

17

PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

If you were to look around your home, you would see a fairly limited variety of output devices that are used to help you to interface with the electronics in your home. Regardless of how technical they are, just about everyone has seen a light-emitting diode (LED) or pressed a button that produces a simple sound. These basic interfaces can be interfaced easily with PIC[®] microcontrollers and allow you to create your own applications. Many of these interfaces can be added to an application with very little circuitry or software effort, and they do a lot to enhance an application and make it easier to work with. Under the covers of the various products around your home are another set of very common devices with standard interfaces. These devices and their interfaces provide additional functionality to the microcontrollers built into the products, and being standard devices with standard interfaces, the amount of effort to add them to applications is actually quite minimal.

In this chapter I will be introducing you to a number of user interface and hardware function expansion devices that you can use in your own applications. Later in this book when I start introducing you to different applications, you will see these devices in action. The purpose of this chapter is to give you a more generic perspective on this aspect of PIC microcontroller application development and help you to understand the requirements of these devices and what impact they will have on the application development process.

LEDs

The most common form of output from a microcontroller is the LED. As an output device, it is cheap and easy to wire to a microcontroller. Generally, LEDs require anywhere from 5 to 25 mA of current to light (which is often within the output sink/source specification for most microcontrollers). What you have to remember, though, is that LEDs are diodes, which means current flows in one direction only. The typical circuit that I use to control an LED from a PIC microcontroller input-output (I/O) pin is shown in Fig. 17.1.

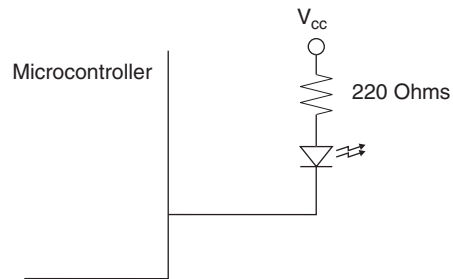


Figure 17.1 The basic circuit for driving an LED consists of a current-limiting resistor and a PIC microcontroller I/O pin capable of sinking enough current to turn on the LED.

With this circuit, the LED will light when the microcontroller's output pin is set to 0 (ground potential). When the pin is set to input or outputs, a 1, the LED will be turned off. This is a general convention for operation that came about owing to simple transistor logic and the Intel 8051, which could not source a significant amount of current; it could only sink enough current to turn on an LED. The popularity of this approach has led to a generation of engineers who design circuitry that turns on LEDs when the logic output is low. If you are new to electronics, this will definitely seem counterintuitive, but it is something that you are going to have to accept (like the fact that current is measured in the direction opposite to electron flow).

The 220- Ω resistor is used for current limiting and will prevent excessive current that can damage the microcontroller, LED, and power supply. The reason why I use a 220- Ω resistor is because when I was a student, I was told that I could never go wrong with it—this is true, but it is also equally true for 330 and 470 Ω in most applications. When you are designing your own applications, you should look at the LED's datasheet to understand its forward voltage as well as "on current" to properly calculate the best current-limiting resistor value.

To calculate the correct voltage, start with the formula:

$$V_{\text{applied}} = V_{\text{LED}} + I_{\text{LED}} \times R_{\text{current limiting}}$$

or rearranged to find $R_{\text{current limiting}}$, the formula becomes

$$R_{\text{current limiting}} = (V_{\text{applied}} - V_{\text{LED}}) / I_{\text{LED}}$$

Using this formula for an LED that has a 1.5-V forward voltage and lights at 5 mA in a system that provides 3.3 V of power, the current-limiting resistor can be calculated:

$$R_{\text{current limiting}} = (3.3 \text{ V} - 1.5 \text{ V}) / 5 \text{ mA} = 360 \Omega$$

In this situation, I would use a 330- Ω current-limiting resistor.

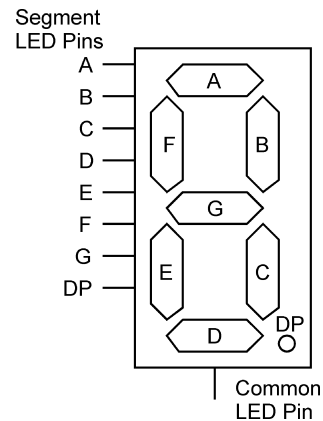


Figure 17.2 The seven-segment LED can have a common anode or cathode.

MULTISEGMENT LED DISPLAYS

Probably the easiest way to output numeric (both decimal and hex) data is via *seven-segment LED displays*. These displays were very popular in the seventies (if you're old enough, your first digital watch probably had seven-segment LED displays) but have been largely replaced by LCDs.

However, seven segment LED displays (Fig. 17.2) are still useful devices that can be added to a circuit without a lot of software effort. By turning on specific LEDs (each of which lights up a “segment” in the display), the display can be used to output decimal numbers.

Each one of the LEDs in the display is given an identifier, and a single pin of the LED is brought out of the package. The other LED pins are connected together and wired to a common pin. This common LED pin is used to identify the type of seven-segment display (as either common cathode or common anode). Wiring one display to a microcontroller is quite easy—it is typically wired as seven [or eight if the decimal point (DP) is used] LEDs wired to individual pins. The most important piece of work you'll do when setting up seven-segment LED displays is matching and documenting the microcontroller bits to the LEDs. Spending a few moments at the start of a project will simplify wiring and debug of the display later.

The typical method of wiring multiple seven-segment LED displays together is to wire them all in parallel and then control the current flow through the common pin. Because the current is generally too high for a single microcontroller pin, a transistor is used to pass the current to the common power signal. This transistor selects which display is active. In Fig. 17.3, four common-cathode seven-segment displays are shown connected to a microcontroller.

In this circuit, the microcontroller will shift between the displays, showing each digit in a very short time slice. This is usually done in a timer interrupt handler. The basis for the interrupt handler's code is listed below:

```
Int:
- Save Context Registers
- Reset Timer and Interrupt
```

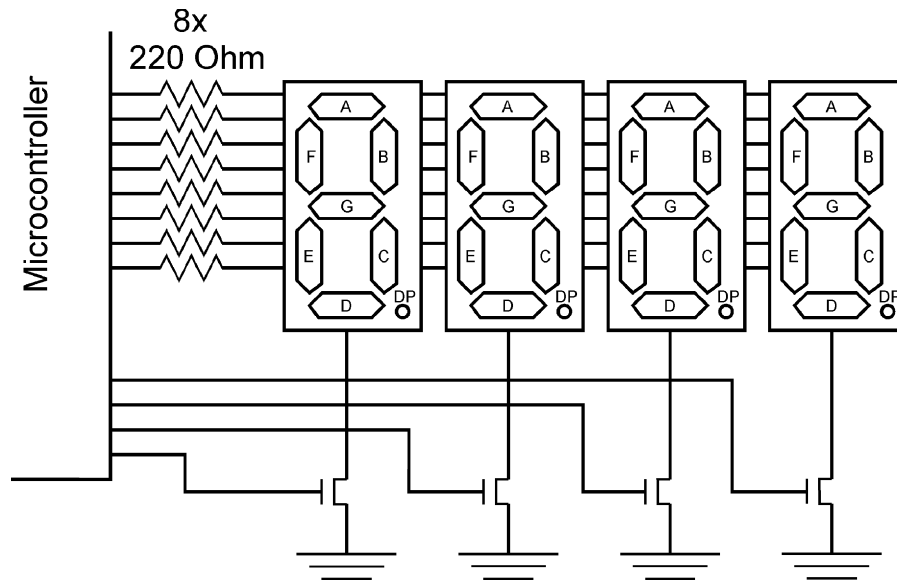


Figure 17.3 Multiple seven-segment LED displays can have common segment wiring with their common pins individually controlled, and the application scan through them faster than the human eye can perceive.

```

- LED_Display = 0 ; Turn Off all the LEDs
- LED_Output = Display[ Cur ]
- Cur = (Cur + 1) mod #Displays ; Point to Next
Sequence Display
- LED_Display = 1 << Cur ; Display LED for
Current Display
- Restore Context Registers
- Return from Interrupt

```

This code will cycle through each of the digits (and displays), having current go through the transistors for each one. To avoid flicker, I generally run the code so that each digit is turned on/off at least 50 times per second. The more digits you have, the faster you have to cycle the interrupt handler (i.e., eight seven-segment displays must cycle at least 400 digits per second, which is twice as fast as four displays).

You may feel that assigning a microcontroller bit to select each display LED to be somewhat wasteful (at least I do). I have used high-current TTL demultiplexor (i.e., 74S138) outputs as the cathode path to ground (instead of discrete transistors). When the output is selected from the demultiplexor, it goes low, allowing current to flow through the LEDs of that display (and turning them on). This actually simplifies the wiring of the final application as well. The only issue is to make sure that the demultiplexor output can sink the maximum of 140 mA of current that will come through the common-cathode connection.

Along with seven-segment displays, there are 14- and 16-segment LED displays available that can be used to display alphanumeric characters (A–Z and 0–9). By following the same rules as used when wiring up a seven-segment display, you shouldn't

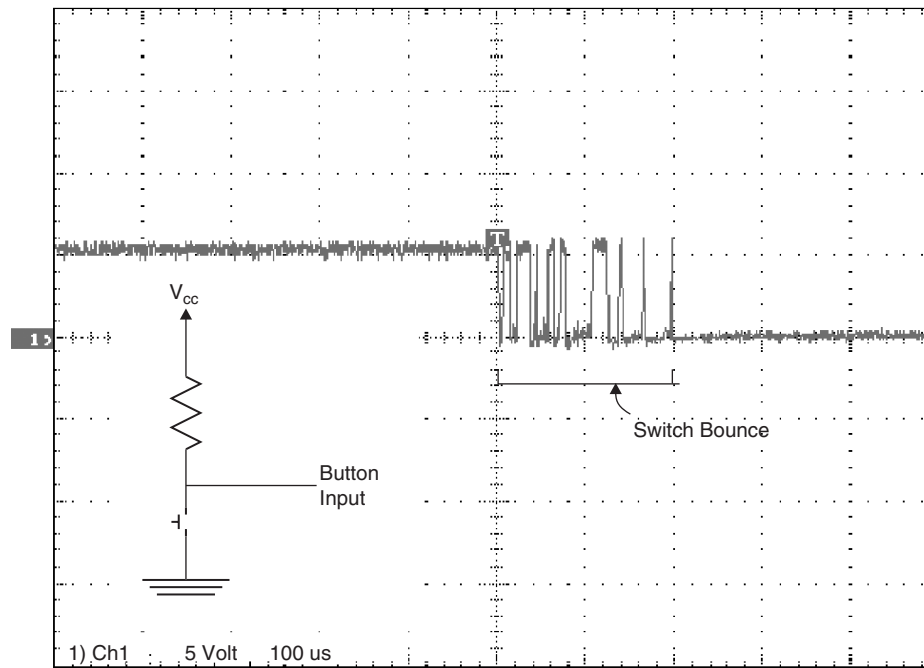


Figure 17.4 When a switch makes or breaks a connection, the contacts bounce against each other, possibly being interpreted as multiple button presses.

have any problems with wiring the display to a PIC microcontroller. In Chap. 21 I show how 7- and 16-segment LEDs can be used to display letters and numbers.

Switch Bounce

When a button is opened or closed, we perceive that as a “clean” operation that really looks like a step function. In reality, the contacts of a switch bounce when they make contact, resulting in the jagged signal shown in Fig. 17.4.

When this signal is passed to a PIC microcontroller, the microcontroller can recognize this as multiple button presses, which will cause the application software to act as if multiple, very fast button presses have taken place. To avoid this problem so that the switch press is treated like an idealized press, the step function I mentioned earlier and show in Fig. 17.5 is used. The signal will have to be *debounced*. There are two common methods used for debouncing button inputs.



Figure 17.5 The ideal waveform to come out of a switch is a clean transition like this one.

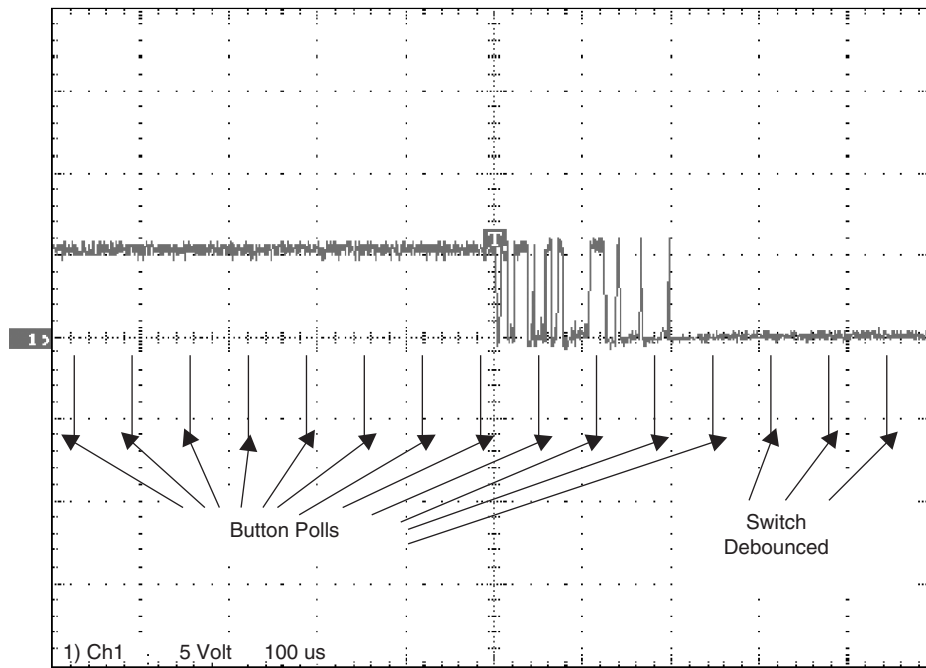


Figure 17.6 Polling the line continuously to wait for the bouncing to stop.

The first is to poll the switch line at short intervals until the switch line stays at the same level for an extended period of time. A button is normally considered debounced if it does not change state for 20 ms or more. By polling the line every 5 ms, this debouncing can be conceptualized quite easily, as shown in Fig. 17.6.

The advantage of this method is that it can be done in an interrupt handler, and the line can be scanned periodically with a flag set if the line is high and another flag if the line is low. For the indeterminate stage, neither bit would be set. This method is good for debouncing keyboard inputs.

The second method is to poll the line continually and wait for 20 ms to go by without the line changing state. The algorithm I use for this function is

```
do;
  while (Button == High);          // Poll Until Button is Pressed
  for (Dlay = 0; (Dlay < 20 ms) and (Button == Low); Dlay++);
until (Dlay >= 20 ms);
```

This code will wait for the button to be pressed and then poll it continuously until either 20 ms has passed or the switch has gone high again. If the switch goes high, the process is repeated until it is held low for 20 ms. This method is well suited to applications that don't have interrupts and only have one button input and no need for processing while polling the button. As restrictive as it sounds, there are many applications that fit these criteria.

This method also can be used with interrupt inputs along with TMR0 in the PIC microcontroller, which eliminates the restrictions I just detailed. The interrupt handler behaves

like the pseudocode below when one of the “port changes on interrupt” bits is used for the button input:

```

interrupt ButtonDebounce()    // Set Flags According to the
{                               // Debounced State of the Button

    if (T0IF == 1) {           // TMR0 Overflow, Button Debounced
        T0IF = 0; T0IE = 0;    // Reset and Turn off TMR0 Interrupts
        if (Button == High) {
            Pressed = 0; NotPressed = 1; // Set the State of the Button
        } else {
            Pressed = 1; NotPressed = 0;
        } // fi
    } else {                   // Port Change Interrupt
        Pressed = 0; NotPressed = 0; // Nothing True
        RBIF = 0;              // Reset the Interrupt
        TMR0 = 20msDlay;       // Reset Timer 0 for 20 msecs
        T0IF = 0; T0IE = 1;    // Enable the Timer Interrupt
    } // fi
} // End ButtonDebounce

```

This code waits for the input pin to change state and then resets the two flags indicating the button state and starts TRM0 to request an interrupt after 20 ms. After a port-change interrupt, notice that I reset the button state flags to indicate to the mainline that the button is in a transition state and is not yet debounced. If TMR0 overflows, then the button is polled for its state, and the appropriate button state flags are set and reset.

The mainline code should poll the `Pressed` and `NotPressed` flags when it is waiting for a specific state. In Chap. 20 I will show how this method using TMR0 and interrupts can be implemented with or without interrupts.

If you don't want to use the software approaches, you can use a capacitor to filter the bouncing signal and pass it into a Schmidt trigger input. Schmidt trigger inputs have different thresholds depending on whether the signal is rising or falling. For rising edges, the trigger point is higher than for falling edges. Schmidt trigger inputs have the symbol put in the buffer shown in the circuit presented in Fig. 17.7.

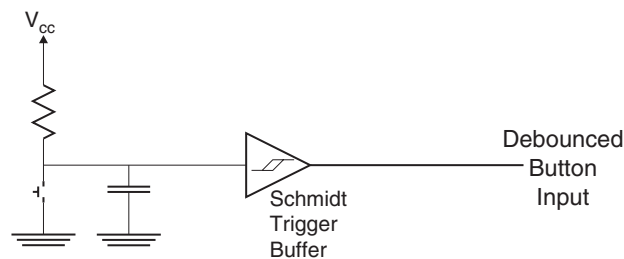


Figure 17.7 Bouncing into the PIC microcontroller can be eliminated by adding a filter to the button input line.

This method is fairly reliable but requires a Schmidt trigger gate in your circuit. There may be a Schmidt trigger input available in your PIC microcontroller, but check the datasheet to find out which states and peripheral hardware functions can take advantage of it.

One comment I want to make about debouncing switches is to choose buttons with a positive “click” when they are pressed and released. These have reduced bouncing and are a lot easier to work with than other switches that don’t have this feature. I have used a number of switches over the years that don’t have this click, and they can be a real problem in circuits with intermittent connections and unexpected bouncing that occurs while the button is pressed and held down.

Matrix Keypads

Switch matrix keyboards and keypads are really just an extension to the simple buttons of the preceding section with many of the same concerns and issues to watch out for. The big advantage that a matrix keyboard gives you is the ability to handle a large number of pins for a relatively small number of PIC microcontroller pins. The PIC microcontroller is well designed for simply implementing switch matrix keypads, which, like liquid-crystal displays (LCDs) explained in the next section, can add a lot to your application with a very small investment in hardware and software.

A switch matrix is simply a two-dimensional matrix of wires with switches at each vertex. The switch is used to interconnect rows and columns in the matrix, as can be seen in Fig. 17.8. This diagram may not look like the simple button, but it will become more familiar when I add pull-ups on the rows and switchable ground connections on the columns, as I show in Fig. 17.9.

In this case, by connecting one of the columns to ground, if a switch is closed, the pull-down on the row will connect the line to ground. When the row is polled by an I/O pin, a 0, or low voltage, will be returned instead of a 1 (which is what will be returned if the switch in the row that is connected to the ground is open).

As a rule of thumb, the number of PIC microcontroller I/O pins required to implement a switch matrix keypad is twice the square root of the number of keys rounded up to the nearest whole number. For example, if you want to implement a 29-button keypad

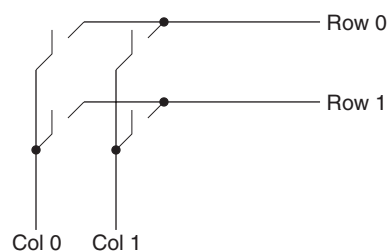


Figure 17.8 Multiple buttons can be wired together in a matrix to minimize the number of I/O pins required.

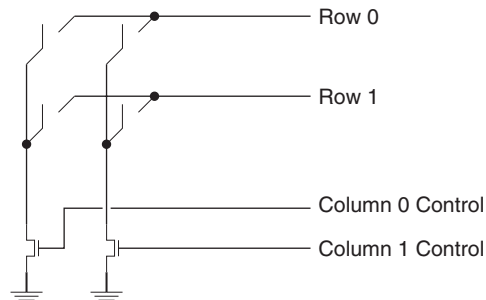


Figure 17.9 To read the state of the buttons, the columns are brought to ground, and the pulled-up rows are polled.

in your application, you would find the square root of 29 (which is 5.4) and round it up to the nearest whole number (6) and double it to discover that you should plan for 12 PIC I/O pins for the keypad. Again, this is a basic rule of thumb; I'm sure that you're thinking that the optimal number of PIC I/O pins would be 11. The error comes in because 29 is less than halfway between 25 and 36 (two evenly square rooted numbers); if it were 31, you would need 12 I/O pins to support the keypad.

As I said earlier, the PIC microcontroller is well suited to implementing switch matrix keyboards with PORTB's internal pull-ups and the ability of the I/O ports to simulate the open-drain pull-downs of the columns, as shown in Fig. 17.10. Normally, the pins

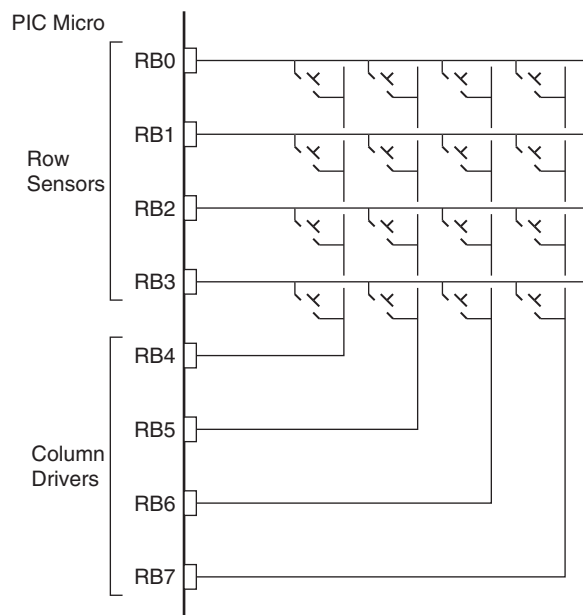


Figure 17.10 A 16-button keypad would be wired as shown here, requiring only eight PIC I/O pins.

670 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

connected to the columns are left in tristate (input) mode. When a column is being scanned, the column pin is output-enabled, driving a 0, and the four input bits are scanned to see if any are pulled low.

In this case, the keyboard can be scanned for any closed switches (buttons pressed) using the code

```
int KeyScan()                // Scan the Keyboard and Return
when a                       // key is pressed
{

int i = 0;
int key = -1;

while (key == -1) {

for (i = 0; (i < 4) & ((PORTB & 0x00F) == 0x0F0); i++);

switch (PORTB & 0x00F) {     // Find Key that is Pressed
case 0x00E:                 // Row 0
    key = i;
    break;
case 0x00D:                 // Row1
case 0x00C:
    key = 0x04 + i;
    break;
case 0x00B:                 // Row2
case 0x00A:
case 0x009:
case 0x008:
    key = 0x08 + i;
    break;
else                         // Row3
    key = 0x0C + i;
    break;
} // hctiws
} // elihw

return key;

} // End KeyScan
```

The `KeyScan` function will only return when a key has been pressed. This routine will not allow keys to be debounced or for other code to execute while it is executing.

These issues can be resolved by putting the key scan into an interrupt handler that executes every 5 ms:

```
Interrupt KeyScan( )        // 5 msec Interval Keyboard Scan
{
```

```

int i = 0;
int key = -1

for (i = 0; (i <4) & ((PORTB & 0x00F) == 0x00F)); i++);

if (PORTB & 0x00F) != 0x00F) { // Key Pressed
  switch (PORTB & 0x00F) { // Find Key that is Pressed
    case 0x00E: // Row 0
      key = i;
      break;
    case 0x00D: // Row1
    case 0x00C:
      key = 0x04 + i;
      break;
    case 0x00B: // Row2
    case 0x00A:
    case 0x009:
    case 0x008:
      key = 0x08 + i;
      break;
    else // Row3
      key = 0x0C+i;
      break;
  } // hctiws
  if (key == KeySave) {
    keycount = keycount + 1; // Increment Count
    // <-- Put in Auto Repeat Code Here
    if (keycount == 4)
      keyvalid = key; // Debounced Key
  } else
    keycount = 0; // No match - Start Again
  KeySave = key; // Save Current key for next
    // 5 ms
  } // fi // Interval
} // End KeySave

```

This interrupt handler will set `keyvalid` variable to the row/column combination of the key button (which is known as a *scan code*) when the same value comes up four times in a row. This time scan is the debounce routine for the keypad. If the value doesn't change for four intervals (20 ms in total), the key is determined to be debounced.

There are two things to notice about this code. The first is that in both routines I handle the row with the highest priority. If multiple buttons are pressed, then the one with the highest bit number will be the one that is returned to the user. The second point is that this code can have an autorepeat function added to it very easily. To do this, a secondary counter has to be first cleared and then incremented each time the `keycount`

variable is 4 or greater. To add an autorepeat key every second (200 intervals), the following code is added in the interrupt handler

```
if (keycount == 4) {
    keyrepeat = keyrepeat - 1; // Decrement the Key Auto Repeat Value
    if (keyrepeat == 0) {
        keyrepeat = 200; // Restart the 1 second Auto
                        // Repeat count
        keycount = 3; // Reset the counter
        keyvalid = key; // Return the key
    }
} else // Reset the Auto Repeat Counter
    keyrepeat = 1; // End Outputting the Value with Auto
                // Repeat
```

LCDs

LCDs can add a lot to your application in terms of providing a useful interface for the user, debugging an application, or just giving it a professional look. The most common type of LCD controller is the Hitachi 44780, which provides a relatively simple interface between a processor and an LCD. Using this interface is often not attempted by new designers and programmers because it is difficult to find good documentation on the interface, initializing the interface can be a problem, and the displays themselves are expensive.

I have worked with Hitachi 44780-based LCDs for a while now, and I have to say that I don't believe any of these perceptions. LCDs can be added quite easily to an application and use as few as three digital output pins for control. As for cost, LCDs often can be pulled out of old devices or found in surplus stores for less than a dollar.

The purpose of this section is to give you a brief tutorial on how to interface with Hitachi 44780-based LCDs. I have tried to provide the all the data necessary for adding LCDs successfully to your application. In the book I use Hitachi 44780-based LCDs for a number of different projects.

The most common connector used for 44780-based LCDs is 14 pins in a row, with pin centers 0.100 in apart, with the pinout listed in Table 17.1.

As you probably would guess from this description, the interface is a parallel bus that allows simple and fast reading/writing of data to and from the LCD. This waveform to write an ASCII byte out to the LCD's screen is shown in Fig. 17.11. The ASCII code to be displayed is 8 bits long and is sent to the LCD either 4 or 8 bits at a time. If 4-bit mode is used, two nybbles of data (sent high 4 bits and then low 4 bits with an E clock pulse with each nybble) are sent to make up a full 8-bit transfer. The E clock is used to initiate the data transfer within the LCD.

Sending parallel data as either 4 or 8 bits is the primary mode of operation. While there are secondary considerations and modes, deciding how to send the data to the LCD is the

PIN	DESCRIPTION
1	Ground
2	Vcc
3	Contrast voltage
4	R/S—Instruction/register select
5	R/W—Read/write LCD registers
6	E—Clock
7–14	D0–D7—Data pins

most critical decision to be made for an LCD interface application. Eight-bit mode is best used when speed is required in an application and at least 10 I/O pins are available. Four-bit mode requires a minimum of 6 bits. To wire a microcontroller to an LCD in 4-bit mode, just the top 4 bits (DB4–7) are written to, with first the high nybble followed by the low nybble (which will be described in the next section).

The R/S bit is used to select whether data or an instruction is being transferred between the microcontroller and the LCD. If the bit is set, then the byte at the current LCD cursor position can be read or written. When the bit is reset, either an instruction is being sent to the LCD or the execution status of the last instruction is read back (whether or not it has completed).

The different instructions available for use with the 44780 are shown in Table 17.2, and the bit descriptions for the different commands are

Set cursor move direction:

ID—Increment the cursor after each byte written to display if set

S—Shift display when byte written to display

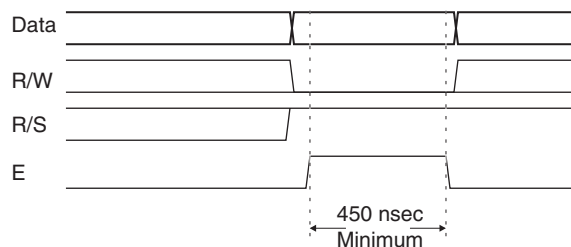


Figure 17.11 The Hitachi 44780-based LCD data write waveform.

674 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

TABLE 17.2 HITACHI 44780—BASED LCD CONTROLLER COMMANDS										
R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	INSTRUCTION/ DESCRIPTION
4	5	14	13	12	11	10	9	8	7	Pins
0	0	0	0	0	0	0	0	0	1	Clear display
0	0	0	0	0	0	0	0	0	1	* Return cursor and LCD to home position
0	0	0	0	0	0	0	1	ID	S	Set cursor move direction
0	0	0	0	0	0	1	D	C	B	Enable display/cursor
0	0	0	0	0	1	SC	RL	*	*	Move cursor/shift display
0	0	0	0	1	DL	N	F	*	*	Reset/set interface length
0	0	0	1	A	A	A	A	A	A	Move cursor to CGRAM
0	0	1	A	A	A	A	A	A	A	Move cursor to display
0	1	BF	*	*	*	*	*	*	*	Poll the busy flag
1	0	H	H	H	H	H	H	H	H	Write hex character to the display at the current cursor position
1	1	H	H	H	H	H	H	H	H	Read hex character at the current cursor position on the display

*Not used/ignored. This bit can be either 1 or 0.

Enable display/cursor:

D—Turn display on(1)/off(0)

C—Turn cursor on(1)/off(0)

B—Cursor blink on(1)/off(0)

Move cursor/shift display:

SC—Display shift on(1)/off(0)

RL—Direction of shift right(1)/left(0)

Set interface length:

DL—Set data interface length 8(1)/4(0)

N—Number of display lines 1(0)/2(1)

F—Character font 5 × 10(1)/5 × 7(0)

Poll the busy flag:

BF—This bit is set while the LCD is processing

Move cursor to CGRAM/display:

A—Address

Read/write ASCII to the display:

H—Data

Reading data back is best used in applications that require data to be moved back and forth on the LCD (such as in applications that scroll data between lines). The busy flag can be polled to determine when the last instruction sent has completed processing.

For most applications, there really is no reason to read from the LCD. I usually tie R/W to ground and just wait the maximum amount of time for each instruction (4.1 ms for clearing the display or moving the cursor/display to the home position; 160 μ s for all other commands). As well as making my application software simpler, this also frees up a microcontroller pin for other uses. Different LCDs execute instructions at different rates, and to avoid problems later on (such as if the LCD is changed to a slower unit), I recommend just using the maximum delays listed here.

In terms of options, I have never seen a 5×10 LCD display. This means that the F bit in the `SetInterface` instruction always should be reset (equal to 0).

Before you can send commands or data to the LCD module, the module must be initialized. For 8-bit mode, this is done using the following series of operations:

- 1 Wait more than 15 ms after power is applied.
- 2 Write 0x030 to LCD and wait 5 ms for the instruction to complete.
- 3 Write 0x030 to LCD and wait 160 μ s for the instruction to complete.
- 4 Write 0x030 *again* to LCD and wait 160 μ s or poll the busy flag.
- 5 Set the operating characteristics of the LCD:
 - Write `SetInterface` length.
 - Write 0x010 to turn off the display.
 - Write 0x001 to clear the display.
 - Write “set cursor move direction” to set cursor behavior bits.
 - Write “enable display/cursor” to enable display and optional cursor.

In describing how the LCD should be initialized in 4-bit mode, I will specify writing to the LCD in terms of nybbles. This is so because initially, just single nybbles are sent (and not two, which make up a byte and a full instruction). As I mentioned earlier, when a byte is sent, the high nybble is sent before the low nybble, and the E pin is toggled each time 4 bits are sent to the LCD. To initialize in 4-bit mode, the following sequence of commands is sent:

- 1 Wait more than 15 ms after power is applied.
- 2 Write 0x03 to LCD and wait 5 ms for the instruction to complete.
- 3 Write 0x03 to LCD and wait 160 μ s for instruction to complete.
- 4 Write 0x03 *again* to LCD and wait 160 μ s (or poll the busy flag).
- 5 Set the operating characteristics of the LCD:
 - Write 0x02 to the LCD to enable 4-bit mode.

676 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

All following instruction/data writes require two-nybble writes:

- Write `SetInterface` length.
- Write `0x01/0x00` to turn off the display.
- Write `0x00/0x01` to clear the display.
- Write “set cursor move direction” to set cursor behavior bits
- Write “enable display/cursor” to enable display and optional cursor.

Once the initialization is complete, the LCD can be written to with data or instructions as required. Each character to display is written like the control bytes, except that the R/S line is set. During initialization, by setting the S/C bit during the “move cursor/shift display” command, after each character is sent to the LCD, the cursor built into the LCD will increment to the next position (either right or left). Normally, the S/C bit is set (equal to 1) along with the R/L bit in the “move cursor/shift display” command for characters to be written from left to right (as with a teletype video display).

One area of confusion is how to move to different locations on the display and, as a follow-on, how to move to different lines on an LCD display. Table 17.3 lists how different LCD displays that use a single 44780 can be set up with the addresses for specific

TABLE 17.3 HITACHI 44780—BASED LCD TYPES AND CHARACTER LOCATIONS

LCD	TOP LEFT	NINTH	SECOND LINE	THIRD LINE	FOURTH LINE	COMMENTS
8 × 1	0	N/A	N/A	N/A	N/A	Note 1
16 × 1	0	0x040	N/A	N/A	N/A	Note 1
16 × 1	0	8	N/A	N/A	N/A	
8 × 2	0	N/A	0x040	N/A	N/A	Note 1
10 × 2	0	0x008	0x040	N/A	N/A	Note 3
16 × 2	0	0x008	0x040	N/A	N/A	Note 2
20 × 2	0	0x008	0x040	N/A	N/A	Note 2
24 × 2	0	0x008	0x040	N/A	N/A	Note 2
30 × 2	0	0x008	0x040	N/A	N/A	Note 2
32 × 2	0	0x008	0x040	N/A	N/A	Note 2
40 × 2	0	0x008	0x040	N/A	N/A	Note 2
16 × 4	0	0x008	0x040	0x020	0x040	Note 2
20 × 4	0	0x008	0x040	0x020	0x040	Note 2
40 × 4	0	N/A	N/A	N/A	N/A	Note 4

Notes:

¹Single 44780/no support chip.

²44780 with support chip.

³44780 with support chip. This is quite rare.

⁴Two 44780s with support chips. Addressing is device-specific.

character locations. The LCDs listed are the most popular arrangements available, and the layout is given as number of columns by number of lines.

The “ninth character” is the position of the ninth character on the first line. Most LCD displays have a 44780 and support chip to control the operation of the LCD. The 44780 is responsible for the external interface and provides sufficient control lines for 16 characters on the LCD. The support chip enhances the I/O of the 44780 to support up to 128 characters on an LCD in two lines of eight. From the table, it should be noted that the first two entries (8×1 and 16×1) only have the 44780 and not the support chip. This is why the ninth character in the 16×1 does not appear at address 8 and shows up at the address that is common for a two-line LCD.

I’ve included the 40-character by 4-line (40×4) LCD because it is quite common. Normally, the LCD is wired as two 40×2 displays. The actual connector is normally 16 bits wide, with all 14 connections of the 44780 in common, except for the E (strobe) pins. The individual E strobe lines are used to select between the areas of the display used by the two devices. The actual pinouts and character addresses for this type of display can vary among manufacturers and display part numbers. Note that when using any kind of multiple-44780 LCD display, you probably should display the cursor of only one of the 44780s at a time.

Cursors for the 44780 can be turned on as a simple underscore at any time using the “enable display/cursor” LCD instruction and setting the C bit. I don’t recommend using the B (block mode) bit because this causes a flashing full-character square to be displayed, and it really isn’t that attractive.

The LCD can be thought of as a teletype display because in normal operation, after a character has been sent to the LCD, the internal cursor is moved one character to the right. The “clear display” and “return cursor and LCD to home position” instructions are used to reset the cursor’s position to the top right character on the display. An example of moving the cursor is shown in Fig. 17.12.

To move the cursor, the “move cursor to display” instruction is used. For this instruction, bit 7 of the instruction byte is set, with the remaining 7 bits used as the address

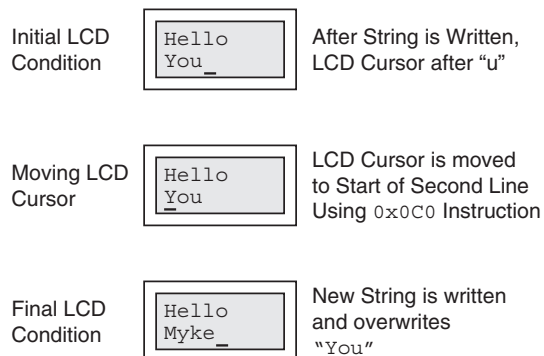


Figure 17.12 Text appears at the current LCD cursor position; to overwrite characters, the cursor can be moved.

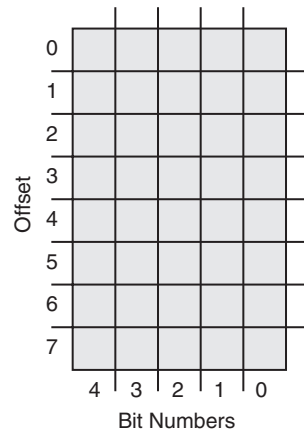


Figure 17.13 The custom LCD characters have to fit in a 7×5 pixel box.

of the character on the LCD to which the cursor is to move. These 7 bits provide 128 addresses, which matches the maximum number of LCD character addresses available. Table 17.3 should be used to determine the address of a character offset on a particular line of an LCD display.

The character set available in the 44780 is basically ASCII. I say *basically* because some characters do not follow the ASCII convention fully (probably the most significant difference is 0x05B, or \, is not available). The ASCII control characters (0x008 to 0x01F) do not respond as control characters and may display funny (Japanese) characters. Eight programmable characters are available and use codes 0x000 to 0x007. They are programmed by pointing the LCD's cursor to the character generator RAM (CGRAM) area at eight times the character address. The next 8 bytes written to the RAM are the line information of the programmable character starting from the top. The "character box" representation is shown in Fig. 17.13.

I like to represent this as eight squares by five, as is shown in the diagram to the right. Earlier I noted that most displays were 7 pixels by 5 for each character, so the extra row may be confusing. Each LCD character is actually 8 pixels high, with the bottom row normally used for the underscore cursor. The bottom row can be used for graphic characters, although if you are going to use a visible underscore cursor and have it at the character, I recommend that you don't use it (i.e., set the line to 0x000).

Using this box, you can draw in the pixels that define your special character and then use the bits to determine what the actual data codes are. When I do this, I normally use a piece of graph paper and then write hex codes for each line, as I show in Fig. 17.14, to produce a diagram of a simple "smiley face."

For some "animate" applications, I use character rotation for the animations. This means that instead of changing the character each time the character moves, I simply display a different character. Doing this means that only 2 bytes (moving the cursor to the character and the new character to display) have to be sent to the LCD. If animation were accomplished by redefining the characters, then 10 characters would have to be sent to

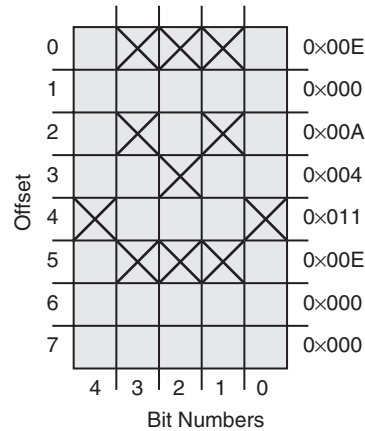


Figure 17.14 A custom character definition.

the LCD (one to move into the CGRAM space, the eight defining characters, and an instruction returning to display RAM). If multiple characters are going to be used or more than eight pictures for the animation, then you will have to rewrite the character each time.

The user-defined character line information is saved in the LCD's CGRAM area. This 64-byte space of memory is accessed using the "move cursor into CGRAM" instruction in a similar manner to that of moving the cursor to a specific address in the memory with one important difference. This difference is that each character starts at eight times its character value. This means that user-definable character 0 has its data starting at address 0 of the CGRAM, character 1 starts at address 8, character 2 starts at address 0x010 (16), and so on. To get a specific line within the user-definable character, its offset from the top (the top line has an offset of 0) is added to the starting address. In most applications, characters are written to all at one time with character 0 first. In this case, the instruction 0x040 is written to the LCD, followed by all the user-defined characters.

The last aspect of the LCD to discuss is how to specify a contrast voltage to the display. I typically use a potentiometer wired as a voltage divider (Fig. 17.15). This will provide an easily variable voltage between ground and V_{CC} that will be used to specify the contrast (or darkness) of the characters on the LCD screen. You may find that different LCDs work differently, with lower voltages providing darker characters in some and higher voltages doing the same thing in others.

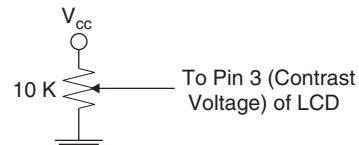


Figure 17.15 Contrast control can be provided by a pot wired as a voltage divider.

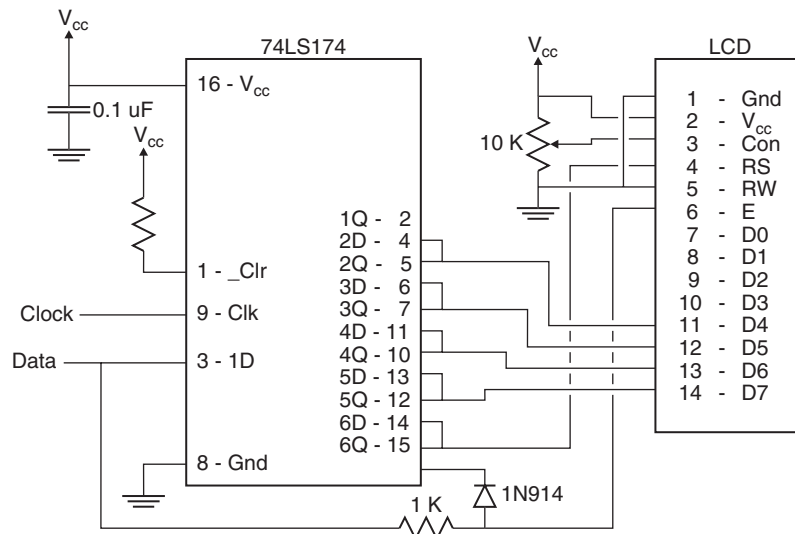


Figure 17.16 With this circuit, you can control an LCD with just two wires.

TWO-WIRE LCD CONTROL

There are a number of different ways to wire up an LCD. Earlier I noted that the 44780 could interface with 4 or 8 bits. To simplify the demands of microcontrollers, a shift register is often used to reduce the number of I/O pins to three. This can be further reduced by using the circuit shown in Fig. 17.16, in which the serial data is combined with the contents of the shift register to produce the E strobe at the appropriate interval. This circuit ANDs (using the 1-k Ω resistor and 1N914 diode) the output of the sixth D flip-flop of the 74LS174 and the data bit from the device writing to the LCD to form the E strobe. This method requires one less pin than the three-wire interface and a few more instructions of code.

I normally use a 74LS174 wired as a shift register (as shown in the schematic diagram) instead of a serial-in/parallel-out shift register. This circuit should work without any problems with a dedicated serial-in/parallel-out shift register chip, but the timings/clock polarities may be different. When the 74LS174 is used, note that the data is latched on the rising edge (from logic low to high) of the clock signal. Figure 17.17 is a timing diagram for the two-wire interface and shows the 74LS174 being cleared, loaded, and then the E strobe when the data is valid and 6Q and incoming data are high.

In the diagram to the right I have shown how the shift register is written to for this circuit to work. Before data can be written to it, loading every latch with zeros clears the shift register. Next, a 1 (to provide the E gate) is written, followed by the R/S bit and the 4 data bits. Once the latch is loaded correctly, the data line is pulsed to strobe the E bit. The biggest difference between the three-wire and the two-wire

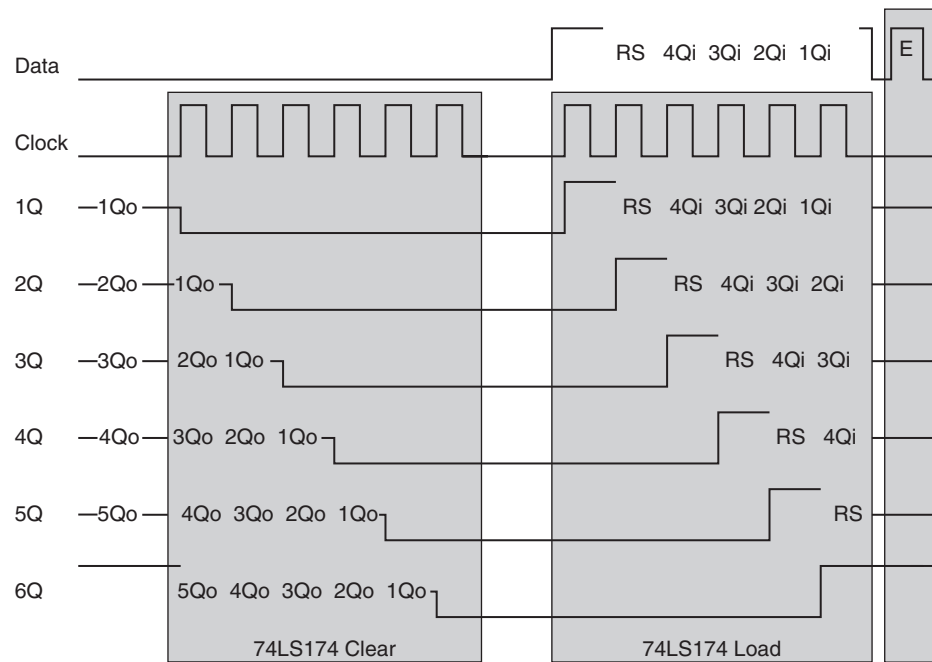


Figure 17.17 The synchronous serial waveform of the data coming in showing the bits propagating through the shift register.

interface is that the shift register has to be cleared before it can be loaded, and the two-wire operation requires more than twice the number of clock cycles to load 4 bits into the LCD.

I've used this circuit with the PIC microcontroller, BASIC Stamp, 8051, and AVR, and it really makes the wiring of an LCD to a microcontroller very simple. A significant advantage of using a shift register, as in the two circuits shown here, is the lack of timing sensitivity that will be encountered. The biggest issue to watch for is to make sure that the E strobe's timing is within specification (i.e., greater than 450 ns); the shift register loads can be interrupted without affecting the actual write. This circuit will not work with open-drain-only outputs (something that catches up many people).

One note about the LCD's E strobe is that in some documentation it is specified as high level active, whereas in others it is specified as falling-edge-active. It *seems* to be falling-edge-active, which is why the two-wire LCD interface presented below works even if the line ends up being high at the end of data being shifted in. If the falling edge is used (as in the two-wire interface), then make sure that before the E line is output on 0, there is at least a 450-ns delay with no lines changing state.

The following C routine could be used to write to the two-wire LCD interface:

```
LCDNybble(char Nybble, char RS)
{
int i;
```

682 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

```
Data = 0; // Clear the '174
for (i = 0; i < 6; i++) { // Repeat for six bits
    Clock = 1; Clock = 0; // Write the "0"s into the '174
} // rof

Data = 1; // Output the "AND" Value
Clock = 1; Clock = 0;

Data = RS; // Output the RS Bit Value
Clock = 1; Clock = 0;

for (i = 0; i < 4; i++) { // Output the Nybble
    if ((Nybble & 0x008) != 0)
        Data = 1; // Output the High Order Bit
    else
        Data = 0;
    Clock = 1; Clock = 0; // Strobe the Clock
    Nybble = Nybble << 1; // Shift up Nybble for Next Byte
} // rof

Clock = 1; Clock = 0; // Toggle the "E" Clock Bit

} // End LCDNybble
```

Analog I/O

In the following sections I want to introduce you to some of the practical aspects of working with analog data with the PIC microcontroller. This includes position sensing using potentiometers. At first glance, you may think that an ADC-equipped microcontroller is required for this operation, but there are a number of ways of doing this with strictly digital inputs. Along with discussing how it can be done with ADCless microcontrollers, I'll also show how the IBM PC carries out the function (it doesn't use an ADC either).

For analog output, I will focus on the theory and operation behind PWM analog control signals. This method of control is very popular and is a relatively simple way to provide analog control of a device. It also can be used for communication of analog values between devices without needing any type of communication protocol. The PIC microcontroller has some built-in hardware that makes the implementation of PWM input and output quite easy to do.

While I discuss audio I/O, I want to make it clear that audio input and output capabilities cannot be provided in the PIC microcontroller without significant front-end signal processing and filtering. Output from the PIC microcontroller is limited to simple beeps without special hardware.



Figure 17.18 A potentiometer wired as a voltage divider can be read using the PIC microcontroller's ADC input.

READING POTENTIOMETERS

One of the more useful human input devices is the dial. Rather than relying on some kind of digital data such as a button or character string, the dial allows users a freer range of inputs, as well as positional feedback information, in a mechanical device. For most people, reading a potentiometer value requires setting the potentiometer as a voltage divider and reading the voltage between the two extremes at the wiper, as shown in Fig. 17.18. However, there is a very elegant way of reading a potentiometer's position using the digital input of a PIC microcontroller, as I will show in this section.

Note that I consider the measurement to be of the potentiometer's position, not its resistance. This is an important semantic point; as far as using a potentiometer as an input device is concerned, I do not care what the actual resistance is of its position, just what the position is. The method of reading a potentiometer's position using a digital I/O pin that I am going to show you depends very much on the parts used and will vary significantly between implementations.

The method of reading a potentiometer uses the characteristics of a charged capacitor discharging through a resistor. If the charge is constant in the capacitor, then the time to discharge varies according to the exponential curve shown in Fig. 17.19.

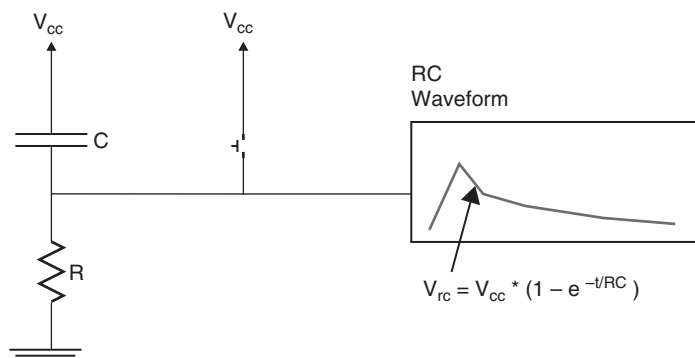


Figure 17.19 A capacitor's charge-decay response can be modified by a potentiometer, and the time taken can be measured by a PIC microcontroller.

684 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

The charge in a capacitor is proportional to its voltage. If a constant voltage (i.e., from a PIC microcontroller I/O pin) can be applied to a capacitor, then its charge will be constant. This means that in the voltage-discharge curve shown in Fig. 17.19, if the initial voltage is known, along with the capacitance and resistance, then the voltage at any point in time can be predicted.

The equation in the figure, i.e.,

$$V(t) = V_{\text{start}}(1 - e^{-t/RC})$$

can be reworked to find R if V , V_{start} , t , and C are known:

$$R = -t/C * \ln((V_{\text{start}} - V)/V_{\text{start}})$$

Rather than calculate the value, though, you can make the approximation of 2 ms for a resistance of 10 k Ω and a capacitance of 0.1 μF with a PIC microcontroller that has a high-to-low threshold of 1.5 V. To measure the resistance in a PIC microcontroller, I use the circuit shown in Fig. 17.20. In this circuit, the PIC microcontroller's I/O pin outputs a high that charges the capacitor (with some leakage through the potentiometer resistor).

After the capacitor is charged, the pin is changed to input, and the charge in the capacitor draws through the resistor with a voltage determined by the $V(t)$ formula. When the pin first changes state, the voltage across the resistor will be greater than the threshold for some period of time. When the voltage across the potentiometer falls below the voltage threshold, the input pin value returned to the software will be 0. If the time required for voltage across the pin to go from a 1 to a 0 is recorded, it will be proportional to the resistance between the potentiometer's wiper and the capacitor.

The pseudocode for carrying out the potentiometer read is

```
int ReadPot() // Return the
Potentiometer's Position
{
    int i;
```

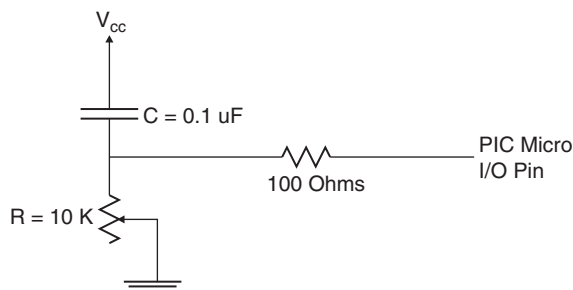


Figure 17.20 A circuit that can measure the resistance of a potentiometer by the time it takes for a capacitor to discharge through it.

```
pin = output; pin = 1;           // Charge the Capacitor
for (i = 0; i < charge; i++);
pin = input;                     // Let the Capacitor Discharge
for ( i = 0; pin == 1; i++);
return I ;

} // End ReadPot
```

The PIC microcontroller assembly-language code for implementing this potentiometer read is not much more complex than this pseudocode. Later in this book I will give you some examples of how potentiometer reads are actually accomplished.

The 100- Ω resistor between the PIC microcontroller pin and the RC network is used to prevent any short circuits to ground if the potentiometer is set so that there is no resistance in the circuit at all. This method of reading a potentiometer's position is very reliable, but it is not very accurate, nor is it particularly fast. When setting this up for the first time in a specific circuit, you will have to experiment to find the actual range it will display. This is due to part variances (including the PIC microcontroller) and the power-supply characteristics. For these reasons, I do not recommend using the potentiometer/capacitor circuit in any products. Tuning the values returned will be much more expensive than the cost of a PIC microcontroller with a built-in ADC.

PWM I/O

The PIC microcontroller, like most other digital devices, does not handle analog voltages very well. This is especially true for situations where high-current voltages are involved. The best way to handle analog voltages is to use a string of varying-wide pulses to indicate the actual voltage level. This string of pulses is known as a PWM analog signal and can be used to pass analog data from a digital device, control dc devices, or even output an analog voltage. In this section I want to discuss PWM signals and how they can be used with the PIC microcontroller. In the discussion of TMR1 and TMR2 earlier in this book, I presented how PWM signals were implemented and read using the CCP built-in hardware of the PIC microcontroller. For this section I will show how PWM signals can be used for I/O in PIC microcontrollers that do not have the CCP module built in.

A PWM signal is a repeating signal that is on for a set period of time that is proportional to the voltage that is being output. A PWM signal is shown in Fig. 17.21. I call the on time the *pulse width* in the figure, and the *duty cycle* is the percentage of the on time relative to the PWM signal's *period*.

To output a PWM signal, the following code could be implemented, although there is no way of changing the values while it is running (unless you were to include an interrupt handler):

```
Period = PWMPeriod;           // Initialize the Output
On = PWMOn;                   // Parameters
```

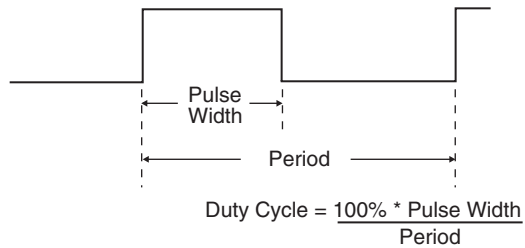


Figure 17.21 The basic PWM waveform with its features labeled.

```
while (1 == 1) {
    PWM = ON;                // Start the Pulse
    for (i = 0; i < On; i++ ); // Output ON for "On" Period of
                                // Time
    PWM = off;               // Turn off the Pulse
    For ( ; i < PWMPeriod; i++ ) ; // Output off for the rest of the
                                // PWM Period
} // end while
```

To avoid the problem of all the resources being devoted to the PWM code, I would recommend using the TMR0 interrupt to create the PWM timing:

```
Interrupt PWMOutput() // When Timer Overflows, Toggle "On" and "Off"
{ // and Reset Timer to correct delay for Value

    if (PWM == ON) { // If PWM is ON, Turn it off and Set Timer
        PWM = off; // Value
        TMR0 = PWMPeriod - PWMOn;
    } else { // If PWM is off, Turn it ON and Set Timer
        PWM = ON; // Value
        TMR0 = PWMOn;
    } // fi

    INTCON.T0IF = 0; // Reset Interrupts

} // End PWMOutput TMR0 Interrupt Handler
```

This code is quite easy to port into PIC microcontroller assembly language. For example, if the PWM period were 1 ms (executing in a 4-MHz PIC microcontroller), a divide by four prescaler value could be used with the timer, and the interrupt handler assembly-language code would be

```
org 4
Int: ; Interrupt Handler
```

```

    movwf  _w          ; Save Context Registers
    movf   STATUS, w   ; - Assume TMR0 is the only enabled Interrupt
    movwf  _status
    btfsc  PWM         ; Is PWM O/P Currently High or Low?
    goto  PWM_ON
    nop                    ; Low - Nop to Match Cycles with High

    bsf    PWM         ; Output the Start of the Pulse

    movlw  6 + 6       ; Get the PWM On Period
    subwf  PWMOn, w    ; Add to PWM to Get Correct Period for
; Interrupt Handler Delay and Missed cycles
; in maximum 1024 μsec Cycles
    goto  PWM_Done

PWM_ON:          ; PWM is On - Turn it Off

    bcf    PWM         ; Output the "Low" of the PWM Cycle

    movf   PWMOn, w    ; Calculate the "Off" Period
    sublw  6 + 6       ; Subtract from the Period for the Interrupt
; Handler Delay and Missed cycles in maximum
; 1024 μsec Cycles
    goto  PWM_Done

PWM_Done:       ; Have Finished Changing the PWM Value
    sublw  0          ; Get the Value to Load into the Timer
    movwf  TMR0

    bcf    INTCON, T0IF ; Reset the Interrupt Handler

    movf   _status, w  ; Restore the Context Registers
    movwf  STATUS
    swapf  _w, f
    swapf  _w, w

    retfie

```

In this code, TMR0 is loaded in such a way that the PWM period is always 1 ms (or a count of 250 “ticks” with the prescaler value of 4). To get the value added and subtracted from the total, I first took the difference between the number of ticks to get 1 ms (250) and the full timer range (256). Next, I counted the total number of instruction cycles of the interrupt handler (which is 23) and divided it by 4 and added the result to the 1-ms difference. The operation probably seems confusing because I was able to optimize the time for the PWM signal off:

Time Off = Period - ON

to

688 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

```

movf   ON, w
  sublw 6 + 6
  sublw 0
movwf  TMR0

```

I realize that this doesn't make any sense the first time you look at it, and I will go through it by showing how it works. Using the original equation, you should note that this calculates the number of cycles to be delayed by TMR0, but the actual value to be loaded into TMR0 is calculated as

$$\begin{aligned}
 \text{TMR0 Delay Value} &= 0x100 - (\text{Time Off}) \\
 &= 0x100 - (\text{Period} - \text{ON}) \\
 &= 0x100 - (256 - 250 + \text{Interrupt Execution} - \text{ON}) \\
 &= 0x100 - (6 + 6 - \text{ON}) \\
 &= 0x100 - (12 - \text{ON}) \\
 &= 0x100 - 12 + \text{ON} \\
 &= 0xF4 + \text{ON}
 \end{aligned}$$

Going back to the three instructions that load TMR0, you can show that they execute as

```

movf   ON, w           ; w = ON
sublw  6 + 6          ; w = 6 + 6 - w
                        ; = 12 - ON
                        ; = 12 + 0xFF ^ ON + 1
sublw  0              ; = 13 + 0xFF ^ ON
                        ; w = 0 - w
                        ; = 0 - (13 + 0xFF ^ ON)
                        ; = 0 + 0xFF ^ (13 + 0xFF ^ ON) + 1
                        ; = 0xFF ^ 13 + 0xFF ^ 0xFF ^ ON + 1
                        ; = 0xFF ^ 13 + ON + 1
                        ; = 0xF4 + ON

```

which is (surprisingly enough) the same result that was found with the “TMR0 delay value” equation earlier. The formula in itself is not that impressive—except that it dovetails very well with the PWM on half of the code. This optimization probably belongs in another chapter on optimization, but to be honest with you, I came up with it using nothing but trial and error along with the “feeling” that this kind of optimization were possible. This is an example of what I mean when I say that you should look for opportunities when processing data in the PIC microcontroller. More often than not, you will come up with something like these few instructions that are very efficient and integrate different cases.

Note that in this code, the PWM signal will never fully be on (a high dc voltage) or fully off (a ground level dc voltage). This is so because when the routine enters the sub-routine handler, it changes the output regardless of whether or not it is required for the length of the interrupt handler. In actuality, if you time it out, you will see that the 23 instruction cycles that the interrupt handler takes between changing the value works out to a 2.4 percent loss of full-on and full-off. This should not be significant in most

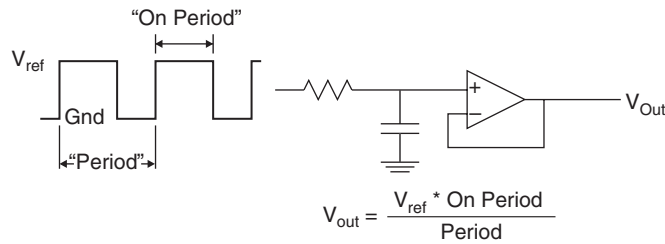


Figure 17.22 The PWM output can be converted to an analog voltage using an RC low-pass filter.

applications and will serve as a “heartbeat” to let the receiver know that the PIC microcontroller is still functioning, even though the output is “stuck” at an extreme.

In the preceding example I expended quite a bit of energy in making sure that the period remains the same regardless of the on time. This was done to make sure that the changes in the duty cycle remained proportional to the changes in the on period. This is important if the PWM output is going to be passed through a low-pass filter, as shown in Fig.17.22, to output an analog voltage.

In many applications where a PWM signal is communicating with another digital device, this effort to ensure that the period is constant is not required. In these cases, a timer is used to time the on period. This can be shown as the pseudocode

```

Int TimeOn() // Time the Width of an incoming Pulse
{
    int i = 0;

    while (PWMIP == off); // Wait for the Pulse to Start

    for ( ; PWMIP == ON; i++ ); // Time the Pulse Width

    return i; // Return the Pulse Width
} // end TimeOn

```

with the actual PIC microcontroller assembly-language code being quite simple but dependent on the maximum pulse width value being timed—very long pulses will require large counters or delays in between the PWM input (PWMIP in TimeOn above) poll.

Passing analog data back and forth between digital devices in any format is not going to be accurate owing to the errors in digitizing the value and restoring it. This is especially true for PWM signals, which can have very large errors owing to the sender and receiver not being properly synched up and the receiver not starting to poll at the correct time interval. In fact, the measured value could have an error of upward of 10 percent from the actual value. This loss of data accuracy means that analog signals should not

be used for data transfers. However, as I will show later in this book, PWM signals are an excellent way to control analog devices such as LEDs and motors.

When using a PWM signal for driving an analog device, it is important to make sure that the frequency is outside the range of what a human can perceive. As noted in the LED section earlier, this frequency is 30 Hz or more. For motors and other devices that may have an audible whine, however, the PWM signal used should have a frequency either below 50 Hz or 20 kHz or more to ensure that the signal does not bother the user (although it may cause problems with the user's dogs). The lower PWM frequency is probably surprising, but there are a lot of small motors that work very well with the low PWM frequency.

The problem with the higher frequencies is that the granularity of the PWM signal decreases. This is due to the inability of the PIC microcontroller (or whatever digital device is driving the PWM output) to change the output in relatively small time increments from on to off relative to the size of the PWM signal's period. In the preceding example code, four instruction cycles (of 1 μ s each) are the lowest level of granularity for the PWM signal that results in about 250 unique output values. If the PWM signal's period was decreased to 100 μ s from 1 ms for a 10-kHz frequency, the same code would have only 25 or so unique values that could be output. In this case, to retain the original code's granularity, the PIC microcontroller would have to be sped up 10 times (not possible for most applications) or another way of implementing the PWM would have to be found.

Audio Output

When I discuss the PIC microcontroller's processing capabilities with regard to audio, I tend to be quite disparaging. The reason for this is the lack of hardware multipliers in low-end and mid-range PIC microcontrollers and the inability of all the devices to natively handle greater than 8 bits in a floating-point format. The PIC microcontroller processor has been optimized for responding to digital inputs and cannot implement the real-time processing routines needed for complex analog I/O.

Despite this, you can implement some surprisingly sophisticated audio output that goes beyond simple beeps and boops using a circuit such as the one shown in Fig. 17.23.

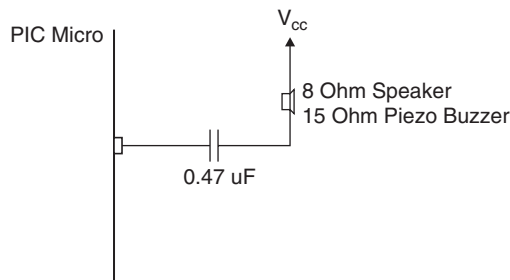


Figure 17.23 The PIC microcontroller can drive a simple piezo speaker.

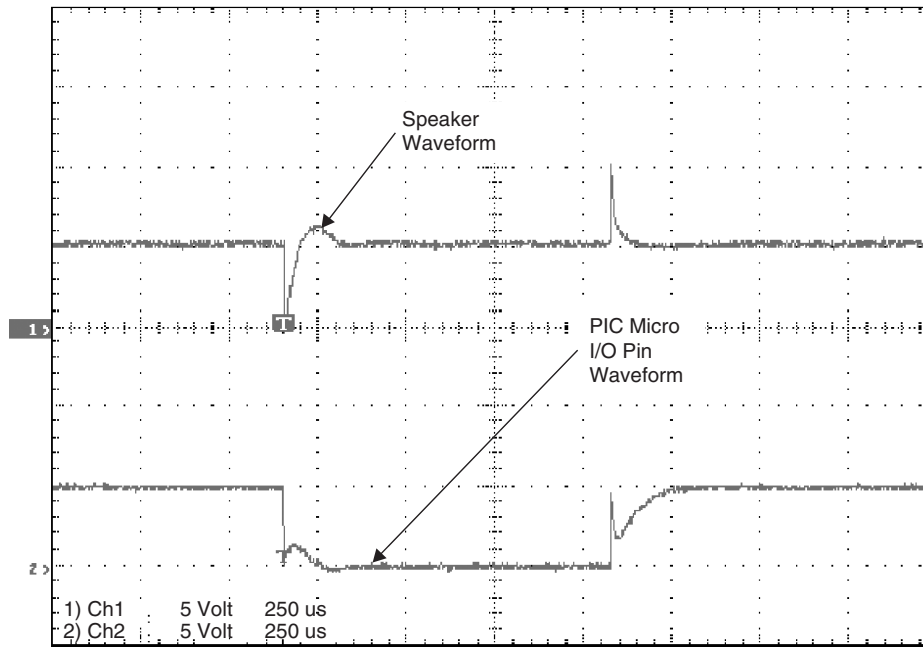


Figure 17.24 The voltage waveform across the speaker is probably not what you expect.

This passes dc waveforms through the capacitor (which filters out the kickback spikes) to the speaker or piezo buzzer. When a tone is output, your ear will hear a reasonably good tone, but if you were to look at the actual signal on an oscilloscope, you would see the waveform shown in Fig. 17.24 both from the PIC microcontroller's I/O pin and from the piezo buzzer itself.

The PIC microcontroller pin, capacitor, and speaker are actually quite a complex analog circuit. Note that the voltage output on the PIC microcontroller's I/O pin is changed from a straight waveform. This is due to the inductive effects of the piezo buffer. The important thing to note in this figure is that the upward spikes appear at the correct period for the output signal.

Timing the output signal generally is accomplished by toggling an output pin at a set period within the TMR0 interrupt handler or using the CCP module to produce a PWM tone. To generate a 1-kHz signal in a PIC microcontroller running at 4 MHz, you can use the following code (which does not use the prescaler) for TMR0 and the PIC microcontroller's interrupt capability:

```

org      4
int:
    movwf    _w          ; Save Context Registers
    bcf     INTCON, TOIF ; Reset the Interrupt
    movlw   256 - (250 - 4)
    movwf   TMR0        ; Reset TMR0 for another 500 µsecs

```

692 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

```

btfsc  SPKR      ; Toggle the Speaker
goto   $ + 2
bsf    SPKER     ; Speaker Output High
goto   $ + 2
bcf    SPKER     ; Speaker Output Low
swapf  _w, f     ; Restore Context Registers
swapf  _w, w
retfie

```

There are two points to notice about this interrupt handler: The first is that I don't bother saving the STATUS register's contents because neither the zero, carry, nor digit carry flags are changed by any of the instructions used in the handler. The second point to notice is the reload value of TMR0 to generate a 1-kHz output in a 4-MHz PIC microcontroller (an instruction clock period of 1 μ s); I have to delay 500 cycles for the wave's high and low. Because TMR0 has a divide by two counts on its input, I have to wait a total of 250 ticks. When I reload TMR0, note that I also take into account the cycles taken to get to the reload (which is 7 or 8), divide them by 2, and take them away from the reload value.

For this handler, the reload value may be off by one cycle depending on how the main-line executes, for a worst-case error of 0.2 percent, or 2,000 ppm. This level of accuracy is approximately the same as what you would get for a ceramic resonator; and the change in the frequency should not cause a noticeable warbling (changes in the frequency) of the output.

When developing applications that output audio signals, I try to keep the tone within the range of 500 Hz to 2 kHz. This is well within the range of human hearing and is quite easy to create the software for. When you look at the "Christmas tree" in Chap. 21, you can see how this is done for creating simple tunes on the PIC microcontroller.

Relays and Solenoids

Some real-life devices that you may have to control by a microcontroller are electromagnetic relays, solenoids, and motors. These devices cannot be driven directly by a microcontroller because of the current required and the noise they generate. This means that special interfaces must be used to control electromagnetic devices.

The simplest method of controlling these devices is to just switch them on and off and by supplying power to the coil in the device. The circuit shown in Fig. 17.25 is true for relays (as is shown), solenoids (which are coils that draw an iron bar into them when they are energized), and dc motors (which will only turn in one direction).

In this circuit, the microcontroller turns on the Darlington transistor pair, causing current to pass through the relay coils, closing the contacts. To open the relay, the output is turned off (or a 0 is output). The shunt diode across the coil is used as a kickback suppressor. When the current is turned off, the magnetic flux in the coil will induce a large back EMF (voltage) that has to be absorbed by the circuit or there may be a voltage spike that can damage the relay power supply and even the microcontroller. This diode *never* must be

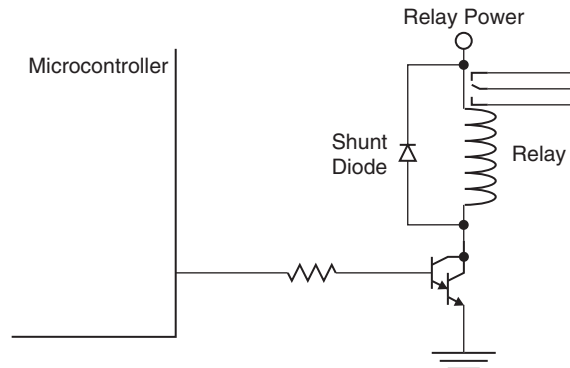


Figure 17.25 A relay can be controlled using a high-current transistor and a kickback-suppression diode.

forgotten in a circuit that controls an electromagnetic device. The kickback voltage is usually on the order of several hundred volts for a few nanoseconds. This voltage causes the diode to break down and allows current to flow, attenuating the induced voltage.

Rather than designing discrete circuits to carry out this function, I like to use integrated chips for the task. One of the most useful devices is the ULN2003A (Fig. 17.26) or the ULN2803 series of chips, which have Darlington transistor pairs and shunt diodes built in for multiple drivers.

Asynchronous (NRZ) Serial Interfaces

Asynchronous long-distance communications came about as a result of the Baudot teletype. This device mechanically (and, later, electronically) sent a string of electrical signals (which we would call a *packet of bits*, shown in Fig. 17.27) to a receiving printer.

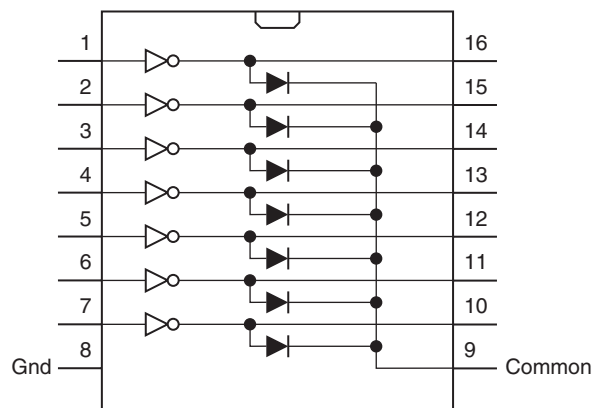


Figure 17.26 The ULN2003A (along with modern variants) is an efficient way to control multiple relays.

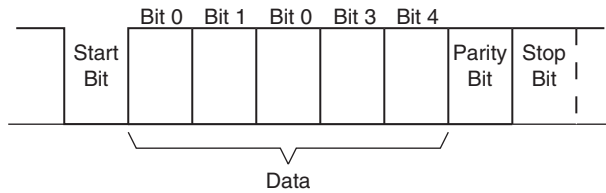


Figure 17.27 Every NRZ asynchronous packet consists of a start bit, a stop bit, and data bits.

With the invention of the teletype, data could be sent and retrieved automatically without having to have an operator sitting by the teletype all night unless an urgent message was expected. This data-packet format is still used today for the electrical asynchronous transmission protocols described below.

Before going on, there is one point that some people get unreasonably angry about, and that's the definition and use of the terms *data rate* and *baud rate*. The baud rate is the maximum number of possible data-bit transitions per second. This includes the start, parity, and stop bits at the ends of the data packet shown in the figure, as well as the 5 data bits in the middle. I use the term *packet* because we are including more than just data (there is also some additional information in there as well), so *character* and *byte* (if there were 8 bits of data) are not appropriate terms. This means that for every 5 data bits transmitted, 8 bits in total are transmitted (which means that nearly 40 percent of the data transmission bandwidth is lost in teletype asynchronous serial communications).

The data rate is the number of data bits that are transmitted per second. For this example, if you were transmitting at 110 baud (which is a common teletype data speed), the actual data rate would be 68.75 bps (or assuming 5 bits per character, 13.75 characters per second).

I tend to use the term *data rate* to describe the *baud rate*. This means that when I say *data rate*, I am specifying the number of bits of *all* types that can be transmitted in a given period of time (usually 1 second). I realize that this is not absolutely correct, but it makes sense to me to use it in this form, and I have used it consistently throughout this book (and I have not used the term *baud rate*).

With only 5 data bits, the Baudot code could transmit only up to 32 distinct characters. To handle a complete character set, a specific five-digit code was used to notify the receiving teletype that the next 5-bit character would be an extended character. With the alphabet and most common punctuation characters in the primary 32 characters, this second data packet wasn't required very often.

As discussed in Chap. 16, when waiting for a character, the PIC microcontroller USART receiver polls the line repeatedly at 1/16 bit period intervals until a 0 (space) is detected. The receiver then waits half a cycle before polling the line again to see if a glitch was detected and not a start bit. Once the start bit is validated, the receiver hardware polls the incoming data once every bit period multiple times (again, to ensure that glitches are not read as incorrect data).

The stop bit was provided originally to give both the receiver and the transmitter some time before the next packet is transferred (in early computers, the serial datastream was created and processed by the computers and not by custom hardware, as in modern

computers). It should be noted that the stop bit is always a 1, which is the reason for asynchronous communications being known as NRZ, or never return zero—at the end of the packet, the stop bit makes the line a 1, and the receiver knows that new data is coming in when a 0 start bit is detected.

The parity bit is a crude method of error detection that was first brought in with teletypes. The purpose of the parity bit is to indicate whether the data was received correctly. An odd parity bit meant that if all the mark bits in a packet after the start and before the stop bits were counted, the result would be an odd number. Even parity is checking all the data and parity bits and seeing if the number of mark bits is an even number. Along with even and odd parity, there are mark, space, and no parity. Mark parity means that the parity bit is always set to a 1, space parity is always having a 0 for the parity bit, and no parity is eliminating the parity bit altogether.

The most common form of asynchronous serial data packet is 8-N-1, which means 8 data bits, no parity, and 1 stop bit. This reflects the capabilities of modern computers to handle the maximum amount of data with the minimum amount of overhead and with a very high degree of confidence that the data will be correct.

I stated that parity bits are a crude form of error detection. I said this because they can only detect one bit error (i.e., if 2 bits are in error, the parity check will not detect the problem). If you are working in a high-induced-noise environment, you may want to consider using a data protocol that can detect (and, ideally, correct) multiple bit errors.

RS-232

In the early days of computing (the 1950s), while data could be transmitted at high speed, it couldn't be read and processed continuously. Thus a set of "handshaking" lines and protocols were developed for what became known as *RS-232 serial communications*.

With RS-232, the typical packet contained 7 bits (which is the number of bits in an ASCII character). This simplified the transmission of human-readable text but made sending object code and data (which were arranged as bytes) more complex because each byte would have to be split up into two nybbles (which are 4 bits long). Further complicating this is the fact that the first 32 characters of the ASCII character set are defined as special characters (i.e., carriage return, back space, etc.). This meant that the data nybbles would have to be converted (shifted up) into valid characters (this is why if you ever see binary data transmitted from a modem or embedded in an e-mail message, data is either sent as hex codes or as the letters A to Q). With this protocol, to send a single byte of data, 2 bytes (with the overhead bits resulting in 20 bits in total) would have to be sent (and surprisingly enough, to send data would take twice as long as sending a text file of the same length).

As I pointed out earlier, modern asynchronous serial data transmission is normally 8 bits at a time, which will avoid this problem and allow transmission of full bytes without breaking them up or converting them.

The actual RS-232 communications model is shown in Fig. 17.28, and in RS-232, the different devices are wired according to the functions they perform. DTE stands for *data terminal equipment* and is meant to be the connector used for computers (the PC uses this type of connection). DCE, or *data communications equipment*, was meant for modems that transfer data to other long-distance devices.

696 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

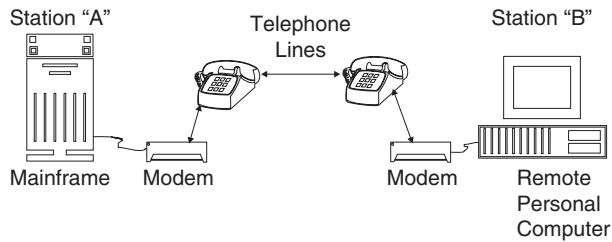


Figure 17.28 The PC communications model.

Understanding how different equipment fits in the RS-232 model is critical to connecting two devices successfully by RS-232. With a pretty good understanding of the serial data, we can now look at the actual voltage signals. As I mentioned earlier, when RS-232 was first developed into a standard, computers and the electronics that drove them were still very primitive and unreliable. Because of this, we've got a couple of legacies to deal with.

The first is the voltage levels of the data. A mark (1) is actually -12 V , and a space (0) is $+12\text{ V}$. From the figure, you should see that the hardware interface is not simply a TTL or CMOS level buffer. Later in this section I will introduce you to some methods of generating and detecting these interface voltages. Voltages in the switching region ($\pm 3\text{ V}$) may or may not be read as a 0 or 1 depending on the device. You always should make sure that the voltages going into a PIC microcontroller RS-232 circuit are in the valid regions.

Of more concern are the handshaking signals. These six additional lines (which are at the same logic levels as the transmit/receive lines and are shown in Fig. 17.29) are used to interface between devices and control the flow of information between computers. The "request to send" (RTS) and "clear to send" (CTS) lines are used to control data flow between the computer (DCE device) and the modem (DTE device). When the PC is ready to send data, it asserts (outputs a mark) on RTS. If the DTE device is capable of receiving data, it will assert the CTS line. If the PC is unable to receive data (i.e., the buffer is full or it is processing what it already has), it will deassert the RTS line to notify the DTE device that it cannot receive any additional information.

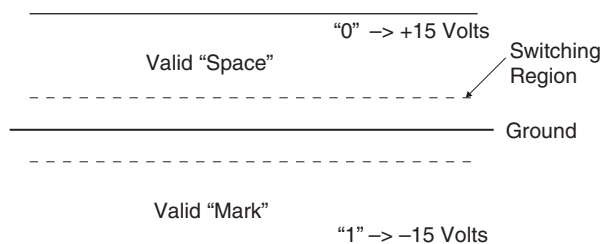


Figure 17.29 RS-232 data is specified at unusually high levels to and from which your circuit will have to translate.

The “data transmitter ready” (DTR) and “data set ready” (DSR) lines are used to establish communications. When the PC is ready to communicate with the DTE device, it asserts DTR. If the DTE device is available and ready to accept data, it will assert DSR to notify the computer that the link is up and ready for data transmission. If there is a hardware error in the link, then the DTE device will deassert the DSR line to notify the computer of the problem. Modems will deassert the DSR line if the carrier between the receivers is lost.

There are two more handshaking lines that are available in the RS-232 standard that you should be aware of, even though chances are that you will never connect anything to them. The first is the “data carrier detect” (DCD), which is asserted when the modem has connected with another device (i.e., the other device has picked up the phone). The “ring indicator” (RI) is used to indicate to a PC whether or not the phone on the other end of the line is ringing or is busy. These lines are used very rarely in PIC microcontroller applications because the AT command set provides a text message for these functions.

There is a common ground connection between the DCE and DTE devices. This connection is critical for the RS-232 level converters to determine the actual incoming voltages. The ground pin never should be connected to a chassis or shield ground (to avoid large current flows or be shifted and prevent accurate reading of incoming voltage signals). Incorrect grounding of an application can result in the computer or device with which it is interfacing resetting or having the power supplies blow a fuse or burn out. The latter consequences are unlikely, but I have seen it happen. To avoid these problems, make sure that chassis and signal grounds are separate or connected by a high-value (hundreds of kilohm) resistor.

Before going too much farther, I should expose you to an ugly truth: The handshaking lines are almost never used in RS-232 (and not just PIC microcontroller RS-232) communications. Normally, three-wire RS-232 connections are implemented as in Fig. 17.30. I normally accomplish this by shorting the DTR/DSR and RTS/CTS lines

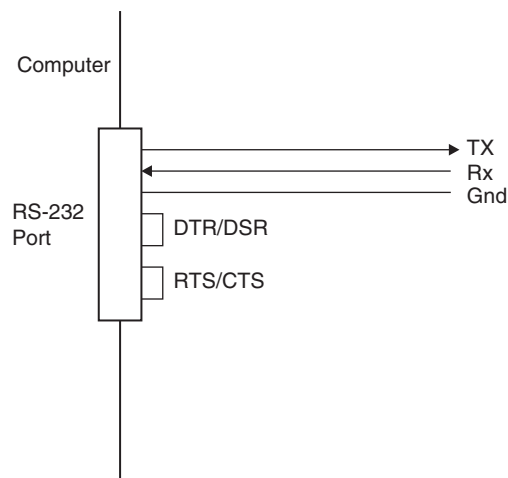


Figure 17.30 In most applications, only three wires are actually used to transmit data between devices.

together at the PIC microcontroller end. The DCD and RI lines are left unconnected. With the handshaking lines shorted together, data can be sent and received without having to develop software to handle the different handshaking protocols.

A couple of points on the three-wire RS-232: The first is that it cannot be implemented blindly; in about 20 percent of RS-232 applications that I have had to do over the years, I have had to implement some subset of the total seven-wire (transmit, receive, ground, and four handshaking lines) protocol lines. Interestingly enough, I have never had to implement the full hardware protocol. This still means that four out of five times if you wire the connection as shown in Fig. 17.30, the application will work.

With the three-wire RS-232 protocol, there may be applications where you don't want to implement the hardware handshaking (the DTR, DSR, RTS, and CTS lines); you may want to implement software handshaking. There are two primary standards in place. The first is known as the *XON/XOFF protocol*, in which the receiver sends an XOFF (DC3 or character 0x013) when it can't accept any more data. When it is able to receive data, it sends an XON (DC1 or character 0x011) to notify the transmitter that it can receive more data.

The final aspect of the RS-232 I want to discuss is the speeds at which data is transferred. When you first see the speeds (such as 300, 2,400 and 9,600 bps), they seem rather arbitrary. The original serial data speeds were chosen for teletypes because they gave the mechanical device enough time to print the current character and reset before the next one came in. Over time, these speeds have become standards, and as faster devices have become available, the speed have been doubled continually (i.e., 9,600 bps is 300 bps doubled five times).

To produce these data rates, the PIC microcontroller's USART uses a clock divider to produce a clock 16 times the data rate. The PIC microcontroller's operating clock is divided by integers to get the nominal RS-232 speeds. This might seem like it won't work out well, but because of RS-232's strange relationship with the number 13, the situation isn't as bad as it may seem.

If you invert (to get the period of a bit) the data speeds and convert the units to microseconds, you will discover that the periods are almost exactly divisible by 13. This means that you can use an even megahertz oscillator in the hardware to communicate over RS-232 at standard data rates. For example, if you had a PIC microcontroller running with a 20-MHz instruction clock and you wanted to communicate with a PC at 9,600 bps, you would determine the number of cycles to delay by

- 1** Finding the bit period in microseconds. For 9,600 bps, this is 104 μ s.
- 2** Dividing this bit period by 13 to get a multiple number. For 104 μ s, this is 8.

Now, if the external device is running at 20 MHz (which means a 200-ns cycle time), you can figure out the number of cycles as multiples of 8×13 in the number of cycles in 1 μ s. For 20 MHz, 5 cycles execute per microsecond. To get the total number of cycles for the 104- μ s bit period, you simply evaluate

$$20 \text{ cycles}/\mu\text{s} \times 13 \times 5 \mu\text{s/bit} = 1,300 \text{ cycles/bit}$$

700 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

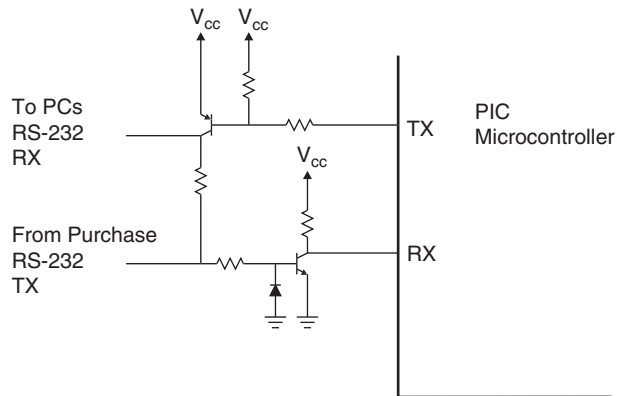


Figure 17.32 Voltage-stealing RS-232 interface that inverts the RS-232 signals to and from the PIC microcontroller.

handshaking interface). Ground for the incoming signal is connected to the “processor” ground (which is not the case’s ground).

Along with the MAX232, Maxim and some other chip vendors have a number of other RS-232 charge-pump-equipped devices that will allow you to handle more RS-232 lines (to include the handshaking lines). Some charge-pump devices that are also available do not require the external capacitors that the MAX232 chip does, which will simplify the layout of your circuit (although these chips do cost quite a bit more).

The next method of translating RS-232 and TTL/CMOS voltage levels is to use the transmitter’s negative voltage. The circuit in Fig. 17.32 shows how this can be done and will be demonstrated later in this book. This circuit relies on the RS-232 communications only running in half-duplex mode (i.e., only one device can transmit at a given time). When the external device wants to transmit to the PC, it sends the data either as a mark (leaving the voltage returned to the PC as a negative value) or as a space by turning on the transistor and enabling the positive-voltage output to the PC’s receivers. If you go back to the RS-232 voltage specification drawing, you’ll see that +5 V is within the valid voltage range for RS-232 spaces. This method works very well (consuming just about no power) and is obviously a very cheap way to implement a three-wire RS-232 bidirectional interface.

As an aside, before going on to the last interface circuit, I should point out a big advantage of the preceding circuit. The advantage results from the fact that the PIC microcontroller’s RS-232 transmitter circuit receives its negative voltage from the PC’s RS-232 transceiver *if* the external device (with this circuit) is connected to a PC. This means that the PC’s TX and RX are connected together, and it can “ping” (send a command that is ignored by the external device) via the RS-232 port to see if an external device is connected to it.

To do this, you need to specify the ping character as something that the external device can recognize and modify so that the PC’s software can recognize that the interface is working. The method that I have employed in the past is to use a microcontroller with “bit banging” software to change some mark bits when it recognizes that a ping character is being received.

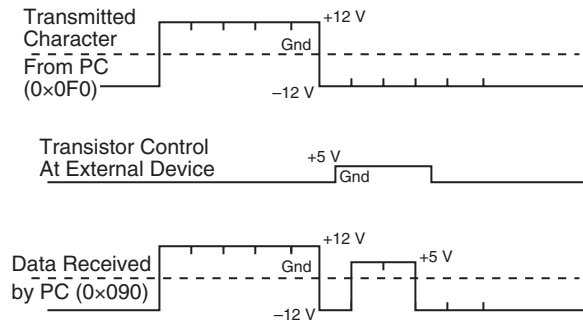


Figure 17.33 Sending 0xF0 and letting the receiver modify it is one way of detecting whether or not there is a receiver and if it is active.

In Fig. 17.33 I show a ping character of 0xF0 that is modified by the external device (by turning on the transistor) to change some bits into spaces. If the PC receives nothing at all, then there is nothing attached to it, and if it receives 0xF0, then the external device is not active.

With the availability of many CMOS devices requiring very minimal amounts of current to operate, you might be wondering about different options for powering your circuit. One of the most innovative that I have come across is using the PC's RS-232 ports themselves as powering devices that are attached to it using the circuit shown in Fig. 17.34.

When the DTR and RTS lines are outputting a space, a positive voltage (relative to ground) is available. This voltage can be regulated and the output used to power the devices attached to the serial port (up to about 5 mA). For extra current, the TX line also can be added into the circuit as well, with a break being sent from the PC to output a positive voltage.

The 5 mA is enough current to power the transistor/resistor type of RS-232 transmitter and a PIC microcontroller running at 4 MHz, along with some additional hardware (such as an LCD). You will not be able to drive an LED with this circuit, and you may find that some circuits that you normally use for such things as pull-ups and pull-downs will consume too much power, and you'll have to specify different resistance values.

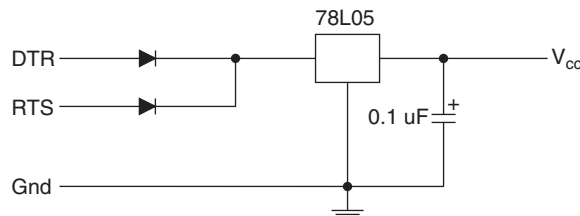


Figure 17.34 The unused handshaking lines can be used to provide power for your PIC microcontroller application.

702 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

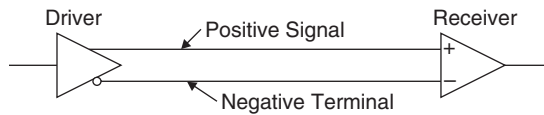


Figure 17.35 For very long distances, you can transmit data using differential-pair wiring.

Now, with this method of powering the external device, you do not have use of the handshaking lines, but the savings of not having to provide an external power supply (or battery) will outweigh the disadvantages of having to come up with a software ping-pong and handshaking protocol. Externally powering a device attached to the RS-232 port is ideal for input devices such as serial mice that do not require a lot of power.

RS-485/RS-422

So far in this book I have discussed single-ended asynchronous serial communications methods such as RS-232 and direct NRZ device interfaces. These interfaces work well in home and office environments but can be unreliable in environments where power surges and electrical noise can be significant. In these environments, a double-ended or differential-pair connection is optimal to ensure the most accurate communications.

A *differential-pair* serial communications electrical standard consists of a balanced driver with positive and negative outputs that are fed into a comparator that outputs a 1 or a 0 depending on whether or not the positive line is at a higher voltage than the negative line. Figure 17.35 shows the normal symbols used to describe a differential-pair connection.

There are several advantages to this data-connection method. The most obvious one is that the differential pair doubles the voltage swing sent to the receiver, which increases its noise immunity. This is shown in Fig. 17.36; when the positive signal goes high, the negative voltage goes low. The change in the two receiver inputs is 10 V rather than the 5 V of a single line. This is assuming that the voltage swing is 5 V for the positive and negative terminals of the receiver. This effective doubling of the signal voltage reduces the impact the electrical interface has on the transmitted signal.

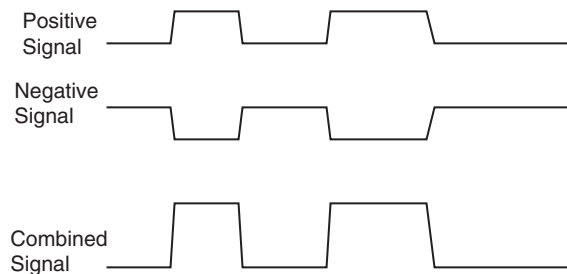


Figure 17.36 Differential data transmission consists of two opposite lower-voltage signals creating a high-quality signal.

Another benefit of differential-pair wiring is that if one connection breaks, the circuit will operate (although at reduced noise-reduction efficiency). This feature makes differential pairs very attractive in cars, aircraft, and spacecraft, where loss of a connection could be catastrophic.

To minimize ac transmission-line effects, the two wires should be twisted around each other. Twisted-pair wiring can either be bought commercially or made simply by twisting two wires together; twisted wires have a characteristic impedance of $75\ \Omega$ or greater.

A common standard for differential-pair communications is RS-422. This standard, which uses many commercially available chips, provides

- 1 Multiple-receiver operation
- 2 Maximum data rate of 10 Mbps
- 3 Maximum cable length of 4000 m (with a 100-kHz signal)

Multiple receiver operation, as shown in Fig. 17.37A, allows signals to be broadcast to multiple devices. The best distance and speed changes with the number of receivers of the differential pair, along with its length. The 4000 m at 100 kHz or 40 m at 10 MHz

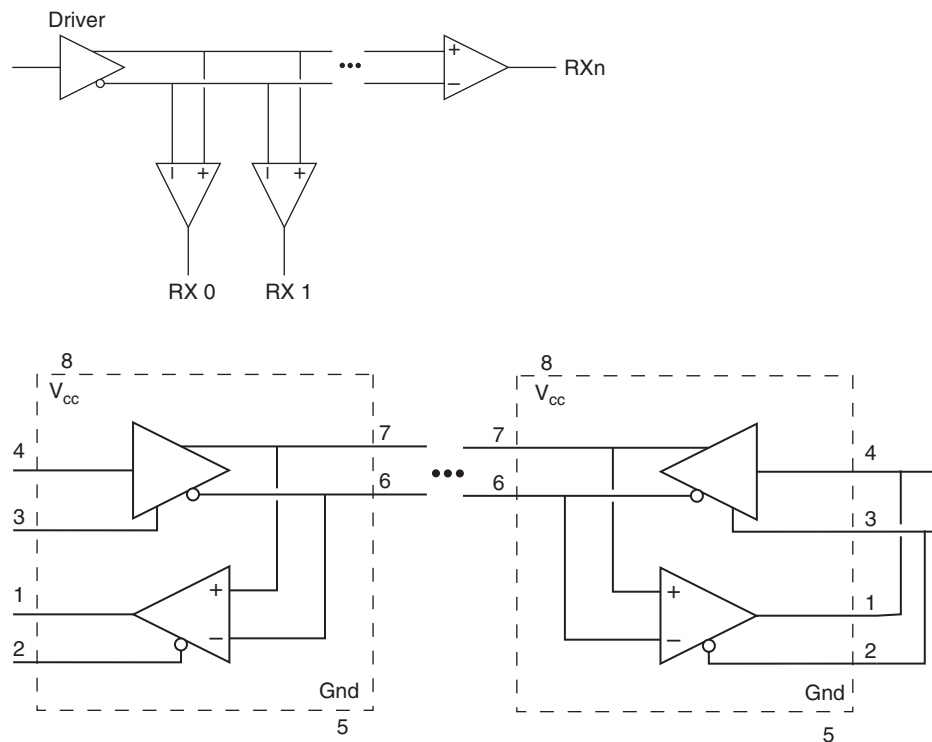


Figure 17.37 A. The RS-422 protocol allows multiple receivers on the same differential-pair wiring. B. The 75176 driver chip is used to create a single RS-485 differential-pair communications medium.

are examples of this balancing between line length and data rate. For long data lengths, a few-hundred-ohm terminating resistor may be required between the positive terminal and the negative terminal at the end of the lines to minimize reflections coming from the receiver and affecting other receivers.

RS-422 is not as widely used as you might expect; instead, RS-485 is much more popular. RS-485 is very similar to RS-422, except that it allows multiple drivers on the same network. The common chip is the 75176, which has the ability to drive and receive on the lines, as shown in Fig. 17.37.

In the right 75176 of Fig. 17.37, I show the RX and TX and two enables tied together. This results in a two-wire differential I/O device. Normally, the 75176s are left in RX mode (pin 2 reset) unless they are driving a signal onto the bus. When the unused 75176s on the lines are all in receive mode, anyone can take over the lines and transmit data.

As with the RS-422, multiple 75176s (up to 32) can be on the RS-485 lines with the capability of driving or receiving. When all the devices are receiving, a high (1) is output from the 75176. This means that the behavior of the 75176 in the RS-485 (because these are multiple drivers) is similar to that of a dotted AND bus; when one driver pulls down the line, all receivers are pulled low. For the RS-485 network to be high, all unused drivers must be off, or all active drivers must be transmitting a 1. This feature of the RS-485 is taken advantage in small system networks such as CAN.

The only issue to be on the lookout for when creating RS-485/RS-422 connections is to keep the cable polarities correct (positive to positive and negative to negative). Reversing the connectors will result in lost signals and misread transmission values.

Synchronous Serial Interfaces

For synchronous data communications in a microcontroller, a clock signal is sent along with serial data, as shown in Fig. 17.38. The clock signal strobes the data into the receiver, and the transfer can take place on the rising or falling edge of the clock. This method of transmission is different from the asynchronous protocol by the provision of the clock line from the transmitter; in asynchronous communications, the receiver is expected to provide a clock for timing the incoming data. The clock used in asynchronous transmission must be very precise and closely matched to the receiver, whereas in synchronous transmission there is only one clock, and the receiver does not need to be precisely tuned for accepting the transmitter's data.



Figure 17.38 Synchronous communications consist of data being latched into the receiver using a clock from the transmitter.

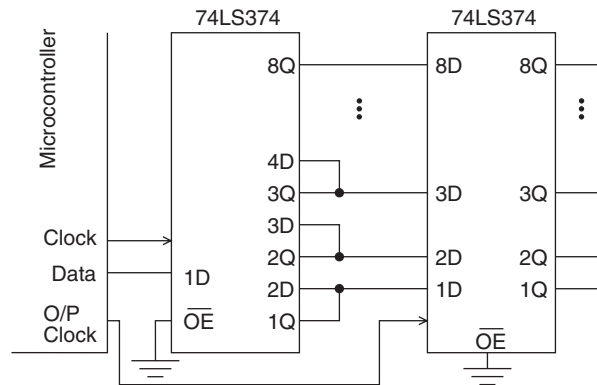


Figure 17.39 Converting synchronous serial data to parallel bits is quite easy to do.

A typical circuit using discrete devices could be that in Fig. 17.39. This circuit converts serial data into eight digital outputs that all are available at the same time (when the O/P clock is strobed). For most applications, the second 374 providing the parallel data is not required. This serial-to-parallel conversion also can be accomplished using serial-to-parallel chips, but I prefer using 8-bit registers because they are generally easier to find than other TTL parts.

There are two very common synchronous data protocols—Microwire and SPI. These methods of interfacing are used in a number of chips (such as the serial EEPROMs used in the BASIC Stamps). While the Microwire and SPI standards are quite similar, there are a number of differences that should be noted.

I consider these protocols to be methods of transferring synchronous serial data rather than microcontroller network protocols because each device is addressed individually (even though the clock/data lines can be common between multiple devices). If the chip select for the device is not asserted, the device ignores the clock and data lines. With these protocols, only a single master can be on the bus.

If a synchronous serial port is built into the microcontroller, the data transmit circuitry might look like that in Fig. 17.40. This circuit will shift out 8 bits of data. For protocols such as Microwire, where a start bit is sent initially, the start bit is sent using direct reads and writes to the I/O pins. To receive data, a similar circuit would be used, but data would be shifted into the shift register and then read by the microcontroller.

The Microwire protocol is capable of transferring data at up to one megabit per second. Sixteen bits are only transferred at a time. After selecting a chip and sending a start bit, the clock strobes out an 8-bit command byte (labeled OP1, OP2, A5 to A0 in the diagram above), followed by (optionally) a 16-bit address word transmitted and then another 16-bit word either written or read by the microcontroller.

The SPI protocol is similar to Microwire, but with a few differences:

- 1** SPI is capable of up to 3 Mbps data transfer rate.
- 2** The SPI data word size is 8 bits.

706 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

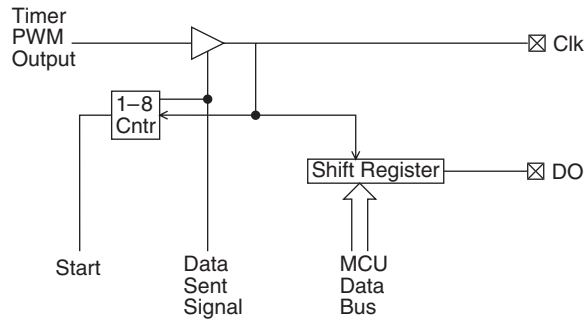


Figure 17.40 The synchronous serial hardware built into the PIC microcontroller allows data to flow through the input and output pins.

- 3 SPI has a “hold” that allows the transmitter to suspend data transfer.
- 4 Data in SPI can be transferred as multiple bytes known as “blocks” or “pages.”

Like Microwire, SPI first sends a byte instruction to the receiving device. After the byte is sent, a 16-bit address is optionally sent, followed by 8 bits of I/O. As noted earlier, SPI does allow for multiple-byte transfers. An SPI data transfer is shown in Fig. 17.41.

The SPI clock is symmetric (an equal low and high time). Output data should be available at least 30 ns before the clock line goes high and read 30 ns before the falling edge of the clock.

When wiring up a Microwire or SPI device, there is one trick that you can do to simplify the microcontroller connection, and that is to combine the DI and DO lines into one pin. Figure 17.42 is identical to what was shown earlier in this chapter when interfacing the PIC microcontroller into a circuit where there is another driver. In this method of connecting the two devices, when the data pin on the microcontroller has completed sending the serial data, the output driver can be turned off, and the microcontroller can read the data coming from the device. The current-limiting resistor between the data pin

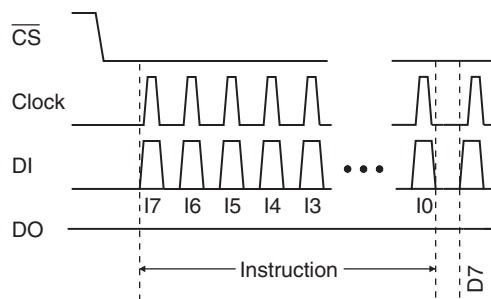


Figure 17.41 The SPI protocol is a very typical synchronous serial protocol with a device chip select.

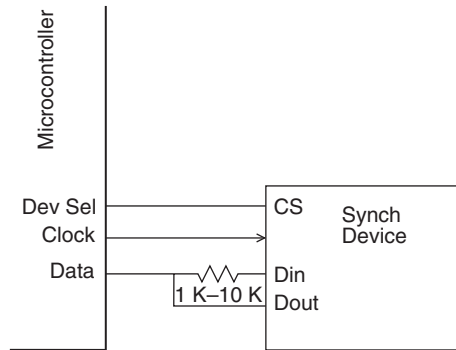


Figure 17.42 The input and output bits can be combined to simplify the wiring between devices.

and DI/DO limits any current flows when both the microcontroller and the device are driving the line.

I2C

The most popular form of microcontroller network is I2C (also known as I²C or “eye-squared-see”), which stands for “inter-intercomputer communications.” This standard was developed originally by Philips in the late seventies as a method to provide an interface between microprocessors and peripheral devices without wiring full address, data, and control busses between devices. I2C also allows sharing of network resources between processors (which is known as *multimastering*).

The I2C bus consists of two lines, a clock line (SCL) that is used to strobe data (from the SDA line) from or to the master that currently has control over the bus. Both these bus lines are pulled up (to allow multiple devices to drive them).

An I2C-controlled home entertainment system might be wired as in Fig. 17.43. The two bus lines are used to indicate that a data transmission is about to begin, as well as pass the data on the bus.

To begin a data transfer, a master puts a “start condition” on the bus. Normally, when the bus is in the idle state, both the clock and the data lines are not being driven (and are

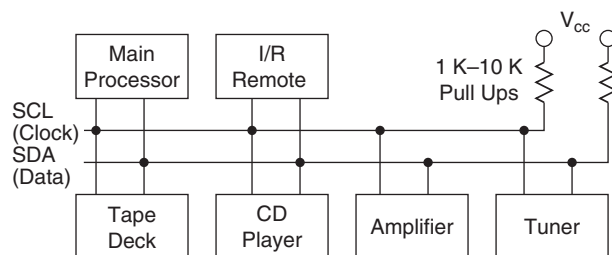


Figure 17.43 The I2C protocol is designed to support multiple devices on a single network wiring.

708 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

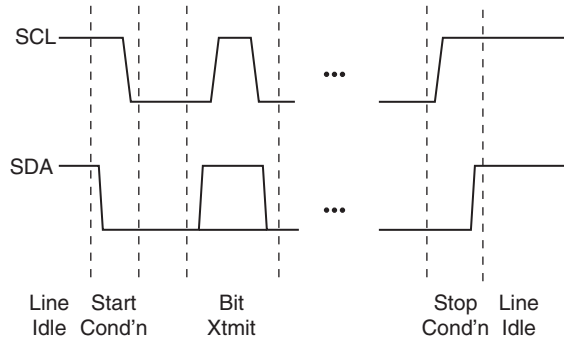


Figure 17.44 The I2C waveforms are similar to other synchronous serial protocols.

pulled high). To initiate a data transfer, the master requesting the bus pulls down the SDA bus line, followed by the SCL bus line. During data transmission, this is an invalid condition because the data line is changing while the clock line is active/high.

Each bit is then transmitted to or from the slave (the device the message is being communicated with by the master), with the negative clock edge being used to latch in the data, as shown in Fig. 17.44. To end data transmission, the reverse is executed; the clock line is allowed to go high, which is followed by the data line.

Data is transmitted in a synchronous (clocked) fashion. The most significant bit is sent first, and after 8 bits are sent, the master allows the data line to float (it doesn't drive it low) while strobing the clock to allow the receiving device to pull the data line low as an acknowledgment that the data was received. After the acknowledge bit, both the clock and the data lines are pulled low in preparation for the next byte to be transmitted or a stop/start condition is put on the bus. Figure 17.45 shows the data waveform.

Sometimes the acknowledge bit will be allowed to float high, even though the data transfer has completed successfully. This is done to indicate that the data transfer has completed and the receiver (which is usually a slave device or a Master that is unable to initiate data transfer) can prepare for the next data request.

There are two maximum speeds for I2C (because the clock is produced by a master, there really is no minimum speed)—standard mode runs at up to 100 kbps, and fast mode

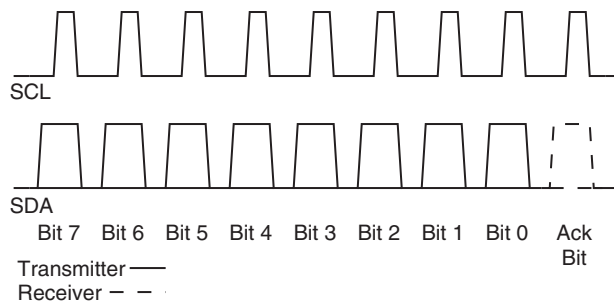


Figure 17.45 An I2C data packet.

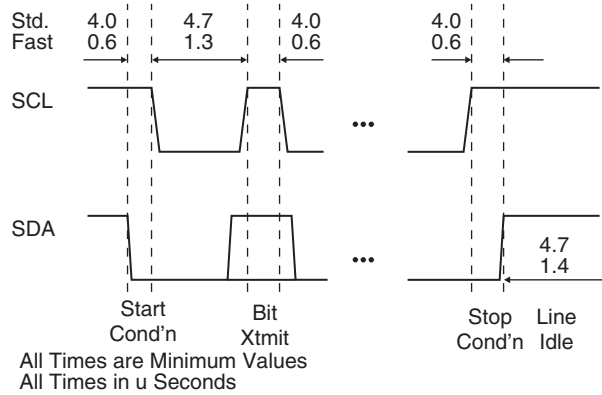


Figure 17.46 Bit timing for I2C data.

can transfer data at up to 400 kbps. Figure 17.46 shows the timing specifications for both the standard (“std.” or 100-kHz data rate) and fast (400-kHz data rate).

A command is sent from the master to the receiver in the format shown in Fig. 17.47. The receiver address is 7 bits long and is the bus address of the receiver. There is a loose standard to use the most significant 4 bits to identify the type of device, whereas the next 3 bits are used to specify one of eight devices of this type (or further specify the device type).

As I just said, this is *loose standard*. Some devices require certain patterns for the second 3 bits, whereas others (such as some large serial EEPROMS) use these bits to specify an address inside the device. In addition, there is a 10-bit address standard in which the first 4 bits are all set, the next bit reset, and the last 2 bits are the most significant 2 bits of the address, with the final 8 bits being sent in a following byte. All this means is that it is very important to map out the devices to be put on the bus and all their addresses.

This is really all there is to I2C communication, except for a few points. In some devices, a start bit has to be resent to reset the receiving device for the next command (i.e., in a serial EEPROM read, the first command sends the address to read from and the second reads the data at that address).

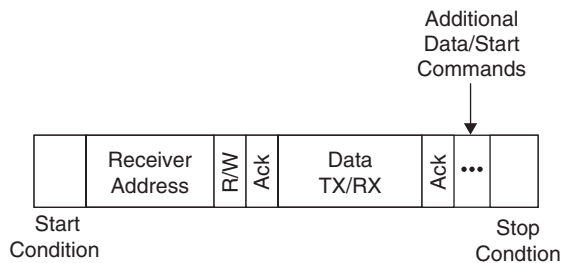


Figure 17.47 Multiple packets make up an I2C message.

710 PIC MCU INPUT AND OUTPUT DEVICE INTERFACING

The last point to note about I2C is that it's a multimastering device, which is to say that multiple microcontrollers can initiate data transfers on the bus. This obviously results in possible collisions on the bus (which is when two devices attempt to drive the bus at the same time). Obviously, if one microcontroller takes the bus (sends a "start condition") before another one attempts to do so, there is no problem. The problem arises when multiple devices initiate the "start condition" at the same time.

Actually, arbitration in this case is really quite simple. During the data transmission, hardware (or software) in both transmitters synchronize their clock pulses so that they match each other exactly. During the address transmission, if a bit is expected to be a 1 by a master is actually a 0, then it drops off the bus because another master is on the bus. The master that drops off will wait until the "stop condition" and then reinitiate the message. I realize that this is hard to understand with just a written description.

A "bit banging" I2C interface can be implemented in software of the PIC microcontroller quite easily. However, owing to software overhead, the fast mode probably cannot be implemented—even the standard mode's 100 kbps will be a stretch for most devices. I find implementing I2C in software to be best when the PIC microcontroller is the single master in a network. In this way, it doesn't have to be synchronized to any other devices or accept messages from any other devices that are masters and are running a hardware implementation of I2C that may be too fast for the software slave.