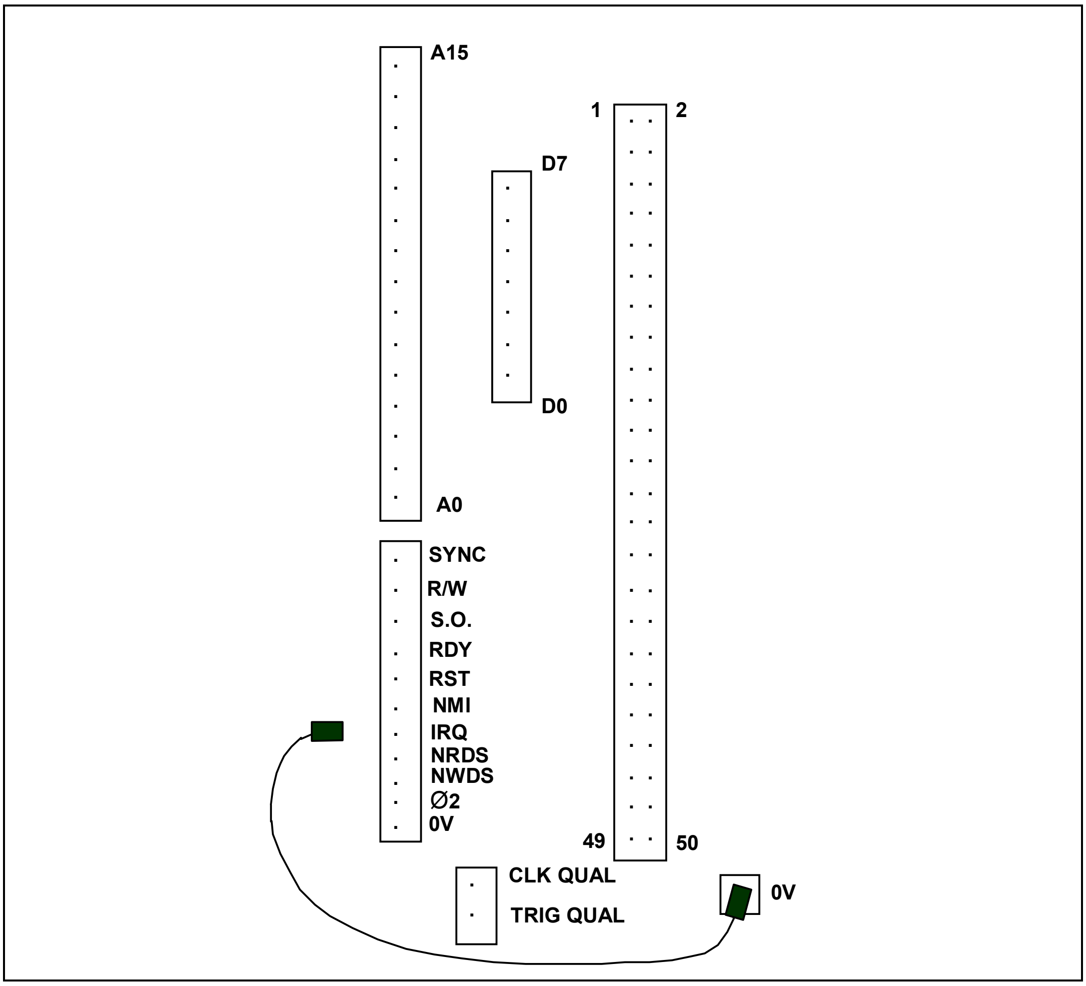
```
                         ORG   $0500       ;IRQ routine start address
0500   A9                LDA   #$88
0501   88
0502   85                STA   $40         ;Changes marker in
0503   40                                  ;location 0040H
0504   40                RTI               ;Return to dummy program
```

Interrupt Vectors:

```
                         ORG   $0202
0202   00                WORD  $0500       ;Points to location
0203   05                                  ;0500H
```

Enter the above program, $\overline{\text{IRQ}}$ routine and user interrupt vectors into MAC III memory.

You will be able to produce a maskable interrupt by again using one of the short jumper leads supplied with this manual. Connect the lead to 0V but **do not** connect the other end to $\overline{\text{IRQ}}$ for the time being:



Run the main program (from 0400_H). Press reset and examine the contents of location 0040_H. You will find this to be 55_H, since no interrupt has occurred.

Run the main program again and carefully **touch** the free end of the jumper lead on the $\overline{\text{IRQ}}$ pin. Press the reset key and examine the contents of location 0040_H. You should find this now to be 88_H, since an $\overline{\text{IRQ}}$ **has** occurred.

Change the instruction at location 0400_H to **SEI** (Opcode 78_H) and run the main program again. If you press rest and then examine the contents of location 0040_H you will find that these are 55_H again, indicating that an interrupt has **not** occurred.

Run the main program again but this time produce a maskable interrupt by carefully touching the free end of the jumper lead on the $\overline{\text{IRQ}}$ pin. Press the reset key and examine the contents of location 0040_H. You should now find that the contents of location 0040_H are **still** 55_H. This is because the new instruction at location 0400_H has **prevented** maskable interrupts from being acknowledged.

16.8a **The effect of removing the instruction at location 0400_H in the program for Worked Example 16.8 would be to:**

a change the marker value saved

b load different interrupt vectors

c prevent the main program from being interrupted

d have no effect

16.9 Worked Example

Write a program that will load location 0040_H with the value AA_H. If a $\overline{\text{NMI}}$ interrupt occurs, location 0040_H must be reloaded with the value $0F_H$. If an $\overline{\text{IRQ}}$ interrupt occurs, location 0040_H must be reloaded with the value 71_H. This is a combination of Worked Examples 16.6 and 16.8:

```
Main Program:
                         ORG   $0400      ;Main program start address
0400   58                CLI              ;Enable maskable interrupts
0401   A9                LDA   #$AA       ;Save marker value in
0402   AA                                 ;location 0040H
0403   85                STA   $40
0404   40
0405   4C   HERE:        JMP   HERE       ;Wait forever - dummy
0406   05                                 ;main program
0407   04
Interrupt Vectors:
                         ORG   $0200
0200   00                WORD  $0500      ;NMI vectors - 0500H
0201   05
0202   20                WORD  $0520      ;IRQ vectors - 0520H
0203   05
NMI routine:
                         ORG   $0500      ;NMI routine start address
0500   A9                LDA   #$0F       ;Saves marker for NMI
0501   0F                                 ;in location 0040H
0502   85                STA   $40
0503   40
0504   40                RTI              ;Returns to main program
IRQ routine:
                         ORG   $0520      ;IRQ routine start address
0520   A9                LDA   #$71       ;Saves marker for IRQ
0521   71                                 ;in location 0040H
0522   85                STA   $40
0523   40
0524   40                RTI              ;Returns to main program
```

Load the program, vectors and interrupt routines into MAC III memory and run the main program. Press reset and check that location 0040_H contains AA_H, indicating that no interrupts have occurred.

Run the main program again and then use the short jumper lead to produce a non-maskable interrupt. This can be done by again connecting to 0V and carefully touching the free end on the $\overline{\text{NMI}}$ pin.

You will now find that location 0040_H contains $0F_H$. This shows that a $\overline{\text{NMI}}$ has been acknowledged. Run the main program again and use the jumper lead to produce a maskable interrupt by connecting to 0V and carefully touching the free end on the $\overline{\text{IRQ}}$ pin. Examine the contents of location 0040_H. You will find that location 0040_H contains 71_H. This shows that the $\overline{\text{IRQ}}$ has been acknowledged.

16.9a **The effect of removing the instruction at location 0400_H in the program for Worked Example 16.9 would be to:**

a change the marker value saved

b cause continuous interrupts

c prevent the main program from being interrupted

d only allow a NMI to interrupt the main program

16.10 Practical Assignment

Write a program that will continually output 99_H at Port A. If a non-maskable interrupt occurs, the piezo sounder should also be activated.

16.10a **In your program for Practical Assignment 16.10, the section of the program that configures the Ports is the:**

a Main Program

b NMI routine

c IRQ routine

d Delay Subroutine

16.11 Software Interrupt (BRK)

Recall that a Software Interrupt is an **instruction** that causes an interrupt response. The interrupts we have seen up to now are caused by a **logic level change** on a 6502 pin (**hardware interrupt**).

The 6502 Break instruction (BRK) has an interrupt response that is almost **identical** to the $\overline{\text{IRQ}}$ response (they even share the same interrupt vectors).

The differences are:

1. When a BRK occurs, the B-Flag is **set**. This allows a BRK and an $\overline{\text{IRQ}}$ to be distinguished.
2. Setting the Interrupt Disable Flag cannot mask a BRK.

The BRK instruction is useful for debugging or to return control to the monitor program. You will have already seen this in the MAC III breakpoint facility.

16.12 Reset

This interrupt input has priority over all others. The reset response is initiated by a logic low (0) to high (1) transition on the RESET pin of the 6502.

The response to such a transition is as follows:

1. A delay of 6 clock cycles occurs. This is to allow internal initializations to take place.
2. The Interrupt Disable Flag is **set** to prevent maskable interrupts from being acknowledged.
3. The contents of location FFFC_H are fetched and placed in the **low byte** of the program counter.
4. The contents of location FFFD_H are fetched and placed in the **high byte** of the program counter.
5. Program execution continues from the location pointed to by the program counter.

Note that the present program counter contents are **not** pushed onto the stack. This is because it will not be necessary to **return** from a reset. Examine the contents of locations FFFC_H and FFFD_H in the MAC III.

You should find that these form the address F022_H. So, the MAC III monitor program starts at location F022_H.

Note: Depending upon the version of the MAC III monitor, you may find different values in locations FFFC$_H$ and FFFD$_H$. However, these locations will **always** contain the start address for the MAC III monitor.

If you examine MAC III memory from location F022$_H$, you can see the first few instructions of the monitor program:

```
F022   78   SEI                   ;Disable maskable interrupts
F023   D8   CLD                   ;Set to Hexadecimal Arithmetic
F024   A9   LDA  #$00
F025   00
F026   8D   STA  $0266            ;Clear location 0266H
F027   66
F028   02
F029   A2   LDX  #$80
F02A   80
F02B   9A   TXS                   ;Set Stack Pointer to 0180H
```

The MAC III allows three types of reset:

Cold Reset	Takes place when power is applied to MAC III board. This type of reset will set variables to their start-up values and clear the contents of RAM.
Monitor Restart	Takes place when the reset switch is pressed **twice** (with a delay of about 0.5 seconds). This type of reset will set variables to their start-up values but **not** clear the contents of RAM.
Warm Reset	Takes place whenever the reset switch is pressed. This type of reset will not change any variables or programs.

16.12a The interrupt vectors for a 6502 Software Interrupt (BRK) are at locations:

a FFF8$_H$ and FFF9$_H$

b FFFA$_H$ and FFFB$_H$

c FFFC$_H$ and FFFD$_H$

d FFFE$_H$ and FFFF$_H$

16.12b **The 6502 interrupt which does not save the current program counter contents on the stack is:**

- a BRK
- b IRQ
- c NMI
- d Reset

16.13 Auto-Run Programs

Normally the "rEAdy" message is displayed for all types of restart, provided the MAC III is not connected to a Personal Computer. However, it is possible to configure MAC III such that **any** given program in RAM or EPROM can be **automatically** executed upon completion of loading from cassette or RS232 port.

To auto-run a RAM program, location 0206_H must be loaded with a non-zero value and then the MAC III must be reset.

Try this now by loading location 0206_H with 01_H and entering a program (via the keypad) which only consists of an unconditional jump to the Applications Module demonstration program thus:

```
0206  01

0400  4C        JMP  $F600
0401  00
0402  F6
```

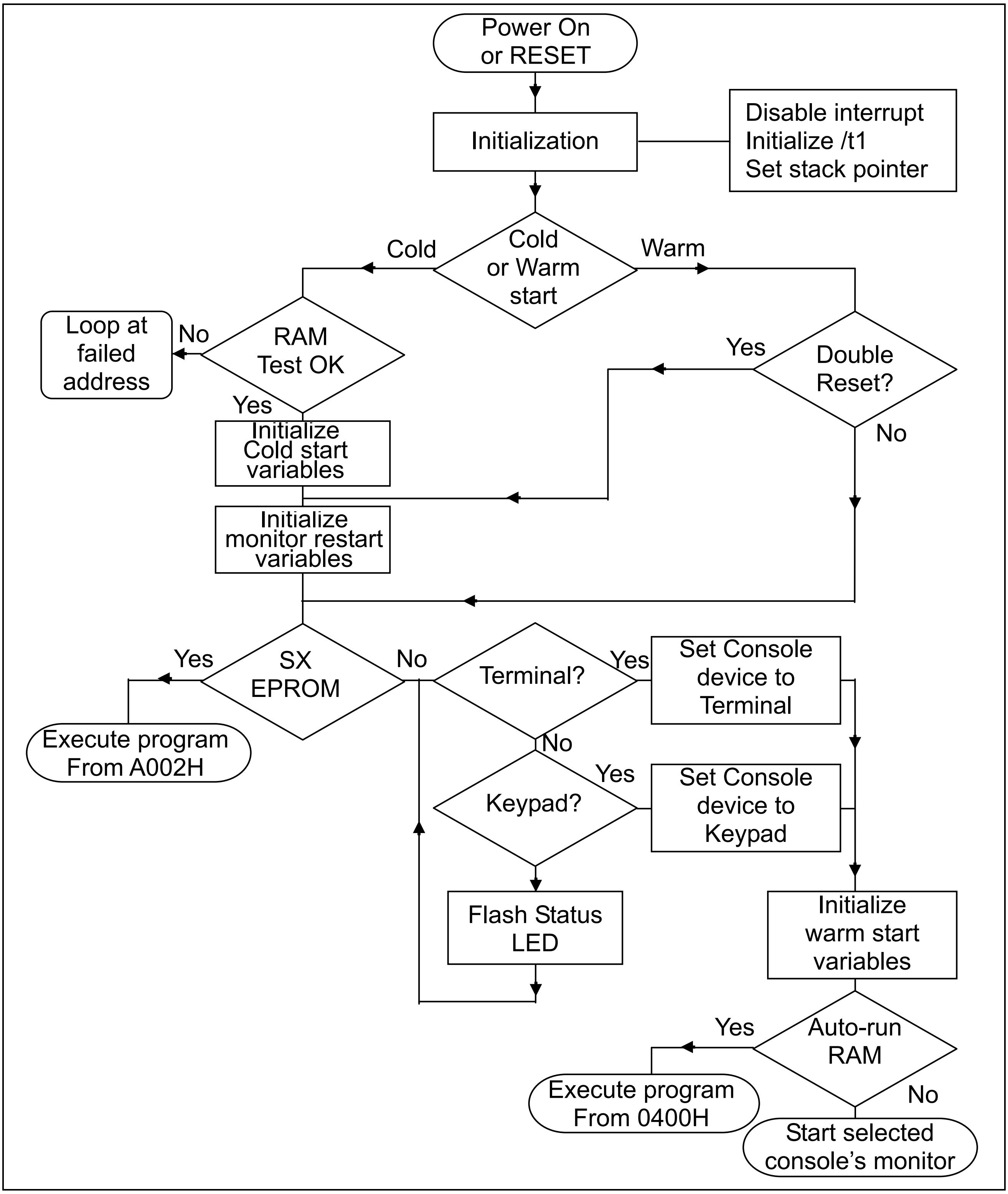
Now, press the reset switch and the Applications Module program will run.

An auto-run program may be stopped by a double closure of the reset key (Monitor Restart).

Programs may also be auto-run from EPROM. The start address for the user EPROM is $A000_H$. The first few bytes of an auto-run EPROM must be:

A000	53	ASCII code for "S"
A001	58	ASCII code for "X"
A002	XX	First byte of program

Clearly then, the reset routine must examine the contents of location 0206_H (to see if a RAM auto-start is required) and also check locations $A000_H$ and $A001_H$ (to see if a ROM auto-start is required).


Power On
or RESET
Initialization
Disable interrupt
Initialize /t1
Set stack pointer
Cold
or Warm
start
Cold
Warm
RAM
Test OK
No
Loop at
failed
address
Yes
Double
Reset?
Yes
No
Initialize
Cold start
variables
Initialize
monitor restart
variables
SX
EPROM
Yes
No
Execute program
From A002H
Terminal?
Yes
Set Console
device to
Terminal
No
Keypad?
Yes
Set Console
device to
Keypad
Flash Status
LED
Initialize
warm start
variables
Auto-run
RAM
Yes
No
Execute program
From 0400H
Start selected
console's monitor

16.13a **The MAC III location that is used to control the Auto-Run facility is:**

a 0200_H

b 0206_H

c $FFFC_H$

d $FFFE_H$

16.14 Practical Assignment

Write a program that displays "HELLO" on the MAC III display. If a non-maskable interrupt occurs the display should change to "NON MASK". If a maskable interrupt occurs the display should change to "MASKABLE".

16.14a **In your program for Practical Assignment 16.14, the number of different interrupt service routines is:**

a 1

b 2

c 3

d 4

16.14b **Your program for Practical Assignment 16.14 is to be modified such that it will not respond to maskable interrupts. The part of the program that must be altered is the:**

a main program

b IRQ routine

c NMI routine

d display subroutine

Student Assessment 16

1. **An input to a microprocessor that causes it to suspend the current program is called:**

   a a Break
   b an Interrupt
   c a Stop
   d a Wait

2. **Usually, when an interrupt service routine has been completed:**

   a the microprocessor must be reset
   b an interrupt will occur
   c the interrupted program is resumed
   d a Break occurs

3. **The process of a microprocessor periodically checking a peripheral to see if it is ready for data transfer is called:**

   a Nested Input/Output
   b Interrupt Input/Output
   c Stack Input/Output
   d Polled Input/Output

4. **The main advantage of Interrupt Input/Output, as compared with Polled Input/Output is that it:**

   a does not waste microprocessor time
   b is more reliable
   c can handle errors more effectively
   d prevents bus conflicts

Continued ...

Student Assessment 16 Continued ...

5. **An interrupt service routine that has been interrupted by a second interrupt is an example of:**

   - a Interrupt Input/Output
   - b Maskable Interrupts
   - c Nested Interrupts
   - d Software Interrupts

6. **Interrupt inputs that the microprocessor may ignore are said to be:**

   - a High Priority
   - b Maskable
   - c Nested
   - d Non-maskable

7. **The 6502 instruction that allows maskable interrupts to be acknowledged is:**

   - a `CLI`
   - b `BRK`
   - c `RTI`
   - d `SEI`

8. **The 6502 instruction mnemonics for an indirect Jump to the location pointed to by locations 0400_H and 0401_H are:**

   - a `JMP $0040`
   - b `JMP ($0040)`
   - c `JMP $0400`
   - d `JMP ($0400)`

Student Assessment 16 Continued ...

9. The vector for the 6502 NMI interrupt is at locations:

a FFF8$_H$ and FFF9$_H$

b FFFA$_H$ and FFFB$_H$

c FFFC$_H$ and FFFD$_H$

d FFFE$_H$ and FFFF$_H$

10. The 6502 instruction that is usually found at the end of an interrupt service routine is:

a `BRK`

b `RTI`

c `RTS`

d `JMP  ($0200)`

11. The highest priority 6502 interrupt is:

a BRK

b IRQ

c NMI

d Reset

12. The 6502 addressing mode used to redirect interrupt vectors is called:

a absolute

b absolute indirect

c absolute indexed

d zero page

Appendix 1 Standard Programming Sheet

Address	Machine Code	Label	Assembly Language	Comments

Appendix 2 MAC III System Calls

Introduction

This section lists the system subroutines available to the user. The calls are divided into three groups:

System calls	A collection of routines some of which interface with various devices on the MAC board. Most of the calls in this group are also used by the monitor itself.
Math system calls	A collection of ASCII, decimal and hexadecimal conversion routines.
User system calls	A collection of routines commonly required by user programs.

The following points apply to all monitor system calls:

- Input and output parameters use the various registers and memory addresses indicated in the description for each call, registers not specified are unaffected. Memory addresses within the system workspace are also used by many of the calls.
- If an error condition arises during a call the routine will exit with the carry flag set and an error code number in the Accumulator.
- The label **PTR** is a 16-bit pointer stored as two bytes at address 0000 (low byte) and 0001 (high byte). It is used as a pointer in many of the system calls. When using the LJ 6502 cross assembler software a label, **PTR**, could be set with an equate statement at the start of your program:

```
PTR         equ  0
```

- The label **NUMBER** is a 16-bit store used by the number conversion routines. It is arranged as two 8-bit bytes at addresses 0002 (low byte) and 0003 (high byte). When using the LJ 6502 cross assembler software a label, **NUMBER**, could be set with an equate statement at the start of your program:

```
NUMBER      equ  2
```

Device Interface System Calls

Device interface system calls read from or write to a specified device. The device number is always passed in the accumulator. The device numbers for each device are:

Device	**Name**	**Number**
/t1	Terminal port 1	0
/t2	Terminal port 2	1
/p	Centronics port	2
/kd	Keypad/display unit	3
/cas	Cassette interface	4

Transmitting a carriage return character (0D) to the keypad/display unit causes subsequent data to appear starting from the leftmost LED display. Transmitting a linefeed (0A) or a formfeed (0C) clears the LED display and performs an automatic carriage return.

If more than eight characters are sent to the keypad/display, the display scrolls to the left to accommodate the new characters.

READ Read raw data from a device

Address:	C000	
Input:	Acc	Device number
	Y	Maximum number of bytes to read
	PTR	Address of input buffer
Output:	Y	Number of bytes actually read.
Function:	Reads the specified number of bytes from the device number given.	

Data is returned exactly as read from the device, without additional processing such as backspace or line delete. The routine exits when a carriage return character, 0D, is entered or when Y bytes have been read.

The number of bytes actually read is returned in the Y register.

If the ESCape character (1B) is entered, the error code 25 will be returned with the carry flag set.

READLN Read edited data from a device

Address:	C004	
Input:	Acc	Device number
	Y	Maximum number of bytes to read
	PTR	Address of input buffer
Output:	Y	Number of bytes actually read
Function:	Similar to READ except that that it reads data until a carriage return character is encountered, line editing also takes place, ie. line delete, backspace.	

The last byte to be entered must be a carriage return. If more than the maximum number of characters are entered, subsequent characters, except for carriage return, line delete or backspace, will be ignored. For example, a READLN call of one byte will accept only a carriage return and ignore any other characters.

WRITE Write raw data to device

Address:	C008	
Input:	Acc	Device number
	Y	Number of bytes to write
	PTR	Address of buffer
Output:	Y	Actual number of bytes written
Function:	Outputs Y bytes to the specified device, data is written with no processing or editing.	

WRITLN Write edited data to a file

Address:	C00C	
Input:	Acc	Device number
	Y	Maximum number of bytes to write
	PTR	Address of buffer
Output:	Y	Actual number of bytes written
Function:	Outputs Y bytes from the buffer to the specified device.	

This call is similar to WRITE except that it writes data until a carriage return character is encountered, or Y bytes are written. Line editing takes place. If the device supports auto-linefeed, then a linefeed is sent after each carriage return. The extra linefeeds are not included in the character count.

EXIT Terminate a program

Address: C010
Input: Acc Error code to return to calling program
Output: None
Function: Exits the program and returns control back to the MAC monitor. If the Acc register contains a non-zero value, the error message corresponding to that error code will be displayed on the terminal screen.

PERR Print error message

Address: C014
Input: Acc Error code
Output: None
Function: Prints an error message to the terminal.

The error number is printed as two decimal bytes, for example:

```
lda  #50
jsr  $c014
```

Displays: ERROR 50 :

If the error is a standard MAC error, an error message string is also printed, for example:

```
lda  #$14
jsr  $c014
```

Prints: ERROR 20 : Device not ready

If the error code is zero, this routine does nothing.

MATH SYSTEM CALLS

The math system calls make use of the 16-bit number store NUMBER at address 0002-0003 previously described.

AHEXTO Convert ASCII hex to value

Address:	C020	
Input:	PTR	Address of string
Output:	NUMBER	Value
	PTR	Updated to point to first non hex character
Error output:	Carry set if no hex digits, error code 06 returned.	
Function:	Converts the ASCII hexadecimal string pointed to by PTR, into a value in NUMBER. Conversion stops at the first non-hexadecimal digit.	

ADECTO Convert ASCII decimal to value

Address:	C024	
Input:	PTR	Address of string
Output:	NUMBER	Value
	PTR	Updated to point to first non hex character
Error output:	Carry set if no decimal digits, error code 06 returned.	
Function:	Converts the ASCII decimal string pointed to by PTR, into a hexadecimal value in NUMBER. Conversion stops at the first non-decimal digit.	

TOAHEX Convert value to ASCII hex

Address:	C028	
Input:	NUMBER	Value to be converted
	Y	Number of digits output required
	PTR	Address of buffer
Output:	ASCII	string in callers buffer
	PTR	Updated past string
Function:	Converts the value in NUMBER into an ASCII hexadecimal string in the buffer pointed to by PTR.	

If the Y register specifies more digits than the number represents, leading zeroes will be inserted. If Y specifies less digits than the converted number, the least significant digits will be returned.

TOADEC Convert value to ASCII decimal

Address:	C02C	
Input:	NUMBER	Value to be converted
	Y	Number of digits output required
	PTR	Address of buffer
Output:	ASCII	string in callers buffer
	PTR	Updated past string
Function:	Converts the value in NUMBER into an ASCII decimal string in the buffer pointed to by PTR.	

If the Y register specifies more digits than the number represents, leading zeroes will be inserted. If Y specifies less digits than the converted number, the least significant digits will be returned.

USER SYSTEM CALLS

The user trap calls are a collection of routines which provide a more convenient interface to the operating system. The standard output device is the device through which the user is currently interacting. The system calls in this section will output to the terminal, if the MAC is being used through the terminal, or to the keypad/display if the MAC is being used through the keypad/display unit. The current standard output device is called the **console device**.

RDCHAR Read one character

Address:	C040	
Input:	None	
Output:	Acc	Character code
Function:	Reads one character from the keyboard buffer.	

This call uses READ to get one character from the console device. If the keypad is the console device, then the ASCII key code is returned (i.e. the keypad $\boxed{\textbf{M}}$ key is returned as ASCII 77, or "M"). If an escape character is returned by READ, then the escape character (1B) is returned in the A register and not an escape error. You cannot generate an escape character from the keypad.

RDBYTE Read ASCII hexadecimal byte

Address: C044
Input: None
Output: Acc Hexadecimal byte
Function: Read a 2 digit hexadecimal number from the console device.

If the console device is the terminal this function is implemented using READLN, a two digit number followed by a carriage return is required. (normal line editing is allowed) If the console device is the keypad/display this function is implemented using READ to read two of the hexadecimal keys in succession.

An illegal number error is returned for non-hex entry.

WRCHAR Write one character

Address: C048
Input: Acc Character code
Output: None
Function: Writes the character in Acc to the console.

This function is implemented using WRITE to send the character to the console device.

WRBYTE Write byte in ASCII hexadecimal

Address: C04C
Input: Acc Hexadecimal byte to write
Output: None
Function: Writes the hexadecimal byte as two ASCII characters to the console.

This function is implemented using TOAHEX and WRITE to send the byte to the console device.

GETIN Get character from keyboard

Address: C050
Input: None
Output: Acc Character code
C Carry set if no character available
Function: GETIN is used to see if there is a key pressed on the keypad or a character in the RS232 receive buffer, (depending on which console device is active).

If no character is available, the call returns with the carry flag set. If a character is available its ASCII code is returned in the Acc and the carry is cleared. The character is not echoed to the display.

WT1MS Wait for one millisecond

Address: C054
Input: None
Output: None
Function: Delays for one millisecond.

This function uses a software delay loop which has been calculated to produce an accurate delay of 1ms. Interrupt service routines may cause the delay length to change.

WTNMS Wait for n milliseconds

Address: C058
Input: Acc Number of milliseconds to wait
Output: None
Function: Waits for Acc * milliseconds.

This function effectively calls WT1MS the number of times contained in the accumulator.

CRLF Output carriage return, linefeed

Address: C05C
Input: None
Output: None
Function: Outputs carriage return, linefeed sequence.

This function is implemented using Write to write a carriage return and linefeed character to standard output.

CLRSCR Clear screen

Address: C060
Input: None
Output: None
Function: Clears the console screen.

This function is implemented by writing ASCII FormFeed (0C) to standard output. On the keypad/display unit, all LEDs are cleared and the cursor is reset to the leftmost LED.

LEDON Switch on Status LED

Address: C064
Input: None
Output: None
Function: Switches on the status LED.

LEDOFF Switch off the Status LED

Address: C068
Input: None
Output: None
Function: Switches off the status LED. This function is the complement of LEDON.

Appendix 3 ASCII Codes

Character	ASCII Code (hex)	Character	ASCII Code (hex)	Character	ASCII Code (hex)
space>	20	@	40	`	60
!	21	A	41	a	61
"	22	B	42	b	62
#	23	C	43	c	63
$	24	D	44	d	64
%	25	E	45	e	65
&	26	F	46	f	66
'	27	G	47	g	67
(	28	H	48	h	68
)	29	I	49	i	69
*	2A	J	4A	j	6A
+	2B	K	4B	k	6B
,	2C	L	4C	l	6C
-	2D	M	4D	m	6D
.	2E	N	4E	n	6E
/	2F	O	4F	o	6F
0	30	P	50	p	70
1	31	Q	51	q	71
2	32	R	52	r	72
3	33	S	53	s	73
4	34	T	54	t	74
5	35	U	55	u	75
6	36	V	56	v	76
7	37	W	57	w	77
8	38	X	58	x	78
9	39	Y	59	y	79
:	3A	Z	5A	z	7A
;	3B	[	5B	{	7B
<	3C	\	5C	|	7C
=	3D	]	5D	}	7D
>	3E	^	5E	~	7E
?	3F	_	5F		