By Al Williams, WD5GNR

Bridge the Digital Divide—*Basically*

If you think working with microprocessor-based circuits is frustrating and complicated, you haven't been introduced to the Basic Stamp—a powerful, yet friendly, digital building block that puts scores of smart circuits at your fingertips. Here's how to get started.

Hams often speak in strange lingo. As a ham you should be able to easily decipher the following:

TU OM ES GL 73 DE WD5GNR.

But what about the next group?

```
BSF PORTA,3
MOVF SBYTE,W
ADDWF PCL,F
```

Unless you know about PIC microprocessors, those three lines probably seem like gibberish. Because an increasing number of ham radio projects depend on microprocessors, don't be surprised if you can decipher this new "code" sometime in the not-too-distant future.

Microprocessors (or microcontrollers) offer unparalleled flexibility and can replace dozens or even hundreds of other components in modern radio systems. Microprocessors no longer require expensive, special-purpose programming hardware, so more designers are using them instead of ordinary logic ICs.

Most microcontrollers have limited memory capacities, however, so designers often use assembly language—like the snippet above—to control the microprocessor and conserve precious memory space. Although that approach leads to efficient programs, many people find assembly language difficult to learn and tricky to work with—especially the bare-bones assembly languages that usually accompany microcontrollers.

If you want to design and build modern projects, you may not have to learn assembly language. The fastest possible route from idea to finished product is via a line of small controllers that use a simple-to-learn *Basic*-like language called *PBasic*. Although *PBasic* is easy to learn and use, it has powerful commands tailored to the types of tasks hams and computer experimenters want microcontrollers to perform. On top of that, these microcontrollers—called Basic Stamps—require practically no external components.

The Hardware

There are several types of Basic Stamp processors, all made by Parallax.[1] More expensive Stamps have more memory and are faster than the less expensive chips. The two most common Stamps are the BS1 and the BS2. The BS1 has eight I/O pins and stores 256 bytes of instructions. Although 256 bytes of program storage (and 14 bytes for variables) sounds like a miniscule amount in these days of megabytes and gigabytes, for a small controller, it's often more than you'll need. The BS2 has 16 I/O pins and also stores 2048 bytes of instructions (along with 32 bytes of data).

You can buy the BS1 on a small PC board or as a 14-pin SIP. The BS2 looks like a standard 24-pin IC, but it's really a tiny PC board with IC pins. In addition to the BS2, you can also get the BS2SX, BS2E or BS2P which have more memory and features but are more expensive. These processors are also faster than the BS2, but consumes more power. For most projects, the BS2 is a good trade-off between cost, speed and power. In this article I'll focus on the BS2 exclusively. If you learn to use it, you'll have no problems with the other chips.

One of the best features of the Basic Stamp is how easy it is to connect the hardware. The BS2 has its own regulator and clock circuits. You can simply connect a 9-V battery (the voltage can range from 6 to 15 V) and the Basic Stamp is ready to go. It's that simple. Of course, you also need to connect whatever circuitry you want to monitor or control, but the microprocessor itself merely needs a power supply. If your project already has a dc supply, you can probably use that instead.

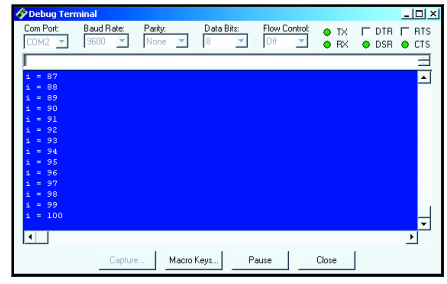

Figure 1— One handy use for the Stamp's built-in port is to print informational messages for debugging purposes right back to the Stamp program. You can do this with the *DEBUG* statement, as shown here.

The BS2 is quite different from traditional microprocessors that need regulated power supplies (perhaps even multi-voltage supplies) and external clock circuits.

Programming

The Stamp, super chip that it is, won't do a thing without programming. The BS2 connects to a PC using the PC's serial port, but because the BS2 doesn't have a standard RS-232 port, you may need to devise a way to connect a serial cable to your project. Typical BS2 projects have

[1]Notes appear on page 34.

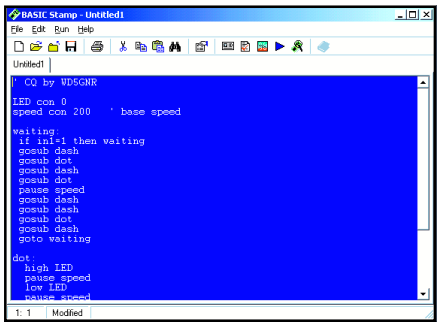

Figure 2—Using *Parallax* software to create a program to call CQ. The complete program listing is shown elsewhere in this article.

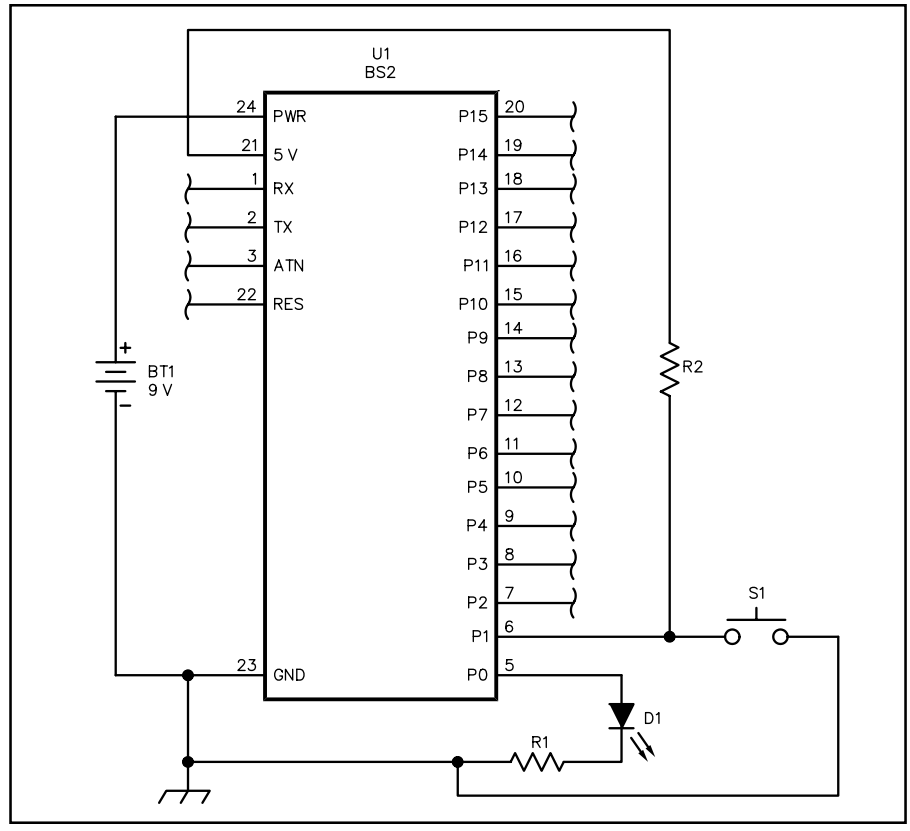

Figure 3—You can use this simple circuit to start experimenting with the Stamp.
B1—9-V battery
U1—BS2-IC (Basic Stamp)
D1—Red LED
R1—470 Ω, ¼-W resistor
R2—10 kΩ, ¼-W resistor
S1—SPST pushbutton switch

RS-232 ports anyway, so that's not always a problem. The port requires only four pins, so if you only need to program the device you can put a simple four-pin connector on the board and wire a custom cable.

You can also buy a carrier board from the manufacturer. This is simply a PC board with some uncommitted holes and a serial port connector. If you want to work on a solderless breadboard (highly recommended for getting started) you can create a custom cable, use a prototyping adapter,[2] or buy the Stamp in a special form that can plug into a breadboard.[3]

In addition to the cable you'll need special software, which you can freely download from the Internet. You can also get the entire manual for the chip at the same Web site in Adobe PDF format at no charge.[4]

The programming software is shown in Figure 2. You simply enter your program and press Control+R (or the Run|Run menu) to download the program and execute it. Once you program the Stamp, it will remain programmed indefinitely—even if the power cycles. The PC connection is only required to program the chip.

The Stamp editor works like most *Windows*-based editors. If you create a new file, however, it appears as a tab near the top of the editor. This allows you to switch between multiple files even though only one can reside in the Stamp at one time. (To begin with, work with one file at a time.)

Your First Program

Once you have a programming cable and a 9-V battery (or other suitable power supply), you're almost ready to go. The only remaining task is to connect an external device to measure or control. Figure 3 shows a simple test circuit that consists of an LED and a switch. The pull-up resistor (R2) makes sure that P1 is high unless you press the switch (S1). The Stamp can read the state of P1 and change the state of P0 to turn the LED on or off.

Each of the 16 I/O pins on the Stamp can be used as an input or an output. By default, the pins start as inputs, but you can change them to outputs at any time. You can even have pins that are sometimes inputs, sometimes outputs, depending on your program.

One potentially confusing Stamp convention is that the I/O pin numbers don't correspond to the IC pin numbers. For example, P0, the first I/O pin, is actually IC pin 5; P1 is pin 6, etc. It's easy to get confused.

Here's a simple BS2 program that will blink the LED connected to P0:

```
again:
  toggle 0
  goto again
```

If you've ever written a *Basic* program you can probably figure this out. Consider the program line by line:

again: This is a label. You can use almost any word you like, but it must begin with a letter and can't contain spaces. The label ends with a colon. Think of a label as a bookmark that holds a place in your program. Later, you can return to the bookmark to start the program at the labeled step.

toggle 0 This command forces pin 0 to be an output. If the output is a logical 0 (the default), the command toggles it to a logical 1 (about 5 V). If the output is already a logical 1, the command toggles it to a logical 0.

goto again This command simply executes the program back at the **again:** label.

This simple program makes the LED blink so fast that it looks like it's on all the time (unless you watch the output pin with an oscilloscope). Let's slow things down with the pause command:

```
again:
  toggle 0
  pause 500
  goto again
```

The pause command makes the program stop for the specified number of milliseconds. With a value of 500, the program will blink the LED at about 1 Hz (a half-second on and a half-second off).

The toggle command is a good example of the special commands that the Stamp uses to control external devices. You can find a complete list of Stamp commands in Table 1. Simple *Basic* constructs such as **for** and **if** are present, along with a host of I/O-related commands such as **pwm**, **freqout**, **pulsin**

Table 1
Basic Stamp II Commands

Command	*Description*	*Notes*
Branch	Jump to a label based on an index	
Button	Monitors a button	Provides debounce and auto repeat functions
Count	Counts pulses	
Data	Stores data in EEPROM	Stores at compile time
Debug	Write to the debug terminal	Usually uses your PC
Dtmfout	Generate DTMF tones	
End	Halt program execution	
For/Next	Basic-language loop	
Freqout	Generate one or two tones	
Gosub/Return	Subroutine call	
Goto	Unconditional jump	
High	Sets I/O pin to logic 1	Implies output
If/Then	Basic-language control flow	Conditional goto only
Input	Changes I/O pin to input	
Lookdown	Searches a table for a match	
Lookup	Finds an entry in a table	
Low	Sets I/O pin to logic 0	Implies output
Nap	Pause in low power mode	
Output	Changes I/O pin to output	
Pause	Pause program execution	1 mS resolution
Pulsin	Measures a pulse	2 µS resolution
Pulsout	Generates a pulse	2 µS resolution
Pwm	Pulse width modulation	Use to control motor speeds or generate analog voltages
Random	Generate a random number	
Rctime	Measures RC network charge	Use to read a potentiometer or any resistive or capacitive sensor
Read	Reads data from EEPROM	
Reverse	Changes I/O pin to opposite state	
Serin	Read RS-232 data	243 baud to 50 K baud
Serout	Send RS-232 data	243 baud to 50 K baud
Shiftin	Synchronous serial input	Use with SPI EEPROMs, A/Ds, etc
Shiftout	Synchronous serial output	Use with SPI EEPROMs, A/Ds, etc
Sleep	Pause in low power mode	
Toggle	Reverses logic level of I/O pin	Implies output
Write	Writes data to EEPROM	Stores at run time
Xout	Generates X-10 commands	Requires external hardware

and so forth.

Commands such as **toggle** require a pin number. You can also access pins as variables. For example, to read the switch as a binary digit, you can refer to **in1**. You can also directly set the LED's state using **out0**. Here's a simple program that turns on the blinking LED when you push the button:

```
waitbtn:
  if in1=1 then waitbtn
again:
  toggle 0
  pause 500
  goto again
```

Notice that the **if** statement can only jump to a label (such as waitbtn). You can't use the **goto** keyword, and you can't execute any statements. You can only jump to a label.

Of course, there is more than one way to accomplish any programming task. Here's another way to blink the LED:

```
again:
  high 0
  pause 500
  low 0
  pause 500
  goto again
```

Here, the **high** and **low** commands set the exact state of the output pin instead of switching it to the opposite state.

Subroutines are fundamental to re-using *Basic* code, and the Stamp supports the **gosub** command as any *Basic* should. A **gosub** command transfers control to a label much like a **goto**. Unlike a **goto**, however, a return statement will go back to the line following the **gosub**. For example:

```
again:
  gosub blink
  goto again
blink:
  toggle 0
  pause 500
  return
```

Not only is this easier to read, but it also saves space when you need to blink the LED from more than one place in your program. Even with this simple slate of commands, you can write some ham radio software right away. Listing 1 shows a very simple program that blinks CQ on the LED after you press the button. This isn't the most efficient way to write the program, but it works.

Notice that the program in Listing 1 uses several statements (near the top) that use the **con** keyword. This defines a

Listing 1. Sending CQ

```
' CQ by WD5GNR

LED con 0
speed con 200   ' base speed

waiting:
 wait for switch
  if in1=1 then waiting
  gosub dash
  gosub dot
  gosub dash
  gosub dot
  pause speed
  gosub dash
  gosub dash
  gosub dot
  gosub dash
  goto waiting

dot:
  high LED
  pause speed
  low LED
  pause speed
  return

dash:
  high LED
  pause speed*3
  low LED
  pause speed
  return
```

constant. By using a constant in the **pause** statements, you can change the Morse code speed by changing one number. Without the constant you'd have to change every pause statement separately. Lines that start with an apostrophe are comments and don't affect the execution of the program.

Learning More

One of the strengths of the Stamp is that many of its *PBasic* commands replace dozens —or even hundreds— of lines of assembly language.

There is a wealth of information on the Web about the Basic Stamp. The Parallax Web site is a good place to start. You might also enjoy my Basic Stamp FAQ.[5] If you like to read paper instead of computer screens, you'll enjoy my Basic Stamp book.[6]

With the Basic Stamp you can build computer controls into your ham radio projects with very little investment. Yes, the Stamp is more expensive than a barebones microprocessor, but the price is more reasonable after you deduct the cost of the extra components, the development tools, the complexity and the time spent writing and debugging assembly language.

One of the strengths of the Stamp is that many of its *PBasic* commands replace dozens—or even hundreds—of lines of assembly language. That means you can write programs in minutes that would take hours or days using traditional methods. A great example of this is the Stamp's **DTMFOUT** command. You can use this to easily generate TouchTones on any output pin. You can connect a piezo speaker (or filter the output and feed it to a transmitter). Suppose you have such a speaker connected to pin 0 of the Stamp. Here's the entire program required to dial a telephone number:

```
DTMFOUT 0,[1,8,0,0,5,5,5,1,2,1,2]
```

That's it! How do you trigger it? Simply turn the Stamp on. Push a button to make the Stamp dial and hold it down until it's done. If you really wanted to wait for a button (and leave the Stamp on all the time), you could change the program a little:

```
top:
' switch 0 when on
  if in1=1 then top
  dtmfout 0,[1,8,0,0,5,5,5,1,2,1,2]
done:
if in1=0 then done
' wait for button up
  goto top
```

Generating TouchTones in software usually requires sophisticated wave synthesis techniques to generate the two simultaneous sine waves. But when using *PBasic*, you don't care. You simply use the **DTMFOUT** command and the Stamp does the rest!

Other sophisticated commands can handle serial I/O, pulse-width modulation, resistance or capacitance measurements and pulse counting. Of course, the Stamp also handles sophisticated integer math, something that's usually troublesome with ordinary microcontrollers.

Variables

If you're going to use math, you'll probably want to use variables. The Stamp provides several registers (working memory areas for data) that you can address by name. It's usually better, however, to ask the Stamp to assign registers to variables that have meaningful names. For example:

```
adin      var     byte
counter   var     word
```

This defines two variables. One is a byte (eight bits) and the other is a word (16 bits). The byte's name is **adin** and the word is named **counter**. Variables can also be of type **bit** (a single bit) or **nib** (four bits). The *PBasic* program automatically assigns registers to these variables (until you run out of registers and get an error).

You can also use the same syntax to provide an alias for another variable. This is useful if you want to reuse a single register in two non-conflicting places. For example:

```
tmpvar    var     adin
' can't use tmpvar and adin together
```

You can also give names to constants. For example:

```
pi100     con     314
limit     con     100
```

Math

The Stamp can do full-featured, 16-bit integer math. That includes multiplication and division, which are usually unpleasant to do on a microcontroller. What Stamps can't do is handle floating-point numbers. Therefore, 10/3 (10 divided by 3) results in an answer of 3, which can cause problems.

Another subtle point is that the Stamp evaluates math expressions from left to right, which is not how you normally work an equation. For example, consider this statement:

```
X=3+5*2
```

In high school math you learned that the correct answer is 13 (you do the multiplication before the addition). The Stamp, however, goes strictly left to right, so it computes the answer as 16 (addition first). Luckily, *PBasic* (for the Stamp II) supports parenthesis, so you could write:

```
X=3+(5*2)
```

This will produce the answer you expect without having to rearrange the equation.

There are several tricks to eliminating floating-point math. Sometimes you simply need to rearrange your equation. Suppose you read a value from an analog-to-digital converter (ADC). The byte is in a variable (**adin**). In addition, you have a constant defining the reference voltage input to the ADC (nominally 5 V). Because the ADC returns a number between 0 and 255 (a span of 256), each count is equivalent to about 19.5 mV (5/256) if the reference voltage is 5 V. Consider this code:

```
adin      var     byte
value     var     word
ref       con     5
value = ref/256*adin
```

This won't work because ref/256 is 0, so **value** will always be 0. You must rewrite the equation so the multiplication occurs first:

```
value = adin*ref/256
```

Even then, **adin** must rise above 52 before **value** can reach 1, which wastes a lot of resolution. What if you measured decivolts instead of volts (that is, use 0.1 V units)? Now, the reference value is 50 (5 V is 50 decivolts). So now, **value** will change for every five or six increases in **adin**.

You can't carry this reasoning too far, though. Suppose you decide to go to one more decimal point (centivolts, or 0.01 V increments). Of course, the ADC can produce only about half of that resolution. For the sake of argument, however, don't worry about that yet.

Expressing the reference value in centivolts results in a **ref** constant of 500. The problem is, when you multiply **adin** by ref, the maximum result is 255*500 or 127500. The largest 16-bit number is 65535. The Stamp will quietly overflow and produce an incorrect result.

Serial Capabilities

The Stamp has a special built-in half-duplex serial port (this is the port you use to program it). Your program can also use this serial port, or you can use any pin as a TTL-level serial input or output. One handy use for the built-in port is to print informational messages for debugging purposes right back to the Stamp program. You can do this with the **DEBUG** statement. For example:

```
i var word
```

```
for i = 1 to 100
debug ?i
next
```

This produces the output in Figure 1. For more general-purpose serial communications you can use the **Serin** and **Serout** commands. You can specify any of the 16 general-purpose I/O pins (0-15) or you can use the special pin number 16 to specify the built-in port. You can also control the baud rate and certain other parameters.

Because of the Stamp's robust inputs, you can actually connect a serial output through a 22-kΩ series resistor directly to an I/O pin. This works even though the pin may have to absorb ±12 V (just don't forget the resistor or you may damage the Stamp). You can usually drive an RS-232 receiver directly from a Stamp pin, although using 0 and 5 V for RS-232 signaling isn't standard.

Of course, you can also use an RS-232 driver (like the Maxim MAX232 chip[7]) to generate (and accept) true RS-232 signaling levels. The Stamp can support either mode of operation. The built-in port contains a level converter that "steals" –12 V from the transmitter (which limits it to half-duplex operation). You can find more details in the Stamp manuals.

Don't forget that the Stamp does one thing at a time. Therefore, if you're waiting for serial data, you can't do anything else until the data arrives, or your timeout expires. Similarly, if data arrives while you're not listening, it's simply lost. That means accommodating serial data requires careful planning and some form of handshaking (which the Stamp supports).

Pulse-Width Modulation

Another intriguing Stamp capability is pulse-width modulation. The **PWM** command allows you to generate a pulse stream with a specific duty cycle. For example, if you set the **PWM** command to 128, the output pulses will be high as much as they are low (50% duty cycle). Changing the value to 64 will make the output low more often than high (25% duty cycle).

You can use this pulse stream to control the brightness of an LED (or lamp) or even the speed of a motor. Although the Stamp has enough muscle to drive an LED, you'll need some extra circuitry to drive a motor. One of the most useful things you can do with pulse-width modulation is to use a simple RC network to integrate the pulses into a voltage. This allows you to create an analog voltage on an output pin with very little external circuitry. The voltage across an external capacitor will be proportional to the PWM duty cycle. So, if the duty cycle is 128, the voltage will be about 2.5 V.

The only problem with the Stamp's PWM system is that the Stamp does not multitask. Therefore, when you use PWM to charge a capacitor to a certain voltage, you have to eventually stop and do something else. The Stamp automatically switches the I/O pin to an input state, which has a high resistance. Unfortunately, the rest of your circuit may present enough of a load to rapidly discharge the capacitor.

This isn't always a problem. For example, suppose you create a capacitance meter (see below). You decide to make the output a voltage you can read with your digital voltmeter. Your meter's input resistance is probably 10 MΩ or more, so as long as you make the Stamp execute the **PWM** command regularly (maybe once a second or so), you won't see any significant error in the output.

On the other hand, suppose the voltage is driving a light bulb, which will quickly discharge the capacitor. In this case it's best to buffer the **PWM** output with an op amp.

Resistance and Capacitance

My high school math teacher always said, "You have to use what you know to discover what you don't." Computers aren't good at measuring analog quantities such as resistance and capacitance. On the other hand, they are *very* good at measuring time. You can use the Stamp to measure the time it takes for an RC network to charge or discharge, and that time relates to the value of a resistor and a capacitor in the network.

If you use a fixed capacitor you can measure the resistance (perhaps a potentiometer or a thermistor). If you provide a fixed resistor, the time will be proportional to a changing capacitance. The command that measures time is **Rctime**. You can charge or discharge the network (using the **High** or **Low** commands) and invoke **Rctime** to determine how long it takes the capacitor's voltage to reach the opposite state.

Pulse Measurements

The Stamp can also measure pulses using **Pulsin**, which returns the width of a positive- or negative-going pulse. You can also count the number of pulses over a given period using the **Count** command. These commands are excellent for measuring relatively low frequencies.

If you want to generate pulses you can use the **Pulsout** command. Don't forget, however, while you're measuring or generating pulses, nothing else is happening. So you can't constantly monitor pulses—you'll eventually have to stop to do additional processing.

Brave New World

What will you do with a Basic Stamp? Here are some ham radio ideas:

- A remote control for an RS-232 transceiver (use **Rctime** to read potentiometers and **Pulsin** to monitor an optical encoder).
- A simplex repeater (use a digital speech recorder).
- A control system for an auto-tuner.
- A smart rotator controller.

Basic Stamps aren't useful in every application. But for the many tasks they will handle, you can't find anything easier to program. The expense of the chip is minor compared to the expense of buying special hardware and software to program other microcontrollers—not to mention the expense of hours of frustrating programming in assembly language!

There's plenty more to learn about Basic Stamps, but luckily, there are plenty of online resources and books to help.[8] There is also an active e-mail reflector that supports the Stamp.[9] Be sure to check out my Basic Stamp FAQ[10] and my book on the Stamp,[11] which has many projects and some tips on how to move from the Stamp to the PIC, a more traditional microcontroller.

If you've been putting off learning about microcontrollers, the Stamp is the perfect way to get your feet wet. Just be warned: Once you've done one project, you'll think of at least a hundred you'll never have time to start!

Notes:

[1]Parallax Inc, 599 Menlo Dr, Suite 100, Rocklin, CA 95765; **www.parallaxinc.com.**

[2]See **www.al-williams.com/awce/asp2.htm** for more about the solderless breadboard adapters.

[3]The OEM Stamp can plug into a breadboard. See **www.parallaxinc.com/html_files/products/oem_stamp_brief.asp**.

[4]**www.parallaxinc.com/html_files/downloads/download.htm** has the software and documentation. You can also download free course material from **www.stampsinclass.com**. Although the course material is meant for classroom use, you can easily use them for self-study.

[5]The Basic Stamp FAQ is at **www.al-williams.com/wd5gnr/stampfaq.htm**.

[6]*Microcontroller Projects with Basic Stamps*, published by CMP Books, is available from the ARRL.

[7] Maxim's Web site is **www.maxim-ic.com**.

[8] See footnote 4.

[9] Log into **groups.yahoo.com** and sign up for the basicstamps group.

[10] See footnote 5.

[11] See footnote 6.

You can contact the author at 310 Ivy Glen, League City, TX 77573; **alw@al-williams.com**.