Apple Instrumentation & Control

Circuits & Software

Jerome Oleksy

860882

TK 7887.5 .044 1986
Oleksy, Jerome E.
Apple instrumentation and control.

	DATE DUE		
10-3-86			
15 Nov 86			
4 Sept. '87			

Apple® Instrumentation and Control: Circuits and Software

Apple® Instrumentation and Control: Circuits and Software

JEROME E. OLEKSY

A Reston Book
Prentice-Hall, Inc.
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data
Oleksy, Jerome E.
Apple Instrumentation and control.
1. Computer interfaces. 2. Apple computer.
I. Title.
TK7887.5.044 1986 001.64'4 85-2348
ISBN 0-8359-9222-5

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

A Reston Book
Published by Prentice-Hall, Inc.
A Division of Simon & Schuster, Inc.
Englewood Cliffs, NJ 07632

CONTENTS

PREFACE

Interfacing a computer to real-world loads results in making the computer do what you want it to do. You can make the computer measure several input quantities, such as voltage, temperature, or fluid level. Then, based on the results of these measurements, your computer can control an industrial process, or simply control a home appliance. Apple computer owners have a very powerful and flexible system available to them. This book will show you how to make use of that system.

The projects given in this book will teach you how to interface to parallel loads, such as printers and switches and motors, and to serial ports, such as those used in communicating over long distances. It will teach you how to use your computer as a measuring instrument, such as a DVM, a storage scope, or a waveform generator. But most importantly, it will teach you how to interface to almost any device you choose. In other words, rather than being a collection of "ten things to do with your computer," this book will teach you how to do your own interfacing to a wide variety of devices.

In addition to the hardware aspects of interfacing, this book will demonstrate the best of both software worlds: high-level language and machine language. In order to keep the size of the book reasonable, I will not attempt to teach BASIC. Nor will I teach assembly language programming. But instead I will show how the two languages are tied together in practical applications programs. You will learn why BASIC is better for some purposes, but machine language is better for others.

From my industrial experience as a supervisor of a hardware and software design group, I have learned the need for good software design techniques. I will show by a few examples how you can painlessly design software that is not only very readable but also easily modified to cope with any later changes that may arise.

The book will present several projects that you can build, usually for just a few dollars worth of parts. These parts can be assembled on a printed circuit board to plug into one of the Apple's slots. The format for each application will include:

1. A general discussion of the application—what, why, how
2. Hardware details
3. Software details
4. Suggestions for modification

This book will be valuable to serious hobbyists, as well as to engineers and technicians who use computers on the job, or who have a need to know about computer interfacing. It will also be valuable to schools teaching computer technology, both from the theoretical aspect, as well as for the many lab projects that can be derived from it.

Lastly, a copy of all BASIC and machine language programs listed in this book is available without documentation on a DOS 3.3 16 sector diskette. Write to the author c/o INTRON Company, 12932 Carpenter Rd., Garfield Hts., Ohio 44125.

Jerome E. Oleksy

Apple® Instrumentation and Control: Circuits and Software

CHAPTER 1

Introduction to Apple Interfacing

Interfacing a computer refers to tying the computer to external devices and/or systems. That is what this book is all about. We will discuss how to connect your Apple computer to lamps, motors, switches, printers, A/D and D/A converters, and much more. We will also talk about how to use your Apple®* as a test instrument, such as a DVM, a storage scope, and a waveform generator. Finally, we will discuss how to build a general purpose serial interface so that you can tie your Apple to a wide variety of commercial equipment.

Besides the hardware aspects of interfacing, we will discuss the software involved in making your computer do what you want it to do. In addition to creating many useful, ready-to-run programs, we will develop good programming procedures so that your programs will be not only readable, but easy to modify. If you save all of the programs listed in this book on disc, you will have created a library of useful routines that can be incorporated into different applications programs later.

This chapter will give you a general overview of what parts of the computer we will be using, what the control signals do, and what the symbols mean. The hardware and software in this book will run on Apple® II, II +, and IIe** computers.

1-1 THE HARDWARE

This book will not attempt to duplicate or replace the vast amount of technical information and specifications already contained in your Apple Reference Manual. Instead, we will indicate which signals are to be used for each project and which memory addresses to use. You should keep the Reference Manual handy whenever you do any interfacing. Whenever you want to modify a project significantly, be sure to consult your Reference Manual to determine such things as maximum loading on any signal line, maximum current to be drawn from a given power supply, and range of addresses affected.

*Apple® is a registered trademark of Apple Computer, Inc., Sunnyvale, California.

**Apple® II, II Plus, and IIe are registered trademarks of Apple Computers, Inc.

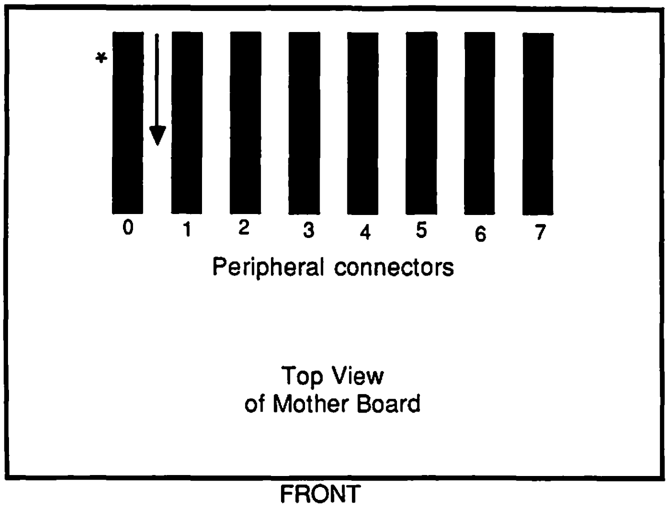


Figure 1-1
Location of Apple peripheral connectors

If you remove the cover of your Apple computer, you will see the interface connectors (slots) along the back of the mother board. These slots are numbered 0 through 7, as shown in figure 1-1. (Apple IIe does not have a slot 0.) We will be using these peripheral slots to interface the computer to outside world devices. Generally, slot 1 is used to connect to a printer, and slot 6 is used to connect to floppy disc drives. The remainder of the slots are general purpose and we will use different ones from time to time.

All of the peripheral connectors are tied to the microprocessor's address bus, data bus, and control bus. The connector pinout is shown in figure 1-2. Interface cards having 50 pin-edge connectors with 100 mil spacing are plugged into the slots and communicate with the MPU via the various buses. A wide variety of interface cards are commercially available, but often the commercial card is an expensive overkill for a simple project you wish to handle. So we will be using custom-built interface cards for our projects. You can purchase high quality, bare, prototype, printed circuit cards such as the Apple A2B0001X Hobby/Prototyping board or the VECTOR 4609 board. Both of these boards are specifically designed to plug into your Apple with your custom-built circuit on it. They allow you to use wire wrap or point-to-point connections

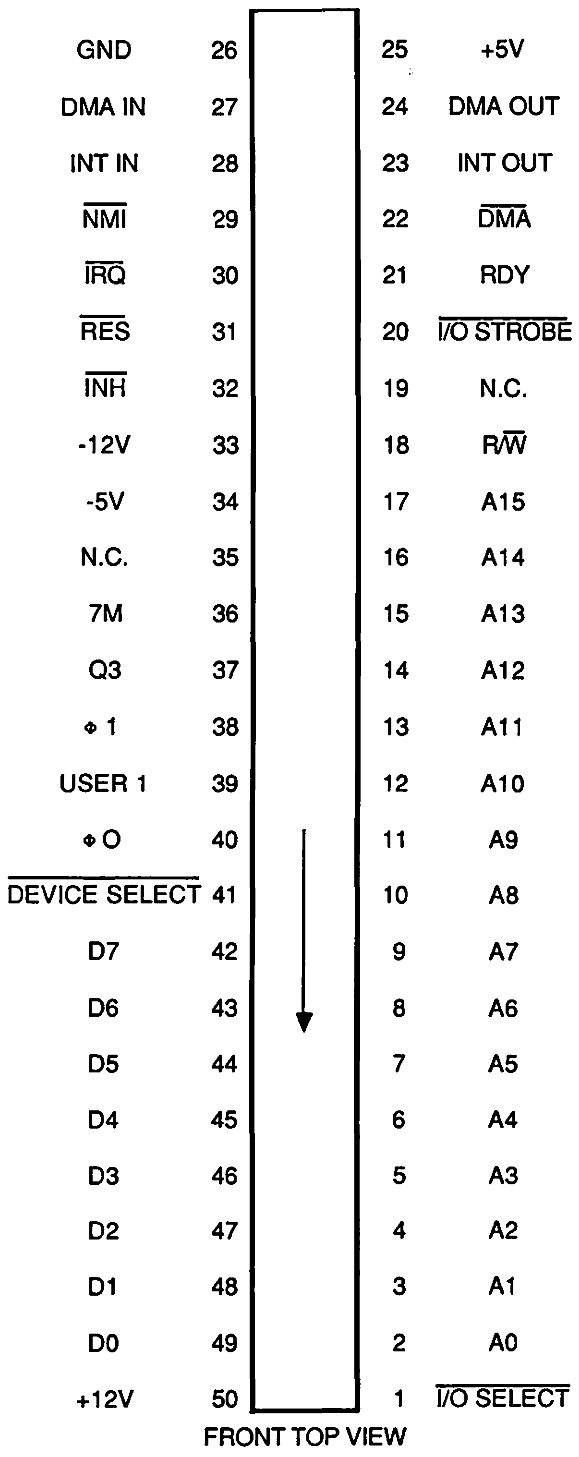

GND 26 | 25 +5V
DMA IN 27 | 24 DMA OUT
INT IN 28 | 23 INT OUT
$\overline{\text{NMI}}$ 29 | 22 $\overline{\text{DMA}}$
$\overline{\text{IRQ}}$ 30 | 21 RDY
$\overline{\text{RES}}$ 31 | 20 $\overline{\text{I/O STROBE}}$
$\overline{\text{INH}}$ 32 | 19 N.C.
-12V 33 | 18 R/$\overline{\text{W}}$
-5V 34 | 17 A15
N.C. 35 | 16 A14
7M 36 | 15 A13
Q3 37 | 14 A12
Φ 1 38 | 13 A11
USER 1 39 | 12 A10
Φ O 40 | 11 A9
$\overline{\text{DEVICE SELECT}}$ 41 | 10 A8
D7 42 | 9 A7
D6 43 | 8 A6
D5 44 | 7 A5
D4 45 | 6 A4
D3 46 | 5 A3
D2 47 | 4 A2
D1 48 | 3 A1
D0 49 | 2 A0
+12V 50 | 1 $\overline{\text{I/O SELECT}}$
FRONT TOP VIEW

Figure 1-2
Peripheral connector pinout

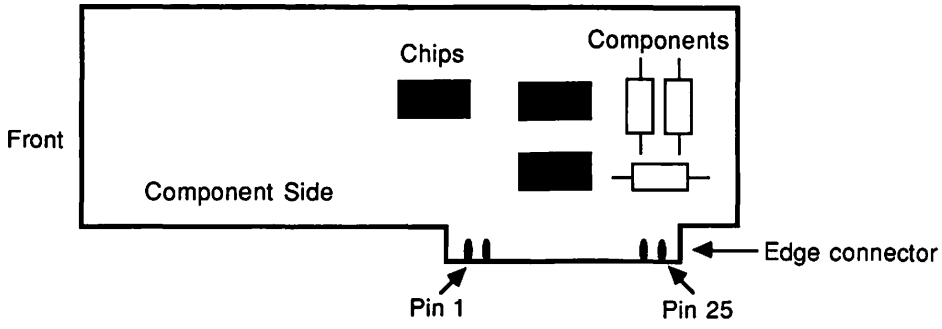


Figure 1-3
Hobby/prototype board

to build your prototype. Figure 1-3 shows a typical prototype board with a few components mounted on it.

Since the 6502 microprocessor uses memory-mapped input/output (I/O), Apple reserves certain memory locations for I/O devices. Figure 1-4 shows the memory map of the Apple computer. Note that a 4K block of memory, from $C000 to $CFFF, is reserved for I/O. (The dollar sign ($) signifies a hexadecimal number.) The peripheral connectors are decoded by the Apple to respond to addresses within that range. We will discuss the specific addresses for each connector in later chapters.

1-2 THE SIGNALS

As was stated, the peripheral connector links the peripheral card to the MPU via a host of signal lines. Table 1-1 gives a brief description of the peripheral signals. Your Apple Reference Manual gives this table in greater detail, but it is repeated here for your convenience. Most of the signals are self-explanatory, but we will discuss a few here.

Address lines A0-A15 are output lines from the 6502 microprocessor. The MPU outputs the address of the memory location it wants to talk to when doing a read or a write. These lines can be used on the peripheral card to decode a particular address, for example, to activate a printer. Similarly, data lines D0-D7 connect to

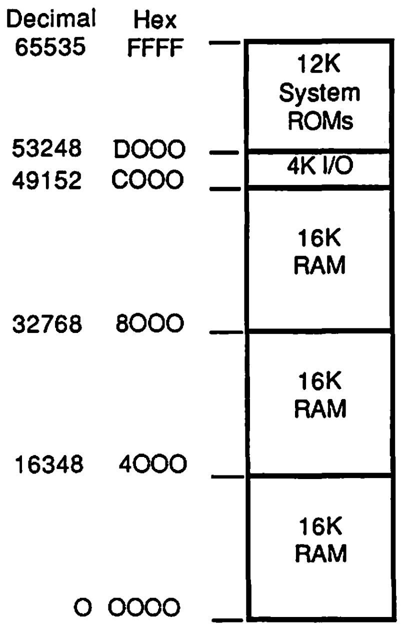


Figure 1-4
Apple memory map

the MPU and are used to transfer 8 bits of data or instruction to or from the MPU.

The power supply lines will be used to power our interface cards. If you modify any of the circuits given in the book, be sure not to exceed the maximum current rating of any supply. And be careful to double check your interface card for possible wiring errors before plugging into the connector or possible damage or malfunction of the computer may result.

Note that some of the signal names, such as $\overline{\text{NMI}}$ and $\overline{\text{IRQ}}$, have *overbars*. The overbars indicates that the signals are *active LOW*. An active LOW signal is one that operates when the level is LOW (at ground potential). If a signal name is *not overbarred*, that signal is *active HIGH*. An active HIGH signal activates something or causes some event to occur when the level is HIGH (logic one).

To expand on the previous point for a moment, refer to figure 1-5. The gates shown in the figure are drawn to MIL SPECS (military specifications). This form of symbol is used by most manufacturers of computer equipment and clearly describes what

TABLE 1-1
Peripheral Connector Signal Description

Signal	*Description*
$\overline{\text{I/O SELECT}}$	This line, normally HIGH, becomes LOW when the MPU references page $Cn, where n is the slot number.
A0-A15	Buffered address bus.
R/$\overline{\text{W}}$	Read/Write signal.
SYNC	On peripheral connector 7 only, connected to video timing generator's SYNC signal.
$\overline{\text{I/O STROBE}}$	This line goes LOW during Φ0 when the address bus contains an address between $C800 and $CFFF.
RDY	RDY input on 6502.
$\overline{\text{DMA}}$	Pulling this line LOW disables the 6502's address bus and halts the MPU.
INT OUT	Daisy-changed interrupt output to lower priority devices.
DMA OUT	Daisy-changed DMA output to lower priority devices.
+5v	+5v power supply. 500 ma is available for all peripheral devices.
GND	System ground.
DMA IN	Daisy-chained input from higher priority devices.
INT IN	Daisy-changed input from higher priority devices.
$\overline{\text{NMI}}$	Nonmaskable interrupt.
$\overline{\text{IRQ}}$	Interrupt request.
$\overline{\text{RES}}$	System reset input.
$\overline{\text{INH}}$	When this line is pulled LOW, all ROMs on Apple board are disabled
−12v	−12v power supply. Maximum current for all peripheral boards is 200 ma.
−5v	−5v power supply. Maximum current for all peripheral boards is 200 ma.
COLOR REF	On connector 7 only, this pin is connected to color refrence signal.
7M	7 MHz clock.
Q3	2 MHz asymmetrical clock.
Φ1	MPU's phase one clock.
USER 1	This line, when pulled LOW, disables all internal I/O address decoding.
Φ0	MPU's phase 0 clock.
$\overline{\text{DEVICE SELECT}}$	This line becomes active LOW on each peripheral connector when the address bus is holding an address between $C0n0 and $C0nF, where n is the slot number plus $8.
D0-D7	Buffered bidirectional data bus.
+12v	+12v power supply. Maximum current for all peripheral boards is 250 ma.

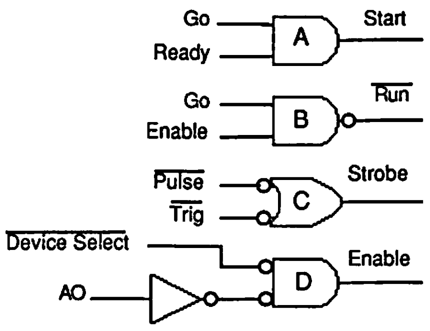


Figure 1-5
Gate symbols showing use of signal names

each gate does. For example, look at AND gate A. You will recall that an AND gate has a HIGH output only when both of its inputs are HIGH. The fact that no *bubbles* appear on any line indicates active HIGH inputs and an active HIGH output. The signal names are consistent with the symbol. Signal GO and signal READY must both be HIGH in order for output signal START to be HIGH.

NAND gate B has a bubble on its output. The output of this gate will be LOW when both inputs are HIGH. Notice that the input signals do not have overbars, but the output signal $\overline{\text{RUN}}$ *does* have an overbar. This signifies that when GO and ENABLE are both high, $\overline{\text{RUN}}$ will be LOW.

Likewise, gate C, which is also a NAND gate, is drawn in its OR form. This form more clearly shows that if either $\overline{\text{PULSE}}$ *or* $\overline{\text{TRIG}}$ is LOW, output STROBE will be HIGH.

Finally, NOR gate D will have its output signal ENABLE active HIGH when $\overline{\text{DEVICE SELECT}}$ is LOW and A0 is HIGH. The inverter on the A0 line makes the input of gate D LOW when A0 is HIGH.

Now, let's get back to table 1-1. Three control signals that are commonly used to activate peripheral cards are $\overline{\text{I/O SELECT}}$, $\overline{\text{I/O STROBE}}$, and $\overline{\text{DEVICE SELECT}}$. Signal $\overline{\text{I/O SELECT}}$ is active LOW whenever the MPU references an address in the range \$Cn00 through \$CnFF, where n is the slot number. For example, $\overline{\text{I/O SELECT}}$ (pin 1) of slot 4 will be LOW whenever the MPU references any location in the range \$C400 through \$C4FF. But pin 1 of all other slots will be inactive HIGH at those times. $\overline{\text{I/O SELECT}}$ can be used to enable a block of 256 bytes of memory, such as ROM, on a peripheral card. This ROM could contain a specific driver subroutine to handle the I/O operation. By having the subroutine on ROM, the user does not have to tie up RAM to hold the routine. And remember, since this area of memory is reserved specifically for this particular I/O slot there will not be any interference with any other information stored elsewhere in memory.

Signal line $\overline{\text{I/O STROBE}}$ goes active LOW whenever the MPU references any address in the 2K byte range from \$C800 through \$CFFF. It is common to all slots. This signal can also be used to activate some on-board memory. But if you use $\overline{\text{I/O STROBE}}$, be sure none of the commercial cards that you may have are also using some of that address range. Otherwise, your card and the other card may both be activated at the same time, causing contention on the data bus.

$\overline{\text{DEVICE SELECT}}$ is a control signal that we will use frequently. As explained in table 1-1, $\overline{\text{DEVICE SELECT}}$ goes LOW whenever the MPU references an address between \$C0n0 and \$C0nF, where n is the slot number +8. For example, $\overline{\text{DEVICE SELECT}}$ will be LOW when the MPU outputs any address between \$C090 and \$C09F, which are the 16 lowest addresses assigned to slot 1. Similarly, $\overline{\text{DEVICE SELECT}}$ will be LOW whenever the MPU outputs an address between \$C0A0 and \$C0AF, the lowest 16 addresses assigned to slot 2, and so on. We can use $\overline{\text{DEVICE SELECT}}$ along with address lines A0-A3 to specify any one of 16 different I/O devices for any given slot. You will see the details of how this signal is used in several applications throughout this book.

1-3 THE SOFTWARE

It is assumed that you understand the BASIC programming language. In this book you will find many complete BASIC programs that are ready to run. You will also learn how to develop your own efficient and clear programs so that they can be easily read and modified later.

```
]LIST

10000  REM  * HEX TO DEC CONVERT *
10010 :
10020  INPUT "ENTER HEX VALUE ...";HEX$
10030 SIZE =  LEN (HEX$)
10040 DECIMAL = 0
10050  FOR A = 1 TO SIZE
10060 DIGIT$ =  MID$ (HEX$,A,1)
10070  IF  ASC (DIGIT$) >  = 48 AND  ASC (DIGIT$) <  = 57 THEN AMOUNT = (16 ^ (S
IZE - A)) * ( ASC (DIGIT$) - 48)
10080  IF  ASC (DIGIT$) >  = 65 AND  ASC (DIGIT$) <  = 70 THEN AMOUNT = (16 ^ (S
IZE - A)) * ( ASC (DIGIT$) - 55)
10090 DECIMAL = DECIMAL + AMOUNT
10100  NEXT A
10110  PRINT "$";HEX$;" = ";DECIMAL
10120  END



]RUN
ENTER HEX VALUE ...2E
$2E = 46



 RUN
ENTER HEX VALUE ...300
$300 = 768



]RUN
ENTER HEX VALUE ...C0A2
$C0A2 = 49314
```

Figure 1-6
Hex-to-decimal converter listing

You will use machine language (machine code) in some of the applications. Complete machine language listings will be given in the text for any routines used. So if you are not familiar with machine language programming, do not worry. All the information you need to enter the routine and make it work will be given.

In some of the applications, you will be using hexadecimal or binary numbers as well as decimal numbers. You may have to convert from hexadecimal to decimal or from decimal to hexadecimal if you wish to change some addresses from those given in the listings. A couple of simple BASIC routines can aid you in converting if you do not know how. Figure 1-6 shows the listing of a routine to convert from hexadecimal to decimal. It also shows a few sample runs of the program. When you run the program, you will

```
]LIST

5  REM  * DECIMAL TO BIN CONVERT *
10  DIM S(16)
20  INPUT "DECIMAL =? ";V
30  FOR N = 1 TO 16
40 Q = V / 2
50 Q1 =  INT (Q)
60 S(N) = (Q - Q1) * 2
70 V = Q1
80  NEXT N
90  PRINT
95  PRINT "BINARY= "
100  FOR N = 16 TO 1 STEP  - 1
110  PRINT S(N);
115  IF N = 5 OR N = 9 OR N = 13 THEN  PRINT " ";
120  NEXT N




]RUN
DECIMAL =? 45

BINARY=
0000 0000 0010 1101


]RUN
DECIMAL =? 49153

BINARY=
1100 0000 0000 0001



]RUN
DECIMAL =? 32767

BINARY=
0111 1111 1111 1111
```

Figure 1-7
Decimal-to-binary converter listing

be asked to enter a hexadecimal value of up to several digits. The computer will then calculate and print the decimal equivalent of the hexadecimal value. For example, if you enter the hexadecimal value 2E, the computer will print the decimal value 46. If you enter the hexadecimal value 300, the computer will print the decimal value 768, and so on.

Figure 1-7 shows the listing for a decimal to binary conversion. You can enter any decimal value up to 65,535 and the computer will print out the binary equivalent of that number. Note in the sample run that the binary number is printed out to 16 digits and that the digits are arranged in groups of four. This is done so that you can easily change the binary value to its hexadecimal equivalent if you want to. For example, in the second sample run, the user entered the decimal value 49153, and the computer printed out the binary value

```
1100 0000 0000 0001.
```

You can mentally convert this binary number to the hexadecimal number $C001.

CHAPTER 2

Parallel I/O

We frequently want to pass information into or get data out of our computer. The fastest and most common way of doing this is by passing the data in 8-bit bytes, each byte representing a character, a number, a control code, or something similar. When all 8 bits are transferred at the same time, the process is referred to as *parallel I/O* (parallel input/output). The circuits or devices that actually pass the data are called I/O *ports*. This chapter discusses some typical ways of building I/O ports.

2-1 MEMORY-MAPPED I/O

Since the Apple computer uses a 6502 microprocessor as its MPU, all input/output ports are *memory mapped*. That is, whenever the computer has to read information from an input device, for example, a bank of switches, it does the same operation as if it were reading from a memory location. Similarly, whenever the computer must send information to an output port, such as a group of indicator lamps, it performs a write operation exactly as if it were writing into memory. The interface hardware, therefore, must make the corresponding input or output device "look like" a memory location to the computer.

The programmer can specify the address of the memory location to be written into by using the POKE command. For example, if you want to store the decimal value 14 in memory location 862, you simply use the following line in your program:

```
POKE 862,14
```

The largest value you can store in any one memory location is 255 decimal, corresponding to $FF, since the memory locations are 8 bits wide.

Following are three different structures you can use with the POKE command:

```
10 POKE 862,14
```

```
10 LET MEM = 862
20 POKE MEM, 14

10 LET MEM = 862
20 LET N = 14
30 POKE MEM, N
```

The third structure is the most general and the most self-documenting way. We will discuss program documentation more in later chapters.

The output device address must not be the same as that used by any actual RAM or ROM location. Otherwise, confusion will result from two devices (memory and the output port) both being enabled at the same time. Normally, some portion of a computer's total memory space is allocated to I/O devices so that the aforementioned problem does not arise.

The Apple computer has reserved memory addresses from $C000 to $CFFF (49152 to 53247) for I/O space or additional user memory. As mentioned in chapter 1, Apple also has provided special interface control signals at each interface slot to aid in I/O decoding. A suitable control signal for our output latch is $\overline{\text{DEVICE SELECT}}$. $\overline{\text{DEVICE SELECT}}$ is active LOW for 16 addresses for each slot as shown in table 2-1.

TABLE 2-1
Peripheral Slot I/O Locations Enabled By $\overline{\text{DEVICE SELECT}}$

Slot	*Address Range*	
0*	C080-C08F	49280-49295
1	C090-C09F	49296-49311
2	C0A0-C0AF	49312-49327
3	C0B0-C0BF	49328-49343
4	C0C0-C0CF	49344-49359
5	C0D0-C0DF	49360-49375
6	C0E0-C0EF	48376-49391
7	C0F0-C0FF	49392-49407

*Slot 0 is available on Apple II and Apple II+, but not on the Apple IIe. On the older models, slot 0 is normally used for a language card or ROM card, whose function is built into the main circuit board of the IIe.

2-2 OUTPUT LATCHES

Figure 2-1 shows a simple way to build an output port that looks like a memory location to the computer. The actual output device is a 74LS373 octal latch chip. The 74LS373 latches data from the data bus at the instant the signal labeled LATCH ENABLE is driven high. The signal LATCH ENABLE is generated by a device decoder, which can simply be a few gates or decoder chips driven by the address bus and the control bus. Latching of the output data is

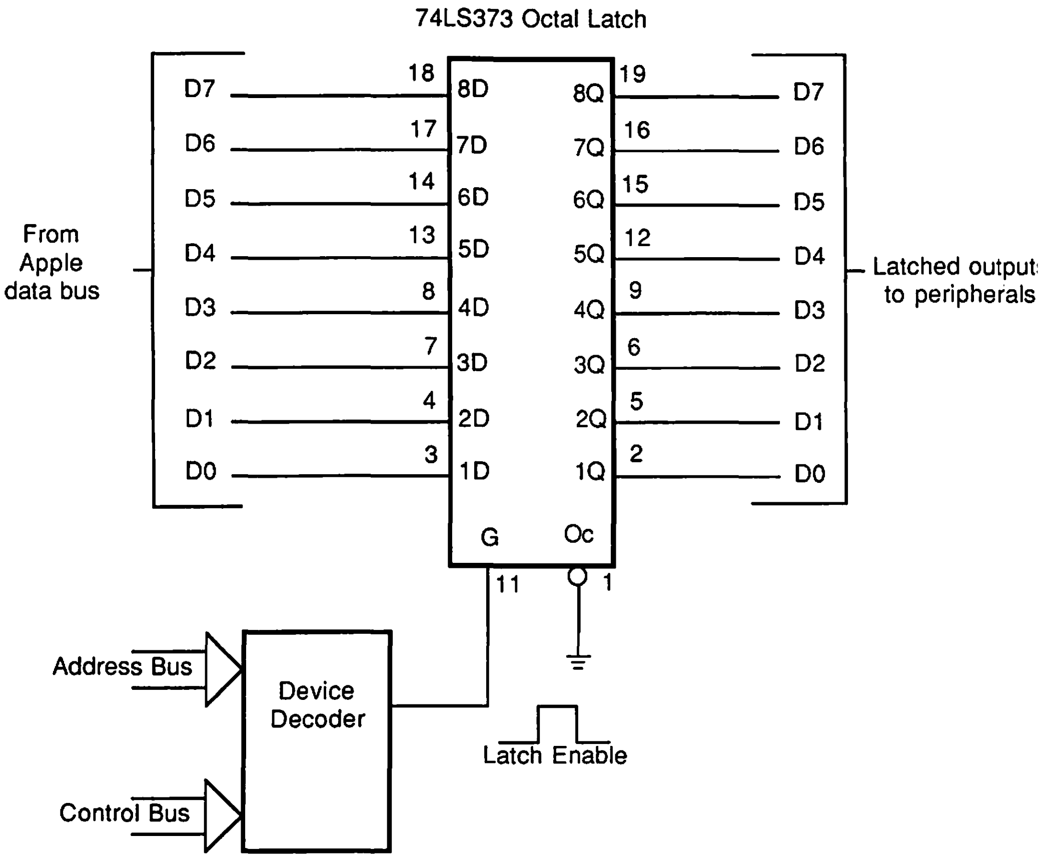


Figure 2-1
Parallel output port

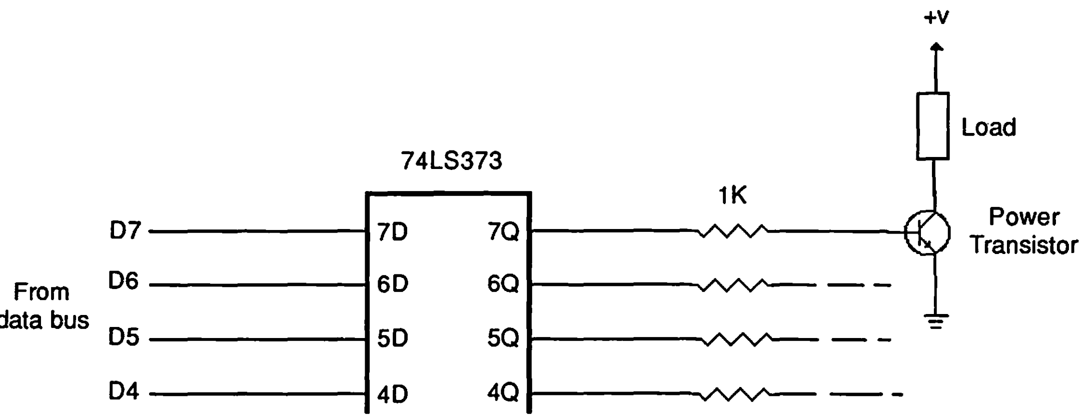


Figure 2-2
Using a power transistor to deliver more current to a load

necessary because the data is only present on the data bus for about 500 ns while the computer is writing to memory. Immediately after the write operation, the MPU uses the data bus to fetch the next instruction. But the output data must remain stable as long as it is needed. The latches retain the data as if it were stored in memory until the next time LATCH ENABLE goes HIGH.

Our latch provides TTL-compatible outputs to drive other circuits. If more output current is needed than is available from the latch, you can connect a power transistor, like the one shown in figure 2-2, to each latch output. When the latch output goes HIGH, the power transistor turns on, applying power to the load. This method can also be used to interface loads that must operate with a higher dc voltage than the normal +5v of the latch.

As a very simple interfacing example, suppose we build our output latches on a circuit board using only the hardware shown in figure 2-3. This circuit uses a pair of 74LS175 quad latches, which have both Q and $\overline{Q}$ outputs. The outputs of the latch chips drive the LEDs to display the data that was on the data bus when the LATCH ENABLE went HIGH. Notice that the $\overline{Q}$ outputs of the latches are used rather than the Q outputs. The reason is that the latch outputs can sink (return to ground) up to 16 ma of output current, but they cannot source (supply) much current. So when the Q output is latched HIGH (logic 1), the $\overline{Q}$ output goes LOW, pulling the cathode

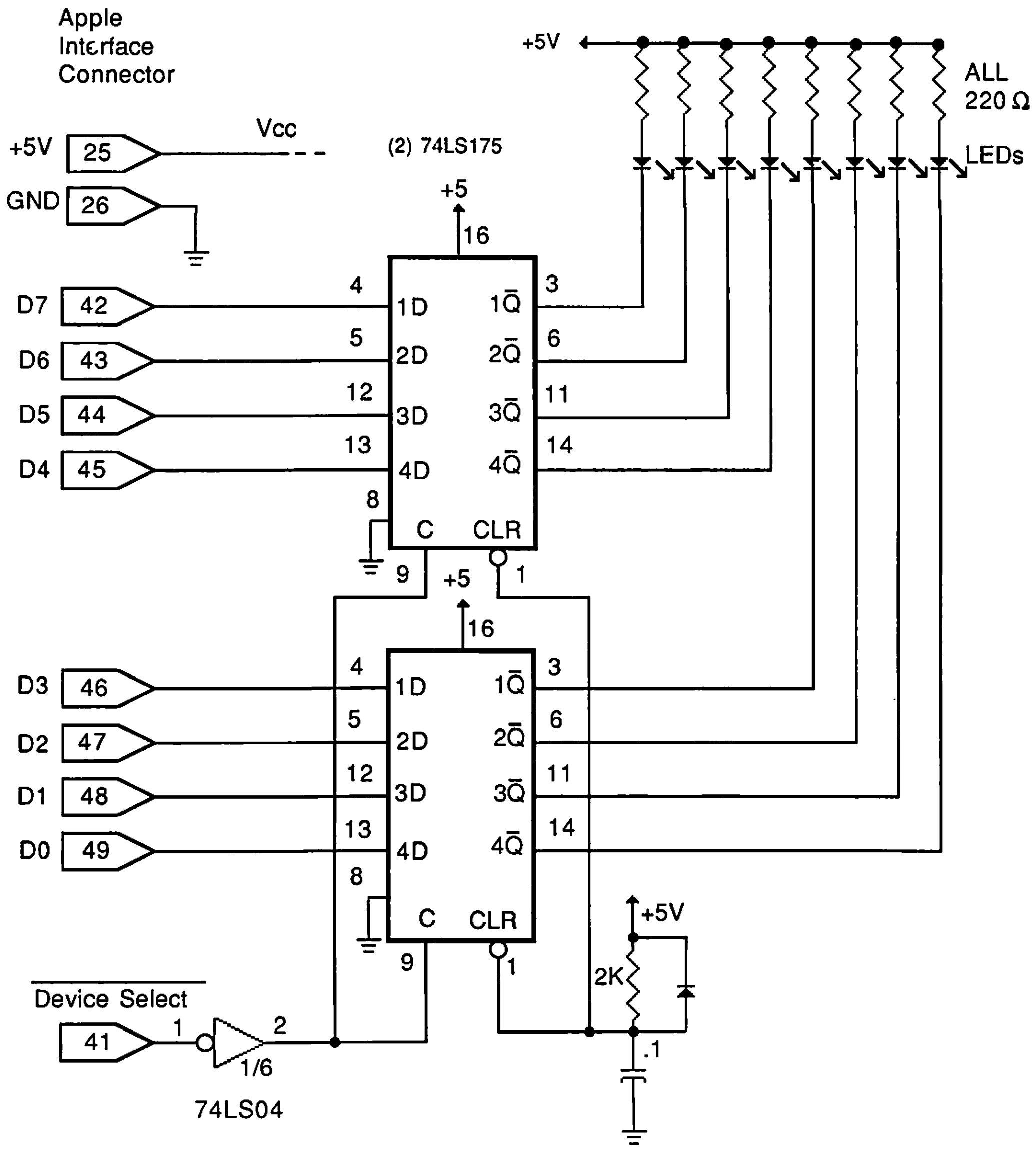


Figure 2-3
Display drivers

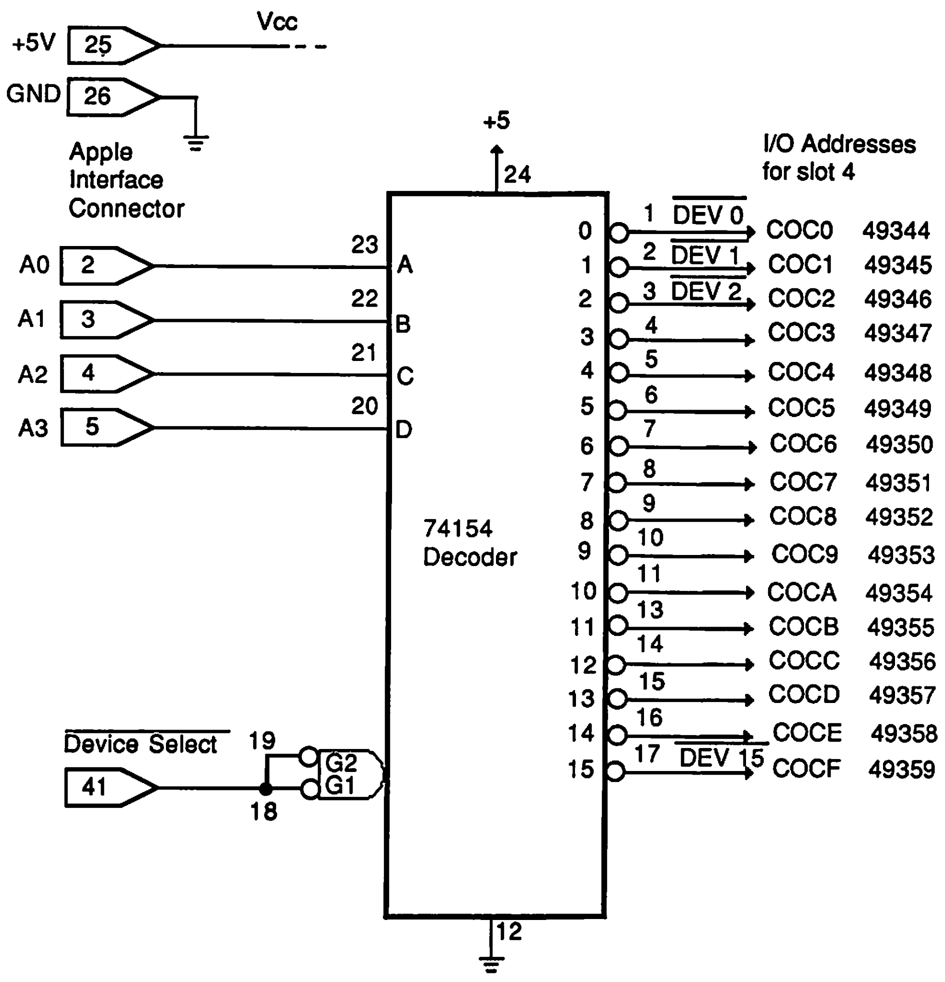


Figure 2-4
Decoding 16 unique addresses for each slot

of the corresponding LED to ground and causing it to light. Each LED has a series resistor to limit the current through it to about 10 to 15 ma.

Note also that the $\overline{\text{CLEAR}}$ inputs of the latch chips are connected to an R-C network which acts as a power-up reset. This ensures that all Q outputs are in the off (LOW) state when the computer is first powered up.

Suppose we plug the card into slot 4 of our Apple. We can now display data on the LEDs by executing the BASIC instruction

```
POKE 49344,N
```

where N is the value we want to display and 49344 is the address of slot 4.

At the instant the computer executes the POKE (store) instruction, the signal on pin 41 of that slot, called DEVICE SELECT, is pulled active (LOW) for about 500 ns. This drives the output of the 74LS04 HIGH, thus enabling the latches and causing data from the data bus to appear on the LEDs.

If we plug our card into slot 5 instead of slot 4, we simply change the instruction to POKE 49360,N. In other words, our hardware will work in any slot (1-7). We simply must tell the computer which slot to send the data to by giving the slot's appropriate address.

Here is a simple routine to output a binary count from 00000000 to 11111111 at a 1-second counting rate to the LEDs connected to the card in slot 5:

```
10 FOR N = 0 TO 255
20 POKE 49360,N
30 FOR D = 1 TO 760 : NEXT D
40 NEXT N
```

Line 30 inserts a time delay of approximately 1 second. You can change the delay period by simply changing the final value for D.

As was mentioned, DEVICE SELECT is LOW for 16 addresses in each slot, so the latch in slot 4 actually responds to any address from 49344 to 49359. If you want to have several I/O devices connected to slot 4, you can further decode the I/O address by using address lines A_3-A_0, as shown in Figure 2-4. each of the 16 devices will then have a unique address.

2-3 INPUT BUFFERS

Whenever we want to input a byte of data to our computer, we memory map an input port and use the PEEK command, as if we were reading from memory. For example, if you want to read the

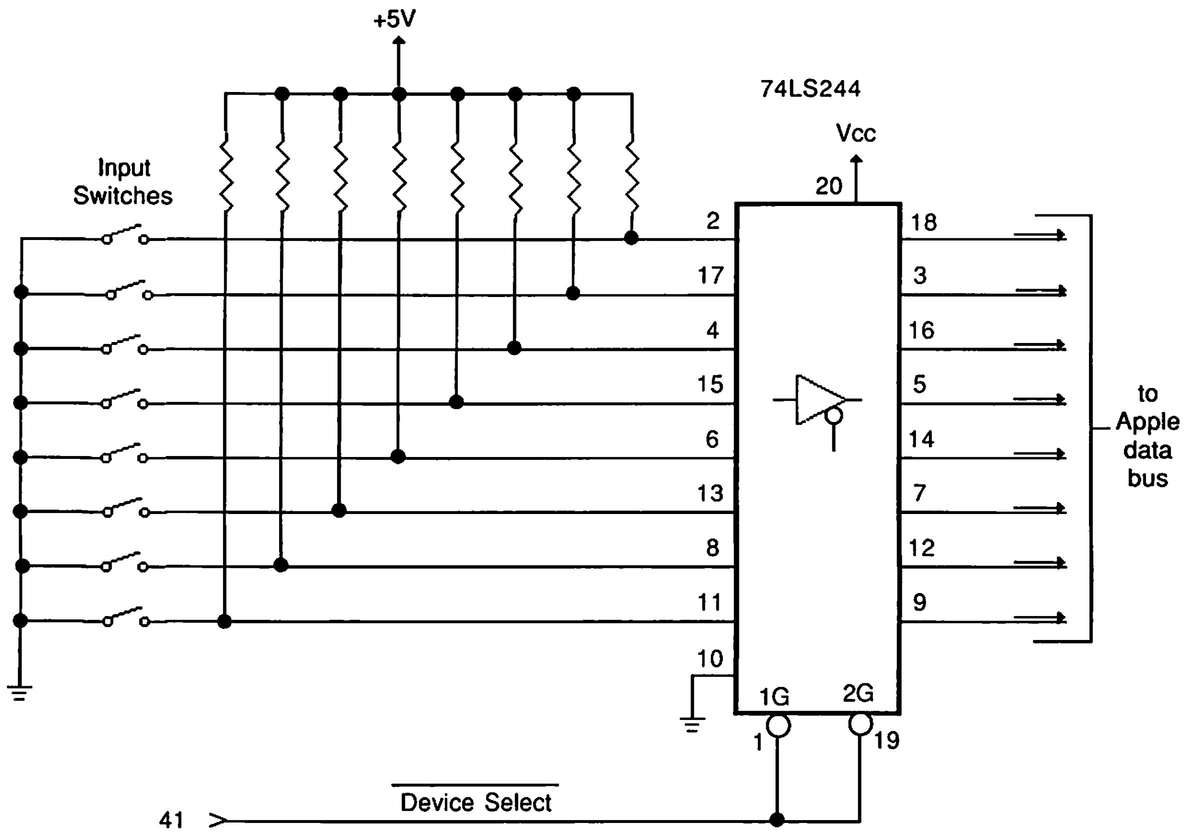


Figure 2-5
Using an octal buffer to input switch data

value of the data byte in memory location 768, you simply use the following line in your program:

```
LET N = PEEK (768)
```

After execution, the variable N will contain the value of the data byte in memory location 768. For better program documentation, you can use a label for the memory location, such as:

```
LET TEMP = 768
LET N = PEEK (TEMP)
```

Our interfacing job will be to make our input port "look like" a memory location to our computer.

A very simply input device is shown in figure 2-5. The 74LS244 octal buffer is used to connect the input data (from switches, keybord, etc.) to the Apple data bus. The 74LS244 is a tristate buffer, meaning that its outputs can be at a HIGH level, at a LOW level, or in the high-impedance state. Driving the 1G and 2G control inputs LOW enables the chip so that the buffers connect the inputs to the data bus. But when the 1G and 2G inputs are inactive (HIGH), all buffer outputs are in their high-impedance states and effectively disconnect the inputs from the data bus. This action is necessary so that the computer can use the data bus for other operations. In other words, the 1G and 2G inputs are only driven active at the instant the computer wants to read in new data. Otherwise, the 1G and 2G inputs are kept inactive.

In figure 2-5 we see that the switch inputs will only be connected to the data bus while $\overline{\text{DEVICE SELECT}}$ is LOW. So if we have the buffer mounted on a p-c card located in slot 4, all we have to do to get the switch data into our computer is to execute the following line of code:

```
LET N = PEEK (49344)
```

When this line is executed, $\overline{\text{DEVICE SELECT}}$ (pin 41) of slot 4 goes LOW for about 500 ns, during which time the computer expects to read a value from memory. That value can be used later in the program.

Figure 2-6 shows a simple way to build both an input port and an output port on the same p-c card and have them respond to different addresses. When $\overline{\text{DEVICE SELECT}}$ and A_0 are both LOW (address 49344 for slot 4), the input buffer will be enabled. But when

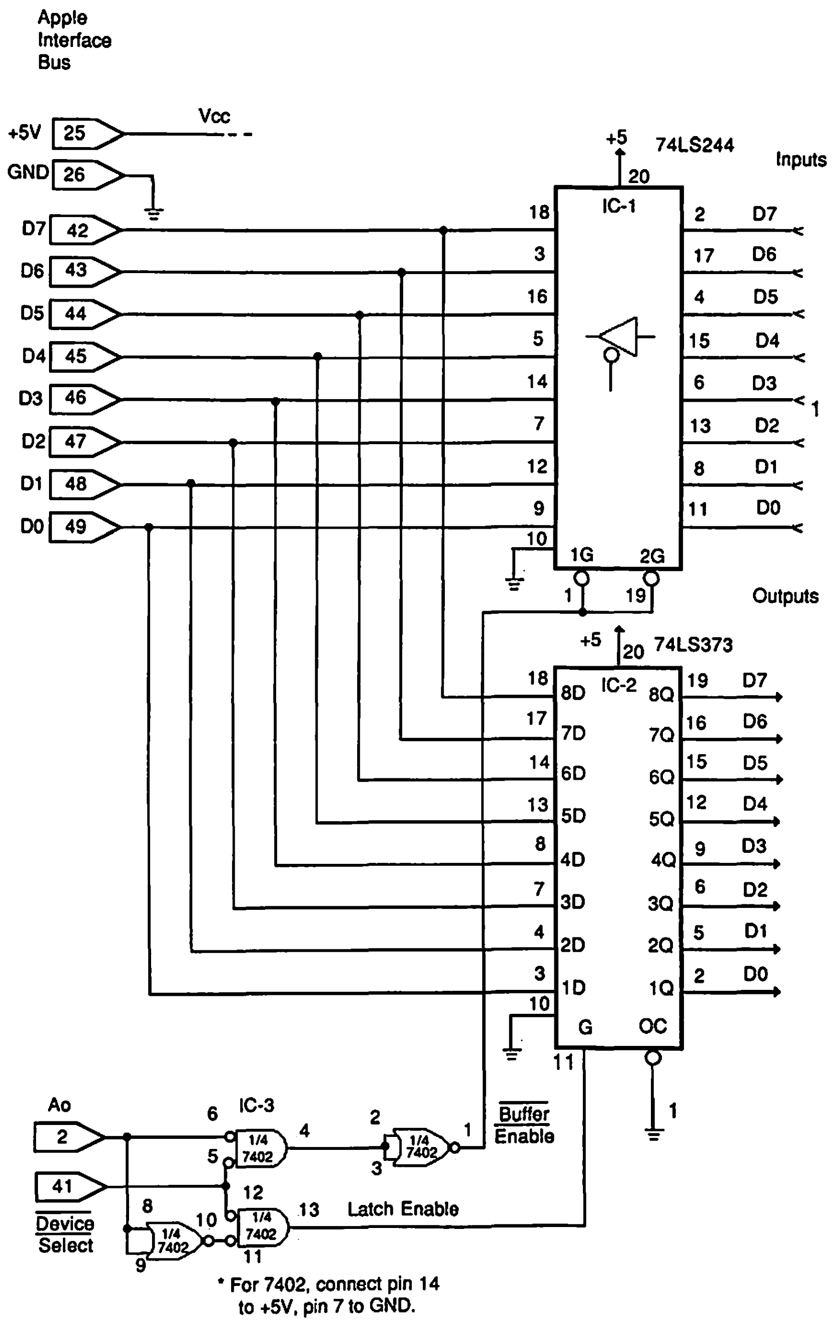


Figure 2-6
Complete parallel I/O port

DEVICE SELECT is LOW and A_0 is HIGH (address 49345), the output latch will be enabled. The following lines of program will input data from the switches and output the same data to the output latches:

```
10 LET BYTE = PEEK (49344)
20 POKE 49345, BYTE
```

By using an address decoder like the one shown in Figure 2-3, you can have up to 16 I/O ports in one slot in any combination of inputs and outputs.

2-4 CENTRONICS-TYPE PRINTER INTERFACE

One common application for a parallel I/O port is as the connection to a printer. And a commonly used type of interface is the Centronics™* type interface. Centronics printers were some of the first entries into the inexpensive printer market, and hence their interface circuitry became somewhat of a standard. Figure 2-7 shows how to convert our parallel I/O port of figure 2-6 into a Centronics-type interface. That is, by adding a chip we will be able to communicate with Centronics-type printers.

Two lines are used to read the printer status before sending a byte to the printer. The status signals is called BUSY and OUT-PAPER. If either of these two signals is active (HIGH), no data should be sent to the printer. But if both signals are inactive (LOW), the data byte to be sent to the printer should be latched into the output port and, simultaneously, a $\overline{\text{DATA STROBE}}$ signal should be sent to the printer.

Figure 2-7 shows that the status signals appear on input lines D7 and D6 of the input buffer (IC-1 of figure 2-6) and that the data output lines come from the latch. In addition, a 74121 one-shot generates a short (about 500 ns), LOW signal called $\overline{\text{DATA STROBE}}$ each time new data is latched into IC-2 of figure 2-6.

*Centronics™ is a trademark of Centronics Data Computer Corp.

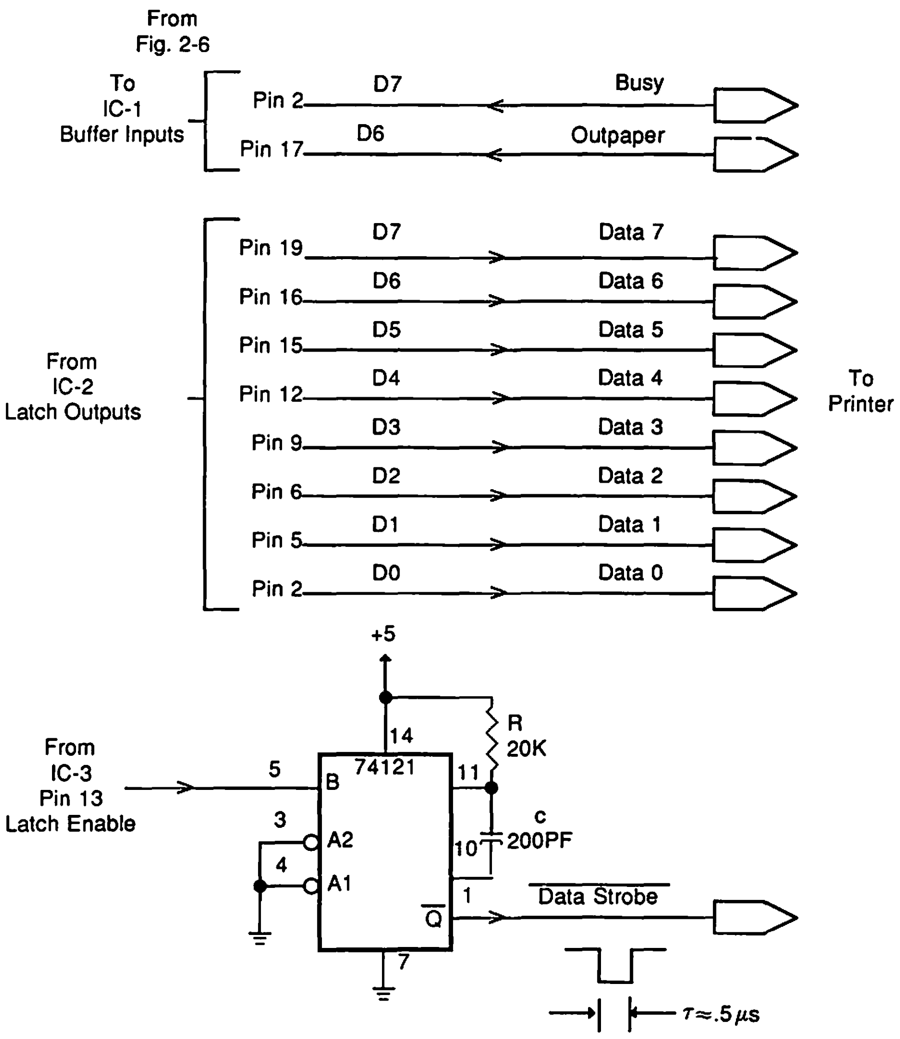


Figure 2-7
Centronics-type printer interface

Although a machine code routine is more efficient for this type of output, we can send bytes to the printer using BASIC. Figure 2-8 shows the listing of a program to send the contents of memory locations 768 through 800 to the printer. Note that the addresses of the FIRST and LAST bytes to be printed are defined in the

```
]LIST

2  REM  ** CENTRONICS PRINTER OUTPUT ROUTINE **
10  LET FIRST = 768
20  LET LAST = 800
30  LET BUFF = 49344
40  LET PRNTR = 49345
50  GOSUB 1000
60  END
1000  FOR MEM = FIRST TO LAST
1010 STS =  PEEK (BUFF)
1020  IF STS > 63 GOTO 1010
1030  LET BYTE =  PEEK (MEM)
1040  POKE PRNTR,BYTE
1050  NEXT MEM
1060  RETURN
```

Figure 2-8
Centronics printer output routine

initialization portion of the program, as are the addresses of the I/O ports. Lines 1010 and 1020 check to see that status bits D7 and D6 are both LOW before sending the next byte to the printer.

The output routine is used as a BASIC subroutine, which can be called whenever we want to send a message to the printer. Before calling the subroutine, of course, the user must define locations FIRST and LAST.

2-5 PROGRAMMABLE INTERFACE CHIPS

Since so much parallel I/O is done with buffers and latches, manufacturers of interface chips (ICs) make large-scale, programmable chips that can combine the functions of several chips. We will discuss using the MOS Technology 6520 Peripheral Interface Adapter (PIA), which is one of the simpler programmable chips. By learning how to use the PIA, you will get a good insight into using many other types of programmable interface chips. Figure 2-9 shows the pinouts of the 6520.

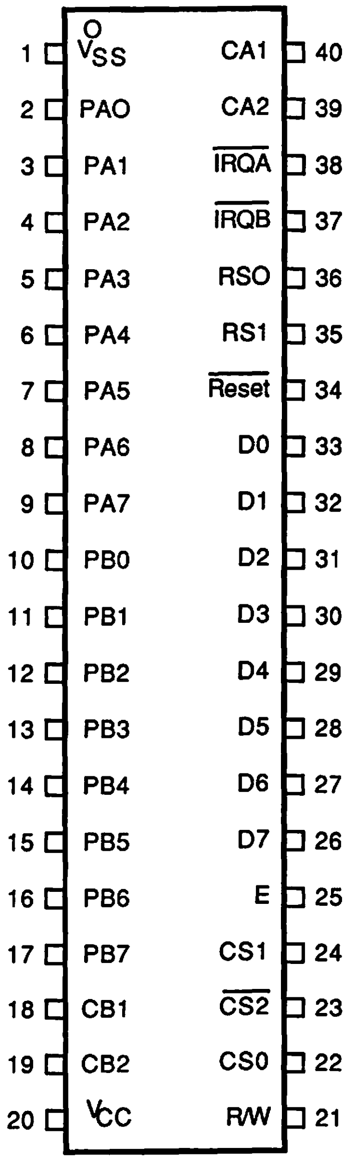


Figure 2-9
Pin assignments of the 6520 and 6820

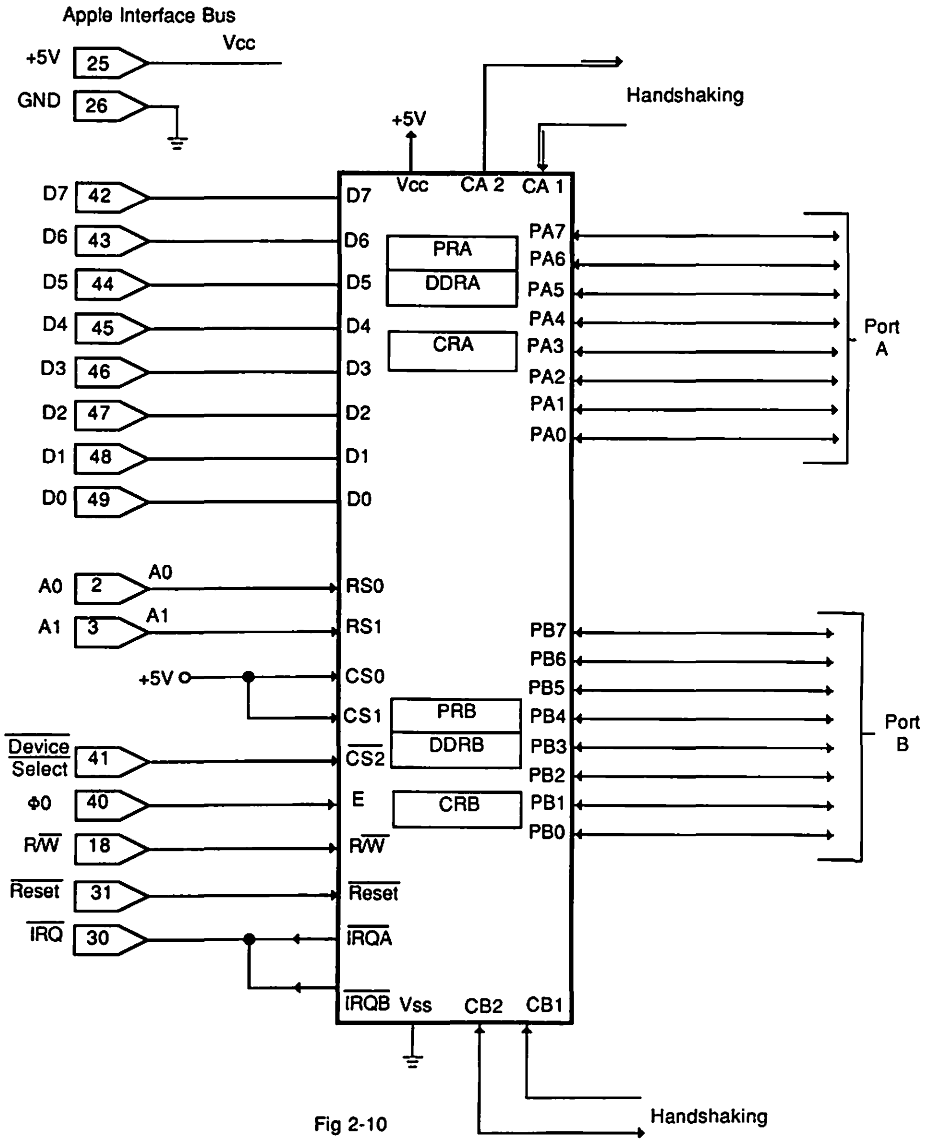


Figure 2-10
PIA connections to the interface bus

The 6520, which is just like the Motorola 6820 PIA, has two 8-bit ports, called port A and port B. Each bit line in each port can be configured to act as an input or an output line. In other words, you can program the PIA to act as one 8-bit input port (input buffer) and one 8-bit output port (latch). Or you can make a few of the lines of port A act as inputs and the remainder as outputs, in any combination. The same can be done with port B. In addition, some of the address decoding is done on board the chip, thereby reducing external gating. The single PIA chip can easily replace all three chips of figure 2-6.

Figure 2-10 shows the internal registers in the PIA. Note that each port has a *peripheral register* (PRA or PRB), which connects to the I/O pins; a *data direction register* (DDRA or DDRB), which tells each pin whether to act as an input or output; and a *control register* (CRA or CRB), which is used to select various options.

Before you can use the PIA, you must tell it how you want it to operate. That is, you must send it a few program bytes to configure it to your application. To program the PIA, first clear all registers to zero using the system $\overline{\text{RESET}}$. Second, you must send each data direction register a direction byte. Any bit in the direction register that is a zero (0) will make the corresponding bit line act as an input. Similarly, a one (1) in any position in the data direction register will make the corresponding data line act as an output pin. For example, sending the bit pattern 01101110 to the data direction register of port A will make bit lines PA7, PA4, and PA0 act as input lines and lines PA6, PA5, PA3, PA2, and PA1 act as latch outputs. See figure 2-11.

To place the direction byte into the data direction register of port A, you can use the POKE command, storing the direction byte into the memory location assigned to DDRA. Although there are six registers in the PIA, there are only four unique addresses to which the PIA responds. The peripheral register and the data direction register of port A both respond to the same address. Which register is actually chosen depends on the state of bit 2 in control register A. The same is true for port B. Table 2-2 shows how the register select (RS) inputs are used to select the various registers.

After writing the direction byte to the data direction register of port A, bit 2 of CRA must be set in order to gain access to peripheral register A, and hence to the I/O pins.

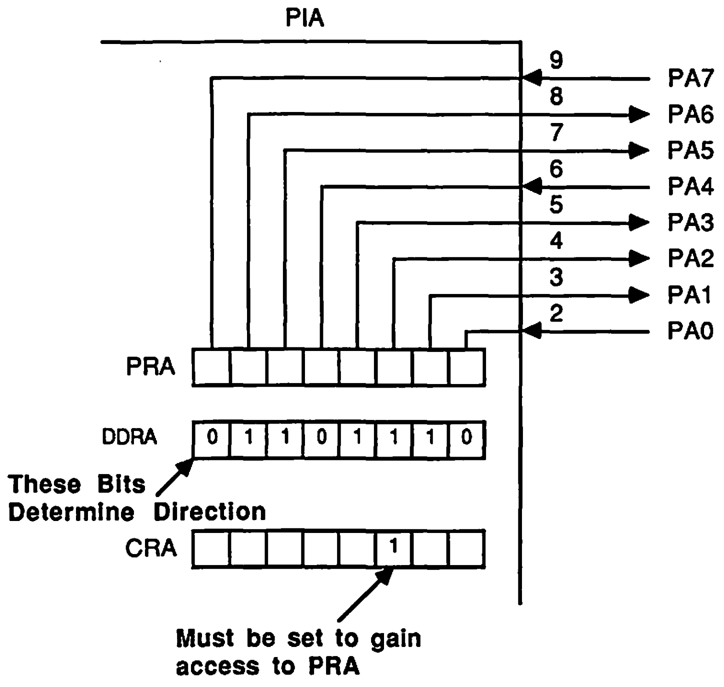

PIA
9
8
7
6
5
4
3
2
PA7
PA6
PA5
PA4
PA3
PA2
PA1
PA0
PRA
DDRA
0 1 1 0 1 1 1 0
These Bits Determine Direction
CRA
1
Must be set to gain access to PRA

Figure 2-11
PIA initialization example

TABLE 2-2
PIA Register Selection

		Control Register Bit		
RS1	*RS0*	*CRA-2*	*CRB-2*	*Location Selected*
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

X = Don't Care

As an example of an initialization sequence, let's suppose that we want to configure port A as an input port and port B as an output port. Let's also suppose that the PIA is mounted on a p-c card plugged into slot 4 and that it is wired as shown in figure 2-10. Here is the initialization routine:

```
30 POKE 49344, 0 : REM  MAKE PORT A AN INPUT PORT
40 POKE 49345, 4 : REM  SET BIT 2 OF CRA
50 POKE 49346, 255 : REM  MAKE PORT B AN OUTPUT PORT
60 POKE 49347, 4 : REM  SET BIT 2 OF CRB
```

Line 30 is not really necessary if the initialization follows a hardware reset. But the chip can be reinitialized at any time without a hardware reset. Keep in mind, however, that you must clear bit 2 of each control register before you can send a byte to its data direction register.

Once the chip is initialized, it acts just like an input buffer at 49344 and an output latch at 49346. You use the PEEK and POKE commands to access the data registers, just as if they were separate chips, like those in figure 2-6.

The PIA can also generate interrupt requests and perform handshaking with peripheral devices. However, we will postpone any application of these features until after we have discussed machine code programming.

CHAPTER 3

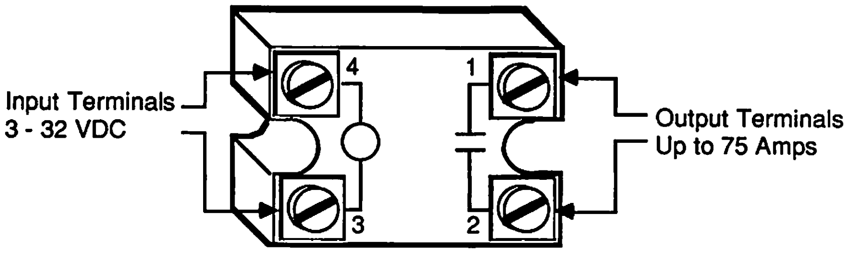
Interfacing Power Control Devices

Besides being able to tie our computer to TTL loads, we can control ac power loads, such as lamps, motors, and heaters. However, we must use some interface circuitry that will allow us to turn the high-voltage ac loads on and off without damaging the low-voltage circuits in our computer. This chapter will discuss using commercially available control circuitry as well as less expensive circuits you can build yourself. We will also discuss using the computer as a simple, programmable controller, like those used in industrial automation applications such as assembly, welding, food packaging, and paint spraying.

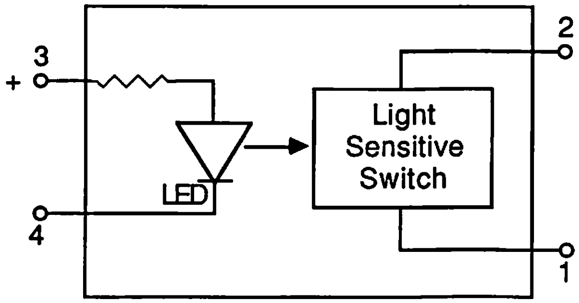
3-1 SOLID-STATE RELAYS

Electromagnetic relays have been used for years to enable low-voltage circuitry to control the turn on and turn off of high-voltage or high-power loads. But magnetic relays have many undesirable characteristics, such as contact bounce, sticking of contacts, and sensitivity to vibration. Solid-state relays have overcome most of these bad features and still allow control of high-power loads. A typical solid-state relay and its equivalent circuit are shown in figure 3-1. As you can see, the input control voltage can be anywhere from 3 volts to 32 volts dc. When the control voltage is applied, the solid-state relay is in its ON state. When ON, the output circuit effectively acts like a closed switch and can carry several amperes, depending on the model number. When in its OFF state, the solid-state relay can withstand ac line voltages of 240 volts without damage.

Figure 3-2 shows a simple interface between a computer and an ac load (lamp). Note that the control input terminal of the solid-state relay connects to the $\overline{Q}$ output of a latch. The latch is the same as was discussed in chapter 2. When the Q output goes HIGH ($\overline{Q}$ goes LOW), the solid-state relay turns on, completing the circuit between the lamp and the ac line. The lamp current can be quite high, perhaps 10 amps or more, but the latch only has to output a control current of about 4 ma to activate the relay. In addition, no high voltage is ever coupled back to the latch or computer circuitry. The input section of the solid-state relay is optically coupled to the



(A) Physical Package Less Than 2" x 3"



(B) Equivalent Circuit

Figure 3-1
Solid-state relay

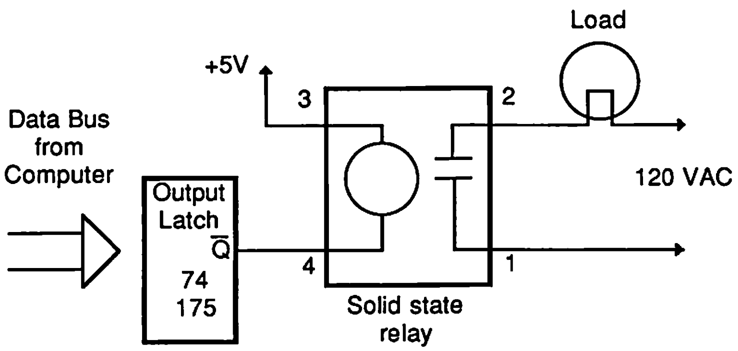


Figure 3-2
Output latch drives SSR, which controls load current

output, so there is never any direct electrical connection between its input and output sections. Although a separate solid-state relay is needed for each load that is being controlled, you can control up to 8 solid-state relays with a single octal latch.

3-2 USING TRIACS FOR POWER CONTROL

The solid-state relay is an excellent power control device, although it is somewhat expensive. You can build a less expensive control circuit using a bidirectional ac switch, commonly called a TRIAC. figure 3-3A shows a TRIAC being used in a circuit to control the current through a lamp. The TRIAC's three terminals are labeled T1, T2, and Gate. Without going into a lot of theory about the TRIAC, let's see how it is used in this circuit.

In figure 3-3A, switch S1 is open so the Gate is returned to T1 through resistor R2. In this condition, the TRIAC is off and acts like an open switch so no load current flows. In figure 3-3B, switch S1 is closed. With S1 closed, the Gate is pulled toward T2 on the next half cycle of the line voltage. When a voltage is applied between the Gate and T1, the TRIAC fires (turns on suddenly). It continues to conduct until the ac line voltage falls to zero. If S1 remains closed, the TRIAC fires again on the next half cycle of the line voltage. The firing and turning off continues until S1 is opened. As far as the load is concerned, the TRIAC simply acts as if it were a closed switch. But when S1 is opened, no voltage is applied between the Gate and T1. Therefore, as soon as the ac line voltage falls to zero the next time, the TRIAC shuts off. The waveforms of figure 3-3C illustrate the operation of the TRIAC.

Now let's see how we can tie the TRIAC to our computer. Figure 3-4 shows how S1 and R1 of figure 3-3 are replaced by an optoisolator. The optoisolator, a Clairex CLM8000, is a small, 4-terminal device containing an LED and a photoconductive cell. The photoconductive cell has a high resistance (500K ohms) when dark but becomes a low resistance (about 400 ohms) when the LED shines on it. So to turn on the TRIAC from our computer, we simply tie the cathode end of the LED to the $\overline{Q}$ output of our latch. When we want to fire the TRIAC, we output a byte to the latch, which will

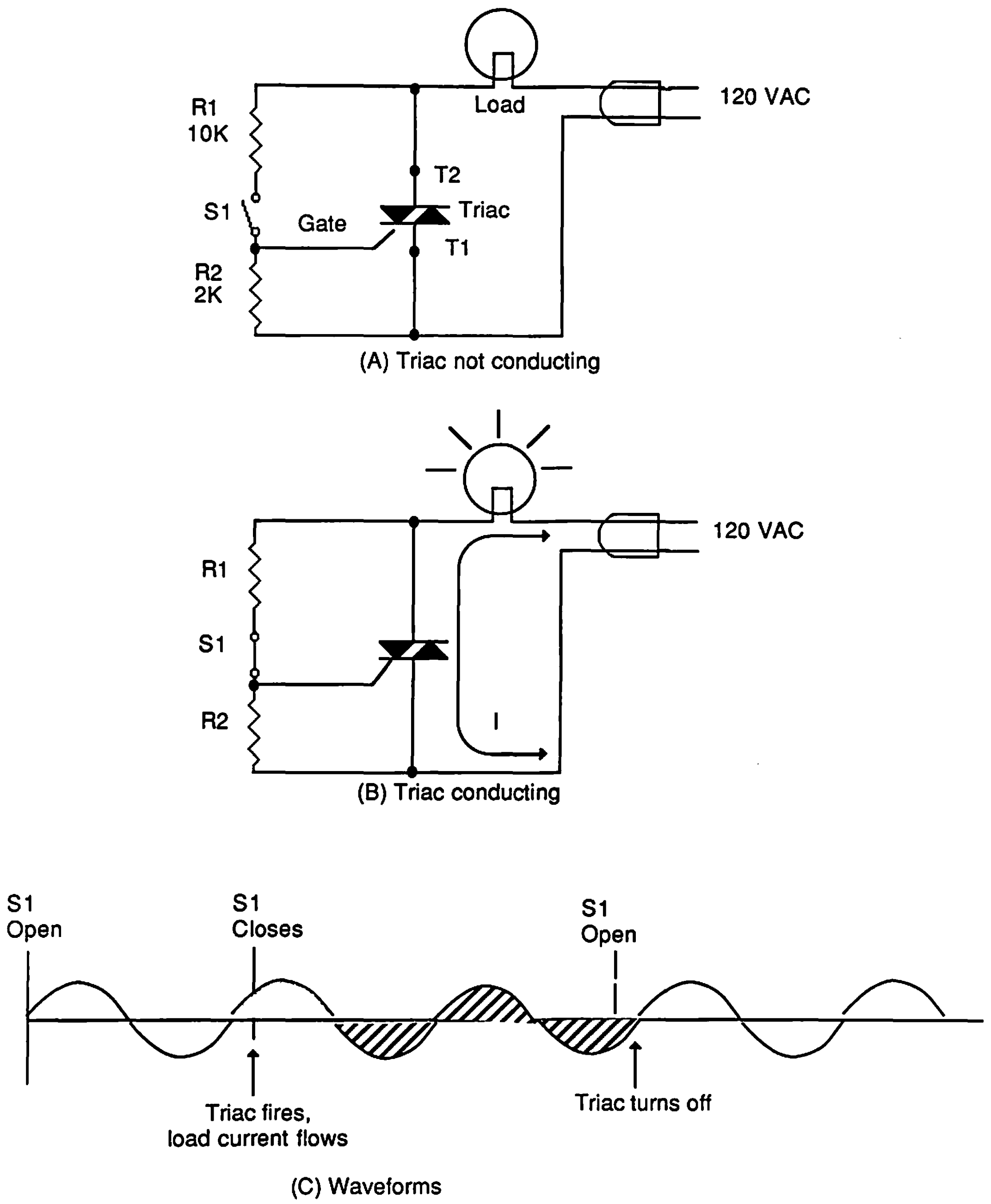


Figure 3-3
Controlling load current with a triac

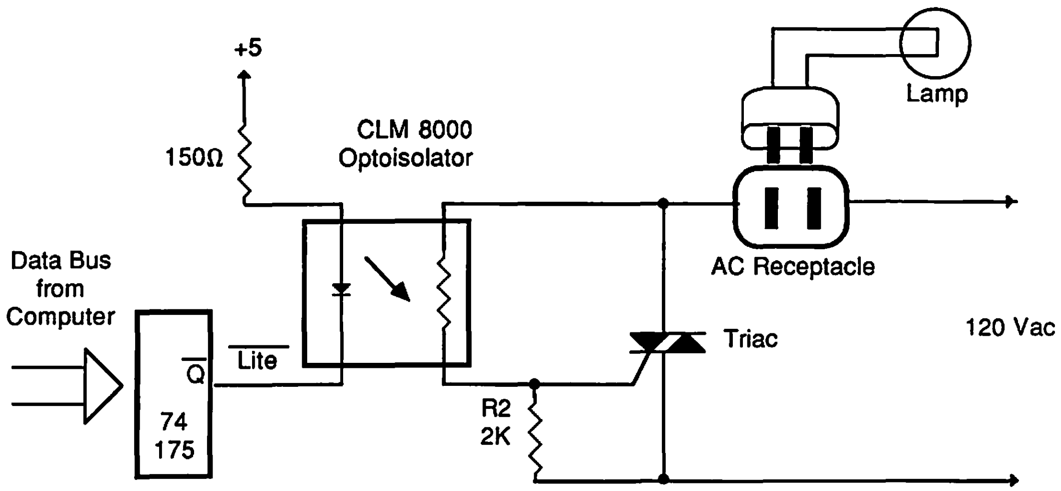


Figure 3-4
Interfacing a triac to a latch with an optoisolator

make the signal $\overline{\text{LITE}}$ go active (LOW). The TRIAC will remain on until we make $\overline{\text{LITE}}$ go inactive (HIGH) again.

Figure 3-5 shows the listing of a program to make the computer control the duration of the ON time of a lamp, say for use as an exposure timer in a darkroom. Of course, the same program and circuitry could be used to control weld time, paint-spray time, and so on. Lines 60-80 control the time duration. Line 70 causes a time delay of about 1 second each time it is executed. To change the increments of the delay to 0.1 second, simply change line 70 to : FOR T = 1 TO 70: NEXT T.

Another inexpensive and easy-to-use device is the Motorola MOC3010 optocoupler, shown in figure 3-6. The MOC3010 has a built-in TRIAC intended for 120-vac operation, but it can only carry a load current of 100 ma. To use it in a low, power-high voltage application, simply connect the TRIAC to the load as in figure 3-4. Then drive the LED from the latch output, as before. Note that there is no gate connection on the MOC3010. If you want to use the MOC3010 to drive a higher power circuit, connect it as shown in figure 3-7. Notice that the heavy line current flows through a power TRIAC, which is turned on when the small TRIAC in the MOC3010 fires.

```
]LIST

10   REM   *** TIMER CONTROL ***
20   LET TIMER = 49345
25   HOME
30   PRINT "ENTER TIME DURATION IN SECS"
40   PRINT "TIMING WILL START WHEN YOU PRESS RETURN"
45   PRINT
50   INPUT "DURATION => ";D
55   POKE TIMER,1
60   FOR S = 1 TO D
70   FOR T = 1 TO 700: NEXT T
80   NEXT S
90   POKE TIMER,0
100  GOTO 25
```

Figure 3-5
Timer control listing

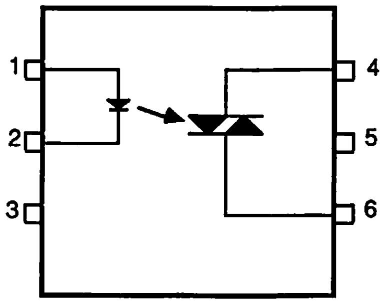


Figure 3-6
MOC3010 optocoupler

One other point to notice about figure 3-7 is the use of resistor R3 and capacitor C1 across the power triac. These components are necessary to ensure that the TRIAC is properly turned off when working into an inductive load, such as the motor.

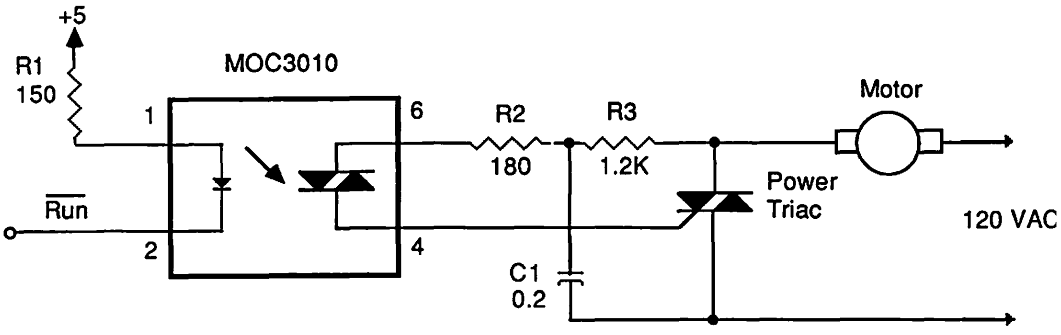


Figure 3-7
Using the MOC3010 to interface to a high-current load

3-3 PROGRAMMABLE CONTROLLERS

Programmable controllers (PCs) are used in industry to control any process that can be automated, such as packaging, assembly, material handling, measuring, painting, and welding. The decisions as to what is to be done are essentially made by a computer, which

1. Scans several inputs to determine their states.
2. Performs logic operations on the inputs to decide what is to be done next.
3. Turns outputs on or off accordingly.

Basically, the PC simplifies the hard-wired logic used previously in automation. By using a PC, less wiring is needed, changes are easier to make, and more complex tasks can be performed. A method of diagramming an automatic control system called a ladder diagram has been in use for many years and is commonly used with PCs. A typical ladder diagram is shown in figure 3-8. The vertical lines at each side are connected to the ac power mains, 120 vac in this case. Then each individual circuit connected to the mains are shown as individual rungs on a ladder. L1 represents the load for rung 1, L2 is the load for rung 2, and so on. Normally, only one load is allowed for each rung. The switches in series with each load determine whether the load is turned on or off.

Let's examine the ladder diagram more carefully. In order for power to be applied to L1, either switches S1 and S2 or switches S3 and S2 must be closed. The following simple logic equation describes the rung:

L1 = S1 AND S2 OR S3 AND S2

Our programmable controller first scans input switches S1, S2, and S3, then performs the logic, and finally turns L1 on or off accordingly. The circle around L1 simply means that it is an output. In reality, it may be a lamp, a motor, a heater, or any type of power load. To turn L1 on, our computer simply has to set a bit in a latch,

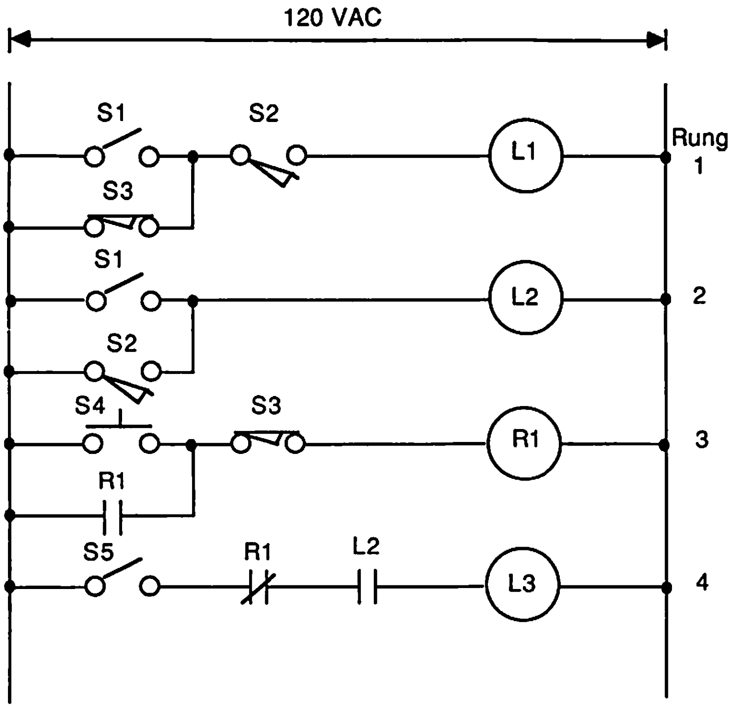


Figure 3-8
Ladder diagram

which drives an optoisolator, which turns on a triac connected to the load.

The inputs to the computer can be a variety of switches. For example, S1 is an ordinary toggle switch, possibly a starter switch. S2 is shown as a normally open limit switch, whereas S3 is a normally closed limit switch. S4 in rung 3 is a momentary push button. The same switch may be examined more than once by the computer to control more than one rung.

The output of rung 3, called R1, represents a control relay. Control relays were common with hard-wired logic, but the relay may not actually exist in the PC system. A relay is shown in the ladder diagram to help you understand the operation, but no real load R1 has to exist. R1 is simply a variable that the computer keeps track of, whose state will be used to solve other rungs. R1 is shown as a latching relay. That is, if S3 is closed and someone presses S4, R1 will energize. Contacts of R1 bridge S4, so even when S4 is released, R1 remains in the ON state. To make R1 drop out, S3 has to be opened. R1 also has a set of contacts in rung 4. The slash line through the contacts indicates normally closed contacts, that is, contacts that will be closed when the relay is deenergized.

Now let's see how we can use the Apple computer to solve the rung equations. We can perform logic operations in BASIC with the Apple. For example, if we execute the equation L1 = S1 AND S2 OR S3 AND S2, the variable L1 will be assigned the value of 1 either if S1 and S2 are nonzero or if S3 and S2 are nonzero. Otherwise, L1 will be zero. The result of a logic operation will always be either a 1 or a 0. Figure 3-9 shows the listing of a program that can be used to control the system whose ladder diagram is shown in figure 3-8. Although the input and output portions of the program are not shown here, before the equations can be solved, the computer must have values for all inputs. Study the listing until you see the correspondence between the listing and the ladder diagram.

Notice that an output can also be examined to determine the conditions for another rung. For example, L2 is examined in rung 4. If L2 is on (true), contacts L2 in rung 4 will be closed. Our control program can also contain subroutines, for example to generate time delays that can be used to control weld time.

To help crystallize our understanding of using the Apple as a tiny, programmable controller, let's study a typical application, the

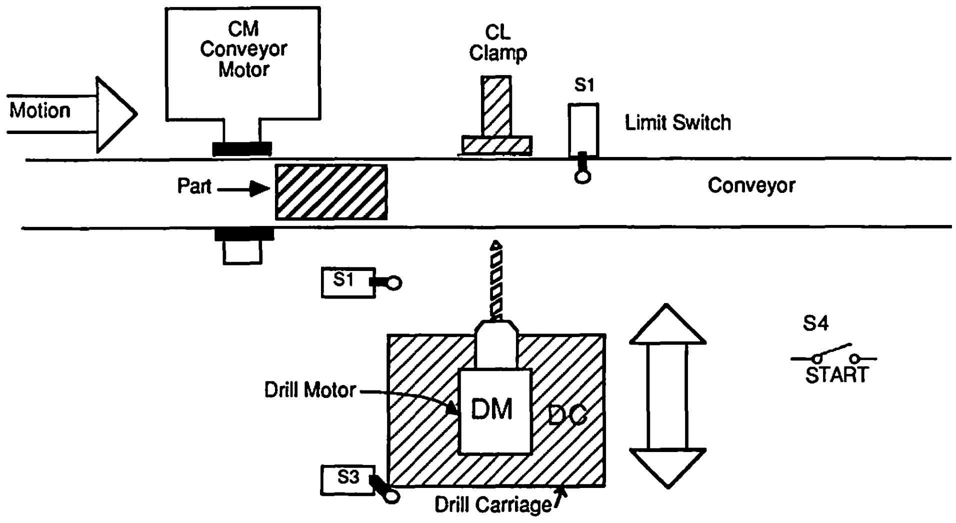
```
]LIST

1  REM  *********************************
2  REM  * CONTROL PROGRAM FOR FIG. 3-8 *
3  REM  *********************************
10  REM  ***  DO INITIALIZATION  ***
100  REM  ***  SCAN INPUT SWITCHES  ***
199  REM  ***  BUILD RUNGS  ***
200 L1 = S1 AND S2 OR S3 AND S2
210 L2 = S1 OR S2
220 R1 = S4 AND S3 OR R1 AND S3
230 L3 = S5 AND  NOT R1 AND L2
299  REM   ***  DO OUTPUT  ***
399  GOTO 100: REM  * LOOP AGAIN *
```

Figure 3-9
Control program for circuit of figure 3-8

Automatic Driller of figure 3-10. In this appliction, one hole is automatically drilled in each part moving along a conveyor. Here is how it works: The conveyor motor is turned on and parts are placed on the conveyor line. When a part hits limit switch S1, the conveyor motor (CM) is turned off. A clamp (CL) is energized, which holds the part for drilling. Then a motor is energized to move the drill carriage forward (DCF). At the same time, the drill motor (DM) is also turned on, and drilling begins. Drilling continues until the carriage hits limit switch S2, indicating that the desired depth has been reached. The DCF motor is turned off, and another motor DCR is turned on to reverse the direction of the drill carriage. When the drill carriage is fully retracted, it hits limit switch S3, causing DCR and DM to turn off. The clamp (CL) is released, and the conveyor motor (CM) is turned on again to move out the drilled part and bring in another part.

The laddder diagram for the Automatic Driller is shown in figure 3-11, and the program listing is shown in figure 3-12. We assume that we have an input buffer and an output latch in slot 5, as in chapter 2. In the actual system wiring, *the switches do not connect to the ac line.* They are simply single-input switches, like those in figure 2-5, and they are periodically examined to see if they are open or closed. Also, in the actual system, the loads CM, CL, and so



INPUTS

- S1 Normally open limit switch, Closes when part detected
- S2 Normally open limit switch, Closes when drill is at depth
- S3 Normally closed limit switch, Opens when carriage is retracted
- S4 Start switch

OUTPUTS

- L1 Conveyor Motor
- L2 Clamp
- L3 Drill Carriage Forward
- L4 Drill Carriage Reverse
- L5 Drill Motor

Figure 3-10
Automatic driller

on, *do not connect directly to switches S1,* and so on. The loads are connected to the Apple through optoisolators, and they are turned on and off by triacs, as shown in figure 3-4.

Referring to the program listing, we first initialize by telling the computer the address we want assigned to our input and output ports. Line 100 begins the input section. We get the switch data in line 120. The switches are wired to the input buffer in slot 5 such that S1 is brought in on data line 0, S2 on data line 1, S3 on data line 2, and S4 on data line 3. The value obtained for V in line 120 is a decimal number representing the binary combination of switches S1 through S4. But we must separate the individual bits representing the open or closed switches. So lines 130 through 180 perform a

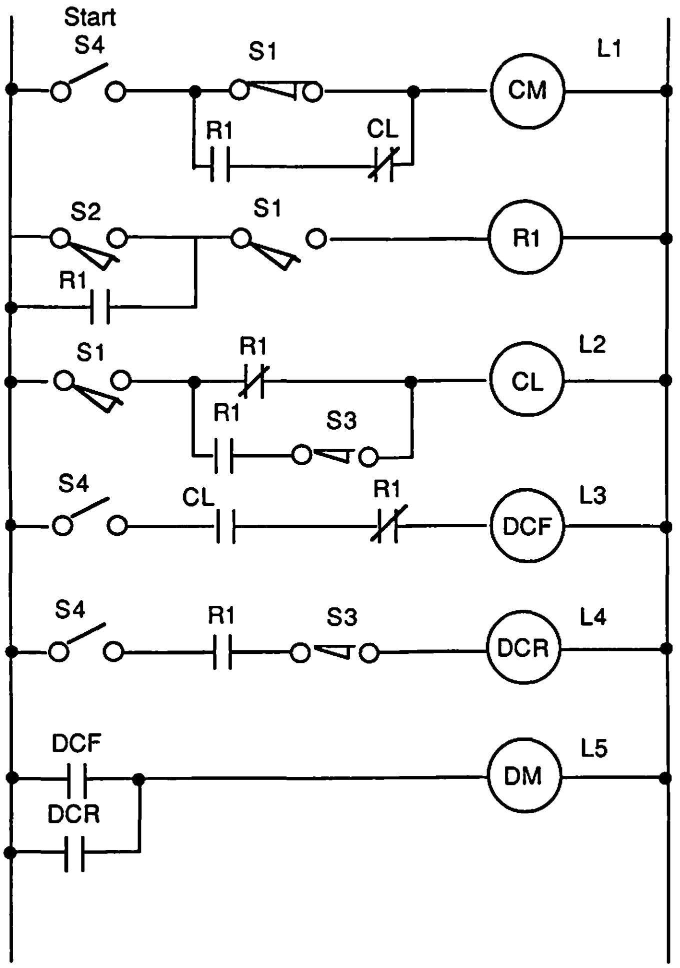


Figure 3-11
Ladder diagram for automatic driller

decimal-to-binary conversion and place values into an array in such a way that S(1) will be 0 if open and 1 if closed. A similar conversion is performed for the other switches.

Line 200 of the program listing begins construction of the rungs. Then the output routine begins at line 300. First, we assemble an output value called BYTE by doing a binary-to-decimal conversion of all load bits. We then POKE this value to the output port called DRILLER, which latches outputs HIGH or LOW, de-

```
]LIST

1  REM  ******************************
2  REM  *      AUTOMATIC DRILLER     *
3  REM  *  (C) 1983  BY  J. OLEKSY  *
4  REM  ******************************
10  LET SWITCHES = 49344
20  LET DRILLER = 49345
30  DIM S(8)
100  REM   ***  GET SWITCH DATA AND PUT INTO ARRAY ***
120  LET V =  PEEK (SWITCHES)
130  FOR N = 1 TO 8
140 Q = V / 2
150 Q1 =  INT (Q)
160 S(N) = (Q - Q1) * 2
170 V = Q1
180  NEXT N
199  REM  *** BUILD RUNGS ***
200 L1 = S(4) AND  NOT S(1) OR S(4) AND R1 AND  NOT L2
210 R1 = S(2) AND S(1) OR R1 AND S(1)
220 L2 = S(1) AND  NOT R1 OR S(1) AND R1 AND S(3)
230 L3 = S(4) AND L2 AND  NOT R1
240 L4 = S(4) AND R1 AND S(3)
250 L5 = L3 OR L4
299  REM  *** DO OUTPUT ***
300 BYTE = L8 * 128 + L7 * 64 + L6 * 32 + L5 * 16 + L4 * 8 + L3 * 4 + L2 * 2 + L
1 * 1
310  POKE DRILLER,BYTE
320  GOTO 100: REM  * LOOP AGAIN *
```

Figure 3-12
Automatic driller listing

pending on which loads should be energized. Finally, we jump back to line 100 to begin another input routine. The process repeats itself over and over.

Since we have only one input port and one output port, we can handle up to 8 inputs and 8 outputs. If more inputs and outputs are required, we can build additional input buffers and output latches, giving them each a different address. Each time we read from an input port, we must do steps similar to those in lines 120 through 180 to assign a 1 or 0 bit to each switch. And, each time we do an output, we must assemble a new output bit pattern like we did in line 300. Since a considerable amount of time is spent in these conversions, if a large number of I/O points is needed, it would be better to use a language other than BASIC. In a later chapter we will see how to use machine code routines for fast I/O. But if you only need to control a few I/O points, or you do not need high speed, the techniques shown in figure 3-12 will work sufficiently. The BASIC control program is easy to read and easy to modify.

CHAPTER 4

Interfacing to Analog Inputs

So far, we have discussed transferring digital information into and out of the computer. But there are many applications where the information we handle is *analog* rather than digital. That is, the information we deal with is a parameter like temperature, pressure, velocity, or some other physical quantity. The value of that parameter may be anywhere within a continuous range of values from some minimum to some maximum value.

Whenever we want to input such a physical parameter, we normally use a transducer to change the quantity into a voltage. Then we feed the voltage to an *analog-to-digital* (A/D) *converter,* which changes the specific value of voltage into a digital value (binary number) that the computer can input through a parallel port (input buffer).

Similarly, whenever the computer must output an analog voltage, for example, to control the speed of a motor, we do a *digital-to-analog* (D/A) conversion. That is, the digital value representing the output voltage is fed by the computer to an output port (latch). The output port feeds a D/A converter, which changes the digital number into a corresponding dc voltage.

In this chapter, we will examine a few simple A/D converters. We will also look at a few typical applications, such as monitoring temperature, pressure, and liquid level.

4-1 SINGLE-INPUT A/D CONVERSION

There are a variety of off-the-shelf devices that can be used to convert an analog voltage into a digital value. From the user's point of view, two of the most important characteristics of these devices are *conversion time* (how long it takes to convert an analog voltage into a digital number) and *resolution* (how many bits are in the digital number). We will first look at an ADC0804, which is a simple, inexpensive, 8-bit A/D converter with a conversion time of 100 microseconds.

The 0804, shown in figure 4-1, has an on-board clock to keep its internal operation going. All you have to do is connect an external R-C network (R1-C1) to pins 4 and 19. When connected as

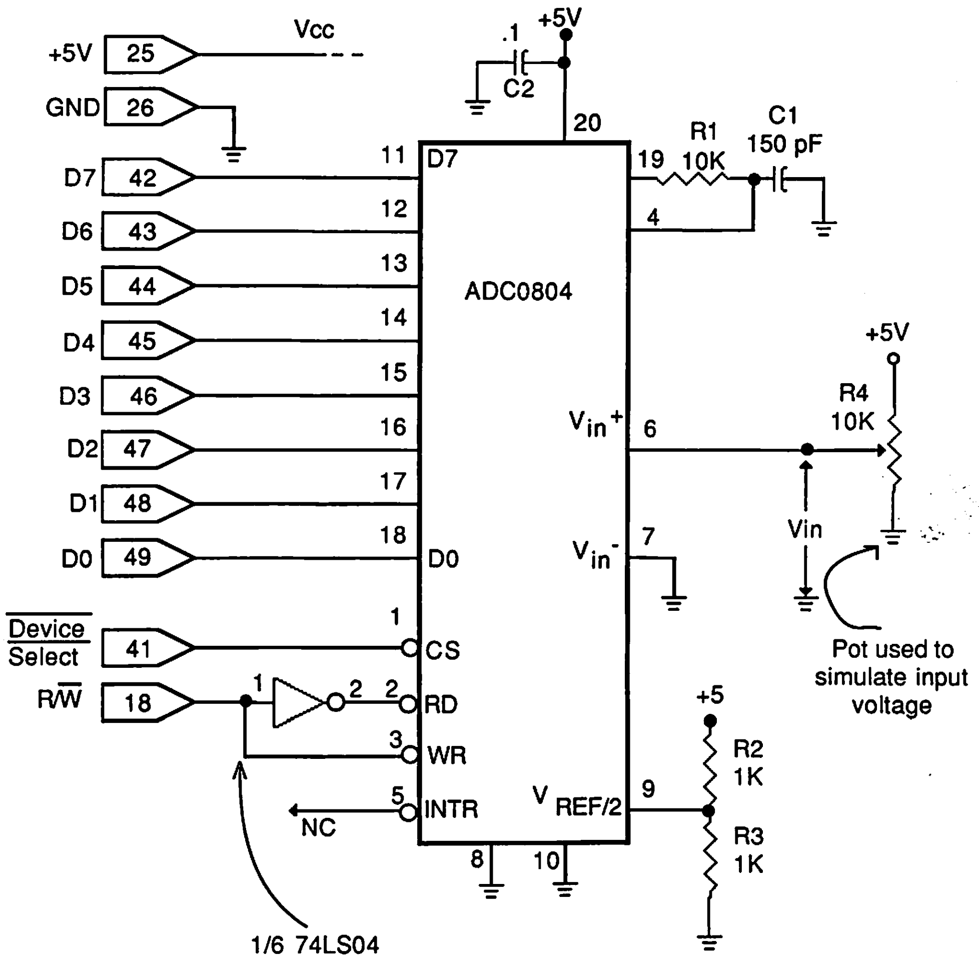


Figure 4-1
Single-input A/D converter

shown, the 0804 will convert a dc input voltage in the range of 0 to +5 volts into a digital value from 00 to FF hex (0-255 decimal).

Since the 0804 is an 8-bit device, it effectively divides the maximum voltage (+5v) by 256, and outputs the closest approximation of that voltage (within 1 part in 256). A 10-bit A/D will give the closest approximation within 1 part in 1024, and so on. So you choose an A/D with 8, 10, 12, or 14 bits resolution, depending on the degree of accuracy you require.

Referring back to figure 4-1, when you want to start a conversion, you signal the 0804 to begin by driving its chip select ($\overline{\mathrm{CS}}$) and write ($\overline{\mathrm{WR}}$) inputs active (LOW). After about 100 µs, the digital representation of the analog voltage at its +V input can be read on its data outputs by driving $\overline{\mathrm{CS}}$ and $\overline{\mathrm{RD}}$ active (LOW). When $\overline{\mathrm{CS}}$ is inactive, its data outputs are in the high-impedance state (tristate), so the data output lines can be connected directly to the computer's data bus. No buffers are needed.

Potentiometer R4 is used to obtain a variable dc input voltage for test purposes. R4 does not have to be mounted on the pc card inside the computer. Simply connect a pair of wires to +Vin and GND, and lead them out of the rear of the computer. The lead from +Vin can then be connected to the test pot or to any source of dc input voltage you wish to measure. Do not let +Vin exceed +5v.

National Semiconductor Corporation recommends that a 2.5-volt reference diode be used at input pin 9 for greater accuracy. But the simple voltage divider shown works reasonably well if you do not need a high degree of precision.

Let's assume that the 0804 is wired to a card plugged into slot 4 of the Apple. The following program segment will cause a number corresponding to the voltage at the +V input to be displayed on the computer screen:

```
10 POKE 49344, 0
20 LET V = PEEK(49344)
30 PRINT V
```

Line 10 starts the conversion. Line 20 reads in the value.

Although the chip requires approximately 100 µs between the start conversion signal and the time the conversion is complete, BASIC is slow enough that we do not need any time delays between the two instructions. In a later chapter we will see how to use the interrupt ($\overline{\mathrm{INTR}}$) output of the chip as a *conversion complete* signal. We will use that signal when running machine code programs, which execute much faster than BASIC.

The number displayed on the screen can easily be modified to represent the voltage by multiplying it by a scaling factor, such as:

```
30 VOLTS = V * 5/255
40 PRINT VOLTS
```

```
]LIST

5  REM   ***   SIMPLE DVM DEMO   ***
10  HOME
20  VTAB 10: HTAB 22
30  PRINT "VOLTS"
40  POKE 49344,0
50  LET V =  PEEK (49344)
60 V1 = V * 5 / 255
70  IF V1 < 1 THEN  GOSUB 300
80  IF V1 =  > 1 THEN  GOSUB 200
90  VTAB 10: HTAB 15
100  PRINT VOLTS;"   "
110  FOR D = 1 TO 125: NEXT D
120  IF VOLTS > 4.99 GOTO 10
130  GOTO 40
200 VOLTS = ( INT ((V1 * 100) + .5)) / 10
210  RETURN
300 VOLTS = ( INT ((V1 * 1000) + .5)) / 10
310  RETURN
```

Figure 4-2
Simple DVM demo

We can use the Apple as a simple digital voltmeter by using the 0804 in slot 4 and running the program listed in figure 4-2.

The program causes the voltage at the slider of pot R4 to be read in and displayed in the middle of the screen. The subroutines at lines 200 and 300 round off the voltage to three significant figures or less before displaying its value. If less than 3 digits are being displayed, three blank spaces are printed after the voltage value to erase any possible carry over from a previous display. The program is in a continuous loop from line 130 to line 40, but if the input voltage becomes greater than 4.99 volts, line 120 causes a jump back to line 10, which clears the screen each time it is executed. The result is that the value 5 VOLTS flashes on and off, acting as an over-range indicator.

Of course, our simple DVM can only measure positive dc voltages between 0 and +5v. If you want to measure higher voltages or negative or ac voltages, you have to modify the input to the 0804, as is done with ordinary voltmeters. For example, to measure an ac voltage, first rectify the ac, then feed the rectified and filtered dc to the 0804. You can use software to do any required scaling, for instance, to display rms rather than peak values. To measure a negative voltage, feed the input analog voltage through an op amp wired as an inverter, then pass it on to the 0804.

4-2 USING GRAPHICS TO DISPLAY PHYSICAL QUANTITIES

Our digital voltmeter displays numbers corresponding to a measured voltage. But sometimes it is more useful to graphically display the value of some physical parameter. Figure 4-3 shows the listing of a program that will take the digital value of the input voltage from our A/D converter and display it as a vertical bar graph. The length of the bar is proportional to the magnitude of the input voltage. Although low-resolution graphics are used here for simplicity, high-resolution graphics can also be used.

If you wire the A/D converter as in figure 4-1 and bring out the input leads and connect them to a pot, you will see that the height of the vertical bar is dependent on the pot setting. Of course, rather than use a pot, the input voltage can be generated by a temperature sensor, a light sensor, a strain gage, or whatever you choose. You simply have to modify the voltage generating source to give a dc voltage between 0 and +5 volts. Then you can display that parameter as a bar graph.

In the next section, we will discuss how to monitor several analog inputs with a single A/D converter. By using a different color for each of the bar graphs, an interesting display of several parameters can be viewed simultaneously.

Let's look at an application using an A/D converter to control a graphics display. Suppose we want to build a display system to be used in a chemical plant where fluids are stored in tanks. The display should be built so that the plant supervisor can view the fluid level in several tanks simultaneously. Rather than simply connecting several meters on a wall, we decide to use a graphics

```
]LIST

100  REM  *** VERT BAR GRAPH ***
110  REM  * NEED A/D CONVERTER *
120  REM  *     IN SLOT 4       *
130  GR
140  COLOR= 7
150  POKE 49344,0
160 V =  PEEK (49344)
170 Y = (V / 255) * ( - 39) + 39
180 Y =  INT (Y)
190  VLIN 39,Y AT 25
200  COLOR= 0
210  IF Y <  = 0 GOTO 140
220  VLIN Y - 1,0 AT 25
230  GOTO 140
```

Figure 4-3
Vertical bar graph of dc input voltage

display showing each tank and its contents. We can use different colors for each different chemical for easy identification.

All we need is a level sensor in each tank that gives a voltage corresponding to the level of the fluid inside. We can scale the voltage up or down to get a 0-5v range. Then we feed the voltage to our A/D converter. We display the voltage graphically, in a manner similar to the bar graph mentioned earlier.

Figure 4-4 shows the listing of a program that constructs the shape of a tank then fills it with the color of fluid specified by the user. As the fluid level changes, the tank can be seen to fill and empty accordingly. Although only one tank is shown, the same idea can be used to display several tanks. In addition, the program can be modified to display fill and drain pipes to show the complete system.

The program begins with an initialization subroutine at line 250, which prompts the user to input some dimensions. The user specifies the position of the left side, the right side, as well as the top and bottom of the tank. Next the user specifies the colors to be used for the tank and for the fluid.

On returning from the user input subroutine, the low-resolution graphics mode is activated, and construction of the tank

```
]LIST

10  REM  ***********************
20  REM  * FLUID LEVEL MONITOR *
30  REM  *   J. OLEKSY 1984    *
40  REM  ***********************
49  REM
50  REM  *** INITIALIZE ***
51  REM
55  LET TNK = 49344
60  GOSUB 250: REM --GET DIMENSIONS--
70  GR
80  GOSUB 340: REM --BUILD TANK--
89  REM
90  REM  *** GET LEVEL ***
91  REM
100  POKE TNK,0
110 V =  PEEK (TNK)
120 Y = (V / 255) * (TP - BTTM) + BTTM
130 Y =  INT (Y)
139  REM
140  REM  *** ADJUST LEVEL ***
141  REM
150  COLOR= C1
160  FOR X = LFTSD TO RGTSD
170  VLIN BTTM - 1,Y AT X
180  NEXT X
190  COLOR= 0
200  FOR X = LFTSD TO RGTSD
210  IF Y <  = TP GOTO 230
220  VLIN Y - 1,TP AT X
230  NEXT X
240  GOTO 100: REM --LOOP AGAIN--
249  REM
250  REM  *** GET DIMENSIONS ***
251  REM
260  PRINT "ENTER TANK LIMITS"
270  INPUT "LEFT SIDE? <2-35>";LFTSD
280  INPUT "RIGHT SIDE? <4-37>";RGTSD
290  INPUT "TOP? <0-20> ";TP
300  INPUT "BOTTOM? <5-39> ";BTTM
310  INPUT "FLUID COLOR? <0-7>";C1
320  INPUT "TANK COLOR? <0-7>";C2
330  RETURN
339  REM
340  REM  ***  BUILD TANK  ***
341  REM
350  COLOR= C2
360  PLOT LFTSD - 2,TP
370  PLOT RGTSD + 2,TP
380  FOR S = TP TO BTTM
390  PLOT LFTSD - 1,S
400  PLOT RGTSD + 1,S
410  NEXT S
420  FOR B = LFTSD TO RGTSD
430  PLOT B,BTTM
440  NEXT B
450  RETURN
```

Figure 4-4
Fluid-level monitor listing

begins in the subroutine at line 340. Once the tank is built, the program returns to a continuous loop beginning at line 100 and ending at line 240. The loop begins with an input routine to get the level of the fluid (read from the A/D converter). Line 120 scales the digital value of the input so that the height of the fluid displayed is proportional to the voltage read in from the sensor.

Next, the tank is filled with the fluid, beginning from the left side and going to the right side, and from the bottom to the height Y. Blank space is painted from the level Y to the top of the tank. Then the loop repeats over and over again. Typing CTRL-C will exit the loop.

Try using the program by connecting a pot to the A/D converter input leads as was done in the bar graph demonstration. Vary the pot and you will see the tank fill and empty.

4-3 MULTIPLE ANALOG INPUTS

Whenever we want to monitor several analog inputs, we can use a multiplexed A/D converter, such as the ADC0809. The 0809 has eight analog inputs. One of those inputs is digitized, depending on a 3-bit address code applied to its channel-select inputs. A simple hookup between the 0809 and the Apple is shown in figure 4-5.

The conversion time for any one channel is 100 microseconds, just like the 0804. Unlike the 0804, the 0809 does not have an on-board clock, so the Apple Φ1 signal is fed to a flip flop, giving a converter clock frequency of about 500 kHz, which is typical for the 0809.

To start conversion, the address of the desired channel must be applied to inputs A, B, and C. Note that address lines A_0, A_1 and A_2 are used in this case. This is consistent with the fact that $\overline{\text{DEVICE SELECT}}$ is active for the lowest 16 addresses for any slot. The address bits are applied while the computer is doing a write (POKE) to the address of the desired channel. When write and $\overline{\text{DEVICE SELECT}}$ go active, the output of gate 2 goes HIGH. The rising edge of this signal at the START input of the converter resets the chip. The falling edge of the same pulse, applied to the address latch enable (ALE) input of the 0809, latches in the three address bits and begins conversion.

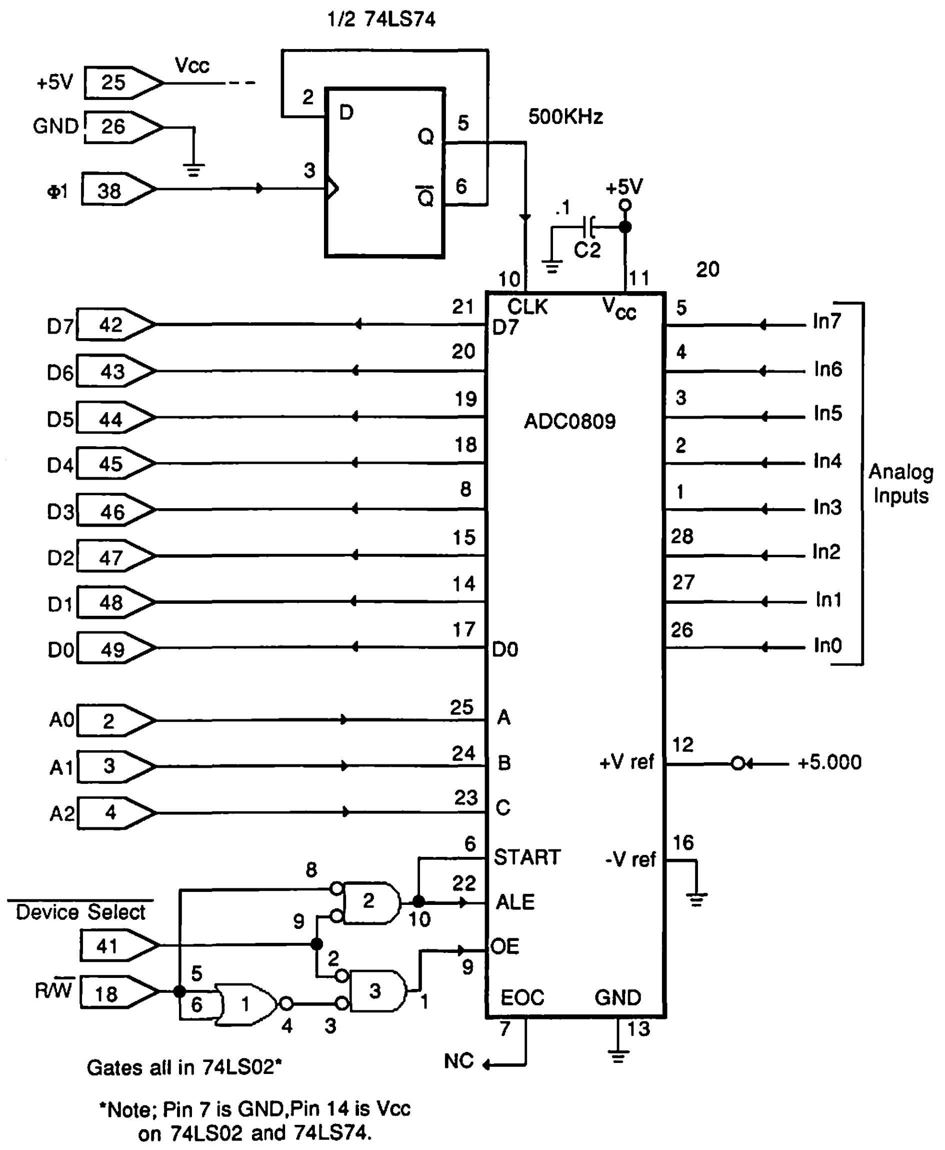


Figure 4-5
ADC0809 8-bit A/D converter with multiplexed inputs

After about 100 μs, the digital equivalent of the selected input can be read on the tristate data output pins of the ADC0809 by driving output enable (OE) HIGH. A simple routine to read in and display the analog values on inputs 0, 1, and 2 of the 0809 is as follows:

```
100  POKE 49344,0
110  LET V0 = PEEK(49344)
120  POKE 49345,0
130  LET V1 = PEEK(49345)
140  POKE 49346,0
150  LET V2 = PEEK(49346)
160  PRINT V0, V1, V2
```

(Card plugged into slot 4)

Line 100 starts conversion for channel 0, and line 110 reads in its digital value. Similarly, line 120 starts conversion for channel 1, and line 130 reads in its digital value, and so on. Actually, you do not have to specify a different address for each PEEK instruction. Lines 110, 130, and 150 could all use PEEK(49344). It is only necessary to specify the address of each channel in a POKE command. The address latched in by the POKE command is the address of the channel you will read from with the next PEEK command.

Figure 4-6 shows the listing of a simple data logging program. This program samples three analog input channels approximately once every 10 seconds and prints the values of voltages measured. The interval between measurements is controlled by the time delay in line 170. If you want to change the time interval, simply change the maximum value for D. A more accurate way to control the time interval between samples is to use a clock/calendar chip that generates pulses or interrupts each time a sample is to be taken.

Line 30 defines a scaling function that takes the input value from the A/D converter and changes it to a value between 0 and 5 with no more than three significant figures. Line 90 specifies the number of events to be recorded, which in this case is 10. You can obtain a hard copy printout of the measured values by simply having your printer turned on and typing PR#1 before typing RUN. Figure 4-7 shows a sample run of the program. Input voltages were arbitrarily changed during the run for demonstration purposes.

```
]LIST

10  REM  DATA LOGGING
15  LET ADC = 49344
20  PRINT "EVENT     ";"VO          ";"V1          ";"V2"
25  PRINT
30  DEF  FN S(V) =  INT (((V * 5 / 255) * 100) + .5) / 100
90  FOR T = 1 TO 10
100  POKE 49344,0
110 V =  PEEK (ADC)
115 VO =  FN S(V)
120  POKE 49345,0
130 V =  PEEK (ADC)
135 V1 =  FN S(V)
140  POKE 49346,0
150 V =  PEEK (ADC)
155 V2 =  FN S(V)
160  PRINT T;
162  PRINT  TAB( 10)VO;
164  PRINT  TAB( 20)V1;
166  PRINT  TAB( 30)V2
170  FOR D = 1 TO 7000: NEXT D
180  NEXT T
```

Figure 4-6
Listing for data logging routine

```
]RUN

EVENT     VO        V1        V2

1         1.82      1.51      1.96
2         2.08      2.27      2.35
3         2.31      2.98      2.57
4         2.49      3.55      2.8
5         2.67      4         3.04
6         2.88      4.86      3.39
7         3.08      5         3.59
8         3.33      4.9       3.9
9         3.82      3.8       4.16
10        4.04      2.75      4.49
```

Figure 4-7
Sample run of data logging routine

The input voltages could come from sensors that measure physical parameters, such as wind speed, temperature, pressure, or humidity. In that case, suitable scaling factors would have to be used for each input, and appropriate headings would have to be printed on the hard copy.

A/D converters are also available with 16 multiplexed channels. Two such devices are the ADC0816 and ADC0817, which perform similarly to the 0809.

CHAPTER 5

Structuring Your Applications Programs

Although this book is primarily hardware oriented, this chapter will briefly discuss some software design principles. Too many hardware designers neglect to use good software design practices. The result is programs that are difficult to read and understand and almost impossible to modify without a great deal of effort. The common opinion is if a program "works," that is all that matters. And the quicker one can get it to work, the better. There is some truth in that. After all, if a program is written, say for a programmable controller that turns motors on and off at some intervals, and if it does the job, what more could be asked? But what happens if the number of motors to be controlled is changed, if the time intervals are changed, or if other input conditions must be examined before a decision is made to run or not to run? The result might be a complete rewriting of the program or just a simple modification to the existing program, depending on the readability and flexibility of the original software.

Industry has recognized for years that too much time and money is wasted in redesigning poorly planned programs. Program writers must remember that writing the code for a system, whether in high-level or machine language, is only one part of the software development effort. The whole spectrum of software development has been said to include all of the following:

Definition and design
Coding
Debugging and testing
Documentation
Maintenance

Writing program code can be integrated effectively into the entire operation if adequate planning is done in the beginning. Part of that planning includes an organized approach to coding, often called *top-down design* or *structured programming*. This chapter will discuss some aspects of a structured approach.

5-1 DEFINE AND DESIGN

Before we can begin to build any system, whether it is a programmable controller, a data-logging weather station, or whatever, we must *define* what the system should do. Many small system designs gradually *evolve* into finished products. But often the system is so large that several people, even different design teams, may work on different parts of the system simultaneously. In this case, it is necessary to clearly outline, or block diagram, the entire system. It is at this point, early in the design phase, that decisions must be made as to what the limits of the system are, what hardware-software tradeoffs will be made, and so on. For instance, if a keyboard is needed to input some values, should a software encoded keyboard or a keyboard controller chip be used? The software keyboard will probably be less expensive, but it will also probably be slower. Which is more important for this particular application? This and similar decisions should be made early in the design phase with both hardware and software personnel giving their opinions. Even if you are designing the entire system on your own, consider the options before proceeding. Sound decisions made early can save headaches later.

Some things to consider in the definition phase are: What inputs and outputs must the system have? Will the hardware inputs be parallel binary information, which can be brought in through an input buffer, or will you need a serial input or an A/D converter? What outputs are needed? Will you have to drive motors and lamps by means of triacs, or will you need a D/A converter or maybe just a graphics display?

For example, let's define the Fluid Level Monitor program that we discussed in chapter 4. Our objective is to continually keep track of the level of fluid in a tank. Let's assume the tank will have a float gage that will produce a dc output voltage between 0 and +5v, depending on the level in the tank. When the tank is empty, the gage output will be 0v, and +5v will be its output when the tank is full. So the input to our computer will be through an A/D converter.

The only output we need now is a graphics display for the plant supervisor to watch. However, we want to prompt the user to enter values for the tank dimensions, as well as for the colors of the tank and the fluid. We do this with an eye to the future, in case we later decide to show more than one tank at a time on the screen.

Now that the system requirements have been defined, you can proceed to the *design* phase. You would not begin a hardware design from the bottom up by saying, "I think I'll use a 7400 and connect pin 3 to pin 4, and then. . . ." In other words, you would not start at the lowest level without having some overall plan to work from. You probably would start by drawing an overall block diagram of the system then gradually *refine* the block diagram into smaller and smaller chunks as you get deeper into the design. This process is called *top-down design*. Good software is designed in a similar manner. You begin with an overall plan, or outline, and gradually refine it into smaller and smaller modules as you get deeper into it.

One approach to designing software is to draw a flowchart describing what things are to be done and the order in which they are to be done. Unfortunately, many people do not like to write flowcharts first. This may be partially because to draw the flowchart well, you must have most of the details worked out in your mind beforehand. In addition, it is not very easy to change the flowchart many times without ending up with an unreadable mess. In practice, most good flowcharts are generated *after* the software is written to help the reader or troubleshooter understand the program. But there is an easy way for you to plan your program without a flowchart. You simply write an *outline* for the program, just as you would write an outline for a theme paper or a chapter for a book.

The outline for your program can consist of a series of one-line REM statements. Each REM statement will later be used to introduce and describe a subroutine that will do some small part of the overall task. By breaking up your large program into several small chunks (subroutines), you can easily write, test, and debug each of these parts, just as you can easily test and debug small modules in a large electronic system. Each module can be written, tested, and

```
]LIST

10  REM  ***  FLUID LEVEL MONITOR  ***
20  REM   --GET TANK DIMENSIONS--
30  REM  --BUILD TANK--
40  REM  --GET FLUID LEVEL--
50  REM   --ADJUST FLUID LEVEL DISPLAY--
```

Figure 5-1

Fluid-level monitor outline

stored until needed. After several modules have been written, they can be linked together and tested.

The outline for the Fluid Level Monitor program is shown in figure 5-1. For now, do not worry about line numbers or how many lines of code will be required for each part. The outline is simply written to indicate what things are to be done and in what order they are to be done. We can fill in more of the details later.

5-2 WRITING THE CODE

Some programming languages such as PASCAL lend themselves to good program structuring. One reason is that they read almost like plain English language and they do not use cryptic variable names like A1 and V2. In addition, PASCAL uses a *main driver,* or control, section of program that calls *procedures* to be performed as they are needed. We can set up our BASIC programs in a structured format by using the same principles. We can write a short *driver* program that calls *subroutines* to perform the individual small tasks.

Most programmers are aware that subroutines are used for a series of instructions which are executed more than once in a program. By using the subroutine, the *code* for that routine is written only once, and it may be accessed any time it is needed by using the command GOSUB NN. The last instruction in a subroutine must always be RETURN. When RETURN is executed, the program control returns to the line following the one that called the subroutine.

Subroutines have other valuable features besides eliminating the rewriting of code. They make the program more readable (better documentation), they aid in testing and debugging, and they make the overall program easier to write.

To illustrate these points, let's expand our outline of figure 5-1, as shown in figure 5-2. Notice that the major outline steps of lines 20, 30, 40, and 50 have become parts of what is called the main driver. They control the order in which the subroutines (which actually do the tasks) are called. *The main driver controls the entire program execution.* All other sections of the program are subordinate to the main driver. All subroutine calls and branches return to the

```
]LIST

10  REM  ***  FLUID LEVEL MONITOR  ***
15  REM  *** MAIN DRIVER ***
20  GOSUB 1000: REM --GET TANK DIMENSIONS--
25  GR
30  GOSUB 1200: REM --BUILD TANK--
40  GOSUB 100: REM --GET FLUID LEVEL--
50  GOSUB 200: REM --ADJUST FLUID LEVEL DISPLAY--
60  GOTO 40
100  REM *** GET FLUID LEVEL ***
110  REM  ..get value from A/D converter
120  REM  ...and scale it for graphics display
190  RETURN
200  REM  *** ADJUST FLUID LEVEL DISPLAY ***
210  REM  ..paint in fluid from bottom
220  REM  ...to height Y
290  RETURN
1000  REM  *** GET TANK DIMENSIONS ***
1010  REM  ..ask user to enter values
1020  REM  ...for sides, top, and bottom of tank
1090  RETURN
1200  REM  *** BUILD TANK ***
1210  REM  ..construct sides and bottom of tank
1220  REM  ...using lo-res graphics
1290  RETURN
```

Figure 5-2
Defining the subroutines of the fluid-level monitor

main driver. So by reading through the main driver subroutine calls, the reader can get an overview of what the entire program does. Only when the reader wants to know exactly how a certain part of the program is done does he or she have to read through the actual subroutine code.

Reading through the main driver of figure 5-2, we see that first a subroutine (GOSUB 1000) is called which asks the user to input position values for the top, bottom, and sides of the tank. After returning from that subroutine, the low-resolution graphics mode is set and the next subroutine is called, which draws the shape of the tank. Then, on returning to line 40, the next subroutine is called (GOSUB 100), which reads in an input from the A/D converter and scales it for the graphics display. Then line 50 calls the subroutine that fills the tank to the proper level with the color representing the fluid. The GOTO 40 puts the program in a continuous loop, getting the new fluid level and adjusting the graphics display accordingly. (To get out of the loop, hit RESET or CTRL-C.)

A couple of points should be noted here. The order in which the tasks are done depends on the order in which the subroutines are called, not on the line numbers of the subroutines themselves. Whenever Applesoft BASIC®* calls a subroutine, it searches for the starting line number beginning with the lowest numbers. Therefore, it is able to find low-numbered lines faster than high-numbered lines. So whenever you have subroutines that are called frequently or must be executed quickly, use low line numbers. Seldom called subroutines or subroutines that can execute slowly, say for displaying user prompts, can be assigned higher line numbers with no detrimental effects. This is why the GET FLUID LEVEL and ADJUST subroutines are placed at lower line numbers than the other two subroutines.

Having refined our original outline to the level of figure 5-2, we can proceed to write and test the actual subroutines. Since it usually does not matter with which subroutine you start, do the easy ones first. If you can not quite figure out how to handle a certain part of the program, put it off until later. Often your mind will subconsciously work on the problem, and you will find a solution eventually. Also, simply working on another part of the program might give you an idea for the difficult part.

Figure 5-3 shows the listing for the complete FLUID LEVEL MONITOR program in a reasonably well-documented form. Notice that while REM statements are used to set off each subroutine for clarity, few plain-language REM statements or comments are used overall. Although comments should be used if they will aid the reader, it is possible to use coding techniques that make the program almost self-documenting. For example, notice the use of *variable names* in the GET TANK DIMENSIONS subroutine. Line 1015 assigns the variable name LFTSD to the user's input for the left side limit. The contraction LFTSD is formed by simply dropping the vowels from the words LEFT SIDE. By using the contraction, the

Applesoft BASIC® is a registered trademark of Apple Computer, Inc.

```
]LIST

10  REM  ***  FLUID LEVEL MONITOR  ***
12  LET TNK = 49344
14  REM
15  REM  *** MAIN DRIVER ***
16  REM
20  GOSUB 1000: REM --GET TANK DIMENSIONS--
25  GR
30  GOSUB 1200: REM --BUILD TANK--
40  GOSUB 100: REM --GET FLUID LEVEL--
50  GOSUB 200: REM --ADJUST FLUID LEVEL DISPLAY--
60  GOTO 40
99  REM
100  REM *** GET FLUID LEVEL ***
101  REM
110  POKE TNK,0
115 V =  PEEK (TNK)
120 Y = (V / 255) * (TP - BTTM) + BTTM
125 Y =  INT (Y)
190  RETURN
199  REM
200  REM  *** ADJUST FLUID LEVEL DISPLAY ***
201  REM
210  COLOR= C1
215  FOR X = LFTSD TO RGTSD
220  VLIN BTTM - 1,Y AT X
225  NEXT X
230  COLOR= 0
235  FOR X = LFTSD TO RGTSD
240  IF Y <  = TP GOTO 250
245  VLIN Y - 1,TP AT X
250  NEXT X
290  RETURN
999  REM
1000  REM  *** GET TANK DIMENSIONS ***
1001  REM
1010  PRINT "ENTER TANK LIMITS"
1015  INPUT "LEFT SIDE? <2-35>";LFTSD
1020  INPUT "RIGHT SIDE? <4-37>";RGTSD
1025  INPUT "TOP? <0-20> ";TP
1030  INPUT "BOTTOM? <5-39> ";BTTM
1035  INPUT "FLUID COLOR? <0-7>";C1
1040  INPUT "TANK COLOR? <0-7>";C2
1090  RETURN
1199  REM
1200  REM  *** BUILD TANK ***
1201  REM
1210  COLOR= C2
1215  PLOT LFTSD - 2,TP
1220  PLOT RGTSD + 2,TP
1225  FOR S = TP TO BTTM
1230  PLOT LFTSD - 1,S
1235  PLOT RGTSD + 1,S
1240  NEXT S
1245  FOR B = LFTSD TO RGTSD
1250  PLOT B,BTTM
1255  NEXT B
1290  RETURN
```

Figure 5-3
Complete listing of the fluid-level monitor

reader will easily see where that variable is used in other parts of the program, and he or she will more easily follow the program flow than if single-letter or alphanumeric variable names are used. Be careful in choosing variable names that you avoid the use of reserved words. Often, dropping the vowels solves the problem for you. For example, line 115 gets the value of the fluid level from the A/D converter. The actual line reads V = PEEK(TNK). The word *tank* was abbreviated TNK because if the entire word TANK were used, Apple would interpret that word as a trig function TAN K. In general, using descriptive names for variables will make your program much easier to read.

Using names is also very helpful when referring to I/O ports. For example, line 12 equates TNK with the I/O address 49344 (slot 4 base address). If sometime later you decide to move the card to another slot, you simply have to change line 12. On the other hand, if you had used the address 49344 in every line that caused an input or output, you would have to search through the entire program listing and change every line referring to the old address.

Before leaving the topic of design, let's discuss a few more general points. In the old days of programming, when hardware was very expensive and rather slow, programs were given merit on their speed and conciseness. If program A ran faster than program B or took up less memory space than program B, A was considered a better program. That is not the case any more, or at least it should not be. Hardware is less expensive and much faster today than it was several years ago. But the programs are more complex and extensive, which makes debugging and maintenance time consuming and expensive. So *keep it simple—don't be tricky*. A program that is written simply and clearly is far easier to debug and modify than a tricky program. Avoid the temptation to show how clever you are. You will be appreciated much more by those who have to use and modify your program if you use a clear, organized approach.

Of course, there will be times when a certain section of a program must run fast, for example, when you need to rapidly update a display or to input and store a few hundred samples of some rapidly changing voltage. At these times, you may have to resort to tricks to gain the needed speed. Just remember to include a few comments or REM statements to explain to the reader exactly what you are doing. Whenever you need to do fast I/O, the best way is to call a machine-language subroutine to handle that part of the program. In the next chapter we will discuss how to use BASIC and machine code within the same program. Essentially, though, we will still use the main driver/subroutine format, keeping our programs readable.

Remember to save your programs frequently. Get into the habit of saving your current file on the disc every 15 minutes or so and every time you get up to get a cup of coffee or answer the phone. The few seconds taken to save your file will prevent a lot of anguish if there is a momentary power failure or if someone else sits down at the terminal while you are gone. Also be sure to make at least one backup disc with duplicates of all of your files. It is a good idea to rotate the two discs at least daily. That is, use disc A as your primary disc on Monday, Wednesday, and Friday, and use disc B on the other days. This will prevent an unhappy surprise if one of the discs is defective.

5-3 TESTING AND DEBUGGING

After you write a subroutine, you will want to *test* and *debug* it. You do not have to wait until all of the subroutines are finished. You can test each one independently, although sometimes it is just as well to link two or more together for testing. For example, suppose we want to test the program of figure 5-3. A good place to start would be to test the GET TANK DIMENSIONS and BUILD TANK subroutines at the same time. We could easily do this by putting a STOP instruction, say at line 35. When we type RUN, the program will ask us for the tank limits, and then it will construct the tank. After we are sure that this section of the program works, we can remove the STOP statement. We will want to try different values for the dimensions to make sure that all desired values will work. This aspect of testing, sometimes called *program validation,* checks to see that all permissible values of inputs work.

It is often a good idea to insert *traps* after the user inputs to prevent the program from "bombing" because of bad values. For example, line 1015 prompts the user to enter the position of the left side of the tank. The prompt also shows the limits of the left side. If the user wants to input a value outside those limits, we want to prompt him or her to input a new value. This can be accomplished by inserting a new line in our program, such as

```
1018   IF LFTSD <2 OR LFTSD >35 GOTO 1015
```

Similarly, we can put traps after each INPUT line. Also consider using default values, in case the user does not want to be bothered entering specific numbers or does not care what the actual value is. Figure 5-4 shows an expanded version of the GET TANK DIMENSIONS subroutine with traps after each input statement. It also includes default values.

To test the GET FLUID LEVEL subroutine without using the A/D converter input, we can delete lines 110 and 115 and put in a new line as follows:

```
115 INPUT "V= ";V
```

When you run the program, you will be prompted to input the value of V, which would normally come from the A/D converter. You will then enter a value between 0 and 255. If you wish, you can also put in a temporary line such as:

```
126  PRINT Y
```

to see if the calculations of lines 120 and 125 are correct.

```
]LIST

1000  REM  *** GET TANK DIMENSIONS ***
1001  REM
1005  INPUT "WANT TO ENTER NEW TANK DIMENSIONS? <Y/N>";A$
1006  IF A$ = "Y" GOTO 1010
1007 LFTSD = 2:RGTSD = 37:TP = 0:BTTM = 39:C1 = 2:C2 = 7: REM  DEFAULT VALUES
1008  GOTO 1090
1010  PRINT "ENTER TANK LIMITS"
1015  INPUT "LEFT SIDE? <2-35>";LFTSD
1018  IF LFTSD < 2 OR LFTSD > 35 GOTO 1015
1020  INPUT "RIGHT SIDE? <4-37>";RGTSD
1023  IF RGTSD < 4 OR RGTSD > 37 GOTO 1020
1025  INPUT "TOP? <0-34> ";TP
1028  IF TP < 0 OR TP > 34 GOTO 1025
1030  INPUT "BOTTOM? <5-39> ";BTTM
1033  IF BTTM < 5 OR BTTM > 39 GOTO 1030
1035  INPUT "FLUID COLOR? <0-7>";C1
1038  IF C1 < 0 OR C1 > 7 GOTO 1035
1040  INPUT "TANK COLOR? <0-7>";C2
1043  IF C2 < 0 OR C2 > 7 GOTO 1040
1090  RETURN
```

Figure 5-4
Adding traps and user options

CHAPTER 6

Using BASIC and Machine Language Effectively

In the previous chapters, we have used only BASIC language. BASIC is quite powerful and easy to use, but it is also rather slow. In many applications, our computer must execute certain sections of a program as rapidly as possible. The best way of getting the computer to operate fast is to use machine language rather than BASIC.

The purpose of this chapter is not to teach machine-language programming but rather to show how to combine the use of machine-language subroutines with a BASIC driver to give us the best of both software worlds. Sufficient directions will be given in this chapter for you to enter and execute a few machine-language routines even if you are totally unfamiliar with the techniques. But if you want to eventually write your own machine-language programs, you should study one of the many books on the subject, which can be found in any good computer store or mail-order book store.

6-1 WHEN TO USE BASIC—WHEN TO USE MACHINE LANGUAGE

Since BASIC is much easier to write and to read than machine language, the controlling or executive program should usually be written in BASIC. Let's look at a few examples of where to use BASIC.

Use BASIC in the *initialization* part of a program when you assign names to variables and I/O ports, as was discussed in an earlier chapter. Remember, using names for variables and I/O ports makes your program much easier to read and modify.

Whenever *menus* are to be displayed to the user, BASIC should be your choice. While it is possible to print messages on the screen using machine language, why do things the hard way? With BASIC it's a snap.

Whenever the user is requested to *input* information by means of the keyboard, again BASIC is the simplest language to use. Since the user is extremely slow in response compared to the computer, there is no need for fast I/O here. Also be sure to use BASIC traps and user prompts, as was demonstrated in an earlier chapter.

Any *number crunching* or *mathematics*, other than simple addition, subtraction, or comparison, should also be done in BASIC.

Outputs to ordinary *printers* and *inputs from disc drives* are extremely easy tasks using BASIC. Since the printer or disc drive is probably slower than the BASIC programs controlling them, no advantage would be gained by going to machine language.

Finally, *fixed* or *slowly moving graphics* can be easily generated by BASIC using the techniques explained in your computer's reference manual or tutorial text. Only when higher-speed, moving graphics are needed will you have to go to machine language. (Programming high-speed graphics requires considerable expertise in the use of machine language and will not be discussed here.)

So when should you use *machine language*? Whenever you need a certain section of the program to execute at *high speed*. For example, in chapter 4 we used an A/D converter to measure the value of some voltage source. We then displayed the value of that voltage numerically on the video screen, much the same as is done by a DVM. But suppose our input voltage is rapidly changing. To view a rapidly changing waveform, you normally use a scope. In fact, in chapter 7 we will see how to build a storage scope with the computer. Under the control of a fast machine-language subroutine, we will sample the output of the A/D converter hundreds of times each second and store the instantaneous values of voltage in successive memory locations. Then upon returning from the high-speed input routine, we will use BASIC to display the instantaneous values graphically, giving us a trace similar to what we would see on a real-time scope.

Similarly, we can generate any desired waveform in real time by POKING the instantaneous values of the voltage waveform into successive memory locations. Then we can call a machine-language subroutine that rapidly outputs these values successively to a D/A converter, thereby generating the desired waveform.

Another use of a fast-output routine is tone generation. If you did all of the steps in the Applesoft Tutorial, you learned how to make the built-in speaker tick or buzz. But you can get practically any audio frequency tone out of the speaker. All you have to do is toggle it on and off at a sufficiently high rates of speed. BASIC cannot do this fast enough—but machine language can. Machine language can do jobs like toggling a speaker *hundreds of times faster* than BASIC. Therefore, much higher frequency tones are possible.

Whether you are aware of it or not, every program you run calls machine-language subroutines located within the Apple's monitor ROM. These subroutines quickly do the myriad of tasks needed to calculate, display, print, or do whatever must be done to make the program work. The calling of these subroutines is transparent to the BASIC programmer, who is only concerned with the BASIC program. Nevertheless, many ROM-based subroutines are called even for the simplest BASIC operation. The real advantage in calling your own specialized machine-language subroutine at a critical point in the program is that *your subroutine is custom designed* to do a specific task. Since your custom routine does not usually have to call many other routines to accomplish its task, it executes very fast.

6-2 ENTERING MACHINE-LANGUAGE PROGRAMS INTO RAM

Most machine-language programs, especially longer ones, are first written in *assembly language* using an *assembler* program. When using an assembler, the programmer does not need to look up the actual op codes for each instruction, but rather he or she simply types in the mnemonics for each instruction and the assembler converts the mnemonics into the corresponding machine codes.

You may want to become familiar with assembly-language programming if you intend to develop many of your own specialized programs or subroutines. There are several good texts on this rather advanced subject, so it will not be covered here. However, a programmer does not need an assembler if the machine-code listing is already available. For the applications discussed in this book, the complete assembly- and/or machine-language programs will be given. So all we need to discuss is how to enter the machine code into memory and how to run it.

As described in the Apple's Reference Manual, the monitor has routines to allow easy entry of machine code directly into RAM. The first thing you do is get into the monitor mode by typing

```
CALL -151
```

You should then see an asterisk (*) prompt, indicating that you are in the monitor.

Next, type in the hex address of the RAM location where you want to enter the first byte of the program. Then type a colon, followed by the hexadecimal byte you want in that location. Then hit the space bar once, and type in the byte you want in the next location. You do not need to type in each new address—it automatically increments each time a new byte is entered. You can continue to type in up to 85 bytes as you enter your program. If more than 85 bytes are needed, you simply type in the address of the first of a group of bytes and enter the bytes as before. The last byte of the program should be 60 (RTS). When you have typed in the final byte, hit RETURN and your program will be entered in RAM.

For example, let's enter a short machine-code routine into RAM starting at location 0300 hex. First, type

```
CALL -151
```

Then when you see the asterisk prompt, type

```
*0300: A0 00 AD 30 C0 88 D0 04 C6 01 F0 08 CA D0
F6 A6 00 4C 02 03 60
```

and hit RETURN.

The program is now in RAM. To examine the program in RAM, and also to see the use of the monitor's *disassembler*, type

```
300L
```

The L stands for LIST. You should now see a listing of your program on the video screen, which looks like figure 6-1. This listing is generated by the disassembler in the Apple. The disassembler converts the op codes of the machine code program into their corresponding mnemonics and prints the program in a more readable format.

Let's look at the listing. The leftmost column shows the hex address where each op code is stored, the second column shows the op code, and the third/fourth column shows the operand (value to be operated on) or the operand address. For example, in this listing the op code A0 in memory location $0300 stands for Load Register Y, shown as LDY in the mnemonic field. The value 00 is to be loaded into the Y register, so the operand is 00. On the second line of the listing, we see that in location $0302 we have the byte AD, which is the op code for the load accumulator (LDA) instruction.

```
Address  Op Code  Operand/Address  Mnemonic  Operand/Address

0300-    A0 00                     LDY       #$00
0302-    AD 30 C0                  LDA       $C030
0305-    88                        DEY
0306-    D0 04                     BNE       $030C
0308-    C6 01                     DEC       $01
030A-    F0 08                     BEQ       $0314
030C-    CA                        DEX
030D-    D0 F6                     BNE       $0305
030F-    A6 00                     LDX       $00
0311-    4C 02 03                  JMP       $0302
0314-    60                        RTS
0315-    68                        PLA
0316-    28                        PLP
0317-    60                        RTS
0318-    00                        BRK
0319-    00                        BRK
031A-    00                        BRK
031B-    00                        BRK
031C-    00                        BRK
031D-    00                        BRK
```

Figure 6-1
Disassembled listing of machine code program

The next two bytes, 30 and C0, give the address of the byte to be loaded into the accumulator. Note that the computer expects to find the *low byte first* then the high byte. However, in the operand/ address field following the mnemonic field, the addresses are shown with high byte first for easy reading. You can see that it is much easier to read through the disassembled listing than it is to read through the original machine-code program you keyed in. Your last program byte 60 is in memory location 0314. Any bytes LISTed after that are simply values that were already in RAM before you entered your program.

When run, this program generates a tone in the Apple's speaker. Prior to running the program, you must store a pitch parameter (number) in memory location 0000 and a duration parameter in memory location 0001. You can do this either through BASIC POKE commands or while in the monitor (asterisk prompt), just as you entered the program above. For example, if you type

```
*00: C3 C0
```

then hit RETURN, you will hear a middle C on the musical scale when you run the program.

Figure 6-1 shows that first the Y register is cleared then the speaker is toggled. The program then branches back and forth decrementing the Y and the X registers. Each time the X register goes to zero, it is reloaded with the pitch parameter and the speaker is toggled. Each time the Y register goes to zero, the duration parameter is decremented. When the duration parameter (contents of 0001) goes to zero, the program returns to the driver that called it. Therefore, each time you run the routine, you must load memory location 0001 with a duration value.

To run this short program, type

```
*300G
```

The G stands for GO. If your program runs correctly, you should hear a tone for a short time. You can experiment with changing the pitch and duration parameters on your own. Table 6-1 shows the approximate correspondence between the pitch parameters for an Apple II+ and the notes of the musical scale. The values are shown in decimal, the way you would enter them from BASIC.

Machine-language programs are often less forgiving than BASIC. Since a minor error can easily cause the program to bomb, it is usually a good idea to double check your program after entering it to make sure you entered it correctly. If it does bomb or you observe strange things happening, first try typing CTRL-C. This should bring the computer back to BASIC. If CTRL-C does not work, hit the RESET button. If the computer still does not respond, you may have to shut down and start up again.

TABLE 6-1
Pitch Parameters to Be Used with Routine of Figure 6-1

Pitch	*Value*
C	195
D	175
E	155
F	146
G	130
A	115
B	103
C2	98

Let's assume that your program runs OK. To get out of the monitor mode and back to BASIC, type CTRL-C.

Machine-language programs can be stored in any area of RAM not used by the computer either for BASIC or for its own internal housekeeping. A convenient area for small routines is in page three, except for the upper 16 locations from \$03F0 to \$03FF. The Apple uses those locations for itself. Locations \$0300 through \$03EF (decimal 768 to 1007) are available for user RAM.

6-3 SAVING AND LOADING MACHINE-LANGUAGE PROGRAMS WITH DISCS

Once you have entered your machine-code program in RAM, you will want to save it on disc. (Assume that you have already returned to BASIC by typing CTRL-C.) The format for saving a machine-language program already in RAM is

```
BSAVE filename, A$aaaa,L$
```

where the hexadecimal digits following the letter A give the starting address of the program, and the digits following the L give the length of the program in bytes. These are not optional; you must supply them.

For example, suppose we want the sound-generating routine mentioned in section 6-2 saved with the filename SOUNDGEN.

This program starts at $0300 and is $15 (21 decimal) bytes long. To save the program simply type

```
BSAVE  SOUNDGEN, A$300,L$15
```

and hit RETURN. The program will be saved as a binary file and will be listed in the disc's catalog as a B file.

You have the option of specifying either the address or the length, or both, in decimal. For example, BSAVE SOUNDGEN, A$300,L21 or BSAVE SOUNDGEN, A768,L21 will also work.

To load a previously saved binary program into the same area of memory from which it was originally saved, simply type

```
BLOAD  filename
```

You can load a binary file into a different area in memory by typing

```
BLOAD  filename, A$aaaa
```

where the hexadecimal digits following the A specify the address where you want to begin loading your file. This feature allows you to accumulate a *library* of useful machine-language subroutines from which to choose for use in particular applications. You can easily load them anywhere you choose, without regard to where they initially resided. Again, you can optionally specify the address in hexadecimal or in decimal.

6-4 CALLING MACHINE-LANGUAGE SUBROUTINES FROM WITHIN A BASIC PROGRAM

You learned how to load and run a machine-language program using monitor commands. Now we will look at an effective way of using machine-language subroutines controlled by a BASIC driver.

Assume that the machine-language program SOUNDGEN is already loaded into RAM starting at $300 (768 decimal). Figure 6-2 is the listing of a simple BASIC program that calls the machine-language subroutine. Notice that the program begins in BASIC,

```
]LIST

10  REM  ** MACHINE LANG CALL DEMO **
20 DUR = 200: REM  DURATION PARAMETER
30  INPUT "ENTER PITCH ";PITCH
40  POKE 0,PITCH
50  POKE 1,DUR
60  CALL 768
70  GOTO 30
```

Figure 6-2
BASIC program that calls a machine language routine

```
]LIST

10  REM  ** MACHINE LANG CALL DEMO **
15 SOUNDGEN = 768
20 DUR = 200: REM  DURATION PARAMETER
30  INPUT "ENTER PITCH ";PITCH
40  POKE 0,PITCH
50  POKE 1,DUR
60  CALL SOUNDGEN
70  GOTO 30
```

Figure 6-3
Calling a subroutine by name

```
]LIST

10  REM  ** MACHINE LANG CALL DEMO **
12 D$ =  CHR$ (4): REM  CHR$(4) IS CTRL-D
13  PRINT D$;"BLOAD SOUNDGEN,A768"
15 SOUNDGEN = 768
20 DUR = 200: REM  DURATION PARAMETER
30  INPUT "ENTER PITCH ";PITCH
40  POKE 0,PITCH
50  POKE 1,DUR
60  CALL SOUNDGEN
70  GOTO 30
```

Figure 6-4
Loading a binary file from within a BASIC program

jumps to the machine-language subroutine, and returns to BASIC. The CALL 768 works similarly to a GOSUB instruction, except that the address of the first instruction (768) is specified rather than a BASIC statement number.

When you run the program, you will be prompted to enter a PITCH parameter. You should then key in a number between 1 and 255 and hit RETURN. The computer will play a note corresponding to the value you keyed in. If you wish, you can alter the DURation parameter as well.

To pass a value from BASIC to the machine-language subroutine, we first execute the BASIC instruction POKE 0, PITCH (line 40). After the CALL instruction is executed, the machine-language instruction LDX $00 loads our PITCH value into the X register to be used in the subroutine.

When the CALL instruction is executed, the return address of the next BASIC instruction is automatically pushed onto the stack. This also happens for a GOSUB instruction. The last instruction of the machine-language subroutine must be return from subroutine (RTS). When the RTS is executed, program control returns to BASIC.

To make the program more readable (self-documenting), you can use a subroutine name rather than the starting address in the CALL instruction, as is shown in figure 6-3. Line 15 equates the subroutine name SOUNDGEN with the starting address 768. Then line 60 calls SOUNDGEN by name.

The program of figure 6-3 assumes that the machine code was already loaded in RAM. But we can have our BASIC program load the machine-language program when we run it by embedding DOS commands within the BASIC program. (See figure 6-4.)

When line 13 is executed, the computer will load the binary file SOUNDGEN from the disc into RAM, starting at location 768 decimal. Then the remainder of the program will run as before.

Figure 6-5 shows the complete listing for a program to generate simple tunes with the Apple. It demonstrates the use of a BASIC driver, which first loads the machine-language subroutine then calls various subroutines as they are needed. Study this program to see the interaction of the various parts. You can enhance the program by passing a duration parameter for each tone, as well as a pitch parameter.

```
]LIST

1  REM   ******************
2  REM   *                *
3  REM   *  MUSIC MAKER   *
4  REM   *                *
5  REM   * J. OLEKSY 1984 *
6  REM   *                *
7  REM   ******************
10  REM
12 D$ =  CHR$ (4): REM  CHR$(4) IS CTRL-D
13  PRINT D$;"BLOAD SOUNDGEN,A768"
15 SOUNDGEN = 768
20 DUR = 200: REM  DURATION PARAMETER
25  GOSUB 500
100  REM
101  REM  ** PLAY NOTES **
102  REM
110  FOR N = 1 TO 50
120 PITCH = S(N)
130  IF PITCH = 0 THEN  END
140  POKE 0,PITCH
150  POKE 1,DUR
160  CALL SOUNDGEN
170  NEXT N
500  REM
501  REM  ** GET USER INPUTS **
502  REM
510  DIM S(50)
519  PRINT
520  PRINT "   KEYBOARD NUMBERS 1 THROUGH 8"
522  PRINT "CORRESPOND TO NOTES ON MUSICAL SCALE"
523  PRINT "   C,D,E,F,G,A,B,C2 RESPECTIVELY"
524  PRINT
530  PRINT " ENTER THE NOTES YOU WANT PLAYED"
531  PRINT
532  PRINT "   TYPE 0 AFTER THE LAST NOTE"
534  PRINT
540  FOR N = 1 TO 50
550  INPUT P
555  IF P = 0 GOTO 600
560  ON P GOTO 571,572,573,574,575,576,577,578
571 S(N) = 195: NEXT N
572 S(N) = 175: NEXT N
573 S(N) = 155: NEXT N
574 S(N) = 146: NEXT N
575 S(N) = 130: NEXT N
576 S(N) = 115: NEXT N
577 S(N) = 103: NEXT N
578 S(N) = 98: NEXT N
600  RETURN
```

Figure 6-5
Program to play simple tunes

```
]LIST

10  REM  ** APPLE ORGAN **
20 MEM = 0
25 MUSIC = 768
30  PRINT "ENTER NOTES"
40  GET N
50  ON N GOTO 210,220,230,240,250,260,270,280,290
60  POKE MEM,V
63  POKE 1,200
100  CALL MUSIC
105  GOTO 40
210 V = 195: GOTO 60
220 V = 175: GOTO 60
230 V = 155: GOTO 60
240 V = 146: GOTO 60
250 V = 130: GOTO 60
260 V = 115: GOTO 60
270 V = 103: GOTO 60
280 V = 98: GOTO 60
```

Figure 6-6
Apple organ listing (assumes MC routine of Figure 6-1 in RAM)

A variation of the MUSIC MAKER program is shown in figure 6-6. This program, called APPLE ORGAN, allows you to play the number keys on the keyboard as if they were keys on an organ. As in the MUSIC MAKER program, the duration of each note is fixed.

Often a machine-language subroutine is used to obtain information that will be used later in the BASIC program, for example, a byte read from an input port. So the information obtained by the machine-language routine must be passed back to the BASIC driver. The act of transferring information back and forth between various segments of a program is known as *passing parameters*.

One simple and effective way of passing parameters, which we have already seen, is to use certain designated RAM locations that are accessible by all program segments. For example, we could designate memory location 0000 as our temporary "mailbox." If our machine-language program must transfer the byte from the accumulator back to the BASIC program, we simply execute the machine-language instruction.

```
STA $00
```

prior to doing the return from subroutine. Then once we are back in BASIC, we execute an instruction such as

```
LET BYTE = PEEK(0)
```

The BASIC variable BYTE will then have the value that was in the accumulator. If more than a single byte must be passed, we simply reserve as many memory locations as necessary to hold our parameters. We will use this method in the next chapter.

CHAPTER 7

Using the Computer as a Storage Scope

We have all used oscilloscopes to view waveforms in real time, that is, as they are happening. But sometimes the waveform we want to view occurs only occasionally, such as a transcient pulse. Or perhaps the waveform we want to view is not periodic, and we cannot sync on any portion of it to allow viewing. An example of this is the signals on the data bus of any microprocessor as a program is being run. Since the data on the bus is changing frequently and the pattern is not repetitive, we see garbage on the scope.

But if we can somehow store the transient pulse as it occurs, then we can "play it back" later for viewing. The *storage scope* allows us to do just that. In this chapter we will discuss using the computer as a simple storage scope. We will use an A/D converter to digitize instantaneous values of a waveform, and we will store them in memory. After storing enough values, we will have the computer plot the stored values to reconstruct the original waveform.

7-1 BARE BONES STORAGE SCOPE

Figure 7-1 shows a very simple setup to demonstrate our storage scope. We use the ADC0804 that we studied in chapter 4. If you do not remember what the 0804 does, you can review chapter 4 as needed.

Notice that an ac signal generator is capacitively coupled to the +V input so that the ac voltage is superimposed on a dc level. The slider of the pot connected to the +5v supply should be adjusted to about midrange as a starting value. The ac signal generator should be adjusted for an output signal of about 2v p-p, at a frequency of a few hundred hertz. With the input values adjusted in this manner, the total signal voltage applied to the +V input is within the normal range of the ADC, and each time a sample is taken the total input voltage will be converted to a number between 0 and 255.

Note also that the interrupt ($\overline{\text{INTR}}$) output (pin 5) of the 0804 is connected to the interrupt request ($\overline{\text{IRQ}}$) input (pin 30) of the edge connector. The $\overline{\text{INTR}}$ output of the ADC goes LOW to signal when

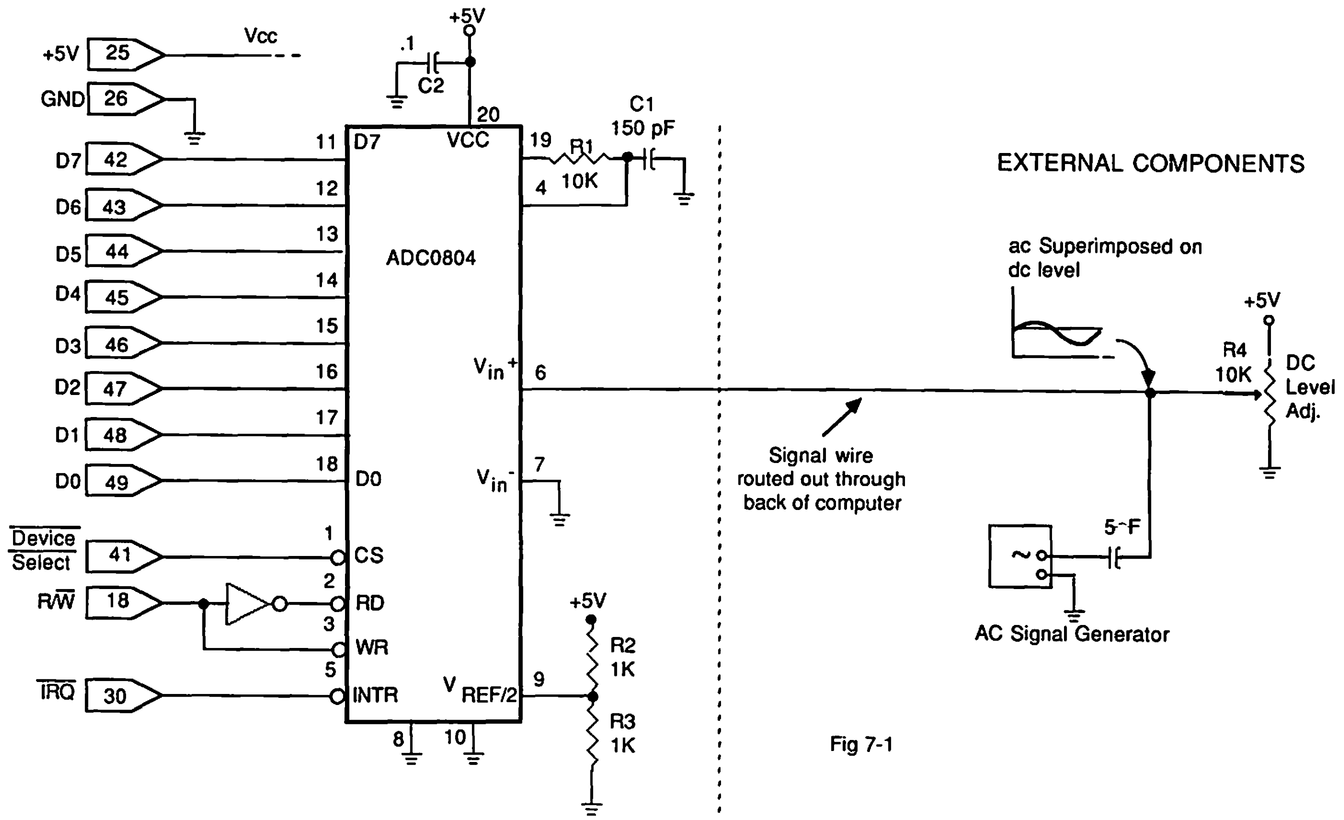


Figure 7-1
Circuit for storage scope

conversion is complete. $\overline{\text{INTR}}$ gets reset (goes inactive HIGH again) when the processor reads data from the ADC.

Assuming that we have the hardware set up as needed, with the ADC0804 card plugged into slot 4, let's examine the program shown in figure 7-2. During initialization we BLOAD the machine-language program FILBUF, whose assembly-language listing is shown in figure 7-3. We will discuss FILBUF later. For now let's continue on with the BASIC program.

Line 50 equates the name BUFFER with address 3072 ($0C00). BUFFER is an area in free RAM where we will store our samples of digitized input voltage. We will save 255 samples in memory locations 3072 through 3327.

Line 60 tells our BASIC program where to find the machine-language subroutine FILBUF.

Line 70 POKEs values into locations 1022 ($03FE) and 1023 ($03FF). These values, called the *interrupt vector,* tell the computer where to go ($0320) to find the interrupt service routine.

The use of interrupts requires explanation. Recall that in chapter 4 we were using only BASIC to access the ADC. Since BASIC operates relatively slowly, we did not have to wait for the 0804 to tell us when the conversion was complete. We simply POKEd the $\overline{\text{WR}}$ input of the chip to start conversion, then we immediately did a PEEK at the same address, assuming that the conversion was complete. But if we need to get many successive samples and we want to get them as quickly as possible, we use a machine-language subroutine to get them. However, the machine-language routine can read in the sample, store it in memory, and be ready to read in the next sample before the ADC has completed the next conversion. So the best way to handle the timing problem is to have the ADC tell the computer when its conversion is complete by requesting an interrupt.

Here is how the Apple handles an interrupt request:

1. The interrupting device (the ADC) requests an interrupt by pulling the interrupt request ($\overline{\text{IRQ}}$) line (pin 30) LOW.
2. The processor finishes its current instruction and then tests the interrupt mask (I) flag in its flag register. If the mask is set, the processor ignores the interrupt request and continues on with its present program.

```
]LIST

10  REM  ********************
12  REM  *   STORAGE SCOPE   *
14  REM  *  J. OLEKSY 1984   *
18  REM  ********************
19  REM
20  REM  *** INIT ***
21  REM
30 D$ = "": REM  CTRL-D
40  PRINT D$"BLOAD FILBUF"
50 BUFFER = 3072
60 FILBUF = 768
70  POKE 1022,32: POKE 1023,03: REM   * INTERRUPT VECTOR *
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  CALL FILBUF: REM  * GET INPUTS FROM ADC *
120  HGR : HCOLOR= 7
130  GOSUB 700: REM  * PLOT AXES *
140  GOSUB 500: REM  * PLOT WAVEFORM *
150  GOSUB 800: REM  * GET USER INPUT *
160  GOTO 110: REM  * REPEAT LOOP *
499  REM
500  REM  *** PLOT WAVEFORM ***
501  REM
510 MEM = BUFFER
520 X = 0
530 Y =  PEEK (MEM)
540  HPLOT X,Y
550  FOR X = 1 TO 254
560 MEM = MEM + 1
570 Y =  PEEK (MEM)
580  HPLOT  TO X,Y
590  NEXT X
595  RETURN
699  REM
700  REM  *** PLOT AXES ***
701  REM
710  FOR X = 0 TO 250 STEP 10
720  HPLOT X,80
730  NEXT X
740  FOR Y = 0 TO 160 STEP 10
750  HPLOT 0,Y
760  NEXT Y
770  RETURN
799  REM
800  REM   *** USER INPUT ***
801  REM
810  INVERSE
815  VTAB 23
820  PRINT "HIT SPACEBAR FOR NEW TRACE"
830  PRINT : PRINT "TYPE Q TO QUIT"
840  GET A$
850  PRINT : PRINT : PRINT
860  NORMAL
870  IF A$ = "Q" GOTO 890
880  RETURN
890  TEXT : HOME : END
```

Figure 7-2
BASIC listing for storage scope

```
SOURCE FILE: FILBUF
0000:           ***********************
0000:           *                     *
0000:           *       FILBUF        *
0000:           *                     *
0000:           *    J. OLEKSY 1984   *
0000:           *                     *
0000:           * FILBUF READS VOLTAGE *
0000:           * VALUES FROM A/D CONV *
0000:           * IN SLOT 4 AND STORES *
0000:           * THEM IN 255 MEMORY  *
0000:           * LOCATIONS STARTING  *
0000:           *     AT $0C00        *
0000:           *                     *
0000:           ***********************
----- NEXT OBJECT FILE NAME IS FILBUF.OBJ0
0300:                   ORG  $300
C0C0:           CNVRTR  EQU  $C0C0
0C00:           BUFFER  EQU  $0C00
0300:08         FILBUF  PHP              ;SAVE REGISTERS
0301:48                 PHA
0302:98                 TYA
0303:48                 PHA
0304:8A                 TXA
0305:48                 PHA
0306:A2 00              LDX  #0          ;POINT TO FIRST LOC
0308:8D C0 C0           STA  CNVRTR      ;START CONVERSION
030B:58                 CLI
030C:E0 FF      LOOP    CPX  #$FF        ;ENOUGH SAMPLES?
030E:D0 FC              BNE  LOOP        ;  IF NOT-LOOP AGAIN
0310:78                 SEI              ;YES,ENOUGH-GO BACK TO DRIVER
0311:68                 PLA              ;RESTORE REGISTERS
0312:AA                 TAX
0313:68                 PLA
0314:A8                 TAY
0315:68                 PLA
0316:28                 PLP
0317:60                 RTS
0318:           ***********************
0318:           *                     *
0318:           *  INTERRUPT SERVICE  *
0318:           *       ROUTINE       *
0318:           *                     *
0318:           ***********************
----- NEXT OBJECT FILE NAME IS FILBUF.OBJ1
0320:                   ORG  $320
0320:AD C0 C0   INTSRV  LDA  CNVRTR      ;GET SAMPLE
0323:8D C0 C0           STA  CNVRTR      ;START NEXT CONVERSION
0326:9D 00 0C           STA  BUFFER,X    ;SAVE SAMPLE
0329:E8                 INX              ;POINT TO NEXT BUFFER POS
032A:58                 CLI
032B:40                 RTI

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 7-3

Assembly listing for FILBUF

3. If the I flag is *not* set, the processor pushes its accumulator, flag register, and program counter contents onto the stack. It then sets the I flag to prevent further interrupts.
4. Next the processor loads the program counter low byte from 1022 ($03FE) and the program counter high byte from 1023 ($03FF).
5. The computer then effectively jumps to the address now loaded in the program counter and fetches its next instruction. The two bytes in locations 1022 and 1023, called the *interrupt vector,* must have been loaded by the programmer with the starting address of the interrupt service routine prior to any interrupt requests.
6. The last instruction in the interrupt service routine must be ReTurn from Interrupt (RTI). When the RTI instruction is decoded, the processor pulls bytes off the stack, thus restoring the accumulator, flag register, and program counter to what they were prior to the interrupt request.
7. Finally, the program resumes where it left off prior to the interrupt.

Looking at our BASIC program again, our main driver starts at line 100. We first call the machine-language subroutine FILBUF, which fills the buffer with 255 samples of the input waveform. Then the Hi-res graphics mode is set and the BASIC subroutine at line 700 plots our axes. Next the routine at line 500 is called, which takes successive values from the buffer, plots them, and connects the plotted points together, thus reconstructing the original waveform that was stored in memory. The user is then asked if a new trace is desired. If the spacebar is hit, a new waveform is sampled, stored, and displayed. If nothing is done, the previous waveform remains on the screen.

Figure 7-3 shows the assembly-language listing for FILBUF. It begins by EQUating the names CNVRTR and BUFFER to the addresses of the ADC and memory buffer, respectively. It then pushes all registers onto the stack so no problems will be encountered when returning to the main driver. Next the X index register is cleared, and the ADC is given a "start conversion" signal by the instruction on line 25. The interrupt mask is cleared so that interrupts will be acknowledged.

The next two instructions, CPX and BNE, hold the processor in a tight test loop while waiting for interrupt. When a conversion is complete, the interrupt request causes the program counter to be loaded with $0320, the starting address of the interrupt service routine. (Remember that the address $0320 was POKEd into $03FE and $03FF in line 70 of the BASIC program.) The service routine, starting at line 44, loads the accumulator with the digitized value of the input voltage and starts a new conversion. It then stores the present value in the buffer location whose base address is $0C00, using the X register contents as an index. Next the index register is incremented to point to the next available location, the interrupt mask is cleared, and the processor returns to where it left off in the test loop. Note that the interrupt mask must be cleared each time we return from interrupt so that the next interrupt can be acknowledged.

The interrupt process is repeated 255 times, and each time a new sample is stored in the buffer. When the X register contents reach $FF, the program falls through, sets the interrupt mask to prevent further interrupts, and restores all registers to what they were prior to calling FILBUF. Then control returns to BASIC at line 120.

You do not need an assembler to run the program. Simply enter the machine code for the subroutine using the method described in chapter 6. The machine code dump is shown in figure 7-4 for your convenience.

When you run this program, experiment with the level adjust pot and with the amplitude and frequency adjustments on the signal generator to see their effects on the display.

```
*300.32B

0300- 08 48 98 48 8A 48 A2 00
0308- 8D C0 C0 58 E0 FF D0 FC
0310- 78 68 AA 68 A8 68 28 60
0318- 00 00 00 00 00 00 00 00
0320- AD C0 C0 8D C0 C0 9D 00
0328- 0C E8 58 40
```

Figure 7-4
Machine code listing for FILBUF

7-2
USING A TRIGGERED SWEEP

In the STORAGE SCOPE program covered in section 7-1, the computer began taking in samples as soon as the spacebar was hit. This may be satisfactory for some simple uses, but just as with a real-time scope, we may want to have some *trigger* circuit begin sampling the input waveform when the input signal crosses some critical value or when some external event occurs. We will now add a few components to our A/D converter sampling system that will cause sampling to begin only after a trigger signal occurs.

The circuit of figure 7-5 shows the necessary modifications for using an external trigger. We will refer to this process as a *triggered sweep,* as in a real-time scope. Note that the input signal is applied to a comparator, as well as to the ADC. With the slope switch in the position shown, whenever the input voltage (total value of ac + dc) is less positive than the voltage at the slider of the trigger level pot, the output of the comparator is LOW. Then when the input voltage gets more positive than the trigger level setting, the comparator output switches HIGH. We use the transition from LOW to HIGH to clock the 74LS74 D-type flip-flop. The flip-flop output is used as a flag to tell the computer to begin sampling.

Besides adding the extra hardware, we will need another subroutine to handle the trigger flag. Figure 7-6 shows how the BASIC program is modified to use the new subroutine. Line 45 BLOADS the machine-language subroutine TRIGCK, which checks for the TRIGGER FLAG. Line 65 tells BASIC where TRIGCK is located, and line 110 calls TRIGCK instead of calling FILBUF. The remainder of the BASIC program is identical to that of figure 7-2. All lines beyond 500 are omitted from this listing for simplicity, but they are identical to those in figure 7-2. Also, the machine-language program FILBUF is used as before.

Now let's take a look at figure 7-7, the assembly-language listing for the TRIGCK subroutine. This subroutine is called by line 110 of the BASIC driver instead of FILBUF being called. TRIGCK arms the trigger circuit by clearing the trigger flag. The processor does this by writing to address C0C1 in line 18. If you examine the hardware diagram of figure 7-5, you will see that when $\overline{\text{DEVICE SELECT}}$ is active LOW and A1 is HIGH, which is the case

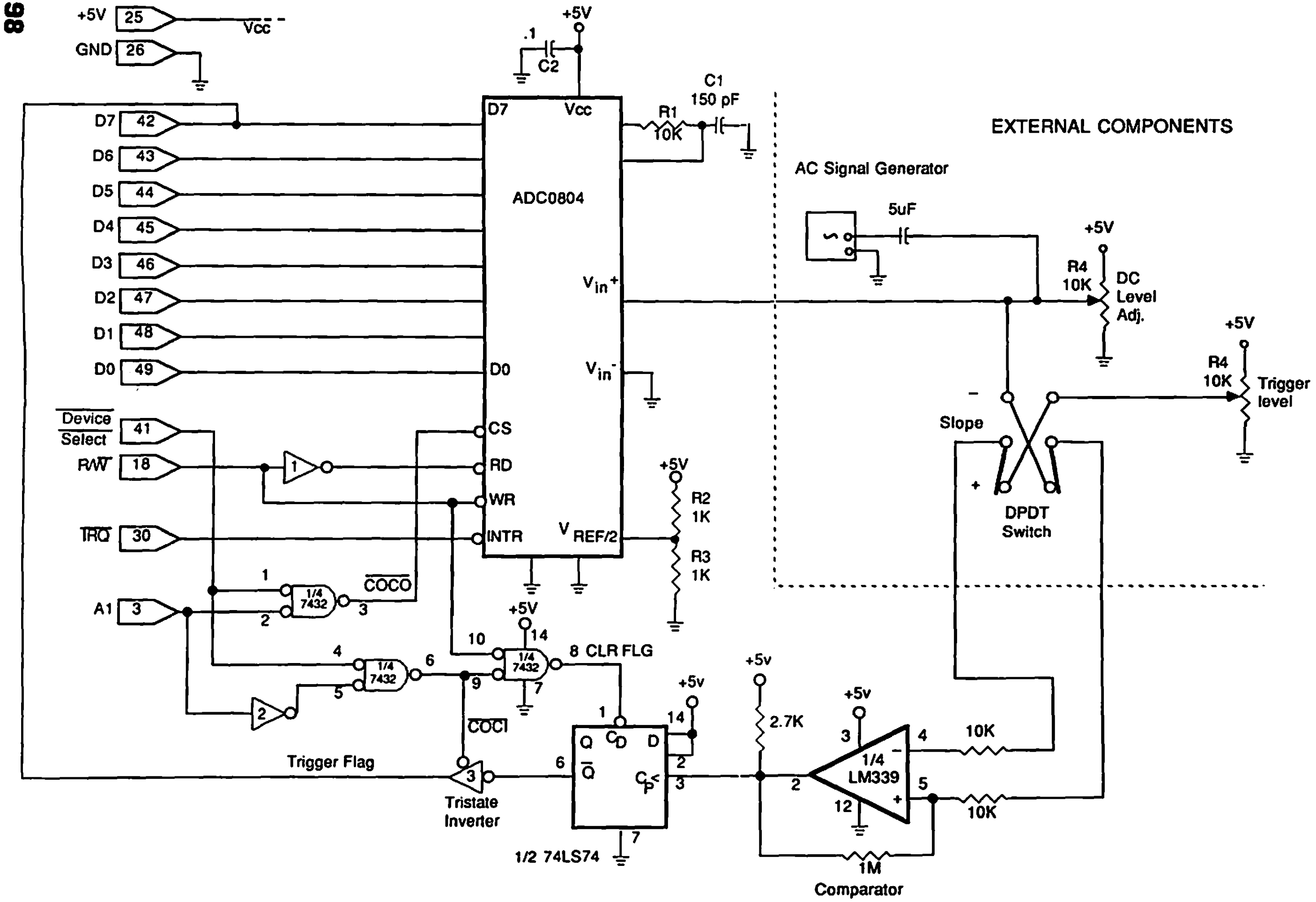


Figure 7-5
Triggered-sweep modifications

```
]LIST

10  REM  ********************
11  REM  * TRIGGERED SWEEP  *
12  REM  *   STORAGE SCOPE  *
14  REM  *  J. OLEKSY 1984  *
18  REM  ********************
19  REM
20  REM  *** INIT ***
21  REM
30 D$ = "": REM  CTRL-D
40  PRINT D$"BLOAD FILBUF"
45  PRINT D$"BLOAD TRIGCK"
50 BUFFER = 3072
60 FILBUF = 768
65 TRIGCK = 816
70  POKE 1022,32: POKE 1023,03: REM    * INTERRUPT VECTOR *
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  CALL TRIGCK: REM  * GET SAMPLES *
120  HGR : HCOLOR= 7
130  GOSUB 700: REM  * PLOT AXES *
140  GOSUB 500: REM  * PLOT WAVEFORM *
150  GOSUB 800: REM  * GET USER INPUT *
160  GOTO 110: REM  * REPEAT LOOP *
499  REM
500  REM  *** PLOT WAVEFORM ***
```

Figure 7-6
Storage scope listing with triggered-sweep modifications

when the processor writes to C0C1, the output at pin 6 of the 7432 goes LOW. Also, since $\overline{\text{WR}}$ is active LOW, the output at pin 8 of the 7432 goes LOW, thus clearing the flip flop.

Next, the listing of TRIGCK shows that the processor polls the TRIGGER FLAG by loading the accumulator from FLAG (C0C1). Note that when the processor reads from C0C1, pin 6 of the 7432 enables the tristate inverter (3), which places the complement of the $\overline{\text{Q}}$ side of the flip flop on data line D7. The processor reads the data bus and, if bit 7 is not set reads the flag over and over again. When the input signal causes the fip flop to set, the TRIGGER FLAG will go HIGH. When the processor reads a HIGH TRIGGER FLAG, it takes the next instruction from line 21, which causes a jump to subroutine FIBUF. FILBUF works exactly the same as before. After 255 samples of the input waveform are stored, FILBUF returns to TRIGCK, which then returns to BASIC. Of course to use TRIGCK, you must first BSAVE it on disc.

```
SOURCE FILE: TRIGCK
0000:               ************************
0000:               *                      *
0000:               *       TRIGCK         *
0000:               *                      *
0000:               *  POLLS FOR TRIGGER   *
0000:               *       FLAG           *
0000:               *                      *
0000:               *  DESTROYS:NOTHING    *
0000:               *  CALLS: FILBUF       *
0000:               *  OUTPUTS:NOTHING     *
0000:               *                      *
0000:               ************************
----- NEXT OBJECT FILE NAME IS TRIGCK.OBJ0
0330:                       ORG  $330
C0C1:               FLAG    EQU  $C0C1
0300:               FILBUF  EQU  $300
0330:08                     PHP
0331:48                     PHA
0332:8D C1 C0       CLRFLG  STA  FLAG       ;RESET TRIGGER FLAG
0335:AD C1 C0       POLL    LDA  FLAG       ;FLAG SET?
0338:10 FB                  BPL  POLL       ;NO-POLL AGAIN
033A:20 00 03               JSR  FILBUF     ;YES-GET SAMPLES
033D:68                     PLA
033E:28                     PLP
033F:60                     RTS              ;BACK TO BASIC

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 7-7
Assembly language listing for TRIGCK

Referring back to figure 7-5 for a moment, the three inverters could all be in the same package, for example, parts of a 74LS240 or a similar chip. The only one that must be tristate is inverter 3, whose output is connected to the data bus. The ac signal generator, slope switch, dc level adjust, and trigger level pot must all be mounted outside the computer. The remainder of the circuitry can be mounted on the card in slot 4. The hardware will work in any available slot, but remember to change the addresses of FLAG and CNVRTR by changing the EQUates in the FILBUF and TRIGCK routines if you use a different slot. With the hardware and software given in these examples, input signals with frequencies up to a few hundred hertz can be stored and displayed.

CHAPTER 8

Using the Computer as a Waveform Generator

In chapter 7 we saw how to read in many instantaneous values of voltage from an analog-to-digital converter so that we could "store" an input waveform in memory. Now we will see how to use the computer to generate a real-time waveform by using a digital-to-analog converter. The DAC produces a dc output voltage that is proportional to its binary inputs. The process of outputing dc voltages or real-time waveforms has many applications in testing and sound synthesis.

8-1 USING A DIGITAL-TO-ANALOG CONVERTER

A digital-to-analog converter (also called a *D/A* converter or *DAC*) is used whenever we want to generate a voltage proportional to some digital value. One simple off-the-shelf DAC is the DAC0801, whose diagram is shown in figure 8-1. The DAC0801 has 8 digital inputs and, therefore, generates a dc output voltage corresponding to the digital inputs with a resolution of 1 part in 256. It has a fast settling time of about 100 nanoseconds.

The actual value of output voltage depends on the way the chip is wired. When wired as shown, the output voltage is adjustable from 0v to a full-scale value up to 10v or so with binary inputs from 00 to $FF, respectively. A simple hookup to the computer is obtained by driving the DAC inputs from the outputs of a latch located in any suitable slot. The latch is needed to keep the digital inputs of the DAC constant while the computer does other things. By POKEing various values to the DAC, corresponding values of dc output voltage will be developed at the output (pin 6) of the 741 op amp. The 0801 output (pin 4) actually produces an *output current* that is proportional to the digital inputs. Therefore, the 741 op amp is used to convert the output current into an *output voltage*.

Before we can use the circuit, we must calibrate the DAC's output for zero and full scale. Let's assume that the DAC's inputs are connected to the outputs of a latch chip in slot 4. (See chapter 2 for the latch circuit.) To zero the output we simply type

```
POKE 49345, 0
```

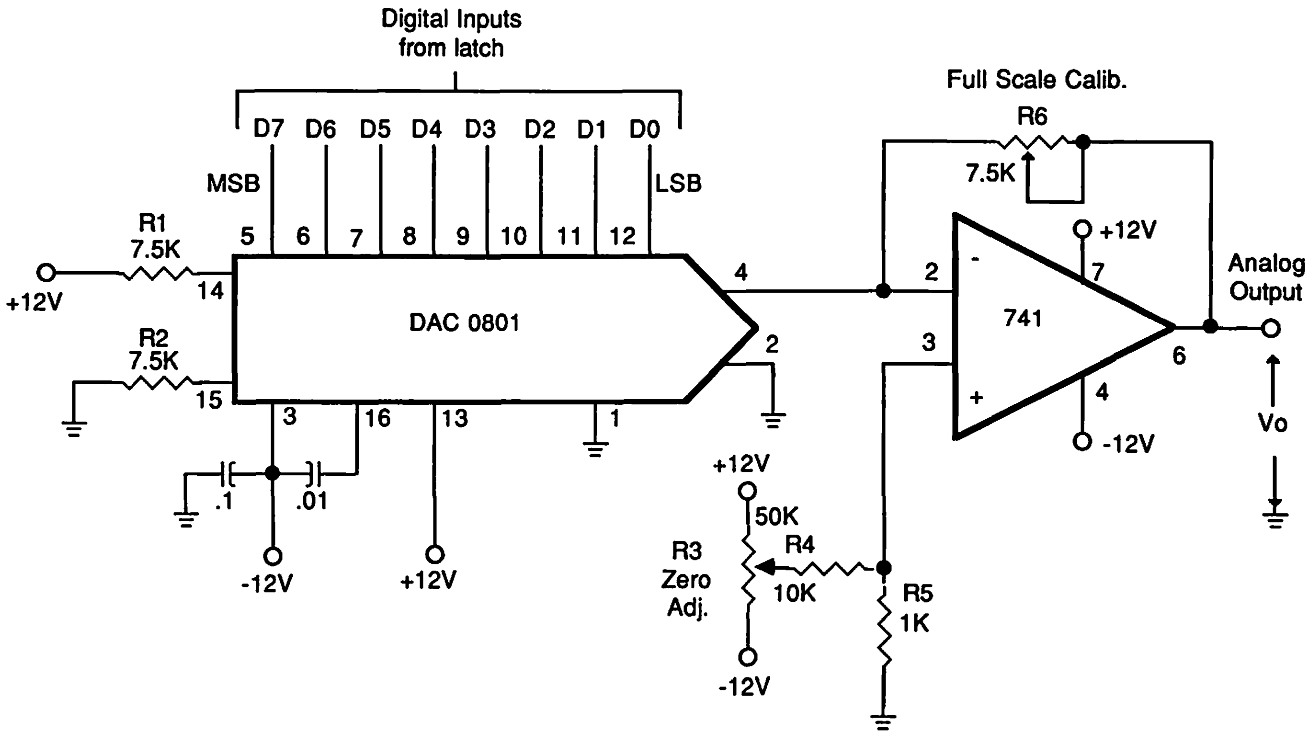


Figure 8-1
Digital-to-analog converter

```
]LIST

5  REM  *** OUTPUTTING DC LEVELS ***
10  LET DAC = 49345
20  INPUT "ENTER DESIRED OUTPUT VOLTAGE (0-5V)";V
25  PRINT
30  LET BYTE = V * 255 / 5
40  POKE DAC,BYTE
50  GOTO 20
```

Figure 8-2
Listing for outputting dc levels with DAC in slot 4

Then, using a voltmeter or scope connected to pin 6 of the 741, we adjust the zero adjust pot R3 until the output reads zero volts.

To calibrate for full scale we type

```
POKE 49345, 255
```

We will now see the output voltage rise to some positive value. We simply adjust the full-scale calibration pot R6 until the output reads exactly the full-scale value we want, say +5v. After adjusting the full-scale value, we should go back and check zero again and, finally, adjust the full-scale value again since the two adjustments are interactive. The calibration procedure is the same regardless of the full-scale value you want. Using the zero and full-scale adjustments as shown ensures accuracy without having to use precision resistors.

Let's assume that you have the DAC circuit adjusted for a full-scale output of +5v. The program of figure 8-2 can be used to output any desired dc voltage from 0 to +5v. Line 30 scales the value of BYTE for your selected dc output voltage. Of course, you must modify this line if you decide to use a different full-scale value.

One application of this circuit could be to use the dc voltage to control the speed of a dc motor, say for a chart recorder. For example, the user could be asked to input the desired chart speed, say in millimeters/second. Then the program could convert the user's input into an appropriate digital value and send it to the DAC, which in turn drives a power amp connected to the chart-drive motor. Once the speed byte is latched in, the computer is free to do other things.

```
]LIST

10  REM  *** SAWTOOTH WAVE GENERATOR ***
20  LET DAC = 49345
30  FOR LEVEL = 0 TO 255
40  POKE DAC,LEVEL
50  NEXT LEVEL
60  GOTO 30
```

Figure 8-3
Listing to generate sawtooth waves

Rather than outputing constant dc values, we can generate a staircase waveform by periodically outputing higher and higher values to the DAC. A simple program to do this is shown in figure 8-3. Or we can easily change the saw-tooth waveform into a symmetrical, triangular wave by adding the following lines:

```
52 FOR LEVEL = 254 TO 1 STEP -1
54 POKE DAC, LEVEL
56 NEXT LEVEL
```

Finally, we can stretch out each step of the waveform by adding a suitable time delay after each POKE command.

In general, the DAC allows us to output a time-varying voltage or a dc voltage, whose value depends on a byte that we store in an output latch.

8-2 USING BASIC TO GENERATE SINE WAVES

Besides simple staircase waveforms, we can generate practically any desired waveform to output through the DAC. All we have to do is use the powerful mathematical capabilities of BASIC to calculate the instantaneous values of the waveform then output these values to the DAC.

For example, we can generate a sine wave by calculating a number of instantaneous values of the waveform using the equation

V = A * SIN(T)

where V = instantaneous value
A = maximum amplitude of the wave
T = angle in radians

All we have to do is put the calculation in a loop in which the value of T varies in small steps from 0 to 2 pi radians. As we calculate each value for V, we store it in memory to be recalled later, much like we did in the Storage Scope program.

Figure 8-4 shows the listing of such a program. It calculates 100 instantaneous values for V and stores them in a buffer. After all values have been stored, the plotting routine is called, which takes the values from the buffer and displays them on the screen. This program does not send any values to the DAC, but it does demonstrate the mathematical construction of a sine wave.

Notice that most of this program is identical to the Storage Scope program of figure 7-2. To change the Storage Scope program into a Sine Wave Generator program:

1. Load the Storage Scope program into the computer.
2. Delete lines 10 through 70.
3. Retype new lines 10 and 50.
4. Change line 110 as shown.
5. Change the Plot Waveform subroutine (line 500) as shown.
6. Change line 720 as shown.
7. Add the new subroutine that starts at line 1000.

(Notice how easy it is to use a library of existing subroutines to build new applications programs. In this example, we used much of a program already stored on disc. All we did was to add a little and change a little.)

When you run the program, you will be asked to enter a maximum value for the sine wave. After hitting RETURN, you will notice a pause while the computer calculates 100 values for the sine wave and POKEs them into the buffer. Then you will see the waveform plotted on the CRT screen.

The User Input subroutine at line 800 gives the viewer the option of viewing the waveform as long as desired, of generating a new waveform, or of quitting the program.

The Sine Wave Calculation subroutine (line 1000) takes the user input for maximum amplitude of the sinewave and calculates

```
]LIST 0,999

10  REM  *** SINEWAVE GENERATOR ***
50 BUFFER = 768
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  GOSUB 1000: REM  * SINWAVE CALCULATIONS *
120  HGR : HCOLOR= 7
130  GOSUB 700: REM  * PLOT AXES *
140  GOSUB 500: REM  * PLOT WAVEFORM *
150  GOSUB 800: REM  * GET USER INPUT *
160  GOTO 110: REM  * REPEAT LOOP *
499  REM
500  REM  *** PLOT WAVEFORM ***
501  REM
505 START = 0:FINISH = 99
510 MEM = BUFFER
520 X = START
530 Y = 159 -  PEEK (MEM) * .623
540  HPLOT X,Y
550  FOR X = START TO FINISH
560 Y = 159 -  PEEK (MEM) * .623
570  HPLOT  TO X,Y
580 MEM = MEM + 1
583  NEXT X
585  IF START = 100 THEN  RETURN
590 START = 100:FINISH = 199:MEM = BUFFER
592  GOTO 550
595  RETURN
699  REM
700  REM  *** PLOT AXES ***
701  REM
710  FOR X = 0 TO 250 STEP 10
720  HPLOT X,159
730  NEXT X
740  FOR Y = 0 TO 160 STEP 10
750  HPLOT 0,Y
760  NEXT Y
770  RETURN
799  REM
800  REM   *** USER INPUT ***
801  REM
810  INVERSE
815  VTAB 23
820  PRINT "HIT SPACEBAR FOR NEW TRACE"
830  PRINT : PRINT "TYPE Q TO QUIT"
840  GET A$
850  PRINT : PRINT : PRINT
860  NORMAL
870  IF A$ = "Q" GOTO 890
880  RETURN
890  TEXT : HOME : END
999  REM
```

Figure 8-4
Listing for sine wave generator

```
]LIST 1000,2000

1000  REM  *** SINEWAVE CALCULATIONS ***
1001  REM
1010 MEM = BUFFER
1020  PRINT "PEAK-TO-PEAK AMPLITUDE? (0-5V)
1030  INPUT A1
1040  FOR T = 0 TO 360 / 57.2958 STEP 360 / 5729.58
1050 V = 128 + 25.4 * A1 *  SIN (T)
1060 V =  INT (V)
1070  POKE MEM,V
1080 MEM = MEM + 1
1090  NEXT T
1095  RETURN
```

Figure 8-4 *(continued)*

100 values for V in steps of 3.6 degrees. You can use different increments to plot more or less than 100 points. Line 1050 scales and offsets the value for V so that plotting will begin at the middle of the plotting screen when V = 0v.

Now that we can calculate and store sufficient instantaneous values to construct a sine wave, let's add a subroutine to output these values to a DAC. Figure 8-5 shows the new program.

Lines 820 and 830 prompt the user to select a new waveform or to generate a real-time waveform with the DAC. The user thus has an opportunity to see the waveform on the computer CRT before outputing it. When the user is satisfied with the shape, he/she types R to go to the real-time generation mode.

The Real-Time Generation subroutine (line 200) causes values to be taken from the memory buffer and sent out to the DAC in a continuous loop. The DAC's output is the sine wave riding on a dc level of +2.5v. This assumes that the DAC is adjusted for a full-scale output of +5v, as described in section 8-1. You can vary the zero level and full-scale output level of the circuit of figure 8-1 by adjusting the two pots.

To stop sine wave generation, type CTRL-C, which will stop the program.

```
]LIST 0,799

10  REM  *** REALTIME SINEWAVE GENERATOR ***
50 BUFFER = 768
60 DAC = 49345
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  GOSUB 1000: REM  * SINWAVE CALCULATIONS *
120  HGR : HCOLOR= 7
130  GOSUB 700: REM  * PLOT AXES *
140  GOSUB 500: REM  * PLOT WAVEFORM *
150  GOSUB 800: REM  * GET USER INPUT *
155  IF A$ = "R" THEN  GOSUB 200: REM  * BEGIN REALTIME GEN *
160  GOTO 110: REM  * REPEAT LOOP *
199  REM
200  REM  *** REALTIME GENERATION ***
201  REM
210  FOR MEM = BUFFER TO BUFFER + 99
220 V =  PEEK (MEM)
230  POKE DAC,V
240  NEXT MEM
250  GOTO 210
499  REM
500  REM  *** PLOT WAVEFORM ***
501  REM
505 START = 0:FINISH = 99
510 MEM = BUFFER
520 X = START
530 Y = 159 -  PEEK (MEM) * .623
540  HPLOT X,Y
550  FOR X = START TO FINISH
560 Y = 159 -  PEEK (MEM) * .623
570  HPLOT  TO X,Y
580 MEM = MEM + 1
583  NEXT X
585  IF START = 100 THEN  RETURN
590 START = 100:FINISH = 199:MEM = BUFFER
592  GOTO 550
595  RETURN
699  REM
700  REM  *** PLOT AXES ***
701  REM
710  FOR X = 0 TO 250 STEP 10
720  HPLOT X,159
730  NEXT X
740  FOR Y = 0 TO 160 STEP 10
750  HPLOT 0,Y
760  NEXT Y
770  RETURN
799  REM
```

Figure 8-5
Real-time sine wave generator

```
]LIST 800,2000

800  REM   *** USER INPUT ***
801  REM
810  INVERSE
815  VTAB 23
820  PRINT "HIT SPACEBAR FOR NEW TRACE"
830  PRINT : PRINT "TYPE R FOR REALTIME GENERATOR"
840  GET A$
850  PRINT : PRINT : PRINT
860  NORMAL
880  RETURN
999  REM
1000  REM   *** SINEWAVE CALCULATIONS.***
1001  REM
1010 MEM = BUFFER
1020  PRINT "PEAK-TO-PEAK AMPLITUDE? (0-5V)"
1030  INPUT A1
1040  FOR T = 0 TO 360 / 57.2958 STEP 360 / 5729.58
1050 V = 128 + 25.4 * A1 *  SIN (T)
1060 V =  INT (V)
1070  POKE MEM,V
1080 MEM = MEM + 1
1090  NEXT T
1095  RETURN
```

Figure 8-5 *(continued)*

8-3 GENERATING OTHER WAVEFORMS

We are not limited to generating only sine waves with our DAC. We can generate almost any desired waveform simply by using the appropriate equations to calculate the instantaneous values. Figure 8-6 shows the listing of a program that can generate sine waves, triangular waves, rectangular waves, or complex waves, according to the user's option.

Actually, a complex wave is any nonsinusoidal wave. But we are using the term *complex* here to specifically refer to a waveform constructed from a fundamental sine wave plus a variety of other sine waves that are harmonically related to the fundamental sine wave.

```
]LIST 0,699

10  REM  **********************
12  REM  * WAVEFORM GENERATOR *
14  REM  *   J. OLEKSY 1984   *
18  REM  **********************
50 BUFFER = 768
60 DAC = 49345
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  GOSUB 900: REM  * GET WAVEFORM CHOICE *
115  ON CHOICE GOSUB 1000,2000,3000,4000
120  HGR : HCOLOR= 7
130  GOSUB 700: REM  * PLOT AXES *
140  GOSUB 500: REM  * PLOT WAVEFORM *
150  GOSUB 800: REM  * GET USER INPUT *
155  IF A$ = "R" THEN  GOSUB 200: REM  * BEGIN REALTIME GEN *
160  GOTO 110: REM  * REPEAT LOOP *
199  REM
200  REM  *** REALTIME GENERATION ***
201  REM
210  FOR MEM = BUFFER TO BUFFER + 99
220 V =  PEEK (MEM)
230  POKE DAC,V
240  NEXT MEM
250  GOTO 210
499  REM
500  REM  *** PLOT WAVEFORM ***
501  REM
505 START = 0:FINISH = 99
510 MEM = BUFFER
520 X = START
530 Y = 159 -  PEEK (MEM) * .623
540  HPLOT X,Y
550  FOR X = START TO FINISH
560 Y = 159 -  PEEK (MEM) * .623
570  HPLOT  TO X;Y
580 MEM = MEM + 1
583  NEXT X
585  IF START = 100 THEN  RETURN
590 START = 100:FINISH = 199:MEM = BUFFER
592  GOTO 550
595  RETURN
699  REM
```

Figure 8-6
Listing for waveform generator

```
]LIST 700,1999

700  REM  *** PLOT AXES ***
701  REM
710  FOR X = 0 TO 250 STEP 10
720  HPLOT X,159
730  NEXT X
740  FOR Y = 0 TO 160 STEP 10
750  HPLOT 0,Y
760  NEXT Y
770  RETURN
799  REM
800  REM   *** USER INPUT ***
801  REM
810  INVERSE
815  VTAB 23
820  PRINT "HIT SPACEBAR FOR NEW TRACE"
830  PRINT : PRINT "TYPE R FOR REALTIME GENERATOR"
840  GET A$
850  PRINT : PRINT : PRINT
860  NORMAL
880  RETURN
899  REM
900  REM  *** GET WAVEFORM CHIOCE ***
901  REM
910  TEXT : HOME
920  PRINT : PRINT  TAB( 6)"*** WAVEFORM GENERATOR ***": PRINT : PF
930  PRINT "WAVEFORM TYPES": PRINT "-------- -----"
940  PRINT  TAB( 10)"(1)  SINEWAVE"
942  PRINT  TAB( 10)"(2)  TRIANGULAR WAVE"
944  PRINT  TAB( 10)"(3)  RECTANGULAR WAVE"
946  PRINT  TAB( 10)"(4)  COMPLEX WAVE"
948  PRINT : PRINT  TAB( 10)"(0)  EXIT PROGRAM"
950  PRINT : INPUT "CHOOSE ONE  ==> ";CHOICE
960  IF CHOICE = 0 THEN 990
970  IF CHOICE < 0 OR CHOICE > 4 THEN 900
980  RETURN
990  HOME : VTAB 10: PRINT "GOODBYE FOR NOW": END
999  REM
1000  REM  *** SINEWAVE CALCULATIONS ***
1001  REM
1010 MEM = BUFFER
1020  PRINT "PEAK-TO-PEAK AMPLITUDE? (0-5V)"
1030  INPUT A1
1040  FOR T = 0 TO 360 / 57.2958 STEP 360 / 5729.58
1050 V = 127 + 25.4 * A1 *  SIN (T)
1060 V =  INT (V)
1070  POKE MEM,V
1080 MEM = MEM + 1
1090  NEXT T
1095  RETURN
1999  REM
```

Figure 8-6 *(continued)*

```
]LIST 2000,3999

2000  REM  *** TRIANGULAR WAVE CALC ***
2001  REM
2010 MEM = BUFFER
2020  PRINT : INPUT "MAX AMPLITUDE? (0-5V)";V1
2030  INPUT "ENTER % POSITIVE DUTY CYCLE (1 TO 99)";P
2040 VM = V1 * 255 / 5
2045  REM  * POS SLOPE CALC *
2050  FOR N = 0 TO P
2060 V = VM * N / P
2070  POKE MEM,V
2080 MEM = MEM + 1
2090  NEXT N
2095  REM  * NEG SLOPE CALC *
2100  FOR M = 1 TO 100 - P
2110 V = VM - VM * (M / (100 - P))
2120  POKE MEM,V
2130 MEM = MEM + 1
2140  NEXT M
2150  RETURN
2999  REM
3000  REM  *** RECTANGULAR WAVE ***
3001  REM
3005 MEM = BUFFER
3010  PRINT : INPUT "ENTER POSITIVE DUTY CYCLE (1 TO 99)";DUTY
3020  INPUT "MAX AMPLITUDE? (0-5V) ";A1
3030 V1 = A1 * 255 / 5
3040  FOR N = 1 TO DUTY
3050  POKE MEM,V1
3060 MEM = MEM + 1
3070  NEXT N
3080  FOR M = DUTY + 1 TO 100
3090  POKE MEM,0
3100 MEM = MEM + 1
3110  NEXT M
3120  RETURN
3999  REM
```

Figure 8-6 *(continued)*

When using the COMPLEX WAVE subroutine, you will be asked to enter the amplitude of the fundamental, or lowest, frequency component of the waveform. Then you will enter the amplitude of each harmonic as a percentage of the fundamental amplitude. Keep in mind, however, that peak instantaneous amplitude of the total waveform must not be greater than +5v nor less

```
]LIST 4000,5000

4000  REM  *** COMPLEX WAVE ***
4001  REM
4005 MEM = BUFFER
4010  PRINT : INPUT "PEAK-TO-PEAK AMPLITUDE OF FUNDAMENTAL? (0-5V) ";A1
4020  PRINT : PRINT "YOU WILL NOW ENTER THE AMPLITUDES"
4022  PRINT "   OF HARMONICS 2 THROUGH 7"
4024  PRINT "       IF NOT PRESENT, ENTER 0"
4026  PRINT : PRINT "ENTER AMPLITUDE AS A PERCENTAGE OF"
4028  PRINT "   THE FUNDAMENTAL,  SUCH AS .5 FOR 50%, ETC."
4030  PRINT : INPUT "2ND HARMONIC AMPLITUDE? ";A2
4032  INPUT "3RD HARMONIC AMPLITUDE? ";A3
4034  INPUT "4TH HARMONIC AMPLITUDE? ";A4
4036  INPUT "5TH HARMONIC AMPLITUDE? ";A5
4038  INPUT "6TH HARMONIC AMPLITUDE? ";A6
4040  INPUT "7TH HARMONIC AMPLITUDE? ";A7
4050  PRINT : PRINT "CALCULATING .....(PLEASE WAIT)"
4060  REM  * BEGIN CALCULATIONS *
4070  FOR T = 0 TO 360 / 57.2958 STEP 360 / 5729.58
4080 V = 127 + 25.4 * A1 * ( SIN (T) + A2 *  SIN (2 * T) + A3 *  SIN (3 * T) + A
4 *  SIN (4 * T) + A5 *  SIN (5 * T) + A6 *  SIN (6 * T) + A7 *  SIN (7 * T))
4090 V =  INT (V)
4100  POKE MEM,V
4110 MEM = MEM + 1
4120  NEXT T
4130  RETURN
```

Figure 8-6 *(continued)*

than 0v or you will get an error message when trying to POKE that value into memory. You can omit any harmonics you do not want in the output. For example, if you want a wave composed of only odd harmonics, simply enter a 0 (zero) for the amplitude of each even harmonic.

The subroutine uses all harmonics up to the seventh. You can modify it to include more harmonics. The computer takes several seconds to do all the calculations because the equation gets rather long. But it is fun to play around with a variety of harmonics to see what the resultant waveform will look like. (See figure 8-7 for sample runs of this program.)

Notice that subroutines at lines 200, 500, 700, 800, and 1000 are exactly the same as in the listing of figure 8-5. All we have to do is modify the MAIN DRIVER and add subroutines at lines 900, 2000, 3000, and 4000. Once again, the point is that an existing applications program can be easily modified if it is made up of organized subroutines using a top-down approach.

```
]RUN

     *** WAVEFORM GENERATOR ***


WAVEFORM TYPES
-------- -----
         (1)  SINEWAVE
         (2)  TRIANGULAR WAVE
         (3)  RECTANGULAR WAVE
         (4)  COMPLEX WAVE

         (0)  EXIT PROGRAM

CHOOSE ONE  ==> 1
PEAK-TO-PEAK AMPLITUDE? (0-5V)
?4
HIT SPACEBAR FOR NEW TRACE

TYPE R FOR REALTIME GENERATOR
```

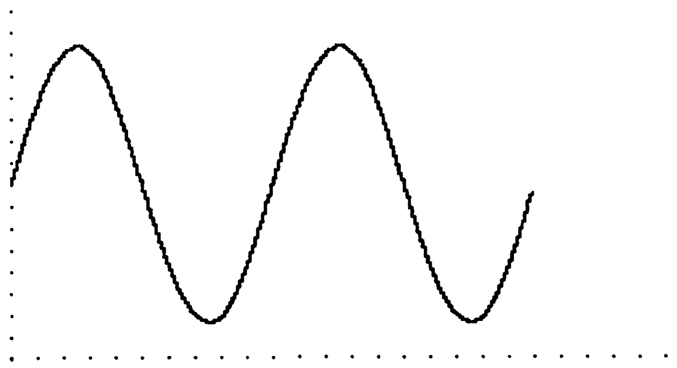


Figure 8-7
Sample runs of waveform generator
(A) Sine wave
(B) Triangular wave
(C) Rectangular wave
(D) Complex wave

```
]RUN

     *** WAVEFORM GENERATOR ***


WAVEFORM TYPES
-------- -----
         (1)  SINEWAVE
         (2)  TRIANGULAR WAVE
         (3)  RECTANGULAR WAVE
         (4)  COMPLEX WAVE

         (0)  EXIT PROGRAM

CHOOSE ONE  ==> 2

MAX AMPLITUDE? (0-5V)3
ENTER % POSITIVE DUTY CYCLE (1 TO 99)80
HIT SPACEBAR FOR NEW TRACE

TYPE R FOR REALTIME GENERATOR
```

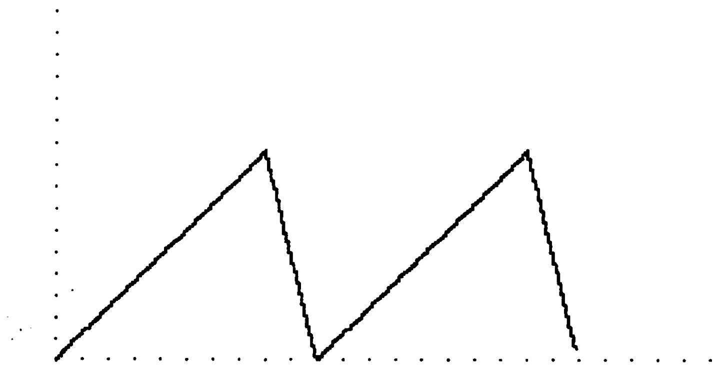


Figure 8-7 *(continued)*

```
]RUN

     *** WAVEFORM GENERATOR ***


WAVEFORM TYPES
-------- -----
         (1)  SINEWAVE
         (2)  TRIANGULAR WAVE
         (3)  RECTANGULAR WAVE
         (4)  COMPLEX WAVE

         (0)  EXIT PROGRAM

CHOOSE ONE  ==> 3

ENTER POSITIVE DUTY CYCLE (1 TO 99)25
MAX AMPLITUDE? (0-5V) 4.5
HIT SPACEBAR FOR NEW TRACE

TYPE R FOR REALTIME GENERATOR
```

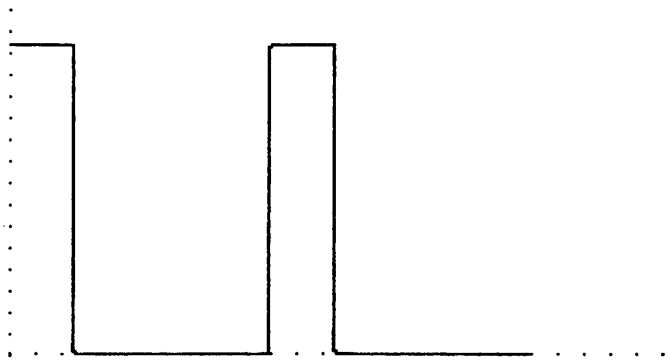


Figure 8-7 *(continued)*

```
]RUN

     *** WAVEFORM GENERATOR ***


WAVEFORM TYPES
-------- -----
         (1)  SINEWAVE
         (2)  TRIANGULAR WAVE
         (3)  RECTANGULAR WAVE
         (4)  COMPLEX WAVE

         (0)  EXIT PROGRAM

CHOOSE ONE  ==> 4

PEAK-TO-PEAK AMPLITUDE OF FUNDAMENTAL? (0-5V) 3

YOU WILL NOW ENTER THE AMPLITUDES
   OF HARMONICS 2 THROUGH 7
      IF NOT PRESENT, ENTER 0

ENTER AMPLITUDE AS A PERCENTAGE OF
   THE FUNDAMENTAL,  SUCH AS .5 FOR 50%, ETC.

2ND HARMONIC AMPLITUDE? 0
3RD HARMONIC AMPLITUDE? .333
4TH HARMONIC AMPLITUDE? 0
5TH HARMONIC AMPLITUDE? .2
6TH HARMONIC AMPLITUDE? 0
7TH HARMONIC AMPLITUDE? .14

CALCULATING .....(PLEASE WAIT)
HIT SPACEBAR FOR NEW TRACE

TYPE R FOR REALTIME GENERATOR
```

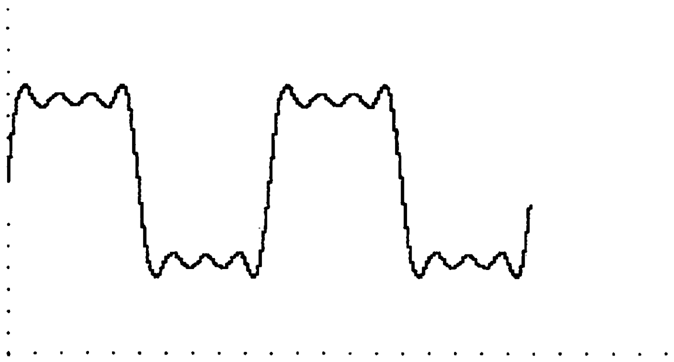


Figure 8-7 *(continued)*

8-4 GENERATING HIGHER FREQUENCIES

By now you will have noticed that the real-time waveforms we can generate are all quite low in frequency. But, as you have probably guessed, we can increase the output frequency by using a machine-code subroutine to output our instantaneous values to the DAC.

Figure 8-8 shows the disassembled machine-code (MC) listing for an output routine. Using this routine, you can output any of the four types of waveforms at frequencies of about 660 Hz. The MC routine must first be loaded into memory, then it must be called at the appropriate time. You can easily modify the program in figure

```
*384L

0384-    A2 00        LDX    #$00
0386-    BD 00 03     LDA    $0300,X
0389-    8D C1 C0     STA    $C0C1
038C-    E8           INX
038D-    E0 64        CPX    #$64
038F-    D0 F5        BNE    $0386
0391-    4C 84 03     JMP    $0384
0394-    00           BRK
0395-    00           BRK
0396-    00           BRK
0397-    00           BRK
0398-    00           BRK
0399-    00           BRK
039A-    00           BRK
039B-    00           BRK
039C-    00           BRK
039D-    00           BRK
039E-    00           BRK
039F-    00           BRK
03A0-    00           BRK
```

Figure 8-8
Disassembled listing of MC routine to output higher frequencies

8-6 to use the MC routine (assuming that the MC routine is already loaded in memory) by adding the following lines:

```
70 FASGEN = 900
205 CALL FASGEN
```

Refer back to chapter 6 if you forgot how to load MC programs.

When you type R for the real-time generator, the MC routine will be called and will do the same job as the original BASIC subroutine at line 200, but much faster.

You can get even higher output frequencies if you are willing to accept lower resolution, that is, less points per wave. For example, by inserting four additional INX instructions following the one in location $38C, every fifth calculated point will be sent out to the DAC, giving a total of 20 points per waveform rather than 100. This will increase the output frequency to a little over 2 mHz.

Finally, you can slow down the output frequency by inserting a small time delay between each output instruction.

CHAPTER 9

Serial I/O

The fastest way to transfer a data byte from one point to another is to send the entire 8-bit byte all at once. In other words, transfer all 8 bits in parallel, like we did in earlier chapters. This scheme works well if the sender and the receiver are not too far apart. If they are far apart, transmission problems occur that we do not have to consider at close range. To avoid these transmission problems, *serial* (one bit at a time) transmission is used. In this chapter we will discuss the concepts of serial data transmission and typical serial I/O circuits used with computers.

9-1 COMMUNICATING WITH DISTANT DEVICES

If you have studied any communications theory, you probably have heard of transmission lines and impedance matching. Briefly stated, every transmission line (wire or cable used to send and receive signals) has a *characteristic impedance.* The characteristic impedance is a function of the physical geometry of the transmission line, that is, coax cable, ribbon lead, and so on, and includes the wire size and spacing as well. For example, ribbon cable used to connect a TV to its antenna typically has a characteristic impedance of 300 ohms.

An infinitely long transmission line will "look like" a pure resistance equal to its characteristic impedance, as seen by the driving source. While we do not work with infinitely long lines, the transmission line will still look like its characteristic impedance if the receiving end of the line is terminated in a load resistance equal to the characteristic impedance of the line. The line is then said to be properly *terminated,* or we say that the load is *matched* to the line impedance. If the load does not match the line impedance, some of the signal sent down the line may be reflected back and interfere with new signals being sent. The result is that the total signal seen at the receiving end is a composite of the reflected signal superimposed on the incoming signal. Of course, this makes the total signal different from what it should be and can cause problems of misinterpretation or false triggering.

So why didn't we have to consider matching impedances in any circuit we built thus far? It turns out that reflections only become a problem when the line is some appreciable fraction of a wavelength long, say ¼ wave or more. We can calculate the wavelength of a signal using the equation

$$W = V/F$$

where W = wavelength of the signal
V = velocity of wave (typically equal to the speed of light, or 300,000,000 meters/second)
F = frequency of signal

For example, what is the wavelength of a 60 Hz signal?

$$W = V / F = 300{,}000{,}000 / 60 = 5{,}000{,}000 \text{ meters,}$$
or about 3100 miles

So we do not have to worry about reflections until the line becomes several hundred miles long. But what if the signal frequency is 300 MHz? Then the wavelength becomes

$$W = V / F = 300{,}000{,}000 / 300{,}000{,}000 = 1 \text{ meter}$$

So a line even a fraction of a meter long will give us trouble if not properly terminated.

You might be thinking that we really do not use frequencies that high in our computer. But according to Fourier's theory, any wave can be shown to be made up of a fundamental signal and a number of harmonics. Square waves or pulses, for example, contain many many odd harmonics. So whenever we try to send a rectangular pulse down a transmission line, the fast leading and trailing edges of the pulse contain component frequencies that can easily extend upwards of 100 MHz. This means that if our transmission line is just a few meters long, we might have reflections that can interfere with the received signal unless the line is properly terminated.

Here is where the problem comes in. It is very difficult to match impedances in logic circuits. For one thing, the input impedance of a gate, for example, is not even a constant value, but it changes depending on whether the input signal is a HIGH or LOW level. However, we can avoid the difficulty by simply lowering the bandwidth of frequencies we transmit. We can do this by slowing

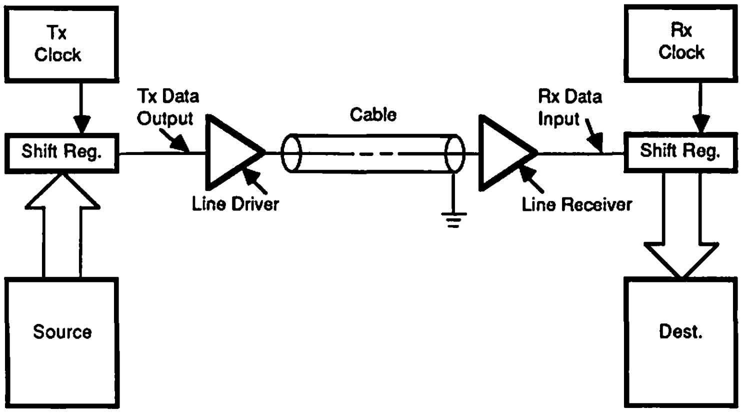


Figure 9-1
One-way serial communications link

down the rise and fall times of the pulses. And, if we stretch out each pulse for a longer duration, the pulse will still be quite rectangular, even though the rise and fall times are lowered drastically.

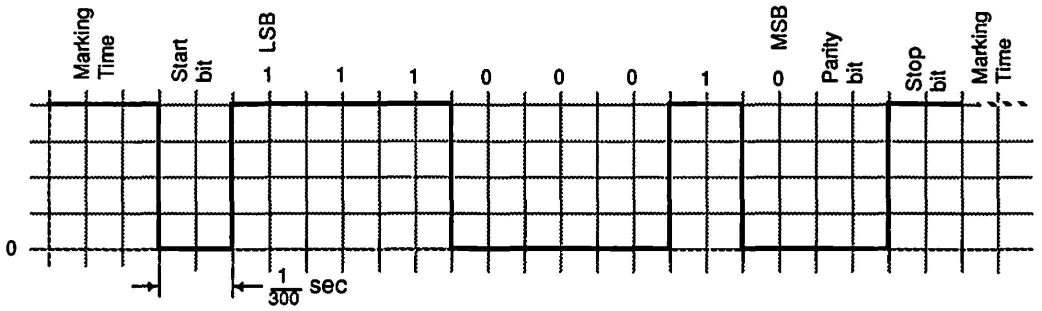
Rather than put circuitry on each of eight lines to lengthen the rise and fall times, and also tie up the computer while waiting for long duration pulses, the technique of serial data transmission was developed. The general idea is shown in figure 9-1. The source (some logic circuit) wants to send a byte to the destination (some other logic circuit). So the source transfers a byte in parallel to a shift register. The shift register is then clocked by the transmitter (Tx) clock, and the data byte shifts out one bit at a time. The Tx clock operates at some low frequency, typically 300 Hz, which makes each pulse at least 1/300 second long, rather than a fraction of a microsecond. The output of the serial-shift register feeds a *line driver,* which could simply be an op amp whose output rise and fall times (slew rates) are quite long compared to the rise and fall times of the logic circuits. The line driver feeds the transmission line, which may be several meters long. At the end of the transmission line is another linear circuit, called a *line receiver,* which could simply be another op amp or some other transistor amplifier circuit. (The input impedance of this receiver circuit is quite constant as opposed to the input impedance of a logic circuit.) The output of the line receiver feeds into a serial in-parallel out-shift register, which is clocked at the same rate as the transmitter shift register. So as the data bits are being clocked out of the transmitter, they are being clocked into the receiver. Once the received byte is in the receiver shift register, it is transferred in parallel into the destination circuit.

Although the transmission of each byte takes much longer this way, we are assured that the byte we send is the byte that will be received.

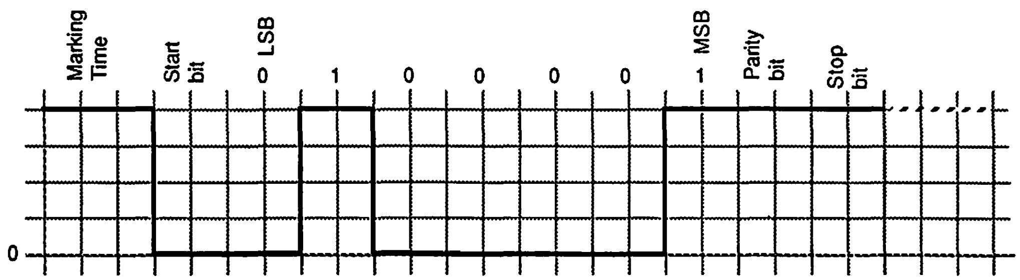
9-2 FORMAT FOR SERIAL TRANSMISSION

In order for the receiver to understand each transmitted character, a definite format must exist for each transmitted character. The commonly used format for asynchronous serial data transmission is shown in figure 9-2. Part A of the figure shows the waveform that would be observed at the transmitter output pin (shown as TxData in figure 9-1) if the circuit were transmitting the byte 47 hex, which is the ASCII code for the letter G. This form of transmission is called *asynchronous* because no clock or sync characters are sent with the data.

When the transmitter is powered up but is not sending any characters, it outputs a constant HIGH level. The transmitter is then said to be *marking time,* so the HIGH level is known as a *mark.* Then when transmission begins, the transmitter (serial-shift register) outputs a LOW *start* bit. Following the start bit, we see the 8 bits of the byte 47 hex, with the LSB leaving the shift register first as it would for a shift right register. Following the MSB, the transmitter inserts an even *parity* bit. Parity bits are used to ensure error-free communications. If *even* parity is used, as shown here, then the total number of logic HIGHs of the character plus the parity bit must be an *even* number. The receiver, of course, checks each received character for even parity. If it ever detects an *odd* number of logic HIGHs in any character, an error flag is set. Finally, the transmitter inserts a HIGH *stop* bit to signal the end of the character.



(A) Transmitting 47H at 300 Baud using 8 bits/character, even parity, and one stop bit.



(B) Sending ASCII Character B using 7 bits/character, odd parity, one stop bit.

Figure 9-2
Format for asynchronous serial transmission of characters

Once the character format is decided upon, each character must be transmitted exactly the same way. That is, each transmitted character must have a LOW *start* bit, followed by the 8 *character bits*

(LSB first), followed by an even *parity bit*, and terminated by a HIGH *stop* bit. Then the transmitter can either mark time, if no other characters are going to be immediately transmitted, or it can send another start bit for the next character.

Some features of the transmission format can be changed when setting up a system. For example, the system users may decide that 7 bits per character are sufficient, particularly for sending ASCII characters. Or odd parity may be chosen or perhaps no parity bit at all. Also, in older teletype (TTY) systems, 2 stop bits were required to allow for settling time. In the next topic, we will see how these options are chosen by the user. Two things that are not optional, however, are the first bit sent must be a LOW start bit and the last bit sent must be a HIGH stop bit. As another example, figure 9-2B shows the waveform you would see at the transmitter output if you were transmitting the ASCII character B (42 hex) using 7 bits per character, odd parity, and one stop bit.

Besides the requirement that the transmitter and the receiver must agree on the format for each character, the two must also agree on the frequency, or rate, at which the bits are being shifted. Again we have some options, but there are some standard bit rates (called *baud* rates) that are in common use. Older TTY systems commonly use 110 baud (that is 110 Hz) as the shifting frequency. Modems (modulator-demodulators) that are used to link computers to other computers over telephone lines typically use either 300 baud or sometimes 1200 baud for higher speed communications. The commonly used baud rates are 110, 150, 300, 600, 1200, 2400, 4800, 9600, and 19.2K baud.

The baud rate refers to the rate at which the bits are being shifted out of the transmitter (and into the receiver). For example, if you were transmitting a character having a start bit, 7 bits per character, even parity, and one stop bit, you would be sending a total of 10 bits per transmitted character. Then if you were transmitting at 300 baud, each bit would take 1/300 second to shift out. So the total time to transmit one complete character would be $10 \times 1/300 = 1/30$ second. In other words, you would be sending 30 characters per second. This is sometimes abbreviated 30 cps. On the other hand, at 9600 baud, the rate commonly used to link computers to programmable controllers, each character takes approximately one millisecond to transmit. The format is the same at higher baud rates as at lower rates, but long messages are transmitted much faster.

The clock signals for the transmitter and receiver are sometimes derived by dividing the system clock down to the proper frequency and sometimes obtained from separate crystal-controlled clock chips. These special chips, called *baud-rate generators,* usually have some method (like using jumper wires or switches) of obtaining any of the standard baud rates. We will use one of these baud-rate generators in a later section.

9-3 OFF-THE-SHELF UART

As you have probably guessed by now, in order to insert start bits, stop bits, and parity bits and shift the character out one bit at a time, several small-scale chips would be needed. In addition, at least an equal amount of hardware would be needed in the receiver section. Fortunately, IC manufacturers, such as Motorola and Intel, make a single chip *UART* (universal asynchronous receiver transmitter). The UART incorporates all of the necessary hardware to format and transmit or receive characters serially. In addition, it has several status flags that are used for error checking and other housekeeping chores. Motorola's UART is called an MC6850 Asynchronous Communications Interface Adaptor, or *ACIA*.

As with other Motorola interface chips, the 6850 is memory mapped into the system, which means that the MPU talks to the chip as though it were talking to a memory location. After initialization, whenever the MPU wants to send a byte to some receiver, the MPU simply stores a byte in the transmit data (TxData) register. The UART then takes care of formatting the character with start bit, stop bit, and parity bit and shifts it out at the proper rate. Meanwhile, the MPU is free to do other things. Likewise, when the UART detects an incoming character, it shifts the character in one bit at a time, checks for any errors, and informs the MPU that a character is present, either by generating an interrupt request or by setting a flag. The MPU can get the character (with start bit, parity bit, and stop bit already stripped off) by simply reading from the receive data (RxData) register, just as if it were reading from memory. Note that the MPU is not delayed by the slow incoming character.

Let's take a look at the 6850 chip. Figure 9-3A shows the diagram of the 6850 ACIA and figure 9-3B shows the pinouts. Notice that there are four internal registers which the MPU can access. These are the *transmit data, receive data, status,* and *control* registers. The transmitter data register and the control register are write-only registers, while the receive data and the status registers are read-only registers. For this reason, only two address locations are assigned to the UART. When the register select pin (RS) is LOW, the MPU talks to the control or status register. But when RS is HIGH, the MPU accesses the TxData or RxData register. The chip-select pins must all be active before the 6850 can be accessed.

Notice that like the 6821 PIA, the 6850 communicates with the MPU by means of the 8-bit data bus and that it has a few address lines and control lines which link it to the MPU. On the output side of the ACIA (right side), you will see the TxData output pin and the RxData input pin. In addition to these, there are three modem control/handshake pins called $\overline{RTS}$, $\overline{CTS}$, and $\overline{DCD}$. These control lines will be discussed in detail later.

Like the PIA again, the ACIA must be initialized before it can be used. That is, you must write a byte into its control register to tell the ACIA how you want it to operate. You must tell the ACIA what format you want to use for each character. That is, you must tell it how many bits per character, whether or not you want to use parity, and how many stop bits you want. Bits 4, 3, and 2 of the control byte are used for this purpose, as shown in table 9-1.

Bit 7 of the control byte must be set if you want the ACIA to generate an interrupt request when a received character is ready. If bit 7 is not set, no interrupt request will be generated, but a flag in the status register will still be set when a character comes in.

Bits 6 and 5 are used to control the transmitter interrupt request, as well as to make the $\overline{RTS}$ control line HIGH or LOW. This is also shown in table 9-1. A transmitter interrupt will be requested (if desired) when the TxData register is empty and the transmitter is ready to accept the next character. If you wish, rather than using an interrupt request, you can simply poll the status register to determine whether the TxData register is empty or not. We will discuss the status register in more detail later.

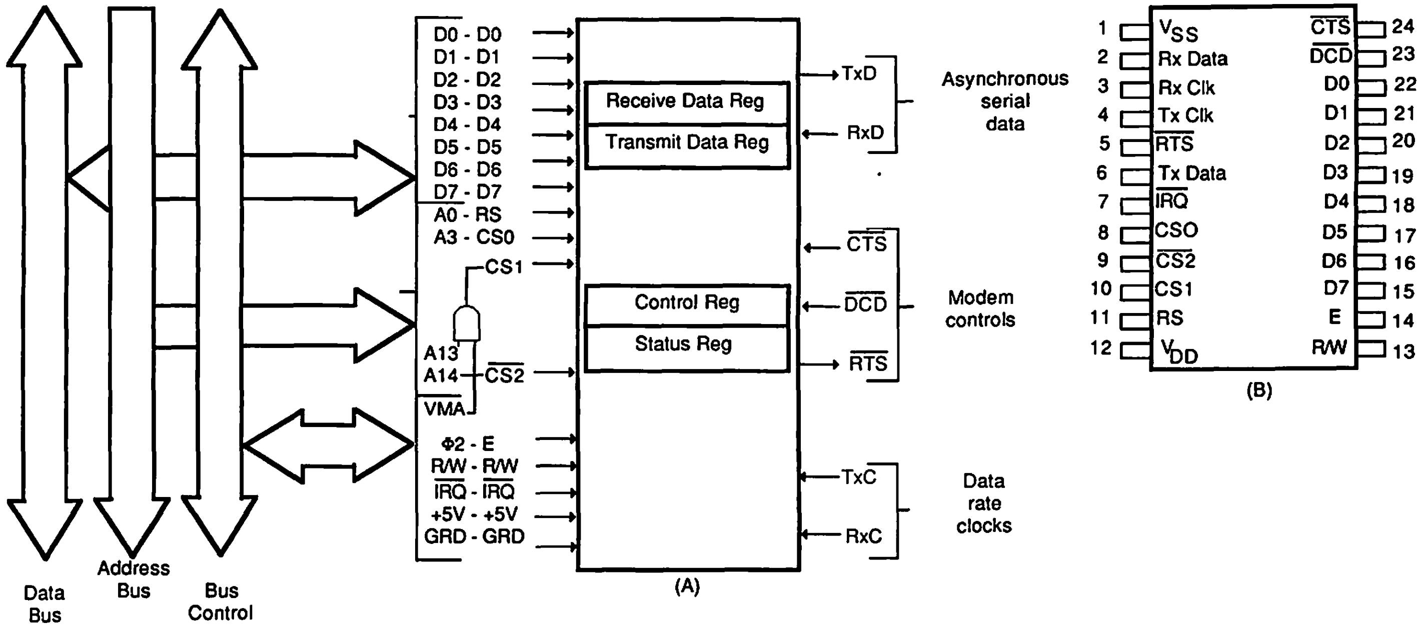


Figure 9-3
Motorola MC6850 ACIA

TABLE 9-1
ACIA Control Register Bits

Bit 7 Receive Interrupt Enable	*Bit 6 & Bit 5 Transmit Control*	*Bit 4, Bit 3, & Bit 2 Word Select*	*Bit 1 & Bit 0 Counter Divide Select*
0-Disabled	00-$\overline{RTS}$=low Transmit interrupt disabled	000 - 7 bits, even parity, 2 stop bits	00 - ÷ 1
1-Enabled	01-$\overline{RTS}$=low Transmit interrupt enabled	001 - 7 bits, odd parity, 2 stop bits	01 - ÷ 16
	10-$\overline{RTS}$=high Transmit interrupt disabled	010 - 7 bits, even parity, 1 stop bit	10 - ÷ 64
	11-$\overline{RTS}$=low Transmit interrupt disabled Transmits a break level on transmit data output	011 - 7 bits, odd parity, 1 stop bit	11 - Master reset
		100 - 8 bits, no parity, 2 stop bits	
		101 - 8 bits, no parity, 1 stop bit	
		110 - 8 bits, even parity, 1 stop bit	
		111 - 8 bits, odd parity, 1 stop bit	

Finally, bits 1 and 0, called the counter divide select bits, are used to reset the chip and to determine the clock divide ratio, as shown in table 9-1. Notice that there is no hardware reset pin on the ACIA. So to reset the chip, you simply write 1s into bit positions 1 and 0 of the control register. This allows you reset and reinitialize the ACIA without shutting down the computer, as would be the case for a hardware reset.

The clock divide ratio requires explanation. You probably have wondered how the transmitter and receiver clocks stay synchronized, since no sync signals are sent. Even if the two clocks operate on the same frequency, it would appear that a slight phase shift between the two might cause the receiver to try to shift in a bit near the rising or falling edge of a transition, thereby possibly getting an occasional false bit. This problem is solved by making the receiver sample clock operate at 16 times the baud rate. This is

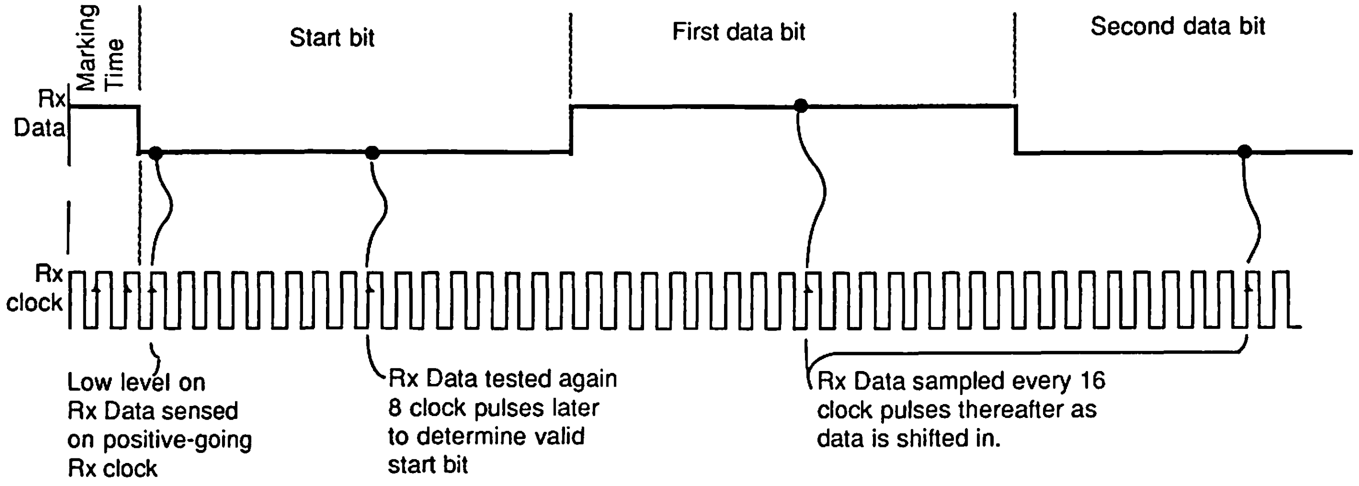

Marking Time
Start bit
First data bit
Second data bit
Rx Data
Rx clock
Low level on Rx Data sensed on positive-going Rx clock
Rx Data tested again 8 clock pulses later to determine valid start bit
Rx Data sampled every 16 clock pulses thereafter as data is shifted in.

Figure 9-4
Waveforms of RxData and RxClock

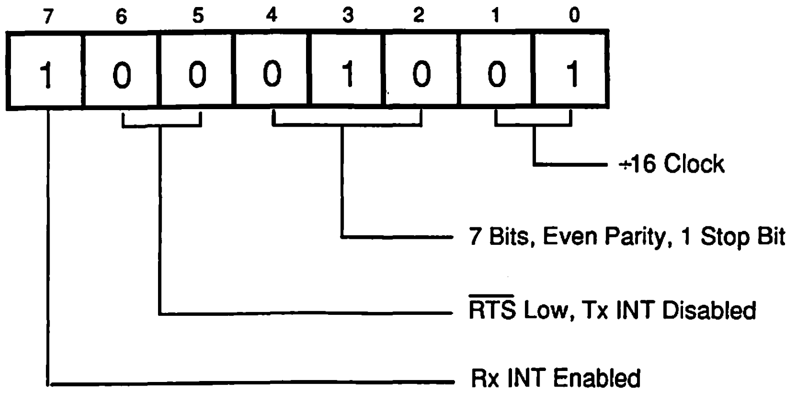
shown in figure 9-4. Let's assume that the transmitter is marking time, as shown in the figure. The receiver continually samples the incoming signal, waiting for a start bit. As long as the receiver senses a HIGH mark level, it continues to sample. But as soon as the receiver senses a LOW level on the incoming line, an internal divide-by-eight counter in the receiver is enabled, and the receiver ticks off eight of its own clock periods. The incoming signal is then sampled again. If the receiver still senses a LOW incoming level, it assumes it is a valid start bit and not just a noise pulse. The receiver then enables an internal divide-by-16 counter and ticks off 16 clock pulses. On the 16th pulse, and every 16th pulse thereafter, the receiver shifts in the incoming signal. Notice that *the incoming signal is thereby sampled near the center of the bit position.* By using this scheme, the receiver automatically resynchronizes its shifting rate for each new incoming character. In fact, using this method, even if the transmitter and receiver clocks were operating at slightly different frequencies (up to a few percent), good communications would still be maintained.

So in most applications, the Rx clock (coming from the baud-rate generator) operates at 16 times the actual baud rate (bit-shift rate). In other words, to receive at 300 baud, the Rx clock input frequency is set at $16 \times 300 = 4800$ Hz, and a clock divide ratio of 16 is chosen on initialization. The Tx clock and Rx clock are both divided by the clock divide ratio. The clock input pins are usually tied together and fed from a single baud-rate generator.

Let's now put together a control byte to set up the ACIA as follows: Rx Interrupt enabled, $\overline{\text{RTS}}$ LOW, Tx Interrupt disabled, 7 bits per character, even parity, one stop bit, and a divide-by-16 clock ratio. The control byte should be 89 hex, as shown in figure 9-5. Prior to sending this control byte, however, we must send the chip a master reset; that is, we should store 03 into its control register.

9-4 USING THE UART

Now let's discuss using the ACIA to send and receive characters with the Apple computer. Figure 9-6 shows how the ACIA can be hooked up to the Apple using one of the interface slots. Notice that



7 6 5 4 3 2 1 0
1 0 0 0 1 0 0 1
÷16 Clock
7 Bits, Even Parity, 1 Stop Bit
$\overline{\text{RTS}}$ Low, Tx INT Disabled
Rx INT Enabled

Figure 9-5
Control byte loaded into ACIA control register

the data lines and control lines connect to the MPU much like those of the PIA. For purposes of testing the ACIA without any line drivers or receivers, or without any other sender or receiver, jumper wires are connected between pins 6 and 2 and between pins 5, 23, and 24, as shown in figure 9-6. We will remove these jumpers later.

Note the use of the AY-5-8116 baud-rate generator chip. This chip has an on-board clock, which uses the externally connected 5.0688 MHz crystal. By making the T_A through T_D inputs HIGH or LOW according to table 9-2, we can generate any desired standard baud rate. We do this by means of a minidip switch mounted on the interface board. The actual output frequency of the AY-5-8116 chip is 16 times the desired baud rate, as was explained earlier. Let's assume that we have the switches set for a baud rate of 300, that is, switches 1 and 3 are open and switches 2 and 4 are closed. Let's also assume that we have the circuit of figure 9-6 built on a card plugged into slot 2.

Figure 9-7 shows the listing of a simple test program written entirely in BASIC. This program will allow us to transmit a single character over and over, so that we can view the TxData output pin with a scope to see if the transmitter is working properly. It will also allow us to test the receiver section of the UART to see if it is working.

Lines 20 through 35 equate the various registers of the ACIA with corresponding addresses of slot 2. Then line 40 resets the chip.

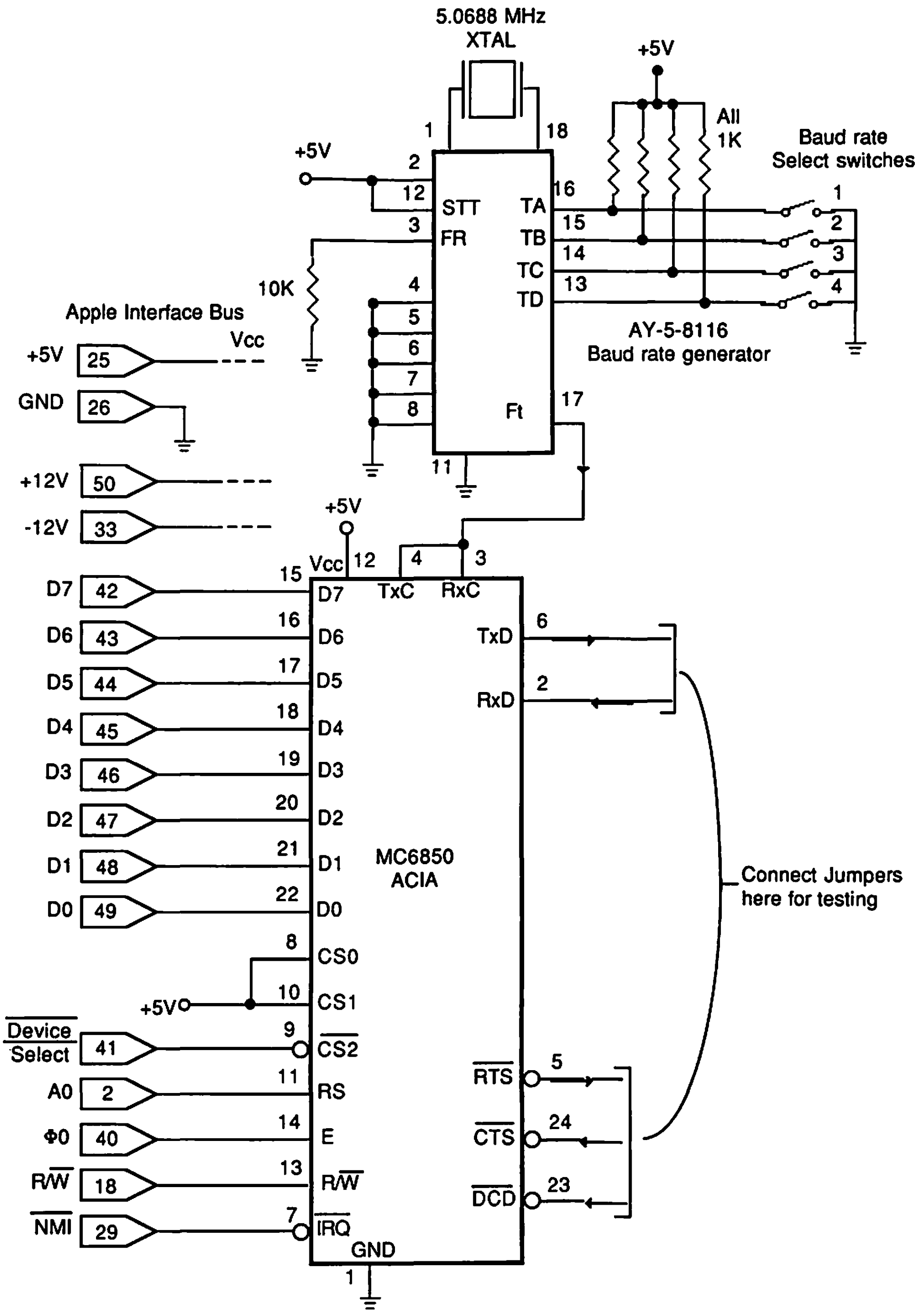


Figure 9-6
Hookup of ACIA to Apple

TABLE 9-2
Divisor Select Inputs for Baud-Rate Generator

Divisor Select *D C B A*	*Desired Baud Rate*
0 0 0 0	50
0 0 0 1	75
0 0 1 0	110
0 0 1 1	134.5
0 1 0 0	150
0 1 0 1	300
0 1 1 0	600
0 1 1 1	1200
1 0 0 0	1800
1 0 0 1	2000
1 0 1 0	2400
1 0 1 1	3600
1 1 0 0	4800
1 1 0 1	7200
1 1 1 0	9600
1 1 1 1	19200

Line 50 sets up a temporary buffer, called DBUF in RAM. We will make more of this buffer in a later version of the program.

Once in the MAIN DRIVER, we immediately go to a subroutine to initialize the ACIA, as was described in a previous section. Since the initialization is menu driven, all you have to do is choose the transmit and receive parameters you want to use. The subroutine will assemble the control byte for the control register and POKE it into the proper location.

Upon returning to the MAIN DRIVER, we go to the subroutine at line 2000, which asks you to input the character to be sent. All you have to do is hit any character on the keyboard and that character will be converted into ASCII code and POKEd into DBUF.

Next, we go to the subroutine at line 3000 and transmit the character. Note that lines 3005 and 3007 make up a polling routine to check whether or not the TxData register is empty and ready for the next character. Actually, only bit 1 of the status register should be checked. But, assuming that no other flags are set, the value of the flags will equal 2 when the TxData register is empty. Table 9-3

```
]LIST 0,1400

10  REM  *** UART TEST ***
11  REM
20  LET TRNSMT = 49313
25  LET RCVR = 49313
30  LET CTRLREG = 49312
35  LET STSREG = 49312
40  POKE CTRLREG,3: REM   RESET
50 DBUF = 768
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  GOSUB 1000: REM  * INIT *
120  GOSUB 2000: REM  * GET MESSAGE *
130  GOSUB 3000: REM  * TRANSMIT CHARACTER *
140  GOSUB 4000: REM  * RECEIVE CHARACTER *
150  GOTO 130
999  REM
1000  REM  *** USER SELECTIONS FOR UART INIT ***
1001  REM
1010  REM
1020  TEXT : HOME
1030  PRINT : PRINT  TAB( 6)"*** UART INITIALIZATION ***": PRINT : PRINT
1040  PRINT "USE VALUE WITH ASTERISK * FOR DEFAULT"
1050  PRINT
1060  PRINT : PRINT "RECEIVER INTERRUPT"
1070  PRINT "-------- ---------"
1080  PRINT
1090  PRINT  TAB( 4)"* (0)  DISABLED"
1100  PRINT  TAB( 6)"(1)  ENABLED"
1110  PRINT : INPUT "CHOOSE ONE ==>";C7
1120  PRINT : PRINT : PRINT "TRANSMIT CONTROL"
1130  PRINT "-------- -------"
1140  PRINT : PRINT  TAB( 1)"* (0)  RTS LOW, TX INT DISABLED"
1150  PRINT  TAB( 3)"(1)  RTS LOW, TX INT ENABLED"
1160  PRINT  TAB( 3)"(2)  RTS HIGH, TX INT DISABLED"
1170  PRINT  TAB( 3)"(3)  RTS LOW, TX INT DISABLED, BREAK "
1180  PRINT : INPUT "CHOOSE ONE ==> ";C6
1190  PRINT : PRINT : PRINT "WORD SELECT"
1200  PRINT "---- ------"
1210  PRINT : PRINT  TAB( 3)"(0)  7 BITS, EVEN PAR, 2 STOP BITS"
1220  PRINT  TAB( 3)"(1)  7 BITS, ODD PAR, 2 STOP BITS"
1230  PRINT  TAB( 1)"* (2)  7 BITS, EVEN PAR, 1 STOP BIT"
1240  PRINT  TAB( 3)"(3)  7 BITS, ODD PAR, 1 STOP BIT"
1250  PRINT  TAB( 3)"(4)  8 BITS, NO PAR, 2 STOP BITS"
1260  PRINT  TAB( 3)"(5)  8 BITS, NO PAR, 1 STOP BIT"
1270  PRINT  TAB( 3)"(6)  8 BITS, EVEN PAR, 1 STOP BIT"
1280  PRINT  TAB( 3)"(7)  8 BITS, ODD PAR, 1 STOP BIT"
1290  PRINT : INPUT "CHOOSE ONE ==>";C4
1300 BYTE = C7 * 128 + C6 * 32 + C4 * 4 + 1
1310  POKE CTRLREG,BYTE
1320  RETURN
```

Figure 9-7
UART test listing

```
]LIST 1400,5000

1999  REM
2000  REM  *** GET USER MESSAGE ***
2001  REM
2005  PRINT "ENTER CHARACTER TO BE SENT": PRINT
2010  GET C$
2020  POKE DBUF, ASC (C$): REM   * CONVERT CHARACTER TO ASCII *
2035  PRINT C$;
2045  PRINT : PRINT
2050  RETURN
2999  REM
3000  REM  *** TRANSMIT ONE BYTE ***
3001  REM
3005 FLAGS =  PEEK (STSREG): REM  * TXDATA EMPTY? *
3007  IF FLAGS < 2 GOTO 3005
3010 D =  PEEK (DBUF)
3020  POKE TRNSMT,D
3030  RETURN
3999  REM
4000  REM  *** RECEIVE ONE BYTE ***
4001  REM
4010 FLAGS =  PEEK (STSREG): REM  * RXDATA FULL? *
4020  IF FLAGS < 3 GOTO 4010
4030 R =  PEEK (RCVR)
4040 L$ =  CHR$ (R): REM  * CHANGE ASCII TO PRINTABLE CHARACTER *
4050  PRINT L$
4060  RETURN
```

Figure 9-7 *(continued)*

shows all of the flags in the status register. In this routine, we are only interested in bit 1. If bit 1 is set, we send the character to the TxData register and return.

Next, the subroutine at line 4000 is called, which polls the status register for RxData register full (flag bit 0). Line 4020 actually tests for TxData register empty and RxData full simultaneously. When a received character is found, it is read in, changed from ASCII to an Applesoft printable character, and printed on the CRT screen.

Upon returning to the MAIN DRIVER again, the transmit and receive operations are repeated endlessly. This allows you to observe the TxData output pin with a scope to see the format of the character. You should see a pattern similar to those in figure 9-2, depending on your choice of characters and transmit format. To stop the program, hit CTRL-C.

Figure 9-8 shows a sample run of the program in figure 9-7.

Now let's consider a more powerful and practical program, as shown in the listing of figure 9-9. This program, called SERIAL I/O, allows you to communicate with other computers using a serial

TABLE 9-3
ACIA Register Contents

	Buffer Address			
Data Bus Line Number	RS • $\overline{\text{R/W}}$ *Transmit Data Register (Write Only)*	RS • R/W *Receive Data Register (Read Only)*	$\overline{\text{RS}}$ • $\overline{\text{R/W}}$ Control Register (Write Only)	$\overline{\text{RS}}$ • R/W Status Register (Read Only)
0	Data bit 0*	Data bit 0	Counter divide select 1 (CR0)	Receive data register full (RDRF)
1	Data bit 1	Data bit 1	Counter divide select 2 (CR1)	Transmit data register empty (TDRE)
2	Data bit 2	Data bit 2	Word select 1 (CR2)	Data carrier detect ($\overline{\text{DCD}}$)
3	Data bit 3	Data bit 3	Word select 2 (CR3)	Clear to send ($\overline{\text{CTS}}$)
4	Data bit 4	Data bit 4	Word select 3 (CR4)	Framing error (FE)
5	Data bit 5	Data bit 5	Transmit control 1 (CR5)	Receiver overrun (OVRN)
6	Data bit 6	Data bit 6	Transmit control 2 (CR6)	Parity error (PE)
7	Data bit 7	Data bit 7	Receive interrupt enable (CR7)	Interrupt request (IRQ)

channel. You will notice that the set up and transmit portions of the program are very similar to those used in the figure 9-7 listing. But this program includes a machine-language interrupt service routine that is used whenever a character is received. The machine-language routine listing is shown in figure 9-10.

In general, the X register is used as the received character pointer (RECPTR), while the Y register is used as the displayed character pointer (DISPTR). Whenever a character is received, it is stored in a BUFFER. Then later, in the BASIC part of the program, the Y register is compared to the X register. If the two are equal, the computer knows that no new characters have to be displayed. But if Y does not equal X, it knows that it must display a character. The character is displayed by the subroutine at line 2000. Then the Y

```
]RUN

     *** UART INITIALIZATION ***


USE VALUE WITH ASTERISK * FOR DEFAULT


RECEIVER INTERRUPT
-------- ---------


   * (0)  DISABLED
     (1)  ENABLED

CHOOSE ONE ==>0


TRANSMIT CONTROL
-------- -------


* (0)  RTS LOW, TX INT DISABLED
  (1)  RTS LOW, TX INT ENABLED
  (2)  RTS HIGH, TX INT DISABLED
  (3)  RTS LOW, TX INT DISABLED, BREAK

CHOOSE ONE ==> 0


WORD SELECT
---- ------


  (0)  7 BITS, EVEN PAR, 2 STOP BITS
  (1)  7 BITS, ODD PAR, 2 STOP BITS
* (2)  7 BITS, EVEN PAR, 1 STOP BIT
  (3)  7 BITS, ODD PAR, 1 STOP BIT
  (4)  8 BITS, NO PAR, 2 STOP BITS
  (5)  8 BITS, NO PAR, 1 STOP BIT
  (6)  8 BITS, EVEN PAR, 1 STOP BIT
  (7)  8 BITS, ODD PAR, 1 STOP BIT

CHOOSE ONE ==>2
ENTER CHARACTER TO BE SENT

E

E
E
E
E
E
E

BREAK IN 4050
```

Figure 9-8
Sample run of UART test

```
]LIST 0,1110

10  REM  ********************
12  REM  *    SERIAL I/O    *
14  REM  *  J. OLEKSY 1984  *
16  REM  ********************
18  REM
20  LET TRNSMT = 49313
25  LET RCVR = 49313
30  LET CTRLREG = 49312
35  LET STSREG = 49312
40  POKE CTRLREG,3: REM  * RESET UART *
45  LET ERRFLAG = 253: POKE ERRFLAG,0
50  LET BUFFER = 24576
55  LET RECPTR = 254: POKE RECPTR,0
60  LET DISPTR = 255: POKE DISPTR,0
65  LET KYBD = 49152
80  POKE 1019,76: POKE 1020,00: POKE 1021,03: REM  * INT VECTOR *
85 D$ = "": REM  CTRL-D
90  PRINT D$"BLOAD SERIN.OBJ0"
95  LET TFLAG = 252
99  REM
100  REM  *** MAIN DRIVER ***
101  REM
110  GOSUB 1000: REM   * INIT UART *
115  HOME : PRINT  CHR$ (12): PRINT "READY": PRINT : PRINT
120 XREG =  PEEK (RECPTR):YREG =  PEEK (DISPTR)
130  IF YREG <  > XREG THEN  GOSUB 2000: REM  * DISPLAY A CHARACTER *
140 K =  PEEK (KYBD): REM   * KEY PRESSED? *
150  IF K > 127 THEN  GOSUB 3000: REM  * SEND A CHARACTER *
160  GOTO 120
999  REM
1000  REM  *** USER SELECTIONS FOR UART INIT ***
1001  REM
1020  TEXT : HOME : PRINT  CHR$ (12)
1030  PRINT : PRINT  TAB( 6)"*** UART INITIALIZATION ***": PRINT : PRINT
1040  PRINT "USE VALUE WITH ASTERISK * FOR DEFAULT"
1050  PRINT
1060  PRINT : PRINT "RECEIVER INTERRUPT"
1070  PRINT "-------- ---------"
1080  PRINT
1090  PRINT  TAB( 6)"(0)  DISABLED"
1100  PRINT  TAB( 4)"* (1)  ENABLED"
1110  PRINT : INPUT "CHOOSE ONE ==>";C7
```

Figure 9-9
Serial I/O program

```
]LIST 1120,5000

1120  PRINT : PRINT : PRINT "TRANSMIT CONTROL"
1130  PRINT "-------- -------"
1140  PRINT : PRINT  TAB( 1)"* (0)  RTS LOW, TX INT DISABLED"
1150  PRINT  TAB( 3)"(1)  RTS LOW, TX INT ENABLED"
1160  PRINT  TAB( 3)"(2)  RTS HIGH, TX INT DISABLED"
1170  PRINT  TAB( 3)"(3)  RTS LOW, TX INT DISABLED, BREAK "
1180  PRINT : INPUT "CHOOSE ONE ==> ";C6
1190  PRINT : PRINT : PRINT "WORD SELECT"
1200  PRINT "---- ------"
1210  PRINT : PRINT  TAB( 3)"(0)  7 BITS, EVEN PAR, 2 STOP BITS"
1220  PRINT  TAB( 3)"(1)  7 BITS, ODD PAR, 2 STOP BITS"
1230  PRINT  TAB( 1)"* (2)  7 BITS, EVEN PAR, 1 STOP BIT"
1240  PRINT  TAB( 3)"(3)  7 BITS, ODD PAR, 1 STOP BIT"
1250  PRINT  TAB( 3)"(4)  8 BITS, NO PAR, 2 STOP BITS"
1260  PRINT  TAB( 3)"(5)  8 BITS, NO PAR, 1 STOP BIT"
1270  PRINT  TAB( 3)"(6)  8 BITS, EVEN PAR, 1 STOP BIT"
1280  PRINT  TAB( 3)"(7)  8 BITS, ODD PAR, 1 STOP BIT"
1290  PRINT : INPUT "CHOOSE ONE ==>";C4
1300 BYTE = C7 * 128 + C6 * 32 + C4 * 4 + 1
1310  POKE CTRLREG,BYTE
1320  RETURN
1999  REM
2000  REM  *** DISPLAY A CHARACTER ***
2001  REM
2005 EF =  PEEK (ERRFLAG)
2007  IF EF = 255 THEN  INVERSE
2010 R =  PEEK (BUFFER + YREG)
2020 L$ =  CHR$ (R)
2030  PRINT L$;
2035  NORMAL
2040 YREG = YREG + 1: IF YREG = 256 THEN YREG = 0
2050  POKE DISPTR,YREG
2060  RETURN
2999  REM
3000  REM  *** SEND A CHARACTER ***
3001  REM
3010 FLAGS =  PEEK (STSREG): REM  * TDRE EMPTY? *
3020  IF FLAGS < 2 GOTO 3010
3025  POKE TFLAG,128
3030  GET C$
3040  POKE TRNSMT, ASC (C$)
3050  RETURN
```

Figure 9-9 *(continued)*

register is incremented, and the program returns to line 150 to scan the keyboard again. In this manner, the display pointer "chases" the received character pointer around in the BUFFER, which is 256 bytes wide.

Here is how the receiver portion of the program works. Line 80 POKEs a nonmaskable interrupt (NMI) vector into locations 1019, 1020, and 1021. Then line 90 loads the MC routine from disc. The user must choose option 1 for the receiver interrupt when presented with the menu. Then, whenever a character comes into the receiver,

```
]

SOURCE FILE: SERIN
----- NEXT OBJECT FILE NAME IS SERIN.OBJO
0300:              1          ORG  $300
6000:              2 BUFFER   EQU  $6000
C0A1:              3 TRNSMT   EQU  $C0A1
C0A1:              4 RCVR     EQU  $C0A1
C0A0:              5 STSREG   EQU  $C0A0
00FE:              6 RECPTR   EQU  $00FE
00FD:              7 ERFLAG   EQU  $00FD
00FC:              8 TFLAG    EQU  $00FC
0300:              9 *
0300:48           10 INTSERV  PHA                ;SAVE REGISTERS
0301:8A           11          TXA
0302:48           12          PHA
0303:A6 FE        13          LDX  RECPTR        ;GET POINTER
0305:AD A0 C0     14          LDA  STSREG        ;CHECK ERROR FLAGS
0308:29 70        15          AND  #$70
030A:D0 22        16          BNE  ERROR
030C:AD A1 C0     17 INCH     LDA  RCVR          ;NO ERRORS,GET CHAR
030F:9D 00 60     18          STA  BUFFER,X      ;SAVE IT
0312:A5 FC        19          LDA  TFLAG         ;SEE IF WE SENT CHAR
0314:30 0D        20          BMI  CLRFLG        ;IF SO, DON'T ECHO
0316:AD A0 C0     21 ECHO     LDA  STSREG        ;CHECK TDRE
0319:29 02        22          AND  #02
031B:F0 F9        23          BEQ  ECHO
031D:BD 00 60     24          LDA  BUFFER,X      ;GET CHAR BACK
0320:8D A1 C0     25          STA  TRNSMT        ;AND ECHO
0323:A9 00        26 CLRFLG   LDA  #0
0325:85 FC        27          STA  TFLAG
0327:E8           28          INX                ;POINT TO NEXT BUFFER LOC
0328:86 FE        29          STX  RECPTR        ;SAVE POINTER
032A:68           30          PLA                ;RESTORE REGISTERS
032B:AA           31          TAX
032C:68           32          PLA
032D:40           33          RTI                ;RETURN TO BASIC
032E:A9 FF        34 ERROR    LDA  #$FF
0330:85 FD        35          STA  ERFLAG
0332:4C 0C 03     36          JMP  INCH

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

Figure 9-10
Assembly listing of receive routine

a nonmaskable interrupt is generated. The Apple immediately jumps to location 1019 ($03FB) for its first instruction of the interrupt-service routine. The instruction at 1019 is a JMP to location $0300 for the first instruction.

Generally, the program remains in a loop within the MAIN DRIVER that continually checks whether any characters are to be displayed and whether a key has been pressed. Whenever a character is received, a nonmaskable interrupt is generated, which causes control to go over to the machine-code routine, which will get the character and put it in a BUFFER.

Examining the interrupt-service routine of figure 9-10, we see that first the accumulator and X registers are saved on the stack. Then the X register is loaded from memory location $00FE (called RECPTR) so as to point to a location in the received-character buffer. Then the contents of the status register (STSREG) are loaded into the accumulator and tested for error conditions. According to table 9-3, bits 4, 5, and 6 must be tested for error conditions.

Bit 4, *framing error,* will be set if the ACIA does not see a logic HIGH stop bit appearing in the proper number of bit positions from the LOW start bit, depending on the user's initialization. An error here indicates either a break condition or that the sender and receiver are using different formats or different baud rates.

Bit 5 is set if a previously received character was overwritten by another received character before being read by the receiver routine. Actually, the receiver of the ACIA is buffered so that when a character is received into the serial-shift register, it is loaded into another register where it can wait to be read by the MPU while another character is being shifted into the serial shift register. But if a second character is completely shifted in before the first character is read by the MPU, the first character is lost. The ACIA then sets the *receiver overrun* flag.

Bit 6 is a parity-error flag.

All three possible error conditions are checked, and if any error is found, the routine branches to an ERROR routine, which sets an error flag by storing $FF into memory location 253, which will be tested by the BASIC display program.

After checking for errors, the received character is stored in the BUFFER and then ECHOed back to the transmitter, which is usual in serial communications. However, if the character was originated by our own keyboard and then received after being echoed back by the other end, we obviously do not want to echo it back again. So the TRANSMIT routine (in BASIC) sets a flag (TFLAG) in memory location $00FC whenever it sends a byte. Our MC receive routine checks to see if TFLAG is set before echoing the character. Then, after restoring the X register and accumulator values, the RTI instruction returns control back to BASIC after pulling the flags back off the stack.

This entire program can be tested using the circuit of figure 9-6.

9-5 RS 232 C INTERFACE

Line drivers and line receivers are used to round off the leading and trailing edges of rectangular pulses and to get a more constant impedance on the line, as was mentioned earlier. We will now look at some commonly used line drivers and receivers.

It was realized years ago that some standardization was necessary if equipment obtained from different manufacturers was to be compatible. So the Electronic Industries Association (EIA) recommended the use of certain features of serial communications, which came to be essentially an industry standard. This standard is called *RS 232 C* and is found on a wide variety of equipment. Generally speaking, if a manufacturer claims that his equipment is RS 232 C compatible, then it should be able to communicate with other RS 232 C equipment.

RS 232 C specifies the electrical, mechanical, and functional characteristics of a serial port, sometimes called a *data interchange*. Some of the electrical specifications are: (1) Transmitters must be able to withstand opens or shorts on the line without damage; (2) receivers must tolerate input signals up to + or − 25 volts without damage; and (3) the input impedance of the receiver must be not less than 3K ohms nor more than 7K ohms. Other specifications are completely described in the EIA standard. We will not discuss these here, especially since RS 232 C circuitry can be purchased in IC form without the user having to worry about the details.

Figure 9-11 shows some RS 232 C line drivers and receivers connected to the ACIA output. The MC1488 and MC1489 chips are commonly available from Motorola. Notice that the MC1488 line driver is connected to +12 and −12 volt power supplies. This increases the amplitude of the signal on the line, thereby increasing the signal-to-noise ratio in a noisy atmosphere. Figure 9-12 shows

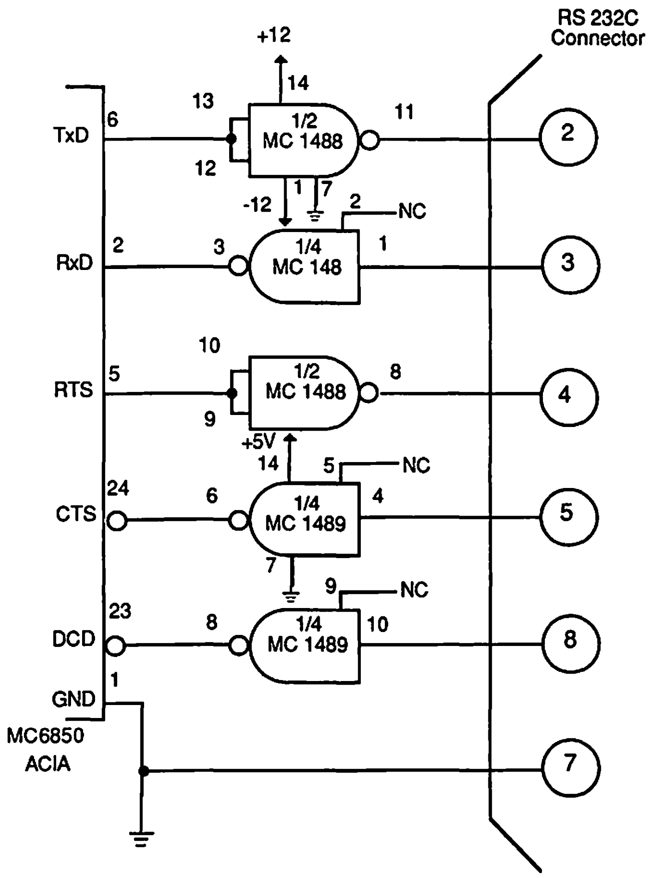


Figure 9-11
Line drivers and line receivers

what the output signal of the line driver looks like when driven from the TxD output of the ACIA. Notice that the signal is 24 volts peak-to-peak, rather than just 5 volts. Also notice that the signal is inverted; that is, a logic HIGH of 5 volts at the TxD output appears as a −12 volt level at the line-driver output. Therefore, the logic one or "mark" level on the line is always −12 volts. (Some circuits use 15-volt supplies, but the idea is the same.) Besides giving a larger amplitude signal, the line drivers round off the leading and trailing edges of the pulses to eliminate the ringing and reflection problems discussed earlier.

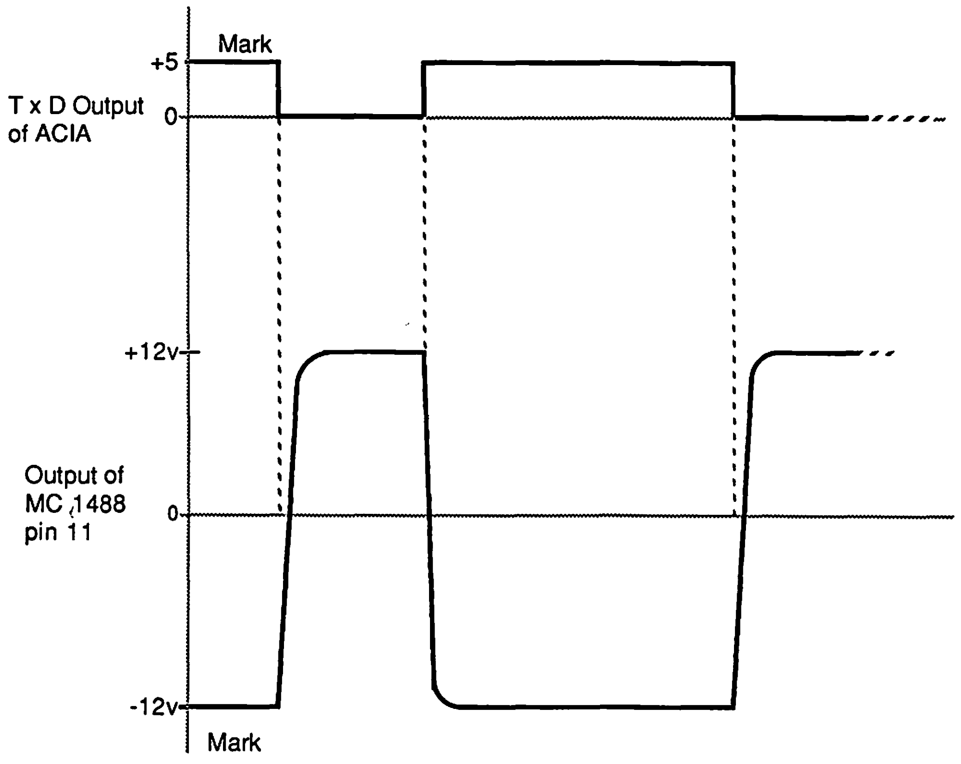


Figure 9-12
MC1488 line driver output

Usually, the RS 232 C connection is made through a 25-pin, D-type connector, as shown in figure 9-13. The pin assignments are also standard. The figure shows the female connector, which would normally be used for the *data terminal*, which is analogous to the old TTY equipment. At the other end (controlling end), the equipment is called the *data set*. The pin connections for the TxD and RxD of the data set are reversed from those of the data terminal.

Although a complete description of RS 232 C is not possible here, let's discuss some of the protocol of communications on the interchange. Whenever the data set is powered up and on line, it outputs a +12 v level on pin 6 of the connector, which is marked DSR (data set ready). Likewise, when the data terminal is on line, it outputs a +12 v on pin 20, which is marked DTR (data terminal ready).

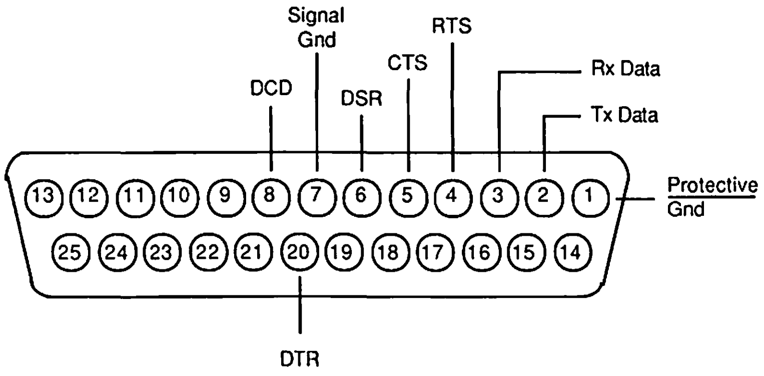


Figure 9-13
25-pin D-type connector commonly used for RS 232 C interface (female end)

Whenever the data terminal wants to transmit a message, it must request permission by making RTS (pin 4) of the connector go active. The ACIA output pin connected to that pin is marked $\overline{\text{RTS}}$ for that reason. So when you want to transmit, you make $\overline{\text{RTS}}$ (pin 5 of the ACIA) go active LOW. If the data set gives permission to transmit, it will make CTS (clear to send) go active. On the ACIA, if $\overline{\text{CTS}}$ is not active, the transmitter will be disabled. Likewise, if $\overline{\text{DCD}}$ is not active, the receiver will be disabled, so pin 8 of the connector must be at +12 v for the receiver to operate.

Since not all RS 232 C links use the RTS, CTS, DTR, and DCD lines, you should read the particular specifications of any system to which you are connecting.

The simplest communications link would consist of using only the TxD, RxD, and ground lines of the connector. If the system to which you are connecting does not use CTS and DCD, be sure to make the $\overline{\text{CTS}}$ and $\overline{\text{DCD}}$ inputs of the ACIA both LOW or the chip will not transmit or receive. One way to do this is to jumper together pins 4, 5, and 8 of the D-type connector. This way, when you make $\overline{\text{RTS}}$ LOW on initialization, you also make $\overline{\text{CTS}}$ and $\overline{\text{DCD}}$ active, so that the transmitter and receiver of your ACIA are both enabled.

CHAPTER 10
IEEE 488 Bus

In chapter 9, we discussed the RS 232 C interface. The RS 232 C is basically the industry standard in serial communications. We will now study the *IEEE 488 bus,* which has become somewhat the industry standard parallel-interface scheme for tying instrumentation to a computer. The IEEE 488 bus was originally developed by Hewlett-Packard, who owns the patent on the system. It is also known as the *HPIB* (Hewlett-Packard Interface Bus) and the *GPIB* (General Purpose Interface Bus). After we examine the bus's structure and the types of signals and commands used on it, we will look at how we can tie our computer to it to control some test equipment.

10-1 BUS OVERVIEW

The IEEE 488 bus allows up to 15 instruments to be tied to a single controller (usually the computer) simultaneously by means of three sets of lines, or buses. These buses are the data bus (8 lines), the interface management bus (5 lines), and the handshaking bus (3 lines). Maximum length of the bus is 20 m if several instruments are used, but the maximum distance between any two devices is 4 m. Maximum data transfer rate is 1 megabyte/sec, but in practice it is usually much slower.

Figure 10-1 shows the interconnection of various instruments to the buses. Notice that there are three types of devices that can be tied to the buses. They are classified as *talkers, listeners,* and *controllers*. Examples of talkers are voltmeters, A/D converters, and frequency counters. Listeners may be instruments such as printers, chart recorders, or signal generators. Some devices can be both talkers and listeners, such as a programmable DMM. The controller is usually a computer that monitors and directs activity on the bus. The controller can talk or listen to any one of the devices by means of standard commands. In addition, the controller can tell one of the devices (a talker) to send information to one or more of the other devices (listeners). That is, one device can send information directly to another device without relaying the information through the computer.

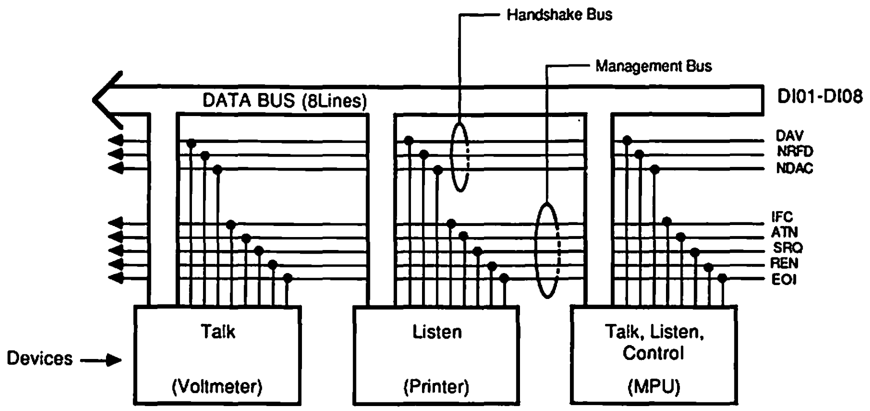


Figure 10-1
IEEE 488 Interface Bus

To ensure reliable communications between the various devices, a well-organized set of control and handshake signals are used. Table 10-1 shows the name and purpose of each line of the bus. The eight data lines DIO1 through DIO8 carry the message bits as well as address information, while the other two buses carry control or status information. The control and status information is transferred by means of an interlocked sequence of signals, which means that one event in the sequence must finish before the next one can begin. For example, let's assume that the voltmeter wants to send some information to the printer. Figure 10-2 shows the timing diagrams for the handshake and data buses:

1. The voltmeter holds DAV HIGH and samples the NRFD line to make sure that the printer is ready for data. If the printer is not ready for data, it holds NRFD active LOW
2. When the printer is ready, the printer pulls NRFD HIGH.
3. Sensing that NRFD is HIGH, the voltmeter then places its data on the data bus and makes DAV go LOW to indicate to the printer that data is available. The printer, in the meantime, was holding NDAC LOW.
4. When the printer senses DAV going LOW, it makes NRFD go LOW.

TABLE 10-1
Functions of the IEEE 488 Bus Lines

Signal Name	*Description*
DIO1-DIO8	Data bus lines.
DAV	DATA VALID. Pulled LOW by talker to inform listeners that data has been placed on DIO lines.
NRFD	NOT READY FOR DATA. Pulled LOW by all listeners. Released by each listener when it becomes ready to receive data.
NDAC	NOT DATA ACCEPTED. Pulled LOW by all listeners. Released by each listener when it has accepted data.
IFC	INTERFACE CLEAR. Driven LOW by controller to bring all interface lines to known state.
ATN	ATTENTION. Driven LOW by controller to gain the attention of devices on bus and to signify that address/control information is on the bus.
SRQ	SERVICE REQUEST. Pulled LOW by any device needing service. Similar to interrupt request.
REN	REMOTE ENABLE. Pulled LOW by controller to ensure that remote control is in effect. For example, front panel controls can be disabled by REN.
EOI	END OR IDENTIFY. Pulled LOW by talker to inform listeners that current byte on data bus is the last byte to be transferred. Pulled LOW by controller together with ATN to initiate a parallel poll sequence.

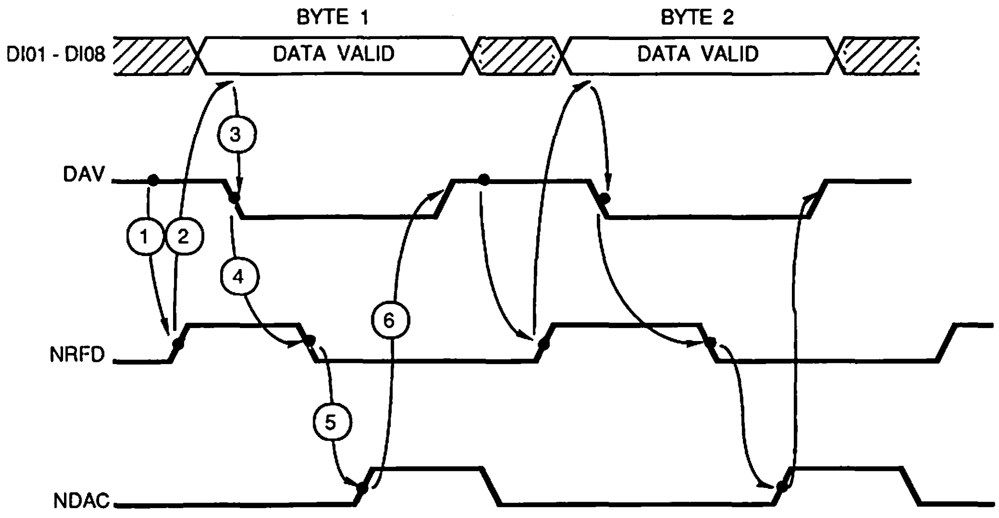


Figure 10-2
Timing diagrams for data and handshake lines

5. Then, when the printer inputs the data, the printer pulls NDAC HIGH, thus informing the voltmeter that data has been accepted.
6. Sensing that data has been accepted, the voltmeter then pulls DAV HIGH again. The process is repeated each time a new data byte is to be transferred.

The signal lines DAV, NRFD, and NDAC are all open collector lines used for handshaking between talkers and listeners. Since they are open collector lines, several instruments may be simultaneously tied to each line, and the data transfer rate will be controlled by the slowest device. Figure 10-3 shows the timing diagrams when several listeners are activated at the same time.

The problem now arises as to how does the talker know that it has permission to talk and how does any individual device know whether or not it should listen? The control signals, issued by the controller, are what direct talkers to talk and listeners to listen. The following is a typical sequence showing how this is accomplished: On power up, the controller takes control of the bus and sends out an IFC signal on the management bus. This places all devices in a

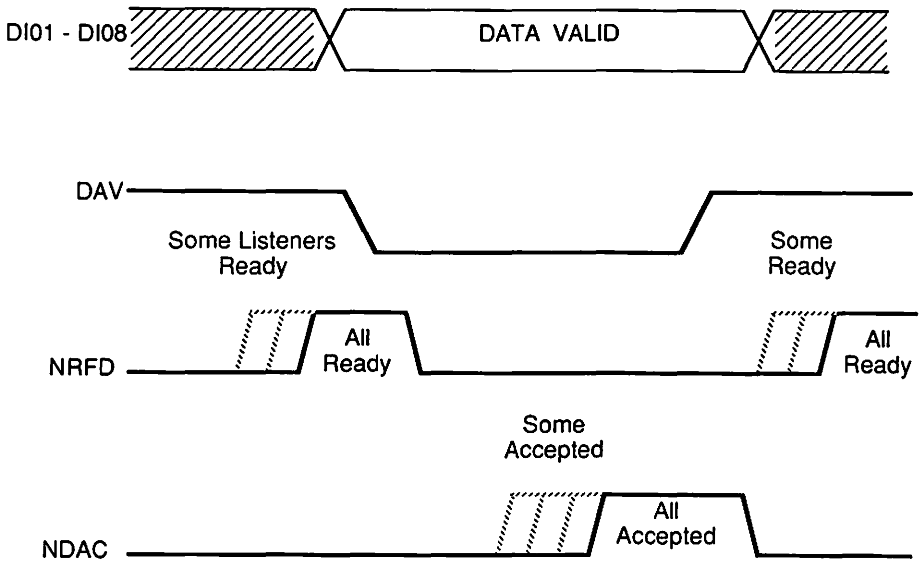


Figure 10-3
Timing diagrams when several listeners are on the bus

known state, with no one talking and no one expecting data. Then, if there are several instruments on the bus, the controller polls the bus to watch for an active SRQ signal. The SRQ signal is equivalent to an interrupt request. In fact, an instrument can be programmed to generate an interrupt request on receipt of an SRQ, if desired.

Let's suppose that our DVM has completed conversion and has data for the printer. The DVM pulls SRQ LOW. The controller (after determining that the DVM is the source of the SRQ) makes ATN go LOW to gain the attention of all devices on the bus. It then outputs a LISTEN command, specifying the address of the listener. The command is issued on the DIO lines and is of the form X01AAAAA, where the AAAAA specifies a 5-bit address of the listener. This address is effectively hardwired into the listener, and when the listener detects its own address in the command, it enters the listen mode. Next, the controller issues a TALK command on the DIO lines, which is of the form X10AAAAA. The talker (our DVM) then begins the sequence of NRFD-DAV-NDAC handshaking and data transfer to the printer, as discussed previously.

Meanwhile, the controller was in the standby state, monitoring the bus but not taking part in the transfer of data. When the transfer of data is complete, the talker sends an EOI signal on the management bus. The controller, on detecting the EOI, takes control again and sends an UNTALK command (X1011111) followed by an UNLISTEN command (X0111111) on the DIO lines. The talker stops talking and the listener stops listening so the bus is now available for other transfers.

Table 10-2 shows the format for the various interface message. While we will not discuss all of the possible messages here, we will study enough of them to make a small system work effectively. For a thorough discussion of all the commands, refer to the IEEE 488 specifications. Not all of the commands are used in every application, particularly in smaller systems. In fact, there may be some differences in the way some systems handle certain events. For example, the response to the SRQ is not standardized. In some systems it may initiate a parallel polling sequence to establish the source of the request; while in other systems, it may cause an immediate jump to a particular service routine.

One important point is that the IEEE 488 bus uses *negative logic*. This means that a TRUE (logic 1) signal appears on the bus as a

TABLE 10-2
IEEE 488 Interface Messages

Command	*Symbol*	*DIO 1-8**
ADDRESSED COMMAND GROUP	ACG	000XXXXX
DEVICE CLEAR	DCL	X0010100
GROUP EXECUTE TRIGGER	GET	X0001000
GO TO LOCAL	GTL	X0000001
LISTEN ADDRESS GROUP	LAG	X01XXXXX
LOCAL LOCKOUT	LLO	X0010001
MY LISTEN ADDRESS	MLA	X01AAAAA
MY TALK ADDRESS	MTA	X10AAAAA
MY SECONDARY ADDRESS	MSA	X11SSSSS
OTHER SECONDARY ADDRESS	OSA	SCG.MSA
OTHER TALK ADDRESS	OTA	TAG.MTA
PRIMARY COMMAND GROUP	PCG	ACG+UCG+LAG+TAG
PARALLEL POLL CONFIGURE	PPC	X0000101
PARALLEL POLL ENABLE	PPE	X110SPPP
PARALLEL POLL DISABLE	PPD	X111DDDD
PARALLEL POLL UNCONFIGURE	PPU	X0010101
SECONDARY COMMAND GROUP	SCG	X11XXXXX
SELECTED DEVICE CLEAR	SDC	X0000100
SERIAL POLL DISABLE	SPD	X0011001
SERIAL POLL ENABLE	SPE	X0011000
TAKE CONTROL	TCT	X0001001
TALK ADDRESS GROUP	TAG	X10XXXXX
UNLISTEN	UNL	X0111111
UNTALK	UNT	X1011111
UNIVERSAL COMMAND GROUP	UCG	X001XXXX

*0 Logical zero (HIGH level on GPIB), 1 Logical one (LOW level on GPIB), X Don't care (received message).

LOW voltage (0 volts), while a FALSE (logic 0) signal appears on the bus as a HIGH (> 2v) level. This does not mean that you must invert all of your thinking about what is going on at the computer end. But on the bus lines themselves, the voltage levels are the opposite of what you would measure on your computer bus. For example, if your computer outputs the byte 00110100, it will appear on the IEEE bus as HHLLHLHH, where H represents >2v and L represents 0v as measured with a voltmeter. IC manufacturers such as Motorola and Texas Instruments make IEEE 488 compatible chips that take care of the inversions for you. The interface circuits are TTL-compatible and use a single +5v power supply.

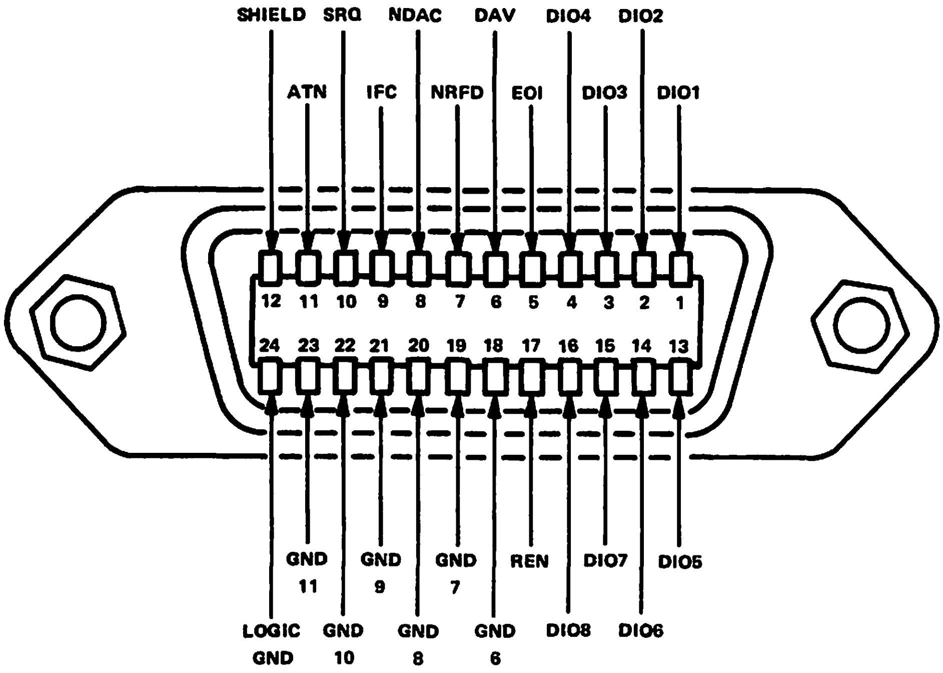


Figure 10-4
Standard IEEE connector

Many instrument manufacturers build instruments capable of tying into the IEEE 488 bus. These instruments use standard 24-pin connectors, such as the one shown in figure 10-4.

10-2 THE TMS9914A GPIB CONTROLLER

It is possible to interface the computer to the IEEE 488 bus using simple buffers and latches, just as it would be possible to control serial communications with buffers and latches. But doing so would require a great deal of computer time and software to monitor and generate control and handshake signals, not to mention a handful of hang-on chips. We saw that by using a programmable interface chip, such as Motorola's ACIA, we can free the computer from all of the formatting, control signal generation, and

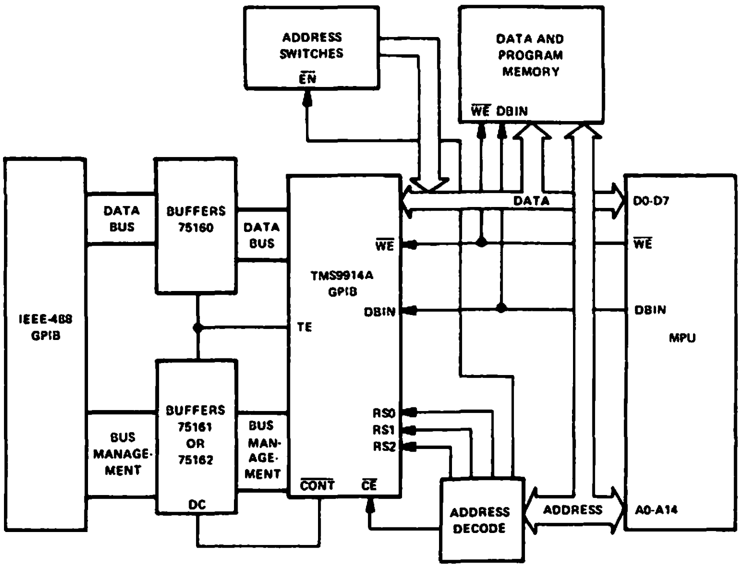


Figure 10-5
Typical TMS9914A application
(Courtesy of Texas Instruments, Inc.)

timing problems. Similarly, by using a chip specifically designed to interface to the IEEE bus, we make the computer's job much simpler. One very popular chip, made by Texas Instruments, is the TMS9914A General Purpose Interface Bus Controller. The chip is normally memory mapped into the system, for example, in one of the computer's peripheral slots. Using this chip, the computer simply talks to or reads from a few memory locations to send or receive data. Handshaking is done automatically without tying up computer time. More importantly, as mentioned before, the computer can direct one device to talk to one or more other devices without having to relay the information.

Figure 10-5 shows the TMS9914A in a typical application. An IEEE 488 compatible instrument, such as a DVM, would have an interface board with all of the circuitry shown in the figure. The MPU, of course, also controls some of the hardware that makes the DVM do its job. The onboard data and program memory contains firmware that makes the MPU control the DVM functions, as well as the TMS9914A functions and initialization. Each instrument must initialize its own TMS9914A on power up. The 75160 and 75161

TABLE 10-3
TMS9914A Read registers
(Courtesy of Texas Instruments, Inc.)

ADDRESS			REGISTER NAME	BIT ASSIGNMENT							
RS2	RS1	RS0		D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	Int Status 0	INT0	INT1	BI	BO	END	SPAS	RLC	MAC
0	0	1	Int Status 1	GET	ERR	UNC	APT	DCAS	MA	SRQ	IFC
0	1	0	Address Status	REM	LLO	ATN	LPAS	TPAS	LADS	TADS	ulpa
0	1	1	Bus Status	ATN	DAV	NDAC	NRFD	EOI	SRQ	IFC	REN
1	0	0	*								
1	0	1	*								
1	1	0	Cmd Pass Thru	DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1
1	1	1	Data In	DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1

*The TMS9914A host interface data lines will remain in the high impedance state when these register locations are addressed. An Address Switch Register may therefore be included in the address space of the device at these locations (see Section 1.5).

TABLE 10-4
TMS9914A Write registers
(Courtesy of Texas Instruments, Inc.)

ADDRESS			REGISTER NAME	BIT ASSIGNMENT							
RS2	RS1	RS0		D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	Int Mask 0			BI	BO	END	SPAS	RLC	MAC
0	0	1	Int Mask 1	GET	ERR	UNC	APT	DCAS	MA	SRQ	IFC
0	1	0	*	xx	xx	xx	xx	xx	xx	xx	xx
0	1	1	Auxiliary Cmd	cs	xx	xx	f4	f3	f2	f1	f0
1	0	0	Address	edpa	dal	dat	A5	A4	A3	A2	A1
1	0	1	Serial Poll	S8	rsvl	S6	S5	S4	S3	S2	S1
1	1	0	Parallel Poll	PP8	PP7	PP6	PP5	PP4	PP3	PP2	PP1
1	1	1	Data Out	DIO8	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1

*This address is not decoded by the TMS 9914A. A write to this location will have no effect on the device, as if a write had not occurred.

chips are specifically designed to interface to the IEEE bus. And the direction of data flow through these buffers is controlled by the TE and $\overline{\text{CONT}}$ outputs of the TMS9914A.

Communications between the MPU and the TMS9914A are accomplished via 13 memory-mapped registers in the TMS9914A. Once the chip is enabled by an address decoder, one of the specific registers is selected by means of address bits applied to 3 register-select pins, RS2, RS1, and RS0, in the same manner that various registers in the PIA and ACIA were selected. However, 6 of the registers are read only, and 7 registers are write only. Table 10-3 shows the function of the read registers, and table 10-4 shows the function of the write registers. We will discuss the operation and purpose of a few of these registers.

Whenever the MPU wants to read in a byte from the GPIB, the MPU makes RS2, RS1, and RS0 all HIGH thereby selecting the data in register and does a read operation (LDA). But when the MPU wants to send a byte out to the GPIB, it makes the register-select pins all HIGH and does a write operation (STA), thereby selecting the data out register.

It was mentioned in the previous section that each talker and listener on the IEEE bus is given a 5-bit address to identify it and that this address is somehow hardwired into the device. You can appreciate the need for these device addresses to be user changeable so that you can buy instruments from different manufacturers and not have to worry about address conflicts. Here is how it is usually done. Each instrument, our DVM for example, has a 5-position dip switch located on the rear panel of the instrument. The user sets the switch to any address (except 11111) that he or she wants the device to respond to. Then on power up, an initialization routine in the instrument's ROM causes the MPU to read from the address switches. The MPU then writes this 5-bit address into the address register (100) of the TMS9914A. Thereafter, whenever that 5-bit address appears on the DIO lines, the instrument knows that it should respond. For example, suppose that the DVM address switches are set to 01000. When the controller issues the TALK command X1001000, the DVM recognizes its address and begins talking. The general form of the TALK command, as shown in table 10-2, is X10AAAAA, where the AAAAA specify the particular device that is to respond.

Registers 000 and 001 are used for interrupt status and control. If you want an interrupt request to be generated when an input byte is available, you set the BI bit in the int mask 0 register. This corresponds to the receive data register full interrupt on the ACIA. Similarly, if you want an interrupt request to be generated when the output register is ready to accept the next output byte, you set the BO bit of register 0. This corresponds to the transmit data register empty interrupt in the ACIA. If you do not want interrupts to be generated, the corresponding BI and BO bits in the int status 0 register can be polled to see whether an input byte is present or the output register is ready. The END bit in the int status register is used to detect the end of a message. It gets set when the TMS9914A detects an EOI signal on the management bus. The int mask 0 register is not cleared by either a hardware or software reset. It will come up in a random state on power up. Therefore, part of the initialization routine must write the desired interrupt mask into it. The same thing must be done for int mask 1 register.

TABLE 10-5
TMS9914A Auxiliary Commands
(Courtesy of Texas Instruments, Inc.)

c/s	f4	f3	f2	f1	f0	MNEMONIC	FEATURES
0/1	0	0	0	0	0	swrst	Software reset
0/1	0	0	0	0	1	dacr	Release DAC holdoff
na	0	0	0	1	0	rhdf	Release RFD holdoff
0/1	0	0	0	1	1	hdfa	Holdoff on all data
0/1	0	0	1	0	0	hdfe	Holdoff on EOI only
na	0	0	1	0	1	nbaf	New byte available false
0/1	0	0	1	1	0	fget	Force group execute trigger
0/1	0	0	1	1	1	rtl	Return to local
na	0	1	0	0	0	feoi	Send EOI with next byte
0/1	0	1	0	0	1	lon	Listen only
0/1	0	1	0	1	0	ton	Talk only
na	0	1	0	1	1	gts	Go to standby
na	0	1	1	0	0	tca	Take control asynchronously
na	0	1	1	0	1	tcs	Take control synchronously
0/1	0	1	1	1	0	rpp	Request parallel poll
0/1	0	1	1	1	1	sic	Send interface clear
0/1	1	0	0	0	0	sre	Send remote enable
na	1	0	0	0	1	rqc	Request control
na	1	0	0	1	0	rlc	Release control
0/1	1	0	0	1	1	dai	Disable all interrupts
na	1	0	1	0	0	pts	Pass through next secondary
0/1	1	0	1	0	1	stdl	Short TI settling time
0/1	1	0	1	1	0	shdw	Shadow handshake
0/1	1	0	1	1	1	vstdl	Very short T1 delay
0/1	1	1	0	0	0	rsv2	Request Service Bit 2

The auxiliary command register (011) is used to enable and disable most of the selectable features of the TMS9914A and to initiate many of its actions. Table 10-5 shows how the desired features are selected by writing various bit patterns into this register. Bits f4-f0 are the 5 least significant bits of this register. The column labeled c/s indicates that the function will be set (enabled) when a 1 is written into the most significant position of the register, and writing a 0 into the MSB will clear (disable) the feature. For example, let's assume that we are using the TMS9914A as a system controller. Suppose we want to send an INTERFACE CLEAR pulse out on the management bus. We do this by first writing the bit pattern 1XX01111 to the auxiliary command register. This sets IFC active. Then, after a short time delay of perhaps 1 ms, we send the bit pattern 0XX01111 to the same register. This second byte causes the TMS9914A to make the IFC command line inactive. Of course, not all of the possible commands are used in every application. But we will see how several other commands are used in a practical application in the next section.

Although, as mentioned previously, negative logic is used on the GPIB lines themselves, the connections between the MPU and the TMS9914A use conventional positive logic. That is, the bit patterns shown in the tables are the same bit patterns that the MPU must output to the TMS 9914A. For example, to set the IFC active, the MPU would execute an instruction like LDA #$8F, then store the accumulator (STA) to the auxiliary command register address of the TMS9914A. To make IFC inactive, the MPU does a LDA #$OF then stores it to the same address.

10-3 CONNECTING THE APPLE TO THE IEEE 488 BUS

Interfacing the Apple computer to the IEEE 488 bus can be accomplished using the TMS9914A along with its buffers, the 75160 and 75161, as shown in figure 10-6. These three chips can be mounted on a simple prototype card and the card can be plugged into any suitable slot. Note that Apple data bus line D7 connects to pin 17 of the TMS9914A, marked D0. D0 of the TMS9914A is the most significant bit of the device. This is in agreement with the designations in tables 10-3 and 10-4. By using $\overline{\text{DEVICE SELECT}}$ as the chip enable and feeding RS2, RS1, and RS0 from A2, A1, and A0 as shown, the TMS9914A will respond to 8 consecutive addresses issued by the Apple. For example, if we placed the card in slot 4, the address range of the interface will extend from 49344 ($C0C0) to 49351 ($C0C7). The chip also needs an external clock input, so Φ1 of the Apple clock (pin 38) is used.

Since the best way to understand the Apple-to-GPIB interface is through an example, we will discuss connecting the Apple to a Keithley Model 192 Programmable DMM. The Model 192 is externally programmable via the GPIB for selection of function (DCV, ACV, or K ohms), range, rate of data capture, and several other features. It has the standard IEEE 488 connector on the rear panel, as well as the 5-bit dip switch for address selection. To select the desired features of the DMM, the controller issues a LISTEN command using the address bits of the DMM. Then the DMM is sent several bytes of data telling it what to do. After sending all of the required bytes, the controller sends an UNLISTEN command,

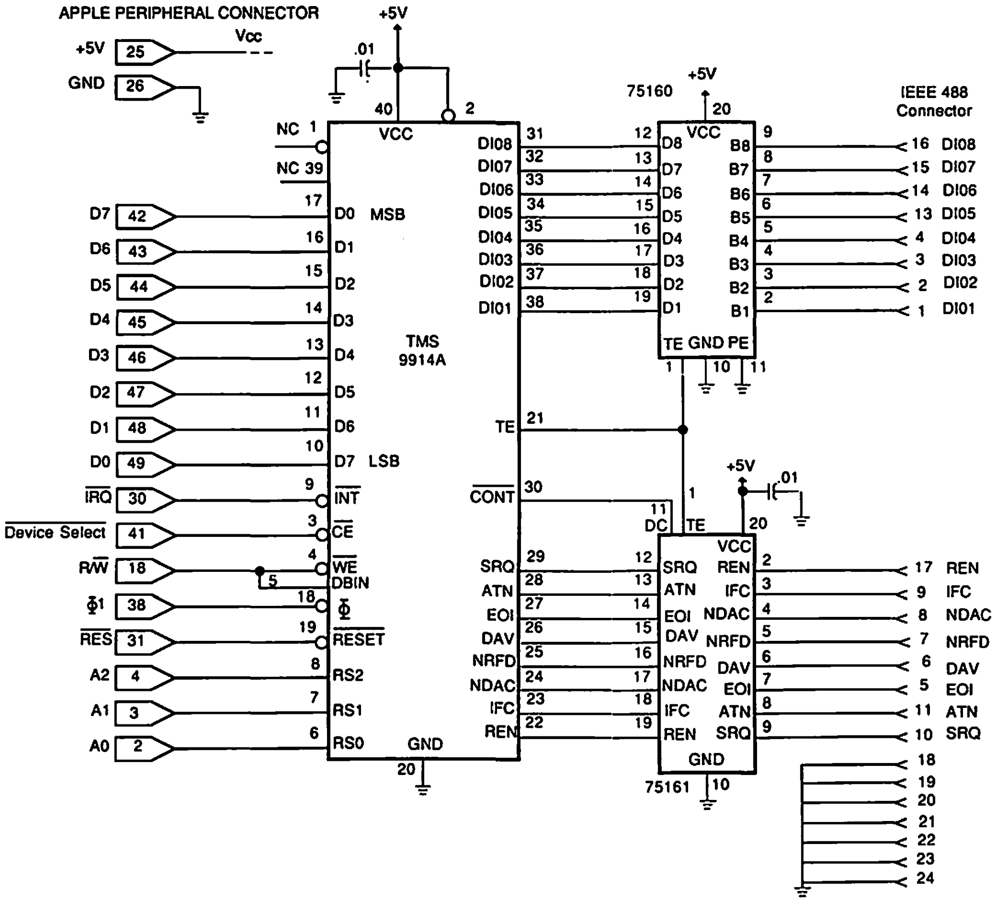


Figure 10-6
IEEE 488 Bus/Apple interface

followed by a TALK command, again using the address bits of the DMM. The DMM then responds by outputting data on the DIO lines according to the selected format. If no format is selected by the user, the DMM defaults to the predetermined format, which is dc volts on the 2000 volt range. We will use the default mode for our first example.

Figure 10-7 shows the listing for a BASIC program to communicate with the DMM via the IEEE 488 bus. While the program can and often is written in assembly language, we will use BASIC for simplicity.

```
]LIST

10  REM  *** SIMPLE IEEE-488 DEMO ***
12  REM  * INTERFACE TO KEITHLEY MOD 192 DMM *
15  REM  * MAP REGISTERS OF 9914 *
20 STTUS = 49344
25 CMD = 49347
30 DIO = 49351
35 DMMT = 72: REM  DMM TALK ADDRESS
100  REM  * INIT CONTROLLER *
110  POKE CMD,128: REM  SET SWRST
120  POKE CMD,147: REM  DISABLE INTERRUPTS
130  POKE CMD,0: REM  CLEAR SWRST
140  POKE CMD,12: REM  TAKE CONTROL
150  POKE CMD,143: REM  SEND IFC
160  POKE CMD,15: REM  RESET IFC
170  POKE DIO,DMMT: REM   TELL DMM TO TALK
180  POKE CMD,11: REM  GO TO STANDBY
190  POKE CMD,137: REM  SET UP 9914 TO LISTEN
200  REM  * INPUT ROUTINE *
210 S =  PEEK (STTUS)
220  IF S < 32 THEN  GOTO 210: REM  POLL BI
230 BYTE =  PEEK (DIO): REM  GET CHARACTER
240 C$ =  CHR$ (BYTE)
245  PRINT C$;
250  IF S < 40 GOTO 210: REM  IF NOT END, GET NEXT BYTE
255  END


]RUN
NDCV+0005.753E+0
```

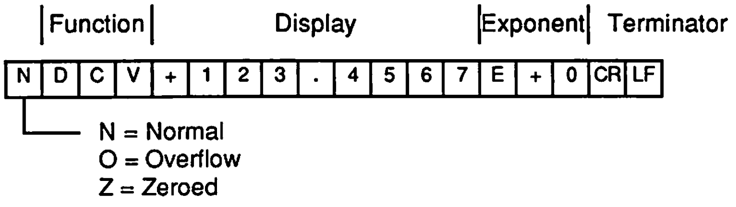
Figure 10-7
Listing of simple IEEE 488 demo

The program begins by assigning labels to the various addresses of the TMS9914A chip of figure 10-6. We will place the interface card in slot 4. If you use any other slot, be sure to change lines 20, 25, and 30 accordingly. Line 35 equates the label DMMT as the DMM talk address. For this example, the DIP switch on the rear of the DMM is set at 01000. So, according to Table 10-2, when the controller wants to tell the DMM to talk, the controller outputs a byte on the DI0 lines corresponding to the MTA message X10AAAAA. Since the DIP switch is set at 01000, the MTA message is X1001000. The X in the address is a don't care bit so we set it LOW, forming the byte 01001000 ($48), which is equivalent to 72 in decimal.

Next we begin the initialization of the TMS9914A. Line 110 POKEs the auxiliary command register with the value 128 (10000000 in binary), causing a software reset (see table 10-5). This is the usual first command on power up. While in the software reset state, the TMS9914A is usually sent the desired interrupt masks. We will not use interrupts in this example, so we POKE the auxiliary command register with 147 (10010011) to disable all interrupts. Line 130 clears the software reset. Line 140 tells our TMS9914A to act as the controller for the bus. Lines 150 and 160 cause the IFC line to be pulsed active LOW for a short time. If you do this in a machine-language routine, be sure to include a time delay (perhaps 1 ms) between the time that IFC goes LOW until it goes HIGH again. BASIC is slow enough that we do not have to include the delay. Line 170 causes the controller to output the byte $48 (01001000), thus establishing the DMM as a talker. Line 190 tells the TMS9914A to listen, while waiting for the data from the DMM.

The DATA INPUT routine, starting at line 200, polls the status register to see if BI (byte in) is HIGH. When BI goes HIGH, the status byte will be 00100000, as can be seen in table 10-3. The decimal equivalent of the status byte is 32. So when BI goes active, the data is read in from the DIO lines, changed to a printable character, and printed on the Apple's CRT. Then the status byte is checked to see if END is active, which indicates the end of the message. The END bit of the status byte is set when the talker makes its EOI handshake line active while sending its last byte. If END is not active, the program loops back to input the next byte. When END is detected (status byte 00101000 or 40 in decimal), the program falls through.

The RUN of the program, shown at the bottom of figure 10-7, shows that the Model 192 DMM sends 16 bytes of data (followed by a carriage return-line feed). The format of the data string is shown



Data Format: 16 Bytes + Terminator

Figure 10-8
Data format for Keithley Model 192 DMM

```
]LIST

10  REM  *** SIMPLE IEEE-488 DEMO ***
12  REM  * INTERFACE TO KEITHLEY MOD 192 DMM *
15  REM  * MAP REGISTERS OF 9914 *
20 STTUS = 49344
25 CMD = 49347
30 DIO = 49351
35 DMMT = 72: REM  DMM TALK ADDRESS
100  REM  * INIT CONTROLLER *
110  POKE CMD,128: REM  SET SWRST
120  POKE CMD,147: REM  DISABLE INTERRUPTS
130  POKE CMD,0: REM  CLEAR SWRST
140  POKE CMD,12: REM  TAKE CONTROL
150  POKE CMD,143: REM  SEND IFC
160  POKE CMD,15: REM  RESET IFC
170  POKE DIO,DMMT: REM   TELL DMM TO TALK
180  POKE CMD,11: REM  GO TO STANDBY
190  POKE CMD,137: REM  SET UP 9914 TO LISTEN
200  REM  * INPUT ROUTINE *
210 S =  PEEK (STTUS)
220  IF S < 32 THEN  GOTO 210: REM  POLL BI
230 BYTE =  PEEK (DIO): REM  GET CHARACTER
240 C$ =  CHR$ (BYTE)
245  PRINT C$;
250  IF S < 40 GOTO 210: REM  IF NOT END, GET NEXT BYTE
260  FOR D = 1 TO 7000: NEXT D
270  GOTO 140


]RUN
NDCV+0008.819E+0

NDCV+0009.414E+0

NDCV+0010.585E+0

NDCV+0011.551E+0


BREAK IN 260
```

Figure 10-9
Listing of simple IEEE 488 demo for periodic sampling

in figure 10-8. In some installations, we might want periodic readings from the DMM, for instance, once every 10 seconds or once every half hour. Figure 10-9 shows a simple modification of the previous program in which the computer inputs the data string from the DMM then after some time delay (line 260) asks for another reading.

In the programs of figures 10-7 and 10-9, the front panel controls of the DMM selected the function, range, and so on. But

TABLE 10-6
IEEE 488 Programming for the Keithley Model 192 DDM

Feature	*ASCII*
FUNCTION	F0 = DCV
	F1 = ACV
	F2 = K OHMS
RANGE	R0 = AUTO
	R1 = 0.2
	R2 = 2
	R3 = 20
	R4 = 200
	R5 = 2000
	R6 = 20 M OHMS
ZERO	Z0 = OFF
	Z1 = ON
TRIGGER	T0 = Cont. on TLK
	T1 = One shot on TLK
	T2 = Cont. on GET
	T3 = One shot on GET
	T4 = Cont. on X
	T5 = One shot on X
RATE	S0 = 4 ms integration (4 ½ d)
	S1 − S8 various rates
DELAY	W0 = 0
	W1 = 10 ms
BUFFER	Q0 = Clear
	Q1 = Store 100 readings
MODE	M0 = SRQ OFF
	M1 = SRQ ON
EOI	K0 = SEND
	K1 = DO NOT SEND
TERMINATOR	Y(LF) = CF LF
	Y(CR) = LF CR
	Y( ) = Any ASCII
	Y(DEL) = None
	X = EXECUTE
	U = Send status bytes

Note: Default = F0R5Z0T0S2W1Q0X0M0Y(LF)

one of the powerful features of a programmable DMM, like the Keithley Model 192, is that the function, range, rate of capture, and so on, can be programmed from a remote computer. Here is how the remote programming works. On power up, the DMM is put into the LISTEN mode, rather than the TALK mode. The computer then sends control bytes (a string of ASCII characters) to the DMM to tell it how it is to operate. After initialization, the DMM is placed in the TALK mode, as before, and sends data to the listener(s).

Table 10-6 shows the various features that can be remotely controlled in the Keithley Model 192. By sending the ASCII bytes F1R4, for example, the DMM is told to place its internal function selector in the ac volts position and its range switch on the 200-volt scale. Only those features that you wish to change must be sent to the DMM. The default values are shown at the bottom of the table.

Figure 10-10 shows the program listing that allows you to remotely program the DMM. Notice that the major portion of the program is the same as that of figure 10-7. The title line (line 10) is changed, a new line (line 40) equates the DMM listen address to 40 decimal (00101000), and line 165 has been added to call a subroutine to initialize the DMM. Otherwise, the program is identical to that of figure 10-7 up to line 255.

The subroutine at line 300 first sets the remote enable line active, then tells the DMM to listen. Next it puts the controller in standby and makes the TMS9914A a talker. The code from lines 350 through 400 allow the user to enter the bytes to the be sent to the DMM, as described in table 10-6. The character X is used to terminate the message and to trigger the DMM to activate the features sent to it. Although the Keithley Model 192 recognizes the character X as an execute command, other instruments might not. A typical remote programming sequence might include having the controller send the remote device a string of ASCII characters, and along with the last character sending an EOI signal, indicating the end of message. The EOI can be sent along with the last character by writing the command 08 to the auxiliary command register, as shown in table 10-5.

Getting back to the program of figure 10-10, once the message terminator is sent, the program falls through to line 410, where the TMS9914A is told to stop talking and take control again. The DMM is taken out of the listen mode, and the program returns to line 170 in the main driver. From there on, execution is the same as that of figure 10-7.

```
]LIST

10  REM  *** IEEE 488 DEMO USING REMOTE PROGRAMMING ***
12  REM  * INTERFACE TO KEITHLEY MOD 192 DMM *
15  REM  * MAP REGISTERS OF 9914 *
20 STTUS = 49344
25 CMD = 49347
30 DIO = 49351
35 DMMT = 72: REM  DMM TALK ADDRESS
40 DL = 40: REM  DMM LISTEN ADDRESS
100  REM  * INIT CONTROLLER *
110  POKE CMD,128: REM  SET SWRST
120  POKE CMD,147: REM  DISABLE INTERRUPTS
130  POKE CMD,0: REM  CLEAR SWRST
140  POKE CMD,12: REM  TAKE CONTROL
150  POKE CMD,143: REM  SEND IFC
160  POKE CMD,15: REM  RESET IFC
165  GOSUB 300: REM  INIT DMM
170  POKE DIO,DMMT: REM  TELL DMM TO TALK
180  POKE CMD,11: REM  GO TO STANDBY
190  POKE CMD,137: REM  SET UP 9914 TO LISTEN
200  REM  * INPUT ROUTINE *
210 S =  PEEK (STTUS)
220  IF S < 32 THEN  GOTO 210: REM  POLL BI
230 BYTE =  PEEK (DIO): REM  GET CHARACTER
240 C$ =  CHR$ (BYTE)
245  PRINT C$;
250  IF S < 40 GOTO 210: REM  IF NOT END, GET NEXT BYTE
255  END
300  REM  * INIT DMM FOR REMOTE OPERATION *
310  POKE CMD,144: REM  SEND REMOTE ENABLE
320  POKE DIO,DL: REM  TELL DMM TO LISTEN
330  POKE CMD,11: REM  GO TO STANDBY
340  POKE CMD,138: REM  SET UP 9914 AS TALKER
350  REM  * SEND MESSAGE *
360  PRINT "ENTER MESSAGE CHARACTERS, TERMINATE WITH X"
365  PRINT
370  GET M$
380  POKE DIO, ASC (M$)
390  PRINT M$;
400  IF M$ <  > "X" GOTO 370: REM  CHECK IF LAST CHARACTER
405  PRINT
410  POKE CMD,10: REM  TELL 9914 TO STOP TALKING
420  POKE CMD,12: REM  TAKE CONTROL AGAIN
430  POKE DIO,63: REM  SEND UNLISTEN COMMAND
440  RETURN
```

Figure 10-10
Listing of IEEE 488 demo using remote programming

```
]RUN
ENTER MESSAGE CHARACTERS, TERMINATE WITH X

FOR3X
NDCV+11.56023E+0


]RUN
ENTER MESSAGE CHARACTERS, TERMINATE WITH X

F1X
NACV+00.00992E+0


]RUN
ENTER MESSAGE CHARACTERS, TERMINATE WITH X

F2R4X
OOHM+400.0000E+3
```

Listing 10-10(b)

The sample RUNs at the end of the program listing were obtained using a dc input to the Model 192. Various features were selected in the different RUNs to show how the DMM is affected.

There are a wide variety of other instruments, made by several different manufacturers, that are IEEE compatible. Instruments such as frequency counters and signal generators can be remotely programmed and told to talk to other instruments. As you can see, the IEEE 488 bus is a very powerful and flexible means of tying together a variety of test equipment for automatic testing or data acquisition.

INDEX

A Reston Book
Prentice-Hall, Inc.
Englewood Cliffs, N.J. 07632

0-8359-9222-5