Putting the Spotlight on BASIC Stamp Projects, Hints, and Tips

Stamp Applications

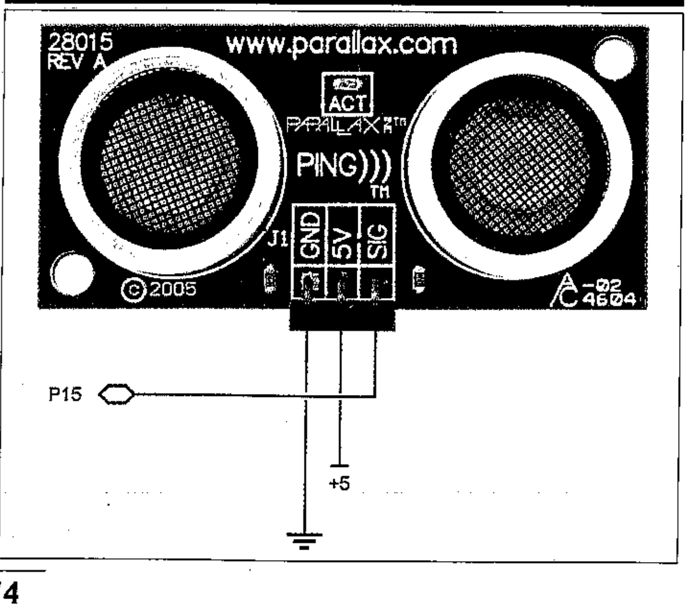
Ping ... I See You

I used to work for a man named Bob who insisted — and quite frequently — that most of us needed to be exposed to the same piece of information five to seven times before that information could be absorbed. I didn't always agree with Bob's philosophy, but in case he was right, I thought we'd work through the mysteries of conditional compilation again. Conditional compilation is worth mastering; it allows us to write one program that will work on nearly any BASIC Stamp module.

Maybe I'm just taking things for granted. Being on "the inside" and close to the development of the BASIC Stamp IDE, I completely understand conditional compilation and how to take advantage of it. Apparently, however, I haven't done a very good job getting the word out, as I keep getting a lot of questions on this subject. So, I'm going to try again.

Let's start from the beginning. Why should we even bother with conditional compilation? Well, it depends, really. If we're going to write a program that will *never* (yeah, right ...) need to run on another BS2 family module, then we don't need to bother. What if, however, we want to share our cool program with a friend who uses a different module? What if we wrote our program for the BS2 and our friend is using a BS2sx? Most programs will run without change, but the use of certain PBASIC keywords will require the code to be updated to run properly on the BS2sx. By using conditional compilation up front, we can save ourselves and others trouble later.

Figure 1. Ping Connections.



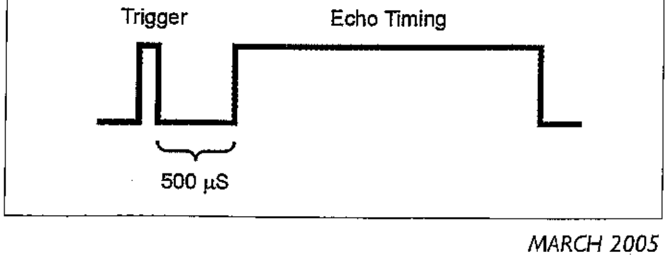
Ping ... Ping ...

Before getting into the gritty details, let's have a little bit of fun with a simple program that actually uses conditional compilation. Sonic range finding modules are very popular with robotics builders and experimenters, and Parallax has recently created a new module called Ping that makes sonar range finding pretty easy. Honestly, I really like the Ping sensor, as it requires only one I/O pin, works with any BASIC Stamp module, and is very low cost.

As you can see by Figure 1, the connection is a no-brainer — connect power (+5 volts), ground (Vss), and a signal line to a free BASIC Stamp pin. With the Ping module, the I/O pin serves as both the trigger output and the echo input. Have a look at Figure 2 and we'll walk through how it works.

Initially, the trigger pin is made an output and a short pulse (five to 10 µS) is used to trigger the Ping (we'll use **PULSOUT** to generate the trigger). The next step is what allows it to be used with any BASIC Stamp. The Ping module delays the trigger to the sonic transmitter element for 500 microseconds. This allows the BASIC Stamp to load the next instruction (**PULSIN**) and be ready for the return echo. Once the echo pulse is measured, a bit of math is used to convert the pulse width to distance.

Figure 2. Ping Signaling.



Let's look at the subroutine that handles the Ping sensor:

```
Get_Sonar:
  Ping = 0
  PULSOUT Ping, Trigger
  PULSIN  Ping, 1, rawDist
  rawDist = rawDist */ Scale
  rawDist = rawDist / 2
  RETURN
```

The code starts by making the output bit of the trigger pin 0, and the reason for this is that **PULSOUT** makes the trigger pin an output, toggles its state, delays, and then toggles that pin back to the original state. Since the Ping module is looking for a low-high-low pulse to trigger the measurement, presetting the pin to 0 makes this happen.

After the trigger is sent, **PULSIN** is used to measure the width of the echo pulse. As I stated earlier, the 500 microsecond delay in the Ping allows **PULSIN** to get loaded and ready. There is no danger of **PULSIN** timing out, as even the BS2p (fastest BASIC Stamp module) won't time out for about 49 milliseconds. For you clever readers who are wondering what happens if we forget to make the signal pin an input after the trigger pulse ... no worries, there is protection on the Ping sensor so that no harm is done if both sides are trying to drive the signal line.

Now, we have to get back to that pesky conditional compilation stuff. Remember that the various BASIC Stamp modules run at different speeds and — with some instructions — the speed differences give us different resolutions. Let's look at the units returned by **PULSIN**:

BS2, BS2e	2.00 ms
BS2sx, BS2p	0.80 ms
BS2pe	1.88 ms

Let's see how conditional compilation lets us handle the differences in the various modules:

```
#SELECT $STAMP
  #CASE BS2, BS2E
    Scale       CON       $200
  #CASE BS2SX, BS2P
    Scale       CON       $0CD
  #CASE BS2PE
    Scale       CON       $1E1
#ENDSELECT
```

The instructions prefaced with "#" are used in the conditional compilation process. These instructions actually get processed before our program is tokenized. This allows constant values and even bits of code we choose to be included in the program based on the BASIC Stamp module in use. So, using the code above, if a stock BS2 module is installed, the constant — called Scale — will have the value $200. If we unplug the BS2 and swap in a BS2p, then we will program the module Scale to have the value $0CD.

Let's get back to the program; we'll cover more conditional compilation later. The raw value from PULSIN is converted to units of one microsecond with this line of code:

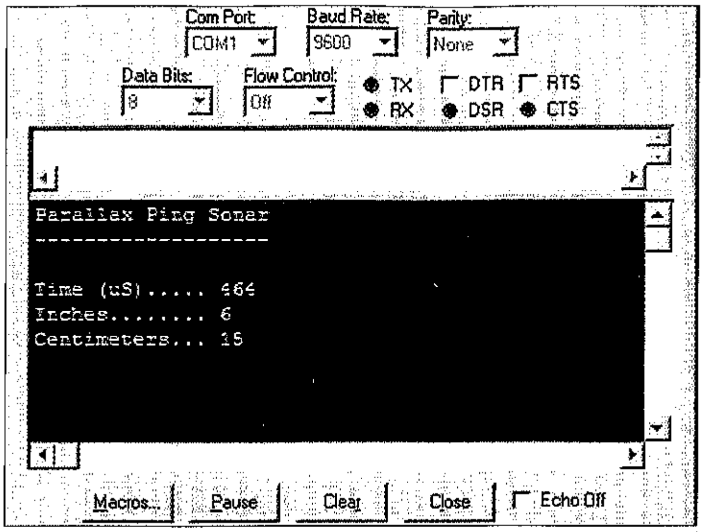


Figure 3. Ping Terminal Output.

```
rawDist = rawDist */ Scale
```

We're forced to use the "*/" (star-slash) operator to

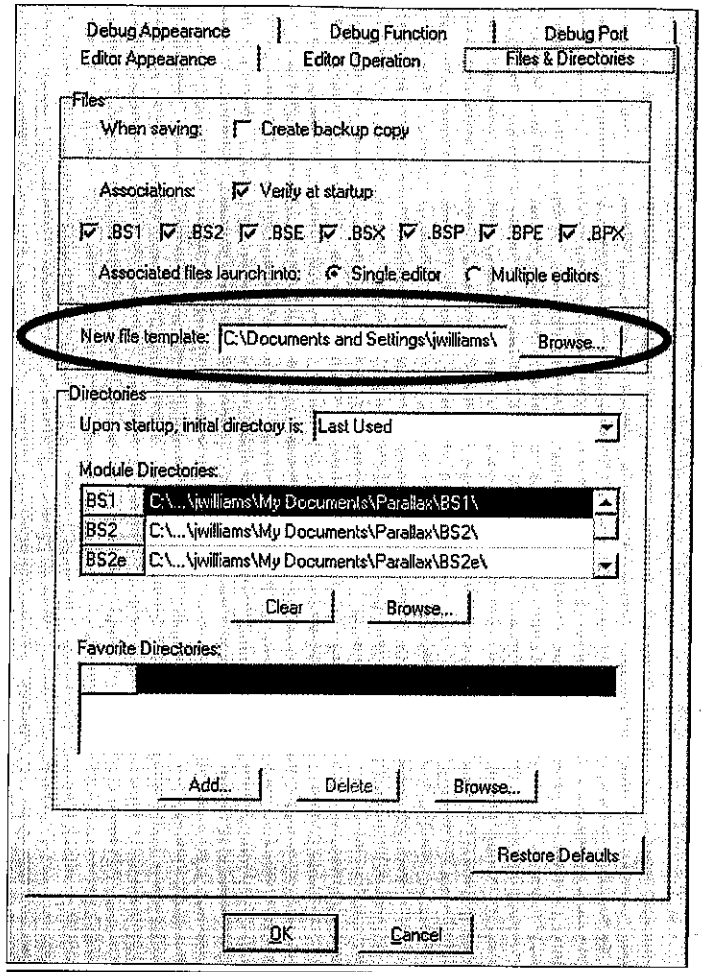


Figure 4. IDE Prefs.

account for the fractional units when using the BS2sx, BS2p, or BS2pe. For review, "*/" works like multiplication, but in units of 1/256. To determine the various values for Scale, we multiply the **PULSIN** units by 256 and take the (rounded) integer result. Things work out like this:

BS2, BS2e	INT(2.00 x 256) = 512 ($200)
BS2sx, BS2p	INT(0.80 x 256) = 205 ($0CD)
BS2pe	INT(1.88 x 256) = 481 ($1E1)

I prefer to use hex notation for values that are used with "*/", as the upper byte represents the whole portion of the value and the lower byte relates the fractional portion (in units of 1/256). The pulse is measured and converted to microseconds, and before returning to the caller, we'll divide the raw value by two. Why? Well, the pulse we've just measured actually accounts for the distance to and from the target — actually twice as wide as we need, hence the division.

Now, to convert to distance: At sea level and room temperature, we assume that sound travels at about 1,130 feet per second, and by multiplying by 12, we get 13,560 inches per second. By taking the reciprocal, we find that it takes about 73.746 microseconds for sound to travel one inch. For those who prefer the metric system, we can convert 13,560 inches to 34,442 centimeters and a timing value of 29.034 microseconds to travel 1 centimeter.

For our conversion code, we'll use the other fraction math operator, "**". This is similar to "*/", except that it uses units of 1/65,536. This means that, in the 16-bit values used by the BASIC Stamp, we can use it to multiply by fractional values of less than one. In our program, we can convert 73.746 microseconds to a constant value like this:

1 / 73.746 → INT(0.01356 x 65536) = 889 ($379)

With that, we can look at the rest of the program:

```
Reset:
  DEBUG CLS,
        "Parallax Ping Sonar", CR,
        "-----------------", CR,
        CR,
        "Time (uS).....      ", CR,
        "Inches........      ", CR,
        "Centimeters...      "

Main:
  DO
    GOSUB Get_Sonar
    inches = rawDist ** RawToIn
    cm = rawDist ** RawToCm

    DEBUG CRSRXY, 15, 3,
          DEC rawDist, CLREOL
    DEBUG CRSRXY, 15, 4,
          DEC inches, CLREOL
    DEBUG CRSRXY, 15, 5,
          DEC cm, CLREOL

    PAUSE 100
  LOOP
  END
```

The Reset section simply sets up the text portion of the Debug Terminal window and, in Main, we measure the distance, do the conversions, and display the results. Figure 3 shows the output of the program.

Stamping Under Any Condition

Time to get back to conditional compilation. While most PBASIC instructions don't require parameter changes when moving from one module to another, there are a few that do:

COUNT	Units for Duration of COUNT window
DTMFOUT	Units for OnTime
FREQOUT	Units for Duration, Freq1, and Freq2
PULSIN	Units for Variable (measured pulse)
PULSOUT	Units for Duration
PWM	Units for Duration
RCTIME	Units for Variable (measured RC delay)
SERIN	Units in Timeout, value of Baudmode
SEROUT	Units in Pace and Timeout, value of Baudmode

The most common issue among BASIC Stamp users when moving from module to module is with **SERIN** and **SEROUT**. So common are these instructions that I have built the following section into my default programming template:

```
#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200         CON       813
    T2400         CON       396
    T4800         CON       188
    T9600         CON       84
    T19K2         CON       32
    TMidi         CON       12
    T38K4         CON       6
  #CASE BS2SX, BS2P
    T1200         CON       2063
    T2400         CON       1021
    T4800         CON       500
    T9600         CON       240
    T19K2         CON       110
    TMidi         CON       60
    T38K4         CON       45
#ENDSELECT

SevenBit        CON       $2000
Inverted        CON       $4000
Open            CON       $8000

Baud            CON       T9600
```

If **SERIN** and **SEROUT** aren't used by a given program, there is no harm done — and it's far handier to have constants predefined than to have to look them up. This gives me the opportunity to bring up another programming tip. I frequently get code that looks like this:

```
SEROUT 15, 16468, [DEC temperature]
```

which is followed by the complaint, "Jon, this used to work with my BS2, but now it doesn't work with my BS2sx."

By now, I'm sure you see that the reason is obvious: by changing from the BS2 to a BS2sx, we are forced to update the baudmode parameter of **SEROUT**. The problem can be averted by using the conditional section above and changing the Baud definition as follows:

```
Baud            CON       Inverted + T9600
```

While we're cleaning up the code to make it easier to maintain, let's give a definition to P15 so that we know the serial output is going to a serial LCD:

```
Lcd             PIN       15
```

Now, the corrected code becomes:

```
SEROUT Lcd, Baud, [DEC temperature]
```

Where else might conditional compilation come in handy? How about program debugging? There is an instruction called **#DEFINE** that can help in this regard. For example:

```
#DEFINE DebugOn = 1
```

While developing and troubleshooting an application, we can do this:

```
#IF DebugOn #THEN
  DEBUG "Value = ", DEC value, CR
#ENDIF
```

in as many places in the program as we need.

Once the program is fully tested and working as desired, changing the DebugOn definition to zero will prevent the **DEBUG** statements in the **#IF-#THEN** section(s) from executing. It's important to understand that conditional definitions are either defined (not zero) or not. In our example above, we could, in fact, remove the **#DEFINE** DebugOn line without harm to the program. When the compiler encounters a conditional block (like **#IF-#THEN**) with an undefined symbol, the section is skipped. I don't recommend this, however, as it can lead to confusion if someone else reads code from which we've removed conditional symbol definitions. It is best to disable the conditional symbol by redefining it as zero.

Another good use of conditional definitions is variable conservation. In our sonar program, for example, practical use would usually not require both standard and metric units. We could do this:

```
#DEFINE MetricUnits = 1
```

and ...

```
#IF MetricUnits #THEN
  distance = rawDist ** RawToCm
#ELSE
  distance = rawDist ** RawToIn
#ENDIF
```

Finally, what about features that exist in the newer BASIC Stamp modules that do not exist in the older ones, LCD control, for example. Well, we can deal with that, too.

There was a project we did some time back that involved the Parallax LCD Terminal AppMod and took advantage of conditional compilation. A program can check for the availability of built-in LCD commands like this:

```
#DEFINE LcdReady = ($STAMP >= BS2P)
```

We can now put this definition to use in the following manner:

```
LCD_Command:
  #IF LcdReady #THEN
    LCDCMD E, char
    RETURN
  #ELSE
    LOW RS
    GOTO LCD_Write
  #ENDIF

LCD_Write:
  #IF LcdReady #THEN
    LCDOUT E, 0, [char]
  #ELSE
    LcdBusOut = char.HIGHNIB
    PULSOUT E, 3
    LcdBusOut = char.LOWNIB
    PULSOUT E, 3
    HIGH RS
  #ENDIF
  RETURN
```

It does take a little bit of planning and extra work to implement conditional compilation, but I think you'll find it fairly easy to do in the end, in addition to being a big time-saver when it comes to moving code from one BASIC Stamp module to another.

Installing a Template

Earlier, I mentioned my default template and its use of serial baudmode values. I've included a copy of my template in the project files at **www.nutsvolts.com** Let me share a tip that may not be obvious. You can have the BASIC Stamp IDE load this template each time you select File / New (or the New icon from the toolbar).

Start by copying the template (template.bs2) to a convenient location. Then, open the Preferences dialog (Edit / Preferences), select the Files & Directories tab, and click on the Browse button located next to the New File Template field. In the Open dialog, navigate to the location where you copied the template file, select it, and then click on Open. Lock in the setting by clicking OK at the bottom of the Preferences dialog. I find the template helps keep my programs organized and I'm sure it will work for you, too — if it doesn't quite do so, modify it until it does!

Until next time, then, Happy Stamping. **NV**

Jon Williams
jwilliams@parallax.com
Parallax, Inc.
www.parallax.com