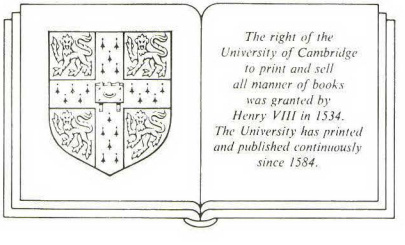
APPLE II in the laboratory

A.F.KUCKES & B.G.THOMPSON
School of Applied and Engineering Physics, Cornell University



CAMBRIDGE UNIVERSITY PRESS
Cambridge
New York New Rochelle Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First published 1987

Printed in Great Britain at the University Press, Cambridge

British Library Cataloguing in Publication Data

Kuckes, A. F.
Apple II in the Laboratory.
1. Physics – Experiments – Data processing
2. Apple II (Computer)
I. Title II. Thompson, B. G.
530'.028'54165 QC52

Library of Congress Cataloguing in Publication Data

Kuckes, Arthur F.
Apple II in the Laboratory.

Bibliography:
Includes index.

1. Physical Laboratories – Data processing.
2. Apple II (Computer) I. Thompson, B. G. II. Title.
III. Title: Apple 2 in the laboratory. IV. Title: Apple two in the laboratory.
QC52.K83 1987 004.165 86–21531
ISBN 0 521 32198 0

Achilles: Before we start, I just was wondering, Mr. Crab—what are all these pieces of equipment, which you have in here?

Crab: Well, mostly they are just odds and ends—bits and pieces of old broken phonographs. Only a few souvenirs (*nervously tapping the buttons*), a few souvenirs of—of the TC-battles in which I have distinguished myself. Those keyboards attached to television screens, however, are my new toys. I have fifteen of them around here. They are a new kind of computer, a very small, very flexible type of computer—quite an advance over the previous types available. Few others seem to be quite as enthusiastic about them as I am, but I have faith that they will catch on in time.

Achilles: Do they have a special name?

Crab: Yes; they are called "smart-stupids", since they are so flexible, and have the potential to be either smart or stupid, depending on how skillfully they are instructed.

Contents

Acknowledgements
We wish to thank the students who struggled with us through the first semester of this course and Professor Bruce Kusse and Tom Hughes for their constructive comments on the notes. We also thank Helen Savey and Bonni Jo Davis for the hours of typing and retyping of the drafts. Andy Draudt drew the pictures of the apparatus.

1 Introduction

The microprocessor has become commonplace in our technological society. Everything from dish washers to astronomical telescopes have chips controlling their operation. While the development of applications for computers has been in constant flux since their introduction, the principles of computer operation and of their use in sensing and control have remained stable. Those are the primary subjects of this course. Once a basic understanding of the principles has been built, further detailed knowledge can be acquired later as the need arises.

This book is designed to be acompanied by extensive laboratory work. Over the years the engineering curriculum has focused more and more on the lecture/recitation format. This has led to an ever increasing emphasis on theoretical developments and a loss of contact with the physical basis of engineering and science. The laboratory provides a vital experience in linking theory with physical reality. It also provides the satisfaction of building something and making it work.

Not all computers are suitable for laboratory use. Large mainframe computers are fast and can handle large amounts of data but are awkward to connect to laboratory equipment. At the other end of the scale, microprocessors are included in many laboratory devices but are programmed to perform only a restricted set of duties. Mini and microcomputers have enough speed and memory for all but the most demanding applications but yet are small enough to be dedicated to individual projects and therefore are widely used in the laboratory.

With the technological strides of recent years, microcomputers (or personal computers) have prodigious capabilities. Since they are also used in business, many languages and programs are available; some of which are even useful in a laboratory. With a single microcomputer, an engineer or scientist can acquire data and control an experiment, analyze the data, display the data and analysis as graphs or tables, and write a report or journal article. Remember that it can't do the thinking!

Microcomputers come with many built-in features. Those included depend on the designers' decisions as to what will sell the most computers. Since the demands of the laboratory are so varied, no computer when taken out of the box can hope to fulfill them. Hence to be useful, a computer must be able to change its capabilities after manufacture. This is done in two ways. One is provide a method of communication (serial or parallel) between the

computer and the laboratory devices (analog to digital converters, voltmeters, etc.) and rely on the devices to be intelligent enough to communicate. Generally this means that the devices need a microprocessor built in which is preprogrammed to communicate with a certain protocol. The other way is to have slots (connectors) in the back of the computer so that circuit boards can be inserted to perform the desired tasks such as analog to digital conversion of serial communication. The computer can then be configured exactly as needed for a particular application. Even the video display or microprocessor can be changed when desired. Further, 'slot machines' are generally the least expensive way to computerize the laboratory.

The APPLE IIe (upon which this book is based) and the IBM-PC (which is the subject of a companion volume) are 'slot machines'. The APPLE IIe (and predecessors, the APPLE II and APPLE II+) is of an older design but has proved its usefulness in innumerable times. Its simple yet versatile architecture makes the computer easy to use and understand but limits its analysis and data volume capabilities. The IBM-PC is a newer design which is faster and has more memory but is slightly more complicated internally. They are both quite suitable for laboratory use. Beware of the APPLE IIc which is a slotless version of the IIe; it will not be able to accept the circuit cards which are necessary for the exercises in this book. IBM-PC clones (design copies) can be used instead of the IBM-PC as long as they have slots where data acquisition boards can be placed.

A computer can be treated as a black box which responds in a predictable way to an input; however, that type of use requires a complete knowledge of the possible inputs and responses. An understanding of how it works inside allows the user to figure out how the computer will respond to an input, or even if it can respond. The capabilities and limitations become transparent. Throughout the book a gradual understanding of what goes on inside a computer will be developed.

Other devices which are used are various sensors, analog to digital converters, digital to analog converters, timers, digital input and output devices, optical encoders, stepping motors, and analog amplifiers. They provide the interface between the computer's digital world and the physical phenomena being studied.

At first, APPLESOFT BASIC is used for programming. This allows simple input and output to be done as one's understanding of the computer grows. Graphing and curve fitting is also dealt with. Later, when the limitations of BASIC become restricting, programs are written in assembly/machine language. The speed difference become evident very quickly.

All of this computer work is done in the context of doing physics experiments. These experiments cover subjects not usually emphasized in introductory courses but which have a wide applicability. They show that, with computer control, conceptually sophisticated experiments can be performed with simple apparatus. In particular, physics of activation temperature, heat diffusion and motion in fluids are explored.

1.1 How to use this book

In this book much of the programming material will be presented by way of example. Programs will be given from which you will be expected to deduce the essence of what is going on and thereby proceed to write your own programs. After seeing and using programs, the more precise and legal description of instructions given in the manuals become easier to comprehend.

This book is written in a tutorial manner in which the exercises and experiments are distributed throughout the text. It would be nice to read up to an exercise and then sit down at the computer to do it. However, the time in the laboratory is short so that this becomes impossible. Before going to the laboratory, read through the text you expect to cover and organize your thoughts about what you will be doing. Also, jot down flow charts and write out programs which you will enter in the computer at the laboratory. Even if they do not run the first time they can be easily changed once the program is in a file. The essence of learning is going through the struggle of getting things to function properly, whether it be in writing programs, building experimental apparatus, or understanding theoretical descriptions.

Many of the details of BASIC programming and machine operation can be found in *APPLE II User's Guide* by L. Poole, M. McNiff and S. Cook or a complete set of APPLE manuals. Assembly language programming is not covered. A source for that is *6502 Assembly Language Programming* by L. Leventhal or *SY6500 Microcomputer Programming Manual* by MOS Technology.

The appendices contain reference material and extended discussions. They are separated from the main text to improve the flow but contain important information and so should be perused once in a while.

1.2 Chapter summary

Chapter 2 begins with an introduction to the operation of the APPLE IIe computer. The AMPERGRAPH utility and APPLESOFT BASIC are used to make some graphs. There is also a brief look inside the APPLE at addresses, data and different types of memory.

Chapter 3 introduces the first Input/Output (I/O) device, the Analog to Digital Converter (ADC). It is used to measure the temperature/resistance characteristics of a thermistor. Further BASIC programming is used to do a least squares fit to the data. The I/O capabilities of a 6522 Versatile Interface Adapter (VIA) are used to control a HEXFET switch on a heater to make a temperature controller.

In Chapter 4, simple BASIC timing loops are used to control a stepping motor. The operating speed of BASIC statements is measured. Then the more sophisticated counters of the 6522 VIA are used to make an accurate interval timer.

Chapter 5 concerns an experiment in thermal diffusion. A heater at one end of a copper rod is turned on for a set interval under program control. The

flow of this heat pulse down the rod is then measured at two locations for about 30 s. A theoretical model is fitted to these data to determine the thermal conductivity and heat capacity of copper. An analog amplifier is used to boost the signal from the thermistor to the ADC.

Chapter 6 is an introduction to assembly language programming and the architecture of the APPLE. The increase in processing speed over BASIC is vividly displayed by sending a square wave to the output port. A digital to Analog converter (DAC) is used to make an X–Y plotter.

In chapter 7, an experiment is constructed which measures the viscosity of glycerine by measuring the speed of a falling sphere. The physics of turbulent as well as smooth fluid flow is discussed. LEDs and photocells are used as position sensors to measure the speed of the sphere. The machine language portion of the data acquisition program is blasted into an EPROM (Erasable Programmable Read Only Memory), which is used to acquire velocity data.

Chapter 8 introduces the concept of interrupt processing. A clock display runs as other computer operations are performed. A modification of this program rings the bell every two seconds while other programs are run.

Chapter 9 contains various topics which are important but do not have a direct bearing on the experiments done in the previous sections.

2 Instrumentation structures and using APPLE II computer

The purpose of an instrument is to make measurements of a particular parameter in a physical process. This requires at least a sensor which responds to the parameter and a display which lets the user record readings which are in some way proportional to the parameter being measured. A thermometer is an instrument which indicates the temperature by quantitatively showing the expansion of a liquid with a temperature increase. A more complete description of the measurement process is shown in Figure 2.1. The arrows show possible but not necessary routes for the flow of information. The computer is able to do many of the tasks which were formerly done by separate units of an instrument. This lets the designer reduce the number of components required to a bare minimum as the experiments in this book show. Many times all that is needed is a sensor to translate the process into an electrical signal.

Another way to think of the computer is as an interface between the experimenter and the experiment (or the user and the measurement). It is able to translate the unintelligible signals from the sensor into a form which is understandable using human senses. One of the best ways of communicating information is by picture. 'A picture is worth a thousand words.' (In fact, it takes roughly a thousand words of computer memory to display a video graphics screen.)

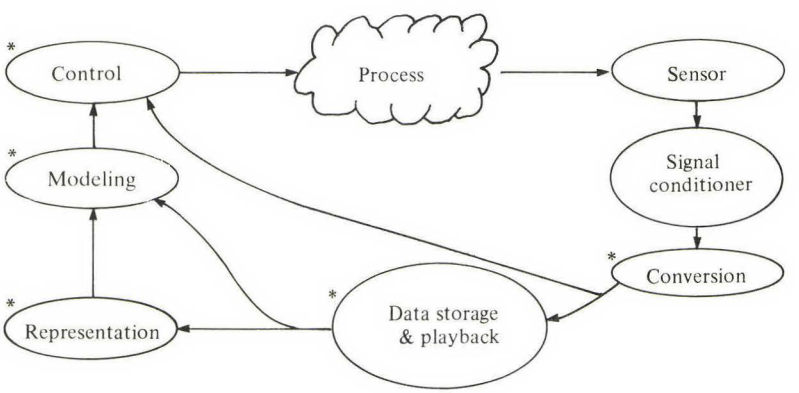


Fig. 2.1. Instrumentation structures
Process: eg temperature as function of time, position as function of temperature.
Sensor: eg temperature or position converted to voltage.
Signal conditioner: eg amplifier, filter.
* Conversion: analog to digital.
* Storage/playback: eg silicon memory, magnetic, papertape.
* Representation: eg numbers, pictures (1000 words of memory!).
* Modeling: mathematical fit to data.
* Control: eg change temperature or position.
The computer has a part in all items with a *.

2.1 Making graphs

Graphing experimental data and mathematical expressions is an important aspect of the work you will do in this course; we will go through a

few programs which will show how such graphics programs are written. The APPLESOFT BASIC interpreter, which you will be using for programming the machine has only rudimentary plotting instructions incorporated in it; the AMPERGRAPH utility appends to the usual APPLESOFT instructions additional graphing instructions which are easy to use. AMPERGRAPH is automatically linked to APPLESOFT BASIC at system start up time. Refer to your AMPERGRAPH manual for descriptions of AMPERGRAPH commands and to the APPLE BASIC programming manual or Poole for APPLESOFT commands.

Exercise 2.1.1 Starting out

(*a*) To get started insert the SYSTEM START disk into the DISK DRIVE unit and turn the computer power switch on. The machine is set up so that it will 'wake up' in APPLESOFT BASIC whose prompt is]. The AMPERGRAPH utility and other programs are on the SYSTEM START disk. To see what files are on this disk, type CATALOG CR (CR means press the key labeled RETURN). On the screen, a listing of the programs will appear. Up to 18 lines of the catalog entries are presented. If this does not include all the files on a particular disk, you can see more by pressing the space bar. The entries marked with an 'A' are APPLESOFT programs.

(*b*) AMPERGRAPH: you will now type into the computer memory a sample APPLESOFT/AMPERGRAPH plotting program. First clear out other programs which may be in the machine by typing NEW CR. Then type in the Program 2.1.1 in Figure 2.2; each line should be followed by CR. Be sure to read the comments in Figure 2.2 to understand each program step. HIMEM is particularly important.

(*c*) Listing: the LIST command will make the APPLESOFT program which you have entered appear on the screen; check the program for errors. The spacing may appear slightly changed but that is normal. To correct any errors simply retype the line number and statements which are incorrect; the old line will be erased. To delete a line, simply type the line number followed by a CR. As you get further along in writing programs, you will find it useful to go through the APPLE manual to learn some further editing features of the APPLE. Notice that the numbering of the statements is not consecutive. This is a good practice in BASIC programs in case you want to put additional program lines between existing lines. To insert a new line between two lines simply type a line with a number between the two line numbers.

(*d*) Printing: next, print a listing of the program on the printer: first turn the power switch to the printer on. Then to direct output to the

Fig. 2.2. An elementary graphing example.

`5 REM PROGRAM 2.1.1` `6 REM ELEMENTARY EXAMPLE`	REMark is a way of recording program documentation, it is not an instruction which is executed.
`8 HIMEM:16383`	HIMEM will limit the usage of memory for APPLESOFT programs to addresses lower than 16383, You must include this before *any* executable (non-REM) APPLESOFT statements in programs using AMPERGRAPH statements to insure that APPLESOFT doesn't write over the AMPERGRAPH program in memory. See Appendix C for the location of AMPERGRAPH in memory.
`10 HGR2`	HGR2 erases all 'dots' stored on page 2 of high resolution graphics and switches the screen to display that page. We will use HGR2 instead of HGR1 in our programs. (Appendix C)
`20 &SCALE,0,10,-1.2,1.2`	All instructions beginning with a & are AMPERGRAPH instructions, read manual for details.
`25 LX$="X"` `27 LY$="COS X"`	In BASIC all variables ending with a $ are string variables which are used to manipulate text. These variable names (LX$ and LY$) are special to AMPERGRAPH
`30 &LABELAXES,2,.2`	An AMPERGRAPH instruction.
`40 for X=0 TO 10 STEP 0.2`	An ordinary APPLESOFT BASIC instruction.
`50 &DRAW, X, COS(X)`	An AMPERGRAPH instruction.
`60 NEXT X`	An APPLESOFT BASIC instruction.
`LIST CR`	A DOS Command.

printer, type PR#1 CR. This will direct subsequent output to the printer as well as to the CRT. Now type LIST CR. To turn off output to printer, type PR#0.

(*e*) Running the program: to run the above program type RUN CR. If the program is correct the graph of cos (*x*) should appear on the screen. If not, find the errors and correct them (debug the program). At this time memory locations devoted to high resolution graphics are being displayed; to get back to displaying text on the screen, type TEXT CR (type carefully since you won't see what you are entering). To get back to viewing the high resolution graphics without wiping out what's there (which the HRG2 instruction will do), type POKE −16304,0:POKE −16299,0 CR.

To print out the graphics display, a control signal must be sent to the printer (actually to the circuit card which controls the printer). The details of this are discussed in the documentation for the printer card you are using. For a MICROBUFFER, to print out the graph you have drawn, type TEXT CR (to see what you are typing), PR#1 CR (to send output to the printer), PRINT CHR$(9) "G2" CR (to print the graph).

(*f*) Formatting a disk: throughout the course you will need to make use of programs and data which you have written or obtained before. The disk is the medium by which these are stored. To save APPLESOFT programs, you first need to initialize a blank disk. Initializing is like erasing the blackboard and then drawing lines on it where the words or numbers will go. The command 'INIT filename CR' does this and also places the program presently in memory into the first part of the disk with the filename given in the command. Only use this command the first time you store a program on a disk since it erases all the existing files on the disk. Later programs are saved on disk with the APPLESOFT command: 'SAVE filename CR'. Place your blank disk into the drive and save this first program on your blank disk with the command: INIT PROGRAM 2.1.2 CR.

Exercise 2.1.2 Simple graphing

Write and run a program that plots the curve $Y = X^2 - 1$ from $X = -2$ to $+2$ with the X and Y axes labeled and a grid superimposed on the plot with a grid line for every increment of 1 for X and 2 for Y. In addition, plot on your graph the data points X, Y as open circles for the points

X	Y
−1.8	3.5
−1.2	1.0
−0.5	−1.0
0.0	−1.3

0.5	0.3
1.2	0.5
2.0	3.5

The BASIC statements READ and DATA are useful in this program. Save the program as an APPLESOFT program on diskette; (SAVE filename) then load (LOAD filename) and RUN it again. Print out the program and the graph on the printer.

For success in writing programs complete small pieces at a time and devise methods so that each piece can be tested separately; then incorporate these pieces into larger sections of the main program. It is also usually a good idea to first write out in words and block diagrams what you are trying to do with the program and/or apparatus. Apparatus B contains a description of RENUMBER, a useful utility on the SYSTEM START disk for merging separate program pieces to facilitate this process.

One note: if you get an error message which reads 'SYNTAX ERROR IN LINE 5*xxxxx*' where the *x*s can be any number, the error is in your AMPERGRAPH statements. Also certain programs are incompatible with AMPERGRAPH. If you have used RENUMBER or INTEGER BASIC/ MINI-ASSEMBLER and you want then to use AMPERGRAPH, you will need to RUN AMPERGRAPH LOADER on the SYSTEM START disk. Be careful to save your program before you do this.

2.2 Addresses and data, RAM and ROM

Inside the APPLE there is an integrated circuit microprocessor which controls the operation of the computer. Connected to it are 16 address wires and 8 data wires which are used to communicate with other parts of the computer. The 16 binary bits of the address wires allow the microprocessor to specify 65536 unique locations. The information transfer is done on the data wires. Eight wires allow 256 unique numbers (or characters) to be represented at one time. It is like having a telephone system which has 65536 telephone numbers and in which the caller can choose from 256 words to send a message. All the calls go through the central switchboard (the microprocessor).

The address wires are connected to several different types of memory and to devices which allow communication between the computer and the outside world. The microprocessor first places the binary representation of the location to be accessed on the address wires. Then after waiting for the computer circuits to select the unique location to which this refers, it either sends or receives a byte of data on the data wires. At the lowest level, this is all that a computer does.

Modern computers usually have several types of memory; early computers had only Random Access Memory (RAM). RAM is essential for any

computer since the fundamental principles of computer operation require the Central Processing Unit (CPU) to repeatedly store and retrieve program instructions, data and memory addresses. The term 'Random Access Memory' means that it may be written to or read from in any order. A severe disadvantage of semiconductor RAM is that it doesn't remember anything after its power is turned off. Some computers have vital portions of their RAM protected by having a battery to provide the power in case of a power line failure.

Read Only Memory (ROM) has data stored in its memory cells at the time of manufacture which it retains permanently. It can be randomly accessed but that access is restricted to the read operation only. A ROM chip can be moved from one place to another without the data being lost as no power is needed to maintain data stored. There are several ROMs in the APPLE computer. One contains the monitor routines which are activated when the computer is turned on. Programs in the monitor initialize the computer and load the Disk Operating System (DOS) and INTEGER BASIC into RAM. The second ROM contains the APPLESOFT BASIC interpreter program which interprets BASIC program instructions. Appendix C contains a description of how the APPLE IIe memory is organized.

3 Thermistor experiments

In the first set of experiments you will make temperature measurements using a thermistor and an ADC. A thermistor is a device whose resistance varies with temperature. The ADC converts an analog voltage (continuous voltage levels) to a digital representation (discrete voltage levels) which can be read by the computer under program control.

3.1 Using the ADC

The ADC 0817 which is installed on the interface card in the APPLE is an eight-bit converter, this means that the range 0–5 V will be divided into $2^8 = 256$ parts. It also is able to select one of 16 input lines on which it will do the conversion. To use the ADC from BASIC is quite easy: an instruction to initiate a voltage conversion is given; a second instruction then reads the result of that conversion. Of the 16 analog input lines (channels) of the ADC, 8 have been brought out onto the prototyping board. Channels 0–7 are on pins 1–8 on the Dual Inline Plug (DIP) connector on the prototyping board. Appendix D has a description of how the ADC works.

The 6502, which is the CPU in the APPLE, uses a system called memory mapping for input and output. To initiate a conversion of the voltage on channel 0 of the ADC you issue the BASIC instruction POKE 49312,0. This instruction would normally mean: store the number 0 ('data') in memory location 49312; however, in this case, the ADC ignores the data, and initiates the conversion of voltage on channel 0. To initiate a conversion in channel 1, the instruction POKE 49313,0 is used; for channel 2, POKE 49314,0 etc. Shortly after issuing the instruction (in considerably less time than the APPLE takes to execute a single BASIC instruction), the result of this conversion can be found in memory location 49312, *regardless* of which channel was converted; thus POKEing and PEEKing access are completely different functions for the ADC. To set a variable, eg, *A*, equal to the converted value in the memory location 49312, use the instruction A = PEEK(49312). PEEK will read the contents of memory location 49312 and will assign an integer in the range 0–255 to the variable *A*. It is good programming practice to use variable names which resemble their meaning. In this case, a better name for the variable *A* is *ADC* or *ADCDATA*. However you must be careful; APPLESOFT BASIC will only look at the first two letters and the suffix (% or $) to distinguish between variables. Thus ADC1 and ADC2 are the same variable.

Exercise 3.1.1 Using the ADC

To get the idea of how to use the ADC connect a 5 kΩ potentiometer across the 5 V power supply on your bench (Figure 3.1) and observe the voltage of the wiper (the center connection on the potentiometer) on the oscilloscope. NOTE: The ground lead of the oscilloscope probe should be connected to the ground of the system, ie, the green wire of the potentiometer. Never connect the oscilloscope probe ground to any point of a circuit which is not ground. Also, it is *extremely* important that the negative output of the power supply be connected to ground of the APPLE at all times.

Before connecting the wiper of the potentiometer to the APPLE observe the voltage of the wiper on the oscilloscope: set the scope trigger control to AUTO so that a continuous trace appears on the screen; be sure the small switch on the probe tip is set to 1x; set the vertical scale to 1.0 V/DIV and the VARIABLE knob to the CALibrated position; set a 0 V baseline by using the ground switch and vertical position knob on the scope.

Turn the knob of the test potentiometer back and forth, the voltage output of the wiper should vary between 0 and 5 V. Now connect the wire from the wiper arm of the potentiometer to the

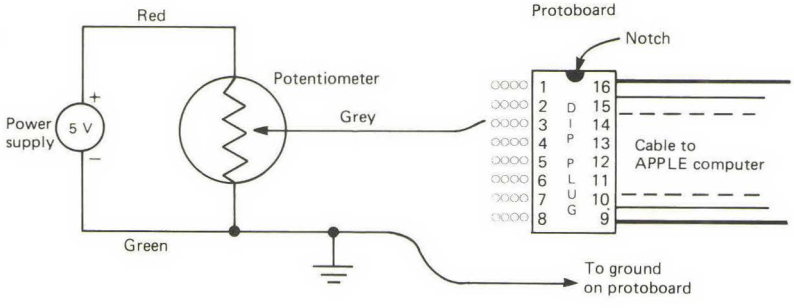
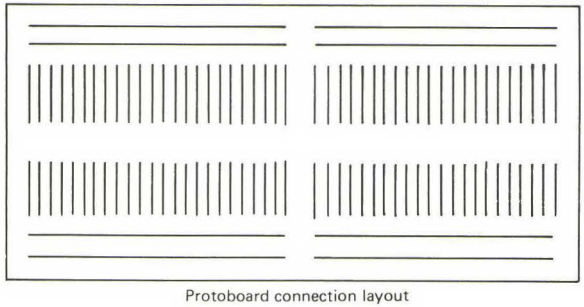


Fig. 3.1. Potentiometer connections and protoboard DIP plug.



Protoboard connection layout

channel 2 ADC analog input on the protoboard; this is pin 3 of the DIP connector. Clear the program currently in the computer (NEW CR) and enter the following BASIC program:

```
 5 REM PROGRAM 3.1.1
10 AD=49312
20 POKE AD+2,0
30 V=PEEK(AD)
40 PRINT V
50 FOR I=0 to 100: NEXT I: REM DELAY
   LOOP
60 GO TO 20
RUN
```

Rotate the potentiometer shaft and note how the voltage on the scope and the computer display changes. The program should print integers in the range 0–255 on the video screen which are proportional to the voltage on the potentiometer. Your particular ADC may not show a count of 255 for 5 V. This is a calibration error which can be corrected for by determining that the range of your ADC is 0–*xxx* rather than 0–255 for a 0–5 V input. The instructions on line 50 in the program are only there to use up time so that the repetition rate between making measurements is sufficiently low to give a readable output on the CRT. To stop the program, simultaneously push the CONTROL key and C (CTL–C). Release them and press RETURN (CR).

The potentiometer is an example of a zero-order instrument; that is, it is a transducer whose output is in direct proportion to its input: $V_{out} = KV_{in}$ where K is the static sensitivity or calibration factor. A perfect zero-order instrument will produce at its output the exact replica of the input signal with only a scale or units change. Of course no instrument or transducer can live up to the perfect response represented by a mathematical formula; all instruments have a range of input values over which tolerable errors occur. It is the responsibility of the designer to determine this range and report the tolerance in the instrument specifications and the responsibility of the user to pay attention to them.

Exercise 3.1.2 Programming the ADC

(*a*) Modify the program and potentiometer connections so that the voltage on channel 5 is read and displayed. Determine what the ADC reading is for the maximum voltage, and what the maximum voltage is.

(*b*) Modify the program and add a second potentiometer so that the program reads the voltage on channel 0 and then, as soon as possible, the voltage on channel 5, the two results should be displayed on the CRT on a single line with a few spaces between the two measurements. Look up the details of the PRINT statement in the reference manuals. Make the program also compute the actual voltage and print them too. The program should make 25 measurements of this kind and then halt. When you get everything running, print out the results on the printer. Also make a printed listing of the program you have written and SAVE the program on your disk.

3.2 ADCs

ADCs come in many sizes and flavors each with a range of usefulness. The following is a description of the most important considerations for choosing and using them.

An ADC has a defined range of input voltages (for example 0–5 V) which it can accurately convert. This range is divided into a number of equal sized pieces (voltages). The integer number output by the ADC corresponds to the number of these which equal the input voltage at the time of the conversion. Figure 3.2 shows how a three binary bit converter converts an input signal to a digital number. Using an ADC is like using a ruler which is graduated in say $\frac{1}{8}''$ markings. All measurements are then made to the nearest $\frac{1}{8}''$. Also it can only measure lengths which are less than the length of the ruler. (You can hop-frog a ruler but you can't do that with an ADC.)

The goal of digital measurements is to get an accurate representation of the input signal. In order to do this the ADC must be able to resolve voltage differences which are significant in the measurement being done. That is, the input voltage range of the ADC must be divided into enough pieces by the digitizer that the voltage change represented by each piece is smaller than the accuracy needed. In the laboratory, the 8-bit converter breaks up the input analog voltage range into $2^8 = 256$ pieces so that the resolution of the converter is $1/256 = 0.4\%$ of the full range or $(5 - 0)/256 = 0.019$ V. Digital

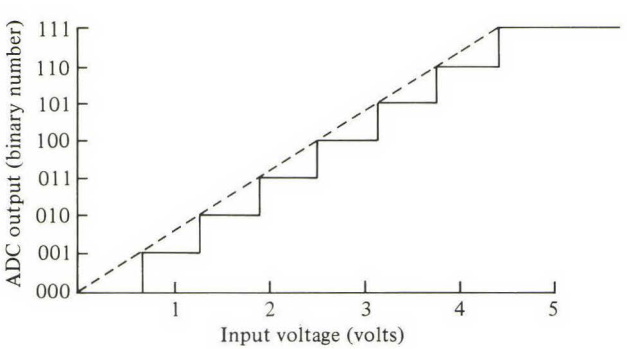


Fig. 3.2. Output of a three bit ADC with input range of 0–5 V. Dashed line is the ideal, solid line the actual response.

audio recording systems use 16-bit converters so that the digitization process is not audible on playback. The ear is a very sensitive detector.

Most of the ADCs on the market are 8, 10, 12, or 16-bit converters. Those above 12-bit require extra care in use since the digitization level is below 1 mV. Extraneous noise from computers or other circuits can creep into the desired signal. Common input voltage ranges are (0 to 5), (−5 to 5) and (−10 to 10). External electronics can be used to shift the voltage from a sensor into the proper range.

Exercise 3.2.1 ADC and sampling

(*a*) The resolution in voltage of an ADC is $\Delta V/2^n$ where ΔV is the total input range and n the number of bits of the digital output. What is the resolution of a 13-bit ADC with input range of +5 to −5 volts? Express your answer in millivolts.

(*b*) Since the amplitude of an analog signal can be adjusted by an amplifier circuit to fill the input range of the ADC, the resolution can be better described by the dynamic range; this is the ratio of the maximum to the minimum voltage measurable by the ADC. The maximum is the ADC input range and the minimum is the resolution calculated above. What is the dynamic range of the 13-bit ADC? The 8-bit used in class? The ratio is usually expressed in decibels (dB), eg, $DR = 20 \log$ (ratio) in dB. Give your answers in both forms; a ratio and in dB.

Another method of converting an analog signal to digital is to input the signal to an electronic circuit (a Voltage Controlled Oscillator or VCO) whose output is a frequency which is proportional to the amplitude of the input voltage, $f_{\text{out}} = f_0 + K_f V_{\text{in}}$. The computer can then measure the frequency of the signal by measuring the time for one cycle of the waveform. To work properly the rate at which the analog input voltage varies must be much less than the frequency output and so the VCO is used for slowly varying signals. The accuracy of this method is also limited.

In order to measure a signal accurately, the rate at which the measurements are taken (the sampling rate) must also be considered. This must be fast enough that all the frequencies contained in the signal can be reproduced. As a quick illustration of the problem, the signal peak in Figure 3.3 will be totally missed if the sampling is done at the time marked with crosses.

As Fourier (1768–1830) showed, any signal can be considered as a superposition of sinusoidal signals of various frequencies. These frequencies generally range from zero (DC) to some maximum frequency, f_{max}, which depends on the physical characteristics of the system generating the signal. The fundamental frequencies of piano range from 27 to 4200 Hz. But the overtones (harmonics) go to much higher frequencies.

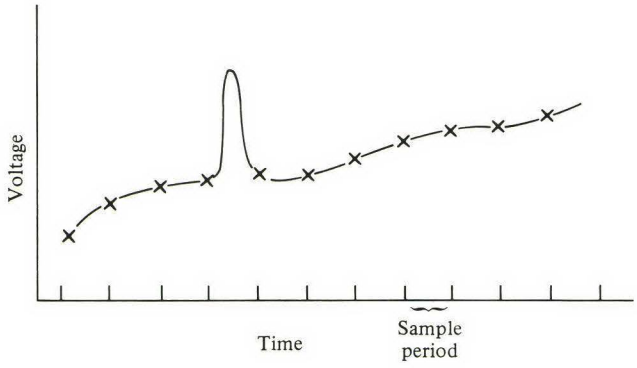


Fig.3.3. Sampling the signal with an ADC at a rate which is too slow. The peak is missed.

In order to accurately reconstruct the original signal from a sampled signal, the ADC sample rate should be at least twice the highest frequency in the input signal, f_{max}. This result is known as the Sampling Theorem and was formulated by Shannon in 1949 building on earlier work by Nyquist (1924). Note that it reads 'the highest frequency in the input signal' not 'the highest frequency of interest'. Even if you are not interested in higher frequencies in the input signal, they must be sampled correctly. If they are sampled at a rate which is less than $2f_{max}$ (the Nyquist frequency), they will masquerade as lower frequency signals (Figure 3.4). This is called aliasing. A good rule of thumb is use a sample rate of at least $2.5f_{max}$. Electronic filters (like the treble and bass controls on a stereo) can be used to limit the frequency range of signals so that the sampling rate can be lowered.

As an example, in the digital recording of music, the audio frequency range of 20–20 000 Hz must be faithfully sampled. Since f_{max} = 20 000 Hz, the Nyquist frequency is 40 000 samples/second. The actual rate used is 48 000 Hz. An ADC which converts the signal in at least 20 μs is needed.

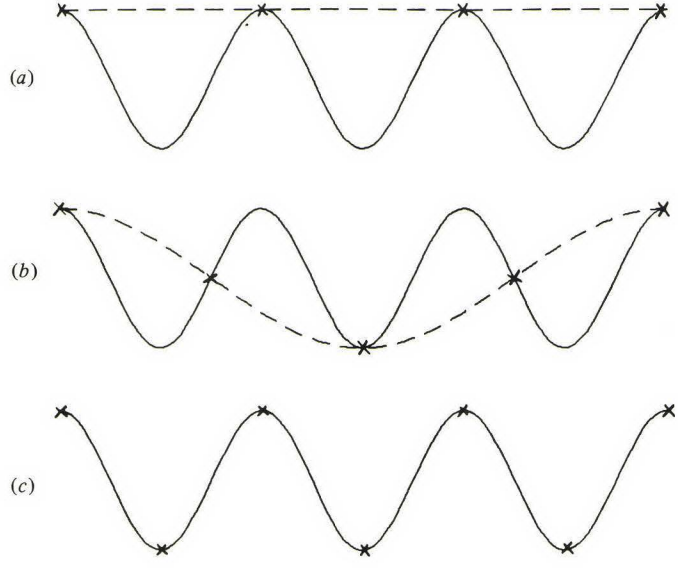


Fig. 3.4. The solid curve is the input waveform. The dashed curve is the waveform reconstructed from the sampled data; (*a*) 1 sample per cycle, (*b*) 1.5 samples per cycle, (*c*) 2 samples per cycle (the Nyquist frequency).

ADCs come in a wide variety of speeds. From low-power devices which convert in milliseconds to fast (less than 10 μs) ones. The first are used in battery operated equipment such as digital multimeters. The fast ones are used in audio and video digital systems.

Exercise 3.2.2 Audio digital sampling

An eight channel digital recording studio wants to faithfully record the audio spectrum from 20 to 20 000 Hz. What must be the sample rate for each channel? The studio wants to use a single multiplexed 16-bit ADC to digitize the signals (the ear is a sensitive detector); what is the maximum conversion time the ADC can have? It is found that ADCs this fast are only available to the military (and at military prices!), but there are some available which are three times slower; how can the system be changed to accommodate slower ADCs and still have the full frequency capability?

There are various ouptut formats for the data coming from ADCs to the computer. This is usually not a large concern when buying them since the computer can convert any format into the one most suitable for its use. Table 3.1 shows some standard output codes for an eight-bit converter with an input range of -10 to 10 V. One LSB (Least Significant Bit) represents $20/256 = 0.078$ V.

Table 3.1 *Comparison of ADC numbering systems*

Input volts	2s complement	Offset binary	Sign bit
+10	0111 1111	1111 1111	1111 1111
+10 − LSB	0111 1110	1111 1110	1111 1110
. . .			
+1 LSB	0000 0001	1000 0001	1000 0001
0	0000 0000	1000 0000	1000 0000
−1 LSB	1111 1111	0111 1111	0000 0001
. . .			
−10 + LSB	1000 0001	0000 0001	0111 1110
−10	1000 0000	0000 0000	0111 1111

3.3 Thermistor resistance vs. temperature characteristics

The first real application to which the ADC will be put is to measure the resistance variation of a thermistor with temperature. The electrical resistance of conductors (metals) increases with increasing temperature. This is a result of the change in the mean free rate between collisions of the free electrons in the conductor with the lattice (stationary ions). As the

system heats the increased amplitude of thermally generated lattice vibrations (phonons) results in increased resistance. Yet for a thermistor, R *decreases* with increasing temperature according to the relation $R = R_0 \exp(T_0/T)$. (You may want to make a quick plot of $y = \exp(1/x)$ to see the rough behaviour of this function.) In this expression, R is resistance (ohms); R_0 is a resistance value corresponding to infinite temperature; T_0 is an activation temperature (K); T is the absolute temperature (K) (0 °C = 273.16 K). The difference is that the thermistor is made from semiconducting materials. In a conductor, every atom donates one or more electrons to the conduction electrons and thus the number of conduction electrons is fixed at a rather large number $\approx 10^{22}/\text{cm}^3$. In a semiconductor the electrons are more tightly bound to the atoms. The energy required to liberate electrons from the atoms is $E_g = k_B T_0$ where k_B = Boltzmann's constant. The probability of an electron being liberated from any given atom by thermal agitation is $p = \exp(-E_g/k_B T) = \exp(-T_0/T)$. Thus the number density of free electrons in a semiconductor varies as $n = n_0 \exp(-T_0/T)$. Note that for $T \ll T_0$, n goes to 0 and the semiconductor becomes an insulator. Since the resistance of a conductor depends inversely on both the number of charge carriers and the mean free path of the carriers the rapid variation of n with T dominates the resistance of a semiconductor over-riding the temperature effect on the mean free path, which can be ignored to a good approximation.

Exercise 3.3.1 Thermistor mathematical models

To show that this last statement is true consider two models of thermistor behavior

$$R_1 = R_0 \exp(T_0/T)$$

and

$$R_2 = AT \exp(T_0/T)$$

where R_0, T_0 and A are constants, R_2 includes the effects of the mean-free-path variation with temperature. Plot $\log(R_1)$ vs. $1/T$ for $T = 3000$ K and $R_0 = 0.02\ \Omega$ in the temperature range 280–400 K. Now plot log R_2 for the same T_0 and adjust A so that $R_2 = R_1$ at 300 K. Show mathematically that R_1 *should* and R_2 *should not* plot as a straight line on this type of plot. Despite this R_2 does appear to be a straight line in this temperature range and so it can be modeled with the equation $R_2' = R_0' \exp(T_0'/T)$. From the graph, find T_0'. Where does the behavior of R_2 start to differ significantly from R_1, at low temperatures or high?

The thermometer/thermistor protoboard, has the circuit diagramed in Figure 3.5. NOTE: when any wiring is done or changed be sure to turn off

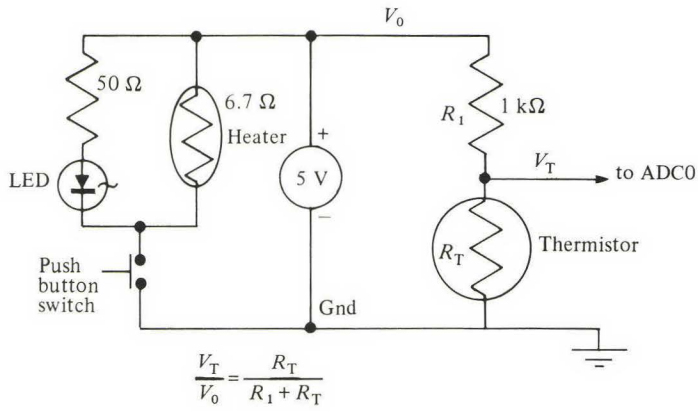
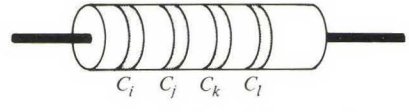


Fig. 3.5. Thermistor and heater circuit.

the power supply. Even though the wiring is very simple it is still a good idea to become accustomed to using wire color codes to help you. Red is used for positive power supply connections, green for ground and the standard electronic color code (Table 3.2) if there is an easy correspondence to data line numbers. Taking the time to do this will make it easier to trace the circuit to find errors when something doesn't function correctly. In addition it is much easier to find test points when a scope or other test instruments are to be used. The voltage across the thermistor should be read into channel 0 of the ADC (ADC0). The push button switch makes it possible to turn the heater on and off manually.

Table 3.2 *The standard electronic color code and resistor identification*

	Color code	
0	Black	
1	Brown	
2	Red	
3	Orange	
4	Yellow	
5	Green	
6	Blue	
7	Violet	
8	Grey	
9	White	
5%	Gold	Tolerance
10%	Silver	Tolerance



$$R = C_i C_j \times 10^{C_k} \pm C_l\%$$

Exercise 3.3.2 Specific heat and power

This exercise is a warm up for Chapter 4 and uses the thermistor circuit but not the computer. The specific heat of a substance is the ratio of the amount of heat added (ΔQ) to the corresponding temperature rise (ΔT) per unit mass (m):

$$C = \Delta Q/m\Delta T \tag{3.3.1}$$

The power P is defined to be the change in heat with a change in time,

$$P = \mathrm{d}Q/\mathrm{d}t \tag{3.3.2}$$

so the amount of heat added to the aluminum block by the heater is the power times the time: $\Delta Q = P\Delta t$. Where the power is the voltage drop v times the current i, $P = iv$ or by using Ohm's law $P = v^2/R$ for a resistance R.

(*a*) While pushing the button, determine how rapidly the temperature rises (degrees/second). By estimating the mass of the aluminum block, *estimate* its specific heat. In the *CRC Handbook of Chemistry and Physics* it lists the specific heat of aluminum as 0.215 cal/g °C and the specific gravity as 2.702 g/cm^3. Convert the specific heat to SI (kg-m-s) units and compare with your rough results. Where does error enter this estimate?

(*b*) Also measure the rate at which it cools and calculate the heat lost per unit time (the power output) due to conduction and convection. Is this result significant for the measurement made in part (*a*)?

(*c*) When you release the button, why doesn't the temperature stop rising immediately?

The thermometer used to measure the temperature of the block has thus far been considered a zero-order transducer (like the potentiometer in Section 3.1). In reality it takes a finite amount of time for the mercury in the glass bulb to heat up in response to the increase in temperature of the block. It is thus a first-order instrument whose response is determined by the differential equation: $\tau(\mathrm{d}T_{\mathrm{out}}/\mathrm{d}t) + T_{\mathrm{out}} = KT_{\mathrm{in}}$ where T_{out} is the change in the reading on the thermometer (output), T_{in} is the change in the block temperature (input), τ is the characteristic response time, and K is the static sensitivity or calibration factor. The solution of this equation for the case of a sudden change in temperature of the block is $T_{\mathrm{out}} = KT_{\mathrm{in}}[1 - \exp(-t/\tau)]$ whose graph is shown in Figure 3.6. This shows that if changes happen quickly enough, the thermometer response does not keep up and the readings will be in error. Notice that if t = τ, the temperature has risen to $(1 - \exp(-1))$or about $\frac{2}{3}$ of the step input change (see Figure 3.6). This provides a quick way to estimate τ

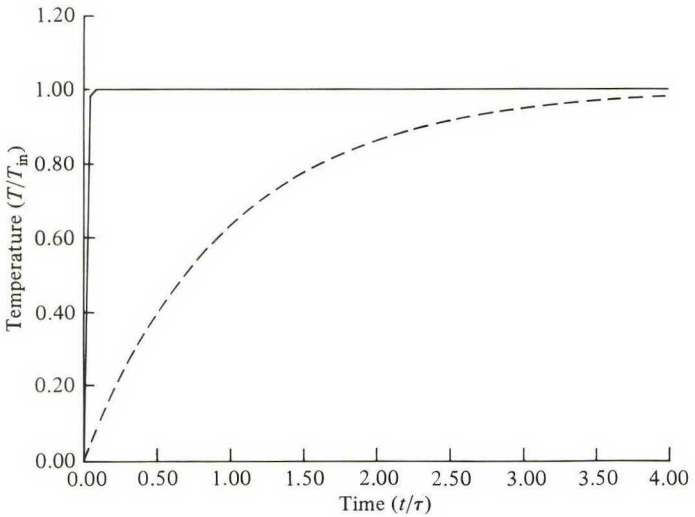


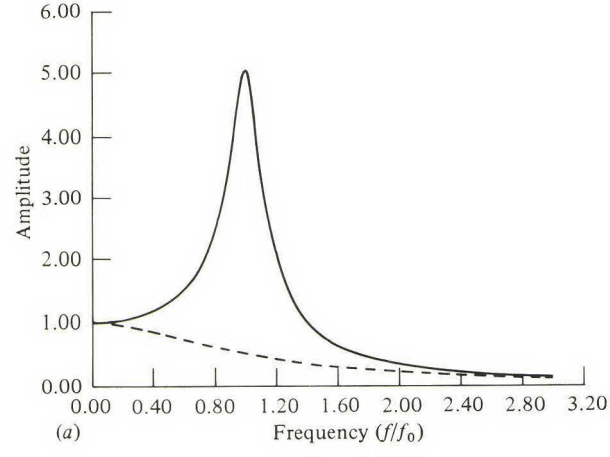
Fig. 3.6. Graph of the time response (dashed line) of a thermometer (first-order transducer) to a step temperature change (solid line) in the surroundings.

Higher-order instrument response characteristics are also common in instrumentation systems. For example, a damped spring used for weighing objects or as an accelerometer is usually modeled by a second-order differential equation. Three parameters are then needed to predict the response to a particular input: the calibration K, the damping constant δ, and the resonant frequency f_0. Each frequency of the input signal is affected in a different way as it passes through the system. In systems of order two or higher a graph of the gain or calibration factor as a function of frequency is useful for designing instruments using a particular transducer. The frequency response characteristics of cassette tape or a stereo phonograph are many times displayed in their advertising literature. The frequency and phase vs. frequency for a second-order transducer is shown in Figures 3.7(*a*) and (*b*). Note that at the resonant frequency a large response can occur if the damping is weak.

In the laboratory, both the thermometer and the thermistor are first-order transducers and so have finite time reponses to a change in temperature. However, the time constant of the thermometer is much larger and so it dominates the response of the system. The time constant of the thermometer can be estimated to be about 1.5 s by watching the temperature reach equilibrium after the power input is stopped (the button is released). In the following, the purpose is to estimate the temperature lag of the thermometer behind the block temperature for a constant power input (ie when the button is kept pushed down, how far behind the actual temperature is the measured temperature?)

As before, the differential equation of the response of a first-order temperature transducer is

$$\tau \, dT_o/dt + T_o = T_i$$



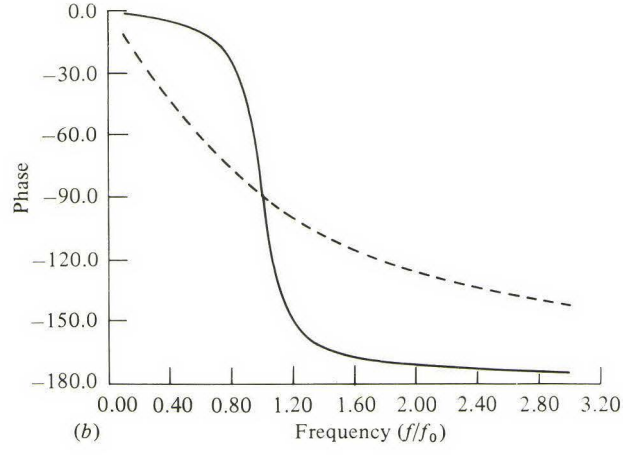


Fig. 3.7. Frequency response of a second order transducer for the damping constant $\delta = 0.1$ (solid line) and for the damping constant $\delta = 1.0$ (dashed line). (*a*) Amplitude vs. frequency: (*b*) phase vs. frequency.

where T_o is the temperature measured minus the initial temperature of the system and T_i is the temperature change input to the system. The power input to the block is $P = \Delta V^2/R$ with R the heater resistance. The power is also the amount of heat energy per unit time which goes into the block $P = \mathrm{d}Q/\mathrm{d}t$. Since the heat capacity of the block is $C_V = \mathrm{d}Q/V\,\mathrm{d}T$, then $C_V = P\mathrm{d}t/V\,\mathrm{d}T$ or the change in temperature of the block with time is $\mathrm{d}T/\mathrm{d}t = p/C_V V$. Since $\mathrm{d}T/\mathrm{d}t = \mathrm{d}T_o/\mathrm{d}t$, substituting into the differential equation above gives

$$T_o - T_i = \tau P/C_V V$$

Exercise 3.3.3 Temperature lag

For the experimental apparatus used in the laboratory, estimate the lag of the thermometer temperature behind the block temperature

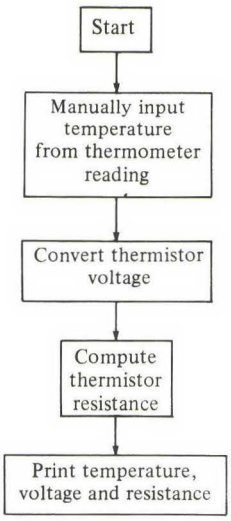


Fig. 3.8. Flow chart for Exercise 3.3.4.

for a constant power input. Assume $\tau = 1.5$ s for the thermometer and estimate the block volume. Do the same estimate for the thermistor using $\tau = 0.4$ s for the time constant. What is the difference between the thermometer lag and the thermistor lag?

Exercise 3.3.4 Thermistor resistance measurement

Write a program which will allow you to enter manually a temperature reading you observe on the thermometer using an INPUT statement and which will then read the thermistor voltage by using the ADC. Check the voltage readings printed out on the CRT screen with those which you get with the oscilloscope. Make a printout of the program when it works. Once you get this working write a few additional statements so that the actual resistance of the thermistor is computed and printed. Follow the flow diagram in Figure 3.8. This can be calculated from the voltage divider relationship shown on Figure 3.5:

$$\frac{V_T}{V_0} = \frac{R_T}{R_1 + R_T} \tag{3.3.3}$$

Make a quick check of the computer code by doing a calculation by hand.

Exercise 3.3.5 Data arrays

Modify the program in Exercise 3.3.4 so that the computer makes a series of measurements and stores them in arrays, ie, $T(I)$, $R(I)$. These symbols mean that measurement number I had a thermometer reading of $T(I)$ and a resistance measurement of $R(I)$. Print the whole array after the last measurement is made. To get out of the input loop use some absurd value of the temperature T (say 0 or 1000) as a flag that no more input is desired. (Make sure that this last value is not included in the data.) Shortly, you will add additional steps so that these data can be stored on the disk as a data file.

3.4 Making and retrieving sequential data files

A subroutine which will produce a data file of temperature and resistance data ($T(I)$ and $R(I)$) is given below. Please read the sections in the reference manuals on subroutines and on data files to supplement the discussion.

Exercise 3.4.1 WRITE data file

Enter the following program and save it on your disk.

```
1000 REM  WRITE DATA FILE
1005 REM PROGRAM 2.3.1
```

```
1010 PRINT CHR$(4) "OPEN"F$
1020 PRINT CHR$(4) "WRITE"F$
1030 PRINT N    : REM  NUMBER OF DATA
     POINTS
1040 FOR I = 1 TO N
1050 PRINT T(I)
1060 PRINT R(I)
1070 NEXT I
1080 PRINT CHR$(4) "CLOSE"F$
1090 RETURN
```

CHR$(4) is a control character which the disk operating system of the APPLE will interpret to mean that disk instructions follow. The instruction "OPEN"F$ means open the file whose title is stored in the variable F$. The print statement following (line 1020) will also be interpreted as an instruction by the DOS because of the CHR$(4); it says that subsequent print statements are to be interpreted as writing data to the disk.

Each PRINT statement following these two will record a piece of data, eg, one number; multiple pieces of data cannot be incorporated in a single PRINT instruction (hence the two lines 1050 and 1060). Since the data will be recorded sequentially, it is the programmer's responsibility to know the order and amount of the data recorded. In this case the first data value put in the file (line 1030) is the number of pairs (T, R) to follow. The program reading this data file can then use this value so that it doesn't read past the end of the file and stop with an error message (OUT OF DATA).

The final PRINT statement (line 1080) has a CHR$(4) so that it will be interpreted as containing a DOS instruction to CLOSE the file F$. It is vital that every OPEN instruction have a CLOSE instruction. The CLOSE instruction will also have the effect of restoring the PRINT instruction so that it will send data to the screen and/or the printer.

In APPLESOFT BASIC, the computer can only send or receive data from one source at a time. While it is set up to write to or read from the disk do not try to input data from the keyboard or output to the screen. Always make the disk access a separate part of your program.

Exercise 3.4.2 Test data WRITE

Test the subroutine with the following program:

```
 10 REM RECORD TEMP/RES DATA
 15 REM PROGRAM 3.4.2
 20 INPUT "OUTPUT FILENAME: ";F$
 30 REM CREATE SOME DUMMY DATA
 40 DIM T(100), R(100)
```

```
 50 N=20
 60 FOR I = 1 TO N
 70 T(I) = I
 80 R(I) = I*I
 90 NEXT I
100 GOSUB 1000
110 END
```

Remember to merge this program with the subroutine (Program 3.4.1) before you RUN it. After running the program, type CATALOG CR and look for the filename you entered.

The critical (and useful!) part of making files is being able to read them back. The following subroutine will do this for this data file:

```
2000 REM READ DATA FILE
2005 REM PROGRAM 3.4.3
2010 PRINT CHR$(4) "OPEN"F$
2020 PRINT CHR$(4) "READ"F$
2030 INPUT N    : REM GET NUMBER OF DATA
2040 FOR I = 1 TO N
2050 INPUT T(I)
2060 INPUT R(I)
2070 NEXT I
2080 PRINT CHR$(4) "CLOSE"F$
2090 RETURN
```

As you can see, its form is quite similar to the previous WRITE subroutine.

Exercise 3.4.3 READ data file

A program which uses this subroutine follows; use it to test the READ subroutine.

```
10 REM OBTAIN TEMP/RES DATA FROM DISK
15 REM PROGRAM 2.3.4
20 INPUT "INPUT FILENAME: "; F$
30 DIM T(1000), R(100)
40 GOSUB 2000
50 REM MAYBE PRINT TO THE SCREEN HERE
60 END
```

Exercise 3.4.4 Temperature and thermistor resistance data file

Modify your thermistor program (Exercise 3.3.5) so that temperature and thermistor resistance arrays are recorded on a diskette. Do

enough tests to be confident that the program you have written generates a disk file and that you are able to read it back.

Using the manual on/off switch on the heater, make a series of measurements of temperature and resistance (about 10–15) of the heater block between room temperature and about 100 °C. Record them as a disk file. Read the data back and print them so that you know you have them.

3.5 Plotting the experimental data

Exercise 3.5.1 Thermistor data plot

Using your knowledge of the AMPERGRAPH instructions and the program examples given in the AMPERGRAPH documentation, write a program to get the data from the disk file and plot as open circle data points the thermistor resistance on the ordinate (y) and the temperature on the abscissa (x). Use degrees Kelvin. Be sure to read about the problems in using RENUMBER and AMPERGRAPH in Appendix D.

The value of graphical plots is that they are capable of displaying and conveying much information very quickly. One obvious weakness of the linearly scaled display of the resistance vs. temperature plot which you have made is that it is difficult to get a good display of the lower values of resistance. When the numerical value of a parameter to be plotted spans a large range, scaling the axis logarithmically is very useful. On a linearly scaled axis each increment of length is proportional to an increment of the parameter being plotted. On a logarithmically scaled axis each increment of length is proportional to the fractional change in the value of the parameter. (If $y = \log(R)$, then $dy = dR/R$.) Often it is more significant to note the fractional change in a parameter than the change in the value of the parameter itself. For example, when plotting stock exchange prices and their change in time, it is much more useful to plot the stock prices on a logarithmic ordinate scale than a linear one.

Exercise 3.5.2 Logarithmic plot

Modify your plot to use an appropriate logarithmic scale on the ordinate. To use a logarithmic scale you must use the &LOGY instruction before the &LABELAXES instruction. Details are given in the AMPERGRAPH documentation. Note that using a logarithmic scale is different than plotting logarithmic values on a linear scale.

This logarithmic plot is a very useful one to display the resistance of a thermistor vs. temperature for purposes of manually detemining the resistance for a given temperature. However, for comparison with mathematical theory it is better to use a different plot. The form of the plot is determined by the particular phenomena being studied. As shown in Section 3.3, the variation of the resistance of a thermistor can be written as:

$$R = R_0 \exp(T_0/T) \tag{3.5.1}$$

where R_0 (Ω) and T_0 (K) are constants. To display graphically the extent to which the measured dependence conforms to this theoretical dependence, it is useful to plot the resistance vs. temperature using a scale such that the resulting plot becomes a straight line. This is easily done by taking the natural logarithm of the resistance for the 'linear' ordinate length and $1/T$ for the 'linear' abscissa length scale. Taking the logarithm of Equation (3.5.1) gives

$$\ln(R) = \ln(R_0) + (T_0/T) \tag{3.5.2}$$

and by setting

$$y = \ln(R) \qquad A = T_0 \qquad x = 1/T \qquad B = \ln(R_0) \tag{3.5.3}$$

Equation (3.5.2) becomes

$$y = Ax + b \tag{3.5.4}$$

which is a straight line. (You'll notice, on close inspection, that the previous plot in Exercise 3.5.2 is not a straight line.)

Exercise 3.5.3 Linearized thermistor data plot

Make a plot of your resistance vs. temperature measurements using a logarithmic scale for R (as in Exercise 3.5.1) and $1/T$ for the abscissa. T is the absolute temperature in degrees Kelvin. Check to see if your data conforms to the model equation (Equation 3.5.1).

3.6 A least squares fit to the data

Finding good values for the parameters R_0 and T_0 in Equation (3.5.1) are important in their own right for investigating the physics of the device; having good values for them is also important for making the temperature controller which you will be shortly called upon to do. By finding values for A and B from the linear plot of Exercise 3.5.3, values for R_0 and T_0 can be easily calculated via Equation (3.5.3). This can be done graphically or by a least squares fit of the data.

In doing the experiment, you have acquired data at a sequence of values of temperature T_i or alternatively $X_i = 1/T_i$. Each of these temperatures T_i yielded an experimental resistance value R_i or alternatively $Y_i^{\text{ex}} = \ln(R_i)$. Equation (3.5.1) yields a theoretical resistance value R_i^{th} for each temperature, ie, for each X_i a theoretical value $Y_i^{\text{th}} = \ln(R_i^{\text{th}})$ is given. The task is to find values for A and for B to minimize the error between the experimental

and theoretical values, $E_i = Y_i^{\text{th}} - Y_i^{\text{ex}}$. A common type of analysis minimizes the sum of the squares of the individual errors. Calling the total square of the error E_T, we get

$$\left.\begin{aligned} E_T &= \sum_i E_i^2 \\ E_T &= \sum_i (Y_i^{\text{th}} - Y_i^{\text{ex}})^2 = \sum_i (AX_i + B - Y_i^{\text{ex}})^2 \end{aligned}\right\} \quad (3.6.1)$$

To minimize this error with respect to the parameters A and B we take derivatives with respect to A and B and set them to zero:

$$\left.\begin{aligned} \partial E_T/\partial A = 0 &= \sum_i 2X_i(AX_i + B - Y_i^{\text{ex}}) \\ \partial E_T/\partial B = 0 &= \sum_i 2(AX_i + B - Y_i^{\text{ex}}) \end{aligned}\right\} \quad (3.6.2)$$

Taking A and B out of the summations and collecting terms gives

$$\left.\begin{aligned} AS_{XX} + BS_X &= S_{XY} \\ AS_X + BS &= S_Y \end{aligned}\right\} \quad (3.6.3)$$

where

$$\left.\begin{aligned} S_{XX} &= \sum_i X_i^2 \qquad S_Y = \sum_i Y_i^{\text{ex}} \\ S_X &= \sum_i X_i \qquad S = \sum_i 1 \\ S_{XY} &= \sum_i X_i Y_i^{\text{ex}} \end{aligned}\right\} \quad (3.6.4)$$

Then solving for A and B

$$\left.\begin{aligned} D &= SS_{XX} - S_X^2 \\ A &= \frac{SS_{XY} - S_X S_Y}{D} \\ B &= \frac{S_{XX}S_Y - S_{XY}S_X}{D} \end{aligned}\right\} \quad (3.6.5)$$

Exercise 3.6.1 Least squares fit to data

Write a program to find values for A and B using a least squares fit and plot this theoretical fit as a line together with your experimental values for temperature and resistance as open circles. Use a logarithmic scale for R and $1/T$ for the x scale. Also obtain the corresponding values for T_0 and R_0.

The least squares fit assumes that the measured data will be randomly scattered about the theoretical fit. The plot in Exercise 3.6.1 does not show this clearly. A quick visual test of this assumption is to make a plot of the difference between the data and the fit ie, plot the errors E_i. These are the residuals.

Exercise 3.6.2 Plot of residuals

Make a plot of the difference between the measured data and the theoretical fit for the data of Exercise 3.6.1. By inspection determine if the assumption of random errors is satisfied.

3.7 Data modeling

The purpose of data modeling is to obtain a mathematical model which represents a set of experimental data. First a model is chosen either on the basis of a theory of the physical process or by guessing the mathematical form which approximates the data. The model will have some parameters which can be adjusted to give a best fit. For example, the model may be a straight line $y = mx + b$ with the slope m and y intercept b as parameters. These can be varied so that the line fits a set of data points.

Many times a model can be fitted to data to sufficient accuracy by hand plotting. The best fit is then subjective to some degree. More accurate determinations of model parameters can be obtained mathematically and computationally. The first step is to form a mathematical estimate of how well the model fits the data. One common measure of the total error in the fit is the sum of the squares of the difference between the y value predicted by the model, y_i^{model} and the y data value, y_i^{data}

$$\text{Total error} = e_2 = \sum_{i=1}^{N} (y_i^{\text{model}} - y_i^{\text{data}})^2 \tag{3.7.1}$$

where N is the number of data points. The difference is squared so that it is always positive. A negative error (point above the curve) adds as much to the total error as an equal positive error. Another measure of the error which is sometimes used is the sum of the absolute values of the difference:

$$e_1 = \sum_{i=1}^{N} |y_i^{\text{model}} - y_i^{\text{data}}| \tag{3.7.2}$$

The total error can be calculated for a set of model parameters. The best fit will be that set which leads to the smallest total error. A brute force way to find the smallest error is to calculate the total error for a wide variety of parameters. The search can be narrowed to smaller parameter variations as the minimum is approached.

This method is sometimes the only possible way to proceed. However for many models, the minimum error can be found by mathematically rather than computationally varying the parameters. Since the model is a function of the parameters $y_i^{\text{model}} = f(p_1, \ldots, p_q; x_i)$ so is the error $e = f(p_1, \ldots, p_q; x_i, y_i^{\text{data}})$. The minimum of a function of a variable is found by solving the equation given by differentiating the function and setting the result equal to zero. In this case the minimum with respect to changes in the parameters is wanted so q equations are formed:

$$\partial e/\partial p_1 = 0;\ \partial e/\partial p_2 = 0;\ \ldots;\ \partial e/\partial p_q = 0$$

These can then be solved simultaneously for the parameters $p_1, p_2, \ldots, p_q$ that give the minimum. (Here it is assumed there is only one minimum and no maximum as is generally the case for physically real models.)

For example: consider the case of the simplest one parameter model, $y = p_1$; that this, the data can be represented by a constant. The total error is

$$e_2 = \sum_{i=1}^{N} (p_1 - y_i)$$

With its derivative

$$\partial e_2/\partial p_1 = \sum_{i=1}^{N} 2(p_1 - y_i) = 0$$

So

$$p_1 = \Sigma\, y_i/\Sigma\, 1 = \Sigma\, y_i/N$$

where the sums go from 1 to N. This is just the average value (mean) of the Y data.

As a second example: consider the case where the data is to be fit to a straight line $y = p_1 + p_2 x$ where p_2 is the slope and p_1 the y intercept. Then

$$e_2 = \sum (p_1 x_i + p_2 - y_i)^2$$
$$\partial e_2/\partial p_1 = p_1 \sum x_i^2 + p_2 \sum x_i - \sum x_i y_i = 0$$
$$\partial e_2/\partial p_2 = p_1 \sum x_i + p_2 \sum 1 - \sum y_i = 0$$

These equations can be solved directly or by forming a matrix representation

$$\begin{pmatrix} \Sigma\, x_i^2 & \Sigma\, x_i \\ \Sigma\, x_i & \Sigma\, 1 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} = \begin{pmatrix} \Sigma\, x_i y_i \\ \Sigma\, y_i \end{pmatrix}$$

and using Cramer's rule from linear algebra to obtain the solution

$$D = N \sum x_i^2 - \left(\sum x_i\right)^2$$
$$p_1 = \left(N \sum x_i y_i - \sum x_i \sum y_i\right)/D$$
$$p_2 = \left(\sum x_i^2 \sum \mathrm{y_i} - \sum x_i \sum x_i y_i\right)/D$$

which are equivalent to Equations (3.6.5). A three parameter polynomial fit $y = p_1 + p_2 x + p_3 x^2$ (parabolic) can be treated in the same way.

As a final example: consider again a one parameter fit to the data but this time use the absolute value total error

$$e_1 = \sum_{i=1}^{N} |p_1 - y_i|$$

Then

$$\partial e_1/\partial p_1 = \sum_{i=1}^{N} \mathrm{sgn}(p_1 - y_i) = 0$$

where

$$\mathrm{sgn}(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

This means that p_1 is adjusted to balance the number of y values which are greater than p_1 with the number less than p_1 (the number of $+1$s and -1s must be equal to make the sum zero). Thus $p_1 = \text{median}(y_i \ldots y_N)$. The median can be calculated by sorting the y_i; then, $p_1 = y_1/2N$. In some situations the median is a better average than the mean value. If an experiment took two days to produce one number and after six days these numbers came out to be 42, 33 and 377, would you believe the mean value of 151 or the median of 42?

The model equations used in calculating the total error need not be as simple as those considered so far. An example would be the resistance variation with temperature of a thermistor $R = R_0 \exp(T_0/T)$. In this case by taking the logarithm and a change of variables, it can be expressed as a linear model:

$$y = ax + b$$

where

$$y = \ln(R), \qquad a = T_0, \qquad x = 1/T, \qquad b = \ln(R_0)$$

However, the model equation may not linearize. For example the expression for the heat flow in a rod $T = T_1(t_1/t)^{1/2} \exp(t_1/t)$ has this characteristic. You must start from the error expression, differentiate and solve the equations.

For some expressions not even this is possible; the trial and error method can be used. However, by using the computational speed of the computer, there is a more elegant way of searching for the best parameters. The Simplex algorithm is an iterative procedure which systematically explores the parameter values of the model. It has the virtue that any computable function can be used as a model and that no derivatives are needed. Another method commonly used is the Levenberg–Marquardt method which requires the use of error function derivatives. A good description of these algorithms can be found in the References.

3.8 Errors in data and parameters

In fitting data to a theoretical model in the least squares method used in Section 3.6, the implicit assumption has been made that each data point has been measured with the same reliability. This is often not the case and it is then important to include a measure of the data reliability when fitting a model to these data. Another result of frequent interest which is not obtainable by the simple least squares fit is to determine how much the fitted parameters can vary without straining the fit to the data (how good is the fit?).

To make a statement of how good a measurement is we usually quote the value measured together with an expected error; for example a voltage is $V \pm \Delta V$ volts. An accepted definition of ΔV is that it is the root mean square (rms) value of the random error inherent in the measurement.

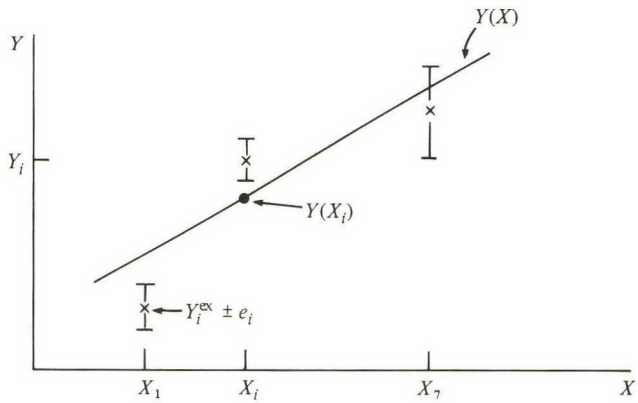


Fig. 3.9. Plot of experimental data together with a theoretical fit and errors bars inherent to each data point: Y_i^{ex} – experimental data points, e_i – error in experimental data points, $Y(X)$ – proposed theoretical fit.

Consider the plot of experimental data and of a proposed theoretical fit indicated in Figure 3.9. It shows the results of a series of measurements which yield the values $Y_i \pm e_i$ at a series of parameter values X_i. Assume the X_i are well determined. The true variation of $Y(X)$ is given as some function of X. For sake of discussion assume that Y is of the form $Y(X) = AX + B$ where the parameters A and B are to be determined.

The total error can now be written as

$$E_T = \sum_i \left[\frac{(AX_i - B) - Y_i^{ex}}{e_i} \right]^2 \tag{3.8.1}$$

where e_i is the error in the data point Y_i. A small error e_i at data point Y_i will cause the difference between the model and the data point to be weighted heavily in the sum. Thus the points with small errors have a stronger affect on the fit. Proceeding as in Section 3.6 gives the same formula for A and B (Equation 3.6.5) except that now

$$\left.\begin{aligned} S_{XX} &= \sum_i X_i^2/e_i^2 \\ S_X &= \sum_i X_i/e_i^2 \\ S_{XY} &= \sum_i X_i Y_i^{ex}/e_i^2 \\ S_Y &= \sum_i Y_i^{ex}/e_i^2 \\ S &= \sum_i 1/e_i^2 \end{aligned}\right\} \tag{3.8.2}$$

By means of error propagation analysis, the errors in the estimates of A and B are determined to be

$$\left.\begin{aligned} e_A^2 &= S/D \\ e_B^2 &= S_{XX}/D \end{aligned}\right\} \tag{3.8.3}$$

Where $D = SS_{XX} - S_X^2$ as before (Equation 3.6.5). The goodness of fit of the data to the model can also be calculated:

$$G = 1 - P\left(\frac{N-2}{2}, \frac{E_T}{2}\right) \tag{3.8.4}$$

where $P(a, x)$ is the incomplete gamma function which is tabulated in most statistics books. If G is greater than 0.1, the fit is good; if less than 0.001 then your model does not fit the data very well. Please see Press *et al. Numerical Recipes* for further information.

Keep in mind that the estimation of the parameters $A \pm e_A$ and $B \pm e_B$ by the least squares method is a statistical one. That is, given the data and the model function, the calculated parameters A and B are the most likely ones for the system. The method assumes that the errors made in the measurements are random. It does not consider any systematic errors which may be lurking in your data. These last need to be ferretted out by careful thought and experimentation.

Exercise 3.8.1 Errors in thermistor data

Make an evaluation of the error in your resistance determinations with the ADC and reanalyze the thermistor data with error considerations. To simplify the error analysis, assume some reasonable constant error ($\Delta R_i = \Delta R$ for all i) and simplify the error equations by factoring the error out of the sums.

3.9 Digital signal processing

Proper use of the ADC requires analog signal conditioning before the ADC samples the data as described in Section 3.2. Once in the computer, a series of samples can then be analyzed to emphasize various features of the data.

If the data has some noise mixed in with a broad trend, a smoothing process can be used to suppress the noise. One common method is to apply an averaging scheme as shown in Figure 3.10. The new point z_i is a weighted average of the old point y_i with its neighbors. Specifically,

$$z_i = \tfrac{1}{4}(y_{i-1} + 2y_i + y_{i+1}) \tag{3.9.1}$$

Equation (3.9.1) can be extended to more points if more smoothing is required.

This way of smoothing data is one example of a digital low-pass filter; it suppresses the high frequency components of the time series. Another way of making a digital filter is by the recursive procedure:

$$z_i = (1 - \alpha)y_i + \alpha z_{i-1} \tag{3.9.2}$$

When applied to a time series, Equation (3.9.2) approximates a low-pass analog resistor–capacitor filter with the parameter α setting the frequency cut-off. Again, more smoothing can be done by including more terms in the recursion. Both recursive and non-recursive filters can be constructed which will act as high-pass filters if the interesting part of the signal is not the trend but the time varying part.

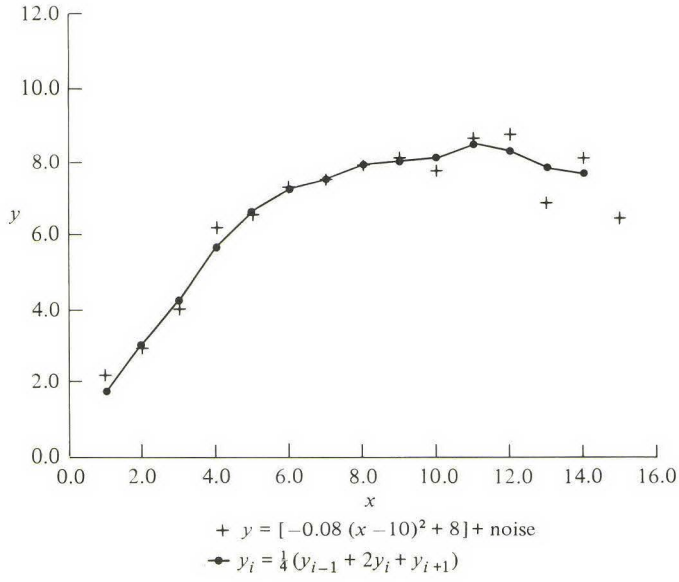


Fig. 3.10. Data smoothing example.

A reminder: digital filtering in no way replaces analog filtering before sampling the signal. Aliasing occurs when the data is sampled and cannot be remedied later.

Much more elaborate signal processing is often required to analyze a set of data. The references contain further information.

3.10 Generation of output using BASIC

The next task you will work on is to use the thermistor in a temperature controller. The computer will not only be measuring the temperature of the block but will turn the heater on and off to maintain the predetermined temperature. To get a feeling of how to send output from the computer you will use a program written in BASIC to generate square waves at an output port of the 6522 VIA interface.

Exercise 3.10.1 Square wave output

Connect the oscilloscope probe of CH 1 to the terminal marked PB0 of port B of the APPLE interface board. Set the switch on the scope probe to 1x, the oscilloscope triggering to AUTO and the CH 1 amplifier to 2.0 VOLTS/DIV. The sweep time should be set to move at a rate of 1 DIV every 2 ms (2×10^{-3} s). Also set the MODE switch to CH 1.

Type in the following program; the program and comments are explained after the program.

```
 5 REM EX 3.10.1
10 POKE 50178,1   Send 1 to DDRB, sets up DRB for output on PBO.
20 POKE 50176,1   Send 1 to DRB, makes PB0 go HI.
30 POKE 50176,0   Send 0 to DRB, makes PB0 go LO.
40 GO TO 20       Loop back to instruction 20.
```

Run the program; on the oscilloscope you should see two parallel lines, which are really square waves. To see them more clearly push the TRIGGERING LEVEL switch in on the oscilloscope and adjust the triggering level to get a steady picture. Measure and record the time the output is HI (ie, +5 V) and the time which it is LO (0 V). Be sure that the VARIABLE potentiometer knob on the SWEEP TIME/CM control (the red one) is in the CAL(ibrated) position.

Line 10 of the program instructs the machine to store the data value 1 in memory location 50178; line 20 to store 1 in memory location 50176, etc. The 6502 CPU uses a memory-mapped system of I/O. This means that certain memory addresses may not really be memory locations but may be connected to the outside world. In this case, memory address 50176 is Port B to which the oscilloscope is connected. The ADC registers at locations 49312–49320, which you used before, are another example.

Inside the APPLE a circuit board has been inserted on which is mounted a 6522 IC which is referred to as a VIA (Versatile Interface Adapter). This IC controls two output ports sometimes referred to as Port A and Port B. Port A is memory location 50177, Port B is 50176. The 6522 controlling Port A and Port B can do many things (therefore the name 'versatile'). For example, the wires from Port B can be programmed to be used either as inputs or as outputs. Line 10 of the program stores a 1 in location 50178. This sets up line PB0 as an output line; storing a 0 in this location would have set up PB0 as an input line. Location 50178 is referred to as DDRB (Data Direction Register B). It is a memory location which controls the direction of data flow of Port B. Port B is sometimes referred to as DRB (Data Register B).

The 6502 is an eight-bit CPU. This means that each operation in the CPU is performed eight bits at a time. When memory is addressed, a byte of data (eight bits) is taken from or written to memory by the CPU. Each memory location is eight bits wide. One manifestation of this is that there are eight lines coming out of Port B labeled PB7 ... PB0. A numerical value is ascribed to each line, PB7 is 128, PB6 is 64, PB5 is 32, etc. Line PB0 has a value of 1. Each of the lines PB0–PB7 has a direct correspondence to one of the data lines of the 6502 CPU.

Each line of Port B is individually programmable for either input or output. Storing 128 in DDRB (location 50178) will program line PB7 for

output and leave all the rest as input lines, storing 80 = 64 + 16 in DDRB will set up PB6 and PB4 as output lines and leave the rest as input lines. The binary (base 2) representation of 80 is 0101 0000 which shows the easy correspondence of the binary representation with the I/O lines. Thus, as stated before, sending a 1 to location 50178 in line 10 of the program sets up PB0 to be an output line and all the rest input lines. The program then proceeds to send 1s and 0s alternately to DRB, ie, location 50176, to generate the square waves which you see on the oscilloscope. If PB7 were an output line, sending 128s and 0s to DRB would generate a square wave on PB7. In the computer a 1 is represented by approximately +5 V on a wire, 0 by approximately 0 V on a wire. To stop the program, which is trapped to run forever; press CTRL-C.

Exercise 3.10.2 Square wave output on PB3

Rewrite program 3.10.1 to generate square waves on line PB3. When you get the program running satisfactorily, stop, print it out and save it on disk.

3.11 POKE and PEEK

POKE and PEEK are conjugate instructions when used with ordinary RAM. The instruction POKE X, Y means store the number Y in memory location X. Conversely Y=PEEK(X) will read the number stored at address X and assign it to variable Y. Since the 6502 is an eight-bit processor (eight data lines) the number Y will range from 0 to 255 ($256 = 2^8$). The 6502 has 16 address lines and so is capable of directly addressing 65536 memory addresses ($65536 = 2^{16}$). Thus X in the POKE statement ranges from 0 to 65535. Memory address 36864 is a RAM location in which eight bits of data can be stored and retrieved without disturbing programs in the computer. The instruction POKE 36864,45 will store the number 45 in address 36864. Type in this instruction in the immediate mode (no line number). The PEEK (X) instruction will read the number stored in memory address X. The number returned will be between 0 and 255. To demonstrate this, enter the immediate instruction PRINT PEEK (36864). The value 45 should be returned if it was preceded by POKE 36864,45. Experiment with other combinations.

3.12 Using a HEXFET to control the heater

The digital signals coming out of the APPLE are feeble and in general cannot drive external circuitry loads directly. HEXFETs are one variety of enhanced mode power FETs (Field Effect Transistors) which are particularly suited for controlling large amounts of power by using the digital control signals coming out of a computer.

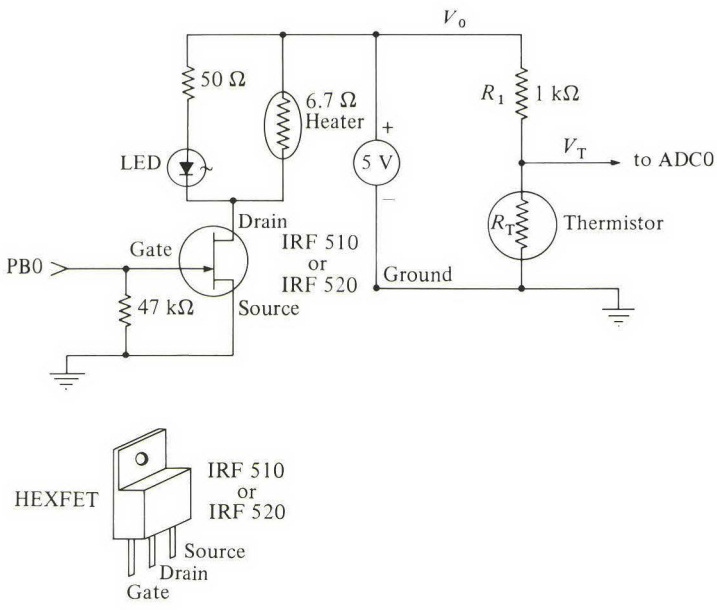


Fig. 3.11. HEXFET connections and pin diagram for thermistor apparatus.

To see how these devices are used, set up the circuit as shown in Figure 3.11. It will act like the push button switch you used earlier but will be controlled by the computer. When a HI signal is applied to the gate of a HEXFET, the device conducts current like a closed switch; when a LO signal is applied, the device acts like an open switch, ie, it has infinite resistance. Connect the gate of the HEXFET to PB0 after resetting the computer with a CTRL-RESET CR. This initiates Port B (as well as the other ports) as an INPUT port so that no potentially dangerous outputs are generated when the computer is idle. The 47 kΩ resistor which is connected between the HEXFET gate and ground is to insure that the HEXFET will be off in the absence of a signal specifically to put it on. That is the case if the port is set as an INPUT line with reset.

Test the circuit using the immediate mode of BASIC. To turn on the HEXFET (and thus the heater) it is first necessary to set Port B up as an OUTPUT port; type in POKE 50178,1 CR (50178 = DDRB). Now, a POKE 50176,1 (50176 = DRB) should turn the pilot light on indicating that the heater is on. To turn the heater off type POKE 50176,0 CR. In doing this and subsequent experiments you must take care that the heater is not left on indefinitely; that will heat the system continuously and destroy the thermometer. To turn to off use POKE 50176,0 or turn the power supply off.

Exercise 3.12.1 Temperature controller

Write a program for a temperature controller following the flow chart in Figure 3.12. The program should ask you to type in a

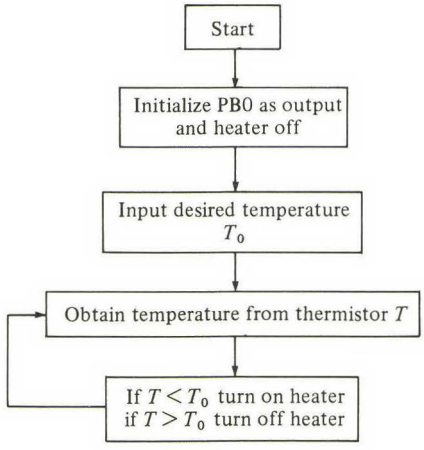


Fig. 3.12. Flow chart for temperature controller.

temperature. The computer should then turn the power to the heater on and off in response to the thermistor voltages read. Run the program and demonstrate that the thermometer does stabilize to the temperature typed in. When testing be sure the Light Emitting Diode (LED) (and heater) are off after you halt your program (POKE 50176,0 CR).

For those interested, try using the statement ON ERR GOTO to detect the control C you used to stop the program and then to ensure that the heater is off before stopping. See the reference manuals.

Exercise 3.12.2 Temperature controller with hysteresis

So that heater is not turning on and off rapidly at the desired temperature, modify the program to turn on the heater when the temperature is below desired temperature minus 1.0° ($T_0 - H$) and turn off the heater when it is above the desired temperature plus 1.0° ($T_0 + H$).

The process of turning the heater on and off used in the program of Exercise 3.12 is called hysteresis. It is used in many process control situations to stabilize the system. A thermostat for a household furnace uses hysteresis so that the furnace doesn't turn on and off too quickly. In a later section you will be using a Schmitt trigger which uses hysteresis to stabilize voltages.

4 Timing

In many experiments, the measurement of interest is the change with time of a particular quantity (eg, dx/dt). One of the most useful capabilities of a computer is to provide accurate and varied timing signals so that these measurements can be made. Indeed, the internal operation of the computer requires the orchestration of many events to the beat of the internal clock. In this section several ways of generating time intervals will be presented.

4.1 Timing loops in BASIC

A simple method of generating time intervals is to use the time required by the computer to execute BASIC instructions. This method is neither precise nor constant but nevertheless is useful in situations where those qualities are not required.

Exercise 4.1.1 Square wave output (BASIC)

(*a*) Run the following BASIC program which uses PB0 as an output. This is a program you have used before so you might have it on your disk. (Note: disconnect the wires to the circuit of the previous experiment before running.)

```
 5 REM EX 4.1.1A
10 POKE 50178,1   Init DDRB0 for output.
20 POKE 50176,0   Put PB0 to LO.
30 POKE 50176,1   Put PB0 to HI.
40 GOTO 20        Repeat.
```

With the program running look at the output (PB0) with the oscilloscope. You may need to adjust the oscilloscope triggering level and timebase to obtain a steady trace. Note the time it takes for one period and the time PB0 is HI and the time it is LO.

(*b*) Now try the following program:

```
 3 REM EX 4.1.1B
 5 B=50176
10 POKE B+2,1
20 POKE B,0
        B,1
        20
```

Again measure the time PB0 in Hi and the time is LO. Why are the HI and LO times different? Why are the times different from those of Program 4.1.1A?

(*c*) Now add the following program lines and LIST the program so you see how they fit in.

```
  REM EX 4.1.1C
 8 N =10
35 FOR I = 1 TO N
38 NEXT I
```

Run the program and note the HI and LO times again. Try different values of N and determine the time required for one FOR–NEXT loop in this program. Also try placing a statement in the middle of the loop. Some interesting ones might be:

```
37 X= I*I +1   (this is called a 'flop')
37 X= I^2 +1
37 REM
37 PRINT "A";
```

Be sure to keep a record of your results.

This type of timing loop could be placed anywhere in a program to provide timebase. However, it suffers from several disadvantages. First, since every BASIC statement takes a different amount of time, it is very difficult to predict the exact amount of time a loop will take. You must resort to trial and error and use an oscilloscope to obtain a particular desired time. Second, the BASIC interpreter is slow when compared to the capability of the microprocessor itself. Frequently BASIC is just too slow to measure the time interval between events in an experiment. A third disadvantage is that the timing is not independent of the program statements. If you change a program line or add a statement even elsewhere in the program, the timing of the loop may change and you will need to readjust it.

Programming in assembly language (more on that later) can solve the first two problems. However, the 6522 VIA which is discussed in Sections 4.4 and 4.5 provides an easier way to do timing which is fast, accurate and independent of the program. The next section illustrates one use of BASIC timing loops.

4.2 Stepping motors

Stepping motors are used to position apparatus of all kinds precisely. A stepping motor rotates a shaft a small increment of a turn for each pulse of electric current it receives. An electric clock is a stepping motor: it rotates a fixed, small amount for every pulse of current it receives from the wall power outlet. The power outlet provides the current which reverses polarity 60 times each second so that by using gears the hands rotate at the

proper speed. A clock motor always rotates in a fixed direction (unidirectional). Some stepping motors can be made to rotate either clockwise or counter-clockwise under computer program control (bidirectional); the one which you will use and the one in the disk drive which positions the reading head are bidirectional.

Exercise 4.2.1 Single step of stepping motor

(*a*) To see how a stepping motor and controller IC are used, make connections to PB0 and PB1 as indicated in Figure 4.1. Write a short BASIC subroutine to generate a single negatively going pulse out on PB0 by setting PB0 and PB1 for output and PB0 HI initially; then send PB0 LO then HI to pulse the motor once.

(*b*) Write a second subroutine which allows you to control the direction of rotation by specifying the level of PB1. The awkward voltage programming for the motor itself is done by the 4202 control Integrated Circuit (IC) so that you need only specify the direction by setting the polarity of the direction control and then applying a short pulse to the stepping input.

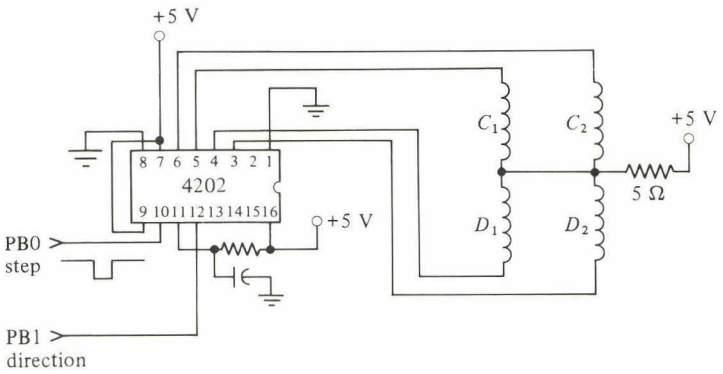


Fig. 4.1. Stepping motor controller (4202) connections.

The mechanics of a stepping motor are shown in Figure 4.2. The rotor is a permanent magnet with 12 sets of north and south poles; the stators each have 12 sets of fingers which can be magnetized electrically. Each stator has 2 coils of wire inside it, labeled C and D. If coil C is energized the fingers marked A become north poles, those marked B become south poles. Coil D energizes the stator with reversed current direction relative to the stator so that A becomes a south pole and B a north.

Figure 4.2(*a*) shows the motor pulled apart to show the relationship of the stators and the rotor. Figure 4.2(*b*) shows the rotor unravelled with its north and south poles lying next to one another. There are two sets of stators, the fingers of each are displaced from one another in azimuth as shown in Figure

4.2(*a*). With the rotor unravelled as shown, it can be pulled to the right by energizing the A_2B_2 stator set with coil D_2 so that A_2 becomes a south pole and B_2 a north pole. The rotor will then move over one step so that the north poles of the rotor lie under the south poles of the stator 2.

The controller IC has two inputs: a direction control and a step control. A HI level on the direction control signals movement in one direction, a LO level in the other. The controller is set so that each time the voltage goes from LO to HI on the step control input, the stepping motor will advance two steps in the appropriate direction. Between pulses the step control should be left HI. In the case of the stepping motor illustrated, one step is $360/(4 \times 12) = 7.5°$ since it has 12 poles and each step moves the rotor one fourth of a pole distance. Thus one pulse on the step control line will move the shaft 15°. The controller IC regulates the current flow in the four windings of the stator. If has logic circuitry within it so that it knows which coil must be energized to step in the specified direction from where it is. This saves you the trouble of programming these details. If the stepping motor shaft is connected to a gearbox with a 200 : 1 gear ratio, the output shaft will turn one revolution for every 200 revolutions of the motor shaft (200 : 1 gear ratio).

Exercise 4.2.2 Maximum stepping rate

(*a*) You have seen that it is only necessary to use two POKE statements which make PB0 go LO and then back to HI to step the motor once. The stepping motor is a mechanical device which is inherently slow; thus it is important that there be a reasonable amount of time between pulses. By using a FOR–NEXT loop to waste time between pulses, determine the maximum number of pulses per second that

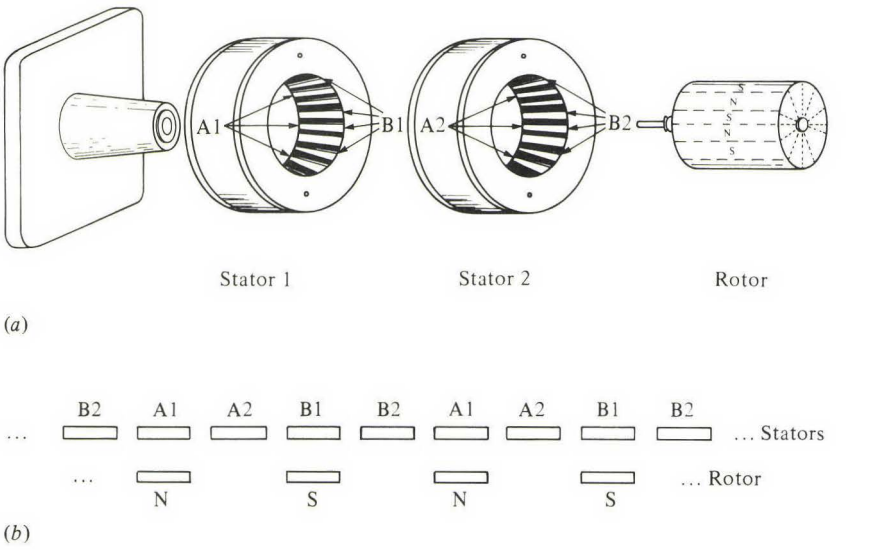


Fig. 4.2. Stepping motor opened up. (*a*) Stators 1 and 2 each have 12 pairs of poles (A1, B1, A2, B2) and 2 coils (C1, D1, C2, D2). Current in coil C makes A poles north and B poles south and current in coil D makes A poles south and B poles north. (*b*) Poles and rotor flattened out to show staggering of stator 1 and stator 2's poles. To move rotor one position to the right from the position shown, turn coil D1 off and D2 on (A2 is then south and B2 north). To move once to left, turn D1 off and C2 on (A2 north and B2 south).

the motor will respond to. Do this by varying time delay of the loop and watching to see whether the motor responds properly or not. For example, a program which give 200 pulses to the stepping motor of Figure 4.2 with the gearbox attached should turn the gearbox output shaft 15°. Use the oscilloscope to measure the time beween pulses.

(*b*) Write a subroutine, to be used by later programs, which will step the motor in the direction specified before entering. The program should delay the proper amount of time for the stepping motor before returning. Set the delay time of the loop so that you never ask the motor to rotate faster than $\frac{1}{2}$ the maximum rate. This will ensure reliable operation.

Exercise 4.2.3 Positioner

Using the subroutine in Exercise 4.2.2 write a program which moves the output shaft of the gearbox to a specified angle. Since it is only the number of pulses and not their detailed timing which is important, a simple BASIC program is adequate. At the outset the current position of the stepping motor should be INPUT to the program. The program should then ask you for the angular position of interest and step the motor to that angle. After it is at this position the program should come back and ask for the next desired position. To avoid truncation errors in calculating the angle, keep track of steps not degrees. The program should accept the position or negative numbers of any magnitude and turn the shortest route to the angle.

4.3 Number systems

To work with I/O devices and with assembly language programs, it is necessary to go back and forth among the representations of numbers in decimal, hexadecimal and binary. Except for a few commands, BASIC statements use decimal (base 10) representations for numbers. However, internally the computer represents all numbers and characters in binary (base 2). This internal conversion to binary is usually not important to the user but becomes so when connecting I/O devices to the computer. Then a binary representation directly corresponds to signal levels on the I/O lines. Hexadecimal numbers (base 16) are a convenient shorthand notation for long binary numbers.

When a decimal number is written down, say 348, what is really indicated is that there are 3 hundreds, 4 tens and 8 ones (Figure 4.3). This can be described by the equation

$$348 = 3 \times 10^2 + 4 \times 10^1 + 8 \times 10^0 \qquad (4.3.1)$$

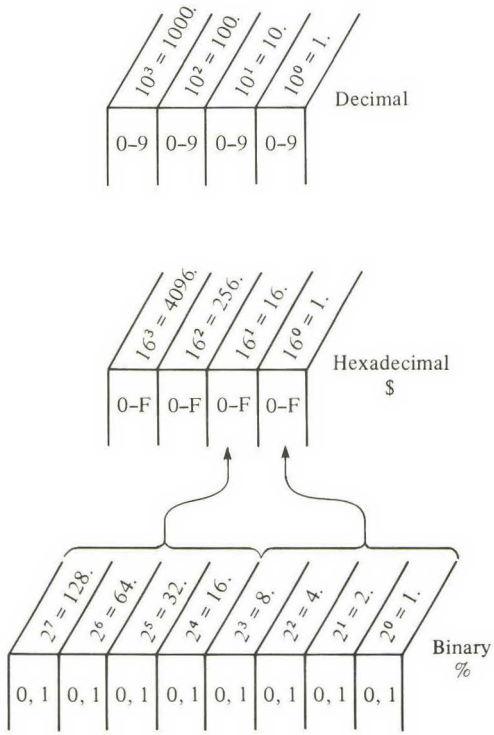


Fig. 4.3. Decimal, binary, and hexadecimal number representations.

In exactly the same spirit a hexadecimal number with the characters 1234, represents

$$\$1234 = 1 \times 16^3 + 2 \times 16^2 + 3 \times 16^1 + 4 \times 16^0 \qquad (4.3.2)$$

It is useful to remember that $16^3 = 4096$, $16^2 = 256$, $16^1 = 16$, and $16^0 = 1$. Hexadecimal numbers are often indicated by a \$ sign preceding the number. Sometimes a period is used to indicate a decimal number even if it is an integer (for example 348.). Each of the characters 1, 2, 3, 4 in Equation 4.3.2) could be a number from 0 to 15 just as in the decimal representation each place (column) has a number between 0 and 9 (Figure 3.3). In hexadecimal, to represent 10, A is used, 11, B, etc., as shown in Table 4.1. As an example $348. = \$15C = 1 \times 256 + 5 \times 16 + 12 \times 1$.

A number will be preceded by a % sign to indicate that the characters which follow are a number in binary representation. Thus

$$\%0101 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \qquad (4.3.3)$$

The magic numbers here are $2^7 = 128$, $2^6 = 64$, $2^5 = 32$, $2^4 = 16$, $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, $2^0 = 1$. In writing down binary numbers it is convenient to write them down in groups, four digits (bits) at a time. This makes it easy to identify the position in which each bit belongs. It also makes it easy to go back and forth between binary and hexadecimal since 4 binary bits = 1 hexadecimal character. Thus, $348. = \$15C = \%0001\ 0101\ 1100$.

To get a feel for the above ideas, use Port BB of the 6522 board to display the output of the eight data lines PBB7–PBB0 on eight LEDs (a schematic

Table 4.1 *Correspondence between binary, hexadecimal, and decimal characters*

Binary %	Hexadecimal $	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

wiring diagram for this is shown in Figure 4.4). Anytime a 1 is put into a particular bit of Port BB the correspoding LED will light up. Data lines PBB7–PBB4 are connected to four red LEDs, data lines PBB3–PBB0 are connected to green LEDs. The 74LS04 integrated circuit between the actual data port lines and the LEDs has what are called line drivers or buffers; they provide the 15 or 20 mA of current required to light up the LEDs. The data lines alone are not capable of generating enough power.

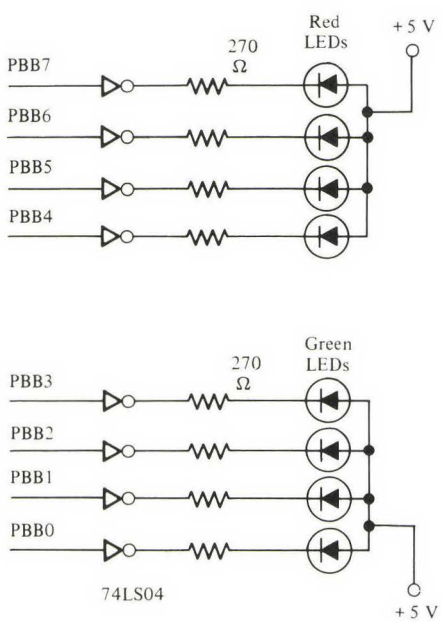


Fig. 4.4. LED number display wiring. Schematic of LEDs on PBB. The 74LS04 IC is a BUFFER/DRIVER to provide current drive for LEDs. A 'HI' on a PBB line will illuminate an LED.

Exercise 4.3.1 LED binary number display

(*a*) Write a BASIC program which sets up all the data lines of PBB for output and sends various numbers to PBB. (DDRBB = 50306, DRBB = 50304).

(i) Make data PBB3, PBB5 and PBB7 HI and the rest LO.

(ii) Send the numbers $28 and $C3 to PBB; remember to convert to decimal. Verify that the expected LEDs light up.

(iii) Send the number %1010 0111 to PBB. What is this in hexadecimal?

Even though PBB is set up for output it is possible to read the data which is in PBB using the PEEK statement in the usual way. Store several numbers in DRBB and note their hexadecimal equivalents and their binary equivalents (the lights which light). Also read them back into the memory using the PEEK instruction and PRINT them out.

(*b*) To get a feeling for counting in binary, enter and run the program below.

```
 5 REM EX 4.3.1B
 6 BA=50304
10 POKE BA+2,255    Set up all PBB lines for output.
20 FOR I=0 TO 255
30 POKE BA,I
40 FOR J=0 TO 200   Insert time delay.
50 NEXT J
60 NEXT I
```

The representation of negative numbers in a computer is one somewhat differently than in common computations. In many ways the system is more logical than the customary one, and it certainly makes things much simpler in a computer. Imagine a continually incrementing binary four-bit counter. A counter is a device which increments each time a pulse is applied. Table 4.2 shows the sequence of digits as the count pulses are added (start at 0 and read upwards). The correspondence to the ordinary number system is shown in the column to the right. Being a four-bit counter the 'readings' repeat every 16 counts, that is, the count after 1111 gives 0000. (This transition is called overflow.) A representation of the numbers 0–15 is naturally done through the correspondence to the counter readings of 0000–1111. One possible way to represent positive *and* negative numbers is to assign the numbers between −8 and 7 to the counter readings of 1000–0111. This is called the two's complement representation. As shown in Table 4.2, −1 becomes 1111, −2 becomes 1110, etc. Notice that the most lefthand bit takes

Table 4.2 *A four-bit counter*

Four-bit binary	Negative numbers interpretation	No-negative numbers interpretation	
1111	−1	15	
1110	−2	14	
1101	−3	13	
1100	−4	12	
1011	−5	11	
1010	−6	10	
1001	−7	9	
1000	−8	8	
0111	7	7	
0110	6	6	
0101	5	5	
0100	4	4	
0011	3	3	
0010	2	2	↑ Add
0001	1	1	↑ Add
0000	0	0	↓ Subtract
1111	−1	15	↓ Subtract
1110	−2	14	
1101	−3	13	
1100	−4	12	
1011	−5	11	
1010	−6	10	
1001	−7	9	
1000	−8	8	

on a special meaning; if it is 1, a negative number is being represented, if 0, a positive one.

Within the computer a particular bit combination, say %1011 will represent 11 one time and −5 at another. An example of this is in the numbering of memory addresses. There is RAM available between $9000 and $9FFF which is normally not used by APPLE routines and can be safely used. Now for instance,

$$\$94AC = 38060. = \%1001\ 0100\ 1010\ 1100 = -27476.$$

Thus typing POKE 38060,163 CR followed by either PRINT PEEK (38060) or PRINT PEEK(−27476) will bring back the number 163 (try it).

Also note that the four-bit counter will repeat its reading every 16 counts. Thus the decimal numbers 11, 27, 43, etc., will all be represented by the same binary combination in a four-bit counter. If we count down 16 counts from 11 we come back to the same binary reading; thus 11 and −5 (which is 11–16) are repesented by the same binary string. Similarly in a 16-bit counter, the bits will repeat every 65536 counts ($2^{16} = 65536$). By the same reasoning the number 38060 and the number −27476 will both be represented by the same 16-bit string since $38060 - 65536 = -27476$.

4.4 Generation of square waves by the 6522

You have used the data output capability of the 6522 VIA to control the heater in the experiment in Chapter 2 and to light the LEDs in Section 4.3. The VIA can do other functions as shown in the data sheets in Appendix G. You need not concern yourself with the details of each function but some practice in deciphering these often cryptic descriptions is valuable. As you work through this section read the data sheets about functions which you have already used and about timers 1 and 2.

There are actually two 6522 VIAs on the card plugged into the APPLE. You have used the VIA1 with registers at memory locations \$C400–\$C40F to control the heater and VIA2 with registers at memory locations \$C480–\$C48F to control the LEDs. Port A (DRA) and Port B (DRB) are used as I/O ports, the I/O function of each line is controlled by DDRA and DDRB. The timers in the 6522 can be programmed to run in several different modes by writing (POKEing) various bit combinations into a special control register called the Auxiliary Control Register or ACR (see Appendix E, Figure 14). The ACR is at memory location 11 (ie, hexadecimal \$B) above the base address and so will be at \$C400 + \$B = \$C40B for VIA1. The mode we will use is with timer 1 (T1) operating continuously and generating a square wave output on PB7. (In Appendix E, read the paragraphs under the heading 'Timer 1 Free-Run Mode', don't worry about 'interrupts' now.) Therefore, bits 7 and 6 in the ACR must be 1. For now, the rest of the bits can be 0, so the ACR should contain %1100 0000 = \$C0 = 192.

The T1 counter counts down at the rate of the internal clock of the APPLE. This rate, called Φ2, is set by a quartz crystal oscillator to a frequency of approximately 1.022727 MHz. The T1 counter counts down at this rate from the 16-bit value loaded into registers 4 and 5 (T1L and T1H). Since T1 is a 16-bit counter, it takes two 8-bit bytes to fully define the PB7 rate. When the counter reaches zero several things happen: the state of PB7 is changed (ie, 0 goes to 1 or 1 goes to 0), the numbers which were originally in the counter registers are automatically reestablished, and counting down begins again. So to obtain a desired rate R at PB7, you need to: calculate the number N of Φ2 cycles in R, make two 1-byte numbers from N and place these into registers 4 and 5 (neglecting the '1.5' and '2' cycle corrections shown in Figure 16 of the Appendix E). As an example, suppose it is desired to have PB7 invert every 0.008 s. This is 8000 μs, so N = 8000 × 1.022727 = 8182. = \$1FF6. HIGH-ORDER = \$1F = 31.; LOW-ORDER = \$F6 = 246. The following program will cause PB7 to invert every 0.008 s. (Note that the period of the square wave is 0.016 s.)

```
 5 REM PROGRAM 4.4.1
10 BA = 50176          Base address of VIA1, $C400.
20 POKE BA+11,192      Set ACR for T1 free run, ie, store $C0 in
                       BA + 11 = ACR.
```

```
30 POKE BA+2,128      Important! Enable PB7 as an output pin,
                      ie, store $80 in DDRB.
40 POKE BA+4,246      Load low-order T1, $F6.
50 POKE BA+5,31       Load high-order T1, $1F and start
                      countdown.
```

RUN the program and look at PB7 with the scope. Notice that the timer keeps operating even after the program has stopped! Press CTRL RESET to stop its operation.

Exercise 4.4.1 Square wave on PB7 VIA 6522

Modify Program 4.4.1 to invert PB7 every 0.005 s; ie, a square wave of period 0.01 s.

4.5 Making an interval timer

Experiments frequently require the measurement of time intervals. The combination of the T1 control of output on PB7 (you have used above) with the pulse counting mode of counter T2 (described below) can provide this on the APPLE. In the ACR, bit 5 controls the mode of operation of T2. When set to 1, the value in the T2 counter registers (low-order is register 8, high-order is register 9) decrements on each HI to LO transition of a signal input to PB6. (Described further in Appendix E, Figures 17 and 19 and text under the heading 'Timer 2 Pulse counting mode'.)

Exercise 4.5.1 T1–T2 interval timer

Put a wire from PB7 to PB6 on the protoboard. This will allow T2 to count the number of PB7 periods (remember the PB7 rate is controlled by T1). The following program will start T2 counting down at the rate of 0.1 s which is coming from PB7.

```
 5 REM EX 4.5.1
10 BA = 50176          Base address of VIA1.
20 POKE BA+11,224      Set ACR bits 7,6,5 to 1, others to
                       0 (%1110 0000).
30 POKE BA+2,128       Enable PB7 as output, PB6 (and
                       also PB5–PB0) as input (put
                       %1000 0000 in DDRB).
40 POKE BA+8,255       Initialize T2 low-order counter to
                       maximum value $FF.
50 POKE BA+9,255       Initialize T2 high-order counter.
60 POKE BA+4,192       Initialize T1 low-order to $C0.
```

```
 70 POKE BA+5,199          Initialize T1 high-order to $C7 and
                           start counter and square wave
                           generation.
100 T = 256 * PEEK(BA+9)   Get T2H (high byte) each unit =
                           256 PB6 HI to LO transitions.
110 T = T + PEEK(BA+8)     Get T2L (low byte) and add to T
120 PRINT "T ="; T         Print T2 value.
```

In this program T2 is set to its maximum value $FFFF. At each HI–LO transition on PB6, the timer/counter decrements one count (eg, T2 = FFFF, FFFE, FFFD, . . .). Since these transitions on PB6 are wired to PB7, they occur every 0.1 s (which is twice the T1 rate).

Now, periodically run the part of the program from instruction 100 on to see that the counter is indeed decrementing. To do this use the command 'GOTO 100 CR' which will pick up the program from where it ended with variable values left as they were. This is different from the command RUN 100 in that the latter will first set all variables to zero (eg, BA) and then begin executing at 100 giving nonsense results.

Subtracting successive values of *T* which appear on the screen should give you the time elapsed between the two PEEKs in units of 0.1 s. check with a watch to see that this is indeed true.

Exercise 4.5.2 Beeper

Write a program which BEEPS the terminal 'bell' at one second intervals. The statement to ring the bell is 'PRINT CHR$(7);' Run PB7 at 0.01 s per cycle. In this program the T2 counter should be set up as in the above program. Then read T2L and T2H periodically and take differences to obtain the desired one second interval.

The one second timing program in Exercise 4.5.2 may be in error when T2 counts down past $0000 because the subtraction done will give a negative time interval. For instance, if the first time determined is $005F = 95. and the second is $0050 = 80., subtracting the second from the first (as is normally done) gives $000F = 15., a valid number. But if the first is $005F = 95. and the second is $FFF5 = 65525., the result is −65430 but should be 107.

There is another problem which occurs when reading the value of time T2. In the statement:

T2 = 256*PEEK(TH) + PEEK(TL)

(where TH is the high byte of T2 and TL the low byte) the peek to TH is done before the PEEK to TL. If a count should come into the timer between the two PEEKs *and* if TL is at $00 when TH is PEEKed, an error will occur. For example:

TH	TL	
2B	01	
2B	00	PEEK(TH) occurs here.
2A	FF	PEEK(TL) occurs here. Gives: T2=256* ($2B)+($FF)=256*43+255.

Both of these problems can be fixed by using additional BASIC statements in your program.

5 Thermal diffusion

The experiments which you will be called upon to do in this chapter give you a chance to apply the 6522 timing concepts and to review the use of the ADC while learning about the phenomenon of diffusion. Specifically, you will be studying thermal diffusion but many of the concepts encompass a variety of other phenomena.

5.1 Heat flow equation

In this section you will explore some of the physical and mathematical considerations of one-dimensional heat diffusion. When heat is added to a material there are two parameters which affect the distribution of temperatures: the specific heat (or heat capacity) and the thermal conductivity. The specific heat indicates how much heat is added to a mass of material for a specified temperature rise. The thermal conductivity indicates how fast the thermal energy is transported through the material.

Consider the flow of heat in a rod as shown in Figure 5.1. The specific heat C of a material is the ratio of the amount of heat added $\mathrm{d}q$ (Joules) to the resulting rise in temperature $\mathrm{d}T$ (degrees Kelvin) per unit mass $\mathrm{d}m$ (kg); thus $C = (\mathrm{d}q/\mathrm{d}T)/\mathrm{d}m$, (see Equation (3.3.1)). For a rod of cross-sectional area A, the volume $\mathrm{d}V = A\,\mathrm{d}z$ and $\mathrm{d}m = \rho\,\mathrm{d}V$ where ρ is the density. So, the amount of heat added to the length $\mathrm{d}z$ of the rod is

$$\mathrm{d}q = C\rho A\,\mathrm{d}T\mathrm{d}z = sA\,\mathrm{d}T\mathrm{d}z \tag{5.1.1}$$

where s is the volumetric heat capacity, $C\rho$.

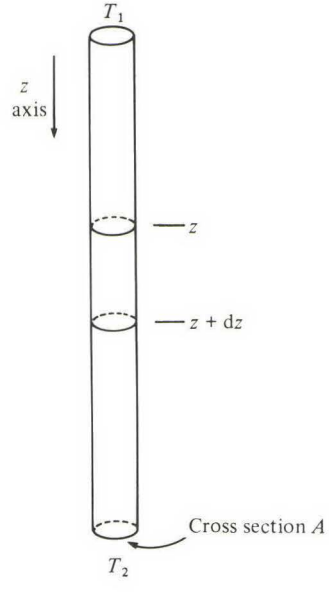


Fig. 5.1. The flow of heat in a rod of specific heat C (J/kg K) and thermal conductivity k (W/m K).

When one end of the rod is hotter than the other there will be a net flow of energy from the hot end to the cool end. The power P (Watts) of this heat flow down the rod is the heat energy per unit time flowing past a point on the rod $P = \mathrm{d}q/\mathrm{d}t$ (Equation 3.3.2)). For one-dimensional heat flow, P is proportional to the temperature gradient $\mathrm{d}T/\mathrm{d}z$, the thermal conductivity k (W/m K) and the cross-sectional area;

$$P = -kA\ (\mathrm{d}T/\mathrm{d}z) \tag{5.1.2}$$

There is a minus sign because heat flows from higher to lower temperatures. In writing this equation, it is assumed that the rod is insulated; no heat escapes from the rod by conduction, convection or radiation. The net heat gain per unit time $\mathrm{d}q/\mathrm{d}t$ in the piece of rod between z and $z + \mathrm{d}z$ is given by the difference in the power flowing in at z and the power flowing out at $z + \mathrm{d}z$, so

$$dq/dt = P(z) - P(z + dz) = -(\partial P/\partial z)dz \tag{5.1.3}$$

Combining Equations (5.1.1), (5.1.2) and (5.1.3) gives the differential equation for heat flow in a rod

$$s(\partial T/\partial t) = k(\partial^2 T/\partial z^2) \tag{5.1.4}$$

This equation has many solutions; if a quantity of heat is added to the rod quickly (a heat pulse), the solution can be written as follows:

$$\left.\begin{aligned} B_1 &= \text{constant} \\ B_2 &= \text{constant} \\ T(t, z) &= B_1 + B_2 \exp(-z^2 s/4kt)/t^{1/2} \end{aligned}\right\} \tag{5.1.5}$$

Details of how this solution can be obtained are found in Appendix F.

Exercise 5.1.1 Impulse heat diffusion solution

(*a*) Show that Equation (5.1.5) is a solution of Equation (5.1.4).

(*b*) Show that B_1 can be interpreted as the starting temperature; that is, $T(t, z)$ at $t = 0$ for $z \neq 0$. $B_1 = T_s$.

The solution (5.1.5) describes the temperature at any point in the rod as a function of time after an impulse of heat has been added at $z = 0$. Before proceeding further it is useful to examine the graphs temperature T vs. distance z of the solution at various times after the impulse. These are shown in Figure 5.2. In this figure, $T = T(t, z) - T_s$. At times near zero, the heat, and thus the excess temperature, is concentrated near $z = 0$. As time progresses the heat diffuses away from the center to larger and larger values of z with the peak temperature decreasing in time.

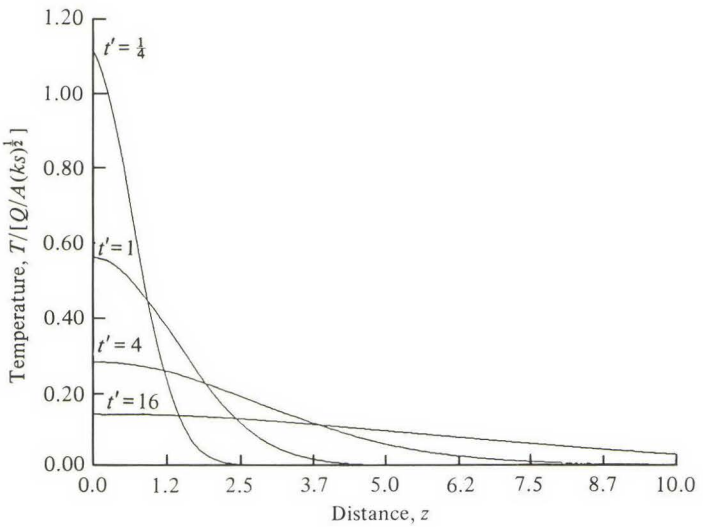


Fig. 5.2. Heat flow in a rod, temperature vs. distance where $t = t's/k$..

An important point to note is that the solution is symmetric with respect to z, just as much heat diffuses up as down the rod. Since there is no heat flow across the cross section at $z = 0$, cutting the rod at $z = 0$ will not modify the form of the solution although now all the heat added goes one way. This half-space rod is the configuration which you will study experimentally.

To obtain a theoretical expression convenient for analyzing a quantitative experiment, it is useful to relate the constant B_2 in Equation (5.1.5) to the total heat Q added to the rod (from $z = 0$ to $z = \infty$) by integrating Equation (5.1.1). Consider T as the excess temperature above $T(0)$, ie, $T = T(t, z) - T_s = T' - T_s$; integrating Equation (5.1.1) from temperature T_s to T' gives

$$dq = sA\,dz(T' - T_s) = sAT\,dz \tag{5.1.6}$$

To integrate from $z = 0$ to $z = \infty$, use Equation (5.1.5) to describe the variation of temperature at any z and t. Then

$$Q = \int_{z=0}^{\infty} sA \frac{B_2 \exp(-z^2 s/4kt)}{t^{1/2}}\,dz = \frac{sA}{t^{1/2}} B_2 \int_0^{\infty} \exp(-z^2 s/4kt)\,dz$$

$$= \frac{sA}{t^{1/2}} B_2 \frac{\pi^{1/2}}{2} \left(\frac{4kt}{s}\right)^{1/2} = B_2(\pi ks)^{1/2} A \tag{5.1.7}$$

solving for B_2 and inserting into Equation (5.1.5)

$$T(t, z) = \frac{Q}{A} \frac{1}{(\pi ks)^{1/2}} \frac{\exp(-z^2 s/4kt)}{t^{1/2}} + T_s \tag{5.1.8}$$

As written Equation (5.1.8) is not in an optimum form for displaying some of the important features it contains. It is often very helpful, particularly for purposes of recognizing the domain of behaviour in a given physical situation, to relate the quantities in an equation to physically significant parameters rather than simply measuring time in seconds, temperature in degrees centigrade, etc. You saw this before in the equation for the thermistor resistance as a function of temperature of Chapter 2. The natural parameters there being R_0 and T_0. For displaying the change in temperature T as a function of time t at a fixed z, Equation (5.1.8) can be written in terms of a characteristic time t_1 and a characteristic temperature T_1 as

$$\left.\begin{aligned} T/T_1 &= (t_1/t)^{1/2} \exp(-t_1/t) \\ t_1 &= sz^2/4k \\ T_1 &= 2Q/Azs\pi^{1/2} \\ T &= T(t, z) - T_s \end{aligned}\right\} \tag{5.1.9}$$

Equations (5.1.9) immediately show several important points. First, the variation of temperature with time at a constant z can be related to just two parameters t_1 and T_1. Second, the characteristic time scale t_1 is proportional to z^2; this is a general property of diffusion phenomena.

Exercise 5.1.2 Graphing the heat diffusion equation

(*a*) Use the AMPERGRAPH utility program to plot T/T_1 as a function of t/t_1 from $t/t_1 = 0.1$ to $t/t_1 = 10$.

(*b*) Show that the temperature T_1 is proportional to the temperature rise which a quantity of heat Q would produce if absorbed by a length z of the rod with the heat distributed over the length of the rod. Find the constant of proportionality.

(*c*) The temperature T_1 can also be related to the maximum values which T assumes, show that the maximum occurs at $t/t_1 = 2$ and at the maximum $T/T_1 = 0.43$.

5.2 Numerical integration of the heat flow equation

Appendix F shows a solution to the differential equation for one-dimensional heat flow for an impulse of heat at $t = 0$. For other starting conditions or parameter dependencies the equation could be much harder, if not impossible, to solve. For example, the thermal conductivity k is really temperature dependent $k = k(T)$ and so cannot be treated as a simple constant parameter. An analytical solution quickly becomes impossible and you must resort to numerical solutions.

General numerical integration of partial differential equations is a broad and difficult subject. The following will be a simple procedure which works in this case but must be used with care. It is really only meant to illustrate a general approach. For further discussion see *Numerical Recipes The Art of Scientific Computing*, by Press *et al.* in the bibliography.

The basic equations for the flow in a rod are the static equation for the heat capacity Equation (5.1.1) and the dynamic equation with the thermal conductivity Equation (5.1.2) which are combined to form the differential equation, Equation (5.1.4). However for purposes of numerical integration, it is best to leave them separate and write them in this form:

$$\Delta Q = -kA\ \Delta T\ \Delta t/\Delta z \tag{5.2.1}$$

$$\Delta T = \Delta Q/As\ \Delta z \tag{5.2.2}$$

where Δ is assumed to approach zero.

Now break up the rod (Figure 5.1) into N_z pieces of length Δz each and consider the ith piece; the heat flowing into this piece in the time Δt will be:

$$Q_{\text{in}} = kA(T_{i-1} - T_i)\ \Delta t/\Delta z \tag{5.2.3}$$

If the temperature in element $i - 1$ is hotter than in the element i then Q_{in} will be positive. The heat flowing out of the piece will be:

$$Q_{\text{out}} = kA(T_i - T_{i+1})\Delta t/\Delta z \tag{5.2.4}$$

The difference of the two is the heat gained or lost in the element:

$$\Delta Q_i = Q_{\text{in}} - Q_{\text{out}} \tag{5.2.5}$$

This heat changes the temperature of the element in proportion to its heat capacity:

$$\Delta T_i = \Delta Q_i/As\ \Delta z \tag{5.2.6}$$

and so

$$T_i^{\text{new}} = T_i^{\text{old}} + \Delta T.$$

Exercise 5.2.1 Integration algorithm

(*a*) Equations (5.2.3)–(5.2.6) can be used to determine the temperature in any element at any time (which is all we want out of a solution to the differential equation) as follows:

First specify Δz and N_z and the temperature T_i in each of the elements $i = 1 \ldots N_z$ at the start, which in the case of the laboratory experiment will be $T_1 = Q$ (from heater)$/As\,\Delta z$ and T_2, T_3, ..., $T_{N_Z} = 0$. Also specify the time step desired Δt.

Next, make the calculations in Equations (5.2.3)–(5.2.3) for each of the elements using the old temperatures to give new temperatures.

Repeat the last step until the desired time is reached.

Now repeat the whole procedure but with a smaller Δz and/or smaller Δt. Compare these results with the previous ones to make sure that they are not sensitive to the size of the steps used. If they are, reduce the step size again.

Since the theory deals with an infinite rod, another parameter which needs to be examined is the length of the rod $N_z \Delta z$. Make sure that it does not affect the results.

(*b*) With a working program in hand, the results can be checked by comparison with the analytical solution. But now the analysis can be taken further; consider the following questions and how the program might change to answer them:

What is the effect of a short rod or a rod with one end clamped at a constant temperature?

What is the effect of a thermal conductivity k which is a function of temperature, eg, $k = k'/T$?

What is the effect of the heat impulse occurring over a longer interval of time?

How does convective and radiative heat loss affect the temperature distribution?

5.3 Experimental setup and program development

The apparatus you will use for these experiments is illustrated in Figure 5.3. In the top of the copper rod (#10 copper wire, 2.59 mm diameter) is set a 3.3 Ω resistor which is used as a heater. Current can be switched into the heater under program control using the IRF 520 HEXFET in a manner similar to that used in Section 3.12. After generating a short pulse of heat by momentarily turning on the HEXFET, the computer will measure the increase in temperature at two positions down the rod using two thermistors. The thermistor positions are as shown on Figure 5.3. A plot of

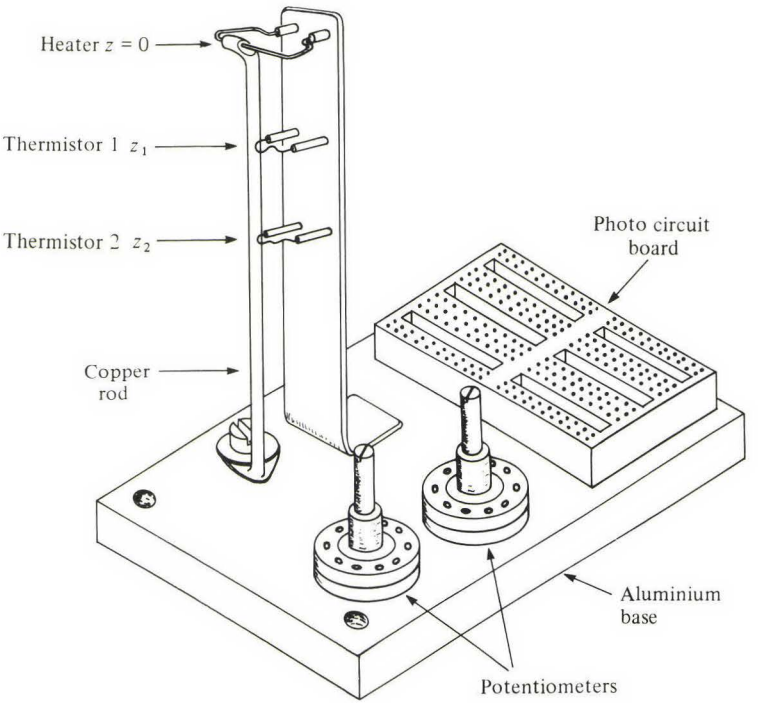
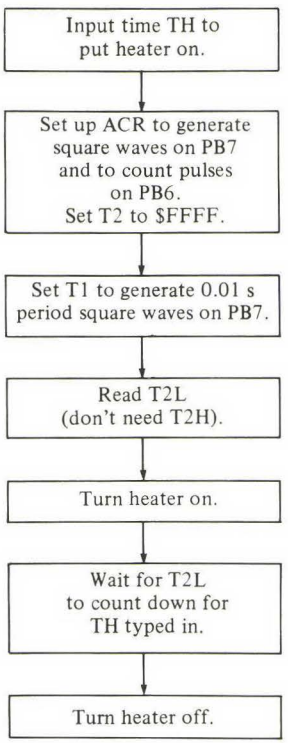


Fig. 5.3. Heat diffusion apparatus.

Fig. 5.4. Flow chart for Exercise 5.3.1.



the temperature vs. time at each of these thermistors will yield values for the heat capacity and thermal conduction constants of copper and also demonstrate the functional dependence of heat diffusion on time and distance.

Exercise 5.3.1 Heat impulse to rod

Write a subroutine which uses the 6522 $T1$–$T2$ timing set-up of Chapter 4 to turn the heater on for an amount of time which you type as input data into the computer. Use PA0 to control the HEXFET. A flow chart outlining the steps in the program is shown in Figure 5.4. Check your program and apparatus by putting an oscilloscope probe between the heater and ground and then turning the heater on for times ranging from 0.1 s to 2 s. Note the voltage across the heater when it is on with the oscilloscope and compute the power being put into the heater. (Remember: do not put the alligator clip to any circuit point which is not at ground potential!)

5.4 Voltage amplifier

The change in temperature of each thermistor from an initial temperature $(T(t, z) - T_0)$ is the significant quantity to measure in this experiment. However the temperature increments and thus the voltage

changes are very small; if the ADC is connected directly to the thermistor as in Chapter 3, the changes are less than the step size of digitization. To overcome this problem an amplifier is used to boost the voltage change. On the protoboard attached to the experimental apparatus is an amplifier using a CA3140 operational amplifier; a schematic diagram is shown in Figure 5.5. It is not necessary to understand the details of this amplifier circuit except to note that the relationship between the three voltages V_A (output) (pin 6), V_1 (pin 2) and V_T (pin 3) is given by

$$V_A = G(V_T - V_1) \tag{5.4.1}$$

For the circuit components used, the gain G is equal to 21.

The amplifier output (V_A) is constrained by the characteristics of the CA3140 to be between 0 V and +3 V. Since a rise in thermistor temperature will lead to a rise in the output voltage of the amplifier, the potentiometer R_1 should be set so that the output voltage of the circuit starts near the lowest voltage before a heat pulse is applied. This will allow the greatest voltage swing as the thermistor heats up without exceeding the 3 V limit. Using the oscilloscope to monitor the output voltage of each amplifier, set the potentiometers (one for each amplifier–thermistor combination) so that the amplifier outputs are about 0.20 V before you start each run. When this is done each potentiometer R_1 has been adjusted to be essentially the same resistance as the thermistor resistance R_T before a temperature pulse is applied. Since the amplifier gain is 21, the change in the output voltage ΔV_A will be 21 times greater than the change in the thermistor voltage ΔV_T.

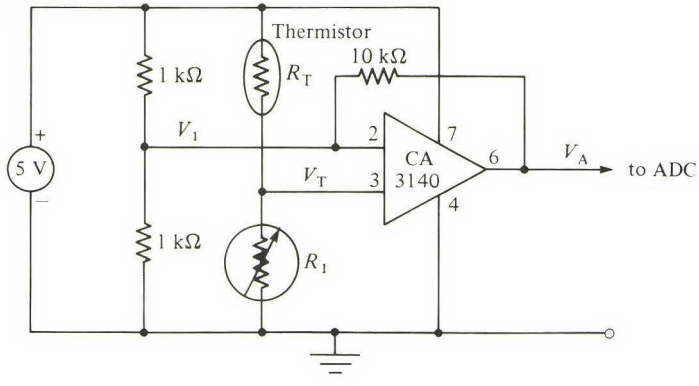


Fig. 5.5. Voltage amplifier circuit for heat flow apparatus.

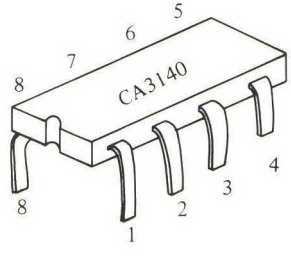


Fig. 5.6. Flow chart for Exercise 5.4.1. use
PRINT PEEK (49312);" ";
– putting a semi-colon and quotes with a few spaces will make it possible to view what is going on using the screen with a minimum of programming.

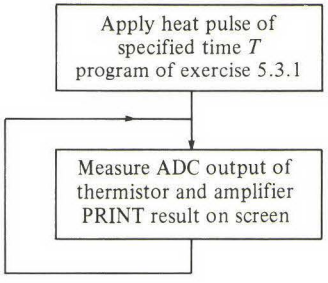
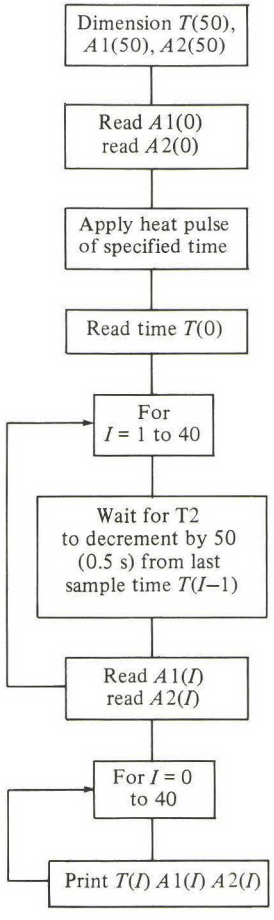


Fig. 5.7. Flow chart for Exercise 5.4.2.



Exercise 5.4.1 Amplifier check

Before writing a detailed program write a simple program to see that the apparatus is functioning following the outline shown in Figure 5.6. When you run this program you should see on the oscilloscope the voltage output rise and then slowly fall. It should start above 0 V and not try to go above 3 V. Stop it after 20 or 30 s using CONTROL C.

Do the same to check thermistor 2, the lower thermistor. You will need to let the apparatus cool down and reset the potentiometer between heat pulses.

Exercise 5.4.2 Heat flow real-time plot

(*a*) The next task is to make thermistor ADC measurements at specified times. To do this, modify the program by putting in waiting loops as indicated by the Figure 5.7. Note that a sample is taken before the heater is turned on (A1(0) an A2(0)). This records the baseline ADC reading. The heating of the rod then changes the ADC reading from this starting value.

(*b*) Now combine this with AMPERGRAPH so that these relatively rough, unprocessed data are plotted in real time while they are being collected. You have enough time (even though BASIC is very slow) to read the time, $A1(I)$ and $A2(I)$ and to get them on a graph as data points before the next reading must be taken $\frac{1}{2}$ s later. the AMPERGRAPH symbol plotting is too slow so use '& DRAW,X,Y: & PENUP' for each point. In addition, it may be too slow to convert the ADC readings to voltages so just leave them as raw data. Also make a data file of the ADC readings $A1(I)$, $A2(I)$ and times $T(I)$ together with the time the heater was kept on (T_H).

5.5 Data analysis

Before proceeding to more data plots and analysis, here are some additional mathematical considerations. We will assume that the temperature and voltage changes at the thermistor are small enough to that their behaviors are adequately described by differentials. Thus: (change in amplifier output voltage) = (gain) × (change in the input voltage)

$$dV_A = G dV_T \tag{5.5.1}$$

The relationship between V_T and R_T is similar to the thermistor experiment of Chapter 3, ie, $V_T/V_0 = R_1/(R_1 + R_T)$ with $V_0 = 5$ V. The relationship between dV_T and thermistor resistance changes dR_T can be obtained by differentiation; the result (which you should work out) is

$$\frac{dV_T}{V_0} = -\frac{dR_T}{R_1}\left(\frac{1}{1 + R_T/R_1}\right)^2 \tag{5.5.2}$$

Noting that R_1 and R_T are adjusted to be nearly equal at the outset gives

$$dV_T/V_0 = -dR_T/4R_T \tag{5.5.3}$$

The next task is to relate change in the thermistor resistance to changes in temperature. The relation between thermistor resistance and temperature is $R_T = R_0 \exp(T_0/T_a)$ as discussed in Chapter 3. Differentiation of R_T with respect to temperature T_a gives

$$\frac{dR_T}{R_T} = -\frac{T_0}{T_a}\frac{dT_a}{T_a} \tag{5.5.4}$$

where T_a is the absolute temperature (K) (not the excess temperature, $T(t, z) - T_s$) and dT_a is a small temperature change due to the heat pulse. Thus if dT_a is small, it can be approximated by the measured temperature change of the apparatus (ie, the excess temperature) and T_a can be approximated by room temperature. Appropriately combining Equations (5.5.3) and (5.5.4) gives the result

$$dT_a = T_a \frac{4}{G}\frac{T_a}{T_0}\frac{dV_A}{V_0} \tag{5.5.5}$$

As Equation (5.5.5) shows, the change in output voltage in volts is not important, only its ratio with V_0. This ratio dV_A/V_0 is equal to the ratio of the change in ADC units to the ADC full scale reading.

Exercise 5.5.1 The thermal conductivity and specific heat of copper

Plot the data which you have taken with the vertical axis in temperature change from the initial temperature (Equation (5.5.5)) and the horizontal axis in seconds (T_0 = 3440 K for the GB32J2 thermistor). As a first step in the analysis of these data use the relations derived in Exercise 5.1.2(*c*), ie, visually estimate the position T_{peak} and height T_{peak} of the peak and calculate t_1 and T_1 from these values. Use these estimates to draw a curve on your graph of the data and check the fit. Then you may want to change your estimate and try another fit.

When you are satisfied with your values of T_1 and t_1 use them to calculate, via Equation (5.1.9), the diffusion constant $D = k/s$, the thermal conductivity k and the heat capacity $c = s/\rho$ where ρ is the density. Make an estimate of the error made in differential evaluation of the temperature change (Equation (5.5.5)) compared with actual temperature change. For doing this estimate, use the maximum change which can be measured using the amplifier circuit employed.

One further consideration can be applied to the data analysis. In deriving Equation (5.1.5) we assumed that the time during which the heater was on

(τ) was very small in relation to the time the heat takes to diffuse down the rod (t_1), ie, it was an impulse of heat (see Appendix F). In doing your experiments this approximation is valid as long as you make $t = 0$ on your graph correspond to the midpoint of the heating time and if the heating time is less than any t_1. Appendix G gives the details.

Exercise 5.5.2 Time shift of heat flow data

Shift the time scale of your heat flow data by $\frac{1}{2}\tau$ and again estimate T_1, t_1, D, s and k.

---

6 APPLE architecture and assembly language programming

Thus far, there has been no need to understand the inner workings of the computer in order to do useful experiments. It has been a black box which responds in a reliable way when given instructions. Just as in using a car, many times this is sufficient; however, to utilize its capabilities as a tool in the laboratory fully, the internal operation of the computer should be understood. In this chapter, we will look under the hood to explore the internal organization of the APPLE and to learn to program the 6502 microprocessor directly.

6.1 Inside the APPLE

A first glance under the cover of the APPLE shows a circuit board with a row of connectors which contain other circuit boards standing vertically. The horizontal board (the mother board) contains the 6502 microprocessor chip and various other chips which control the keyboard and screen and contain the memory cells. The microprocessor is the CPU which controls the system and executes the program instructions. The boards in the connectors perform a variety of other functions. Figure 6.1 shows the general

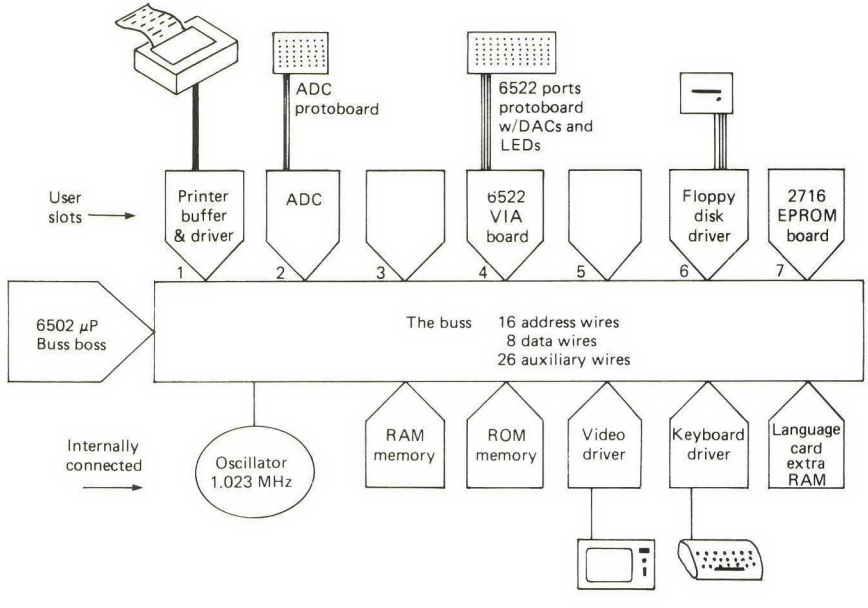


Fig. 6.1. Inside the APPLE IIe.

organization. The different chips and boards communicate with each other via the buss: a group of 50 wires which carry digital signals. There are 16 address lines, 8 data lines, and 26 auxiliary lines. The data lines contain the 8 bits of data which are to be transferred by the CPU. The bits on the address lines contain the binary number of the location from or to which the data will be transferred. The operation of the computer is, at the lowest level, a controlled transfer of bits of data among the various devices.

The CPU uses the auxiliary lines to control the data transfers. The READ/WRITE (R/W) line signals whether the data will be transferred to the CPU (a READ, R/W HI) or out from the CPU (a WRITE, R/W LO). Another auxiliary line is controlled by the oscillator which is the clock that determines how fast the CPU operates and thus, how fast the data transfers will take place. The one in the APPLE generates about one million cycles per second (Figure 6.2). At the beginning of each clock cycle, the CPU puts the binary bits defining an address on the address wires and sets the R/W line to indicate the direction of the data transfer. In the second half of the clock cycle the data transfer takes place on the data lines. The time lag between the first and second parts of the cycle is used by the memory circuits to locate the unique memory cell being addressed.

Since the 6502 CPU has 8 data lines, it is called an 8-bit microprocessor. These 8 binary bits (or 1 byte) can represent various things. They could be a binary data value, a machine instruction code (op-code) or one half of a 16-bit address. The electronic protocol for transferring the data is always the same no matter what the data may represent.

6.2 The 6502 microprocessor

Programs are ultimately stored in the computer as a series of data bytes. All programs written in other languages (eg, BASIC, FORTRAN, Pascal) are translated into this form (by another program!) before they can be executed. The machine language program is executed by the CPU by the following steps:

The CPU (1) reads the next instruction code (op-code) in the series,
(2) decodes the instruction,

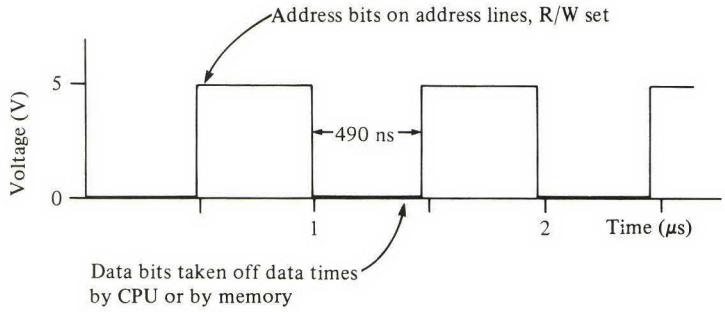


Fig. 6.2. 6502 Timing in the APPLE IIe.

(3) if necessary, reads additional data (such as an address or data byte),
(4) executes the instruction,
(5) starts at step (1) again.

In the 6502 each machine instruction requires 2–7 clock cycles to execute. There are approximately 150 different op-codes of which 56 are fundamental.

The 6502 CPU has six internal memory locations which it uses for executing instructions and to keep track of where it is in the program. These are called registers (Figure 6.3). The most sophisticated instructions are done with the eight-bit accumulator (A register). This is the arithmetic register whch can do addition and subtraction as well as logical operations. The X and Y registers are auxiliary eight-bit registers used principally for counting. The program counter (16 bits) contains the address of the next byte to be accessed in the program. The stack pointer contains the address of the top of the stack which is a group of memory locations used predominantly for executing subroutines. The 6502 uses the memory between $0100 and $01FF for the stack. You will learn more about how this works in Section 6.10. Notice that the high byte of the stack pointer is always $01; only the low byte varies. The last register is the process status (P or F, Figure 6.3) register which is used for a variety of housekeeping chores. Each bit in the register

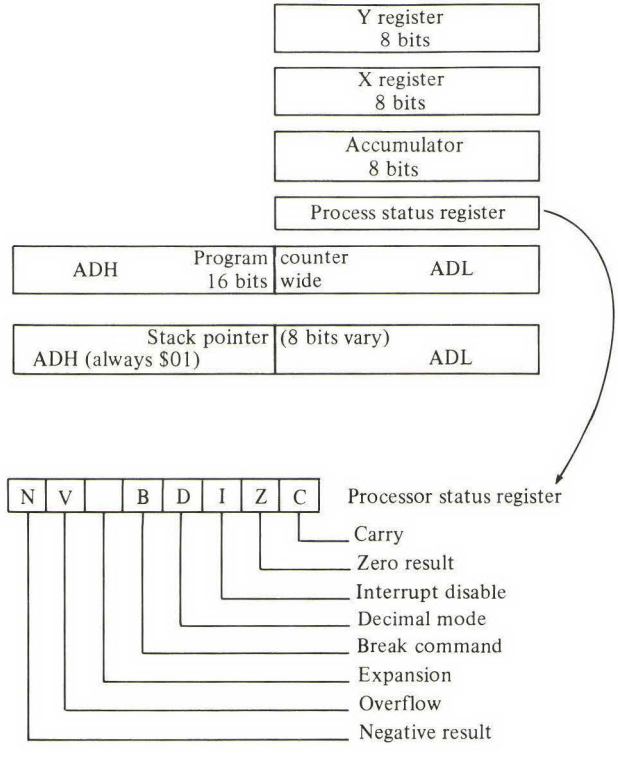


Fig. 6.3. 6502 Internal registers.

indicates a different state of the CPU. The branch machine instructions (Section 6.9) use the bits in this register to determine whether or not a branch should be taken.

6.3 Writing machine language programs

In the APPLE computer, short machine language programs are conveniently written using the 'MINIASSEMBLER' which is in the INTEGER BASIC package of programs loaded into the RAM in the language card when the machine is bootstrapped with the SYSTEM START DISK (the bootstrap procedure is described in detail in Appendix H). The MINIASSEMBLER makes it possible to write programs using mnemonics which directly represent the machine instructions to be executed by the CPU. A program written with these mnemonics is called an assembly language program. When using the MINIASSEMBLER the program is actually stored in machine language (ie, binary op-codes and data) but is displayed in assembly language (ie, mnemonic characters).

Starting with the APPLESOFT prompt ( ]) on the CRT screen, turn the printer on by typing PR#1 CR to get hard copy for your reports. Put the machine under control of INTEGER BASIC by typing INT CR; this will produce the prompt character of INTEGER BASIC, (>). Then type CALL-2458 CR; this will run the MINIASSEMBLER program which starts at memory location −2458., the MINIASSEMBLER program prompt is '!'. With the ! before you, type the program indicated below. SP means press the space bar; CR means press the RETURN key (the spaces marked with SP are mandatory, the other spaces are optional).

Program 6.3.1.

`9300:LDA #01 CR`	Load the accumulator (A) with the number $01.
`SP STA C402`	Store the contents of A in memory location $C402, (DDRB)
`SP LDA #01 CR`	Load A with $01.
`SP STA C400 CR`	Store in DRB.
`SP LDA #00 CR`	
`SP STA C400 CR`	Store 0 in DRB.
`SP JMP 9305 CR`	Jump to instruction coded in location $9305.
`$9300L CR`	Monitor command to list program steps starting at $9300.
`$9300G CR`	Monitor command to run program starting at $9300.

Figure 6.4 shows what the screen and printer should look like after this is completed. Appendix I contains useful information about assembly language programming.

Connect the oscilloscope to the line PB0; you should see square waves. The above program which you have typed into the machine and set running is the machine language equivalent of the BASIC program used to generate

Fig. 6.4. Assembly language program listing and comments.

```
]INT                                        Go to INTEGER BASIC
>CALL-2458                                  Go to MINIASSEMBLER

!9300:LGA#01                                Type in; start loc: instruction
                                            APPLE responds
9300-   A9 01      LDA   #$01              Type in space after instruction
! STAC402

9302-  8D 02 C4    STA   $C402             APPLE responds
! LDA#01

9305-  A9 01       LDA   #$01
! STA C400

9307-  8D 00 C4    STA   $C400             Etc
! LDA#00

930A-  A900        LDA   #$00
! STA C400

930C-  8D 00 C4    STA   $C400
! JMP 9305

930F-  4C 05 93    JMP   $9305             Type $9300L CR
!$9300L                                     – sends control to monitor
                                            9300L – disassembles
9300-  A9 01       LDA   #$01              starting at 9300
9302-  8D 02 C4    STA   $C402
9305-  A9 01       LDA   #$01
9307-  8D 00 C4    STA   $C400
930A-  A9 00       LDA   #$00              Disassembled program
930C-  8D 00 C4    STA   $C400             starting at 9300
930F-  4C 05 93    JMP   $9305
9312-  FF          ???                     Garbage
9313-  FF          ???
9314-  00          BRK                     9300L disassembles
9315-  00          BRK                     20 instructions
9316-  FF          ???                     9300LL would do 40
9317-  FF          ???                     9300LLL would do 60 etc.
9318-  00          BRK
9319-  00          BRK
931A-  FF          ???
931B-  FF          ???
931C-  00          BRK
931D-  00          BRK
931E-  FF          ???
!$9300G                                     Type $9300G CR
                                            $ sends control to monitor
                                            9300G starts program running
```

square waves in Exercise 3.10.1. Figure 6.5 shows the parallel statements between the two programs. To display the square waves set the SWEEP TIME/DIV on the oscilloscope to 5 μs/DIV. Note the time ratio between the machine language program and the BASIC program.

The lines you typed in with the $ as the first character are instructions to be executed by another set of subroutines in the APPLE called the monitor (prompt *). When the MINIASSEMBLER finds a $ as the first character of a line, it sends the rest of the line to the monitor for execution. The line $9300L instructs the monitor to go to memory location 9300 (hex) and to translate the machine codes into assembly codes which are then displayed alongside the machine code (Figure 6.4). This process of translating a machine language program into assembly language is called disassembly. $9300G starts the program execution at 9300 (hex). To stop the program which loops back to itself continually, press CONTROL-RESET.

It is informative to look at the details of this program as displayed to gain insight in the operation of the computer. Beyond the first seven lines the material is irrelevant; it is probably 'garbage' which the L instruction is trying to disassemble.

The first line, LDA #01, instructs the 6502 to load the hexadecimal number $01 into its accumulator. The program starts at memory location $9300, the number shown in the lefthand column of the first line of the program in Figure 6.4. In memory location $9300 the eight-bit number $A9 is stored.

When this program is started at memory location $9300, the 6502 retrieves the number $A9 from memory and decodes it as the instruction 'load the accumulator immediate' (mnemonic LDA). Immediate means that the accumulator of the 6502 is to be loaded with the number stored in the next memory location; in this case, the next address is $9301 and the number is $01. When the instruction $A9 is decoded by the 6502, it also knows that the instruction requires two bytes for its definition and thus the next instruction code will be found in the memory location $9302.

Fig. 6.5. Comparison of equivalent machine language, assembly language and BASIC programs.

Address	Machine code	Assembler code	BASIC statement
9300	A9 01	LDA #$01	10 POKE 50178,1
9302	8D 02 C4	STA $C402	
9305	A9 01	LDA #$01	20 POKE 50176,1
9307	8D 00 C4	STA $C400	
930A	A9 00	LDA #$00	30 POKE 50176,0
930C	8D 00 C4	STA $C400	
930F	4C 05 93	JMP $9305	40 GOTO 20

The next instruction STA $C402 means 'store the number which is in the accumulator, ($01) into memory location $C402'. The data is also retained in the A register. Memory location $C402 is the DDRB. This will set up DRB (memory location 50176 = $C400) with PB0 as an output port. When the 6502 retrieves the number $8D from memory it decodes it as the instruction to store the eight-bits of data in the accumulator in the memory location designated by the data stored in the next two memory locations, ie, $9303 and $9304. The least significant eight bits (low byte) of the address (eg, $02) are stored in the location just after the op-code and the most significant eight bits (high byte) of the address ($C4) in the memory location after this, $9304. This sequence is the protocol of the 6502 for storage of addresses; the low byte of the 16-bit address goes into the lower memory address and the high byte into the next higher address.

The above mode of memory addressing (eg, STA $C402) is called Absolute Addressing because the memory location upon which the instruction acts is explicitly designated. The instruction 'store accumulator with absolute addressing' is a three byte instruction; it requires three memory locations to completely specify the instruction. The memory address upon which an instruction acts is called its operand. The 6502 has about a dozen different ways of defining operands.

The next two instructions listed in the program put $01 into the accumulator from whence it is transferred into Port B. The next two instructions put $00 into Port B. The final instruction JMP $9305 makes the 6502 jump to memory location $9305 to find its next instruction, thereby looping the program interminably upon itself.

To examine but not disassemble the data stored in the memory defining the program above, type $9300.9311 CR. The result displayed on the CRT is shown in Figure 6.6. The above is called a memory dump; it is a simple tabulation of the data stored in the memory locations between $9300 and $931F. With this memory dump before you, take the time to make a step by step review of what occurs when the program is run from memory location $9300.

Saving machine language programs on the disk and retrieving them again can be done from the MINIASSEMBLER or BASIC. Machine language programs are saved as binary files. A binary file is simply a series of data bytes stored on the disk; this sequence may represent a variety of things: data

Fig. 6.6. Memory dump.

```
!$9300.9311

9300- A9 01 8D 02 C4 A9 01 8D
9308- 00 C4 A9 00 8D 00 C4 4C
9310- 05 93
```

Type $9300.9311 CR
9300.9311 is a monitor instruction to dump memory contents from 9300 to 9311 inclusive

bits, a machine language program, a CRT graphics image. They are different from TEXT files which are a sequence of encoded character strings or from APPLESOFT files which are encoded BASIC instructions.

To save a machine language program use:

BSAVE filename, A address, L length

where 'address' is the address of the start of the program and 'length' is the number of bytes you wish to save. These are specified by a decimal number or alternatively using a hexadecimal number preceded by a $. For example

BSAVE EX6.0.0, A$9300, L$11

will save 17 bytes starting at memory $9300 with the filename 'EX6.0.0'. To retrieve a binary file use

BLOAD filename, A address

If you leave off the A parameter, the binary file will be loaded into the location from which it was saved. Otherwise it will be put in the memory starting at the specified address. To use BLOAD as a DOS command within a BASIC program use the instruction PRINT CHR$(4) 'BLOAD filename'.

Exercise 6.3.1 Machine language square waves and BSAVE

(*a*) Write, run, and print out a machine language program which produces square waves at PB4. Examine and record the signals on your oscilloscope. BSAVE and BLOAD the program.

(*b*) Run the program DEMO2 on the AMPERGRAPH disk or one of your own programs which will quickly produce a graph. Data for page 2 of high resolution graphics are stored in the memory locations from $4000 to $5FFF. BSAVE this page of graphics on a disk. Write a BASIC program to BLOAD and display the file. Be sure to set the display to HGR2.

6.4 Operation of a DAC

The purpose of this section will be to explore the use of Digital to Analog Converters (DAC) and to get some practice in assembly language programming. You also will learn how to instruct the computer to go back and forth between BASIC and machine language programs.

You have used an ADC in the previous sections to convert an analog voltage signal external to the computer into a digital signal which the computer can manipulate and store. The inverse operation is done with a DAC. The DAC is an output device which converts the binary number to an analog voltage. They can be used for a variety of purposes. For example, they are used as the output devices for digital music playback and for digital video players. You will use them to drive oscilloscope displays.

There are two DACs connected to the APPLE parallel interface, Figure 6.7; one is on Port AA and one is on Port BB of the second 6522 VIA (base address $C480). They are used by setting up the ports as output and then writing digital numbers into the ports.

For electronic reasons which need not concern us, the DAC you are using uses an inverted representation of numbers, ie, the binary number $00 at its input produces +5 V at its output and $FF at its input +0 V at the output. To generate the conventional conversion between analog voltage and binary

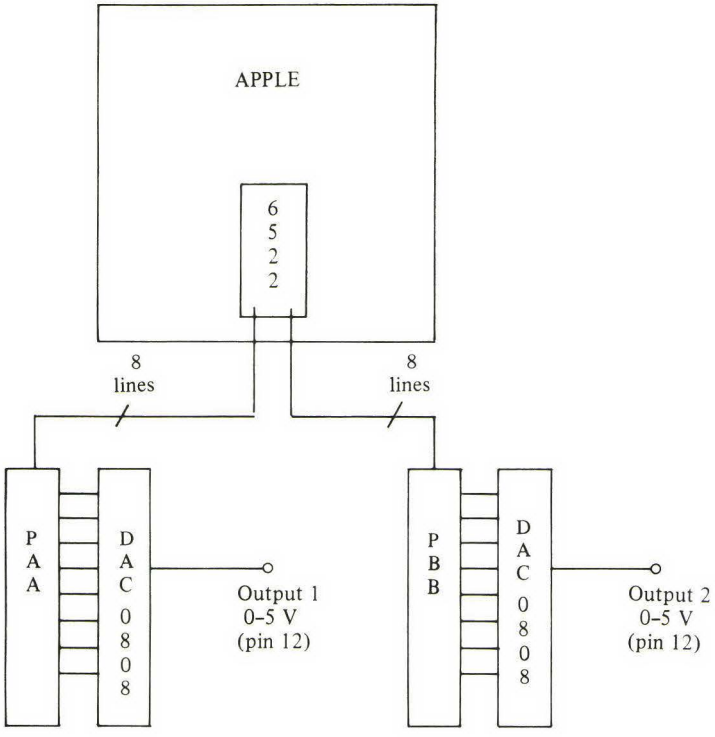
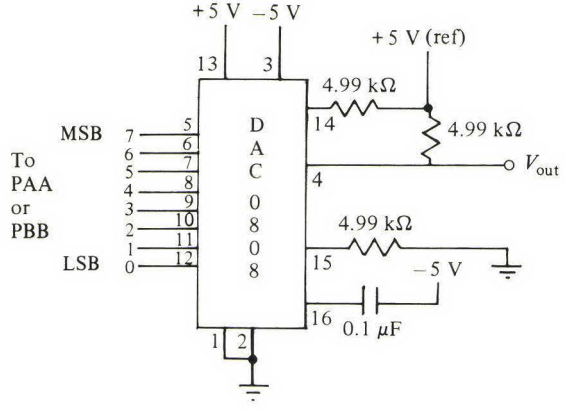


Fig. 6.7. DAC circuits.



numbers the binary numbers at the output to the DAC should be inverted, ie, all ones converted to zeroes and zeroes to ones.

Exercise 6.4.1 DAC sawtooth wave (BASIC)

Using BASIC and Port AA, write a program which will set up the port for output and write the temporal sequence of numbers \$00, \$01, . . ., \$FF, \$00, . . . *ad infinitum* into the port. Observe the output of the DAC with the oscilloscope and note the results. The output is on pin 4 of the DAC chip.

Exercise 6.4.2 DAC sine wave (BASIC)

Write a BASIC program which will output a sine wave from the DAC. Take note of the fact that the sine function goes from −1 to +1; this must be put in a digital range from 0 to 255 for the DAC. Display the sine function in two ways: (*a*) by calculating the sine each time it is needed, (*b*) by using a lookup table. In (*b*) a table (array) of 100 sine values is calculated once and then, when the program needs a value, it is obtained from the array. Observe with the oscilloscope and note the difference in speed of the two methods of programming.

6.5 Indexed addressing

Before proceeding to the use of DACs with assembly language programs, two more assembly language concepts need to be understood; these are indexed addressing and program branching.

A BASIC program sequence to move the data from one area of memory to another is

```
10   BA=36864 : BB=37120
20   FOR I=0 to 99
30   A=PEEK (BA+I)
40   POKE BB+I, A
50   NEXT I
60   END
```

These instructions transfer an array of values from memory locations *BA*,*BA* + 1, *BA* + 2, . . ., *BA* + 99 to *BB*,*BB* + 1, . . ., *BB* + 99. An assembly language program can do the same thing much more quickly using indexed addressing. A program to do this is:

```
9300   LDX   #$00      Get 0 into the X register.
9302   LDA   $9000,X   Get the data from address $9000+X
                        into A.
9305   STA   $9100,X   Store the data in A at address
                        $9100+X.
```

```
9308   INX          Add 1 to the X register.
9309   CPX  #$64    Compare (calculate X – $64) the
                    number in the X register with $64
                    ($64 = 100).
930B   BNE  $9302   If X – $64 is not equal to zero, go back to
                    $9302.
930D   BRK          Stop execution.
```

There are several new instructions here. Read the comments on the right thoroughly to understand how they work. LDX *#xx* (immediate) is similar to LDA *#xx*; it will load the X register with the value of the byte which follows. LDA *$xxxx*,X indicates indexed addressing. This instruction will 'load A with the data which appears at the address computed by adding the value in the X register to the address *$xxxx*.' In the above program if X contains the value $1C, the 6502 will load the data from address $901C ($9000 + $1C) into A. Notice one limitation – since the X register can only range from $00 to $FF (eight bits) the range of addresses which can be 'indexed' is limited to those within 255 from the base address. STA $9100,X operates in an analogous manner. INX stands for 'increment X'; it adds 1 to the X register.

The next instruction, CPX #$64, is 'compare X with the next byte' (notice the immediate mode addressing indicated by #). In this case $64 is subtracted from X and the flags (the data bits in the processor status register) are set according to the result. The N bit is set to 1 for a negative result, Z is set to 1 for a zero result; otherwise, these flag bits go to zero. The last instruction, BNE $9302, tests the Z flag. BNE stands for 'branch if the previous result was not zero'. (More on branching in section 6.9.) Thus the computer will jump to $9302 if Z is zero. Since the state of the Z flag is set by the CPX instruction, the result of these last two instructions is that the program will loop (branch) back to $9302 if X is not equal to $64. Thus, as the program does each loop, the X register increments until it gets to $64, then the branch is not taken and the computer goes on to the break instruction at $930D and stops. The net result is the same as the BASIC program shown before.

Exercise 6.5.1 DAC output in machine language

By using indexed addressing, the BASIC program which you wrote for Exercise 6.4.1 can be translated into assembly language. The procedure is as follows: first, write a BASIC program which POKEs the numbers 0–99 into memory $9000–$9063. These will be the 'data' to be output to the DAC. Next, go to the MINIASSEMBLER and enter a program starting at $9300 which will read the data bytes from address $9000 to $9063 and output them to the DAC. This program will be very similar to the one described above. Add a JMP

instruction at the end which will loop back to the beginning of the program. Then run the machine language program with $9300G, and observe the results with the oscilloscope.

6.6 The CALL and RTS connection

A few embellishments will make the operation of the BASIC–machine language system smoother. Once the machine language program is in the memory, it can be used by a BASIC program through the use of the CALL statement. When BASIC executes a 'CALL address' statement it jumps to the address given and begins to execute the instruction it finds there as a subroutine. To then return to BASIC from the machine language subroutine, the machine language instruction 'return from subroutine' RTS is used. The last instruction of every subroutine is RTS. (More on this later.)

Exercise 6.6.1 BASIC—machine language connection

Try using the assembly language program you wrote in Exercise 6.5.1 in a BASIC program. First go to the MINIASSEMBLER and replace the JMP instruction at the end with RTS. Then go to BASIC and enter and run the following.

```
200   CALL 37632
210   GO TO 200
```

Watch what happens with the oscilloscope and explain the qualitative shape of the waveform.

Exercise 6.6.2 DAC sine wave (BASIC and machine language)

Write a BASIC program which calculates a sine wave table (array) whose amplitude varies between 0 and 255 and which is stored in $9000–$9063 (100 values) and then uses a CALL to a machine language program to show the results on the oscilloscope. Try varying the frequency of the calculated sine wave and observe the effects.

6.7 An X–Y plotter

By using two DACs and the oscilloscope you can make an X–Y plotter, that is a display whose X value is determined by one function and whose Y value by another function of the same parameter. The oscilloscope will display the two input channels in this way if you set the MODE to 'X–Y'. As an example of X–Y plotting, suppose the x axis voltage varied as $\cos(\theta)$ and the y axis voltage varies as $\sin(\theta)$, what would be the figure traced out as successive points were plotted ($\theta = \theta_1, \theta_2, \ldots$)?

Exercise 6.7.1 Lissajous figures on a DAC X–Y plotter

Use the two ports and DACs to plot Lissajous figures. Program the calculations in BASIC and the display output in assembly language.

(*a*) Begin with the simplest figures, a circle:

$$x = \cos(\theta),\ y = \sin(\theta)$$

and a line:

$$x = \cos(\theta),\ y = \cos(\theta)$$

(*b*) Next try:

$$x = \cos(\theta_1),\ y = \sin(\theta_2)$$

where

$$\theta_1 = 2\theta_2 \quad \text{or} \quad \theta_2 = 2\theta_1$$

(*c*) What happens when you vary the relative phase or amplitude of x and y? For example, try a circle again but with

$$x = \cos(\theta),\ y = \tfrac{1}{2}\sin(\theta)$$

then

$$x = \cos(\theta + \tfrac{1}{4}\pi), \quad y = \tfrac{1}{2}\sin(\theta)$$

6.8 Boolean algebra

Normal algebraic variables can take on an infinity of values and are added, subtracted, multiplied, etc. to give new values. Boolean variables are quantities which can take on only two values and are operated upon by AND, OR, NOT, etc to give new values. The two values can be described by 0 and 1, HI and LO, or true and false. (No $\frac{1}{2}$, MIDDLE, or maybe.) The AND operation combines two Boolean variables A and B to produce a third Boolean variable C such that C is 1 if, and only if, both A and B are 1. The AND operation between two Boolean variables is represented by $\wedge$ or by a dot,

$$C = A \cdot B \quad \textit{or} \quad C = A \wedge B$$

Boolean algebra statements are frequently defined by truth tables. Table 6.1 shows the AND operation

Table 6.1 *Truth table for the* AND *operation*

A	B	$C = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

The 6502 has an instruction AND which does exactly this. Each of the eight data bits is considered as a Boolean variable. The AND instruction performs the AND operation between each of the corresponding bits in the accumulator A and some memory location and deposits the result in the accumulator. In the table of instruction codes of Appendix I, and AND operation is written $A \wedge M \rightarrow A$.

An important application of the AND instruction is to help determine whether some bit in a byte is a 0 or a 1. The first step is to isolate the bit and to produce a result dependent upon the value of the bit in the place being tested. This operation is called masking. It is as though we hide the bits of no concern behind a mask and look through a hole in it at the one of interest. The result is independent of the value of the other data bits. The program steps

```
LDA #10
AND $9405
```

will isolate DB4 (Data Bit 4) of the contents of memory location $9405. The AND operation between the 1 loaded into DB4 of the accumulator by the LDA#10 operation and the data in memory location $9405 will produce a 1 in DB4 of the accumulator if DB4 of $9405 is a 1 and 0 if it is a 0. All other data bits of the result in the accumulator will be zero because 0 AND 0 = 0 and 0 AND 1 = 0.

Exercise 6.8.1 AND

To see how this works, use the MINIASSEMBLER to write the machine language program code for the following program starting at memory location $9300.

```
$9300    LDA 9400
         AND 9401
         STA 9402
         RTS
```

Store some random numbers into $9400 and $9401 by typing the appropriate monitor commands from the MINIASSEMBLER. To do this, with the ! prompt before you, type

$9400:D3 5F CR

to store $D3 in $9400 and $5F in $9401. To verify that these are the numbers stored, type (with the prompt ! on the monitor):

$9400.9401 CR

This will display the contents of memory locations from $9400 through $9401 (which is of course just $9400 an $9401). In general, these two numbers could be the beginning and end of any interval.

Store some selected hexadecimal numbers in $9400 and $9401,

run the program and then display the results in locations $9400–$9402. Write out the hexadecimal numbers in binary to demonstrate how AND worked between the two starting numbers. Choose two starting numbers so the entire AND truth table between them can be verified.

The Boolean algebra operation conjugate to AND is OR, which given two Boolean variables A and B, will produce a Boolean variable that C which is 1 if A or B is 1. The OR operation is written

$$C = A + B \quad \text{or} \quad C = A \vee B$$

It is defined by the truth table, Table 6.2.

Table 6.2 *Truth table for the* OR *operation*

A	B	$C = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Exercise 6.8.2 ORA

Rewrite the program in Exercise 5.8.1 using the ORA instruction ($A + M \rightarrow A$) in place of the AND instruction. As above, run the progam and write out in binary form the resulting byte in $9402 and note the relationship between them and the bits you started with in $9400 and $9401. Put the data into $9400 and $9401 which verify the entire truth table.

The third and final Boolean algebra instruction in the 6502 is exclusive OR (EOR). It works on each bit like the AND and OR operations. EOR is written

$$C = A \oplus B \quad \text{or} \quad C = A \veebar B$$

The rule for exclusive OR between A and B is that C will be HI only if either A is HI or B is HI.

Table 6.3 is the truth table for EOR.

Table 6.3 *Truth table for the* EOR *operation*

A	B	$C = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

An important application of EOR is to invert one or more bits in a memory location and leave all the others alone. For example you could produce square waves on PB7 at the same time that the other lines on this port are used for other applications. This inverting property of EOR, that $1 \oplus DB = \overline{DB}$ and $0 \oplus DB = DB$; is easily derived by inspection of the truth table above. ($\overline{DB}$ means DB inverted or 'NOT DB' thus, if DB = 1 then $\overline{DB} = 0$ and if DB = 0 then $\overline{DB} = 1$.)

Exercise 6.8.3 EOR

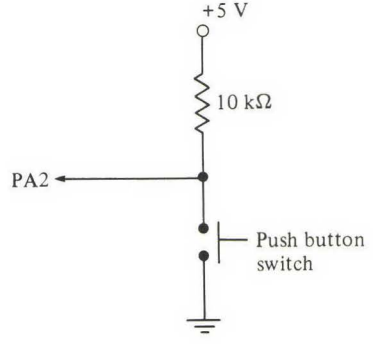
Write a program which inverts DB6 of the data stored in the memory $9401 and leaves the rest of the bits alone. Run it and demonstrate this property by displaying and printing out the contents of $9401 before and after running with several initial values.

6.9 Branching instructions

The 6502 has a group of instructions called branching instructions. They test the byte obtained in a previous operation for various conditions: if the result of that test is true, the program will continue execution at some other location. The BNE instruction in the program in Section 6.5 tested whether or not the result of the CPX instruction was equal to zero; if not, the branch to $9302 was taken and program execution continued from there. If the test result is false, the program counter advances sequentially, as it would in the absence of the instruction. Thus, branching instructions are similar to the IF . . . GOTO statement in BASIC.

To demonstrate how this works wire up the circuit shown in Figure 6.8. Before doing so it is important that you make sure that Port A is set up for INPUT which means that all the data bits in data direction register A be set equal to zero. A quick way of doing this is to simply press CONTROL RESET. To protect electronic components, the 6522 sets all its control registers, eg, DDRA and DDRB to zero whenever the power is turned on or the machine is RESET. If it were otherwise, the possibility, indeed probability, would be present of both the 6522 and the switch which you are installing trying to control data lines leaving the machine. This can lead to a 'short circuit' since the 6522 may try to connect a data line to +5 V, at the same time that the switch which you installed connects to ground. This condition can lead to burned out components so be careful to avoid it.

Fig. 6.8. Push button circuit.



Exercise 6.9.1 Masking and branching

Enter and run the program indicated in Figure 6.9. In formulating programs it is usually easier to write a mnemonic memory location like DRA for $C401 (which is what you type into the MINI-

Fig. 6.9. Masking program.

```
AGAIN   LDA  #04        AGAIN = memory location of
        AND  DRA        beginning of program
        BNE  AGAIN
        BRK

                        DRA = $C401
```

ASSEMBLER) and a mnemonic like AGAIN to indicate the step where the program is to loop upon itself. Then do the proper translation of mnemonics as you use the MINIASSEMBLER. This program will continuously loop back upon itself waiting until you press the switch which you have connected to the PA2. When you do, it will go on to the BRK instruction and stop execution. The APPLE monitor will display the contents of the machine registers, ie, A,X,Y, P and S, together with the value in the program counter when the BRK was encountered.

The 6502 has eight instructions like BNE which branch as the result of a test. These instructions actually test one of four of the bits in the process status register. They are the carry bit (C), the zero bit (Z), the overflow bit (V), and the negative bit (N). These bits are set as a result of what happened in the processor during a previous operation. For example, if the AND operation produced zeros in all eight data bits, the zero bit in the P register would be set to 1. If it did not produce all zeros, the Z bit in the P register would go to 0. If DB7 is 1 as the result of an operation, the N bit in the P register gets set to 1; if the operation produced a 0 in DB7 the N bit goes to 0. Each of the branch instructions tests one of these flag bits (Z,N,C,V) for a 0 or a 1. The effect which each machine instruction has on the P register flags is shown by checks (√) in the right hand column of the 6502 instruction list in Appendix I. Some instructions do not affect the P flags; this is indicated by a dash (–).

An important point to note is that a scheme of relative addressing is used by the 6502 CPU in executing branch instructions (and only branch instructions!). In typing the branch instruction for the switch circuit above into the APPLE you typed BNE followed by the memory location ($9300) where the instruction was to be found if the branch is taken. Look at the code generated and note that in memory location $9305, the number $D0 is stored which is the op-code for BNE. In the following memory location the number $F9 is stored in response to your instruction that the branch go to $9300 if the BNE test is true. If the test is true the 6502 will take the value of the number stored in the location following the branch op-code and add it to the current value of the program counter if the branch is to be taken. Then execution resumes

at that calculated address. To demonstrate this, take the number stored in $9306 and add it to the low byte of the location where the program counter will be if the branch is not taken, ie, $07. You can do this with paper and pencil or using a hexadecimal adding routine which is in the monitor. From the MINIASSEMBLER type $07 + F9 CR. The result is $00, which, together with the high byte of the program counter $93, gives the address $9300 which the CPU will use to find its new instruction if it has to take the branch. The high byte of the offset is assumed to be $FF if the offset is negative.

This scheme of relative addressing has the important consequence that program codes which have branches are intrinsically relocatable in memory. It has the drawback that branches can be taken which are no more than 128 memory locations earlier in the program and not more than 127 steps further on since the largest negative eight-bit number $80 is −128 and the largest positive eight-bit number $7F is 127. In practice this does not cause serious restrictions for short programs.

6.10 Subroutines and use of the stack

Another program branching capability which every computer must have is that of executing subroutines. A subroutine is a sequence of program steps that can be used anywhere in a program by a jump to subroutine (JSR) instruction.

To execute a subroutine, the computer stops fetching instruction op-codes sequentially from memory, jumps (JSR) to the memory location indicated and from there continues fetching instructions until a return from subroutine (RTS) instruction is encountered. It then returns to the original program and resumes fetching instructions in sequential progression where it left off when the subroutine was called. This process is illustrated in Figure 6.10.

In order for the computer to return to the correct place in the calling program, the memory location of the next op-code after the JSR in the calling program needs to be saved. When the instruction JSR is executed, the 6502 stores the memory location of the next op-code after the JSR instruction on the top of the stack. This operation is analogous to writing the return address on a card and placing it on top of a pile. The last instruction of every subroutine is RTS which means return from subroutine. This instruction effectively takes the top card from the pile, reads the return address, puts that location into the program counter and then throws the card away.

The idea of using a stack (the pile of reminder cards) to store addresses, may seem like a tortuous way of doing things. It is, however, an invention which was very important for the development of modern computers.

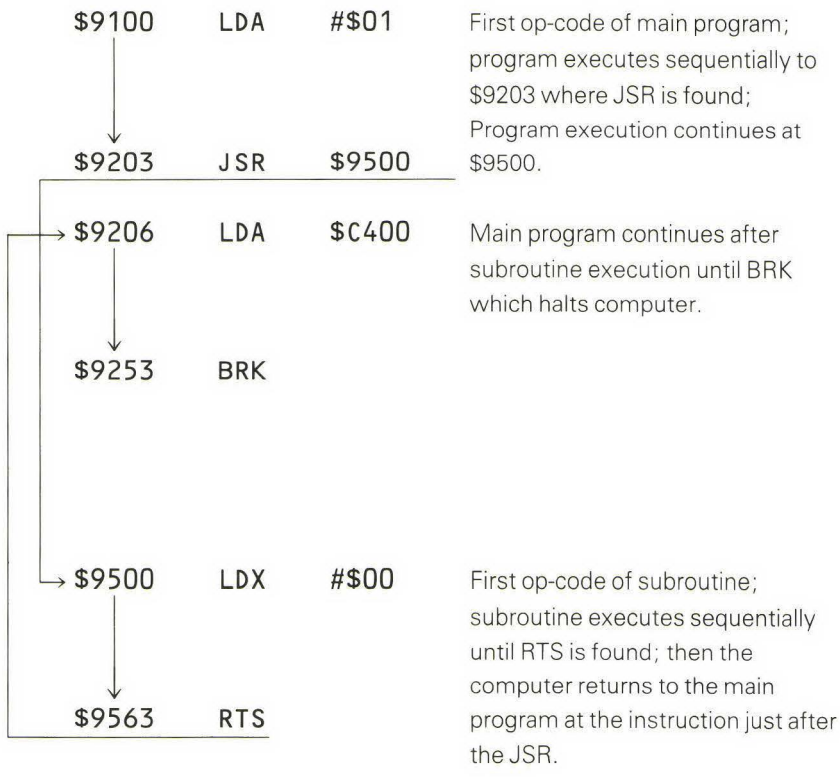


Fig. 6.10. Subroutine execution sequence.

Without something like a stack it is not possible to use ROM to store subroutines. It is said that one of the most awkward things about using the first successful minicomputer, (the DEC PDP8), was that it did not have a stack. The stack is akin to the Reverse Polish Notation used by Hewlett Packard calculators. The last item stored in the stack is the first to be retrieved. In addition to storing the return address for the subroutine, the stack is sometimes used (with care and understanding) to pass numbers from a calling program to a subroutine.

As mentioned in Section 6.2 the memory locations of the stack on the 6502 are those memory locations on page 1 of memory, ie, those memory locations with addresses between $0100 and $01FF. The stack pointer is a 16-bit register in the 6502 which contains the address of the top of the stack. There, after completion of the subroutine, the CPU will find the memory address to which it must return program control. The bookkeeping of the stack is quite automatic in the CPU. From the programming point of view the only thing which you must be sure to do, is to have an RTS instruction for every JSR instruction.

Exercise 6.10.1 JSR

Write a machine language program which starts at $9100 and waits for you to push a switch which is wired to PB3, as shown in Figure 6.8. When the switch closes the machine language routine should call (JSR) the monitor subroutine at address $FBE4 which produces a 0.1 s BEEP. CALL the program from a BASIC program which then prints 'THE BELL RANG' on the CRT monitor after the subroutine has been completed.

6.11 Assembly language timing loops

Frequently it is necessary to estimate the time required for a program to run and to write simple time delay programs in machine language to wait for some event. Time delay loops written in BASIC are not precise because of the way in which the BASIC interpreter functions. Most programs written in machine language run with well-defined and with easily calculable execution time. A copy of the 6502 instruction set with the execution time in number of machine cycles of each instruction is shown in Figure 6.11.

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
ADC		2	3	4		4	4*	4*			6	5*	
AND		2	3	4		4	4*	4*			6	5*	
ASL	2		5	6		6	7						
BCC										2**			
BCS										2**			
BEQ										2**			
BIT			3			4							
BMI										2**			
BNE										2**			
BPL										2**			
BRK													
BVC										2**			
BVS										2**			
CLC									2				
CLD									2				
CLI									2				
CLV									2				
CMP		2	3	4		4	4*	4*			6	5*	
CPX		2	3			4							
CPY		2	3			4							
DEC			5	6		6	7						
DEX									2				
DEY									2				
EOR		2	3	4		4	4*	4*			6	5	
INC			5	6		6	7						
INX									2				
INY									2				
JMP						3							5

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect, X)	(Indirect), Y	Absolute Indirect
JSR						6							
LDA		2	3	4		4	4*	4*			6	5*	
LDX		2	3		4	4		4*					
LDY		2	3	4		4	4*						
LSR	2		5	6		6	7						
NOP									2				
ORA		2	3	4		4	4*	4*			6	5*	
PHA									3				
PHP									3				
PLA									4				
PLP									4				
ROL	2		5	6		6	7						
ROR	2		5	6		6	7						
RTI									6				
RTS									6				
SBC		2	3	4		4	4*	4*			6	5*	
SEC									2				
SED									2				
SEI									2				
STA			3	4		4	5	5			6	6	
STX*			3		4	4							
STY**			3	4		4							
TAX									2				
TAY									2				
TSX									2				
TXA									2				
TXS									2				
TYA									2				

* Add one cycle if indexing across page boundary
** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

Fig. 6.11. Machine instruction execution times (from MOS Technology Microcomputer Programming Manual). The numbers are the machine cycles needed to execute the instruction.

Fig. 6.12. Time delay program.

```
                          Number of clock cycles
        LDX #12                    2
MORE    DEX                        2
        BNE MORE                   3
```

The execution time is five cycles for each circuit of the loop.
$12 = 18.$ times around loop so $18. \times 5. = 90$ μs delay.
Total time for program $90 + 2$ μs $= 92$ μs.

To compute the time required for a program to run we simply note the number of machine clock cycles required of each instruction and add them up. A time delay program is given in Figure 6.12 together with a computation of the number of clock cycles required for it to run. To get longer delays it is easy to write programs with multiple nested loops or using multiple precision addition. An example showing the use of double precision addition is given in Figure 6.13.

Fig. 6.13. Longer time delay program.

```
Number of cycles          Instruction
      2                   LDA #00      Initialize sum low and sum high
      4                   STA SL
      4                   STA SH
     10

      2         AGAIN     CLC
      2                   LDA #01      Add 1. to SL and update SL
      4                   ADC SL
      4                   STA SL

      2                   LDA #00      Double precision add: #00 and
      4                   ADC SH       carry added to SH
      4                   STA SH

      4                   CMP TH       Compare SH to TH, if not equal
      3                   BNE AGAIN    add one more
     29 or 28

      4                   LDA SL       Compare low bytes, SL and TL
      4                   CMP TL
      3                   BNE AGAIN
     11
```

10 μs for init
TH $\times$ 256 $\times$ 29 μs for main loop
TL $\times$ (28 + 11) μs for final loops

Exercise 6.11.1 Machine language timing loops

Write a program using triple precision addition to generate a time delay of 5 s. After each 5 s interval, add one count to the LED display connected to PBB. With a watch measure the time for the contents of PBB to increment to check that your program is correct.

6.12 Indirect addressing

Although you will not have occasion to use indirect addressing in this course it will be discussed briefly because it is often used in assembly language programs.

Indirect addressing is an addressing mode which can be contrasted with absolute addressing that you have already used. The instruction written JMP $9500 (jump absolute) means jump to memory location $9500 and continue program execution with the op-code found there. The indirect instruction written as JMP ($9500) (jump $9500 indirect) instructs the CPU to look in memory locations $9500 and $9501 to find the low and high parts of the address where the next op-code is to be found and from which subsequent program execution is to be continued. The jump indirect instruction is useful for jumping to different parts of a program depending on previous program steps. The previous steps may for instance, write a new address into $9500 and $9501. In general, writing parentheses around a memory address in an assembly language program means 'the contents of'. With the exception of the JMP indirect instruction, the indirect addressing modes of the 6502 are limited to indirect addressing from memory locations lying between $0000 and $00FF.

7 Viscosity measurement

A solid body moving through a fluid has a force pushing on it which depends on the type of fluid. You might imagine that it would be much harder swim in honey than it is in water. The parameter which describes this difference is the viscosity (μ). The drag force also depends upon other parameters much as the surface area of the body and the fluid density, as you will discover in this chapter. The computer will be programmed to measure the speed of a sphere falling through glycerine and to calculate the viscosity. The measurements are made with photosensors and using machine language programming. A short section at the end of the chapter describes the use of an EPROM to record semipermanently a machine language program.

7.1 Force required to move a solid body through a fluid

In this section the physics of a sphere moving in a fluid will be discussed. There are two distinct regimes; if the sphere is moving slowly, the dominant force resisting its motion is due to viscosity. For rapid movement, the inertial resistance of the fluid due to its density is the dominant factor. The magnitude of the resistance and the functional dependence on sphere size, velocity, fluid density and viscosity can be estimated in a rough way for both cases. This gives insight into how the drag force behaves without getting lost in the mathematics. Indeed, with turbulent phenomena exact computations have not been possible.

Viscous resistance of a fluid arises from shear in the velocity profile of flow. If two flat plates have fluid between them, as shown in Figure 7.1, a force is required to move the top one at a constant speed in relation to the bottom one. The force is proportional to the area of the plate and, (if the fluid is characterized by a Newtonian viscosity coefficient), to the relative velocity and inverse distance between plates, ie, to the velocity gradient dv_z/dx.

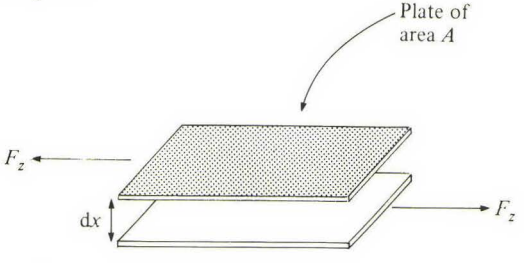


Fig. 7.1. Drag force of a fluid on thin plates, $F_z = -A\mu(dv_z/dx)$. For a 'Newtonian' fluid, the shear force per unit area is proportional to the shear in the velocity, dv_z/dx. The viscosity μ is the proportionality constant.

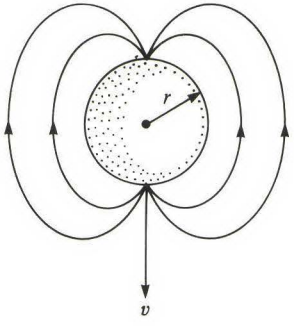


Fig. 7.2. A sphere falling slowly in a fluid, the fluid flow to move fluid from front of sphere to rear extends to about r away from sphere. So, $dv_z/dx \approx v/r$ and $F \approx 4\pi\mu rv$ with $4\pi r^2$ as the area of sphere.

Without doing elaborate computations this simple concept can be used to estimate the viscous resistance of a falling sphere. The effective area of velocity shear is more or less the area of the sphere, $4\pi r^2$ (Figure 7.2). The velocity perturbation resulting from moving the ball through the fluid extends to a distance about equal to the radius of the sphere; thus, the velocity gradient, dv_z/dx, which enters into the viscous drag relation is approximately v/r. Putting these two rough estimates together, an estimate of the viscous drag F_v on the sphere is

$$F_v \approx 4\pi r^2 \mu v/r = 4\pi\mu rv \tag{7.1.1}$$

where μ is the viscosity of the fluid.

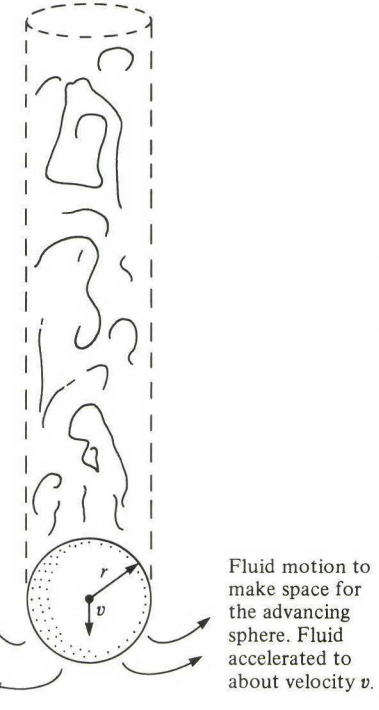
This problem is amenable to exact mathematical analysis; it was first done by Stokes and the relation is known as Stokes' law for the viscous resistance of a sphere moving in a fluid. His result† is

$$F_{\text{Stokes}} = 6\pi\mu rv \tag{7.1.2}$$

Stokes' law is verified experimentally for cases when the sphere's motion is sufficiently slow. The approximate approach used above gives important insight into the physical origin of the Stokes' formula.

More rapid motion leads to a turbulent wake behind the sphere. Though mathematical computation of the drag force in this regime has not been done, relatively simple ideas give a good estimate of the force observed. To move an object rapidly, the speed of the fluid in the path of motion is accelerated from zero to the speed of the sphere and the fluid is pushed aside and then forms a turbulent wake behind the sphere. The turbulence eventually dissipates the kinetic energy of the moving fluid as heat and sound energy without giving any kinetic energy back to the sphere. The drag force on the sphere will be equal to the force required to push the fluid out of the way.

Fig. 7.3. A sphere falling with velocity v and a turbulent Wake. The fluid is accelerated to about velocity v. The volume of fluid displaced each second is $\pi r^2 v$, the cross-sectional area A is πr^2.



Fluid motion to make space for the advancing sphere. Fluid accelerated to about velocity v.

An estimate of the mass of fluid moved per unit time is the mass of the column of pushed aside fluid each second as the sphere falls. This is the product of the cross-sectional area A of the object perpendicular to the direction of motion, the velocity of motion v, and the density ρ of the fluid (Figure 7.3). A maximum guess is that each element of this column is accelerated to the velocity of the moving object by the presssure exerted on the front face of the object.

Therefore the work done by the drag force on the sphere (force × distance) is equal to the kinetic energy of the fluid ($\frac{1}{2}$ × mass of fluid moved × v^2).

$$(F_{\text{est}})(v\Delta t) = \tfrac{1}{2}(\rho_f \pi r^2 v\Delta t)v^2$$

Thus

$$F_{\text{est}} = \tfrac{1}{2}\rho_f v^2 A \tag{7.1.3}$$

is the *estimated* drag on the sphere where A is the cross-sectional area.

† See, for instance, *Geodynamics: Applications of Continuum Mechanics to Geological Problems*, D. L. Turcotte & G. Schubert, Wiley, New York, 1982.

The drag resistance of a blunt object in terms of an *experimentally* determined drag coefficient C_d is by definition

$$F_{drag} = C_D A \rho v^2/2 \tag{7.1.4}$$

The combination $\rho v^2/2$ is called the kinetic pressure of a fluid. The experimentally determined drag coefficient for a sphere moving rapidly through a fluid is $C_d = 0.5$. As you can see, Equation (7.1.3) over estimates the drag on a sphere by a factor of 2. Drag coefficients for other shapes are given in Figure 7.4.

Combining the Stokes relation with the turbulent force gives the total drag force on the falling object as

$$F_{tot} = 6\pi\mu r v + C_D \pi r^2 \tfrac{1}{2}\rho v^2/2 \tag{7.1.5}$$

As Equation (7.1.4) shows, the turbulent drag for a sphere is proportional the square of the velocity; therefore, it is the dominant phenomenon at high velocity whereas viscous drag is more important for a slowly moving sphere.

Fig. 7.4. Experimental drag coefficients (from p. 11.68 of *Mark's Standard Handbook for Mechanical Engineers*, ed. T. Beaumeister, 8th edn, McGraw-Hill, New York, 1978 – used by permission).

Table 4. Drag Coefficients

Object	Proportions	Attitude	C_D
Rectangular plate, sides a and b	$\frac{a}{b}$ = 1, 4, 8, 12.5, 25, 50, ∞	V, b, a	1.16, 1.17, 1.23, 1.34, 1.57, 1.76, 2.00
Two disks, spaced a distance l apart	$\frac{l}{d}$ = 1, 1.5, 2, 3	V, l, d	0.93, 0.78, 1.04, 1.52
Cylinder	$\frac{l}{d}$ = 1, 2, 4, 7	V, l, d	0.91, 0.85, 0.87, 0.99
Circular disk		V, d	1.11
Hemispherical cup, open back		V, d	0.41
Hemispherical cup, open front, parachute		V, d	1.35
Cone, closed base		V, d	α = 60°, 0.5 α = 30°, 0.3

Drag Coefficients of Various Bodies

For **bodies with sharp edges** the drag coefficients are almost independent of the Reynolds number, for most of the resistance is due to the difference in pressure on the front and rear surfaces. Table 4 gives $C_D = D/qS$, where S is the maximum cross section perpendicular to the wind.

For **rounded bodies** such as **spheres, cylinders,** and **ellipsoids** the drag coefficient depends markedly upon the Reynolds number, the surface roughness, and the degree of turbulence in the air stream. A sphere and a cylinder, for instance, experience a sudden reduction in C_D as the Reynolds number exceeds a certain critical value. The reason is that at low speeds (small Re) the flow in the boundary layer adjacent to the body is laminar and the flow separates at about 83° from the front (Fig. 20). A wide wake thus gives a large drag. At higher speeds (large Re) the boundary layer becomes turbulent, gets additional energy from the outside flow, and does not separate on the front side of the sphere. The drag coefficient is reduced from about 0.47 to about 0.08 at a critical Reynolds number of about 400,000 in free air. Turbulence in the air stream reduces the value of the critical Reynolds number (Fig. 21). The Reynolds number at which the sphere drag $C_D = 0.3$ is taken as a criterion of the amount of turbulence in the air stream of wind tunnels.

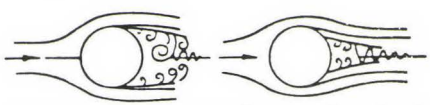


Fig. 20 Boundary layer of a sphere.

The ratio of the turbulent drag force for a sphere to the viscous drag is

$$\frac{F_{\text{turb}}}{F_{\text{vis}}} = C_D \pi r^2 \frac{\rho v^2}{2} \frac{1}{6\pi\mu r v} = \frac{C_D}{24} \frac{\rho(2r)v}{\mu} = \frac{C_D}{24} Re \qquad (7.1.6)$$

$$Re = \frac{\pi 2 r v}{\mu}$$

The parameter Re (dimensionless) is called the Reynolds number; it is used as a measure of the turbulence of the fluid flow. The length $(2r)$ used in defining Re for a given body is usually taken as the length of the chord in the direction of motion. Thus, for a sphere it is the diameter.

Setting Equation (7.1.6) equal to 1, shows that the change from smooth to turbulent flow occurs at a Reynolds number of about 48 (with $C_D = 0.5$). Figure 7.5 is a graph of the drag force vs Reynolds number for the range of Reynolds numbers from 10^{-1} to 10^6 and shows that the transition occurs over a wide range of Reynolds numbers. The smooth flow regime is generally below a Reynolds number of 1 and the turbulent regime above 10^3.

Exercise 7.1.1 Stokes' law

(*a*) For a 2 mm diameter bubble of air rising through glycerine, what is the predicted terminal velocity assuming Stokes' flow? Is this what you observe in the laboratory? What is the Reynolds number? Does it agree with the assumption of Stokes' flow?

(*b*) By using a propeller-like flagella an *E. coli* bacterium 1 μm in diameter can swim about 0.03 mm/s in water. What is the Reynolds number? What is the drag force on the bacterium? If the bacterium can obtain 3×10^{-12} erg per molecule of glucose and can use 10% of that energy for propulsion, how many molecules per second must it metabolize to swim continuously?

Material	*Viscosity (kg/m s)*	*Density (g/cm^3)*
glycerine	2.33 (at. 288 K)	1.24
air	1.78×10^{-5}	1.23×10^{-3}
water	1.0×10^{-3}	1.0

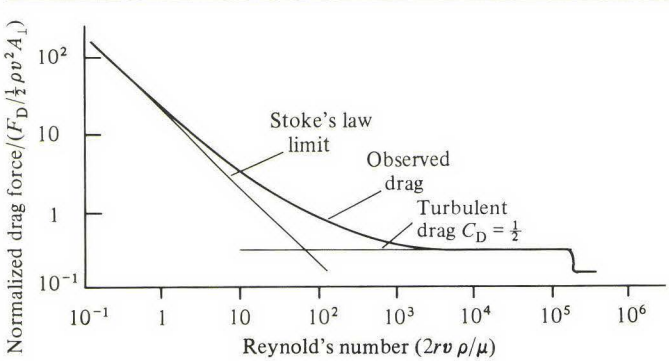


Fig. 7.5. Drag force vs. Reynolds number (from Turcotte & Schubert, *Geodynamics: Application of Continuum Physics to Geological Problems*, Wiley & Sons, New York, 1982).

A sphere starting from rest in a liquid will be acted upon by gravity F_g and buoyancy F_b forces. Once it begins to move, the drag force F_d will act to slow its acceleration. By Newton's laws

$$F_g - F_b - F_d = ma \tag{7.1.7}$$

F_g and F_b are constant regardless of the speed of the ball but F_d is dependent on the speed. If Stokes' flow is assumed, Equation (7.1.7) becomes a differential equation for the velocity of the sphere

$$m(dv/dt) + (6\pi r\mu)v - (F_g - F_b) = 0 \tag{7.1.8}$$

Exercise 7.1.2 Approach to terminal velocity

(*a*) Assume the solution to Equation (7.1.8) is of the form $v = a(1 - e^{t/b})$. Plug into Equation (7.1.8) and find a and b.

(*b*) Plot the velocity vs. time for a glass sphere of 0.60 cm and 0.26 g starting from rest in glycerine. What is the decay time b of the acclerated motion?

(*c*) How far will the sphere fall before attaining 0.95 of the final terminal velocity?

7.2 The experimental apparatus

To measure the viscosity of a fluid the apparatus like that shown in Figure 7.6 will be used. It consists of a column of glycerine into which spheres of various sizes and compositions can be dropped and observed to fall under the influence of gravity. The velocity of the falling sphere can be measured by noting the time at which it moves through each of the four light beams. The essence of the following experimental work is to write programs to measure the required times and to graph the resulting data.

Each of the four light beams which traverse the glycerine column have several elements. LED light source activates a cadmium sulphide photo-resistor whose resistance changes when light shines upon it. To sense this resistance change and to convert it into a digital signal suitable for computer processing, a voltage comparator circuit is used.

An LED is a small solid state light bulb which requires about 10 mA of current and 1.5 V to operate. A higher voltage source is generally used together with a current limiting resistor in series as shown in Figure 7.7. An LED passes current in only one direction so it is important that it be connected with the correct polarity.

A cadmium sulfide photo-resistor, is used in many cameras to compute the exposure time. Like a thermistor, it is a passive device whose resistance changes. The cadmium sulfide sensor being shown has a resistance of over 20 MΩ in the dark and a resistance in the hundreds of ohms in bright sunlight; so its resistance changes by over 100,000 to 1. Although it is quite sensitive

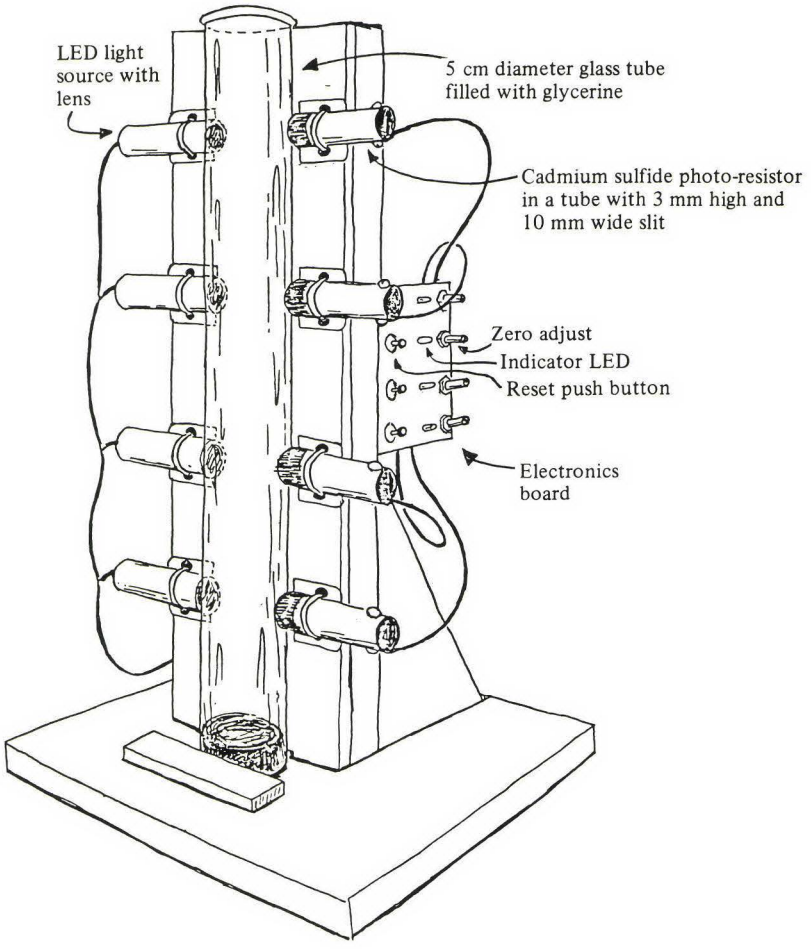


Fig. 7.6. Viscometer apparatus.

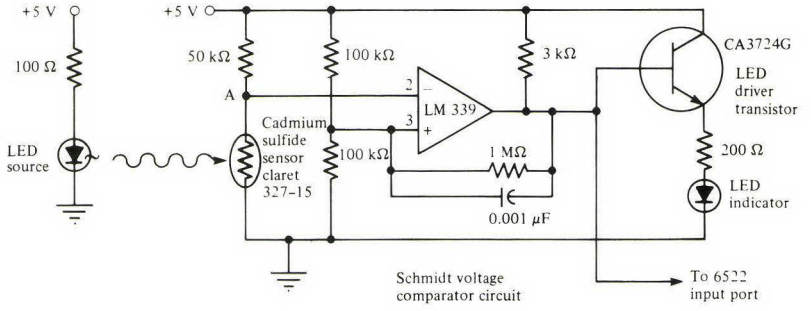


Fig. 7.7. Optical position sensor circuit.

to light, a cadmium sulfide cell is a rather slow device; it takes about 30 ms to fully respond to a sudden change in light level.

To translate the resistance change which the light beam induces in the photo-resistor to a digital signal, a voltage comparator is used (Figure 7.7). The comparator will produce an output of either 5 V or 0 V depending upon whether the input voltage to the + input of the device is greater than or less than the voltage to the − input. Each LM339 has four such comparators in a single 8 × 15 mm integrated circuit chip. The comparator circuit in Figure 7.7 has a little bit of positive feedback incorporated to give latching action; it takes more voltage to turn it on and less voltage to turn it off than just the minute voltage change required to make the comparator switch. This hysteresis is similar to that used in the temperature controller of Chapter 2. The circuit is called a Schmidt trigger and is used frequently with mechanical switches to eliminate chattering.

Exercise 7.2.1 Cadmium sulfide cel resistance and voltage changes

To get a feeling of the voltage changes being registered by the cadmium sulfide light detectors, attach the wires and turn on the 5 volt power to the fluid column apparatus. Fill the column with glycerine and wait until most of the bubbles are gone. The glycerine column needs to be in place for the sensors to focus correctly. Level the apparatus with the screws on the base. Attach an oscilloscope probe to the test point provided on the circuit board and put the oscilloscope in the free running mode with a sensitivity of 1 V/div. This point is the hot (not ground) side of the cadmium sulfide cell (point A, Figure 7.7). The 50 kΩ potentiometer which is in series with the photoresistor should be set so that the voltage at A is about one half the supply voltage, ie, 2.5 V. Break the light beam with a small piece of paper and note the voltage change which occurs. Move the paper across the light beam as fast as you can to get an idea of the minimum response time of the cadmium sulfide cell. Moving the paper vertically will probably give a faster response since the entry slit on the front of the tube is about 3 mm high and about 10 mm wide.

To set the potentiometer level for the experiment, turn it so that LED goes off, then the other way until it just goes on. Test the setting by dropping a medium sized ball.

7.3 The need for using machine language

Even though the data taking rate in the experiment under consideration is modest by most standards, BASIC is too slow to do the job properly.

In addition, testing the value of binary bits associated with the comparator outputs can be done more simply and cleanly in machine language where binary is the natural number system.

To show the speed limitation imposed by BASIC on this experiment, we need only estimate the time scale associated with the apparatus. To make a rough estimate, assume that the maximum velocity of fall to be reckoned with is about 0.3 m/s; this corresponds to the ball falling from the top of the column to the bottom (about 0.6 m) in about 2 s. The light beams have a width of about 3 mm; thus the computer should be able to record an instant of time t with a resolution of t = distance/velocity = 0.003/0.3 = 0.01 s. In this time the computer needs to be able at least to decide that the light beam has been intersected and to record the time of intersection.

Exercise 7.3.1 Speed of a sphere in air

Estimate the time resolution needed to measure with the present apparatus described above, the speed of a sphere falling through air.

For the computer to decide that the light beam has been intersected, the data in an input port must be read and tested. This can be done in BASIC with a WAIT instruction. To estimate the execution time of the WAIT instruction Program 7.3.1 in Figure 7.8 can be used. Square waves are generated on DB7 and fed into DB2 as a simulated signal. These are tested with the WAIT instruction. After the WAIT instruction finds DB2 HI it puts out a pulse on DB0 which is then viewed simultaneously on an oscilloscope with the square wave going into DB2. Thus the execution time can be measured directly. The time at which the WAIT instruction found a 1 in DB2 is recorded by line 65. To go through the WAIT and time recording instructions took 9 ms. Since we require a resolution of 10 ms, the WAIT instruction would be only marginally fast enough for our purposes.

Though testing a bit in APPLESOFT can be done using the WAIT instruction, it has the annoying features that only one state of the bit can be tested. If you want to determine if a bit is HI or LO, Program 7.3.2 in Figure 7.9 can be used. If a number x is written in binary form then dividing it by 2^N has the effect of simply moving the digits N places to the right; if

$$x = \%00ab\quad cd00$$

then $\frac{1}{2}x$ is

$$\tfrac{1}{2}x = \%000a\quad bcd0$$

The INT(X) instruction sets all of the digits to the right of 2^0 place equal to zero. Thus line 70 of Program 7.3.2 with $N + 1 = 3$ does the following operation on the number X = %*abcd efgh*:

$$\%abcd\ efgh - \%abcd\ e000 = \%0000\ 0fgh$$

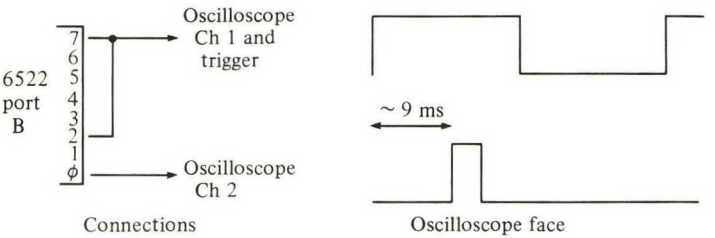


Fig. 7.8. The BASIC WAIT instruction program example.

```
 5 REM PROGRAM 7.3.1
10 BA = 50176                    } Constants
12 C1 = 256
14 C2 = BA + 9
16 C3 = BA + 8
20  POKE BA + 2,129              Set up DDRB out DB7,DB0 set up
30  POKE BA + 11,224             ACR $C0 free run load T1
40  POKE BA + 4,246
50  POKE BA + 5,64
60  WAIT B,4                     Wait for DB2 GO HI and record time
65 T = C1 * PEEK (C2) + PEEK
     (C3)
70  POKE BA,1
80  POKE BA,0                    Put out pulse on DB0
90  GOTO 60
```

Time for wait loop and T measurement ≈ 9 ms

Fig. 7.9. A BASIC program to determine the status of the *N*th bit.

```
 5  REM PROGRAM 7.3.2
10  INPUT "X=";X
40  INPUT "N=";N
50 F = 2^N - 1
60 g = 2^(N + 1)
70  IF X - G * INT (X / G) > F THEN
      GOTO 180
75  PRINT "N TH PLACE IS 0"
76  END
180  PRINT "N TH PLACE IS 1"
190  END
```

This technique for testing bits is useful in some situations but is even slower than a WAIT instruction. We will use machine language to test the sensor bits.

7.4 Machine language program to record fall of a sphere through glycerine

A common technique for controlling experimental apparatus is to use a main program written in BASIC or other high level language for doing the mathematical analysis of data, displaying the experimental results, and plotting data, in conjunction with subroutines written in assembly language which do the bit manipulation and other tasks associated with gathering the data.

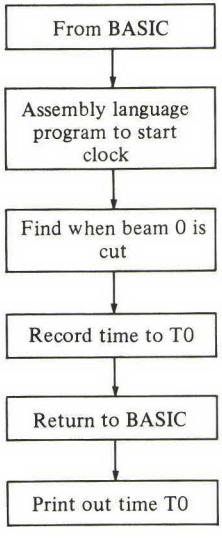
Exercise 7.4.1 Light beam sensing and timing

(*a*) Connect the fluid column outputs to the PB0, PB1, PB2 and PB3 inputs to the 6522 and write an assembly language program to start the clock decrementing the T2 counting registers at 1 ms intervals. Call the program from BASIC and write a few instructions which read the clock registers to check that the clock is functioning properly.

(*b*) Expand your assembly language program so that in addition to starting the clock, it will wait for the first light beam to be cut and store the time that this occurs. Test with a piece of paper interrupting the beam.

(*c*) Write a BASIC program which calls the machine language program and prints out the recorded time. (See Figure 7.10 for a flow chart.)

Fig. 7.10. Flow chart for Exercise 7.4.1.



For debugging assembly language programs you can use the trace command which single steps through the program. Another useful technique is to substitute a BRK instruction for an instruction op-code and then run the program to see whether execution gets to the place in question. The op-code for BRK is 00. When execution is halted by a BRK instruction, the memory location just after the BRK is displayed together with the values in the 6502 registers. By temporarily inserting BRK instructions in different locations, the difficulty can usually be found fairly quickly. It is rare to write an assembly language subroutine (or any program for that matter) which does not require debugging. With this in mind you may wish to leave spaces in the program (by inserting the 'No Operation' instruction, NOP) at places where you may want to put in BRK to debug.

7.5 Graphing scales

Another problem which this experiment presents is that of choosing scales to plot the experimental data gathered. Before making a set of measurements, the best choice of graph scale is not apparent. It is desirable to have the computer choose an appropriate scale for the axis on the graph after the data has been obtained. The length of the scale axes should be such that the data points use as much of the screen as possible. This will make good use of the limited resolution which the APPLE graphics screen offers. It can be done by finding the maximum value to be plotted and multiplying by 1.1 so that there is 10% free space to the right of the maximum data point. Thus, the second parameter in the &SCALE instruction will be set to 1.1*MAX VALUE OF DATA.

To work out where to put the tick marks is slightly more subtle. The graph should have tick marks at even values. For example the oscilloscope uses a 1, 2, 5 spacing for its scales. Program 7.5.1 of Figure 7.11 chooses an appropriate tick mark interval from . . ., 0.001, 0.002, 0.005, 0.01, . . ., 1, 2, 5, . . . etc. Having a total of 5–10 tick marks on the graph seems appropriate. The program starts with the assumption that the points range from $t = 0$ to $t = TM$. It assumes that the values of the ordinate of the graph are known at the outset, ie, it is to go from −10 to 40, and that tick marks on the ordinate are to be placed every 10 units.

Fig. 7.11. BASIC program example for automatic adjustment of graphing scales.

```
3000  REM SUBROUTINE TO SET SCALE
3005  REM PROGRAM 7.5.1
3010  REM INPUT TM THE MAX VALUE OF T
3020  XM = 1.1 * TM                          Max value for XM
3030  LM = 0.43429 * LOG (XM)                Take logarithm base
                                             10 of XM
3040  IM = INT (LM)                          Get mantissa of the
3050  MM = LM - IM                           logarithm
3060  IF MM < 0.301 THEN XT = 0.2 *          If the mantissa
      10 ^ IM: GO TO 3090                    0–0.301 tick marks
3070  IF MM < 0.602 THEN XT = 0.5 *
      10 ^ IM: GO TO 3090
3080  XT = 1 * 10 ^ IM
3090  &SCALE, - XT,XM, - 10,40               Put left edge of scale
3100  LX$ = "TIME"                           at – XT; tick marks
3110  LY$ = "Z"                              will go at intervals of
3120  & LABELAXES,XT,10                       XT from left edge of
3130  &  GR ID, - XT, - 10,XT,10             graph
3140  RETURN
```

7.6 Double precision addition and subtraction

Exercise 7.6.1 Double precision addition

The Program 7.6.1 in Figure 7.12 does a double precision addition between two numbers and stores the result. If x_L and x_H are the low and high parts of x and the same is true of y and z, where do you put x and y before starting and where do you look for $z = x + y$? Show with examples and explanation that the program steps do a correct double precision addition.

Program 7.6.2 of Figure 7.12 shows how a double precision subtraction is done in assembly language. The SBC instruction actually uses the adder inside the microprocessor to do a subtraction. This is done using the following observations: first, subtracting a binary number x from the binary number % 1111 1111 gives the result $\bar{x}$ (x complement) which is just x with all its zeros changed to ones and ones changed to zeros. For example:

1111	1111	$FF
−1011	0011	−$B3
0100	1100	$4C

(*a*) Program 7.6.1

```
9300  18          CLC
9301  AD 00 94    LDA  $9400
9304  6D 02 94    ADC  $9402
9307  8D 04 94    STA  $9404
930A  AD 01 94    LDA  $9401
930D  6D 03 94    ADC  $9403
9310  8D 05 94    STA  $9405
```

(*b*) Program 7.6.2

```
9320   38         SEC
9321-  AD 00 94   LDA  $9400
9324-  ED 02 94   SBC  $9402
9327-  8D 04 94   STA  $9404
932A-  AD 01 94   LDA  $9401
932D-  ED 03 94   SBC  $9403
9330-  8D 05 94   STA  $9405
```

Fig. 7.12. Assembly language arithmetic: (*a*) double precision addition (*b*) double precision subtraction.

Secondly, adding \$01 to \$FF gives \$00 (try it). Therefore:

$$\left.\begin{aligned} y - x &= y + \$01 + \$FF - x = y + \$01 + \bar{x} \\ y - x &= y + \bar{x} + c \end{aligned}\right\} \quad (7.6.1)$$

where c is the carry.

So, when the 6502 executes an SBC instruction, it complements the subtrahend, then adds that result to the minuend and the carry (\$01). That is why the carry is set before doing an SBC.

Exercise 7.6.2 Quadruple precision subtraction

(*a*) Show that the Program 7.6.2 in Figure 7.12 does double precision subtraction as is claimed.

(*b*) Write and test a program which does quadruple precision subtraction of the number x stored in \$9400, \$9401, \$9402, \$9403 (\$9403 contains the most significant part of x, \$9400 the least) from the number y stored in \$9404 . . . \$9407 and place the result z in \$9408 . . . \$940B. You may wish to use indexed addressing but *be careful*! CMP, CPX and CPY change the carry bit.

7.7 The viscometer

Exercise 7.7.1 The viscometer and the viscosity of glycerine

(*a*) Write and test a program outlined by the flow chart in Figure 7.13 which waits for the subsequent light beams to be cut, measures the time interval from the cutting of the first beam and then plots the data on a graph. So that your assembly language program is suitable for putting onto the EPROM in the next section be sure it requires less than 256 bytes and contains no JMP or JSR instructions. The ASL instruction is quite useful for shifting the mask in this program. Store your program as a binary file on a disk. The BASIC program should call the assembly program, then plot the position vs. time for the data obtained.

Use your position vs. time plots to determine the terminal velocity and calculate the viscosity and Reynolds number for several balls of different diameters and densities. It is not necessary to do a least squares fit for each plot. Have the computer use two of the measured times to calculate the velocity and draw a line. You can check visually to make sure the other points fall along the line. If you input the diameter and mass of the ball, the computer can then calculate the viscosity (using Equation (7.1.5)) and Reynolds number (Equation (7.1.6)) and print them on the graph, too. Make

Fig. 7.13. Flow chart for Exercise 7.7.1, the viscometer.

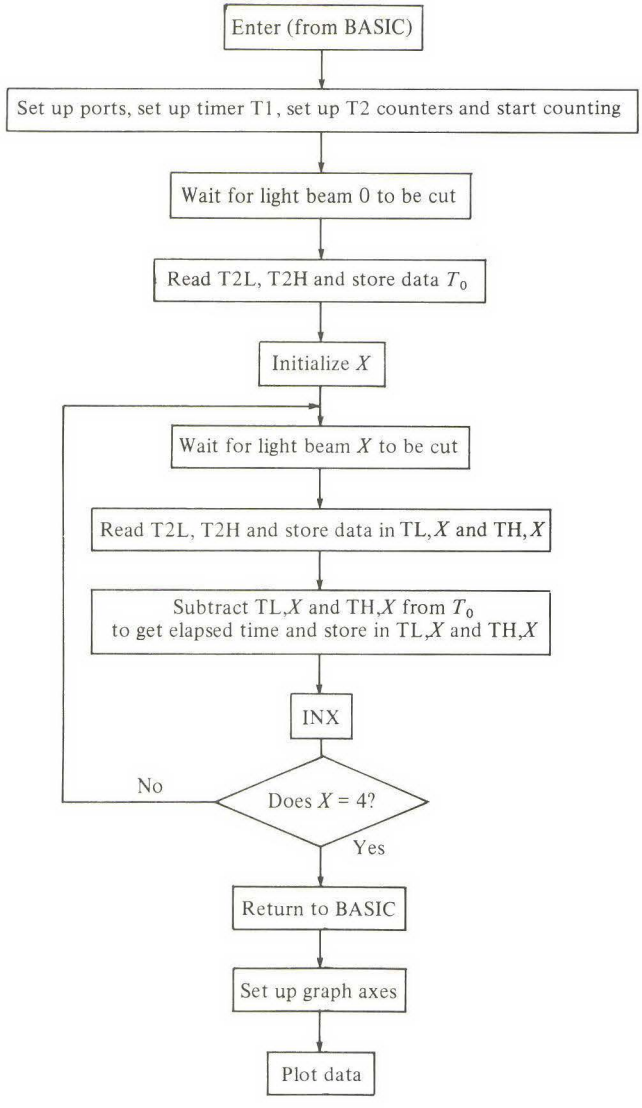


Table 7.1 *Typical diameter and mass of spheres*

Diameter (cm)	Mass (g)	Material
0.60	0.26	glass
1.31	2.7 ± 0.1	glass
1.575	5.2 ± 0.1	glass
0.09	4.1	lead
1.17	9.22	lead
1.45	17.8	lead
0.80	2.02	steel

several graphs with balls which are available to you. Some useful sizes are shown in Table 7.1.

(*b*) Compare your determinations of the viscosity with the value given in a reference book. Note that temperature and water content have a large effect on the viscosity of glycerine. (Exercises 7.7.3 and 7.7.4)

(*c*) Using the data for several balls, make a plot of the drag force on the ball (which equals the gravity minus buoyancy forces) vs. its terminal velocity times its radius (vr). Why is this plot significant? Can the transition to turbulence be seen?

(*d*) Will the timing part of your program work for a ball dropping in air (no glycerine)?

(*e*) Try replacing the glycerine in the column with water and repeating some of the measurements. Lead balls work the best in this case. Since the flow will be well into the turbulent regime ($Re \sim 400$), the viscosity cannot be accurately determined (why?). However, the drag coefficient C_d can be plotted vs. Re by assuming a value for the viscosity of water ($\mu = 0.010$ poise at 20 °C).

One experimental problem with this apparatus is that with larger diameter balls, the walls of the column interface with the flow and affect the motion of the ball. The viscosity value can be corrected with the following empirical formula (Dinsdale & Moore, 'Viscosity and its Measurement', Reinhold Publishing, New York, 1962):

$$\mu_{\text{true}} = \mu_{\text{measured}}[1 - 2.104(r/R) + 2.09(r/R)^3 - 0.95(r/R)^3]$$

where R is the radius of the column and r the radius of the ball.

Exercise 7.7.2 The wall effect

Correct the viscosity values obtained in Exercise 7.7.1 to account for the wall effect.

The temperature and water content of glycerine affects its viscosity greatly. The water content is particularly hard to control since glycerine absorbs water vapor from the air when it stands uncovered.

Exercise 7.7.3 Temperature variation of the viscosity of glycerine

The data shown in Table 7.2 taken from the *Handbook of Chemistry and Physics* and the *American Institute of Physics Handbook* shows the temperature dependence of the viscosity of glycerine. Make a plot of viscosity vs. temperature. Suspecting an exponential dependence, now plot the natural logarithm of the viscosity vs. temperature and find the parameters and the model equation which give the best fit.

Table 7.2

Temperature °C	Viscosity Pa s (MKS unit)
0	12.1
6	6.26
10	3.95
15	2.33
20	1.49
25	0.954
30	0.625

Exercise 7.7.4 The viscosity of aqueous solutions of glycerine

The data shown in Table 7.3 from the *Handbook of Chemistry and Physics* (Chemical Rubber Co., 52nd Edition, page D191) gives the relative viscosity of aqueous solutions of glycerol by percentage weight of glycerol.

(*a*) Plot these data to see the general behavior. Try both linear and log plots.

(*b*) Try fitting these data with the mixture formula:

$$\frac{1}{\mu} = \frac{P}{\mu_1} + \frac{1-P}{\mu_2}$$

where P is the concentration of component one and μ_1 and μ_2 are the viscosities (μ_1 = glycerine, μ_2 = water).

(*c*) Try fitting these data with the Arrhemis formula (Dunstan & Thole, *The Viscosity of Liquids*, Longmans Green and Co., London, 1914).

$$\mu = \mu_1^P \mu_2^{1-P}$$

Table 7.3

% glycerol by weight	Relative viscosity (Viscosity/viscosity of water)
1	1.02
10	1.29
20	1.73
30	2.45
40	3.65
50	5.92
60	10.66
70	23.00
80	59.78
88	147.20
92	383.70
96	778.90
98	1177.00

(*d*) Try fitting a simple exponential to the data above 80% concentration.

7.8 Using an EPROM

Erasable Programmable Read Only Memory (EPROM) is a cross between ROM (which can't be reprogrammed) and RAM (which forgets everything when the power is turned off). Like a ROM, an EPROM requires special equipment to write the data into its memory. It will not forget the data when the power is turned off; but unlike ROM, it can be erased by shining ultraviolet light through a quartz window in the top of the chip. Thus, programs can be developed by erasing and reprogramming improved versions on a single EPROM. In building experimental apparatus it is often convenient and economical to have a simple one board computer dedicated to doing a single task with a ROM or EPROM to store its program.

Exercise 7.8.1 Blasting and using an EPROM

Using a computer which has an EPROM programmer, blast an EPROM with the program you wrote above. If you are using the J. Bell programmer the APPLESOFT program EPROM. BLASTER can be used. Read through these instructions before you begin. Take your disk with the BASIC and assembly language programs from Exercise 7.7.1 to an APPLE computer set up with the EPROM programmer and RUN EPROM. BLASTER (a copy of the program listing for this program is in Appendix J). Then follow the instructions to enter your Exercise 6.7.1 machine language program into the EPROM from your disk. When the

program is done, release the lever and remove the chip from the holder. Return with the programmed EPROM and an EPROM card to your APPLE.

WARNING!!!

Before placing the EPROM into your computer, turn the computer off!

After being sure the computer power is off, pop the cover off the APPLE computer, place the EPROM in the holder in slot 7, being careful that the pin orientation is correct; gently lock it with the lever. From BASIC, the program in the EPROM will now be called at address $C700. Modify your Exercise 7.7.1 BASIC program to call the machine language program in the EPROM. Repeat the tests with the same type of balls to demonstrate that you can reproduce your Exercise 7.7.1 graphs using the EPROM chip and that the data is consistent. When you have finished with this exercise please turn off the computer, remove the EPROM you used and place it in the place designated for blasted EPROMs.

8 Interrupts

Interrupts are an important capability of modern computers. They allow the processing of several independent tasks by the CPU. On large computers they allow multiuser and time sharing activities. On microprocessors they allow the running of a main program while periodically taking data or sending data to a slow device like a printer. Also computer start up, DOS, reset and BREAK instructions make use of the interrupt function.

In the discussion which follows, we will first trace the steps taken by the CPU when it receives an Interrupt Request (IRQ) from other parts of the APPLE and then look into the ways we can cause interrupts to be generated and serviced.

8.1 Interrupts and the CPU

The interrupt sequence is similar to a jump to a subroutine except that it occurs when signalled by wire leading to the CPU (IRQ) line whereas the subroutine jump is a normal executable statement (JSR). When an interrupt signal is present on the IRQ and the interrupt disable bit (I) of the process status register (see Figure 8.1) is 0, the CPU begins processing the interrupt. The interrupt disable bit is used to prevent the CPU from beginning to process the same interrupt again before it is completed the first time. Without it the computer would go into a continuous regression. The I bit is set equal to 1 during an interrupt sequence and further interrupts are ignored until this I bit is returned to zero. This can be done with the CLI

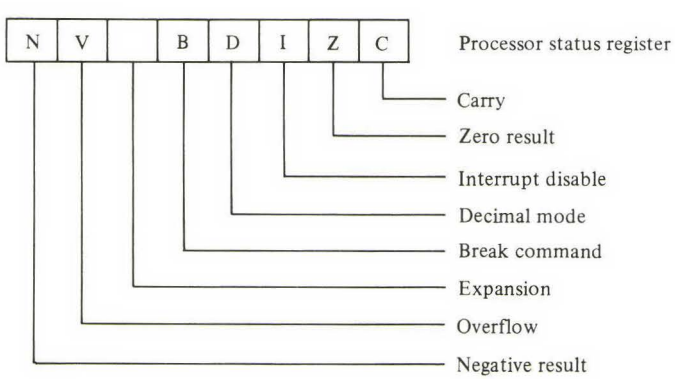


Fig. 8.1. The process status register. If a BRK instruction is executed, a forced interrupt is done and the B bit is set to 1 (this forced interrupt is not masked, ie it is not inhibited when the interrupt disable bit is set to 1). Setting the D bit to 1 (SED instruction) makes the 6502 do binary coded decimal addition (ADC) and subtraction (SBC). IRQ is recognized only if I bit is 0. After IRQ is accepted I bit is automatically set to 1.

instruction but is done automatically at the return from the interrupt service routine.

If the I bit of the process status register is 0 the CPU recognizes an IRQ signal and, after completing the machine language instruction currently in process, stores the program counter (P) and process status register (S) (with the I bit set to 0) on the stack. It is remarkable but necessarily true that if these registers and the A, X, Y registers are restored to their values just before the interrupt, the executing program will continue exactly where it left off as if the interrupt had not occurred. After saving the P and S registers, the CPU then sets the I bit of the process status register to 1 to prevent further interrupts and goes to the top of memory $FFFE.FFFF to find the location of its next instruction (see Figures 8.2 and 8.3).

At Rom addresses $FFFE.FFFF the CPU finds the address $FA40 and begins executing the program at that address. The first instruction (STA $45) saves the accumulator in memory location $45 for later restoration. The next instructions at $FA42.FA47 pull the old value of the process status register

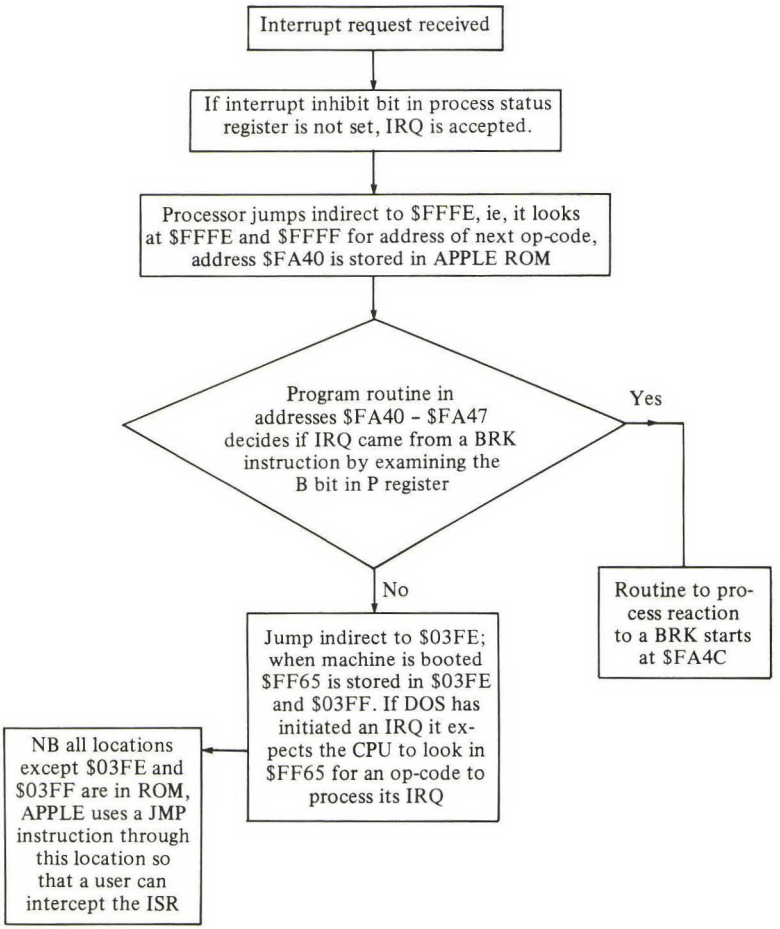


Fig. 8.2. Flow chart for normal generated interrupts.

from the stack and check to see if the interrupt came from a BRK instruction. If it did, the CPU is directed to $FA4C to service the BRK interrupt where the CPU registers are displayed on the screen and the computer halts the program. If the BRK bit was not set, the CPU would execute JMP ($03FE) which is an indirect jump to the memory address stored in $03FE.03FF. Under normal operation the machine would find the address $FF65 there and proceed to further interrupt processing by the APPLE monitor program. This is the APPLE Interrupt Service Routine (ISR).

At the end of servicing an interrupt, the value stored at $45 is returned to the accumulator followed by a return from interrupt instruction (RTI). This instruction pulls the old value of the process status register (with the I bit set to 0) and program counter from the stack and restores them in their appropriate registers. The interrupted program then continues from where it left off.

Note that the X and Y registers are not automatically saved by the normal interrupt sequence. If they are used during the servicing of the interrupt the original values must be saved at the start and restored before returning. Also notice that the indirect addressing JMP at $FA49 using addresses at

```
FFFE- 40 FA                                  After placing the process
                                             status register and program
                                             counter on the stack and
                                             setting the B bit to 1, CPU
                                             goes here to start normal
                                             machine ISR.
FA40-  85 45      STA  $45                   Accumulator saved at $45.
FA42-  68         PLA                      ┐
FA43-  48         PHA                      │
FA44-  0A         ASL                      │
FA45-  0A         ASL                      ├ Break bit of old PSR checked.
FA46-  0A         ASL                      │
FA47-  30 03      BMI  $FA4C               ┘ Branch to break ISR.
FA49-  6C FE 03   JMP  ($03FE)               Jump indirect to ($03FE)
FA4C                                         Routine continues.

FF65   D8         CLD                        Entry point for continued
FF66-  20 3A FF   JSR  $FF3A                 service of normal machine
FF69-  A9 AA      LDA  #$AA                  interrupts.
FF6B-  85 33      STA  $33
FF6D                                         Routine continues.

03FE-  65 FF                                 Address loaded for normal
                                             machine interrupts.
```

Fig. 8.3. Unmodified Machine Generated Interrupt Service Routine.

$03FE.03FF is the only part of the interrupt sequence that causes the CPU to look in the RAM. The other instructions are all in the ROM and cannot be modified by the user. This short trip outside the ROM is what allows the user to enter the interrupt process.

8.2 User controlled interrupt

Your APPLE is equipped with a 6522 VIA which has the capability of generating IRQs by means of its Interrupt Enable Register (IER) (see Appendix E, Figure 4). User controlled interrupts involve programming the IER and intercepting the ISR at $03FE.

The programming of the IER is indicated in Appendix E, Figures 29 and 30. Six events can generate IRQs. We will be concerned with interrupts produced by either the T1 counter or T2 counter reaching zero. The programming of the IER is a two step process. First the bit(s) for the function not being used must be disabled. This is accomplished by placing a zero in bit 7 to indicate a disable action followed by ones in bits to be disabled and a zero in bits not to be disabled. Then enabling is accomplished by placing a one in bit 7 followed by ones in bits to be enabled and zeros in bits not to be enabled. For example to set up T1 for interrupts first %0011 1111 is sent to the IER followed by %1100 0000 sent to the same location. After programming the IER the I bit of the process status register is set equal to zero with a CLI instruction. This signals the computer to accept interrupts.

Now if the address of your own interrupt program (ISR) is put into $03FE.03FF, all non-BRK generated interrupts will be directed to it. Some of these interrupts (those not generated by the VIA) still need to be sent to $FF65. So in your ISR there must be a check to see if the non-BRK generated interrupt was caused by the 6522 VIA or some normal machine interrupt. This check is performed by reading the Interrupt Flag Register (IFR) of the 6522 (see Appendix E, Figure 29).

Bit 7 of the IFR is set any time the VIA produces an IRQ. The other bits are set by the conditions indicated. For example, if T1 generated the interrupt, bit 6 and bit 7 are set. Both bits are cleared by reading the low byte of the T1 counter (T1C.L) or by writing the high byte (T1C.H).

The user controlled interrupt process is now complete. Two programs are necessary. The Interrupt Initialization Routine (IIR) redirects the interrupt process by inserting the memory location of the ISR at $03FE. The IER is programmed. The I bit of the process status register is set to zero and the IFR cleared. The interrupt initialization routine ends with an RTS instruction and is called or run only once to establish the user controlled interrupt conditions.

The second program is a user ISR which will be run on every non-BRK generated interrupt. This ISR begins at the memory location put into $03FE. The first thing that must happen in the ISR is a check to see if the interrupt was generated by the 6522 VIA. This is done by reading the IFR. If the

interrupt was not generated by the 6522 then the CPU is sent back to $FF65 in the ROM. If it was generated by the 6522 then the IFR is cleared and the rest of your ISR executed. At the end of this ISR the old value of the accumulator is retrieved from $45 and the X and Y registers restored if they were used. The ISR is completed with a RTI instruction.

8.3 An ISR

To illustrate the use of an IRQ, a program which continuously displays the elapsed time on the CRT monitor screen in the upper left hand corner, without disturbing the normal APPLE operation, will be used as an example. The procedure uses the T1 timer in the 6522 VIA to generate a continuous stream of interrupts, one every 1/100 s. Each time the CPU is interrupted the ISR increments a three-byte counter TL, TM, TH by one count. After 100 counts (1 s) a display routine is called and data are displayed on the CRT. After this is done program control is returned to the APPLE program in process.

Exercise 8.3.1 Running an interrupt program

Using the MINIASSEMBLER, type in the IIR of Figure 8.4(*a*) and (*b*) and the ISR from Figure 8.5(*a*) and (*b*). Run the IIR from BASIC (call 37120). You should see a display of seconds in the upper line of the CRT which increments every second. Once this is going try running other programs and doing other operations of the APPLE.

In addition to the interrupt processing, several aspects of the ISR of Exercise 8.3.1 are new. The section of the program from $900D to $9027 uses Binary Coded Decimal (BCD) arithmetic. In BCD each byte represents a number from 0 to 99 rather than from 0 to $FF, that is, each nibble (four bits) is allowed to count only to 9 before a carry is taken to the next nibble (see Figure 8.6). The four-bit nibbles thus become direct representations of decimal digits. The SED instruction (set decimal) at $900D puts the CPU into the BCD mode and the CLD at $9027 takes it out. To help in the understanding of the BCD mode replace SED with a NOP in $900D for Exercise 8.3.1 and observe the operation of the ISR.

Another new operation is the direct write to the screen in the program section from $9032 to $906F. When the APPLE is in text mode (TEXT) just as in graphics mode (HGR2), certain memory locations are being read to display the information on the screen. For HGR2 these are $4000.5FFF and for TEXT they are $0400.07FF beginning at the top left of the screen. Each byte is interpreted as one character to be displayed. So a character can be placed anywhere on the screen by writing to one of these memory locations.

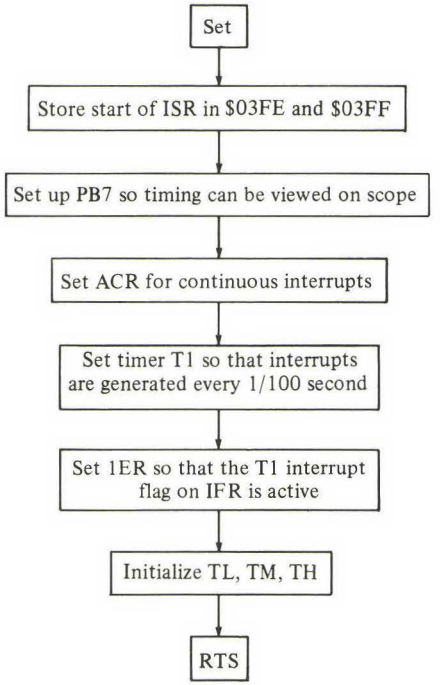


Fig. 8.4. IIR for generating T1 timer interrupts at 0.01 s intervals: (*a*) flow chart; (*b*) program.

```
9100-  78          SEI
9101-  A9 00       LDA  #$00
9103-  8D FE 03    STA  $03FE
9106-  A9 90       LDA  #$90
9108-  8D FF 03    STA  $03FF
910B-  A9 80       LDA  #$80
910D-  8D 02 C4    STA  $C402
9110-  A9 C0       LDA  #$C0
9112-  8D 0B C4    STA $C40B
9115-  A9 EC       LDA  #$EC
9117-  8D 04 C4    STA $C404
911A-  A9 27       LDA  #$27
911C-  8D 05 C4    STA $C405
911F-  A9 3F       LDA  #$3F
9121-  8D 0E C4    STA  $C40E
9124-  A9 C0       LDA  #$C0
9126-  8D 0E C4    STA  $C40E
9129-  AD 04 C4    LDA  $C404
912C-  A9 00       LDA  #$00
912E-  8D 00 94    STA  $9400
9131-  8D 01 94    STA  $9401
9134-  8D 02 94    STA  $9402
9137-  58          CLI
9138-  60          RTS
```

Load $9000 into $03FE.03FF

Set up PB7 output to observe square waves on scope

Set T1 for free run

Set T1L and T1H for 1/100 s

Set IER for T1 interrupts

Clear bits 6 and 7 of IFR

Initialize TL TM and TH

Clear I bit of process status register

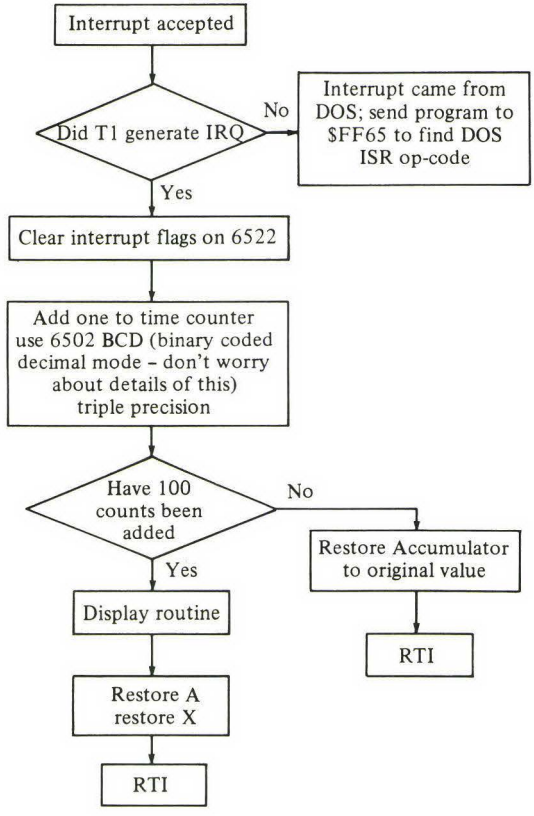


Fig. 8.5. ISR for counting interrupts and displaying the elapsed time: (*a*) flow chart; (*b*) program.

```
9000-  AD 0D C4    LDA  $C40D     Read IFR and send CPU to
9003-  29 C0       AND  #$C0      $FF65 if interrupt did not
9005-  D0 03       BNE  $900A     come from 6522 T1.
9007-  4C 65 FF    JMP  $FF65      Otherwise go to $900A
900A-  AD 04 C4    LDA  $C404     Clear IFR bits 6 and 7
900D-  F8          SED            Set decimal mode
900E-  18          CLC
900F-  A9 01       LDA  #$01
9011-  6D 00 94    ADC  $9400
9014-  8D 00 94    STA  $9400
9017-  A9 00       LDA  #$00      Increment TL, TM and TH
9019-  6D 01 94    ADC  $9401     by 1
901C-  8D 01 94    STA  $9401
901F-  A9 00       LDA  #$00
9021-  6D 02 94    ADC  $9402
9024-  8D 02 94    STA  $9402
9027-  D8          CLD            Clear decimal mode
```

```
9028-  AD 00 94    LDA  $9400
902B-  F0 03       BEQ  $9030
902D-  A5 45       LDA  $45
902F-  40          RTI
9030-  8A          TXA
9031-  48          PHA
9032-  AD 02 94    LDA  $9402
9035-  6A          ROR
9036-  6A          ROR
9037-  6A          ROR
9038-  6A          ROR
9039-  29 0F       AND  #$0F
903B-  18          CLC
903C-  69 30       ADC  #$30
903E-  8D 00 04    STA  $0400
9041-  A9 0F       LDA  #$0F
9043-  2D 02 94    AND  $9402
9046-  18          CLC
9047-  69 30       ADC  #$30
9049-  8D 01 04    STA  $0401
904C-  AD 01 94    LDA  $9401
904F-  6A          ROR
9050-  6A          ROR
9051-  6A          ROR
9052-  6A          ROR
9053-  29 0F       AND  #$0F
9055-  18          CLC
9056-  69 30       ADC  #$30
9058-  8D 02 04    STA  $0402
9068-  A9 0F       LDA  #$0F
905D-  2D 01 94    AND  $9401
9060-  69 30       ADC  #$30
9062-  8D 03 04    STA  $0403
9065-  A9 20       LDA  #$20
9067-  A2 00       LDX  #$00
9069-  9D 04 04    STA  $0404,X
906C-  E8          INX
906D-  E0 23       CPX  #$23
906F-  D0 F8       BNE  $9069
9071-  68          PLA
9072-  AA          TAX
9073-  A5 45       LDA  $45
9075-  40          RTI
```

If $9400 is not zero reload accumulator and RTI. If $9400 is zero go to $9030

Transfer X register to stack

Increment display on screen

Restore X register and accumulator

Binary	%	0010	0101	0110	0100	Binary representation of decimal 9572
Hex	$	2	5	6	4	Hexadecimal representation of decimal 9572
Decimal	—	9	5	7	2	Decimal number
BCD	—	1001	0101	0111	0010	BCD representation of decimal 9572

```
      2          5          6          4
4096|9572   256|1380   16|100    1|4
    8192       1280      96      4
    1380        100       4      0
```

9572. = $2564

Fig. 8.6. Number systems: binary, hexadecimal, decimal, BCD. In BCD each group of four bits represents one decimal digit and thus the binary representations of hex numbers A, B, C, D, E, F are not valid.

b_7 b_6 b_5 / Bits					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	ROW ↓ / COLUMN →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

0/0	NUL	Null
0/1	SOH	Start of Heading
0/2	STX	Start of Text
0/3	ETX	End of Text
0/4	EOT	End of Transmission
0/5	ENQ	Enquiry
0/6	ACK	Acknowledge
0/7	BEL	Bell
0/8	BS	Backspace
0/9	HT	Horizontal Tabulation
0/10	LF	Line Feed
0/11	VT	Vertical Tabulation
0/12	FF	Form Feed
0/13	CR	Carriage Return
0/14	SO	Shift Out
0/15	SI	Shift In

1/0	DLE	Data Link Escape
1/1	DC1	Device Control 1
1/2	DC2	Device Control 2
1/3	DC3	Device Control 3
1/4	DC4	Device Control 4
1/5	NAK	Negative Acknowledge
1/6	SYN	Synchronous Idle
1/7	ETB	End of Transmission Block
1/8	CAN	Cancel
1/9	EM	End of Medium
1/10	SUB	Substitute
1/11	ESC	Escape
1/12	FS	File Separator
1/13	GS	Group Separator
1/14	RS	Record Separator
1/15	US	Unit Separator
7/15	DEL	Delete

Fig. 8.7. ASCII Code. (Reproduced with permission from American National Standard X3.4–1977, copyright 1977 by the American National Standards Institute. Copies may be purchased from the American National Standards Institute, 1430, Broadway, New York, New York 10018.)

To do this the alphabet and punctuation characters must be represented by numbers. In the APPLE (and many other computers) the seven-bit ASCII code (American Standard Code for Information Exchange) is used. This code was first used by teletype machines. Figure 8.7 shows the mapping of characters to numbers. To see how the display and character are related, type in and RUN the following BASIC program.

```
 5  REM PROGRAM 7.3.3
10  FOR I=0 TO 255
20  POKE 1024 + I, I
30  NEXT I
CONTROL — RESET
RUN
```

The program section to display the time on the screen contains parts which isolate each nibble of the sum, convert it to ASCII, then write it to the proper location on the screen.

8.4 T2 generated interrupts

Interrupts generated by T2 can be handled in a similar fashion to those generated by T1. As in previous chapters, if we use the T1–T2 timer pair to give us longer time intervals, PB7 must be set up as an output and a wire connected from PB7 to PB6. The initialization routine (Program 8.3.1) needs to be modified to allow only T2 generated interrupts. T2H and T2L as well as T1H and T1L are initialized. The clearing of the T2 flag in the IFR is done by reading T2L or by writing T2H as indicated in Figure 8.4. Register T2H must be written to again anyway since it does not reload automatically.

Exercise 8.4.1 Writing an interrupt program

Write an IIR and on ISR using the T2 interrupts to ring the bell every second. Use T2 to count down (via T1 signals) from a starting value. When it reaches zero, it should interrupt; then the ISR should reset T2 and ring the bell. To ring the bell, JSR to the bell subroutine at $FBE2. This subroutine uses the Y register so be sure to save it. Test that your interrupt routine is working properly by running other APPLE programs you have on your disk with the bell ringing in the background mode. What happens when you access the disk with the interrupt going? Do you now have a beeping APPLE?

---

9 Other topics

9.1 Hardware for data acquisition and control

There are two styles of hardware for using a microcomputer to acquire data and control equipment. One is exemplified by the APPLE IIe system you have used in the laboratory. The ADC, the DAC and the digital I/O cards are inside the computer and are under direct control of the microprocessor. They have control and data registers which are directly addressable via the buss. External devices (sensors, switches, etc) are connected to the cards. Creative programming can turn the computer into, for example, an oscilloscope (ADC and display) or a signal generator (DAC) as the laboratory exercises have shown.

Other buss systems are in use which, like the slots in the APPLE, allow a microprocessor to be connected to various data acquisition and control devices by simple board replacements. Some of the more widely used ones are IBM-PC buss, S100, STDBUS, MULTIBUS and QBUS.

The second style is to have a separate box next to the computer which has the ADCs, DACs, digital I/O lines and a programmed microprocessor controller. It communicates with the computer via a serial or parallel communication system (see Section 9.2). The box takes care of the data acquisition and control while the computer is used to send control bytes to tell the box what to do and to receive the data for further processing. The limitation of this style is in the speed of communication to the computer and in the number of things the box has been preprogrammed to know how to do. However, some computers do not have card slots (notably the APPLE IIc and the Apple MacIntosh) so that this style of data acquisition and control is the only possible choice.

9.2 Serial data communication

In the exercises you have done, transmission of data to the computer has been direct. The sensors have been connected to the ADC or VIA which are inside the APPLE and connected to the internal buss. This is not always the case. Many newer instruments have means of gathering and storing digital data themselves. To analyze the data, they are transmitted along a cable between the instrument and computer. The methods used for this communication can be split into two broad groups: serial and parallel.

Serial data is transmitted one bit at a time. Each bit follows the previous

one after a preset time interval has passed. This interval must be known to the receiver so that it can synchronize its timing with the transmitter. There are several hardware standards which are used for serial transmission. By far the most widespread is the RS-232C standard. It is used for slow to moderate speed communication (110–19200 bits per second or 'baud') over distances of up to 300 m. Most terminals connected to multiuser computer systems use this standard as do many printers and plotters. At its minimum only two wires are needed: a ground and a signal wire. Since the standard requires that data only go one way on the signal wires, this minimal system would be good only for devices like printers. Most of the time another wire is added to provide two way communication. The RS-232C standard is also used to communicate with a modem which is a device that transmits and receives serial data over the telephone lines. A data rate of 300 or 1200 baud is commonly used.

Figure 9.1 shows how an ASCII character 'K' would be sent using the RS-232C protocol. The start bit signals the beginning of a data word. It is followed by 4–8 data bits. Then sometimes a parity bit is included which is used for error checking. At the end are one or two stop bits. The number of bits and their meaning as well as the rate of transmission must be known at the receiver. Since the receiver restarts its timing at each start bit, it only needs to remain synchronous over the length of the data word.

One problem which arises often is that the transmitter sends data faster than it can be processed at the receiver. The receiver needs to have a way of saying, 'Hold on a moment while I take care of what I already have.' This is done either with another wire which signals a hold or in software by having the receiver transmit characters to signal the transmitter. Most commonly the ASCII character 19 (Control S or XOFF; is HOLD and 17 (Control Q or XON) is GO. The transmission becomes a game of RED LIGHT GREEN LIGHT.

The transmission and reception of serial data is usually done by a UART (Universal Asynchronous Receiver Transmitter). Once it knows the protocol of the data being sent, the UART takes care of the serial interface. It is used by addressing registers; those for the 6551 chip are shown in Table 9.1. On transmission, it translates the byte in its data register to serial form and on reception, it translates the serial data into a byte in the data register.

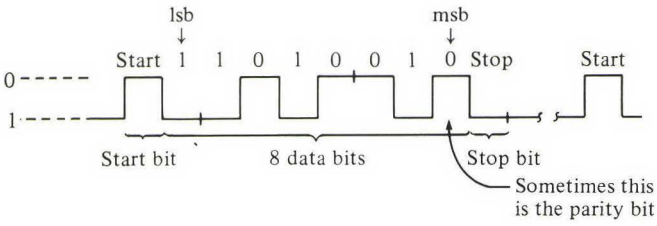


Fig. 9.1. Serial transmission of an ASCII 'K' character. 'K' = Binary 01001011. The time for one bit is 1/BAUD rate.

Table 9.1 *6551 register format*

Register	Write	Read
0	Transmit Data Register	Receiver Data Register
1	Programmed Reset (Data is "Don't Care")	Status Register
2	Command Register	
3	Control Register	

The control register is used to set the protocol of the serial data, the command register is for interrupt control, and the status register is used to signal data transmission and error conditions. The interrupt capability is often used in communication programs so that the computer need not continually monitor the UART status.

The most obvious limitation of serial data transmission is in the speed of communication. A new standard, RS-422A, has been defined to try to alleviate this problem. It offers speeds of 100 000 bits per second over distances of 1500 m. Another limitation is that each device needs a separate cable and interface. Further information about serial communication can be found in the references.

Exercise 9.2.1 Serial communication

Write out the serial sequence (Figure 9.1) which would transmit an ASCII 'j' character on a serial line. Use 1 start, 7 data, no parity, 1 stop bits and 9600 baud. Indicate the time on your picture. Refer to the ASCII chart of Figure 8.7 for the binary code for ASCII 'j'.

9.3 Parallel data communication

In parallel transmission the data in one word are communicated simultaneously by having many wires connecting the transmitter and receiver. The data buss connecting various parts of the computer is one example; each bit of a data byte is stored in a memory location at the same time. To transmit and receive an eight-bit byte of data externally, eight wires are needed as well as several other wires, eg, a R/W wire, to control the direction and timing. A parallel hardware standard has been adopted for laboratory instrumentation which is called IEEE–488. Although the communication distance is limited to a total of 20 m, it can have up to 16 devices simultaneously connected and can transmit data at speeds up to 1 000 000 bytes per second. Many laboratory instruments now have options which allow connection to this buss.

In the IEEE–488 cable there are a total of 24 wires. Eight of these are ground wires which help to increase the noise immunity. There are eight data wires, three data transfer control wires and five management control

wires. The devices on the buss can be designated as either talkers, listeners or controllers. There must be at least one controller which is usually a general purpose microcomputer. It manages the communication by using the management control wires to designate which devices should be listeners and which should be the talker. Only one talker is allowed at one time but the talker device can be changed at any time. For example, a printer would be a listener and a voltmeter would be a talker. Devices can also be active or inactive so, for example, the printer need not be printing all the time.

Communication of a byte of data is synchronized via a handshake mechanism using the three data transfer control wires. Figure 9.2 shows the sequence of signals to transmit one byte after the active talkers and listeners have been designated. Note that a LO level indicates a true condition and a HI level false. The sequence starts by each active listener letting the NRFD (Not Ready For Data) line go HI (false) thus indicating that it is ready to receive data. Due to the open collector design of this signal wire interface, the signal does not go HI until all of the listeners are ready. When the active talker sees the NRFD high it places the data on the data wires and signals that the data is valid by dropping DAV (DAta Valid). The listeners then set NRFD LO and each store the data from the buss. As each completes that task, it lets the NDAC (Not Data ACcepted) signal go HI indicating that the data has been stored. As with the NRFD, the NDAC wire does not go HI until all the listeners have let it go. Thus the slowest listener active on the buss limits the speed of communication. The talker then sends DAV HI indicating that the data is not valid any longer and the listeners drop NDAC.

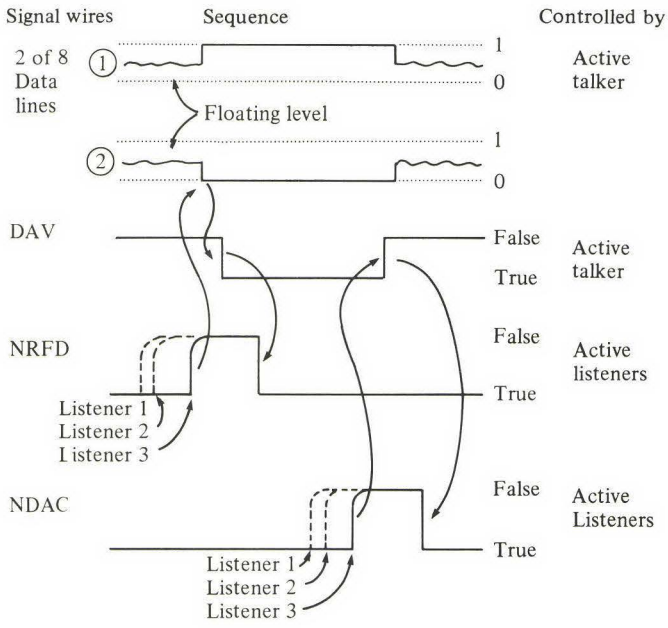


Fig. 9.2. IEEE-488 data transfer protocol. DAV is 'Data Valid', NRFD is 'Not Ready for Data', NDAC is 'Not Data Accepted'.

The buss is then ready for the next byte transfer. This sequence is called a handshake since the data transfer takes place when both the transmitter and receiver have agreed (signalled) that they are ready.

The remainder of the wires in the buss are used for signals between the devices so that the talkers and listeners can be designated and so that the devices can signal emergency conditions. For example, if ATN (ATtentioN) is true it indicates to all the other devices that the controller wants to talk and that everyone else should listen. If SRQ (Service ReQuest) is true a listener is requesting to talk. The full protocol can be found by reading the interface documentation (Hewlett Packard calls it the GPIB interface) or by getting a description from the Institute of Electrical and Electronic Engineers.

Exercise 9.3.1 Parallel communication

Using 6502 assembly language implement IEEE–488 protocol using the 6522 VIA interface. Assume that the eight data lines of Port B are connected to the data lines of the interface and that PA0 is connected to DAV, PA1 to NRFD, PA2 to NDAC. Also assume that the active talker is the computer (6502). Write a program which will transfer 100 bytes from memory locations $9000–$9063 to the active listeners. Use the following outline:

(1) initialize the ports, set the data lines as inputs (floating temporarily), set DAV HI
(2) start loop of 100
(3) look for NRFD HI (all listeners ready)
(4) set data lines as output and put data on lines
(5) set DAV low (signal data is valid)
(6) look for NDAC HI (data accepted by all listeners)
(7) set DAV HI and set data lines as inputs (floating again)
(8) loop back for next byte of data

9.4 Sensors and transducers

In the laboratory work in this book you have used only three kinds of sensors, a potentiometer, a thermistor and a photoresistor, and two controllers, a stepping motor and a HEXFET switch. There are many other kinds of sensor, at least one for each physical parameter which is measured. A good physical understanding of the system to be measured is always the first step. Then, selection or design of the sensor can be done. Some generalized performance characteristics have been discussed in the sections on zero, first, and second-order systems. Understanding the physical and electrical basis of the sensor is also important. Please refer to the references for information on the wide variety available. Keep in mind that there is always room for invention.

9.5 Software for data acquisition and control

Of the large amount of software available for a particular microcomputer, there are two basic types: languages and application programs. The first are the primary tools with which a computer is programmed (eg, BASIC). The second are particular programs which have the computer perform specific tasks (eg, AMPERGRAPH). Both have their places in the use of the computer in the laboratory.

As in the work done in this laboratory, most laboratory computers are programmed in the laboratory using a chosen language. Table 9.2 lists the more popular ones with some comments on their efficacy. A program in an interpreted language is executed as it is run whereas one in a compiled language must be translated into machine code before it can be run. Be sure that the language has the capability of PEEKing and POKEing absolute memory locations.

Most application programs for data acquisition which are available at this time are libraries of subroutines (or procedures or modules) which, when called, do specific tasks. For example, one subroutine would output a number to the DAC and another would get the time from the timer. The libraries are specific to the language and the hardware being used.

What really made microcomputers popular for the home and business were two applications programs: the word processor and the spread sheet. These are versatile programs dedicated to a specific need (such as writing) but general enough to encompass a variety of tasks within that need (such as letters, reports, lists). There are a few programs available which address the need for a generalized data acquisition, storage, analysis and graphing. As the business market saturates, it is to be expected that more and varied programs will be written for the scientist and engineer.

Table 9.2 *Microcomputer languages*

Language	Comments
BASIC	Interpreted or compiled, common, easy to learn, awkward, slow
Assembly	Compiled, most direct control of computer system, awkward
FORTRAN	Compiled, traditional for number-crunching analysis, has complex numbers!, awkward, frequently no PEEK and POKE, libraries available
Pascal	Compiled, structured for easier programming
Ada	Like Pascal but US Department of Defense backing
Modula II	Like Pascal but corrects some weakneses
FORTH	Threaded, can be extended by user, originated for data acquisition and control, somewhat awkward reverse polish constructs
C	Compiled, both low level and high level programming, structured, terse

9.6 Where to go from here

A great deal of useful work can be done in the laboratory by applying the principles you have learned. For those interested, there are several areas of study which extend the topics discussed here. A laboratory course on digital and analog electronics would be useful in understanding sensors and their associated signal conditioning circuits as well as the electrical operation of the computer itself. An introductory course in signal processing and analysis would be useful for general data analysis. For those interested in process automation, a course in systems analysis would be helpful. To keep up on the latest hardware and software in this quickly changing field, consult trade journals. Also get on the mailing lists of suppliers. They will frequently send out product bulletins. But the best way to learn is the way you have learned in this laboratory; that is by doing it.

Appendix A
Laboratory materials and resources

The following is a detailed description of the equipment used in the laboratory at Cornell University together with possible sources for these parts.

Each student work area (Figure A.1) has an APPLE IIe computer with printer and data acquisition cards, a 5 V power supply, and an oscilloscope. The APPLE IIe has the following configuration:

Slot	*Device*
1	Practical Peripherals Microbuffer II+
2	John Bell Engineering A–D Converter (Figure A.2 right)
4	John Bell Engineering 6522 Parallel Interface (Figure A.2 middle)
6	APPLE disk controller
7	John Bell Engineering EPROM Card (Figure A.2 left)

One of the computers in the laboratory has a John Bell Engineering EPROM Programmer attached to the 6522 interface so that EPROMs may be programmed. An ultraviolet EPROM eraser is also available. The Microbuffer II+ is attached to an Epson MX-80 printer. These may be changed to suit as long as the printer buffer/printer combination can print the high resolution graphics of the APPLE.

The text is written for use with the DOS 3.3 operating system for the APPLE. Prodos would probably work too if the appropriate changes are made in the text. The text also assumes that the Mad West Software AMPERGRAPH package is being used. We have not seen comparable packages which could be substituted. For EPROM blasting, the program listed in Appendix J is useful.

We use a B+K Precision Model 1476A dual trace 10 MHz oscilloscope and a Power One model C5–6 power supply. Almost any oscilloscope will do and the only specification which needs to be met on the power supply is that it has a 5 V output at 5 A. Look in the back of BYTE magazine or in surplus catalogs for good prices. We have tied the computer and power supply grounds together permanently so as to minimize grounding problems for the students.

The cables from the data acquisition boards are brought out to a proto-board (Figure A.3) where connections may be made easily. We find that the Super Strip (available through Digi-Key or Jameco) to be versatile.

Fig. A.1. General setup in the laboratory. The computer, disk drive, and monitor are on one wooden stand; the printer is on another which sits over the oscilloscope. The power supply is between the two and the protoboards are on the top of the computer.

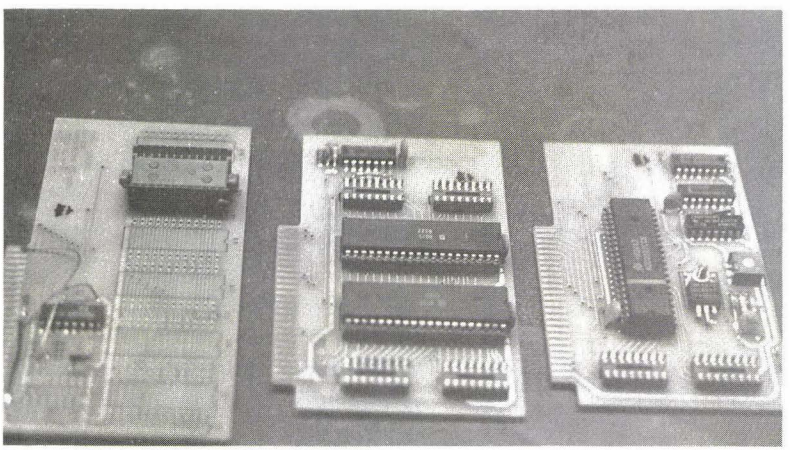


Fig. A.2. Three John Bell Engineering circuit cards with cables removed. From left to right: the EPROM holder, the 6522 VIA cad, and the ADC card.

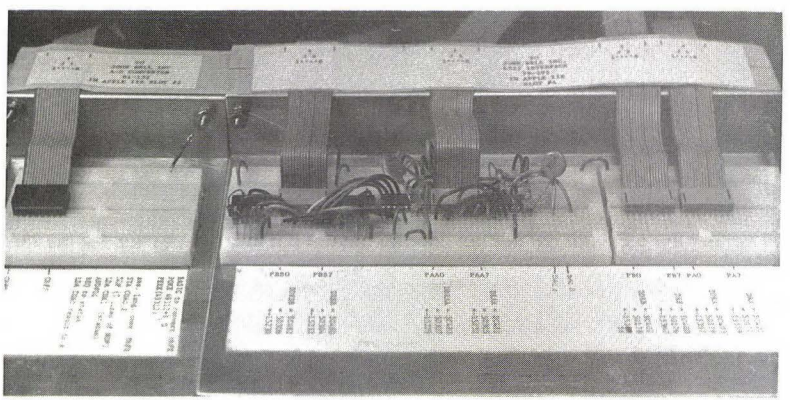


Fig. A.3. A view of the protoboards where the interface cables terminate. The ADC cable is to the left and the four 6522 VIA cables are in the center and to the right. In the center the 6522 VIA #2 Port B is wired to the LEDs and their drivers and Ports A and B are wired to the DACs.

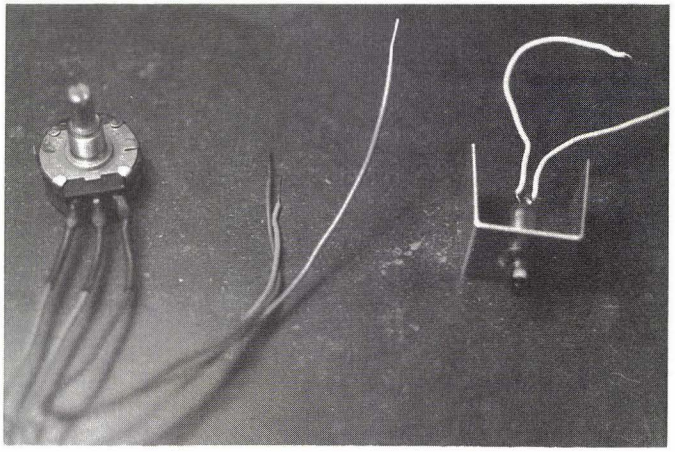


Fig. A.4. The push button and the potentiometer.

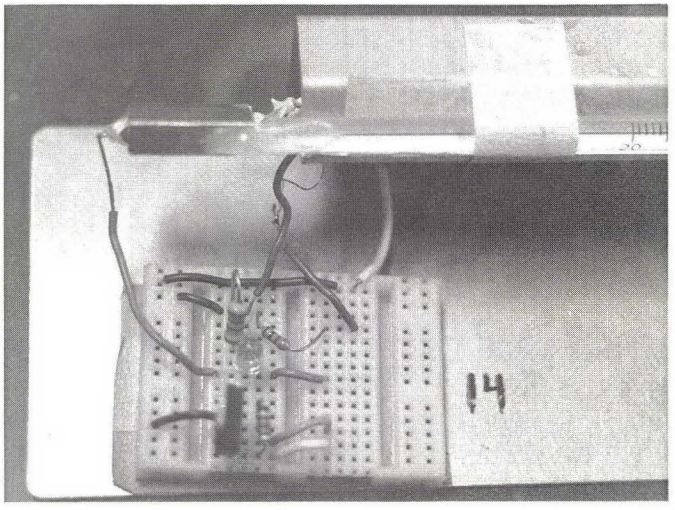


Fig. A.5. View of the thermistor calibration/temperature controller apparatus. The aluminum block at the top holds the heater resistor, the thermistor and the thermometer; all emplaced with conductive grease and some glue to hold them in place. The circuit is constructed on a piece of protoboard.

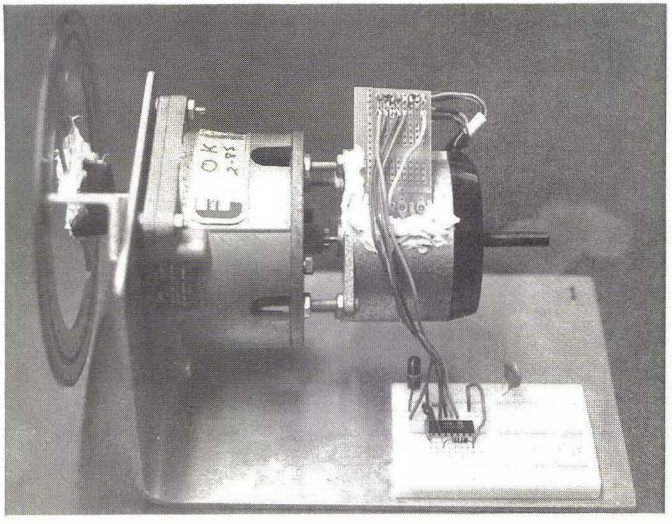


Fig. A.6. Stepping motor apparatus. The protractor is mounted to the left on the output shaft of the gearbox (center). The stepping motor (on the right) is mounted to the gearbox and coupled with a piece of rubber tubing. The circuit is constructed on a piece of protoboard.

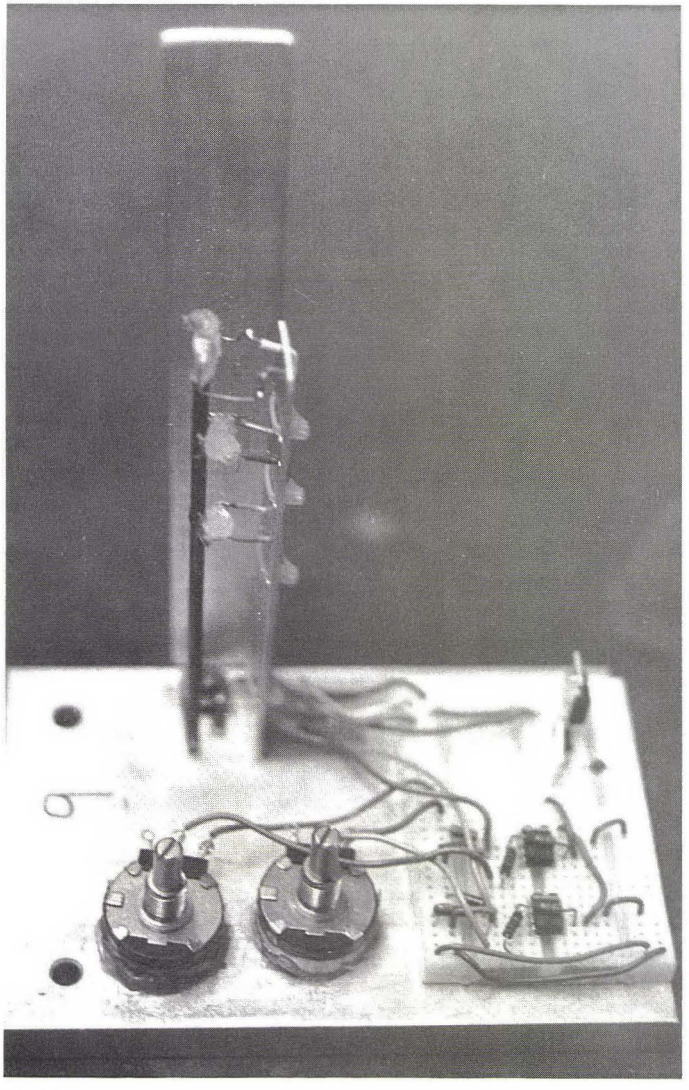


Fig. A.7. Heat flow apparatus. The copper wire is secured to the aluminum base plate and has three holes drilled for mounting the heater resistor and the two thermistors. These are emplaced into the copper with thermal grease and secured with glue. Their leads are supported with a piece of aluminum. The circuit is constructed on a piece of protoboard attached to the base.

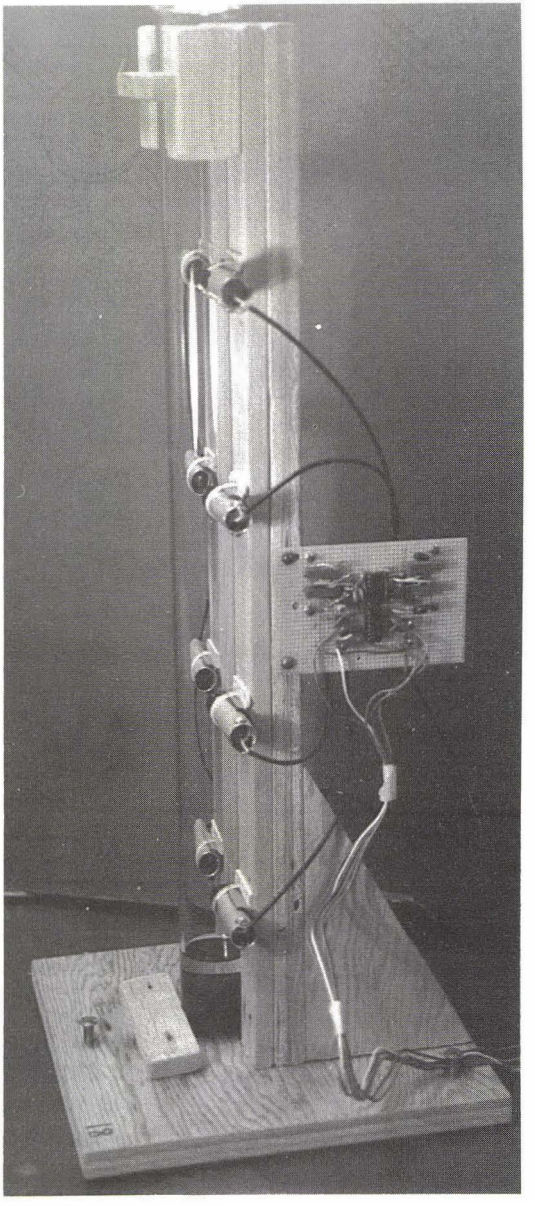


Fig. A.8. Viscometer. A glass tube, the four positions sensors and the electronics are mounted on a wooden base which can be leveled by adjusting three screws.

Laboratory Apparatus

Potentiometer (Figure A.4)
Almost any will do in the resistance range of 100 Ω to 1 MΩ.

Thermistor calibration apparatus (Figure A.5)
A thermistor, thermometer, and heater resistor are mounted in an aluminum block about 2 cm × 2 cm × 2 cm size. The thermistor we use is a Fenwall GB34P2. Others may be substituted by adjusting the bias resistor depending on the room temperature resistance. Heat conductive grease is used in the holes so that the thermometer, thermistor and heater resistor make good thermal contact with the block. The circuit used is shown in Figures 3.5 and 3.4. A standard laboratory mercury thermometer is used but others can be substituted. The HEXFET is an International Rectifier IRF 510. Almost any of that line can be substituted.

Stepping motor (Figure A.6)
The stepping motor apparatus consists of a stepping motor connected to a 200 : 1 gear box by a rubber sleeve and controlled by a UCN-4202A controller (Sprague Electric Co.) which is mounted on a protoboard. Figure 4.1 shows the circuit used. The stepping motor is a surplus item (A. W. Hayden Co. P/N B86138) which may be hard to find but the controller will work with Permanent Magnet stepping motors rated to 500 MA and 15 V. You may have to modify the wiring of the motor to suit the controller. BYTE magazine is a good place to look for surplus motors. The gearbox is from AST/SERVO Systems and again is a surplus item. The reduction ratio is not critical.

LED Output Counter (Figure A.3)
These are simple LEDs with 270 Ω resistors and a 74LS04 driver. The circuit (Figure 4.4) is constructed on the protoboard where the cables from 6522 VIA #2 are attached.

Heat Flow Apparatus (Figure A.7)
The apparatus for the heat flow experiment consists of a copper rod (#10 copper wire, 2.59 mm diameter) mounted vertically on an aluminum base as shown in Figure 5.3. An aluminum support runs parallel to the rod to support the wires to the heater resistor and thermistors. The $\frac{1}{4}$ watt resistor is placed in a hole in the top of the rod. The thermistors (Fenwall GB32J2) are placed in small holes at 2.5 cm and 5 cm down from the resistor. Thermal grease is again used to ensure thermal contact. The standard amplifier circuit employed is shown in Figure 5.5. A protoboard is used to construct the circuit. The only special consideration is that the operational amplifier be able to run on 0–5 V supplies.

Digital to Analog Converter (Figure A.3)
This circuit (Figure 6.7) is constructed on the protoboard where the LEDs and the 6522 VIA #2 cables are attached. The DAC used is a National Semiconductor DAC0808. Others may be substituted with some change in circuitry. The negative voltage necessary for running this chip can be obtained from the Apple buss by a slight modification of the 6522 VIA card.

Viscometer (Figure A.8)
As depicted in Figure 7.6, the viscometer apparatus consists of a glass tube about 5 cm in diameter with a rubber stopper at one end mounted in a wooden frame to which the detectors and electonics are attached. The frame may be leveled by means of the three screws at its base. The light source for the position sensors are green LEDs mounted in 1 cm aluminum tubes with a small focusing lens at one end. The light detectors are CdS photo-resistors (Claret 327–15) mounted in another 1 cm aluminum tube. In front of the sensor is a 3 mm high 10 mm wide slit cut in cardboard. The circuit for one of the sensors is shown in Figure 7.7. The balls used can be of a wide variety however the 'wall effect' becomes very evident for large ones. Table 7.1 shows some we have found useful.

Addresses

John Bell Engineering
400 Oxford Way
Belmont, CA 94002

MADWEST Software
PO Box 9822
Madison, WI 53715

Digi-Key
PO Box 677
Thief River Falls, MN 56701

Jameco Electronics
1355 Shoreway Road
Belmont, CA 94002

Sprague Electric Co.
115 Northeast Cutoff
Worcester, MA 01606

AST/SERVO Systems Inc
930 Broadway
Newark, NJ 07104

Individual pieces or a kit of all the laboratory apparatus can be purchased from:

Vector Magnetics Inc
PO Box 127
Ithaca, NY 14851

Appendix B
Merging programs: use of the RENUMBER program

An efficient method for writing programs is to complete one small piece at a time. Each piece should be tested and understood; even if you have to write another short program to do this. Only then, as a separate task, combine the pieces into larger and larger portions of the main program. It is best first to write out in words, block diagrams and flow charts what you are trying to do with the program and/or apparatus. By doing this the tasks involved become conceptually separated and can then be dealt with as pieces of the whole.

If you follow the procedure outlined above, it will be necessary to store small program segments on the disk and then to put them together to form programs without having to retype all the pieces already tested. The APPLESOFT LOAD command is not satisfactory for this since it will first clear out the program in the machine.

The program RENUMBER on the SYSTEM START disk enables you to merge and renumber BASIC programs. It works somewhat like the AMPERGRAPH program in that it appends some new instructions to BASIC. To use RENUMBER, place the SYSTEM START disk in the drive and type RUN RENUMBER CR. (At this time RENUMBER replaces AMPERGRAPH in memory.) A reminder of how to use it is displayed on the CRT screen. A print of this is given in Figure B.1.

Figure B.2 shows a listing resulting from the use of RENUMBER. The command LOAD DEMO1 was given to put the program DEMO1 from the AMPERGRAPH disk into the machine; the LIST command displays the program. Figure B.2 then shows that the instruction

```
$FIRST 1000,INC 15,S10,E60
```

was executed in the immediate mode; it renumbered the program statements. The listing shows that the first statement numbered in the new numbering scheme is 1000; subsequent statement incrementing at 15 units. The statements in the original program to be renumbered started with instruction 10 and ended with statement 60.

To merge a program in the machine with another, the two instructions &H and &M are used as illustrated in Figure B.3. With the renumbered program of Figure B.2 in the machine, typing &H put it into 'HOLD'. Another program can now be loaded into the APPLE without affecting the program on HOLD. None of the instruction numbers of the two programs can be the

same. Two programs are merged by typing &M CR. The subsequent LIST shows that the new program consists of the two programs put together as one.

The RENUMBER program is thus an editing procedure which makes it possible to combine two programs conveniently into a single larger program. RENUMBER and AMPERGRAPH cannot be in the machine simultaneously. One replaces the other so after RENUMBERing AMPERGRAPH will need to be reloaded. RUN AMPERGRAPH LOADER or STARTUP with the SYSTEM START disk in the drive but be sure you have saved your merged program first!

Fig. B.1. RENUMBER screen.

```
]PR#1
]RUN RENUMBER
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
&                                               &
&           APPLESOFT RENUMBER                  &
&                                               &
&  COPYRIGHT APPLE COMPUTER, INC, 1978  &
&                                               &
&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&


RENUMBER   (DEFAULT VALUES)

& [FIRST 10] [,INC 10] [,S 0] [,E 63999]

MERGE

&H   PUT PROGRAM ON HOLD
&M   MERGE TO PROGRAM ON HOLD

PRESS 'RETURN' TO CONTINUE...  RENUMBER IS INSTALLED
AND READY IF YOU USE'FP', 'HIMEM', OR 'MAXFILES'

  YOU WILL HAVE TO RE-RUN RENUMBER
```

Fig. B.2. Example of a program renumbered.

```
]LOAD DEMO1
]LIST

5  REM DEMO1
6  REM  ELEMENTARY EXAMPLE
7  REM
10  HGR2 : HIMEM: 16383
20  & SCALE,0,10, -1.2,1.2
30  & AXES,0,0,2,.2
40  FOR X = 0 TO 10 STEP .2
50  &  DRAW ,X, COS (X)
60  NEXT X
```

Program in machine to be renumbered.

```
]&FIRST 1000,INC15,S10,E60
```

Type this in +CR to renumber program

```
]LIST

5  REM  DEMO1
6  REM  ELEMENTARY EXAMPLE
7  REM
1000  HGR2 : HIMEM: 16383
1015  & SCALE,0,10, -1.2,1.2
1030  & AXES,0,0,2,.2
1045  FOR X = 0 TO 10 STEP .2
1060  &  DRAW ,X, COS (X)
1075  NEXT X
```

Program with new statement numbers.

Fig. B.3. Example of programs merged.

```
]LIST

5  REM  DEMO1
6  REM  ELEMENTARY EXAMPLE
7  REM
1000  HGR2 : HIMEM: 16383
1015  & SCALE,0,10, -1.2,1.2
1030  & AXES,0,0,2,.2
1045  FOR X = 0 TO 10 STEP .2
1060  &  DRAW ,X, COS (X)
1075  NEXT X
```

Program 1 in machine

```
]&H
```

Type &H CR put

continued

```
PROGRAM ON HOLD, USE "&M" TO RECOVER
]10 REM THE PROGRAM LISTED ABOVE
]20 REM WAS JUST CREATED UNSING
]30 REM THE RENUMBER PROGRAM
]40 REM AND PUT ON "HOLD" USING
]50 REM THE IMMEDIATE INSTRUCTION
]60 REM &H.  I WILL NOW INSERT THE
]70 REM PROGRAM I AM WRITING I.E.
]80 REM INSTRUCTIONS 10 TO 80 INTO
]90 REM THIS PROGRAM.
]&M
]LIST
5  REM DEMO1
6  REM  ELEMENTARY EXAMPLE
7  REM
10  REM  THE PROGRAM LISTED ABOVE
20  REM  WAS JUST CREATED UNSING
30  REM  THE RENUMBER PROGRAM
40  REM  AND PUT ON "HOLD" USING
50  REM  THE IMMEDIATE INSTRUCTION
60  REM  &H.  I WILL NOW INSERT THE
70  REM  PROGRAM I AM WRITING I.E.
80  REM  INSTRUCTIONS 10 TO 80 INTO
90  REM  THIS PROGRAM.
1000  HGR2 : HIMEM: 16383
1015  & SCALE,0,10, -1.2,1.2
1030  & AXES,0,0,2,.2
1045  FOR X = 0 TO 10 STEP .2
1060  &  DRAW ,X, COS (X)
1075  NEXT X
```

Program 1 on hold

Put Program 2 into machine – type or LOAD from disk

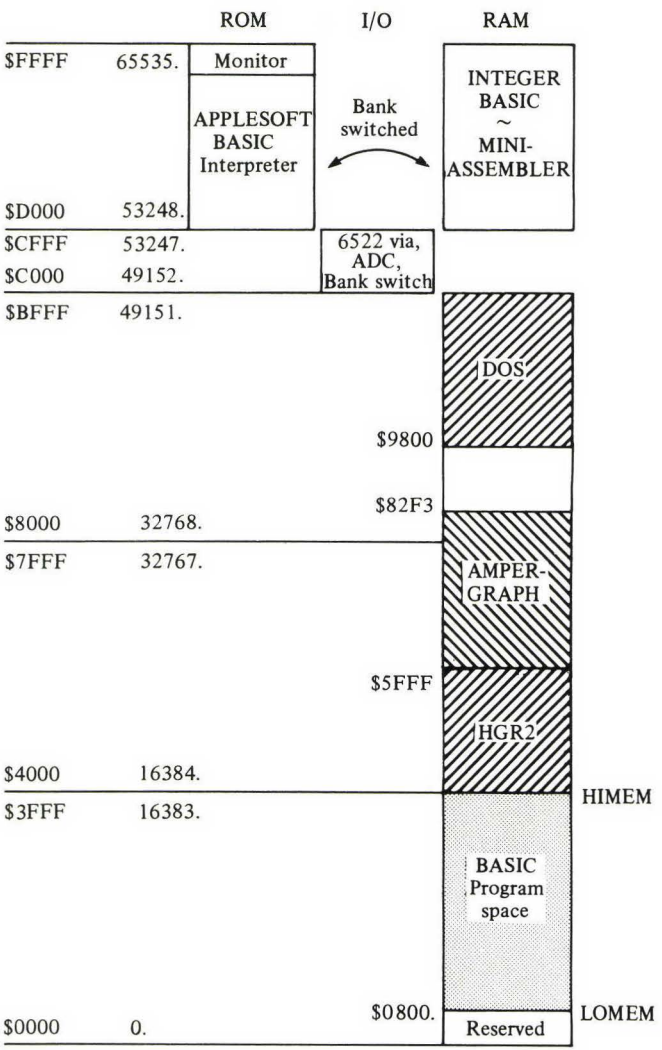
Type &M CR to merge Programs 1 and 2

Programs 1 and 2 merged together

Appendix C
APPLE IIe memory map

Figure C.1 shows how the address space of the APPLE IIe is organized. Both the decimal and the hexadecimal representations of the addresses are given (hexadecimal representation is described in Section 4.3). The main RAM

Fig. C.1. APPLE IIe memory map.



memory is in locations $0000–$BFFF . The addresses from $C000 to $CFFF are reserved for the I/O registers of peripheral devices like the disk drive and the printer. The APPLE IIe also has a block of RAM at addresses $D000–$FFFF which are the same as the ROM addresses and so would normally cause a conflict. But there is a register in the I/O space that determines which memory is being used; it acts like a switch whose position is determined by the bits in the register (a soft-switch). The command INT switches to the RAM memory (called bank-switched RAM) and FP switched back to ROM. When the power is turned on to the computer, the ROM memory is switched on.

Figure C.1 also shows some of the normal memory usage in the APPLE. The monitor and APPLESOFT BASIC interpreter are in the ROM. The INTEGER BASIC interpreter and the MINIASSEMBLER are in the bank-switched RAM and are loaded into the memory by the program on the SYSTEM START disk. The start-up program also loads the DOS into the high addreses of the main memory. Memory locations from $4000 to $5FFF are reserved for HGR2. The text display memory is at locations $0400–$07FF. A BASIC program entered from the keyboard or from a file is stored in memory beginning at $0800. The command HIMEM:16383 instructs the computer not to store any program or variables above this address (16383 decimal is $3FFF hexadecimal). This protects HGR2 from being overwritten by the program. The address space for HGR1 is from $2000 to $3FFF. The reason HGR2 is used instead of HGR1 for graphics display is so that the BASIC program can have as large a memory space as possible by using the HGR1 space for program use.

Figure C.2 shows how BASIC uses the program space which is made available to it by the LOMEM and HIMEM settings.

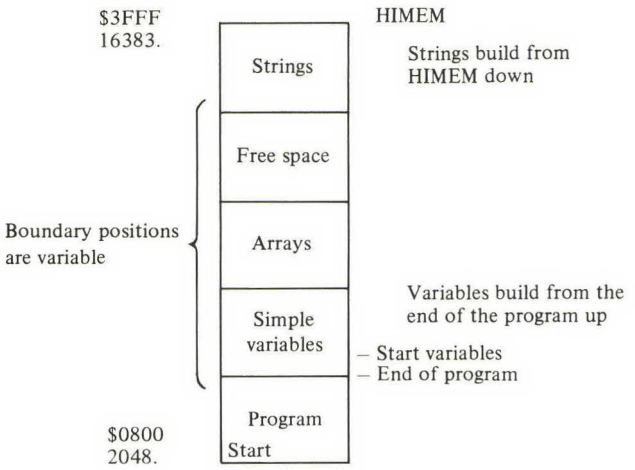


Fig. C.2. BASIC memory usage.

Appendix D
Connections and logic of the ADC

To use apparatus intelligently it helps to understand what is going on inside; the discussion below focuses on giving some insight into what occurs when you do an analog conversion. As with most things, such discussion has many layers of increasing depth and detail. This discussion will go only one veneer down.

The analog to digital conversion is done by an ADC 0817 IC which is connected to the address and data busses and to the R/W (read/write) wire of the APPLE computer (Figure D.1). Addresses 49312–49319 are devoted to doing analog to digital conversions for channels 0–7 on the protoboard to which you have attached your thermistor and potentiometers for measuring voltages.

The BASIC instruction 'POKE address, data' which you used to actuate a voltage conversion is an instruction which says: store the number 'data' in location specified by the number 'address'. The 6502 will write the data to memory by holding the R/W wire LO (Holding it at 0 V specifies a 'write' operation to memory), putting the specified address on the 16 wires of the address buss, and then putting the data on the data buss. An ordinary RAM location at the specified address would respond by storing the number which appears on the data buss. The ADC is not ordinary memory; it is an I/O device connected to the computer. The 6502 uses a system of memory

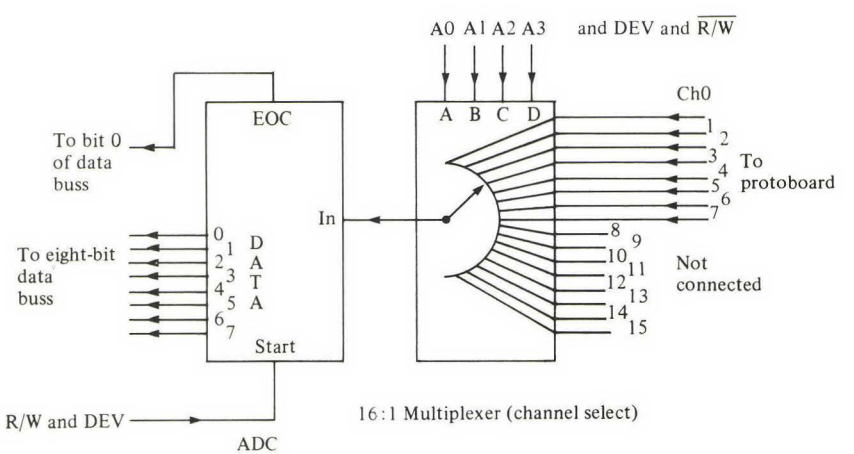


Fig. D.1. ADC connections: DEV is high when address lines A15 . . . A0 have $C0A*x*, where *x* can be any number. DATA 0 . . . 7 is connected to the data buss when R/W and DEV are high and A0 is low. EOC (End of Conversion) is bit 0 of the data when R/W and DEV and A0 are high.

mapped I/O which means that all input and output are handled through special memory locations.

When the ADC 'hears' one of its addresses called, with the R/W line LO requesting it to store data, it disregards what is on the data buss. This makes the number in the data field of the POKE instruction irrelevant. Instead of storing data the ADC switches the analog channel specified by the lower four address bits to its analog to digital conversion section and then starts conversion. The conversion from analog to digital requires about 100 μs for the ADC 0817 which is much less than the time required for a single BASIC instruction. When the conversion is completed, the digital result is stored in a memory register in the ADC. This is located at the base address 49312.

The BASIC instruction 'X=PEEK (address), reads the number in the memory locations specified by 'address' and sets the variable *X* equal to the data read. When the PEEK (49312) instruction is interpreted the 6502 CPU puts the address 49312 on the address buss, sets the R/W line HI to indicate a read and then takes the data off the data buss. By indicating a READ the CPU requests the memory location at 'address' to place the data on the buss. Thus, in response to this request, ADC places on the data buss the data from the last analog to digital conversion which was carried out.

Appendix E
VIA data sheets

Although cryptic, data sheets contain all of the detailed information about a particular device. But, be warned!, they are sometimes inaccurate due to typos, poor editing and even slight misrepresentation of the capabilities. These following data sheets for the 6522 manufactured by Rockwell seem to be accurate.

R6522
VERSATILE INTERFACE ADAPTER (VIA)

DESCRIPTION

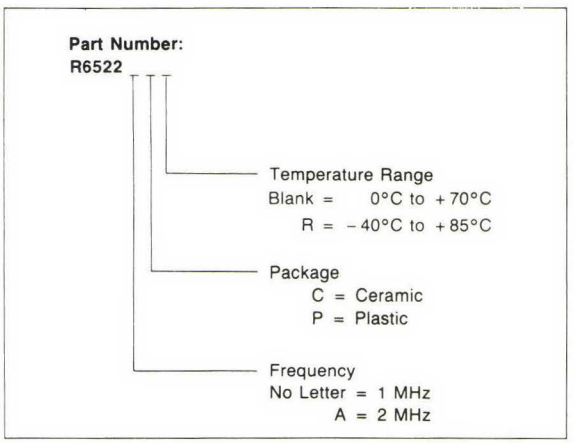
The R6522 Versatile Interface Adapter (VIA) is a very flexible I/O control device. In addition, this device contains a pair of very powerful 16-bit interval timers, a serial-to-parallel/parallel-to serial shift register and input data latching on the peripheral ports. Expanded handshaking capability allows control of bidirectional data transfers between VIA's in multiple processor systems.

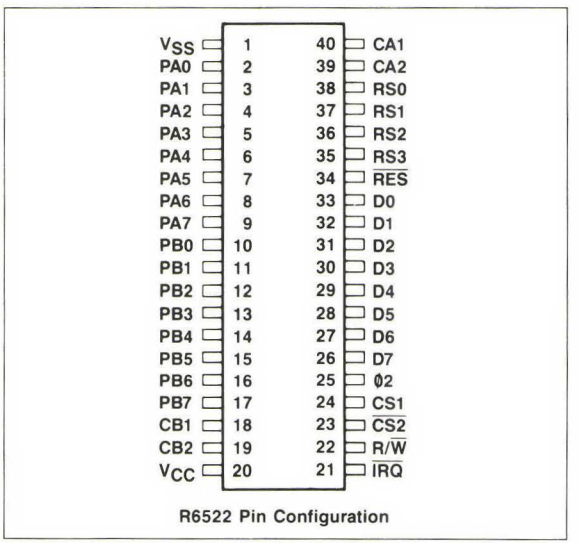
Control of peripheral devices is handled primarily through two 8-bit bidirectional ports. Each line can be programmed as either an input or an output. Several peripheral I/O lines can be controlled directly from the interval timers for generating programmable frequency square waves or for counting externally generated pulses. To facilitate control of the many powerful features of this chip, an interrupt flag register, an interrupt enable register and a pair of function control registers are provided.

FEATURES

- Two 8-bit bidirectional I/O ports
- Two 16-bit programmable timer/counters
- Serial data port
- TTL compatible
- CMOS compatible peripheral control lines
- Expanded "handshake" capability allows positive control of data transfers between processor and peripheral devices.
- Latched output and input registers
- 1 MHz and 2 MHz operation
- Single +5V power supply

ORDERING INFORMATION





R6522 Pin Configuration

INTERFACE SIGNALS

RESET ($\overline{RES}$)

A low reset ($\overline{RES}$) input clears all R6522 internal registers to logic 0 (except T1 and T2 latches and counters and the Shift Register). This places all peripheral interface lines in the input state, disables the timers, shift register, etc. and disables interrupting from the chip.

INPUT CLOCK (PHASE 2)

The input clock is the system $\phi 2$ clock and triggers all data transfers between processor bus and the R6522.

READ/WRITE (R/$\overline{W}$)

The direction of the data transfers between the R6522 and the system processor is controlled by the R/$\overline{W}$ line in conjunction with the CS1 and $\overline{CS2}$ inputs. When R/$\overline{W}$ is low, (write operation) and the R6522 is selected, data is transferred from the processor bus into the selected R6522 register. When R/$\overline{W}$ is high, (read operation) and the R6522 is selected, data is transferred from the selected R6422 register to the processor bus.

DATA BUS (D0–D7)

The eight bidirectional data bus lines transfer data between the R6522 and the system processor bus. During read cycles, the contents of the selected R6522 register are placed on the data bus lines. During write cycles, these lines are high-impedance inputs and data is transferred from the processor bus into the selected register. When the R6522 is not selected, the data bus lines are high-impedance.

CHIP SELECTS (CS1, $\overline{CS2}$)

The two chip select inputs are normally connected to processor address lines either directly or through decoding. The selected R6522 register is accessed when CS1 is high and $\overline{CS2}$ is low.

REGISTER SELECTS (RS0–RS3)

The coding of the four Register Select inputs select one of the 16 internal registers of the R6522, as shown in Table 1.

INTERRUPT REQUEST ($\overline{IRQ}$)

The Interrupt Request output goes low whenever an internal interrupt flag is set and the corresponding interrupt enable bit is a logic 1. This output is open-drain to allow the interrupt request signal to be wire-OR'ed with other equivalent signals in the system.

PERIPHERAL PORT A (PA0–PA7)

Port A consists of eight lines which can be individuallly programmed to act as inputs or outputs under control of Data Direction Register A. The polarity of output pins is controlled by an Output Register and input data may be latched into an internal register under control of the CA1 line. All of these modes of operation are controlled by the system processor through the internal control registers. These lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. Figure 2 illustrates the output circuit.

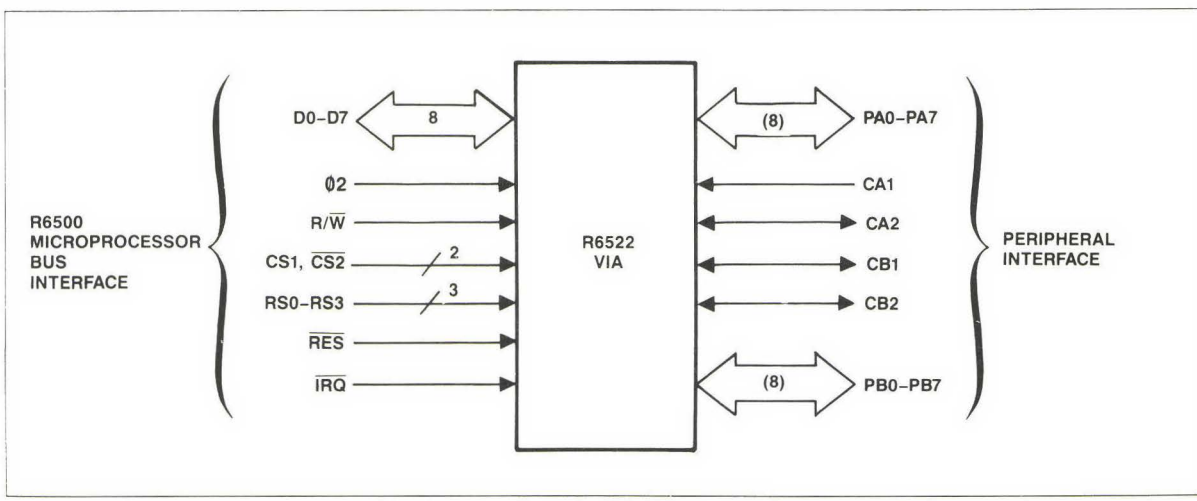


Figure 1. R6522 VIA Interface Signals

PORT A CONTROL LINES (CA1, CA2)

The two Port A control lines act as interrupt inputs or as handshake outputs. Each line controls an internal interrupt flag with a corresponding interrupt enable bit. In addition, CA1 controls the latching of data on Port A input lines. CA1 is a high-impedance input only while CA2 represents one standard TTL load in the input mode. CA2 will drive one standard TTL load in the output mode.

PORT B (PB0–PB7)

Peripheral Port B consists of eight bidirectional lines which are controlled by an output register and a data direction register in much the same manner as the Port A. In addition, the polarity of the PB7 output signal can be controlled by one of the interval timers while the second timer can be programmed to count pulses on the PB6 pin. Port B lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. In addition, they are capable of sourcing 1.0 mA at 1.5 Vdc in the output mode to allow the outputs to directly drive Darlington transistor circuits. Figure 3 is the circuit schematic.

PORT B CONTROL LINES (CB1, CB2)

The Port B control lines act as interrupt inputs or as handshake outputs. As with CA1 and CA2, each line controls an interrupt flag with a corresponding interrupt enable bit. In addition, these lines act as a serial port under control of the Shift Register. These lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. CB2 *can* also drive a Darlington transistor circuit; however, CB1 *cannot*.

Table 1. R6522 Register Addressing

Register Number	RS Coding				Register Desig.	Register/Description	
	RS3	RS2	RS1	RS0		Write (R/$\overline{W}$ = L)	Read (R/$\overline{W}$ = H)
0	0	0	0	0	ORB/IRB	Output Register B	Input Register B
1	0	0	0	1	ORA/IRA	Output Register A	Input Register A
2	0	0	1	0	DDRB	Data Direction Register B	
3	0	0	1	1	DDRA	Data Direction Register A	
4	0	1	0	0	T1C-L	T1 Low-Order Latches	T1 Low-Order Counter
5	0	1	0	1	T1C-H	T1 High-Order Counter	
6	0	1	1	0	T1L-L	T1 Low-Order Latches	
7	0	1	1	1	T1L-H	T1 High-Order Latches	
8	1	0	0	0	T2C-L	T2 Low-Order Latches	T2 Low-Order Counter
9	1	0	0	1	T2C-H	T2 High-Order Counter	
10	1	0	1	0	SR	Shift Register	
11	1	0	1	1	ACR	Auxiliary Control Register	
12	1	1	0	0	PCR	Peripheral Control Register	
13	1	1	0	1	IFR	Interrupt Flag Register	
14	1	1	1	0	IER	Interrupt Enable Register	
15	1	1	1	1	ORA/IRA	Output Register A*	Input Register A*

NOTE: *Same as Register 1 except no handshake.

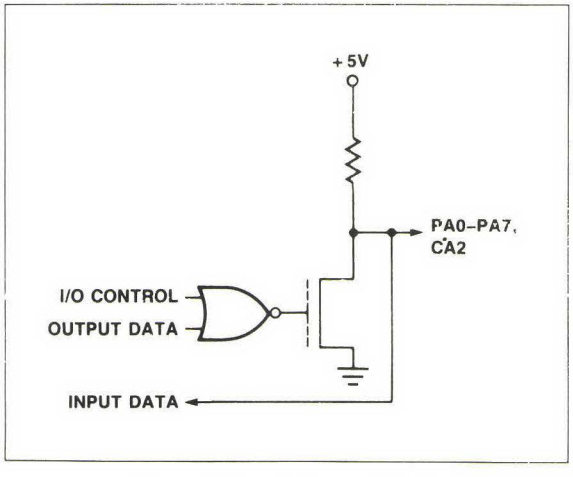


Figure 2. Port A Output Circuit

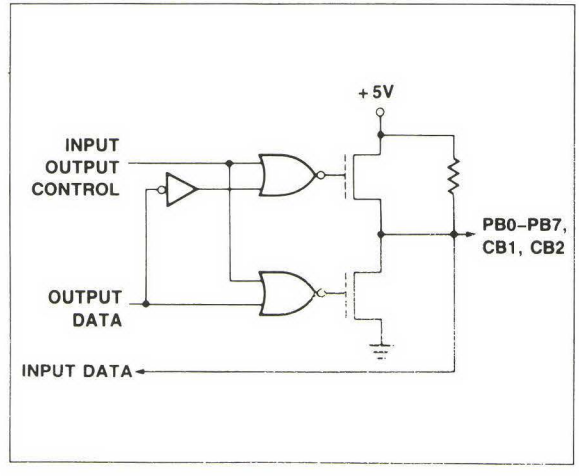


Figure 3. Port B Output Circuit

FUNCTIONAL DESCRIPTION

The internal organization of the R6522 VIA is illustrated in Figure 4.

PORT A AND PORT B OPERATION

The R6522 VIA has two 8-bit bidirectional I/O ports (Port A and Port B) and each port has two associated control lines.

Each 8-bit peripheral port has a Data Direction Register (DDRA, DDRB) for specifying whether the peripheral pins are to act as inputs or outputs. A 0 in a bit of the Data Direction Register causes the corresponding peripheral pin to act as an input. A 1 causes the pin to act as an output.

Each peripheral pin is also controlled by a bit in the Output Register (ORA, ORB) and the Input Register (IRA, IRB). When the pin is programmed as an output, the voltage on the pin is controlled by the corresponding bit of the Output Register. A 1 in the Output Register causes the output to go high, and a "0" causes the output to go low. Data may be written into Output Register bits corresponding to pins which are programmed as inputs. In this case, however, the output signal is unaffected.

Reading a peripheral port causes the contents of the Input Register (IRA, IRB) to be transferred onto the Data Bus. With input latching disabled, IRA will always reflect the levels on the PA pins. With input latching enabled, IRA will reflect the levels on the PA pins at the time the latching occurred (via CA1).

The IRB register operates similar to the IRA register. However, for pins programmed as outputs there is a difference. When reading IRA, the *level on the pin* determines whether a 0 or a 1 is sensed. When reading IRB, however, the bit stored in the *output register,* ORB, is the bit sensed. Thus, for outputs which have large loading effects and which pull an output "1" down or which pull an output "0" up, reading IRA may result in reading a "0" when a "1" was actually programmed, and reading a "1" when a "0" was programmed. Reading IRB, on the other hand, will read the "1" or "0" level actually programmed, no matter what the loading on the pin.

Figures 5 through 8 illustrate the formats of the port registers. In addition, the input latching modes are selected by the Auxiliary Control Register (Figure 14).

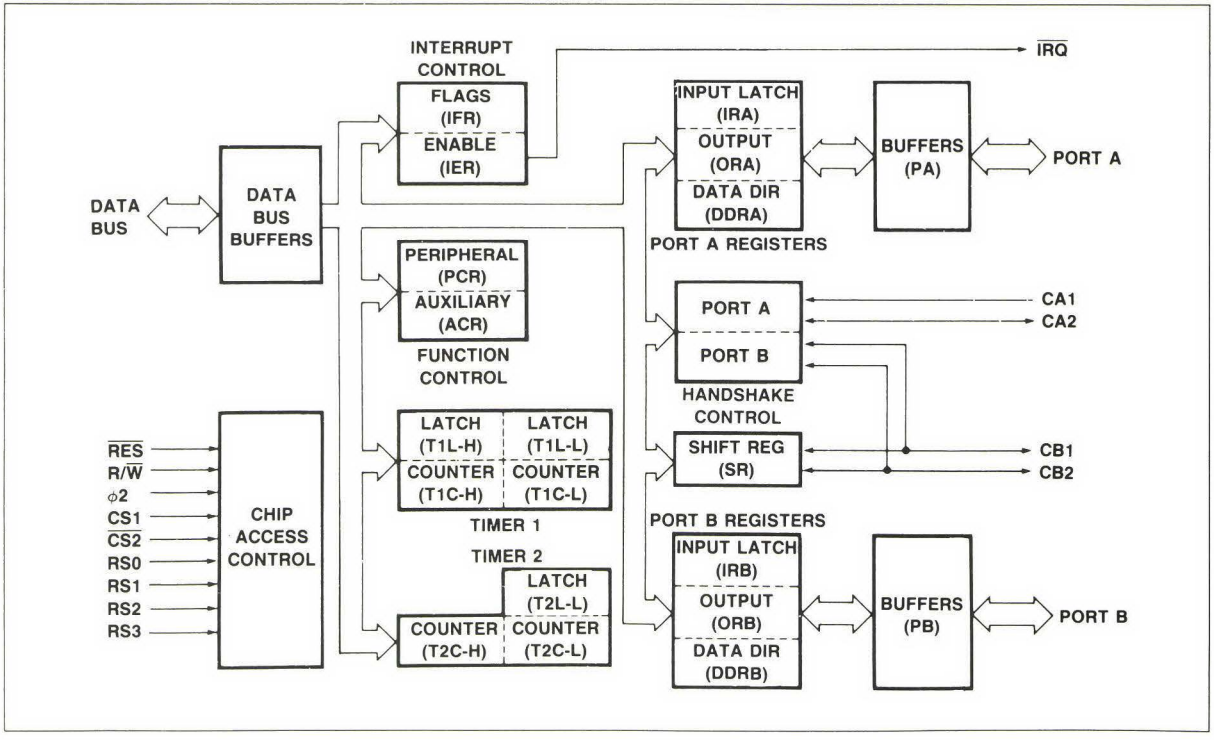


Figure 4. R6522 VIA Block Diagram

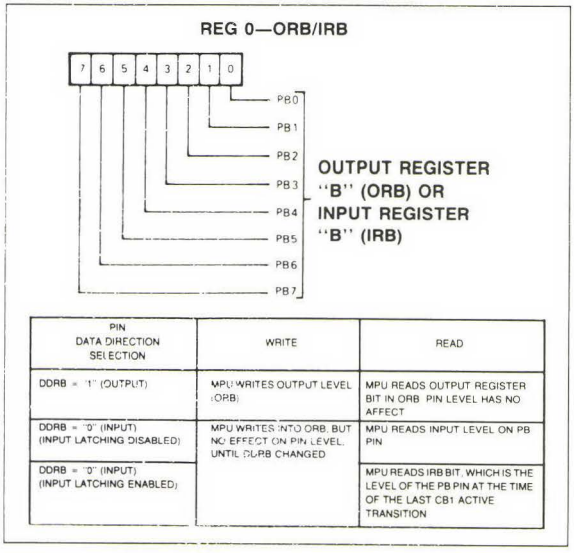
HANDSHAKE CONTROL OF DATA TRANSFERS

The R6522 allows positive control of data transfers between the system processor and peripheral devices through the operation of "handshake" lines. Port A lines (CA1, CA2) handshake data on both a read and a write operation while the Port B lines (CB1, CB2) handshake on a write operation only.

Read Handshake

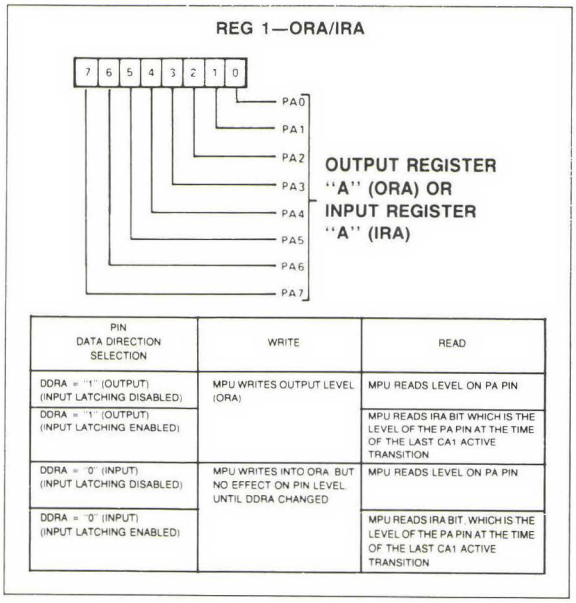
Positive control of data transfers from peripheral devices into the system processor can be accomplished very effectively using Read Handshaking. In this case, the peripheral device must generate the equivalent of a "Data Ready" signal to the processor signifying that valid data is present on the peripheral port. This signal normally interrupts the processor, which then reads the data, causing generation of a "Data Taken" signal. The peripheral device responds by making new data available. This process continues until the data transfer is complete.

In the R6522, automatic "Read" Handshaking is possible on the Peripheral A port only. The CA1 interrupt input pin accepts the "Data Ready" signal and CA2 generates the "Data Taken" signal. The "Data Ready" signal will set an internal flag which may interrupt the processor or which may be polled under program control. The "Data Taken" signal can either be a pulse or a level which is set low by the system processor and is cleared by the "Data Ready" signal. These options are shown in Figure 9 which illustrates the normal Read Handshake sequence.



PIN DATA DIRECTION SELECTION	WRITE	READ
DDRB = "1" (OUTPUT)	MPU WRITES OUTPUT LEVEL (ORB)	MPU READS OUTPUT REGISTER BIT IN ORB. PIN LEVEL HAS NO AFFECT
DDRB = "0" (INPUT) (INPUT LATCHING DISABLED)	MPU WRITES INTO ORB, BUT NO EFFECT ON PIN LEVEL UNTIL DDRB CHANGED	MPU READS INPUT LEVEL ON PB PIN
DDRB = "0" (INPUT) (INPUT LATCHING ENABLED)		MPU READS IRB BIT, WHICH IS THE LEVEL OF THE PB PIN AT THE TIME OF THE LAST CB1 ACTIVE TRANSITION

Figure 5. Output Register B (ORB), Input Register B (IRB)



PIN DATA DIRECTION SELECTION	WRITE	READ
DDRA = "1" (OUTPUT) (INPUT LATCHING DISABLED)	MPU WRITES OUTPUT LEVEL (ORA)	MPU READS LEVEL ON PA PIN
DDRA = "1" (OUTPUT) (INPUT LATCHING ENABLED)		MPU READS IRA BIT WHICH IS THE LEVEL OF THE PA PIN AT THE TIME OF THE LAST CA1 ACTIVE TRANSITION
DDRA = "0" (INPUT) (INPUT LATCHING DISABLED)	MPU WRITES INTO ORA, BUT NO EFFECT ON PIN LEVEL UNTIL DDRA CHANGED	MPU READS LEVEL ON PA PIN
DDRA = "0" (INPUT) (INPUT LATCHING ENABLED)		MPU READS IRA BIT, WHICH IS THE LEVEL OF THE PA PIN AT THE TIME OF THE LAST CA1 ACTIVE TRANSITION

Figure 6. Output Register A (ORA), Input Register A (IRA)

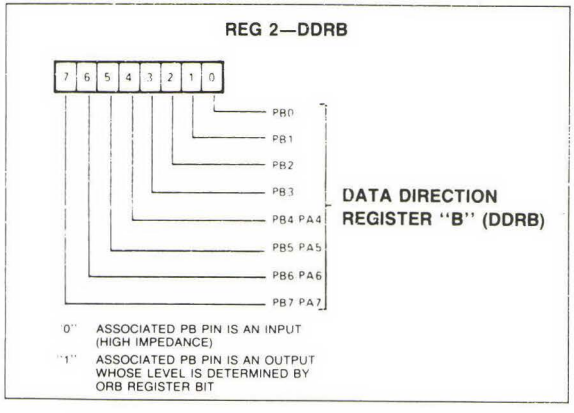


Figure 7. Data Direction Register B (DDRB)

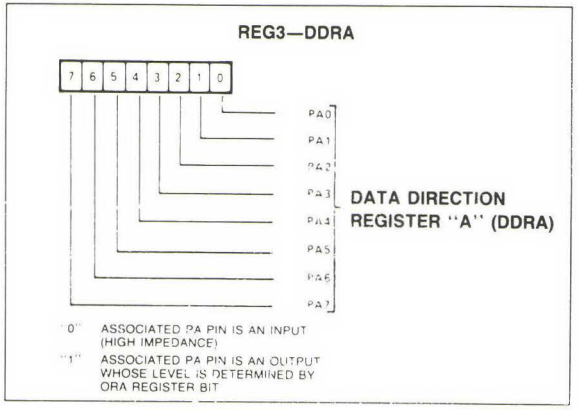


Figure 8. Data Direction Register A (DDRA)

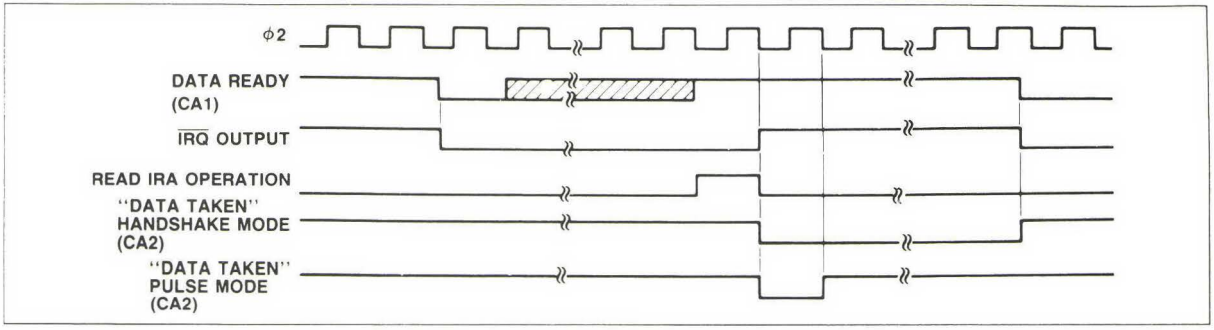


Figure 9. Read Handshake Timing (Port A Only)

Write Handshake

The sequence of operations which allows handshaking data from the system processor to a peripheral device is very similar to that described for Read Handshaking. However, for Write Handshaking, the R6522 generates the "Data Ready" signal and the peripheral device must respond with the "Data Taken" signal. This can be accomplished on both the PA port and the PB port on the R6522. CA2 or CB2 act as a "Data Ready" output in either the handshake mode or pulse mode and CA1 or CB1 accept the "Data Taken" signal from the peripheral device, setting the interrupt flag and clearing the "Data Ready" output. This sequence is shown in Figure 10.

Selection of operating modes for CA1, CA2, CB1, and CB2 is accomplished by the Peripheral Control Register (Figure 11).

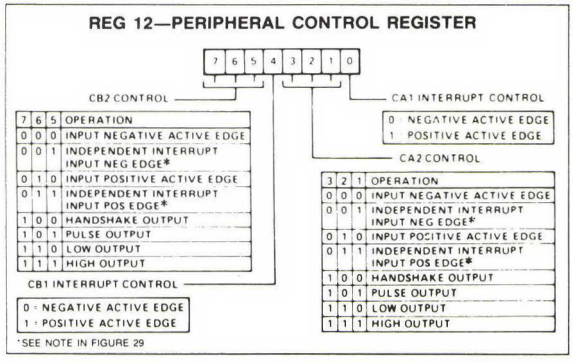


Figure 11. Peripheral Control Register (PCR)

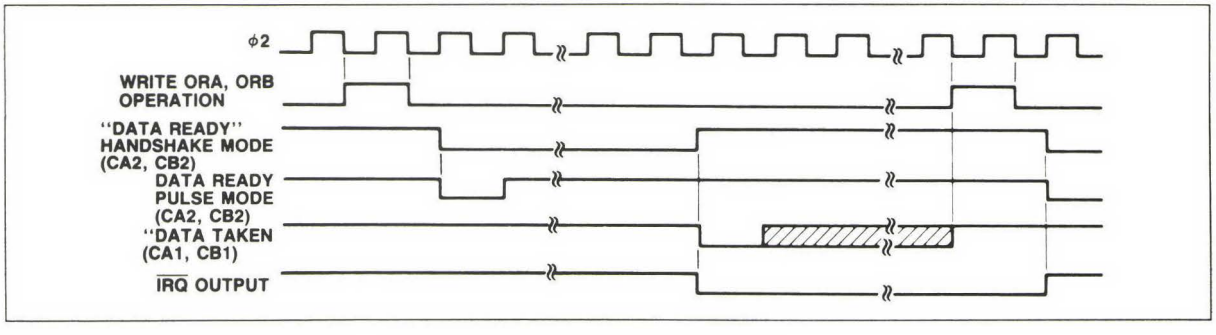


Figure 10. Write Handshake Timing

COUNTER/TIMERS

There are two independent 16-bit counter/timers (called Timer 1 and Timer 2) in the R6522. Each timer is controlled by writing bits into the Auxiliary Control Register (ACR) to select the mode of operation (Figure 14.

Timer 1 Operation

Interval Timer T1 consists of two 8-bit latches (Figure 12) and a 16-bit counter (Figure 13). The latches store data which is to be loaded into the counter. After loading, the counter decrements at Ø2 clock rate. Upon reaching zero, an interrupt flag is set, and $\overline{IRQ}$ goes low if the T1 interrupt is enabled. Timer 1 then disables any further interrupts, automatically transers the contents of the latches into the counter and continues to decrement. In addition, the timer may be programmed to invert the output signal on peripheral pin PB7 each time it "times-out." Each of these modes is discussed separaely below.

Note that the processor does not write directly into the low-order counter (T1C-L). Instead, this half of the counter is loaded automatically from the low order latch (T1L-L) when the processor writes into the high order counter (T1C-H). In fact, it may not be necessary to write to the low order counter in some applications since the timing operation is triggered by writing to the high order latch.

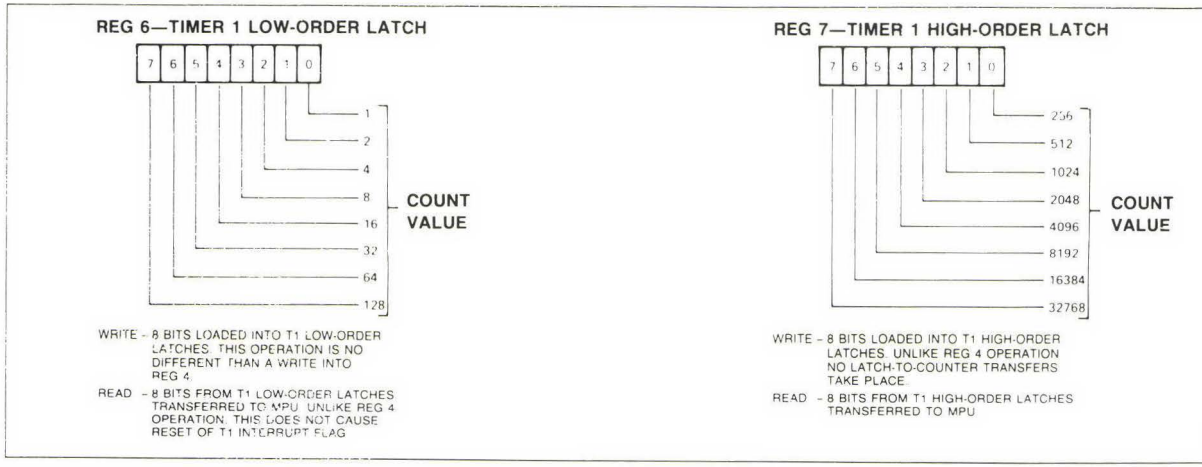


Figure 12. Timer 1 (T1) Latch Registers

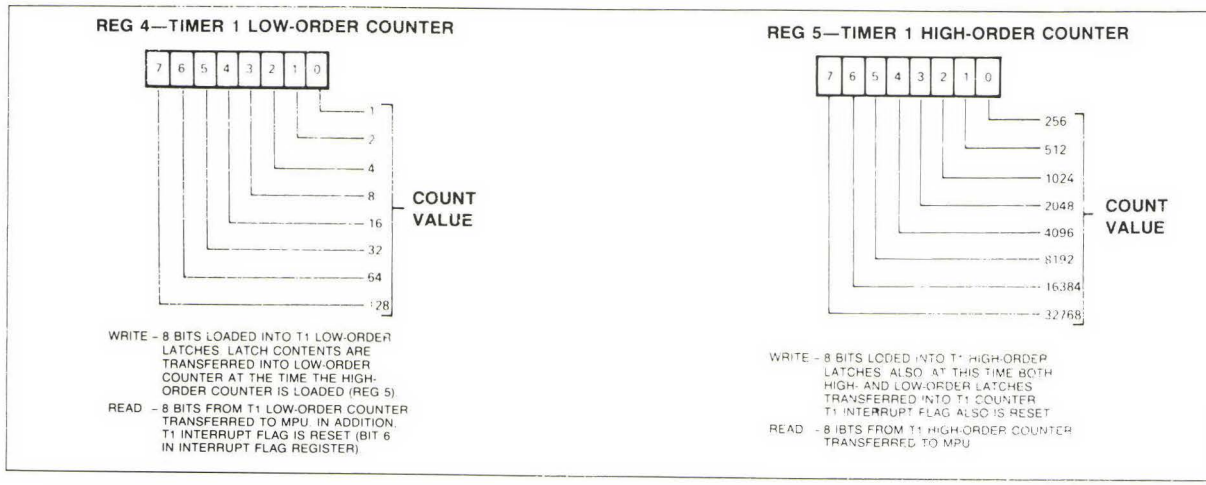


Figure 13. Timer 1 (T1) Counter Registers

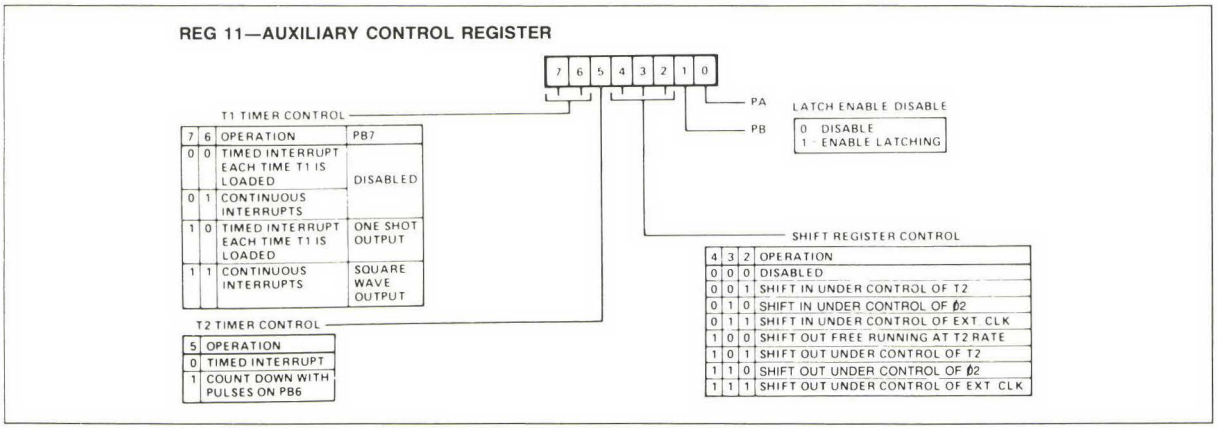


Figure 14. Auxiliary Control Register (ACR)

Timer 1 One-Shot Mode

The Timer 1 one-shor mode generates a single interrupt for each timer load operation. As with any interval timer, the delay between the "write T1C-H" operation and generation of the processor interrupt is a direct function of the data loaded into the timing counter. In addition to generating a single interrupt, Timer 1 can be programmed to produce a single negative pulse on the PB7 peripheral pin. With the output enabled (ACR7 = 1) a "write T1C-H" operation will cause PB7 to go low. PB7 will return high when Timer 1 times out. The result is a single programmable width pulse.

Timing for the R6522 interval timer one-shot modes is shown in Figure 15.

In the one-shot mode, writing into the T1L-H has no effect on the operation of Timer 1. However, it will be necessary to assure that the low order latch contains the proper data before initiating the count-down with a "write T1C-H" operation. When the processor writes into the high order counter (T1C-H), the T1 interrupt flag will be cleared, the contents of the low order latch will be transferred into the low order counter, and the timer will begin to decrement at system clock rate. If the PB7 output is enabled, this signal will go low on the Ø2 following the write operation. When the counter reaches zero, the T1 interrupt flag will be set, the $\overline{IRQ}$ pin will go low (interrupt enabled), and the signal on PB7 will go high. At this time the counter will continue to decrement at system clock rate. This allows the system processor to read the contents of the counter to determine the time since interrupt. However, the T1 interrupt flag cannot be set again unless it has been cleared as described in this specification.

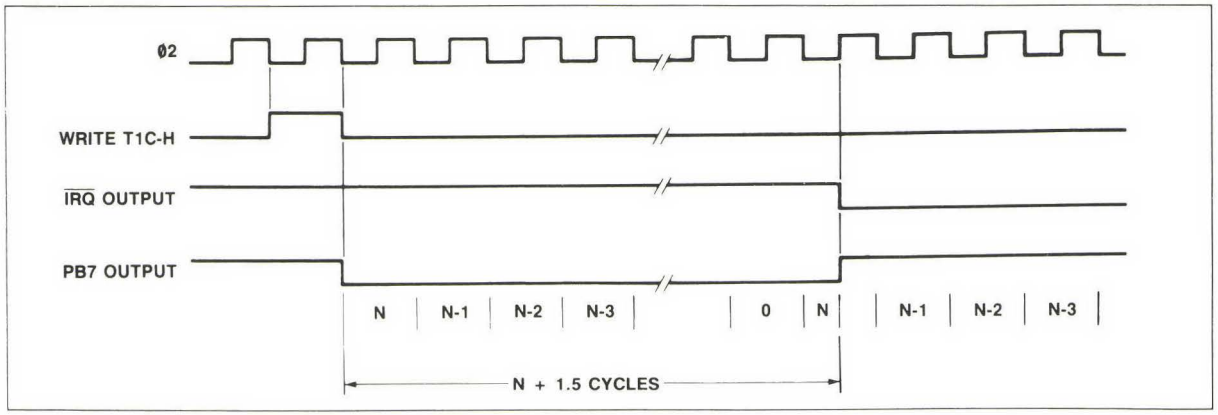


Figure 15. Timer 1 One-Shot Mode Timing

Timer 1 Free-Run Mode

The most important advantage associated with the latches in T1 is the ability to produce a continuous series of evenly spaced interrupts and the ability toproduce a square wave on PB7 whose frequency is not affected by variations in the processor interrupt response time This is accomplished in the "free-running" mode.

In the free-running mode, the interrupt flag is set and the signal on PB7 is inverted each time the counter reaches zero, at which time the timer automatically transfers the contents of the latch into the counter (16 bits) and continues to decrement from there. The interrupt flag can be cleared by writing T1C-H, by reading T1C-L, or by writing directly into the flag as described later. However, it is not necessary to rewrite the timer to enable setting the interrupt flag on the next time-out.

All interval timers in the R6522 are "re-triggerable." Rewriting the counter will always re-initialize the time-out period. In fact, the time-out can be prevented completely if the processor continues to rewrite the timer before it reaches zero. Timer 1 will operate in this manner if the processor writes into the high order counter (T1C-H). However, by loading the latches only, the processor can access the timer during each down-counting operation without affecting the time-out in process. Instead, the data loaded into the latches will determine the length of the next time-out period. This capability is particularly valuable in the free-running mode with the output enabled. In this mode, the signal on PB7 is inverted and the interrupt flag is set with each time-out. By responding to the interrupts with new data for the latches, the processor can determine the period of the next half cycle during each half cycle of the output signal on PB7. In this manner, very complex waveforms can be generated.

A precaution to take in the use of PB7 as the timer output concerns the Data Direction Register contents for PB7. *Both* DDRB bit 7 and ACR bit 7 must be 1 for PB7 to function as the timer output. If one is 1 and the other is 0, then PB7 functions as a normal output pin, controlled by ORB bit 7.

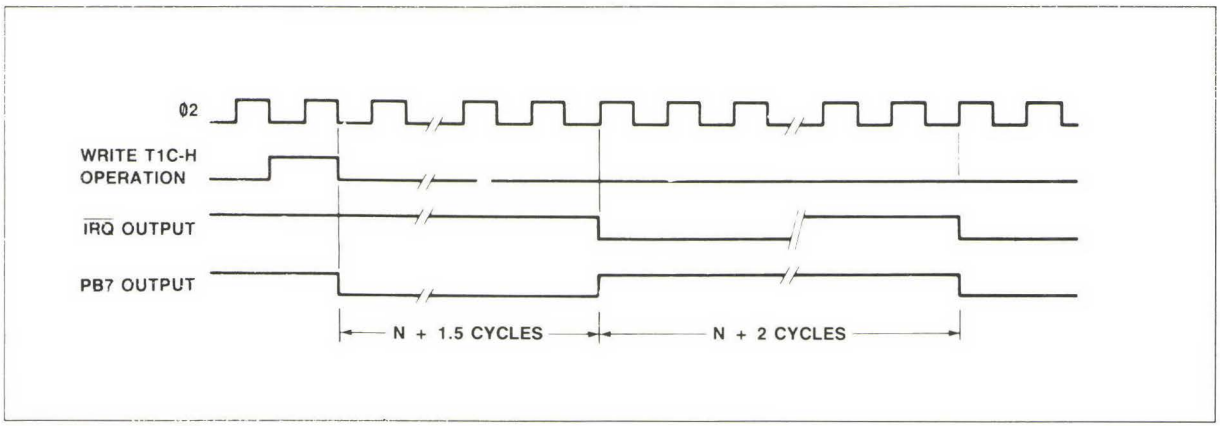


Figure 16. Timer 1 Free-Run Mode Timing

Timer 2 Operation

Timer 2 operates as an interval timer (in the "one-slot" mode only), or as a counter for counting negative pulses on the PB6 peripheral pin. A single control bit in the Auxiliary Control Register selects between these two modes. This timer is comprised of a "write-only" lower-order latch (T2L-L), a "read-only" low-order counter (T2C-L) and a read/write high order counter (T2C-H). The counter registers act as a 16-bit counter which decrements at Ø2 rate. Figure 17 illustrates the T2 Latch/Counter Registers.

Timer 2 One-Shot Mode

As an interval timer, T2 operates in the "one-shot" mode similar to Time 1. In this mode, T2 provides a single interrupt for each "write T2C-H" operation. After timing out, the counter will continue to decrement. However, setting of the interrupt flag is disabled after initial time-out so that it will not be set by the counter decrementing again through zero. The processor must rewrite T2C-H to enable setting of the interrupt flag. The interrupt flag is cleared by reading T2C-L or by writing T2C-H. Timing for this operation is shown in Figure 18.

Timer 2 Pulse Counting Mode

In the pulse counting mode, T2 counts a predetermined number of negative-going pulses on PB6. This is accomplished by first loading a number into T2. Writing into T2C-H clears the interrupt flag and allows the counter to decrement each time a pulse is applied to PB6. The interrupt flag is set when T2 counts down past zero. The counter will then continue to decrement with each pulse on PB6. However, it is necessary to rewrite T2C-H to allow the interrupt flag to set on a subsequent time-out. Timing for this mode is shown in Figure 19. The pulse must be low on the leading edge of Ø2.

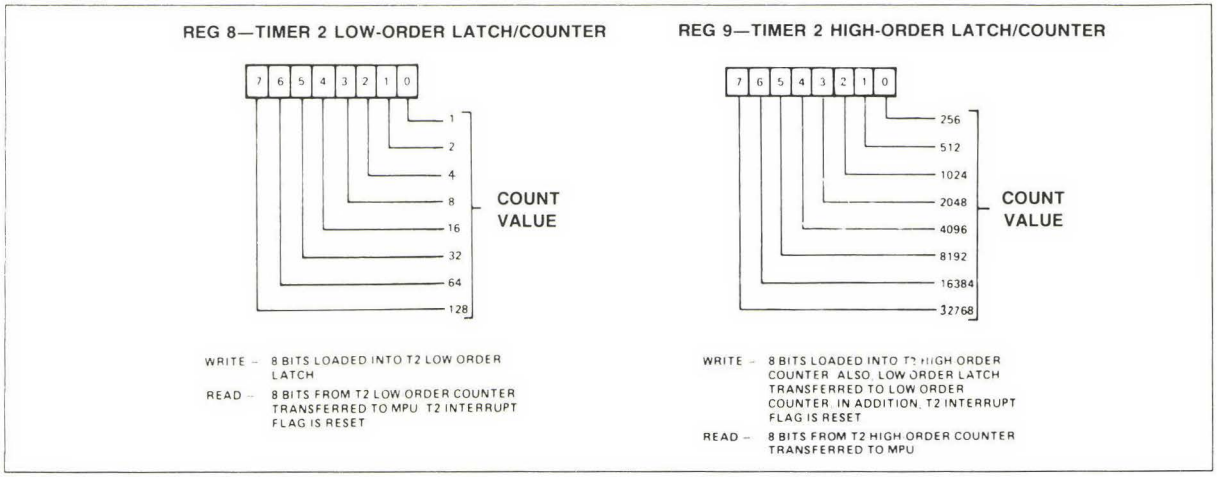


Figure 17. Timer 2 (T2) Latch/Counter Registers

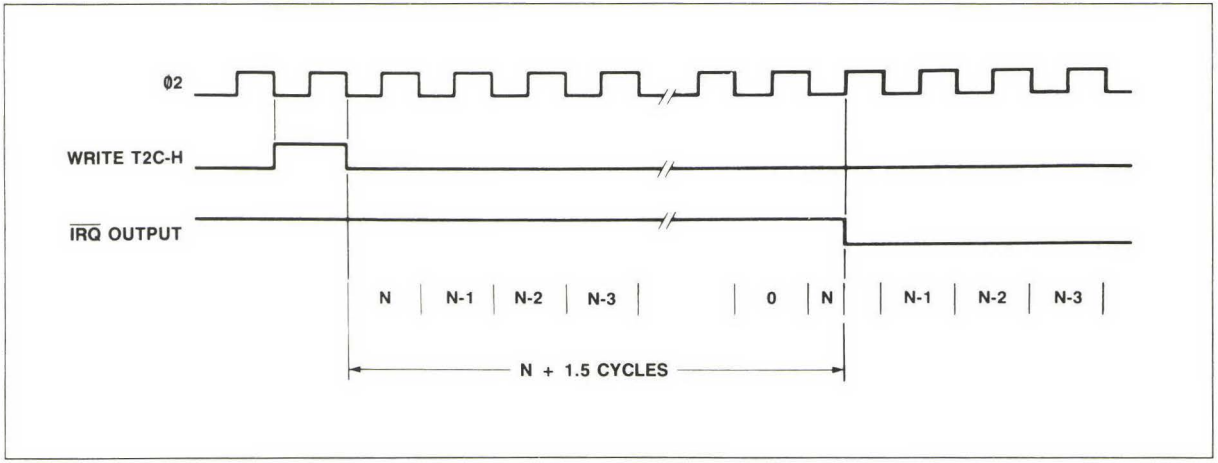


Figure 18. Timer 2 One-Shot Mode Timing

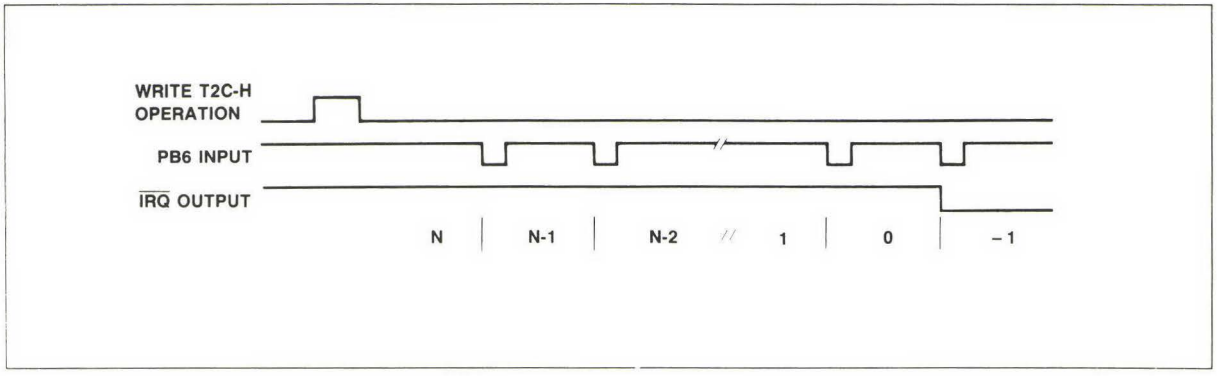


Figure 19. Timer 2 Pulse Counting Mode

SHIFT REGISTER OPERATION

The Shift Register (SR) performs serial data transfers into and out of the CB2 pin under control of an internal modulo-8 counter. Shift pulses can be applied to the CB1 pin from an external source or, with the proper mode selection, shift pulses generated internally will appear on the CB1 pin for controlling external devices.

The control bits which select the various shift register operating modes are located in the Auxiliary Control Register. Figure 20 illustrates the configuration of the SR data bits and Figure 21 shows the SR control bits of the ACR.

SR Mode 0 — Disabled

Mode 0 disables the Shift Register. In this mode the microprocessor can write or read the SR and the SR will shift on each CB1 positive edge shifting in the value on CB2. In this mode the SR interrupt Flag is disabled (held to a logic 0).

SR Mode 1 — Shift In Under Control of T2

In mode 1, the shifting rate is controlled by the low order 8 bits of T2 (Figure 22). Shift pulses are generated on the CB1 pin to control shifting in external devices. The time between transitions of this output clock is a function of the system clock period and the contents of the low order T2 latch (N).

The shifting operation is triggered by the read or write of the SR if the SR flag is set in the IFR. Otherwise the first shift will occur at the next time-out of T2 after a read or write of the SR. Data is shifted first into the low order bit of SR and is then shifted into the next higher order bit of the shift register on the negative-going edge of each clock pulse. The input data should change before the positive-going edge of the CB1 clock pulse. This data is shifted into the shift register during the Ø2 clock cycle following the positive-going edge of the CB1 clock pulse. After 8 CB1↓ clock pulses, the shift register interrupt flag will set and $\overline{IRQ}$ will go low.

SR Mode 2 — Shift In Under Ø2 Control

In mode 2, the shift rate is a direct function of the system clock frequency (Figure 23). CB1 becomes an output which generates shift pulses for controlling external devices. Timer 2 operates as an independent interval timer and has no effect on SR. The shifting operation is triggered by reading or writing the Shift Register. Data is shifted, first into bit 0 and is then shifted into the next higher order bit of the shift register on the trailing edge of each ϕ2 clock pulse. After 8 clock pulses, the shift register interrupt flag will be set, and the output clock pulses on CB1 will stop.

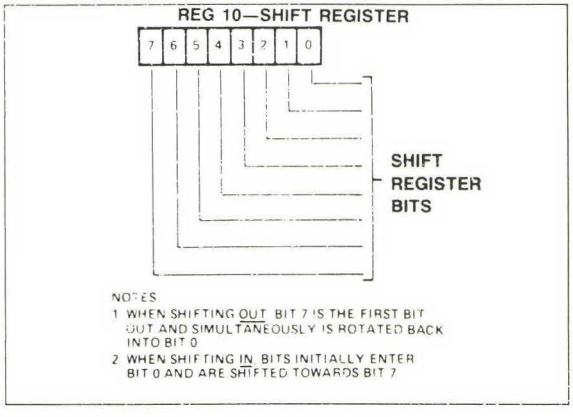
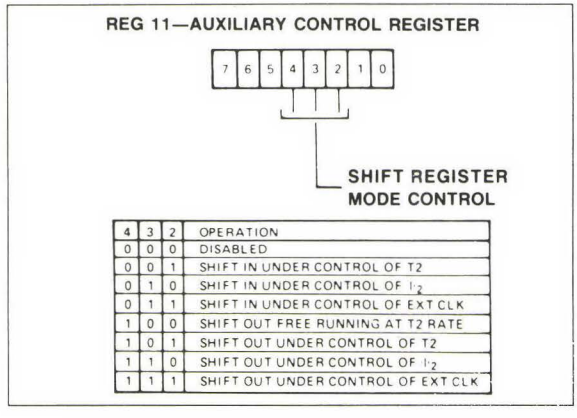


Figure 20. Shift Registers



4	3	2	OPERATION
0	0	0	DISABLED
0	0	1	SHIFT IN UNDER CONTROL OF T2
0	1	0	SHIFT IN UNDER CONTROL OF ϕ_2
0	1	1	SHIFT IN UNDER CONTROL OF EXT CLK
1	0	0	SHIFT OUT FREE RUNNING AT T2 RATE
1	0	1	SHIFT OUT UNDER CONTROL OF T2
1	1	0	SHIFT OUT UNDER CONTROL OF ϕ_2
1	1	1	SHIFT OUT UNDER CONTROL OF EXT CLK

Figure 21. Shift Register Modes

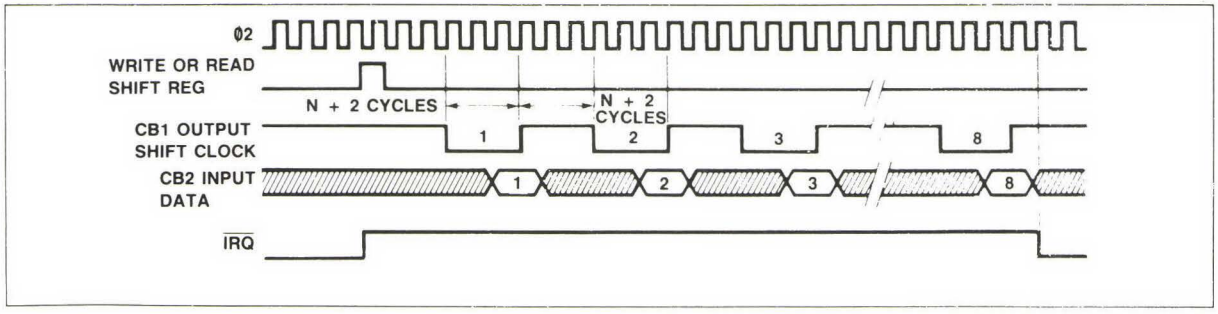


Figure 22. SR Mode 1 — Shift In Under T2 Control

SR Mode 3 — Shift In Under CB1 Control

In mode 3, external pin CB1 becomes an input (Figure 24). This allows an external device to load the shift register at its own pace. The shift register counter will interrupt the processor each time 8 bits have been shifted in. The shift register stops after 8 counts and must be reset to start again. Reading or writing the Shift Register resets the Interrrupt Flag and initializes the SR counter to count another 8 pulses.

Note that the data is shifted during the first system clock cycle following the posiive going edge of the CB1 shift pulse. For this reason, data must be held stable during the first full cycle following CB1 going high.

SR Mode 4 — Shift Out Under T2 Control (Free-Run)

Mode 4 is very similar to mode 5 in which the shifting rate is set by T2. However, in mode 4 the SR counter does not stop the shifting operation (Figure 25). Since the Shift Register bit 7 (SR7) is recirculated back into bit 0, the 8 bits loaded into the shift register will be clocked onto CB2 repetitively. In this mode the shift register counter is disabled.

SR Mode 5 — Shift Out Under T2 Control

In mode 5, the shift rate is controlled by T2 (as in mode 4). The shifting operation is triggerd by the read or write of the SR if the SR flag is set in the IFR (Figure 26). Otherwise the first shift will occur at the next time-out of T2 after a read or write of the SR. However, with each read or write of the shift register the SR Counter is reset and 8 bits are shifted onto CB2. At the same time, 8 shift pulses are generated on CB1 to control shifting in external devices. After the 8 shift pulses, the shifting is disabled, the SR Interrupt Flag is set and CB2 remains at the last data level.

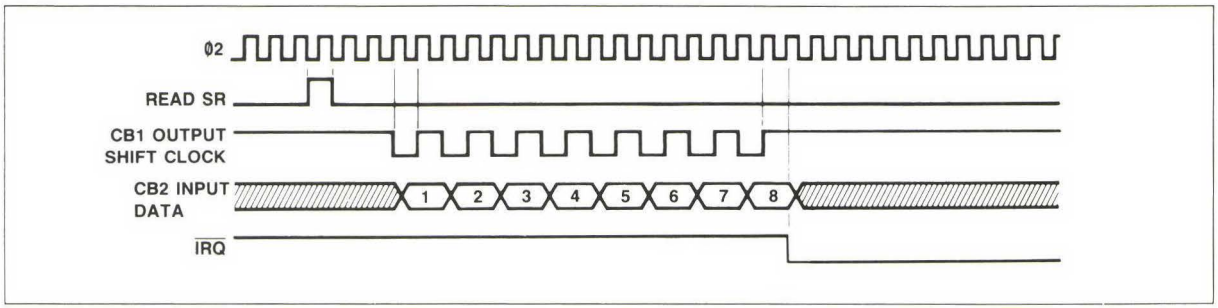


Figure 23. SR Mode 2 — Shift In Center Ø2 Control

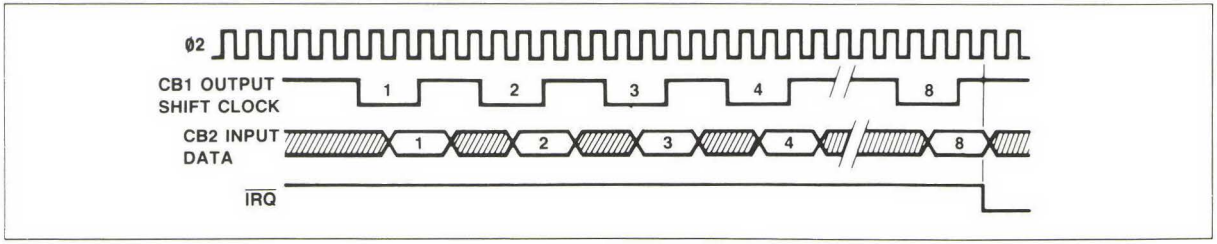


Figure 24. SR Mode 3 — Shift In Under CB1 Control

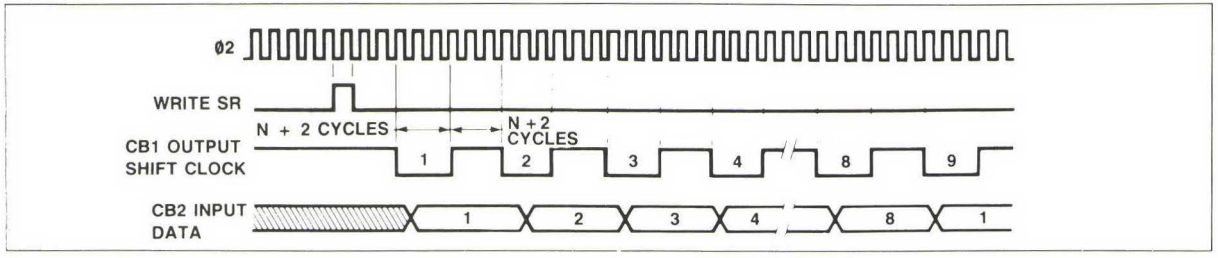


Figure 25. SR Mode 4 — Shift Our Under T2 Control (Free-Run)

SR Mode 6 — Shift OUt Under Ø2 Control

In mode 6, the shift rate is controlled by the Ø2 system clock (Figure 27).

SR Mode 7 — Shift Out Under CB1 Control

In mode 7, shifting is controlled by pulses applied to the CB1 pin by an external device (Figure 28). The SR counter sets the SR Interrupt Flag each time it counts 8 pulses but it does not disable the shifting function. Each time the microprocessor writes or reads the shift register, the SR Interrupt Flag is reset and the SR counter is initialized to begin counting the next 8 shift pulses on pin CB1. After 8 shift pulses, the Interrupt Flag is set. The microprocessor can then load the shift register with teh next byte of data.

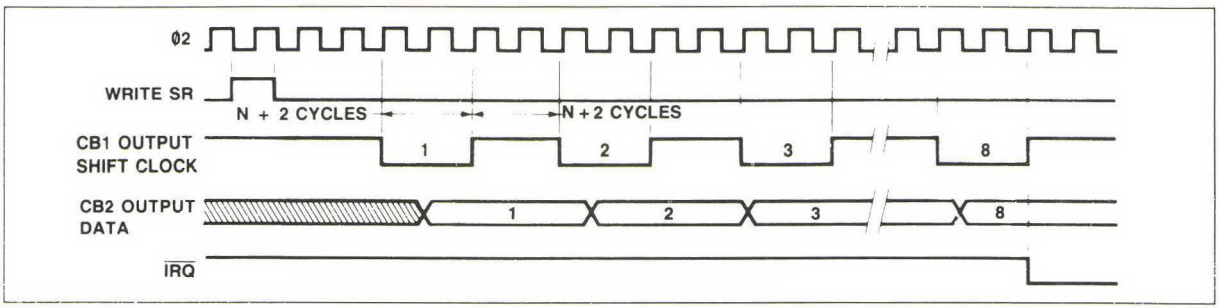


Figure 26. SR Mode 5 — Shift Out Under T2 Control

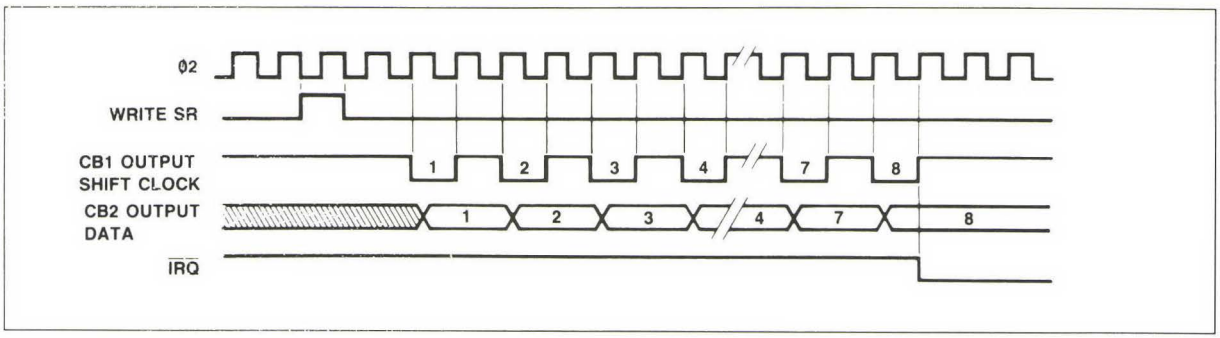


Figure 27. SR Mode 6 — Shift Out Under Ø2 Control

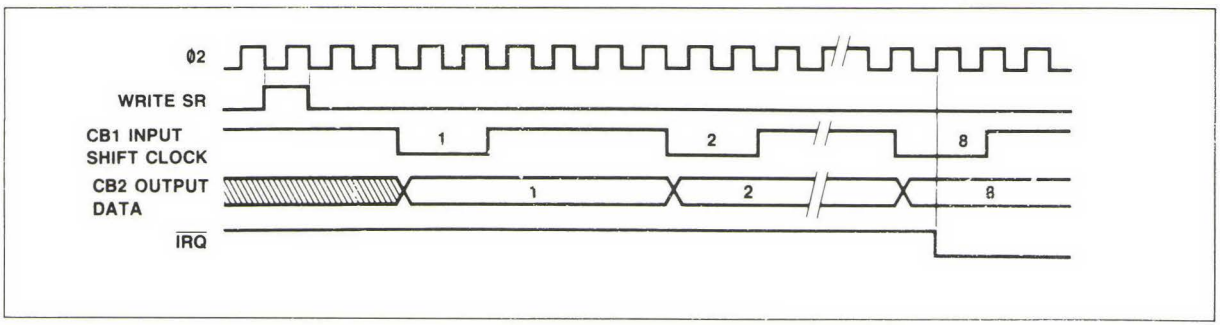


Figure 28. SR Mode 7 — Shift Out Under CB1 Control

Interrupt Operation

Controlling interrupts within the R6522 involves three principal operations. These are flagging the interrupts, enabling interrupts and signaling to the processor that an active interrupt exists within the chip. Interrupt flags are set in the Interrupt Flag Register (IFR) by conditions detected within the R6522 or on inputs to the R6522. These flags normally remain set until the interrupt has been serviced. To determine the source of an interrupt, the microprocessor must examine these flags in order, from highest to lowest priority.

Associated with each interrupt flag is an interrupt enable bit in the Interrupt Enable Register (IER). This can be set or cleared by the processor to enable interrupting the processor from the corresponding interrupt flag. If an interrupt flag is set to a logic 1 by an interrupting condition, and the corresponding interrupt enable bit is set to a 1, the Interrupt Request Output ($\overline{IRQ}$) will go low. $\overline{IRQ}$ is an "open-collector" output which can be "wire-OR'ed" with other devices in the system to interrupt the processor.

Interrupt Flag Register (IFR)

In the R6522, all the interrupt flags are contained in one register, i.e., the IFR (Figure 29). In addition, bit 7 of this register will be read as a logic 1 when an interrupt exists within the chip. This allows very convenient polling of several devices within a system to locate the source of an interrupt.

The Interrupt Flag Register (IRF) may be read directly by the processor. In addition, individual flag bits may be cleared by writing a "1" into the appropriate bit of the IFR. When the proper chip select and register signals are appplied to the chip, the contents of this register are placed on the data bus. Bit 7 indicates the status of the $\overline{IRQ}$ output. This bit corresponds to the logic function: $\overline{IRQ}$ = IFR6 × IER6 + IFR5 × IER5 + IFR4 × IER4 + IFR3 × IER3 + IFR2 × IER2 + IFR1 × IER1 + IFR0 × IER0.

Note:

× = logic AND, + = Logic OR.

The IFR bit 7 is not a flag. Therefore, this bit is not directly cleared by writing a logic 1 into it. It can only be cleared by clearing all the flags in the register or by disabling all the active interrupts as discussed in the next section.

Interrupt Enable Register (IER)

For each interrupt flag in IFR, there is a corresponding bit in the Interrupt Enable Register (IER) (Figure 30). Individual bits in the IER can be set or cleared to facilitate controlling individual interrupts without affecting others. This is accomplished by writing to the (IER) after bit 7 set or cleared to, in turn, set or clear selected enable bits. If bit 7 of the data placed on the system data bus during this write operation is a 0, each 1 in bits 6 through 0 clears the corresponding bit in the Interrupt Enable Register. For each zero in bits 6 through 0, the corresponding bit is unaffected.

Selected bits in the IER can be set by writing to the IER with bit 7 in the data word set to a 1. In this case, each 1 in bits 6 through 0 will set the corresponding bit. For each zero, the corresponding bit will be unaffected. This individual control of the setting and clearing operations allows very convenient control of the interrupts during system operation.

In addition to setting and clearing IER bits, the contents of this register can be read at any time. Bit 7 will be read as a logic 1, however.

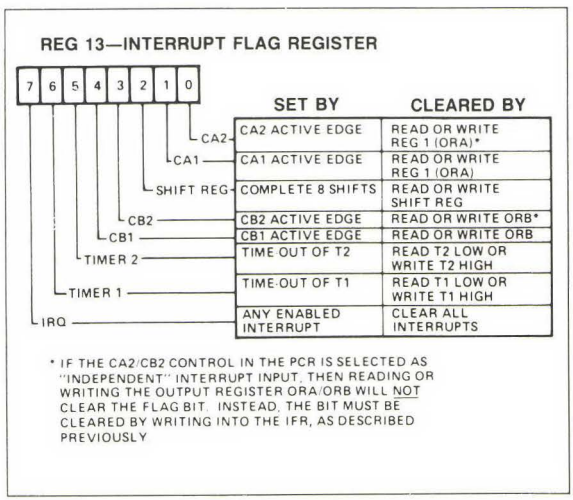


Figure 29. Interrupt Flag Register (IFR)

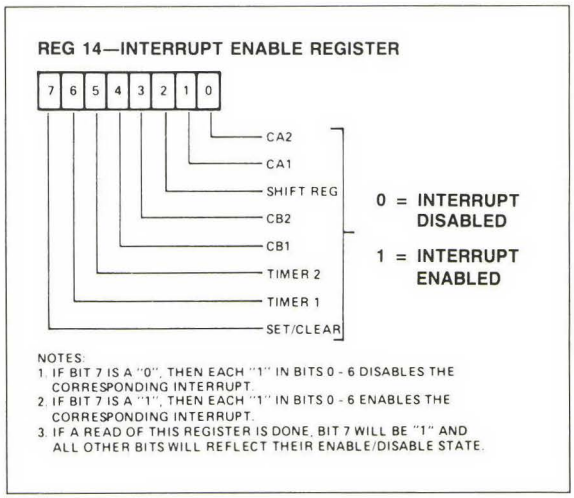


Figure 30. Interrupt Enable Register (IER)

PERIPHERAL INTERFACE CHARACTERISTICS

Symbol	Characteristic	Min.	Max.	Unit	Figure
t_r, t_f	Rise and Fall Time for CA1, CB1, CA2 and CB2 Input Signals	—	1.0	μs	—
t_{CA2}	Delay Time, Clock Negative Transition to CA2 Negative Transition (read handshake or pulse mode)	—	1.0	μs	31a, 31b
t_{RS1}	Delay Time, Clock Negative Transition to CA2 Positive Transition (pulse mode)	—	1.0	μs	31a
t_{RS2}	Delay Time, CA1 Active Transition to CA2 Positive Transition (handshake mode)	—	2.0	μs	31b
t_{WHS}	Delay Time, Clock Positive Transition to CA2 or CB2 Negative Transition (write handshake)	0.05	1.0	μs	31c, 31d
t_{DS}	Delay Time, Peripheral Data Valid to CB2 Negative Transition	0.20	1.5	μs	31c, 31d
t_{RS3}	Delay Time, Clock Positive Transition to CA2 or CB2 Positive Transition (pulse mode)	—	1.0	μs	31c
t_{RS4}	Delay Time, CA1 or CB1 Active Transition to CA2 or CB2 Positive Transition (handshake mode)	—	2.0	μs	31d
t_{21}	Delay Time Required from CA2 Output to CA1 Active Transition (handshake mode)	400	—	ns	31d
t_{IL}	Setup Time, Peripheral Data Valid to CA1 or CB1 Active Transition (input latching)	300	—	ns	31e
t_{AL}	CA1, CB1 Setup Prior to Transition to Arm Latch	300	—	ns	31e
t_{PDH}	Peripheral Data Hold After CA1, CB1 Transition	150	—	ns	31e
t_{SR1}	Shift-Out Delay Time — Time from ϕ_2 Falling Edge to CB2 Data Out	—	300	ns	31f
t_{SR2}	Shift-In Setup Time — Time from CB2 Data In to ϕ_2 Rising Edge	300	—	ns	31g
t_{SR3}	External Shift Clock (CB1) Setup Time Relative to ϕ_2 Trailing Edge	100	T_{CY}	ns	31g
t_{IPW}	Pulse Width — PB6 Input Pulse	2 × T_{CY}	—		31i
t_{ICW}	Pulse Width — CB1 Input Clock	2 × T_{CY}	—		31h
t_{IPS}	Pulse Spacing — PB6 Input Pulse	2 × T_{CY}	—		31i
t_{ICS}	Pulse Spacing — CB1 Input Pulse	2 × T_{CY}	—		31h

PERIPHERAL INTERFACE WAVEFORMS

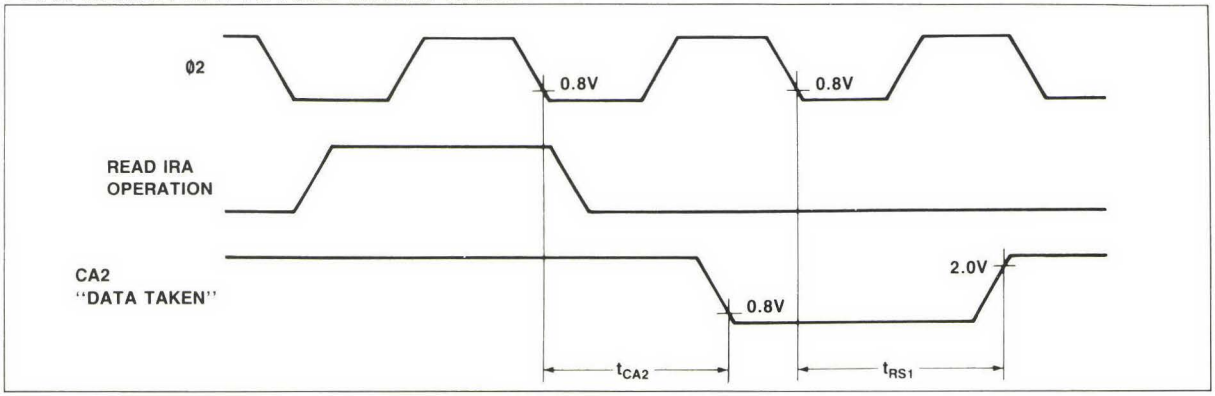


Figure 31a. CA2 Timing for Read Handshake, Pulse Mode

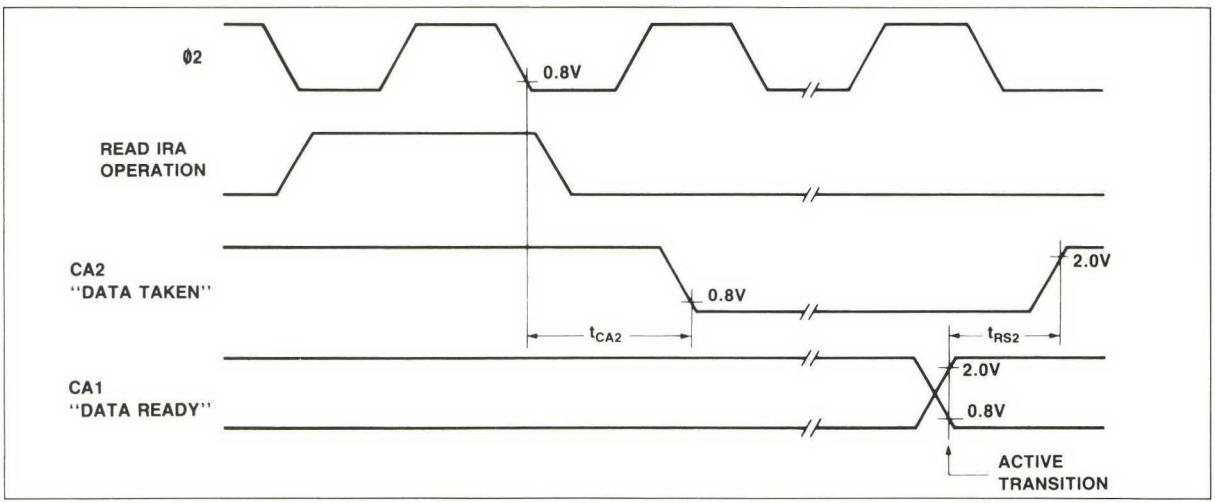


Figure 31b. CA2 Timing for Read Handshake, Handshake Mode

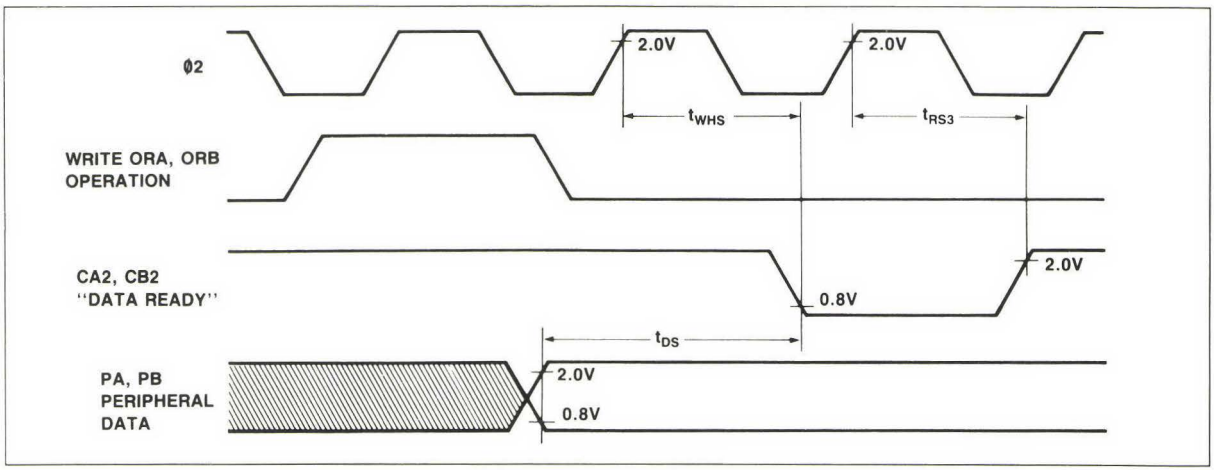


Figure 31c. CA2, CB2 Timing for Write Handshake, Pulse Mode

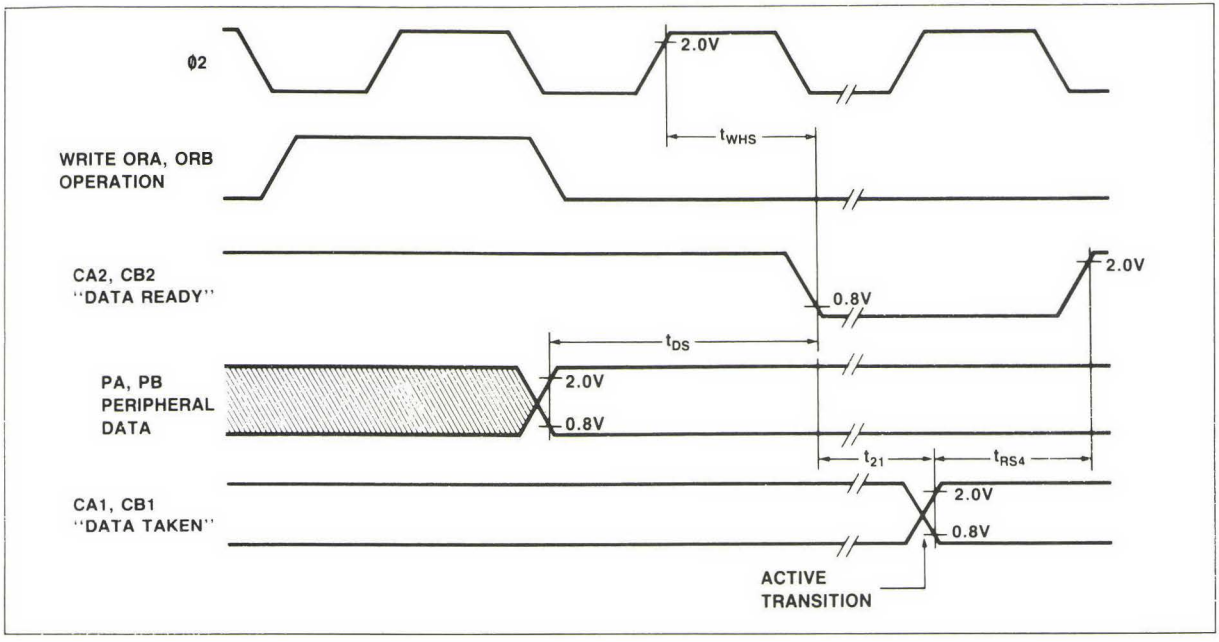


Figure 31d. CA2, CB2 Timing for Write Handshake, Handshake Mode

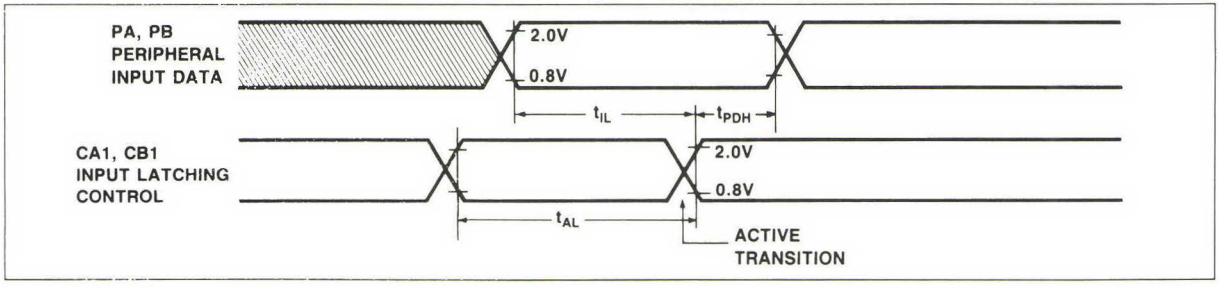


Figure 31e. Peripheral Data Input Latching Timing

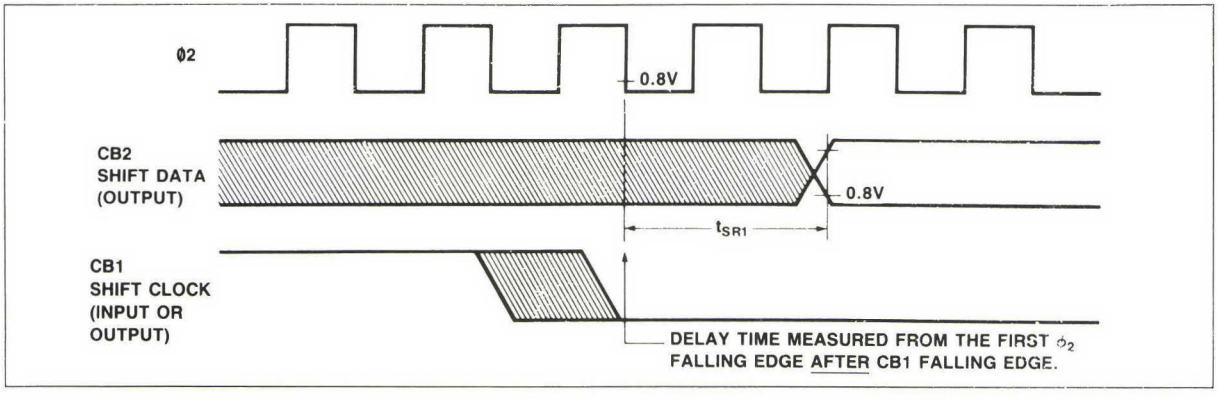


Figure 31f. Timing for Shift Out with Internal or External Shift Clocking

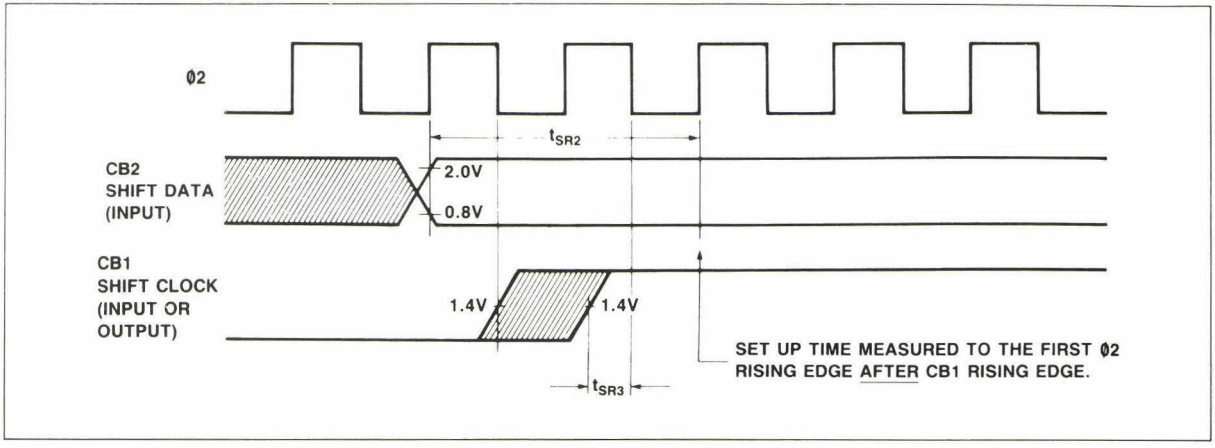


Figure 31g. Timing for Shift in with Internal or External Shift Clocking

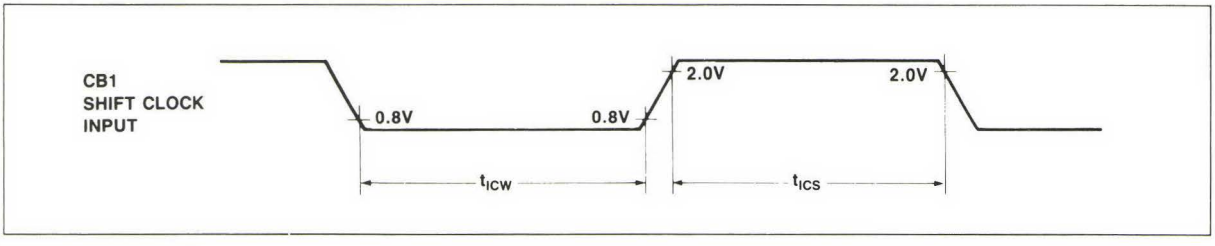


Figure 31h. External Shift Clock Timing

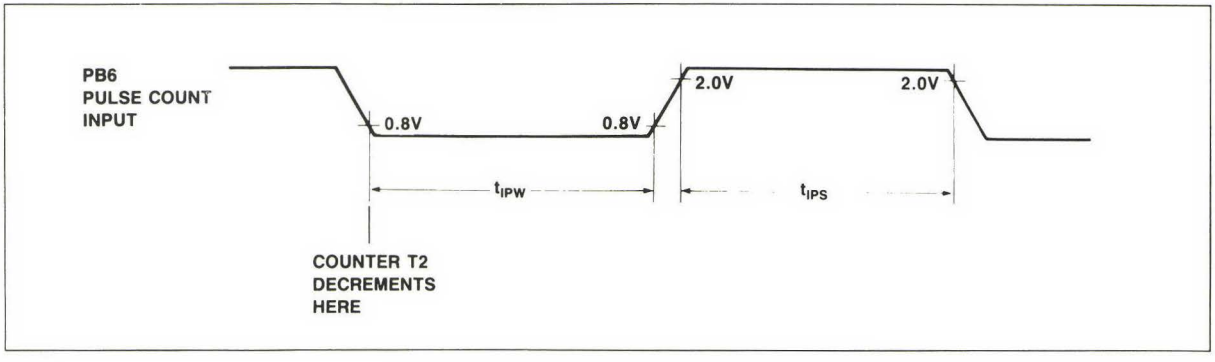
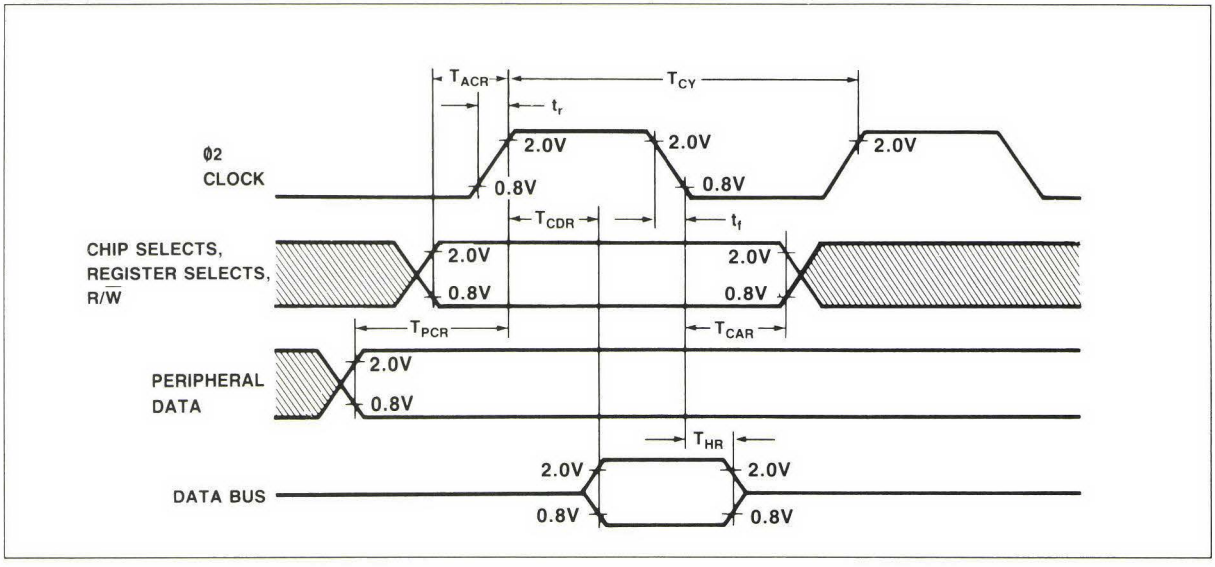


Figure 31i. Pulse Count Input Timing

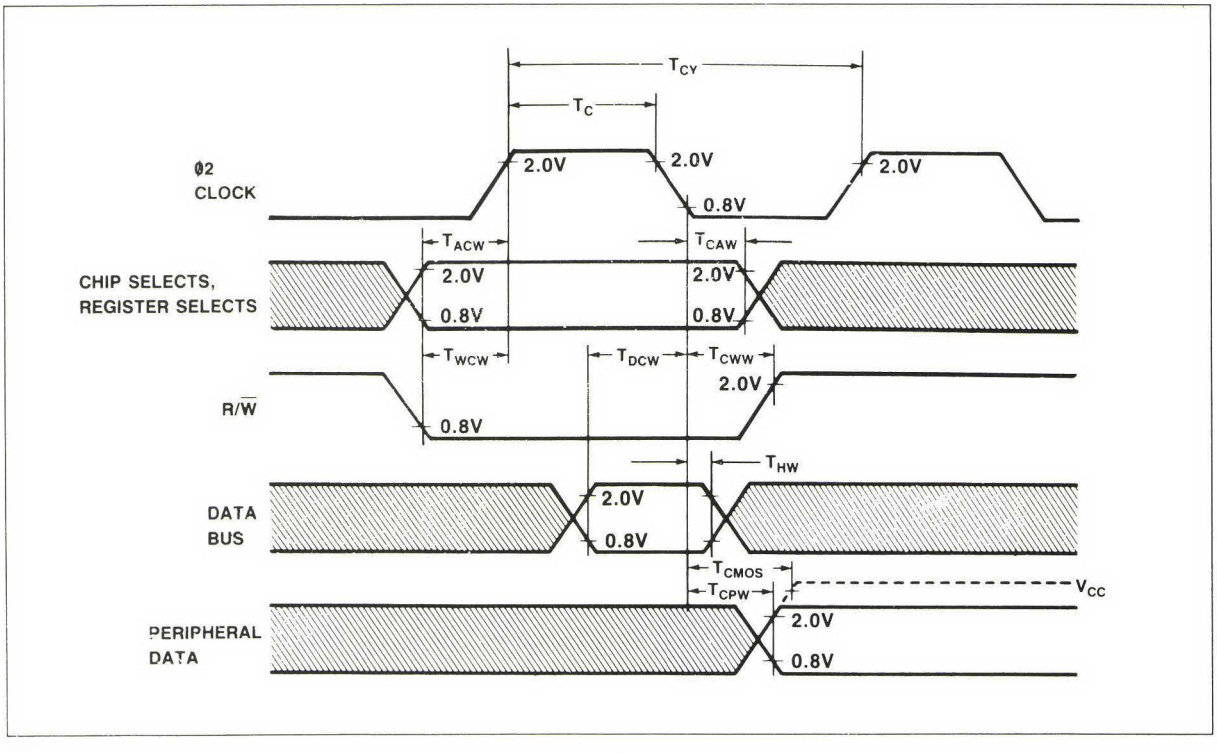
BUS TIMING CHARACTERISTICS

Parameter	Symbol	R6522 (1 MHz)		R6522A (2 MHz)		Unit
		Min.	Max.	Min.	Max.	
READ TIMING						
Cycle Time	T_{CY}	1	10	0.5	10	μs
Address Set-Up Time	T_{ACR}	180	—	90	—	ns
Address Hold Time	T_{CAR}	0	—	0	—	ns
Peripheral Data Set-Up Time	T_{PCR}	300	—	150	—	ns
Data Bus Delay Time	T_{CDR}	—	365	—	190	ns
Data Bus Hold Time	T_{HR}	10	—	10	—	ns
WRITE TIMING						
Cycle Time	T_{CY}	1	10	0.50	10	μs
Ø2 Pulse Width	T_C	470	—	235	—	ns
Address Set-Up Time	T_{ACW}	180	—	90	—	ns
Address Hold Time	T_{CAW}	0	—	0	—	ns
R/$\overline{W}$ Set-Up Time	T_{WCW}	180	—	90	—	ns
R/$\overline{W}$ Hold Time	T_{CWW}	0	—	0	—	ns
Data Bus Set-Up Time	T_{DCW}	200	—	90	—	ns
Data Bus Hold Time	T_{HW}	10	—	10	—	ns
Peripheral Data Delay Time	T_{CPW}	—	1.0	—	0.5	μs
Peripheral Data Delay Time to CMOS Levels	T_{CMOS}	—	2.0	—	1.0	μs
Note: t_R and t_F = 10 to 30 ns.						

BUS TIMING WAVEFORMS



Read Timing Waveforms



Write Timing Waveforms

ABSOLUTE MAXIMUM RATINGS*

Parameter	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	Vdc
Input Voltage	V_{IN}	−0.3 to +7.0	Vdc
Operating Temperature Commercial Industrial	T_A	 0 to +70 −40 to +85	 °C °C
Storage Temperature	T_{STG}	−55 to +150	°C

***NOTE:** Stresses above those listed under ABSOLUTE MAXIMUM RATINGS may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the other sections of this document is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

OPERATING CONDITIONS

Parameter	Symbol	Value
Supply Voltage	V_{CC}	5V ±5%
Temperature Range Commercial	T_A	 0°C to 70°C

DC CHARACTERISTICS

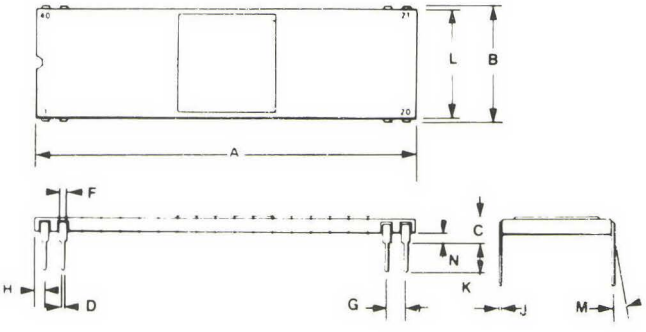
(V_{CC} = 5.0 Vdc ±5%, V_{SS} = 0, T_A = T_L to T_H, unless otherwise noted)

Parameter	Symbol	Min.	Typ.[3]	Max.	Unit	Test Conditions
Input High Voltage	V_{IH}	2.4	—	V_{CC}	V	
Input Low Voltage	V_{IL}	−0.3	—	0.4	V	
Input Leakage Current R/$\overline{W}$, $\overline{RES}$, RS0, RS1, RS2, RS3, CS1, $\overline{CS2}$, CA1, Ø2	I_{IN}	—	±1	±2.5	µA	V_{IN} = 0V to 5.25V V_{CC} = 0V
Input Leakage Current for Three-State Off D0-D07	I_{TSI}	—	±2	±10	µA	V_{IN} = 0.4V to 2.4V V_{CC} = 5.25V
Input High Current PA0-PA7, CA2, PB0-PB7, CB1, CBS	I_{IH}	−100	−200	—	µA	V_{IN} = 2.4V V_{CC} = 5.25V
Input Low Current PA0-PA7, CA2, PB0-PB7, CB1, CB2	I_{IL}	—	−0.9	−1.8	mA	V_{IL} = 0.4V V_{CC} = 5.25V
Output High Voltage All outputs PB0-PB7, CB2 (Darlington Drive)	V_{OH}	 2.4 1.5	 — —	 — —	 V V	V_{CC} = 4.75V I_{LOAD} = −100 µA I_{LOAD} = −1.0 mA
Output Low Voltage	V_{OL}	—	—	0.4	V	V_{CC} = 4.75V I_{LOAD} = 1.6 mA
Output High Current (Sourcing) Logic PB0-PB7, CB2 (Darlington Drive)	I_{OH}	 −100 −1.0	 −1000 −2.5	 — −10	 µA mA	 V_{OH} = 2.4V V_{OH} = 1.5V
Output Low Current (Sinking)	I_{OL}	1.6	—	—	mA	V_{OL} = 0.4V
Output Leakage Current (Off State) $\overline{IRQ}$	I_{OFF}	—	4	±10	µA	V_{OH} = 2.4V V_{CC} = 5.25V
Power Dissipation	P_D	—	450	700	mW	
Input Capacitance R/$\overline{W}$, $\overline{RES}$, RS0, RS1, RS2, RS3, CS1, $\overline{CS2}$, D0-D7, PA0-PA7, CA1, CA2, PB0-PB7 CB1, CB2 Ø2 Input	C_{IN}	 — — —	 — — —	 7 10 20	 pF pF pF	V_{CC} = 5.0V V_{IN} = 0V f = 1 MHz T_A = 25°C
Output Capacitance	C_{OUT}	—	—	10	pF	

Notes:
1. All units are direct current (DC) except for capacitance.
2. Negative sign indicates outward current flow, positive indicates inward flow.
3. Typical values shown for V_{CC} = 5.0V and T_A = 25°C.

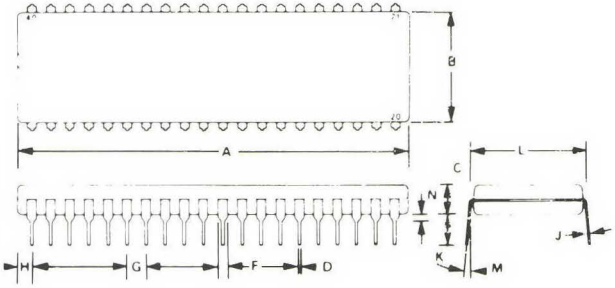
PACKAGE DIMENSIONS

40-PIN CERAMIC DIP



DIM	MILLIMETERS MIN	MILLIMETERS MAX	INCHES MIN	INCHES MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.60	15.37	0.575	0.605
M	0°	10°	0°	10°
N	0.51	1.52	0.020	0.060

40-PIN PLASTIC DIP



DIM	MILLIMETERS MIN	MILLIMETERS MAX	INCHES MIN	INCHES MAX
A	51.28	52.32	2.040	2.060
B	13.72	14.22	0.540	0.560
C	3.55	5.08	0.140	0.200
D	0.36	0.51	0.014	0.020
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.16	0.065	0.085
J	0.20	0.30	0.008	0.012
K	3.05	3.56	0.120	0.140
L	15.24 BSC		0.600 BSC	
M	7°	10°	7°	10°
N	0.51	1.02	0.020	0.040

SEMICONDUCTOR PRODUCTS DIVISION REGIONAL ROCKWELL SALES OFFICES

HOME OFFICE
Semiconductor Products Division
Rockwell International
4311 Jamboree Road
P.O Box C, MS 501-300
Newport Beach, California
92658-8902
(714) 833-4700
TWX: 910 591-1698

UNITED STATES
Semiconductor Products Division
Rockwell International
1842 Reynolds
Irvine, California 92714
(714) 833-4655
TWX: 910 595-2518

Semiconductor Products Division
Rockwell International
3375 Scott Blvd., Suite 410
Santa Clara, California 95054
(408) 980-1900
TLX: 756560

Semiconductor Products Division
Rockwell International
2001 N. Collins Blvd., Suite 103
Richardson, Texas 75080
(214) 996-6500
TLX: 73-307

Semiconductor Products Division
Rockwell International
10700 West Higgins Rd., Suite 102
Rosemont, Illinois 60018
(312) 297-8862
TWX: 910 233-0179 (RI MED ROSM)

Semiconductor Products Division
Rockwell International
5001B Greentree
Executive Campus, Rt. 73
Marlton, New Jersey 08053
(609) 596-0090
TWX: 710 940-1377

FAR EAST
Semiconductor Products Division
Rockwell International Overseas Corp.
Itohpia Hirakawa-cho Bldg.
7-6, 2-chome, Hirakawa-cho
Chiyoda-ku, Tokyo 102, Japan
(03) 265-8806
TLX: J22198

Rockwell Collins International
Tai Sang Commercial Bldg., 11th Floor
24-34 Hennessy Rd.
Hong Kong
(5) 274-321
TLX: 74071 HK

EUROPE
Semiconductor Products Division
Rockwell International GmbH
Fraunhoferstrasse 11B
D-8033 Munchen-Martinsried
West Germany
(089) 857-6016
TLX: 0521/2650 rimd d

Semiconductor Products Division
Rockwell International Limited
Heathrow House, Bath Rd.
Cranford, Hounslow,
Middlesex, TW5 9QW England
(01) 759-2366
TLX: 851-25463

Semiconductor Products
Rockwell Collins Italiana S.P.A.
Via Boccaccio, 23
20123 Milano, Italy
(02) 498.74.79
TLX: 316562 RCIMIL 1

YOUR LOCAL REPRESENTATIVE

7/84

Appendix F
Solution for heat flow in one dimension

The problem at hand is to solve the differential equation for heat flow in one dimension, *vis*

$$\partial T/\partial t = \alpha^2(\partial^2 T/\partial z^2) \tag{F.1}$$

where $\alpha = k/s$ and where the rod extends to infinity on both sides. The initial condition is that the temperature at $t = 0$ is given, ie, $T(z, t = 0) = f(z)$ where $f(z)$ is the given initial temperature distribution along the bar.

To proceed, we try the method of separation of variables by writing $T(z, t) = F(z)G(t)$. Equation (F.1) then becomes

$$\frac{\partial G/\partial t}{\alpha^2 G} = -\frac{\partial^2 F/\partial z^2}{F} \tag{F.2}$$

Since the variables t and z vary independently, each side of Equation (F.2) must be equal to a constant, say q, giving two ordinary differential equations.

$$\left.\begin{aligned} \mathrm{d}G/\mathrm{d}t &= q\alpha^2 G \\ \mathrm{d}^2F/\mathrm{d}z^2 &= qF \end{aligned}\right\} \tag{F.3}$$

The solution for the first is

$$G(t) = K \exp(q\alpha^2 t) \tag{F.4}$$

where K is a constant. If q is positive, this solution grows without limit and thus is not a physically realizable solution. So $q \leq 0$ and we can write it as $q = -p^2$ to force this condition. Equation (F.4) becomes

$$G(t) = K \exp(-p^2\alpha^2 t) \tag{F.5}$$

The second of equations (F.3) can now be recognized as a simple wave equation

$$(\mathrm{d}^2F/\mathrm{d}z^2) + p^2F = 0 \tag{F.6}$$

with the solution

$$F(z) = A \cos(pz) + B \sin(pz) \tag{F.7}$$

So, the solution to the differential equation has the form

$$\begin{aligned} T(z, t; p) &= FG \\ &= [A \cos(pz) + B \sin(pz)] \exp(-p^2\alpha^2 t) \end{aligned} \tag{F.8}$$

where the constant K has been absorbed into A and B. Equation (F.8) is true for any p and any linear combination of solutions with different p will also be a solution. In particular, a general solution is

$$T(z, t) = \int_0^\infty [A(p) \cos(pz) + B(p) \sin(pz)] \exp(-p^2\alpha^2 t)\mathrm{d}p \tag{F.9}$$

Using the initial condition that $T(z, 0) = f(z)$, gives for Equation (F.9)

$$T(z, 0) = \int_0^\infty [A(p) \cos(pz) + B(p) \sin(pz)]\mathrm{d}p \tag{F.10}$$

The Fourier integral theorem gives the following expressions for A and B

$$\left.\begin{aligned} A(p) &= (1/\pi) \int_{-\infty}^{\infty} f(\xi) \cos(p\xi)\mathrm{d}\xi \\ B(p) &= (1/\pi) \int_{-\infty}^{\infty} f(\xi) \sin(p\xi)\mathrm{d}\xi \end{aligned}\right\} \tag{F.11}$$

Using these expressions, Equation (F.9) becomes

$$\begin{aligned} T(z, t) &= (1/\pi) \int_0^\infty \left\{ \int_{-\infty}^{\infty} f(\xi)[\cos(p\xi) \cos(pz) + \sin(p\xi) \sin(p\xi)] \right. \\ &\qquad \left. \times \exp(-p^2\alpha^2 t)\mathrm{d}\xi \right\} \mathrm{d}p \\ &= (1/\pi) \int_0^\infty \left\{ \int_{-\infty}^{\infty} f(\xi) \cos(pz - p\xi) \exp(-p^2\alpha^2 t)\mathrm{d}\xi \right\} \mathrm{d}p \end{aligned} \tag{F.12}$$

Exchanging the order of integration gives

$$T(z, t) = (1/\pi) \int_{-\infty}^{\infty} f(\xi) \left\{ \int_0^\infty \cos(pz - \mathrm{p}\xi) \exp(-p^2\alpha^2 t)\mathrm{d}p \right\} d\xi$$

The inner integral can be found in a table of integrals and is equal to

$$\frac{\pi^{1/2}}{2\alpha t^{1/2}} \exp\left[-\frac{(z - \xi)^2}{4\alpha^2 t} \right]$$

Therefore

$$T(z, t) = \frac{1}{2\alpha(\pi t)^{1/2}} \int_{-\infty}^{\infty} f(\xi) \exp\left[-\frac{(z - \xi)^2}{4\alpha^2 t} \right] \mathrm{d}\xi \tag{F.13}$$

In the physical situation of a very quick impulse of heat given to a rod at $z = 0$, the initial temperature distribution will be (Figure F.1)

$$f(z) = \lim_{\Delta z \to 0} \begin{cases} 0 & z < -\Delta z \\ T_{\max} & -\Delta z < z < \Delta z \\ 0 & \Delta z < z \end{cases}$$

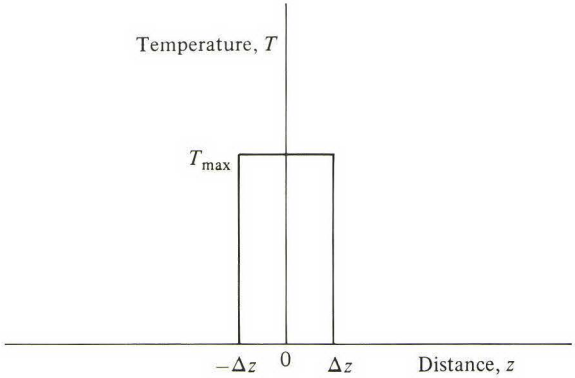


Fig. F.1. Initial temperature distribution on the infinite rod.

Equation (F.13) becomes

$$T(z, t) = \frac{1}{2\alpha(\pi t)^{1/2}} \int_{-\Delta z}^{\Delta z} f(\xi) \exp\left[-\frac{(z - \xi)^2}{4\alpha^2 t}\right] d\xi \tag{F.14}$$

If Δz is small, the exponential in the integral will not vary much across the interval $-\Delta z$ to Δz and so may be evaluated at $\xi = 0$ and be removed from the integral.

$$T(z, t) = \frac{1}{2\alpha(\pi t)^{1/2}} \exp\left(-\frac{z^2}{4\alpha^3 t}\right) \int_{-\Delta z}^{\Delta z} f(\xi) d\xi \tag{F.15}$$

The remaining integral is just a constant so

$$T(z, t) = \frac{B}{t^{1/2}} \exp\left(-\frac{z^2}{4\alpha^2 t}\right) \tag{F.16}$$

where B has absorbed all the constants. Also any constant value, say A, is a solution to the differential equation, so

$$T(z, t) = A + \frac{B}{t^{1/2}} \exp\left(-\frac{z^2}{4\alpha^2 t}\right) \tag{F.17}$$

as is stated as Equation (5.1.5).

That's all folks.

Appendix G
Finite impulse heat flow in a rod

The Equation (5.1.9) describes the flow of heat in a rod when the heat is applied very quickly at one point. The term very quickly means that the ratio of the time that the heater is on (call it τ) to the characteristic time of the system, t_1, is much less than one.

$$\tau/t_1 \ll 1 \tag{G.1}$$

Physically, this means that the heat was put into the rod much faster than it flowed away from the point where it was added.

In doing the experiment, equation (G.1) does not always strictly hold. An impulse of 0.5 s gives a τ/t_1 of about 0.4. In that case, the input of heat can be considered to be made up of a series of heat impulses, each of which has a width $\Delta\tau$ such that

$$\Delta\tau/t_1 \ll 1$$

See Figure G.1.

Thus for each of these smaller intervals $\Delta\tau$, Equation (5.1.9) will hold but must be rewritten with a change of origin:

$$T_i = T_1^i\left(\frac{t_1}{t + \tau}\right)^{1/2} \exp\left(\frac{-t_1}{t + \tau}\right) \tag{G.2}$$

where

$$T_I^i = 2q/Azs\pi^{1/2}$$

and $q = P\Delta\tau$ is the heat put in during one interval and P is the power (assumed to be constant). The total temperature change will be given by the sum of the individual T_i:

$$T = \sum_i T_i$$

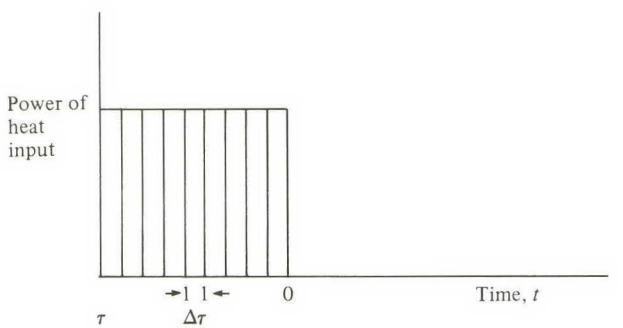


Fig. G.1. Heat input pulse with finite duration.

and the total heat input is

$$Q = \sum_i q$$

If $\Delta\tau$ goes to 0 then the sum goes to an integral:

$$T(t) = \int_0^T T_1' \left(\frac{t_1}{t+\tau}\right)^{1/2} \exp\left(\frac{-t_1}{t+\tau}\right) d\tau \tag{G.3}$$

with $T_1' = 2P/Azs\pi^{1/2}$.

By a suitable change in variable and integration by parts, this integral can be evaluated giving

$$T = 2T_1 \frac{1}{\gamma}\left[(x+\gamma)^{1/2} \exp\left(\frac{-1}{x+\gamma}\right) - x^{1/2} \exp\left(\frac{-1}{x}\right) + \pi^{1/2} \operatorname{erf}\left(\frac{1}{x+\gamma}\right)^{1/2} - \pi^{1/2} \operatorname{erf}\left(\frac{1}{x}\right)\right] \tag{G.4}$$

where

$$T_1 = \frac{2Q}{Azs\pi^{1/2}}$$

as in Equation (5.1.9)

$$\gamma = \tau/t_1$$
$$\chi = t/t_1$$

and

$$\operatorname{erf}(\eta) = \frac{2}{\pi^{1/2}} \int_0^\eta \exp(-\xi^2) d\xi$$

is the error function which can be evaluated using a table or a computer program.

Figure G.2 is a plot of T/T_1 vs. t/t_1 for $\gamma = 0.01$–1.5 and shows the error

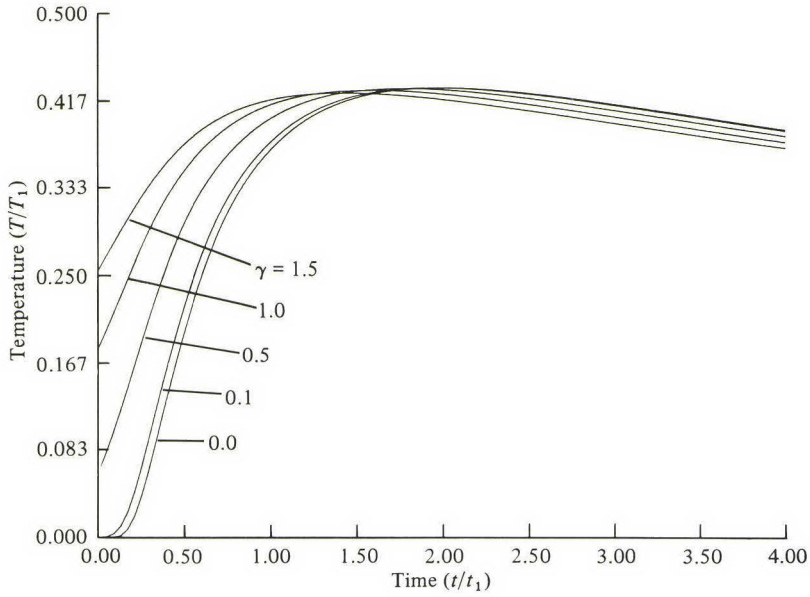


Fig. G.2. Heat flow for a finite heat pulse of length $\gamma = \tau/t_1$ with $t = 0$ at the end of the pulse.

which is made when modeling an experiment with the impulse solution (Equation (5.1.9) when Equation (G.4) is actually more correct. The curve with $\gamma = 0.01$ is essentially equal to the impulse solution Equation (5.1.9). For ratios of $\gamma > 0.1$ an appreciable error is made.

If $t = 0$ is measured from the center of the finite input pulse, a better fit is obtained. Equation (G.4) can be translated to this new origin by the substitution $t \to t - \tau/2$ giving

$$T = 2T_1 \frac{1}{\gamma}\left[(x + \gamma/2)^{1/2} \exp\left(\frac{-1}{x + \gamma/2}\right) - (x - \gamma/2)^{1/2} \exp\left(\frac{-1}{x - \gamma/2}\right) + \pi^{1/2} \operatorname{erf}\left(\frac{1}{x + \gamma/2}\right) - \pi^{1/2} \operatorname{erf}\left(\frac{1}{x - \gamma/2}\right)\right] \tag{G.5}$$

A plot of Equation (G.5) is given in Figure G.3. Ratios of up to $\gamma = 1$ can be tolerated without appreciable error with this time origin.

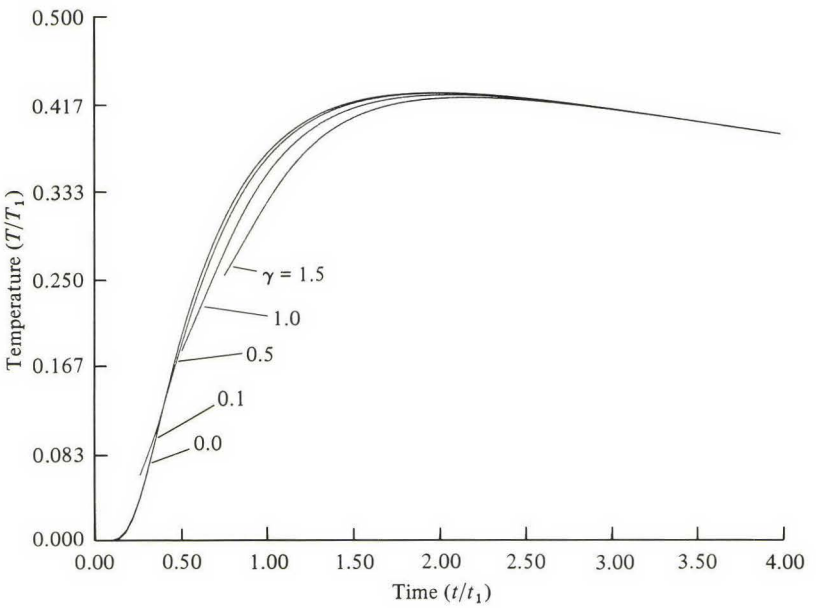


Fig. G.3. As in Figure G.2 but with $t = 0$ in the middle of the pulse.

Appendix H
Bootstrap sequence

A whole series of programs is run automatically when the APPLE computer is turned on. This is called the 'bootstrap' since the computer begins in a state where it is not usable and pulls itself up by its own bootstraps (programs) to a state where it can be programmed or operated via commands from the keyboard.

After the power is turned ON, the RESET sequence begins. The CPU looks in $FFFC, $FFFD (the RESET vector) for an address and begins executing the program at that address. In the APPLE the address in the RESET vector is $FF62 which is in the monitor ROM. Among other housekeeping chores, the monitor program looks for an installed disk drive controller card in slot 6. (If it does not find it, the monitor jumps to APPLESOFT BASIC in ROM.) If it finds it, the drive is turned on (red light on) and the disk is searched for the DOS file. This program file is loaded into RAM (see the memory map of Appendix C) and control is transferred to the DOS program. The DOS program (1) links itself to APPLESOFT BASIC so that disk commands can be used, (2) checks the size of RAM and sets HIMEM to an initial value, and (3) looks for an APPLESOFT program file on the disk called 'HELLO'. If it finds this file, it is loaded and run. On the SYSTEM START disk used in the laboratory, there is a HELLO program which does the following: (1) loads INTEGER BASIC/MINIASSEMBLER into the RAM of the language card. (2) Loads and runs AMPERGRAPH LOADER which links AMPERGRAPH to APPLESOFT BASIC (see memory map, Appendix C). (3) Returns to APPLESOFT BASIC. At this time the APPLE is waiting with the cursor blinking for you to type a command or program line.

The following HELLO Program is used on the SYSTEM START disk so that AMPERGRAPH is automatically linked to BASIC when the computer is turned on. It requires the following files to be on the disk as well:

INT BASIC	From DOS3.3
LOADER .OBJ0	From APPLE DOS3.3
RENUMBER	From APPLE DOS3.3
CHAIN	From APPLE DOS3.3
AMPERGRAPH LOADER	From AMPERGRAPH disk
AMPERGRAPH	From AMPERGRAPH disk

```
10  TEXT : HOME
20 D$ =  CHR$ (4): REM  CTRL-D
30  VTAB 2:A$ = "APPLE II": GOSUB 1000
40  VTAB 4:A$ = "DOS VERSION 3.3 SYSTEM
      MASTER": GOSUB 1000
50  VTAB 7:A$ = "JANUARY 1, 1983" : GOSUB 1000
60  PRINT D$;"BLOAD LOADER.OBJ0"
70  CALL 4096: REM FAST LOAD IN INTEGER BASIC
80  VTAB 10: CALL  - 958:A$ = "COPYRIGHT
     APPLE COMPUTER, INC. 1980,1982":
     GOSUB 1000
90 C = ( - 1101): IF C = 6 THEN PRINT :
     INVERSE :A$ = "BE SURE CAPS LOCK IS
     DOWN": GOSUB 1000: NORMAL
95  PRINT  CHR" (4);"RUN AMPERGRAPH LOADER"
100  PRINT CHR$ (4);"FP"
1000  REM CENTER STRING A$
1010 B =  INT (20 - ( LEN (A$) /2)):
     IF B =  < 0 THEN B = 1
1020  HTAB B: PRINT A$: RETURN
```

Appendix I
Machine language instructions

This appendix contains information about several aspects of machine language programming. Figure I.1 shows a bird's-eye-view of the internal architecture of the 6502 microprocessor chip. The next few pages describe the details of what the 6502 does at each clock cycle for various address modes and instructions and is taken from the MOS Technology Microcomputer Programming Manual (used with permission). Then follows a summary of the 6502 instruction set. For more information about individual instructions, refer to Leventhal's *6502 Assembly Language Programming* or the MOS Technology *6502 Programming Manual*.

Some MINIASSEMBLER tips:

Remember to use the # sign to designate immediate mode addressing. Without it the instruction is translated as an absolute address mode calling an address on the first page of memory (in the first 256 bytes).

You can BLOAD a machine language program from APPLESOFT BASIC as well as from the MINIASSEMBLER. It is sometimes convenient to include it as a program statement eg, PRINT CHR$(4); "BLOAD . . ."

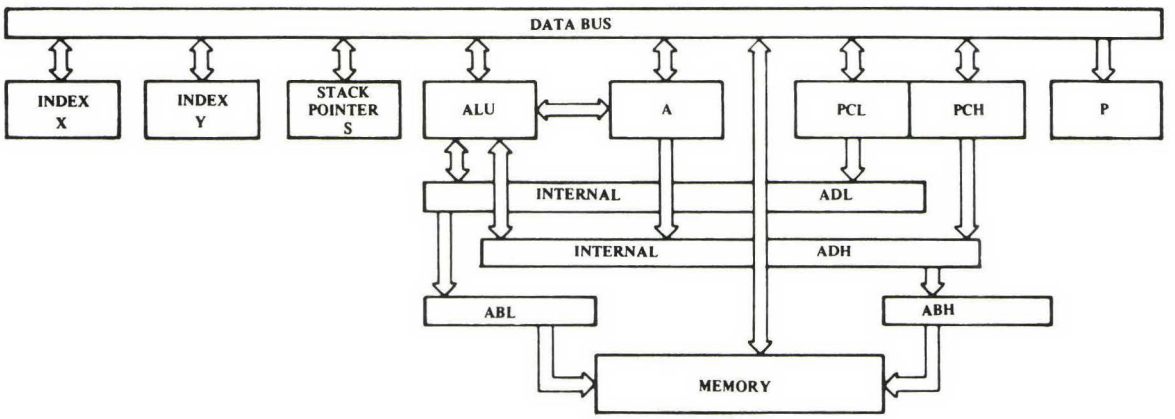
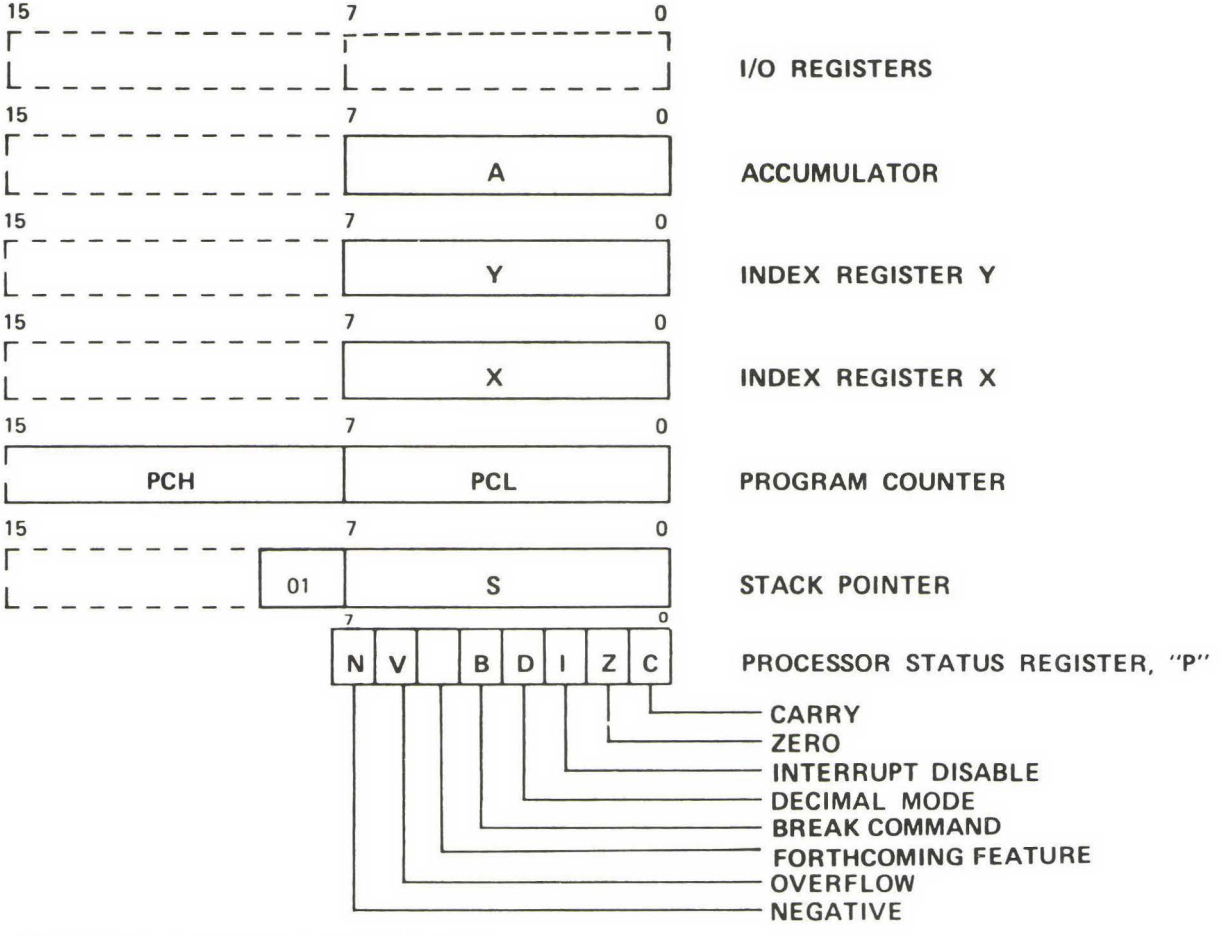


Fig. I.1. Block diagram of the 6502 microprocessor (from *MCS 6500 Microcomputer Programming Manual*, MOS Technology, Norristown, PA, 1976).

MCS6501-MCS6505 MICROPROCESSOR INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

PROGRAMMING MODEL MCS650X



* Solid line indicates currently available features
Dashed line indicates forthcoming members of family

The following notation applies to this summary:

A	Accumulator
X, Y	Index Registers
M	Memory
P	Processor Status Register
S	Stack Pointer
√	Change
_	No Change
+	Add
∧	Logical AND
-	Subtract
∀	Logical Exclusive Or
↑	Transfer from Stack
↓	Transfer to Stack
→	Transfer to
←	Transfer to
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	OPERAND
#	IMMEDIATE ADDRESSING MODE

Note: At the top of each table is located in parentheses a reference number (Ref: XX) which directs the user to that Section in the MCS6500 Microcomputer Family Programming Manual in which the instruction is defined and discussed.

ADC

Add memory to accumulator with carry

ADC

Operation: A + M + C → A, C

(Ref: 2.2.1)

N	Z	C	I	D	V
✓	✓	✓	–	–	✓

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ADC # Oper	69	2	2
Zero Page	ADC Oper	65	2	3
Zero Page, X	ADC Oper, X	75	2	4
Absolute	ADC Oper	6D	3	4
Absolute, X	ADC Oper, X	7D	3	4*
Absolute, Y	ADC Oper, Y	79	3	4*
(Indirect, X)	ADC (Oper, X)	61	2	6
(Indirect), Y	ADC (Oper), Y	71	2	5*

* Add 1 if page boundary is crossed.

AND

"AND" memory with accumulator

AND

Logical AND to the accumulator

Operation: A ∧ M → A

(Ref: 2.2.3.0)

N	Z	C	I	D	V
✓	✓	–	–	–	–

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	AND # Oper	29	2	2
Zero Page	AND Oper	25	2	3
Zero Page, X	AND Oper, X	35	2	4
Absolute	AND Oper	2D	3	4
Absolute, X	AND Oper, X	3D	3	4*
Absolute, Y	AND Oper, Y	39	3	4*
(Indirect, X)	AND (Oper, X)	21	2	6
(Indirect), Y	AND (Oper), Y	31	2	5

* Add 1 if page boundary is crossed.

ASL

ASL *Shift Left One Bit (Memory or Accumulator)* **ASL**

Operation: C ← [7|6|5|4|3|2|1|Ø] ←Ø

N Z C I D V
✓ ✓ ✓ – – –

(Ref: 10.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ASL A	ØA	1	2
Zero Page	ASL Oper	Ø6	2	5
Zero Page, X	ASL Oper, X	16	2	6
Absolute	ASL Oper	ØE	3	6
Absolute, X	ASL Oper, X	1E	3	7

BCC

BCC *Branch on Carry Clear* **BCC**

Operation: Branch on C = Ø

N Z C I D V
– – – – – –

(Ref: 4.1.1.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCC Oper	9Ø	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BCS

BCS *Branch on carry set* **BCS**

Operation: Branch on C = 1

N Z C I D V
– – – – – –

(Ref: 4.1.1.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BCS Oper	BØ	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to next page.

BEQ

BEQ *Branch on result zero*

BEQ

Operation: Branch on Z = 1

(Ref: 4.1.1.5)

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BEQ Oper	FØ	2	2*

* Add 1 if branch occurs to same page.

* Add 2 if branch occurs to next page.

BIT

BIT *Test bits in memory with accumulator*

BIT

Operation: A ∧ M, M_7 → N, M_6 → V

Bit 6 and 7 are transferred to the status register. If the result of A∧M is zero then Z = 1, otherwise Z = Ø

N	Z	C	I	D	V
M_7	✓	–	–	–	M_6

(Ref: 4.2.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	BIT Oper	24	2	3
Absolute	BIT Oper	2C	3	4

BMI

BMI *Branch on result minus*

BMI

Operation: Branch on N = 1

(Ref: 4.1.1.1)

N	Z	C	I	D	V
–	–	–	–	–	–

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BMI Oper	3Ø	2	2*

Add 1 if branch occurs to same page.

* Add 2 if branch occurs to different page.

BNE

BNE *Branch on result not zero*

BNE

Operation: Branch on Z = 0

N Z C I D V
– – – – – –

(Ref: 4.1.1.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BNE Oper	DØ	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BPL

BPL *Branch on result plus*

BPL

Operation: Branch on N = Ø

N Z C I D V
– – – – – –

(Ref: 4.1.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BPL Oper	1Ø	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BRK

BRK *Force Break*

BRK

Operation: Forced Interrupt PC + 2 ↓ P ↓

N Z C I D V
– – – 1 – –

(Ref: 9.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	BRK	ØØ	1	7

1. A BRK command cannot be masked by setting I.

BVC

BVC *Branch on overflow clear*

BVC

Operation: Branch on V = 0

N Z C I D V
– – – – – –

(Ref: 4.1.1.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVC Oper	5Ø	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

BVS

BVS *Branch on overflow set*

BVS

Operation: Branch on V = 1

N Z C I D V
– – – – – –

(Ref: 4.1.1.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Relative	BVS Oper	7Ø	2	2*

* Add 1 if branch occurs to same page.
* Add 2 if branch occurs to different page.

CLC

CLC *Clear carry flag*

CLC

Operation: Ø → C

N Z C I D V
– – Ø – – –

(Ref: 3.0.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLC	18	1	2

CLD

CLD *Clear decimal mode*

Operation: Ø → D

N Z C I D V
– – – – Ø –

(Ref: 3.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLD	D8	1	2

CLD

CLI

CLI *Clear interrupt disable bit*

Operation: Ø → I

N Z C I D V
– – – Ø – –

(Ref: 3.2.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLI	58	1	2

CLI

CLV

CLV *Clear overflow flag*

Operation: Ø → V

N Z C I D V
– – – – – Ø

(Ref: 3.6.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	CLV	B8	1	2

CLV

CMP

CMP *Compare memory and accumulator*

Operation: A - M

N Z C I D V
✓ ✓ ✓ – – –

(Ref: 4.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CMP #Oper	C9	2	2
Zero Page	CMP Oper	C5	2	3
Zero Page, X	CMP Oper, X	D5	2	4
Absolute	CMP Oper	CD	3	4
Absolute, X	CMP Oper, X	DD	3	4*
Absolute, Y	CMP Oper, Y	D9	3	4*
(Indirect, X)	CMP (Oper, X)	C1	2	6
(Indirect), Y	CMP (Oper), Y	D1	2	5*

* Add 1 if page boundary is crossed.

CPX

CPX *Compare Memory and Index X*

Operation: X - M

N Z C I D V
✓ ✓ ✓ – – –

(Ref: 7.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPX #Oper	EØ	2	2
Zero Page	CPX Oper	E4	2	3
Absolute	CPX Oper	EC	3	4

CPY

CPY *Compare memory and index Y*

Operation: Y - M

N Z C I D V
✓ ✓ ✓ – – –

(Ref: 7.9)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	CPY #Oper	CØ	2	2
Zero Page	CPY Oper	C4	2	3
Absolute	CPY Oper	CC	3	4

DEC

DEC *Decrement memory by one*

DEC

Operation: M - 1 → M

N Z C I D V
✓ ✓ – – – –

(Ref: 10.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	DEC Oper	C6	2	5
Zero Page, X	DEC Oper, X	D6	2	6
Absolute	DEC Oper	CE	3	6
Absolute, X	DEC Oper, X	DE	3	7

DEX

DEX *Decrement index X by one*

DEX

Operation: X - 1 → X

N Z C I D V
✓ ✓ – – – –

(Ref: 7.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEX	CA	1	2

DEY

DEY *Decrement index Y by one*

DEY

Operation: Y - 1 → Y

N Z C I D V
✓ ✓ – – – –

(Ref: 7.7)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	DEY	88	1	2

EOR — EOR *"Exclusive–Or" memory with accumulator* — EOR

Operation: A ⊻ M → A

N Z C I D V
✓ ✓ – – – –

(Ref: 2.2.3.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	EOR #Oper	49	2	2
Zero Page	EOR Oper	45	2	3
Zero Page, X	EOR Oper, X	55	2	4
Absolute	EOR Oper	4D	3	4
Absolute, X	EOR Oper, X	5D	3	4*
Absolute, Y	EOR Oper, Y	59	3	4*
(Indirect, X)	EOR (Oper, X)	41	2	6
(Indirect),Y	EOR (Oper), Y	51	2	5*

* Add 1 if page boundary is crossed.

INC — INC *Increment memory by one* — INC

Operation: M + 1 → M

N Z C I D V
✓ ✓ – – – –

(Ref: 10.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	INC Oper	E6	2	5
Zero Page, X	INC Oper, X	F6	2	6
Absolute	INC Oper	EE	3	6
Absolute, X	INC Oper, X	FE	3	7

INX — INX *Increment Index X by one* — INX

Operation: X + 1 → X

N Z C I D V
✓ ✓ – – – –

(Ref: 7.4)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INX	E8	1	2

INY

INY *Increment Index Y by one*

INY

Operation: Y + 1 → Y

N Z C I D V
↕ ↕ – – – –

(Ref: 7.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	INY	C8	1	2

JMP

JMP *Jump to new location*

JMP

Operation: (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

(Ref: 4.0.2)
(Ref: 9.8.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JMP Oper	4C	3	3
Indirect	JMP (Oper)	6C	3	5

JSR

JSR *Jump to new location saving return address*

JSR

Operation: PC + 2 ↓, (PC + 1) → PCL
(PC + 2) → PCH

N Z C I D V
– – – – – –

(Ref: 8.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Absolute	JSR Oper	20	3	6

LDA

LDA *Load accumulator with memory*

LDA

Operation: M → A

N Z C I D V
✓ ✓ – – – –

(Ref: 2.1.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDA # Oper	A9	2	2
Zero Page	LDA Oper	A5	2	3
Zero Page, X	LDA Oper, X	B5	2	4
Absolute	LDA Oper	AD	3	4
Absolute, X	LDA Oper, X	BD	3	4*
Absolute, Y	LDA Oper, Y	B9	3	4*
(Indirect, X)	LDA (Oper, X)	A1	2	6
(Indirect), Y	LDA (Oper), Y	B1	2	5*

* Add 1 if page boundary is crossed.

LDX

LDX *Load index X with memory*

LDX

Operation: M → X

N Z C I D V
✓ ✓ – – – –

(Ref: 7.0)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDX # Oper	A2	2	2
Zero Page	LDX Oper	A6	2	3
Zero Page, Y	LDX Oper, Y	B6	2	4
Absolute	LDX Oper	AE	3	4
Absolute, Y	LDX Oper, Y	BE	3	4*

* Add 1 when page boundary is crossed.

LDY

LDY *Load index Y with memory*

LDY

Operation: M → Y

N Z C I D V
✓ ✓ – – – –

(Ref: 7.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	LDY #Oper	AØ	2	2
Zero Page	LDY Oper	A4	2	3
Zero Page, X	LDY Oper, X	B4	2	4
Absolute	LDY Oper	AC	3	4
Absolute, X	LDY Oper, X	BC	3	4*

* Add 1 when page boundary is crossed.

LSR

LSR *Shift right one bit (memory or accumulator)*

LSR

Operation: Ø → 7 6 5 4 3 2 1 0 → C

N Z C I D V
Ø ✓ ✓ – – –

(Ref: 10.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	LSR A	4A	1	2
Zero Page	LSR Oper	46	2	5
Zero Page, X	LSR Oper, X	56	2	6
Absolute	LSR Oper	4E	3	6
Absolute, X	LSR Oper, X	5E	3	7

NOP

NOP *No operation*

NOP

Operation: No Operation (2 cycles)

N Z C I D V
– – – – – –

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	NOP	EA	1	2

ORA

ORA *"OR" memory with accumulator*

ORA

Operation: A V M → A

N Z C I D V
✓ ✓ – – – –

(Ref: 2.2.3.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	ORA #Oper	Ø9	2	2
Zero Page	ORA Oper	Ø5	2	3
Zero Page, X	ORA Oper, X	15	2	4
Absolute	ORA Oper	ØD	3	4
Absolute, X	ORA Oper, X	1D	3	4*
Absolute, Y	ORA Oper, Y	19	3	4*
(Indirect, X)	ORA (Oper, X)	Ø1	2	6
(Indirect), Y	ORA (Oper), Y	11	2	5

* Add 1 on page crossing

PHA

PHA *Push accumulator on stack*

PHA

Operation: A ↓

N Z C I D V
– – – – – –

(Ref: 8.5)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHA	48	1	3

PHP

PHP *Push processor status on stack*

PHP

Operation: P↓

N Z C I D V
– – – – – –

(Ref: 8.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PHP	Ø8	1	3

PLA

PLA *Pull accumulator from stack*

PLA

Operation: A ↑

N Z C I D V
✓ ✓ – – – –

(Ref: 8.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLA	68	1	4

PLP

PLP *Pull processor status from stack*

PLP

Operation: P ↑

N Z C I D V
From Stack

(Ref: 8.12)

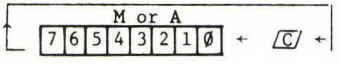
Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	PLP	28	1	4

ROL

ROL *Rotate one bit left (memory or accumulator)*

ROL

Operation:



N Z C I D V
✓ ✓ ✓ – – –

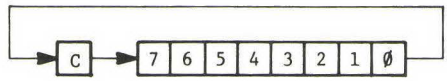
(Ref: 10.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROL A	2A	1	2
Zero Page	ROL Oper	26	2	5
Zero Page, X	ROL Oper, X	36	2	6
Absolute	ROL Oper	2E	3	6
Absolute, X	ROL Oper, X	3E	3	7

ROR

ROR *Rotate one bit right (memory or accumulator)* **ROR**

Operation:



(Ref: 10.4)

N Z C I D V
✓ ✓ ✓ - - -

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Accumulator	ROR A	6A	1	2
Zero Page	ROR Oper	66	2	5
Zero Page,X	ROR Oper,X	76	2	6
Absolute	ROR Oper	6E	3	6
Absolute,X	ROR Oper,X	7E	3	7

Note: ROR instruction will be available on MCS650X microprocessors after June, 1976.

RTI

RTI *Return from interrupt* **RTI**

Operation: P↑ PC↑

N Z C I D V
From Stack

(Ref: 9.6)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTI	4Ø	1	6

RTS

RTS *Return from subroutine* **RTS**

Operation: PC↑, PC + 1 → PC

N Z C I D V
– – – – – –

(Ref: 8.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	RTS	6Ø	1	6

SBC

SBC *Subtract memory from accumulator with borrow*

Operation: A − M − $\overline{C}$ → A

N Z C I D V
✓ ✓ ✓ — — ✓

Note: $\overline{C}$ = Borrow (Ref: 2.2.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Immediate	SBC #Oper	E9	2	2
Zero Page	SBC Oper	E5	2	3
Zero Page, X	SBC Oper, X	F5	2	4
Absolute	SBC Oper	ED	3	4
Absolute, X	SBC Oper, X	FD	3	4*
Absolute, Y	SBC Oper, Y	F9	3	4*
(Indirect, X)	SBC (Oper, X)	E1	2	6
(Indirect), Y	SBC (Oper), Y	F1	2	5*

* Add 1 when page boundary is crossed.

SEC

SEC *Set carry flag*

Operation: 1 → C

N Z C I D V
— — 1 — — —

(Ref: 3.0.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEC	38	1	2

SED

SED *Set decimal mode*

Operation: 1 → D

N Z C I D V
— — — — 1 —

(Ref: 3.3.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SED	F8	1	2

SEI

SEI *Set interrupt disable status*

SEI

Operation: 1 → I

N	Z	C	I	D	V
–	–	–	1	–	–

(Ref: 3.2.1)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	SEI	78	1	2

STA

STA *Store accumulator in memory*

STA

Operation: A → M

N	Z	C	I	D	V
–	–	–	–	–	–

(Ref: 2.1.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STA Oper	85	2	3
Zero Page, X	STA Oper, X	95	2	4
Absolute	STA Oper	8D	3	4
Absolute, X	STA Oper, X	9D	3	5
Absolute, Y	STA Oper, Y	99	3	5
(Indirect, X)	STA (Oper, X)	81	2	6
(Indirect), Y	STA (Oper), Y	91	2	6

STX

STX *Store index X in memory*

STX

Operation: X → M

N	Z	C	I	D	V
–	–	–	–	–	–

(Ref: 7.2)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STX Oper	86	2	3
Zero Page, Y	STX Oper, Y	96	2	4
Absolute	STX Oper	8E	3	4

STY

STY Store index Y in memory

STY

Operation: Y → M

N Z C I D V
– – – – – –

(Ref: 7.3)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Zero Page	STY Oper	84	2	3
Zero Page, X	STY Oper, X	94	2	4
Absolute	STY Oper	8C	3	4

TAX

TAX *Transfer accumulator to index X*

TAX

Operation: A → X

N Z C I D V
✓ ✓ – – – –

(Ref: 7.11)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAX	AA	1	2

TAY

TAY *Transfer accumulator to index Y*

TAY

Operation: A → Y

N Z C I D V
✓ ✓ – – – –

(Ref: 7.13)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TAY	A8	1	2

TYA

TYA *Transfer index Y to accumulator*

TYA

Operation: Y → A

N Z C I D V
✓ ✓ – – – –

(Ref: 7.14)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TYA	98	1	2

TSX

TSX *Transfer stack pointer to index X*

TSX

Operation: S → X

N Z C I D V
✓ ✓ – – – –

(Ref: 8.9)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TSX	BA	1	2

TXA

TXA *Transfer index X to accumulator*

TXA

Operation: X → A

N Z C I D V
✓ ✓ – – – –

(Ref: 7.12)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXA	8A	1	2

TXS

TXS *Transfer index X to stack pointer*

TXS

Operation: X → S

N Z C I D V
– – – – – –

(Ref: 8.8)

Addressing Mode	Assembly Language Form	OP CODE	No. Bytes	No. Cycles
Implied	TXS	9A	1	2

5.5 ABSOLUTE ADDRESSING

Absolute addressing is a 3-byte instruction.

The first byte contains the OP CODE for specifying the operation and address mode. The second byte contains the low order byte of the effective address (that address which contains the data), while the third byte contains the high order byte of the effective address. Thus the programmer specifies the full 16-bit address and, since any memory location can be specified, this is considered the most normal mode for addressing. Other modes may be considered special subsets of this 16-bit addressing mode.

Example 5.5: Illustration of absolute addressing

Clock Cycle	Address Bus	Program Counter	Data Bus	Comments
1	PC	PC + 1	OP CODE	Fetch OP CODE
2	PC + 1	PC + 2	ADL	Fetch ADL, Decode OP CODE
3	PC + 2	PC + 3	ADH	Fetch ADH, Hold ADL
4	ADH, ADL	PC + 3	Data	Fetch Data
5	PC + 3	PC + 4	New OP CODE	Fetch New OP CODE, Execute Old OP CODE

The basic operation of the microprocessor in an Absolute address mode is to read the OP CODE in the first cycle while finishing the previous operation. In the second cycle, the microprocessor automatically reads the first byte after the OP CODE (in this case the address low) while interpreting the operation code. At the end of this cycle, the microprocessor knows that it needs a second byte for program sequence; therefore, 1 more byte will be accessed using the program counter while temporarily storing the address low. This occurs during the third cycle. In the fourth cycle, the operation is one of taking the address low and address high that were read during cycles 2 and 3 to address the operand. For example, in load A, the effective address is used to fetch from memory the data which is going to be loaded in the accumulator. In the case of storing, data is transferred from the accumulator to the addressed memory.

As was illustrated in the review of pipelining, depending on the instruction, it is possible for the microprocessor to start the next instruction fetch cycle after the effective address operation and independent of how many more internal cycles it may take to complete the OP CODE. The only exception to this is the case of "Jump Absolute" in which the address low and address high that are fetched in cycle 2 and cycle 3 are used as the 16-bit address for the next OP CODE. The jump absolute therefore only requires 3 cycles. In all other cases, absolute addressing takes 4 cycles, 3 to fetch the full instruction including the effective address, the fourth to perform the memory transfer called for in the instruction.

5.4 IMMEDIATE ADDRESSING

Immediate addressing is a 2-byte instruction.

The first byte contains the OP CODE specifying the operation and address mode. The second byte contains a constant value known to the programmer. It is often necessary to compare load and/or test against certain known values. Rather than requiring the user to define and load constants into some auxiliary RAM, the microprocessor allows the user to specify values which are known to him by the immediate addressing mode.

Example 5.4: Illustration of immediate addressing

Clock Cycle	Address Bus	Program Counter	Data Bus	Comments
1	PC	PC + 1	OP CODE	Fetch OP CODE
2	PC + 1	PC + 2	Data	Fetch Data, Decode OP CODE
3	PC + 2	PC + 3	New OP CODE	Fetch New OP CODE, Execute Old OP CODE

6.1 ABSOLUTE INDEXED

Absolute indexed address is absolute addressing with an index register added to the absolute address. The sequences that occur for indexed absolute addressing without page crossing are as follows:

Example 6.6: Absolute Indexed; With No Page Crossing

Cycle	Address Bus	Data Bus	External Operation	Internal Operation
1	0100	OP CODE	Fetch OP CODE	Increment PC to 101, Finish Previous Instruction
2	0101	BAL	Fetch BAL	Increment PC to 102, Interpret Instruction
3	0102	BAH	Fetch BAH	Increment PC to 103, Calculate BAL + X
4	BAH,BAL+X	OPERAND	Put Out Effective Address	
5	103	Next OP CODE	Fetch Next OP CODE	Finish Operations

BAL and BAH refer to the low and high order bytes of the base address, respectively. While the index X was used in Example 6.7, the index Y is equally applicable.

Cycle	Address Bus	Data Bus	External Operations	Internal Operations
1	0100	OP CODE	Fetch Instruction	Finish Previous Operation; Increment PC to 0101
2	0101	New ADL	Fetch New ADL	Decode JSR; Increment PC to 0102
3	01FF			Store ADL
4	01FF	PCH	Store PCH	Hold ADL, Decrement S to 01FE
5	01FE	PCL	Store PCL	Hold ADL, Decrement S to 01FD
6	0102	ADH	Fetch ADH	Store Stack Pointer
7	ADH, ADL	New OP CODE	Fetch New OP CODE	ADL → PCL ADH → PCH

* S denotes "Stack Pointer."

In this example, it can be seen that during the first cycle the microprocessor fetches the JSR instruction. During the second cycle, address low for new program counter low is fetched. At the end of cycle 2, the microprocessor has decoded the JSR instruction and holds the address low in the microprocessor until the stack operations are complete.

NOTE: The stack is always stored in Page 1 (Hex address 0100-01FF).

The operation of the stack in the MCS650X microprocessor is such that the stack pointer is always left pointing at the next memory location into which data can be stored.

Return from Subroutine (Example)

Cycle	Address Bus	Data Bus	External Operations	Internal Operations
1	0300	OP CODE	Fetch OP CODE	Finish Previous Operation, 0301 → PC
2	0301	Discarded Data	Fetch Discarded Data	Decode RTS
3	01FD	Discarded Data	Fetch Discarded Data	Increment Stack Pointer to 01FE
4	01FE	02	Fetch PCL	Increment Stack Pointer to 01FF
5	01FF	01	Fetch PCH	
6	0102	Discarded Data	Put Out PC	Increment PC by 1 to 0103
7	0103	Next OP CODE	Fetch Next OP CODE	

As we can see, the RTS instruction effectively unwinds what was done to the stack in the JSR instruction.

The action and events are as follows: The microprocessor user pushes the panic button; the panic switch sensor causes an external device to indicate to the microprocessor an interrupt is desired; the microprocessor checks the status of the internal interrupt inhibit signal; if the internal inhibit is set, then the interrupt is ignored. However, if it is reset or when it becomes reset through some program reaction, the following set of operations occur:

Example 9.2: Interrupt Sequence

Cycles	Address Bus	Data Bus	External Operation	Internal Operation
1	PC	OP CODE	Fetch OP CODE	Hold Program Counter, Finish Previous Operation
2	PC	OP CODE	Fetch OP CODE	Force a BRK Instruction, Hold P-Counter
3	01FF	PCH	Store PCH on Stack	Decrement Stack Pointer to 01FE
4	01FE	PCL	Store PCL on Stack	Decrement Stack Pointer to 01FD
5	01FD	P	Store P on Stack	Decrement Stack Pointer to 01FC
6	FFFE	New PCL	Fetch Vector Low	Put Away Stack
7	FFFF	New PCH	Fetch Vector High	Vector Low → PCL and Set I
8	Vector PCH PCL	OP CODE	Fetch Interrupt Program	Increment PC to PC + 1

As can be seen in Example 9.2, the microprocessor uses the stack to save the reentrant or recovery code and then uses the interrupt vectors FFFE and FFFF, (or FFFA and FFFB), depending on whether or not an interrupt request or a non maskable interrupt request had occurred. It should be noted that the interrupt disable is turned on at this point by the microprocessor automatically.

Example 9.3: Return from Interrupt

Cycles	Address Bus	Data Bus	External Operation	Internal Operation
1	0300	RTI	Fetch OP CODE	Finish Previous Operation, Increment PC to 0301
2	0301	?	Fetch Next OP CODE	Decode RTI
3	01FC	?	Discarded Stack Fetch	Increment Stack Pointer to 01FD
4	01FD	P	Fetch P Register	Increment Stack Pointer to 01FE
5	01FE	PCL	Fetch PCL	Increment Stack Pointer to 01FF, Hold PCL
6	01FF	PCH	Fetch PCH	M→PCL, Store Stack Pointer
7	PCH PCL	OP CODE	Fetch OP CODE	Increment New PC

Note the effects of the extra cycle (3) necessary to read data from stack which causes the RTI to take six cycles. The RTI has restored the stack, program counter and status register to the point they were at before the interrupt was acknowledged.

Appendix J
EPROM blaster program

This listing is here so that you might get some tips from it on how to write BASIC and assembly language programs.

Basic program for EPROM blaster

```
]PR#0
]LIST

10   REM  EPROM.BLASTER
20   REM  A PROGRAM TO PROGRAM 2716 EPROMS
30   REM  USING A J.BELL PROGRAMMER
40   REM  B.THOMPSON
50   REM  15 NOV 83
60   REM
100  REM  MACHINE PROG ENTRY ADDRESSES
105  ONERR  GOTO 9000          This sets up error trap (see 9000)
110  BA = 28672: REM  $7000    Position of EPROM machine
                               language program in memory
120  RE = BA + 3
130  WR = BA + 6               Entry locations
132  CL = BA + 9
135  ER = BA + 13
140  BU = 24576: REM  BUFFER   Buffer for your program
     AT $6000
150  PRINT CHR$ (4);"BLOAD
     EPR.ASS"                  Load EPROM machine language
                               program
160  CALL BA                   Run initialization part
200  HOME                      Clear screen
210  VTAB 3: HTAB 10           Position cursor
220  PRINT "EPROM BLASTER
     PROGRAM"
230  VTAB 6: HTAB 10
240  PRINT "FOR TYPE 2716 EPROM'S"
```

```
250  GOSUB 6100
260  GOSUB 6200
270  GOSUB 6300
280  GOSUB 6400
330  INVERSE
340  VTAB 20: HTAB 10
350  PRINT "SELECT A NUMBER;
355  NORMAL
360  GET S
370  IF (S > 0 AND S < 5)
     GOTO 400
380  VTAB 22: HTAB 10
390  PRINT  CHR$ (7);"SELECTION
     OUT OF RANGE"
395  PRINT TAB( 10);"TRY AGAIN":
     GOTO 330
400  ON S GOTO 2000,3000,
     4000,420
420  HOME
430  VTAB 10: HTAB 19
439  FLASH
440  PRINT "BYE"
441  NORMAL
445  VTAB 24: HTAB 1
450  END
2000 REM  CHECK ERASED EPROM
2100 GOSUB 6020
2110 GOSUB 6100
2130 GOSUB 6700
2165 PRINT
2170 INVERSE
2180 PRINT : PRINT  TAB( 5);
     "NOW PRESS CR TO CHECK
     EPROM ";
2185 NORMAL
2190 INPUT " ";D$
2200 CALL RE
2210 GOSUB 6020
2220 GOSUB 6100
2225 VTAB 13: HTAB 10
2230 IF PEEK (CL) < > 255
     GOTO 2300
2240 PRINT "EPROM FULLY ERASED"
2250 GOSUB 6500
```

Print menu

Reverse video!

Get a character from keyboard

Check for out of range

Go to selected part of program

If selection #, say goodbye

Start erased check

Set up screen

Call read program

If CL location = 255 then EPROM is cleared

```
2270 GOTO 200                          Go back to menu
2300 PRINT CHR$ (7);"EPROM";
2310 FLASH                             If CL not 255 then EPROM not
2320 PRINT "NOT";: NORMAL              erased
2330 PRINT "FULLY ERASED";
     CHR$(7)
2335 PRINT
2340 GOSUB 6500
2350 GOTO 200                          Back to menu


3000 REM  BLAST EPROM                  Write EPROM
3100 GOSUB                             Set up screen
3110 GOSUB 6200
3120 GOSUB 6600
3130 INVERSE                           Check up on operator
3130 PRINT "HAVE YOU CHECKED
     THAT YOUR"
3144 HTAB 10
3145 PRINT "EPROM IS FULLY
     ERASED?
3150 GET A$
3160 IF A$ = "Y" GOTO 3200
3165 GOSUB 6020
3170 VTAB 12: HTAB 15
3175 FLASH
3180 PRINT "YOU SHOULD":
     NORMAL
3185 PRINT : PRINT ; PRINT :
     PRINT
3190 GOSUB 6500
3195 GOTO 200
3200 GOSUB 6600                        Check up again
3210 INVERSE
3220 PRINT "IS YOUR EPROM IN
     THE HOLDER?"
3230 NORMAL                            Wait until "Y"
3240 GET A$: IF A$ < > "Y"
     GOTO 3200
3250 GOSUB 6600
3260 INVERSE
3270 PRINT "ENTER THE FILENAME
     OF THE"
```

```
3275 HTAB 10
3276 PRINT-1"PROGRAM YOU WISH TO
     RECORD"
3277 HTAB 15
3280 NORMAL
3290 INPUT "";F$                    Get filename of your program
3300 VTAB 14: HTAB 10: GOSUB
     6000
3305 GOSUB 6600
3310 PRINT "LOADING"
3320 PRINT CHR$ (4);"BLOAD";        Go get file (if file not found error
     F$;",A$6000"                   occurs here, control switches to
                                    9000)
3400 GOSUB 6600
3410 PRINT "PRESS CR TO BLAST
     EPROM";
3420 INPUT "";D$
3430 GOSUB 6600
3440 FLASH : PRINT "BLASTING":
     NORMAL
3500 CALL WR                        Call blasting routine
3510 GOSUB 6600
3520 E = PEEK (ER) + PEEK
     (ER +1)*256                    Check that it was succcessful
3530 IF E < > 0 GOTO 3600
3540 PRINT CHR$ (7);"A
     SUCCESSFUL BLAST"
3550 GOSUB 6500
3560 GOTO 200
3600 PRINT CHR$ (7);"THE            Report success
     BLAST WAS";
3610 FLASH:PRINT"NOT";:
     NORMAL
3620 PRINT " SUCCESSFUL";
     CHR$ (7)
3630 PRINT
3640 PRINT TAB( 5);"DO YOU
     WANT TO TRY AGAIN?";
3650 GET A$
3660 IF A$ = "Y" GOTO 3400
3670 GOTO 200
4100 GOSUB 6020                     Read EPROM
4110 GOSUB 6300
4120 GOSUB 6700
```

```
4210 GOSUB 6500
4215 CALL RE
4220 GOSUB 6600
4225 HTAB 1
4230 PRINT "DATA AVAILABLE IN
     MEMORY $6800 TO $6FFF"
4240 GOSUB 6500
4250 GOTO 200                          Return to menu


6000 CALL  - 958: RETURN               This will clear screen from cursor
                                       to end of screen
6010 CALL  - 868: RETURN               Clear to end of line
6020 VTAB 10: HTAB 10
6030 GOSUB 6000                        Position cursor to V10,H10 and
6040 RETURN                            clear
6100 VTAB 10: HTAB 10
6110 PRINT "1 CHECK ERASED             } Position to V10,H10 and print
     EPROM"
6120 RETURN
6200 VTAB 12: HTAB 10
6210 PRINT "2 WRITE TO EPROM           } Etc.
     (BLAST)"
6220 RETURN
6300 VTAB 14: HTAB 10
6310 PRINT "3 READ FROM EPROM"         } Etc.
6320 RETURN
6400 VTAB 16: HTAB 10
6410 PRINT "4 EXIT TO                  } Etc.
     APPLESOFT BASIC"
6420 RETURN
6500 HTAB 10: INVERSE
6510 PRINT "CR TO CONTINUE";           } Position and print MSG, wait for
6515 NORMAL                            CR
6520 INPUT " ";D$
6530 RETURN
6600 VTAB 16: HTAB 10: GOSUB           } Position to V16,H10, clear to EOS
     6000
6610 RETURN
```

```
6700 VTAB 16: HTAB 5
6710 PRINT "PLACE THE EPROM
     IN THE HOLDER AND"
6720 PRINT  TAB( 5);"LOCK THE
     LEVER"
6730 PRINT  TAB( 5);"BE SURE
     THAT THE NOTCH"
6740 PRINT  TAB( );"IS
     ORIENTED CORRECTLY!"
6750 RETURN
9000 POKE 216,0
9010 EC = PEEK (222)
9020 IF EC < > 6 THEN 9100
9030 VTAB 14: HTAB 10: FLASH
9039 PRINT CHR$ (7);
9040 PRINT "FILE "F$;" NOT
     FOUND"
9041 PRINT  CHR$ (7);
9045 NORMAL
9050 CALL  - 3288
9060 ONERR  GOTO 9000
9070 GOTO 3250
9100 RESUME
```

Position and print

This routine checks for "file not found" error (no 6). If this error occurs control is returned to the program else if a different error stop program

Assembly language program EPR.ASS

```
                0010 ; Eprom Blaster for 2716s
                0020 ; Using the John Bell eprom blaster and memory-mate
                     ; interface.
                0030 ;
                0040 ; This version is for the Apple IIe with a 6522
                0050 ; interface in slot 5
                0060 ;
                0070                                    VIA addresses:
                0080 PORT1      .DE $C500
                0090 PORT2      .DE $C580
                0100 J1         .DE PORT1+1
                0110 J1DD       .DE PORT1+3
                0120 J2         .DE PORT1+0
                0130 J2DD       .DE PORT1+2
                0140 J3         .DE PORT2+1
                0150 J3DD       .DE PORT2+3
                0160 J4         .DE PORT2+0
                0170 J4DD       .DE PORT2+2
                0180 ;
                0190 ECONTROL   .DE J2
                0200 EDATA      .DE J1
                0210 EADDL      .DE J3
                0220 EADDH      .DE J4
                0230 ;
                0240                                    Control bits for 2716s (information only)
                0250 PGM        .DE %0001               Pin 18, % indicates binary
                0260 \OE        .DE %0010               Pin 20, \ indicates negative (bar).
                0270 \POWER     .DE %0100               Pin 24
                0280 \V24       .DE %1000               Pin 21
                0290 ;
                0300                                    Parameters.
                0310 PROGDATA   .DE $6000
                0320 EPROMDATA  .DE $6800
                0330 ;
                0340                                    Zero page.
                0350 ADDR       .DE $06
                0360 ADDS       .DE $08
                0370 ;
                0380 ;
                0390            .BA $7000
                0400 ;
7000- 4C 0F 70  0410            JMP INIT
7003- 4C 42 70  0420            JMP READ
```

```
7006- 4C 91 70  0430            JMP WRITE
                0440 ;
7009-           0450 CLEAR      .DS 2
700B-           0460 TIME       .DS 2
700D-           0470 ERROR      .DS 2
                0480 ;
700F- A9 60     0490 INIT       LDA #H,PROGDATA   Init indirect pointer
7011- 8D 07 00  0500            STA ADDR+1
7014- A9 00     0510            LDA #L,PROGDATA
7016- 8D 06 00  0520            STA ADDR
                0530 ;
7019- A2 00     0540            LDX #0            Put FF in all of program buffer.
701B- A0 00     0550            LDY #0
701D- A9 FF     0560            LDA #$FF
701F- 91 06     0570 A          STA (ADDR),Y
7021- C8        0580            INY
7022- D0 FB     0590            BNE A
7024- EE 07 00  0600            INC ADDR+1
7027- E8        0610            INX
7028- E0 08     0620            CPX #$08
702A- D0 F3     0630            BNE A
                0640 ;
702C- A9 FF     0650            LDA #$FF          Set up VIAs.
702E- 8D 02 C5  0660            STA J2DD
7031- 8D 83 C5  0670            STA J3DD
7034- 8D 82 C5  0680            STA J4DD
7037- A9 00     0690            LDA #0
7039- 8D 03 C5  0700            STA J1DD          Input for now.
703C- A9 0F     0710            LDA #%1111
703E- 8D 00 C5  0720            STA ECONTROL      Turn EPROM off.
                0730 ;
7041- 60        0740            RTS
                0750 ;
                0760 ;
                0770 READ
7042- A9 FF     0780            LDA #$FF          Init clear flag.
7044- 8D 09 70  0790            STA CLEAR
7047- A9 68     0800            LDA #H,EPROMDATA  Init indirect pointer.
7049- 8D 07 00  0810            STA ADDR+1
704C- A9 00     0802            LDA #L,EPROMDATA
704E- 8D 06 00  0830            STA ADDR
                0840 ;
                0850                              Set up VIAs for read.
7051- A9 00     0860            LDA #$00
```

```
7053- 8D 81 C5  0870            STA EADDL
7056- 8D 80 C5  0880            STA EADDH
7059- 8D 03 C5  0890            STA J1DD
705C- A9 08     0900            LDA #%1000      +24 off, +5 on.
705E- 8D 00 C5  0910            STA ECONTROL
                0920 ;
7061- AD 01 C5  0930 RLOOP      LDA EDATA       Set data from EPROM.
7064- C9 FF     0940            CMP #$FF        Check if EPROM data cleared.
7066- F0 03     0950            BEQ OKFF
7068- 8D 09 70  0960            STA CLEAR
706B  A0 00     0970 OKFF       LDY #$00        Store EPROM data in memory pointed
                                                to by ADDR.
706D- 91 06     0980            STA (ADDR),Y
706F- EE 06 00  0990            INC ADDR
7072- AD 06 00  1000            LDA ADDR
7075- 8D 81 C5  1010            STA EADDL
7078- C9 00     1020            CMP #$00        Test end of page.
707A- D0 E5     1030            BNE RLOOP
707C- EE 07 00  1040            INC ADDR+1      Go to next page.
707F- AD 07 00  1050            LDA ADDR+1
7082- 29 07     1060            AND #$07        Strip high bits.
7084- 8D 80 C5  1070            STA EADDH
7087- EA        1080            NOP
7088- EA        1081            NOP
7089- D0 D6     1090            BNE RLOOP
                1100 ;
                1110                            Done.
708B- A9 0F     1120            LDA #%1111      Turn off EPROM.
708D- 8D 00 C5  1130            STA ECONTROL
7090- 50        1140            RTS
                1150 ;
                1160 ;
                1170 ;
                1180 ;
                1190 WRITE
7091- A9 60     1200            LDA #H,PROGDATA Init indirect pointer.
7093- 8D 07 00  1210            STA ADDR+1
7096- A9 00     1220            LDA #L,PROGDATA
7098- 8D 06 00  1230            STA ADDR
                1240 ;
                1250                            Set up ports
709B- A9 00     1260            LDA #$00
709D- 8D 80 C5  1270            STA EADDH
70A0- 8D 81 C5  1280            STA EADDL
```

```
70A3- A9 02     1290            LDA #%0010
70A5- 8D 00 C5  1300            STA ECONTROL
70A8- A9 FF     1310            LDA #$FF        All outputs
70AA- 8D 03 C5  1320            STA J1DD
                1330 ;
                1340 ;
70AD- A0 00     1350 WLOOP      LDY #$00        Set data from memory.
70AF- B1 06     1360            LDA (ADDR),Y
70B1- C9 FF     1370            CMP #$FF        No need to do FFs.
70B3- F0 27     1380            BEQ NEXTADD
70B5- 8D 01 C5  1390            STA EDATA
                1400                            Start hot blast.
70B8- A9 03     1410            LDA #%0011
70BA- 8D 00 C5  1420            STA ECONTROL
                1430                            Start timer, 50 ms.
70BD- A9 80     1440            LDA #$80
70BF- 8D 0B 70  1450            STA TIME
70C2- A9 F0     1460            LDA #$F0
70C4- 8D 0C 70  1470            STA TIME+1
70C7- EE 0B 70  1480 TLOOP      INC TIME
70CA- AD 0B 70  1490            LDA TIME
70CD- D0 FB     1500            BNE TLOOP
70CF- EE 0C 70  1510            INC TIME+1
70D2- AD 0C 70  1520            LDA TIME+1
70D5- D0 F0     1530            BNE TLOOP
                1540                            End timer.
70D7- A9 02     1550            LDA #%0010      Stop hot blast.
70D9- 8D 00 C5  1560            STA ECONTROL
                1570 ;
70DC- EE 06 00  1580 NEXTADD    INC ADDR        Next data.
70DF- AD 06 00  1590            LDA ADDR
70E2- 8D 81 C5  1600            STA EADDL
70E5- D0 C6     1610            BNE WLOOP
70E7- EE 07 00  1620            INC ADDR+1
70EA- AD 07 00  1630            LDA ADDR+1
70ED- 29 07     1640            AND #$07        Strip high bits.
70EF- 8D 80 C5  1650            STA EADDH
70F2- EA        1660            NOP
70F3- EA        1661            NOP
70F4- D0 B7     1670            BNE WLOOP
                1680 ;
                1690                            Done.
70F6- A9 0F     1700            LDA #%1111      Turn off EPROM.
70F8- 8D 00 C5  1710            STA ECONTROL
```

```
70FB- A9 00     1720            LDA #$00
70FD- 8D 03 C5  1730            STA J1DD
7100- 20 42 70  1740            JSR READ
                1750 ;
                1760                                    Memory compare.
7103- A9 60     1770            LDA #H,PROGDATA
7105- 8D 07 00  1771            STA ADDR+1
7108- A9 00     1772            LDA #L,PROGDATA
710A- 8D 06 00  1773            STA ADDR
710D- A9 68     1780            LDA #H,EPROMDATA
710F- 8D 09 00  1781            STA ADDS+1
7112- A9 00     1782            LDA #L,EPROMDATA
7114- 8D 08 00  1783            STA ADDS
7117- A9  00    1790            LDA #0
7119- 8D 0E 70  1791            STA ERROR+1
711C- 8D 0D 70  1792            STA ERROR
711F- A2 00     1800            LDX #$00
7121- A0 00     1810            LDY #$00
7123- B1 06     1820 CLOOP      LDA (ADDR),Y
7125- D1 08     1830            CMP (ADDS),Y
7127- F0 08     1840            BEQ OKDATA
7129- EE 0D 70  1850            INC ERROR
712C- D0 03     1860            BNE OKDATA
712E- EE 0E 70  1870            INC ERROR+1
7131- C8        1880 OKDATA     INY
7132- D0 EF     1890            BNE CLOOP
7134- E8        1900            INX                     Next page.
7135- EE 07 00  1910            INC ADDR+1
7138- EE 09 00  1920            INC ADDS+1
713B- E0 08     1930            CPX #$08                End for 2716s.
713D- D0 E4     1940            BNE CLOOP
713F- 60        1950            RTS
                1960            .EN
```

```
LABEL FILE:  [ / =EXTERNAL ]

/PORT1=C500                 /PORT2=C580                 /J1=C501
/J1DD=C503                  /J2=C500                    /J2DD=C502
/J3=C581                    /J3DD=C583                  /J4=C580
/J4DD=C582                  /ECONTROL=C500              /EDATA=C501
/EADDL=C581                 /EADDH=C580                 /PGM=0001
/\OE=0002                   /\POWER=0004                /\V24=0008
/PROGDATA=6000              /EPROMDATA=6800             /ADDR=0006
/ADDS=0008                  CLEAR=7009                  TIME=700B
```

```
ERROR=700D          INIT=700F          A=701F
READ=7042           RLOOP=7061         OKFF=706B
WRITE=7091          WLOOP=70AD         TLOOP=70C7
NEXTADD=70DC        CLOOP=7123         OKDATA=7131

//0000,7140,7140
>
```

Appendix K
Bibliography and sources

General APPLE and 6502 programming

Apple IIe Reference manuals, Apple Computer.
These are quite good and contain the fine details and all APPLE hardware and software.

Poole, L., McNiff, M. & Cook, S., *Apple II User's Guide*, Osborne/McGraw-Hill, Berkeley, 2nd edn., 1983.
Good general reference on BASIC programming and the use of the MINIASSEMBLER.

SYNERTEK *6502 Programming Manual*, Publication No. 6500–50, Santa Clara, CA 95051
Details of op-codes and their uses.

Leventhal, L., *6502 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, 1979.
Easier to find than the SYNERTEK book.

General computing

BYTE Magazine.
Good general overview of microcomputing with frequent references to laboratory applications.

General numerical analysis

Press, W., Flannery, B., Tenkolsky, S. & Vetterling, W. *Numerical Recipes, The Art of Scientific Computing*, Cambridge Univ. Press, New York, 1986.

General electronics

Horowitz, P. & Hill, W., *The Art of Electronics*, Cambridge Univ. Press, New York 1980.
The best reference for designing laboratory electronics.

Physical data

Handbook of Chemistry and Physics, ed. R. Weast, 52nd edn, Chemical Rubber Co, Cleveland, OH, 1971.

American Institute of Physics Handbook, ed. D. E. Gray, McGraw-Hill, New York, 1957.

Mark's Standard handbook for Mechanical Engineers, eds. T. Beaumeister, E. A. Abalone & T. Baird, 8th edn, McGraw-Hill, New York, 1978

Physics

Any general introductory physics text will provide a good background.

Sensors and transducers

Doebelin, E. O., *Measurement Systems*, McGraw-Hill, New York, 1983.
A thorough overview of general design and specific devices.

Specific hardware

Witten, I. H., Welcome to the Standards Jungle, *BYTE*, pp. 146–78, February, 1983.
This a close look at serial data communication.

Leibson, S., The Input/Output Primer, Part 3: The Parallel and HPIB (IEEE-488) Interfaces, *BYTE*, pp. 186–208, April, 1982.

Clune, T. R., Interfacing for Data Acquisition, *BYTE*, pp. 269–82, February, 1985.
These two articles provide a good background in how the IEEE-488 works.

Hallgreen, R. C., Putting the Apple II to Work, Part 1: The Hardware, *BYTE*, pp. 152–64, April, 1984.
This and a succeeding article in BYTE in May 1984 describe a particular data acquisition system.

General signal analysis

Bendat, J. S. & Piersol, A. G., *Random Data*, Wiley, New York, 1971.

Otnes, R. K. & Enochson, L., *Applied Time Series Analysis*, Wiley, New York, 1978.

Papoulis, A., *Signal Analysis*, McGraw-Hill, New York, 1977.

Specific signal analysis

Monforte, J., The Digital Reproduction of Sound, *Scientific American*, pp. 78–84, December, 1984.
A good description of the sampling problem and digitization.

Cacerci, M. S. & Cacheris, W. P., Fitting Curves to Data, *BYTE*, pp. 340–62, May, 1984.
A description of the Simplex algorithm.

Report writing

Porawn, J. F., *A Student Guide to Engineering Report Writing*, United Western Press, Soloma Beach, 1985.

Hofstaedter, D., Default Assumptions in Metamathecal Themas, *Scientific American*, November, 1983.
For those interested in exorcising the spectre of maleness from their writing.

Sources

John Bell Engineering, Inc., 400 Oxford Way, Belmont, CA 94002
ADC board
6522 interface board
EPROM programmer.

MADWEST Software, P.O. Box 9822, Madison, WI 53715
AMPERGRAPH, for drawing graphs
AMPERDUMP, for printing graphs.

Electronic chips, stepping motors, etc.
Look in the back of BYTE mgazine for numerous sources for these items.

Index