Column #105, January 2004 by Jon Williams:

An Industrial Cup o' Joe

It's no secret that for as long as BASIC Stamp microcontrollers have existed, Stamp customers have been very industrious ... and now, with a little help from the Stamp PLC, Stamp users can have a nice product that will help them to get industrial.

The Stamp PLC isn't the first time Stamp modules have been fitted for an industrial environment, but it does offer a package that is more similar to large scale, high-dollar products – and is a good fit when a "standard" PLC (and its expensive development software) isn't required. One of my first "pro" uses of a BASIC Stamp was to replace a $1200 PLC that wasn't behaving as desired (it was part of a trade show display). When the display company couldn't reprogram the PLC (they had farmed out that task), my boss asked me to "fix" it. I did by replacing the PLC with a BASIC Stamp 1 module and some triacs. My boss was happy, the display looked and worked great, and my employer saved about $1150.

The Stamp PLC was developed for Parallax by Lawicel HB of Sweden. Lawicel HB specializes in industrial applications and has found the BASIC Stamp quite useful in many projects. With their considerable expertise, they've designed an elegant, industrialized enclosure for 24-pin Stamp modules that fully protects the inputs and outputs (optical isolation), has visual indicators for digital I/O ports, provides for an optional four channel, 12-

bit ADC, and has a clean power-supply for the circuitry. As would be expected, the Stamp PLC mounts on a standard DIN rail, and can be powered with 18 to 36 volts DC (24 VDC is standard for industrial applications). Finally, professionals will appreciate that the Stamp PLC meets the requirements to carry the CE certification mark.

Since we're starting a new year, I thought I'd try something different. As the code for the Stamp PLC is not particularly complex, what I thought I'd do is compare code for two different Stamp modules: the stock BS2 and the Javelin Stamp (my choice for Stamp PLC projects).

Whoa ... this is different; two languages in one article. Yes, it is. The reason for it though is that many BASIC Stamp users have been experimenting with the Javelin Stamp and some are not being as successful as they would like to be right out of the gate. Part of that is human nature; the BASIC Stamp is very easy to use and the Javelin Stamp is quite a sophisticated little beast and takes a bit of time getting used to, especially for those that haven't programmed in Java or C. That doesn't mean that the Javelin Stamp is difficult to use, it's just different. By looking at listings side-by-side it may help those wanting to work with the Javelin Stamp to get up and running, ultimately taking advantage of some of the Javelin's unique features.

Do keep in mind that the Javelin Stamp is still a new kid on the block, and has a lot of really great features that aren't yet available in BASIC Stamps. The timer object, for example, is very useful in the Stamp PLC and you'll see how it gets used to set the I/O scan and process interval to specific rate. Okay, then, let's crack open the Stamp PLC and get started.

Pick a Stamp ... Any 24-pin Stamp

To be perfectly honest with you, one of the trickiest aspects of the Stamp PLC is opening the enclosure so that we can install our 24-pin Stamp. Again, the enclosure is designed well and intended for a grungy industrial environment, so there's no slop. On the back (opposite the I/O connections label) you'll find three slots where you can use a small screwdriver to release the locking tabs that hold the case halves together. Go slowly, as the DIN rail lock is spring-loaded and if you're not careful it can go flying (Ask me how I know....).

Once the case is open you'll see two sockets: a wide 24-pin socket for the Stamp and a skinny-DIP socket for the optional MAX1270 ADC. Install the Stamp so that pin 1 is oriented toward the input connectors; the same for the MAX1270.

Before we put the Stamp PLC back together, we need to configure the MAX1270 input channels. We have two choices: voltage or 4-20 mA inputs. Adjacent to the right side of the MAX1270 you'll see a line of eight header posts; these are organized as two posts for each channel. If we want to measure voltage (which is software configurable for various ranges) we leave the jumper out. If we want a particular channel to measure a 4-20 mA current signal, we need to add the jumper. Figure 105.1 shows the inside of my Stamp PLC with a Javelin Stamp module installed, the MAX1270 installed, and channel four of the ADC inputs configured for a 4-20 mA current signal.

Figure 105.1: Inside of a StampPLC

What's In a Name?

Those of you that know me know that I am a bit of a maniac when it comes to what I consider "proper" code writing – a big part of that is using well-planned constant, variable, and label names to make writing, reading, and especially debugging my code a lot easier. Technically, PBASIC has no formal standards for naming and formatting, but many of us have adopted standards used by other flavors of BASIC (notably, Visual BASIC).

My experience is that Java is less tolerant of free-form (*sloppy*) formatting, and there seems to be a standard that most Java programmers follow. The standard that I follow is articulated in the book, "The Elements of Java Style" (ISBN: 0521777682). I enjoyed this little book so much that I used it as a model for a short document called "The Elements of PBASIC Style" that demonstrates how we at Parallax are now formatting our programs (this is an on-going process, so older listings do not reflect this standard). You can download the PBASIC style document from the Parallax web site.

Let's start with some hardware definitions. The BASIC Stamp first:

```
Clock           PIN     0
Ld165           PIN     1
Di165           PIN     2
AdcCS           PIN     3
AdcDo           PIN     4
AdcDi           PIN     5
```

Okay, these are easy. Now let's look at the same pin definitions in the Javelin Stamp:

```
  static final int CLOCK     = CPU.pin0;
  static final int LD_165    = CPU.pin1;
  static final int DI_165    = CPU.pin2;
  static final int ADC_CS    = CPU.pin3;
  static final int ADC_DO    = CPU.pin4;
  static final int ADC_DI    = CPU.pin5;
```

Okay, before you run out the door screaming ... take a deep breath, it's not so hard. What I can tell you is that setting up a program in the Javelin Stamp does require a bit more effort. What you'll find, though, is that as the program grows very large things get easier to deal with in the Javelin Stamp versus the BASIC Stamp, especially since the Javelin Stamp allows us to have 32K of contiguous memory and a bunch more variable space. When we get into reusable class module it's a whole new world.

Let's press on. The keyword "static" is very important as it allows us to define a constant, variable, or method (a Java function or procedure) as part of a class instead of part of an object. What this lets us do is use that constant, variable, or method without declaring an object. An example of this is the CPU class that declares many static values and methods that we'll use in this program. The keyword "final" makes the value a constant. As Java is a strongly-typed language, we use the "int" keyword to specify the size of our definitions. An int is a 16-bit signed value.

Notice that, as I just mentioned, we're setting our constant (final) values to *static* values from the CPU class. The CPU class is specific to the Javelin Stamp (you won't find the CPU class for elsewhere) and its purpose is to define values and methods that are required for embedded control applications – many times mimicking BASIC Stamp functions. The values for I/O pins are not quite as simple as you might imagine, this is due to the requirements of the methods for writing to and reading from a pin. We don't have to remember the pin values though, as they are defined as constants for us.

Let's Get Digital

The bulk of the Stamp PLC's I/O connections are digital: ten digital inputs and eight digital outputs. One thing to note is that both are on separate electrical systems; this is by design for protection. The digital inputs will take anything from 12 to 36 volts, and have their own ground connection (marked Din GND). The reason for the Din GND connection is that the inputs are optically-isolated from the rest of the circuit. That way, if a high-tension line gets across one of our inputs (accidents do happen), we'll lose the opto-isolator but protect the rest of the device. Figure 105.2 shows a schematic of the input isolation. When no input is present the 4.7K pull-up holds the isolated input line high. When we do get an input signal, the transistor side of the opto-isolator will conduct, pulling the isolated input line low. Note that the input signal is actually driving two LEDs: one in the opto-isolator, the second is the visual indicator that lets us know that an input pin is active.

There is actually a second stage between most of the input isolators and the Stamp: a 74HC165 shift register. The 74HC165 allows us to get eight inputs using just three Stamp pins. We've used the 74HC165 before, but for review, we use it like this: "blip" the Load line low momentarily to load the inputs, then use **SHIFTIN** to move them into the Stamp.

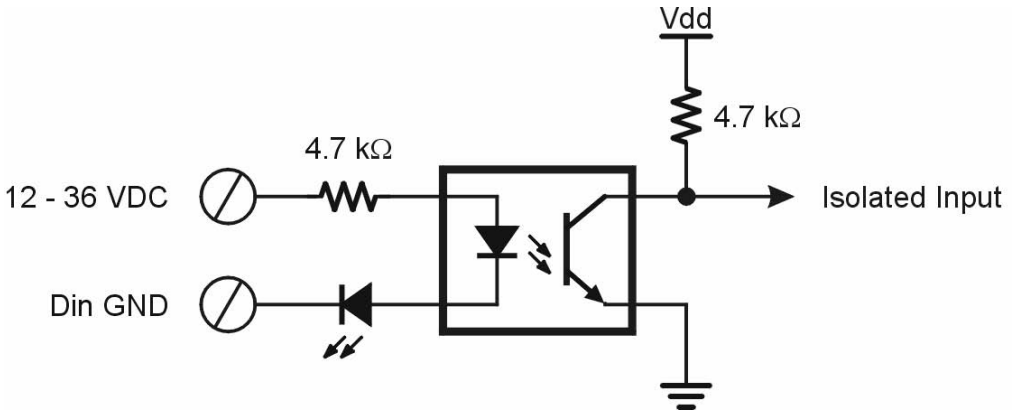

Figure 105.2: StampPLC Input Isolation

The 74HC165 does us another favor: it inverts the inputs – this is great as the opto-isolators invert our Din1 – Din8 inputs. The way this works is by using the inverted serial output line so that we end up getting a "1" bit for active inputs on Din1 – Din8. Figure 105.3 shows the connections for a 74HC165 that uses the inverted data output (QH\) line. The other two Stamp PLC inputs, Din9 and Din10 connect directly to the Stamp after the opto-isolator stage so we have to invert them in software. This is easy for both Stamps.

Okay, here's the code for reading the digital inputs with the BASIC Stamp:

```
Read_DigIns:
  PULSOUT Ld165, 15
  SHIFTIN Di165, Clock, MSBPRE, [dinLo]
  dinHi = 0
  din9  = ~Di9
  din10 = ~Di10
  RETURN
```

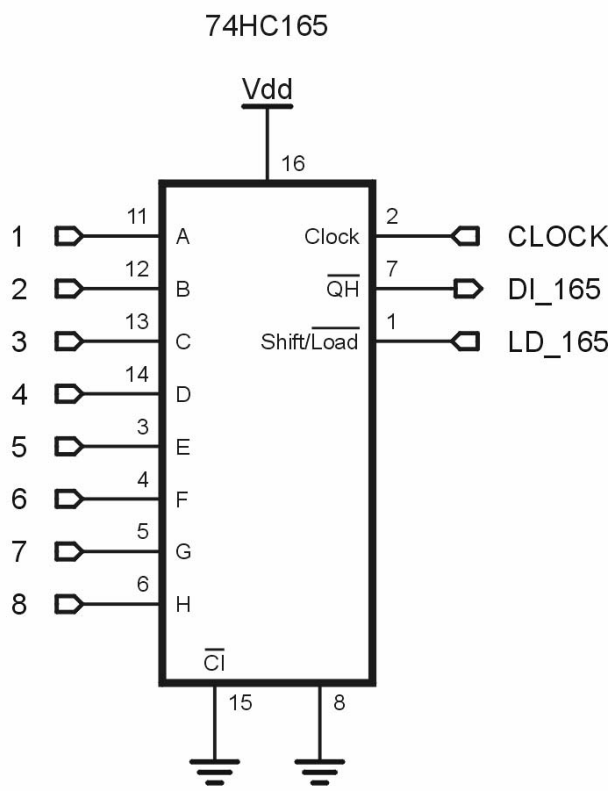

Figure 105.3: Connections for 74HC165 Using Inverted Data Output (QH\) Line

A no-brainer, right? Absolutely. **PULSOUT** is used to load the inputs and **SHIFTIN** moves them into dinLo. Since we're using the inverted output from the 74HC165, a low (zero) on any of its inputs will be clocked in as a high (one) to the Stamp. The next few lines read Din9 and Din10 directly, using the invert operator (~) to correct the input polarity from the opto-isolator stage. At the end of the routine we have a word-sized variable (digIns) that holds the current state of our digital inputs.

Let's have a look at the same code in the Javelin Stamp:

```
static int readDigInputs() {

  int inBits;

  CPU.pulseOut(1, LD_165);
  inBits = CPU.shiftIn(DI_165, CLOCK, 8, CPU.PRE_CLOCK_MSB);

  if (CPU.readPin(DI9) == false) {
    inBits |= 0x100;
  }
  if (CPU.readPin(DI10) == false) {
    inBits |= 0x200;
  }

  return inBits;

}
```

Yes, it looks a bit different but I think that you'll agree that the similarities are, in fact, greater than the differences. The first thing we notice is that our method called readDigInputs() is defined as "static int" – this should make sense based on our earlier discussion. This method is part of our class (not an object) and the result is going to return an integer value to the caller (main part of the program or other methods). Next, we define a local (temporary) variable to hold the inputs before we return them to the caller.

What follows are two lines of code that have a direct correlation to the PBASIC example above. As I stated earlier, the CPU class contains methods for embedded control, many that are duplicates of PBASIC functions. Aside from the basic syntax adjustment, the only item of note is that the CPU.shiftIn() method requires a bit-width value.

The next section reads the direct inputs, albeit indirectly. You see, Java doesn't know anything about hardware like PBASIC does. So the Javelin Stamp provides another method in the CPU class called readPin() that will do this for us. This method will return true if the input is high, false if the input is low. Since our direct inputs are active-low, we will add the appropriate bits to the inBits value.

If you've never programmed in C or Java, this line may look odd:

```
inBits |= 0x100;
```

In Java (as in C) there are occasional programming shortcuts that save us some typing. Here's the long version of that line:

```
inBits = inBits | 0x100;
```

And if we had to do it that way in PBASIC, it would look pretty much the same:

```
inBits = inBits | $0100
```

Once all the bits are collected, we return them to the caller. This comes in handy when we want to look at all the input pins, and is an aid when we want to check the status of just one. Let's have a look at a method for checking a single input:

```
static boolean readDigIn(int digIn) {

  int mask;
  boolean active;

  if (digIn <= DIN10) {
    mask = 0x001 << digIn;
    active = ((readDigInputs() & mask) == mask);
  }
  else {
    active = OFF;
  }

  return active;

}
```

To keep in line with the Javelin Stamp's CPU.readPin() method, readDigIn() will return a value of true when the signal on the input terminal is "high" (hot), and false when there is no signal present. Notice, too, that this method requires a parameter to be passed: the input terminal to check. The code will check this value and if it's out-of-range the method will return false. Assuming a good terminal value, a bit mask is created and compared against the current state of the inputs.

There is no PBASIC version of this method since the BASIC Stamp doesn't pass parameters or return values. How do we do it then? We simply call the Read_DigIns subroutine and look at the aliased bit variable of our choosing.

```
GOSUB Read_DigIns
IF (din7 = IsOn) THEN Go_Do_Something
```

Alright, we're half way through the digital section – more than half way, actually, since the Stamp PLC digital outputs are direct connections and very easy to deal with. Again, let me clarify: when I say direct, I mean that we have a one-for-one control ratio with the output pins; that way when we change an output pin on the Stamp it will directly affect the output. Of course, the Stamp module is protected from the cold cruel world of the industrial environment. The first line of protection is an optical isolation stage similar to Figure 2, then the outputs themselves are driven through high-side drivers.

A high-side driver is designed to control the "hot" side of a circuit with the other side connecting to ground (digital output ground). Since a short circuit could create a real problem for the output stage, the high-side drivers monitor output current and (device) temperature and will shut down an output in the event of a problem.

The code for controlling outputs is as simple as you'd expect. To keep the software simple, output variables use a "1" for on, and "0" for off. This being the case, all we have to do is invert the data and write it to the Stamp port – OUTH (aliased as DOuts) for the BASIC Stamp, PORTB for the Javelin Stamp.

```
Update_DigOuts:
  DOuts = ~digOuts
  RETURN
```

And here's how we do the same thing in the Javelin Stamp:

```
  static void writeDigOutputs(int newOuts) {

    CPU.writePort(CPU.PORTB, (byte)(newOuts ^ 0xFF));

  }
```

The reason that the writeDigOutputs() accepts an integer value is that this will allow the programmer to pass integers or bytes – this makes things easier as the default variable type in the Javelin Stamp is the int. In the CPU.writePort() parameters we end up type-casting out

value to a byte as this is a requirement for the method. In the Javelin Stamp, we invert the bits by using the exclusive-Or operator. Done deal.

In order to take advantage of the Java's parameter passing, let's also create a method for controlling a single output:

```
static boolean writeDigOut(int digOut, boolean newState) {

  boolean status;

  if (digOut <= DOUT8) {
    CPU.writePin(DOUTS[digOut], !newState);
    status = true;
  }
  else {
    status = false;
  }

  return status;

}
```

This method, while simple, is a bit interesting in that it demonstrates how using a return value can be helpful, even though we're executing a command and not looking for a value. The writeDigOut() method returns a Boolean (true or false) to let us know if the output value we passed was valid. To some this will seem like overkill, but when we're creating programs that others will use it's always a good idea to provide helpful ideas like this to protect our friends from themselves!

When a proper value is passed (0 – 7, aliased as DOUT1 – DOUT8), the code will look up the pin value from the DOUTS array and use CPU.writePin() to update the pin. Since the state value is Boolean, the ! (not) operator is used to invert it.

We can't pass parameters in the BASIC Stamp, but what we can do is alias PIN definitions and create useful constants. So here's how we can affect an output when using the BASIC Stamp:

```
Do1 = DirectOn
```

Analog, If You Please

While there are a whole host of industrial applications that are happy to deal with nothing but digital inputs and outputs, the real world often dictates getting involved in analog values. No worries, as I told you earlier we can install a MAX1270 12-bit ADC. The MAX1270 actually has eight channels, but only four are available for us due to the connections involved.

The nice thing about the MAX1270 is that it is software configurable; we can measure 0 to +5 volts, -5 to +5 volts, 0 to +10 volts, or -10 to +10 volts. Pretty cool, huh? The other option that we have with the Stamp PLC is that we can measure 4-20 mA signals. The way this is done is installing a jumper that routes the incoming current through a precision resistor. From the resistor we can read the voltage and determine the level of our signal.

Before we get into reading the channels, let's talk about protecting the analog inputs – a big requirement for getting CE certification on the Stamp PLC. Figure 4 shows the circuitry between an analog connection and the ADC. After the shunt you'll see a 100K, a 1K and a small cap. The 100K helps reduce noise susceptibility of the high-impedance ADC inputs by reducing the input impedance a bit. The 1K limits current to the input and the cap will bypass high-frequency noise. Figure 105.4 shows the analog input protection circuit.

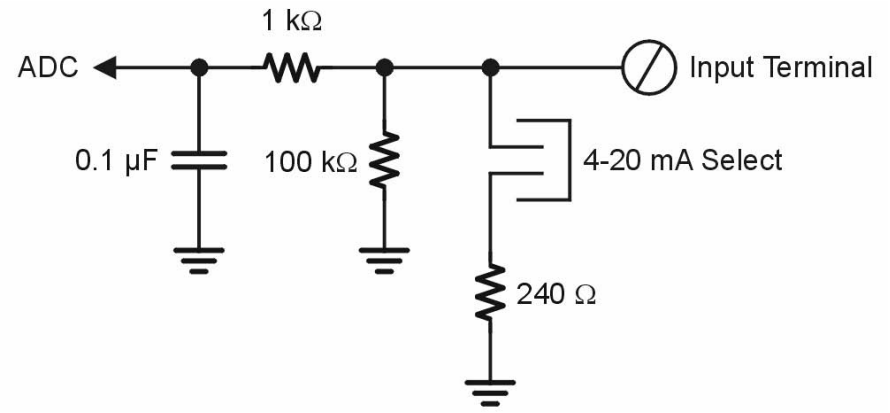

Figure 105.4: StampPLC Analog Input Protection Circuit

The reason for discussing this is that these components will have a small effect on the input signal. I found that when I applied 5.00 volts to an input pin I was reading about 4.75 volts at the ADC. This is no big problem; after we read an ADC channel we'll apply an adjustment factor to scale the input to the "external" signal level.

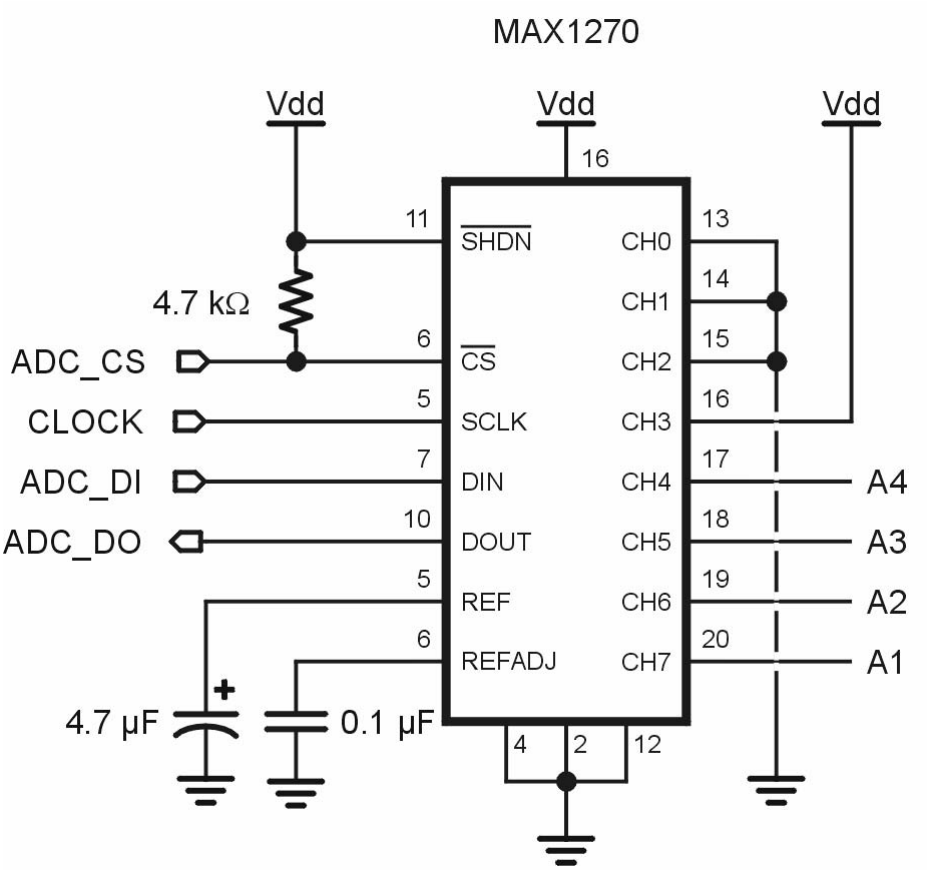

Figure 105.5: StampPLC Connections to MAX1270 ADC

Figure 105.5 shows the Stamp PLC connections to the MAX1270 ADC. As you can see, we're only using four of the eight analog channels; the others are tied to ground or Vdd (this is useful for device verification). The MAX1270, like the 74HC165 is a synchronous serial device. A key difference is that the MAX1270 uses a chip select line so that its data lines can be shared with other devices. The 74HC165 has no chip select, so the only common line between the two is the clock.

Accessing analog data is pretty easy: We select the MAX1270 by taking the CS\ line low, use **SHIFTOUT** to send the configuration byte, the release the device. It takes less than a millisecond for the conversion to take place, but we do need to insert a very short delay to make sure this happens. After that we reselect the device and use **SHIFTIN** to retrieve the 12-bit value.

The configuration byte tells the MAX1270 what range to use and what channel to read. To make things easier, possible configuration bytes are predefined as **DATA** values in the BS2, and in a constant array in the Javelin Stamp. Both versions of the program allow us to set the mode (voltage range) and channel so that the configuration byte can be retrieved.

Let's look at the BASIC Stamp code for reading a channel:

```
Read_ADC:
  READ AdcCfg + (mode * 4 + chan), config
  LOW AdcCS
  SHIFTOUT AdcDo, Clock, MSBFIRST, [config]
  HIGH AdcCS
  adcRaw = 0
  LOW AdcCS
  SHIFTIN AdcDi, Clock, MSBPRE, [adcRaw\12]
  HIGH AdcCS

  adcRaw = adcRaw + (adcRaw ** $D6C) MAX 4095   ' x ~1.05243
```

The program uses to variables, mode and chan, to calculate the table pointer for the configuration byte. The rest of the code should make perfect sense. Remember that **SHIFTIN** and **SHIFTOUT** default to eight bits, so we have to use the \12 modifier with **SHIFTIN** to retrieve all the bits from the MAX1270.

The final line of code above is the adjustment for the input protection circuit. As you can see by the comment, we're multiplying the raw ADC value by 1.05243. In order to get the best resolution, I used my buddy Tracy Allen's trick with the ** (star-star) operator. Star-star, as you'll recall, allows us to multiply values in increments of 1/65536.

Okay, now let's look at the same code in the Javelin Stamp:

```
static int readAnalog(int channel, int mode) {

  int config;
  int adcRaw = 0;

  config = ADC_CFG[mode * 4 + channel];
  CPU.writePin(ADC_CS, false);
  CPU.shiftOut(ADC_DO, CLOCK, 8, CPU.SHIFT_MSB, (config << 8));
  CPU.writePin(ADC_CS, true);
  CPU.delay(1);
  CPU.writePin(ADC_CS, false);
  adcRaw = CPU.shiftIn(ADC_DI, CLOCK, 12, CPU.PRE_CLOCK_MSB);
  CPU.writePin(ADC_CS, true);

  // adjust counts for input divider (protection) circuity
  // -- cal factor = (input on terminal) / (input to MAX1270)
  // -- 1.05243

  adcRaw = adcRaw + (adcRaw / 20) + (adcRaw / 412);

  return adcRaw;

}
```

The bulk of the code is not dramatically different, except that we can pass the mode and channel values as parameters. On thing you'll notice is that the config byte is shifted left by eight in the CPU.shiftOut() method. The reason for this is that the Javelin Stamp uses integers internally and since we're shifting the MSB first, we have to move the byte MSB (bit 7) to an int MSB (bit 15).

Now, there is no equivalent to the ** operator in the Javelin Stamp, and I wanted to keep the program simple and not involve any classes beyond CPU (Note: There is a simple floating point math class available for the Javelin Stamp). What this means, then, is that I had to do a little math to create the adjustment line. This was actually very easy. What I did is split the fractional portion into two parts: 0.05 and 0.00243. Then I entered 0.05 into my calculator

and pressed the reciprocal (1/x) key – that gave me 20. Then I did the same thing with 0.00243 and got 412.

We Are Ready for Timely Control

Up to this point we've created I/O control routines for a generic controller. What do we do now? Well, anything we need or want to do. What I'd like to show you, though, is why the Javelin Stamp can be such a great little controller industrial applications.

The Javelin Stamp has a timer object that runs in the background and can actually control as many timers as we might need in our program. Once we've defined a timer, it simply increments its counts until we reset it with the mark() method. The simplest way to use a Javelin timer is to check its count with the timeOut() method. Let's look at a simplified Javelin shell for a time-specific control application.

```
public static void main() {

  Timer scanTimer = new Timer();
  scanTimer.mark();

  while (true) {

    if (scanTimer.timeout(100)) {
      scanTimer.mark();

      // control logic here

    }
  }
}
```

In this shell a timer is created and immediately reset. With the while loop (which will run forever) the timer is checked to see if it has reached 100 milliseconds. When that condition is true, the timer will be reset and the code that is intended to run every 100 milliseconds will execute. This is a very simple, yet incredibly powerful feature of the Javelin Stamp. And it's tough to duplicate on the BASIC Stamp with **PAUSE**, since any change in our operation code means we have to adjust the **PAUSE** duration.

The full listings include more demo code to study and use, so be sure to download them from Nuts & Volts or the Parallax web site.

Whew....

Okay, I know that for some of you, your heads are spinning – imagine how I feel having spent three solid days of my life writing this! I recently said to a friend that it is much easier to write a program than to write *about* a program. I hope, though, that you found this article valuable, especially if you've been a bit shy about trying the Javelin Stamp. It has taken a little time to catch on, probably because Java is a more disciplined language than PBASIC, and takes some getting used to. That aid, we're seeing more and more BASIC Stamp users give the Javelin Stamp a try and doing some really neat things.

The good news is that Java is growing as a language and is being taught in more and more schools. And remember that you can always run down to your favorite book store and find shelves full of books on programming in Java. While it's true that a lot of the programs in those books won't run on the Javelin Stamp (there is no GUI in an embedded micro and other desktop computer stuff is irrelevant), the concepts and programming strategies are all valid. My favorite book on Java is called "Head First Java" by Kathy Sierra and Bert Bates (ISBN 0596004656). It's the first technical book I ever purchased that is actually fun to work with.

Let me finish by saying that this in my Javelin Stamp PLC program I actually kind of violated the Java "norm" by not breaking my program into multiple classes. The reason is that I wanted to create a project that was easy to use for programmers without much Java experience. As my own Java expertise grows, I am doing that and you can download a modular (with reusable classes for the 74HC165 and MAX1270) version of the Stamp PLC from the Parallax web site.

Okay, I've had enough; have you? I thought so. Until next time, then, Happy Stamping.

```
' =============================================================================
'
'   File....... StampPLC.BS2
'   Purpose.... Stamp PLC Core Routines and Framework for Apps
'   Author..... Jon Williams, Applications Engineer
'               (Copyright 2003 - All Rights Reserved)
'   E-mail..... support@parallax.com
'   Started....
'   Updated.... 05 NOV 2003
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =============================================================================


' -----[ Program Description ]---------------------------------------------
'
' This program provides a set of core routines that can be used for
' creating operational programs for the Stamp PLC.  Conditional compilation
' is used so that the code can run on any 24-pin BASIC Stamp 2 module.
'
' Notes on reading ADC channels:
'
' The inputs are protected and reduce the voltage felt on the MAX1270 input
' pins.  This is accounts for code to get a full-scale count of 4095.
'
' The raw value (counts) from the ADC will be returned in "adcRaw" and
' converted to millivolts and returned in "mVolts."  Be aware that in
' bipolar modes the value of "mVolts" is signed.  A "1" in BIT15 of "mVolts"
' indicates a negative value.  The BASIC Stamp does not support division or
' mulitplication on negative values.


' -----[ Revision History ]------------------------------------------------
'


' -----[ I/O Definitions ]-------------------------------------------------

Clock           PIN     0                       ' shared clock
Ld165           PIN     1                       ' 74HC165 load
Di165           PIN     2                       ' 74HC165 data in (from)
AdcCS           PIN     3                       ' ADC chip select
AdcDo           PIN     4                       ' ADC data out (to)
AdcDi           PIN     5                       ' ADC data in (from)

Di9             PIN     6                       ' direct digital inputs
Di10            PIN     7
```

```
DOuts           VAR     OUTH                    ' direct digital outputs
DOutsLo         VAR     OUTC                    ' -- Do1 - Do4
DoutsHi         VAR     OUTD                    ' -- Do5 - Do8
Do1             PIN     8
Do2             PIN     9
Do3             PIN     10
Do4             PIN     11
Do5             PIN     12
Do6             PIN     13
Do7             PIN     14
Do8             PIN     15

Sio             CON     16                      ' serial IO (prog port)


' -----[ Constants ]-------------------------------------------------------

IsOn            CON     1                       ' for shadow regs
IsOff           CON     0

DirectOn        CON     0                       ' for direct IO pins only
DirectOff       CON     1

#SELECT $STAMP
  #CASE BS2, BS2E, BS2PE
    T1200       CON     813                     ' for programming port
    T2400       CON     396
    T9600       CON     84
    T19200      CON     32
  #CASE BS2SX, BS2P
    T1200       CON     2063
    T2400       CON     1021
    T9600       CON     240
    T19200      CON     110
#ENDSELECT

Baud            CON     T9600                   ' default (matches DEBUG)

Ain1            CON     0                       ' analog channels
Ain2            CON     1
Ain3            CON     2
Ain4            CON     3

AdcUP5          CON     0                       ' unipolar, 0 - 5 v
AdcBP5          CON     1                       ' bipolar, +/- 5 v
AdcUP10         CON     2                       ' unipolar, 0 - 10 v
AdcBP10         CON     3                       ' bipolar, +/- 10 v
Adc420          CON     4                       ' 4-20 mA input


' -----[ Variables ]-------------------------------------------------------
```

```
digIns          VAR     Word                    ' shadow digital inputs
dinLo           VAR     digIns.LOWBYTE          ' Din1 - Din8
dinHi           VAR     digIns.HIGHBYTE         ' Din9 - Din10
din1            VAR     digIns.BIT0
din2            VAR     digIns.BIT1
din3            VAR     digIns.BIT2
din4            VAR     digIns.BIT3
din5            VAR     digIns.BIT4
din6            VAR     digIns.BIT5
din7            VAR     digIns.BIT6
din8            VAR     digIns.BIT7
din9            VAR     digIns.BIT8
din10           VAR     digIns.BIT9

digOuts         VAR     Byte                    ' shadow digital outputs
dout1           VAR     digOuts.BIT0
dout2           VAR     digOuts.BIT1
dout3           VAR     digOuts.BIT2
dout4           VAR     digOuts.BIT3
dout5           VAR     digOuts.BIT4
dout6           VAR     digOuts.BIT5
dout7           VAR     digOuts.BIT6
dout8           VAR     digOuts.BIT7

chan            VAR     Nib                     ' ADC channel (0 - 3)
mode            VAR     Nib                     ' ADC mode (0 - 4)
config          VAR     Byte                    ' configuration byte
adcRes          VAR     Nib                     ' ADC bits (1 - 12)
adcRaw          VAR     Word                    ' ADC result (raw)
mVolts          VAR     Word                    ' ADC in millivolts


' -----[ EEPROM Data ]-----------------------------------------------------

Project         DATA    "Stamp PLC Template", 0

AdcCfg          DATA    %11110000, %11100000, %11010000, %11000000 ' 0-5
                DATA    %11110100, %11100100, %11010100, %11000100 ' +/-5
                DATA    %11111000, %11101000, %11011000, %11001000 ' 0-10
                DATA    %11111100, %11101100, %11011100, %11001100 ' +/-10
                DATA    %11110000, %11100000, %11010000, %11000000 ' 4-20


' -----[ Initialization ]--------------------------------------------------

Setup:
  LOW Clock                                     ' preset control lines
  HIGH Ld165
  HIGH AdcCS
```

```
  DOuts = %11111111                             ' all outputs off
  DIRH = %11111111                              ' enable output drivers

  adcRes = 12                                   ' use all ADC bits


' -----[ Program Code ]----------------------------------------------------

Main:

  ' demo - replace with your code
  '
  GOSUB Read_DigIns
  DEBUG HOME, "Inputs = ", BIN10 digIns, CR, CR

  ' copy inputs to outputs
  ' -- Din9  --> Dout1
  ' -- Din10 --> Dout2
  '
  digOuts = digIns
  GOSUB Update_DigOuts
  IF (din9 = IsOn) THEN Do1 = DirectOn
  IF (din10 = IsOn) THEN Do2 = DirectOn

  ' read single-ended analog inputs
  ' -- display input as millivolts
  '
  mode = AdcUP5
  FOR chan = Ain1 TO Ain4
    GOSUB Read_ADC
    DEBUG "Ain", ("1" + chan), ".... ",
          DEC (mVolts / 1000), ".", DEC3 mVolts, CR
  NEXT

  GOTO Main
  '
  ' end of demo code

  END


' -----[ Subroutines ]------------------------------------------------------

' Scans and saves digital inputs, DIn1 - DIn10
' -- returns inputs in "digIns" (1 = input active)

Read_DigIns:
  PULSOUT Ld165, 15                             ' load inputs
  SHIFTIN Di165, Clock, MSBPRE, [dinLo]         ' shift in
  dinHi = 0                                     ' clear upper bits
  dinHi.BIT0 = ~Di9                             ' grab DIN9
```

```
  dinHi.BIT1 = ~Di10                              ' grab DIN10
  RETURN


' Refreshes digital outputs, DOut1 - DOut8
' -- uses shadow value "digOuts" (1 = output on)

Update_DigOuts:
  DOuts = ~digOuts                                ' refresh outputs
  RETURN


' Reads analog input channel (0 - 5 vdc)
' -- put channel (0 - 3) in "chan"
' -- pass mode (0 - 4) in "mode")
' -- raw value returned in "adcRaw"
' -- "adcRaw" converted to signed "mVolts"

Read_ADC:
  READ AdcCfg + (mode * 4 + chan), config       ' get config
  LOW AdcCS                                     ' select MAX1270
  SHIFTOUT AdcDo, Clock, MSBFIRST, [config]     ' send config byte
  HIGH AdcCS                                    ' deselect MAX1270
  adcRaw = 0
  LOW AdcCS
  SHIFTIN AdcDi, Clock, MSBPRE, [adcRaw\12]     ' read channel value
  HIGH AdcCS

  ' adjust ADC count for input voltage divider
  '
  adcRaw = adcRaw + (adcRaw ** $D6C) MAX 4095   ' x ~1.05243


  ' millivolts conversion
  ' -- returns signed value in bipolar modes
  ' -- uses raw (12-bit) value
  '
  SELECT mode
    CASE AdcUP5
      mVolts = adcRaw + (adcRaw ** $3880)       ' x 1.2207

    CASE AdcBP5
      IF (adcRaw < 2048) THEN
        mVolts = 2 * adcRaw + (adcRaw ** $7100) ' x 2.4414
      ELSE
        adcRaw = 4095 - adcRaw
        mVolts = -(2 * adcRaw + (adcRaw ** $7100))
      ENDIF

    CASE AdcUP10
      mVolts = 2 * adcRaw + (adcRaw ** $7100)   ' x 2.4414
```

```
  CASE AdcBP10
    IF (adcRaw < 2048) THEN
      mVolts = 4 * adcRaw + (adcRaw ** $E1FF)
    ELSE
      adcRaw = 4095 - adcRaw
      mVolts = -(4 * adcRaw + (adcRaw ** $E1FF))
    ENDIF

  CASE Adc420                                   ' -- 4000 to 20000
    mVolts = 5 * adcRaw + (adcRaw ** $1666)     ' x 5.0875

ENDSELECT

' adjust adcRaw for selected resolution
'
IF (adcRes < 12) THEN
  adcRaw = adcRaw >> (12 - adcRes)              ' reduce resolution
ENDIF

RETURN
```

```
import stamp.core.*;

/**
 * Stamp PLC template
 *
 * @version 0.3 -- 11 NOV 2003
 * @author Jon Williams, Parallax (jwilliams@parallax.com)
 */

public class StampPLC {

  // ------------------------------------------------------------------------
  // Stamp PLC constants
  // ------------------------------------------------------------------------

  static final int CLOCK     = CPU.pin0;      // shared clock
  static final int LD_165    = CPU.pin1;      // 74HC165 shift/load
  static final int DI_165    = CPU.pin2;      // 74HC165 data in (from)
  static final int ADC_CS    = CPU.pin3;      // ADC chip select
  static final int ADC_DO    = CPU.pin4;      // ADC data out (to)
  static final int ADC_DI    = CPU.pin5;      // ADC data in (from)

  static final int DI9       = CPU.pin6;      // direct digital inputs
  static final int DI10      = CPU.pin7;

  static final int DO1       = CPU.pin8;      // direct digital outputs
  static final int DO2       = CPU.pin9;
  static final int DO3       = CPU.pin10;
  static final int DO4       = CPU.pin11;
  static final int DO5       = CPU.pin12;
  static final int DO6       = CPU.pin13;
  static final int DO7       = CPU.pin14;
  static final int DO8       = CPU.pin15;

  static final int DOUTS[]   = { DO1, DO2, DO3, DO4, DO5, DO6, DO7, DO8 };

  static final int DIN1      = 0;             // for readDigIn() method
  static final int DIN2      = 1;
  static final int DIN3      = 2;
  static final int DIN4      = 3;
  static final int DIN5      = 4;
  static final int DIN6      = 5;
  static final int DIN7      = 6;
  static final int DIN8      = 7;
  static final int DIN9      = 8;
  static final int DIN10     = 9;

  static final int DOUT1     = 0;             // for writeDigOut() method
  static final int DOUT2     = 1;
  static final int DOUT3     = 2;
  static final int DOUT4     = 3;
```

```
static final int DOUT5     = 4;
static final int DOUT6     = 5;
static final int DOUT7     = 6;
static final int DOUT8     = 7;

static final boolean ON    = true;          // for IO methods
static final boolean OFF   = false;

static final int AIN1      = 0;             // for readAnalog() method
static final int AIN2      = 1;
static final int AIN3      = 2;
static final int AIN4      = 3;

static final int ADC_UP5   = 0;             // unipolar, 0-5 volts
static final int ADC_BP5   = 1;             // bipolar, +/- 5 volts
static final int ADC_BP10  = 2;             // bipolar, +/- 10 volts
static final int ADC_UP10  = 3;             // unipolar, 0-10 volts
static final int ADC_420   = 4;             // 4-20 mA input

static final int ADC_CFG[] = { 0xF0, 0xE0, 0xD0, 0xC0,
                               0xF4, 0xE4, 0xD4, 0xC4,
                               0xF8, 0xE8, 0xD8, 0xC8,
                               0xFC, 0xEC, 0xDC, 0xCC,
                               0xF0, 0xE0, 0xD0, 0xC0 };

// ------------------------------------------------------------------------
// IDE Terminal constants
// ------------------------------------------------------------------------

final static char HOME      = 0x01;


// ------------------------------------------------------------------------
// Stamp PLC methods
// ------------------------------------------------------------------------


// Initialize PLC
// -- setup internal hardware connections
// -- force digital outputs off

static void initPLC() {

  CPU.writePin(CLOCK, false);               // clock = output low
  CPU.writePin(LD_165, true);               // load = output high
  CPU.writePin(ADC_CS, true);               // adc select = output high
  CPU.writePin(ADC_DO, false);              // prep for high pulses
  CPU.writePin(CPU.PORTB, true);            // digital outputs off

}
```

```
// Read digital inputs
// -- returns 10-bit value
// -- "1" = input active

static int readDigInputs() {

  int inBits;

  // read indirect inputs
  CPU.pulseOut(1, LD_165);
  inBits = CPU.shiftIn(DI_165, CLOCK, 8, CPU.PRE_CLOCK_MSB);

  // read direct (active-low) inputs
  if (CPU.readPin(DI9) == false) {
    inBits |= 0x100;
  }
  if (CPU.readPin(DI10) == false) {
    inBits |= 0x200;
  }

  return inBits;

}


// Read single digital input (0 - 9)
// -- returns ON (true) if active

static boolean readDigIn(int digIn) {

  int mask;
  boolean active;

  if (digIn <= DIN10) {
    mask = 0x001 << digIn;
    active = ((readDigInputs() & mask) == mask);
  }
  else {
    active = OFF;                          // digIn out of range
  }

  return active;

}


// Write new value to digital outputs
// -- updates all digital outputs (Dout1 - Dout8)
// -- Note: outputs are active-low
```

```
static void writeDigOutputs(int newOuts) {

  CPU.writePort(CPU.PORTB, (byte)(newOuts ^ 0xFF));

}


// Write new status to single output (0 - 7)
// -- pass ON in newState to activate output; OFF to deactivate
// -- Note: outputs are active-low
// -- returns false if digOut was out of range

static boolean writeDigOut(int digOut, boolean newState) {

  boolean status;

  if (digOut <= DOUT8) {
    CPU.writePin(DOUTS[digOut], !newState);
    status = true;
  }
  else {
    status = false;                          // digOut out of range
  }

  return status;

}


// Read analog channel
// -- pass channel and mode
// -- returns raw ADC count

static int readAnalog(int channel, int mode) {

  int config;
  int adcRaw = 0;

  config = ADC_CFG[mode * 4 + channel];
  CPU.writePin(ADC_CS, false);
  CPU.shiftOut(ADC_DO, CLOCK, 8, CPU.SHIFT_MSB, (config << 8));
  CPU.writePin(ADC_CS, true);
  CPU.delay(1);
  CPU.writePin(ADC_CS, false);
  adcRaw = CPU.shiftIn(ADC_DI, CLOCK, 12, CPU.PRE_CLOCK_MSB);
  CPU.writePin(ADC_CS, true);

  // adjust counts for input divider (protection) circuity
  // -- cal factor = (input on terminal) / (input to MAX1270)
  // -- 1.05243
```

```
  adcRaw = adcRaw + (adcRaw / 20) + (adcRaw / 412);
  // limit to 4095
  adcRaw = (adcRaw <= 4095) ? adcRaw : 4095;

  return adcRaw;

}


// Converts ADC counts to signed millivolts (based on mode)

static int millivolts(int counts, int mode) {

  int c, mV = 0;

  c = counts;

  switch (mode) {

    case ADC_UP5:
      // x 1.2207
      mV = c + (c / 5) + (c / 50) + (c / 1429);
      break;

    case ADC_BP5:
      if (counts < 2048) {
        // x 2.4414
        mV = 2 * c + (2 * c / 5) + (c / 25) + (c / 714);
      }
      else {
        c = 4095 - c;
        mV = 0 - (2 * c + (2 * c / 5) + (c / 25) + (c / 714));
      }
      break;

    case ADC_UP10:
      // x 2.4414
      mV = 2 * c + (2 * c / 5) + (c / 25) + (c / 714);
      break;

    case ADC_BP10:
      if (counts < 2048) {
        // x 4.8828
        mV = 4 * c + (4 * c / 5) + (2 * c / 25) + (c / 357);
      }
      else {
        c = 4095 - c;
        mV = 0 - (4 * c + (4 * c / 5) + (2 * c / 25) + (c / 357));
      }
      break;
```

```
    case ADC_420:
      // returns 4000 - 20000
      // x 5.0875
      mV = 5 * c + (2 * c / 25) + (c / 133);
      break;
  }

  return mV;

}


// ------------------------------------------------------------------------
// Put your custom methods here
// ------------------------------------------------------------------------




// ------------------------------------------------------------------------
// Stamp PLC program
// ------------------------------------------------------------------------

public static void main() {

  int inputs;                              // input scan
  int outputs;                             // data for outputs
  int adc;                                 // adc count
  int mV;                                  // adc in millivolts

  StringBuffer msg = new StringBuffer(128); // buffer for messages
  Timer scanTimer = new Timer();           // timer for scans


  // program code

  initPLC();
  scanTimer.mark();                        // reset timer


  // ***************************************************************
  // *                                                             *
  // *   Replace the demo code below with your application code.   *
  // *                                                             *
  // ***************************************************************


  // main scan
  while (true) {

    if (scanTimer.timeout(100)) {          // scan every 100 ms
```

```
        scanTimer.mark();                     // reset timer

        msg.clear();
        msg.append(HOME);

        msg.append("Inputs = ");
        inputs = readDigInputs();             // collect ins
        writeDigOutputs(inputs);              // copy ins to outs

        // cause DOUT1 to follow DIN9
        writeDigOut(DOUT1, ((inputs & 0x100) == 0x100));
        // cause DOUT2 to follow DIN10
        writeDigOut(DOUT2, ((inputs & 0x200) == 0x200));

        // create binary image of digital inputs
        for (int mask = 0x200; mask > 0; mask >>= 1) {
          if ((mask & inputs) == 0)
            msg.append('0');
          else
            msg.append('1');
        }
        msg.append("\n\n");

        // read and display four single-ended analog channels
        for (int chan = AIN1; chan <= AIN4; chan++) {
          msg.append("Ain");
          msg.append(chan + 1);
          msg.append(" = ");
          adc = readAnalog(chan, ADC_UP5);
          mV = millivolts(adc, ADC_UP5);
          msg.append(mV / 1000);
          msg.append(".");
          msg.append(mV % 1000);
          msg.append("   \n");
        }
        System.out.print(msg.toString());
      }
    }
  }
}
```