By Steve Hageman

PIC Development on a Shoestring

Are you itching to develop PIC-based projects yourself? Here are some ideas on how to go about it.

Ready for another programming assignment! My shoestring development station includes a Microchip programmer, an EPROM eraser and, of course, a PC.

Embedded microprocessor development can seem to be an expensive task, requiring lots of specialized equipment that is beyond the means of most amateurs. Although developing projects using PIC[1] microprocessors can be expensive, it *doesn't have to be*. PIC development can be done with just a handful of parts: an EPROM eraser, some kind of programming language and a low-cost PIC programmer.[2] That's an exact list of the equipment that I use. I don't have access to specialized professional equipment, but I do develop many useful projects with PICs.[3,4]

Why Pick PICs?

PICs are an *enabling* technology. All those logic, control, communication and display functions that our projects need can be packed into a single chip—a chip that can be reconfigured at your pleasure! Similarly, we can add features to our projects that make them even more professional and easier to operate.

The simplest, popular PICs are contained in a single 18-pin package, have 13 I/O pins and need only an external clock for operation. For usefulness, the PIC is truly the '90s version of the venerable 555 timer!

Most of the PICs you likely will be using are flash EEPROM or windowed EPROMs. This means that they can be programmed over and over again. In fact, the legs on my frequently used PICs break off long before the write/erase cycle limit of the PIC is ever reached! In higher-volume commercial applications, PICs are available at lower cost in a one-time programmable (OTP) version. The OTP version allows the code to be burned into the PIC *once*. Obviously, this "cheaper" device wouldn't be cheaper to use when you're developing new code, but when the code is stable, a commercial manufacturer may choose to use OTP versions to reduce cost.

[1]Notes appear on page 51.

Basic PIC Uses

I break PIC applications up into two overlapping categories:

- Stand-alone applications such as CW keyers, ID modules, etc. These PIC applications use the device as a *programmable logic array* (PLA). That is, the program you write for the PIC can take the place of hundreds of discrete logic gates. Such projects usually stand by themselves and have simple user interfaces that may include an LCD (see the sidebar "Simple Project Displays") and some user-interaction switches. My 2-meter PC-controllable FM receiver (see Note 4) is an example of this type of application.
- The other application category involves using the PIC as a PC interface. Because the PIC can communicate easily using RS-232 connections, you can use a PIC as a sophisticated "programmable UART." This follows the electronics industry trend toward *virtual instruments* (VI), that is, electronic equipment that does not have a *physical* front panel, but a communications interface to a computer that processes the instrument's data and displays the instrument's *virtual* front-panel controls. My Personal Network Analyzer (see Note 3) is one example of such a project.[5]

The two PIC-usage categories may overlap, as they do in my 2-meter receiver. That project uses a PIC to operate a user interface, including an LCD, knobs and switches. When it detects the presence of an RS-232 connection, it operates like a virtual radio with a PC controlling the receiver.

PC Control of a Virtual Instrument

The key to operating a virtual instrument is the program running on the PC. MS-DOS is a simple environment in which to operate, but the availability of DOS-based development tools is nonexistent now, except for shareware. This leaves *Windows*—and the tools here are quite good. Microsoft's *Visual Basic* is a very reliable and useful development tool. You may be able to find someone's old copy of *Visual Basic 3* for *Windows 3.1* or *95* development. Or, if you want to start with the latest 32-bit versions, use *Visual Basic 5* or *6* for *Windows 95/98* programming.

Because each PC-based language uses its own way of controlling the serial port, it's impractical to list them all here. But, the serial port is a commonly supported communication method under DOS and *Windows*, so it's a part of all these languages. Check the language manuals and help files for more specific information.

For an example of the PIC code that controls a virtual instrument via RS-232, visit my Web page describing the Personal Network Analyzer project and download the PIC source code.

Which PICs to Use?

At first, you seem to be faced with a bewildering array of PICs from which to choose. Your choice of which PICs to use may well start with the language you'll be using and/or what your programmer can support. I use four devices for all of my fun projects: the mid-range 16C6*x*, 16F84 and 16C7*x*; this keeps costs down and ensures I will always have devices available when I need them. Here are the reasons I chose these devices:

16F84—A low cost 18-pin device with moderate memory size (1 kB). This device is used for basic control applications that won't be large, need analog input (ie, an ADC) or use a lot of I/O pins (My Web site[6] shows several robots that my kids and I have

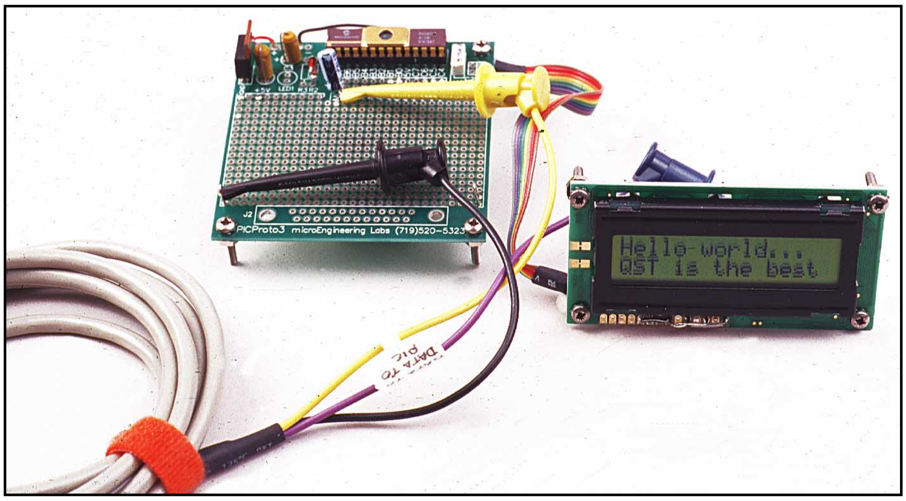

Figure 1—My universal RS-232 debugging cable simply clips to the circuit I'm testing. Three leads are all that is required to get bidirectional communication from a PIC breadboard to any PC terminal program.

built with this device). This is the most popular hobbyist PIC of all time (an older device was called the 16C84). This device uses flash memory and does not need to be erased in an EPROM eraser before it can be reprogrammed.

16C71—Like the 16F84, but it also has a built-in four-channel, 8-bit ADC. This device is used where I need to get analog signals into the PIC for control applications (such as the Personal Network Analyzer; see Note 3). The newest replacement to this device is the 16C711.

16C63—This device is in a 28-pin package, contains one five-bit I/O port and two 8-bit I/O ports. The device also has a hardware UART and 4 kB of memory. I use this device where I need the execution speed of the built in UART, the extra ports or the 4 kB of memory for large, complicated projects.

16C73—Like the 16C63, but also has a built-in, four-channel, ADC like the 16C71. I use this device in all the applications a 16C63 would be used for, but also need an ADC.

With just these four devices, I can keep my investment low and put together nearly any project I likely have the time for! To program these PICs, I use a Microchip PicStart programmer.[7] It's a bit on the expensive side, but it can program *all* of the currently available PICs.

How to Program a PIC

Programming a PIC is no harder than writing a small program for a PC; the exact sequence of steps is different, but it's not difficult to learn. If you want to program in assembly language, the tools are free from Microchip.[8] Programming in higher-level languages such as *BASIC* and *C* can be accomplished using low-cost (under $100) tools.[9,10] These higher-level language tools are usually referred to as *compilers* because they take the high-level statements and compile them into processor-specific machine language. I program my projects in *C*, a high-level language that keeps the code close to the hardware. This is the perfect language to use to talk to a PIC because controlling hardware is the whole idea. However, *BASIC* is also a viable language to use for amateur projects.

Once a program is written (in whatever language you have chosen), the assembler or compiler generates a binary image of the target PIC's memory. This image file is usually called a *hex* file, because it is in a format called the Intel hex 8 format. You don't need to know the technical details of this format for successful application of the PIC. Microchip's tools produce this format, so most of the low-cost programmers available support it as do the compiler manufacturers, and it's become the de-facto standard for PICs. All you really need to know is how to load the image file into your programmer and download it to your PIC, usually a simple task.

Debugging an Application

Debugging is where the fun starts—and stops, sometimes! The need for debugging can arise for two basic reasons: Because the program doesn't appear to work as you planned, or because you think up features to add as you're writing the programs. Professionals may debug with expensive *in-circuit emulators* (ICE systems) that allow stepping through the code line by line while connected to the target hardware. Although this is the most effective way to see how the program actually works from a time standpoint, its cost is usually beyond the means of most amateurs.

Debugging Tip 1

I use a slightly less time-effective debugging method than ICE, but it's a lot friendlier to your pocketbook! Most applications are best built in stages. Each stage adds a function, and I fully test that function before adding the next one. This way, the potential problem areas are kept small. If the program stops working after a new stage is added, then I know to look at the new stage first! Sometimes problems can be found by observing the action of the hardware and looking at the source code again. More often than not, you'll spot an obvious error and be able to correct it.

In fact, the programs for my 2-meter receiver project are built exactly this way. In past projects, I have used LCDs, so I reused this already-tested code to display program output when testing the other hardware code (described later). I had also used the PIC's built-in ADC and RS-232 UART, so these controlling subroutines were reused. I had not used an interrupt-driven rotary encoder before with a PIC, so I wrote some simple programs to experiment with how to best do this. I even used a 16C84 processor for this development because I had a breadboard already built up and wanted to reuse that also. When I had the encoder working correctly, I added the switch inputs and worked on debouncing routines and getting the RC networks connected to these switches properly.

When all the lower-level hardware-control routines were built and tested, I started building the final application safe in the knowledge that when my program said "read a debounced switch," if the program didn't work, I knew it was the *main program's* logic and not the previously written and tested subroutines. This is, in fact, my first approach to developing PIC applications: Work on the hardware-control stuff in small increments. It saves time in the long run.

Debugging Tip 2

My second approach to debugging on a shoestring is simply to keep five or more PICs in an EPROM eraser at all times. Doing so allows me to execute almost instantaneous write, compile, program, test cycles. Many times during development, I may work with a piece of code for only a few minutes before I decide it needs improvement. Sometimes, it just doesn't run at all! Being able to immediately get another blank PIC into the pro-

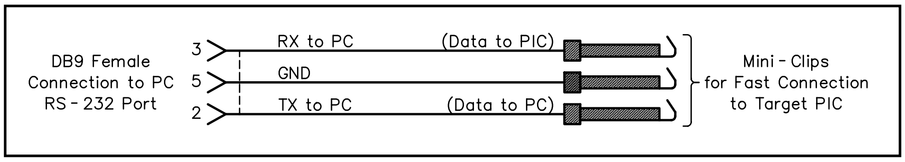

Figure 2—This simple adapter cable allows easy debugging of PIC applications with any PC-based terminal program. Employing clip leads makes it easy to do on-the-spot debugging and probing of the inner workings of your PIC programs. You will want to make one of these early on, because you won't feel like making it when you really need it!

grammer's socket keeps the development cycle short and productive (much like it is on a PC). This also saves time in the long run at the minor expense of having a few extra PICs lying around.

Debugging Tip 3

My third and last debugging tip is to use an RS-232 link to a PC during development, to show what is happening inside the PIC during program execution. In the early stages of development, I usually use the LCD (if the project has one) to show what is happening in the program. I write to the display much like you would use PRINT statements in *BASIC* to print the values of internal variables, or just to print out where the program is while it's running.

As I get farther along in writing the software, I'll likely have the LCD tied into the main program, so it's not convenient to use the LCD for debugging any more. At that point, I switch to using an RS-232 link to a PC. All of the better programming languages have built-in RS-232 serial commands that can be used on PICs with or without UARTs. My compiler recognizes nonUART devices and automatically adds software routines that perform the RS-232 input and output. These routines may take up a little of your code space, but they really help debugging.

In its simplest form, an RS-232 link only uses one PIC pin. The *C* compiler I use can configure any I/O pin for RS-232 I/O, and there is usually one pin available for debugging purposes. The PIC's RS-232 pin is connected to the PC's RS-232 receive pin, a ground wire is added to connect the PC's ground to the PIC's and off we go! (See Figure 2). Using any terminal program (*Windows* or DOS, configured for the right number of bits and data rate), the data from the PIC can be viewed as the program executes. If you need to pause the program at various points, you can add an RS-232 input to another of the PIC's pins and use it to have the PIC wait until it receives a character from the terminal program.[11]

Using RS-232 for debugging really reinforces Debugging Tip 2. That is, you will probably be making rapid changes to the program as you move debugging PRINT statements around, and you want to keep your efficiency high. You won't want to wait while another PIC is erased before trying the next experiment. So, keep plenty of PICs roasting in the EPROM eraser at all times! I'm not suggesting that you keep hacking code until it seems to work! Even the most experienced professionals learn by doing. I'm saying that while "learning more by doing more," you keep your debugging efficiency high.

For your next project, you can probably reuse much of the code that you developed for previous projects and speed your development time even more. As I mentioned earlier, that's how I develop many of the pieces for my projects.

Simple Project Displays

You may notice many articles nowadays that include LCDs. Available from many manufacturers, LCDs are really complete display subsystems.* They are commonly programmed in four-bit nibbles and can display the full upper- and lowercase ASCII character set.

Different display models are available ranging from 16 character and one line to 40 characters and four lines. LCD costs start at under $20. The displays are 14 pin devices, with 11 pins to deal with when programming. LCDs have an 8-bit mode and a 4-bit mode. In 4-bit mode, you only need four data lines and two control lines to completely control the display; see the accompanying figure. Using the four-bit mode saves on I/O pins, which with a PIC, is usually important (see Figure A).

Although it is relatively easy to program these displays, it can be made even easier by buying one of the many add-on serial adapters.† These serial adapters allow the display to be accessed with serial bit streams from a single PIC pin. Interestingly enough, these serial adapters are themselves usually built with PIC microprocessors!

Using one of these displays boils down to three simple functions:

- Initialize the display; this clears the entire display.
- Set the line (ie, the first, second, etc) to write to. This also sets the cursor to the beginning of the line.
- Write text to the display. You can write single characters or entire strings to the display. The display itself takes care of positioning each character.

Translated into *C* code, these statements look like this:

```
        init_lcd();
// Initialize and clear the LCD Display
        first_line();
// Set cursor to first line
        write_lcd("Hello World...");
// Write something
        second_line();
// Set cursor to second line
        write_lcd("QST is the best");
// Write something else
```

The result can be seen in Figure B. These are exactly the functions I include in my PIC projects that use the LCDs. For an example of the code used to drive a display, see my 2-meter FM receiver Web page.‡
—*Steve Hageman*

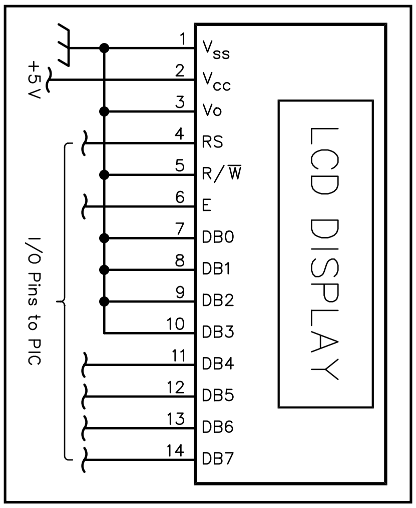

Figure A—By using an LCD's four-bit nibble mode, the number of I/O pins required for operation are reduced to six.

Figure B—These simple-to-use character-based displays really give a professional appearance to our projects. They can be programmed serially themselves, or connected directly to a PIC through six I/O pins.

*For example, see the Optrex character displays available from Digi-Key and others (Digi-Key Corp, 701 Brooks Ave S, Thief River Falls, MN 56701-0677 tel 800-344-4539, 218-681-6674; fax 218-681-3380 **http://www.digikey.com**).

†Serial Backpak from Scott Edwards Electronics, **http://www.seetron.com**

‡**http://www.sonic.net/~shageman/2_meter.html**

Notes

[1]Microchip Technology Inc, 2355 W Chandler Blvd, Chandler, AZ 85224-6199; tel 602-786-7200, fax 602-899-9210; **http://microchip.com**.

[2]See John Hansen, W2FS, "Using PIC Microcontrollers in Amateur Radio Projects," *QST*, Oct 1998, pp 34-40.—*Ed.*

[3]Steve Hageman, "Build Your Own Network Analyzer—*Part 1,*" *QST*, Jan 1998, pp 39-45; *Part 2, QST*, Feb 1998, pp 35-39.

[4]Steve Hageman, "A 2-Meter FM Receiver with PC Control," *QST*, Feb 1999, pp 35-40.

[5]**http://www.sonic.net/~shageman/pna.html**

[6]**http://www.sonic.net/~shageman**

[7]See Note 1.

[8]microEngineering Labs, Inc, Box 7532, Colorado Springs, CO 80933, tel 719-520-5323; fax 719-520-1867; **http://www.melabs.com**.

[9]See Note 6.

[10]*CCS PCM*, A *C* compiler for mid-range PICs is available at **http://www.ccsinfo.com**.

[11]Be sure to note that since we are not using any handshaking lines with the RS-232 connection, set your terminal program's handshaking parameters to *none*, or as it is sometimes called, *no flow control.*

You can contact Steven Hageman at 9532 Camelot Dr, Windsor, CA 95492; **shageman@sonic.net**.