Column #113 September 2004 by Jon Williams:

No More Coffee Spills

Ideas more crazy than this have become successful products!

Have you ever noticed yourself when you've noticed yourself? I seem to do it all the time, suddenly noticing some behavior that had – until that very moment – been completely unconscious. Often times the thing I notice is quite humorous. Like most humans, I'm a creature of habit. My typical daily habit is to wake, check e-mail quickly (in case there's been an overnight customer emergency), jump on the treadmill for about 30 minutes, grab a bite to eat, and then pop by the neighborhood Starbucks for a cup of coffee and a quick chat with a very pleasant lady called Lindsay.

On a recent return trip from Starbucks I noticed myself doing something that actually made me laugh. While steering with my left had, I would reach down with my right hand to adjust the coffee spout such that coffee would not slosh out while cornering. Honestly, I laughed out loud, then immediately thought that if I had an accelerometer, a stepper motor, and a BASIC Stamp, I could keep both hands on the wheel. An idea is born....

A practical idea? Well, probably not, and I'm certainly not going to tear apart the center console on my new SUV to install such a device – but the exercise in designing the circuitry

and code to solve my coffee sloshing problem is still worthwhile. I receive a lot of e-mail asking how one gets "so good" at programming the BASIC Stamp microcontroller. Like getting to Carnegie Hall, it takes practice, practice, practice. Imagine how many tens of thousands of practice shots Michael Jordan shot before and during his career; and every one of them served to prepare him for all those championships. I guess my point is not to wait for a real project to improve your programming skills. Many times it's worth doing a project just for the experience of doing it.

Leaning To And Fro

If you're new to the BASIC Stamp microcontroller, or weren't around for my article on using GPS last November, you may be wondering how we're going to take the output from an accelerometer and use it to point the spout in our coffee-cup lid. To be honest, it's dirt simple: we're going to use the ATN function. ATN (arctangent) returns the angle (in binary radians; 0 to 255) that points to the intersection of two vector values.

The first thing to do, then, is to read the accelerometer outputs (x and y axis) to establish the g-force vectors. You may remember from our previous work with the Memsic 2125 that a 0g output is a five millisecond pulse. Negative g-forces are shorter than five milliseconds; positive g-forces are longer. Reading the pulse outputs is no trouble -- we can use PULSIN to do it. But here's the trick: the resolution of PULSIN gets better as BASIC Stamp modules get faster. Code that will work properly with a BS2 module will not return the correct results when using the BS2p.

You know where I'm going with this: conditional compilation. I covered it briefly in the past, and I think it's good to remind ourselves that this feature is now available in the Version 2.1 compiler. So what are we going to do with it? We're going to use a conditional compilation construct to set a constant value that will be used to scale the raw input from PULSIN to microseconds, always returning the correct value regardless of which BASIC Stamp module is being used.

```
#SELECT $STAMP
  #CASE BS2, BS2E
    Scale       CON     $200
  #CASE BS2SX
    Scale       CON     $0CC
  #CASE BS2P
    Scale       CON     $0C0
  #CASE BS2PE
    Scale       CON     $1E1
#ENDSELECT
```

When we compile a program the first thing that the compiler does is look for conditional-compilation symbols (created with #DEFINE), and then conditional-compilation constructs like ours above. The construct works as we'd expect, but only at compile time. The section that evaluates as true will get compiled into the program; all others will be ignored. If, for example, that we had select a BASIC Stamp 2 module, our program would compile and assign the constant called Scale a value of $200 (same as 2.0 decimal when used with */).

Once you get used to conditional compilation it can be a very powerful tool. How many times have you sprinkled DEBUG statements through a program, only having to go rip them all out when everything is working? With conditional compilation, you can do this:

```
#DEFINE DebugMode = 1
```

Then, in the body of the program we add a bit of logic around our DEBUG outputs:

```
#IF DebugMode #THEN
  DEBUG "Program Status"
#ENDIF
```

Once the program is working properly we can turn-off all the DEBUG statements with one small change:

```
#DEFINE DebugMode = 0
```

How much easier is this? And, as we've all seen, programs are ultimately updated. When that happens, we don't have to go back through and drop in DEBUG statements to check on our updates – we simply re-enable the ones that are there with #DEFINE.

Okay, let's get back to reading the accelerometer (connections are shown in Figure 113.1). The first thing to do is scale the raw input from it to a known value. By using our Scale constant, we will end up with microseconds – and at 0g on either axis, we should get around 5000 (five milliseconds). Remember that the */ operator works like multiplication, with the parameter being expressed in units of 1/256. This lets us multiply by fractional values between 0 and 255.996.

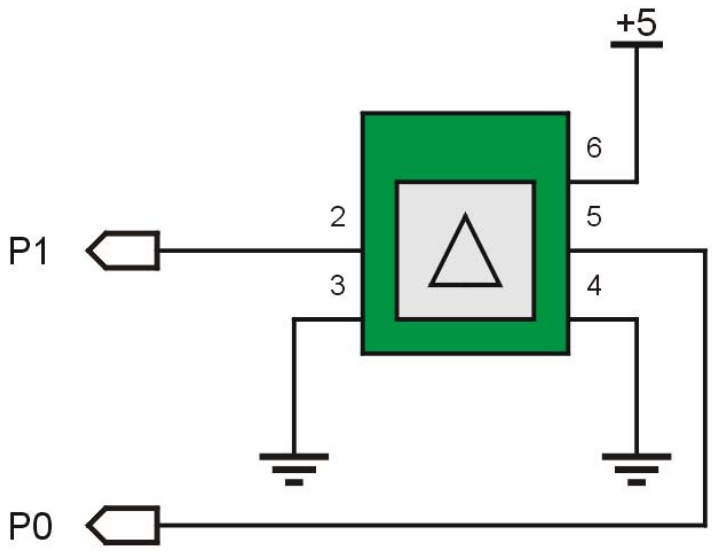

Figure 113.1: Memsic 2125 Schematic

```
Get_Memsic:
  PULSIN XPin, 1, xG
  PULSIN YPin, 1, yG
  xG = xG */ Scale
  yG = yG */ Scale
  RETURN
```

If we run this code in a loop and look at the results we'll see values that range from 3800 to 6200 microseconds. Okay, but what we need is something that fits into the -127 to 127 range required by ATN. It's actually very easy, and uses the "round up" math we learned as kids in math class.

```
Get_Memsic:
  PULSIN XPin, 1, xG
  PULSIN YPin, 1, yG
  xG = xG */ Scale + 50 / 50 - 100
  yG = yG */ Scale + 50 / 50 - 100
  RETURN
```

Right after Scale you'll see "+ 50 / 100" – this rounds up the value and scales it down to 50 as the 0g point. Then we subtract 50 so that 0g returns a zero value. There's an important note here: we need to do the subtraction last, because if we attempt to divide a negative number we will get incorrect results.

If we look at the output now (using the SDEC modifier, of course) we should see values from -12 to +12. Don't be worried about the apparent loss of resolution of our vector. The reason we've divided down so much is to eliminate minor jitter from the sensor that would just cause the stepper to quiver back and forth unnecessarily. What we've done, in fact, is a bit of simple digital filtering.

Okay, now that we can read the output of the accelerometer as vectors that will work with ATN, let's map the output of ATN vis-à-vis the g-force loading on the sensor. Figure 113.2 shows the values I read from my setup. We can simulate acceleration of the car by tilting the front end of the sensor (pins 1 and 6) up. We can simulate braking by tilting the front end of the sensor down.

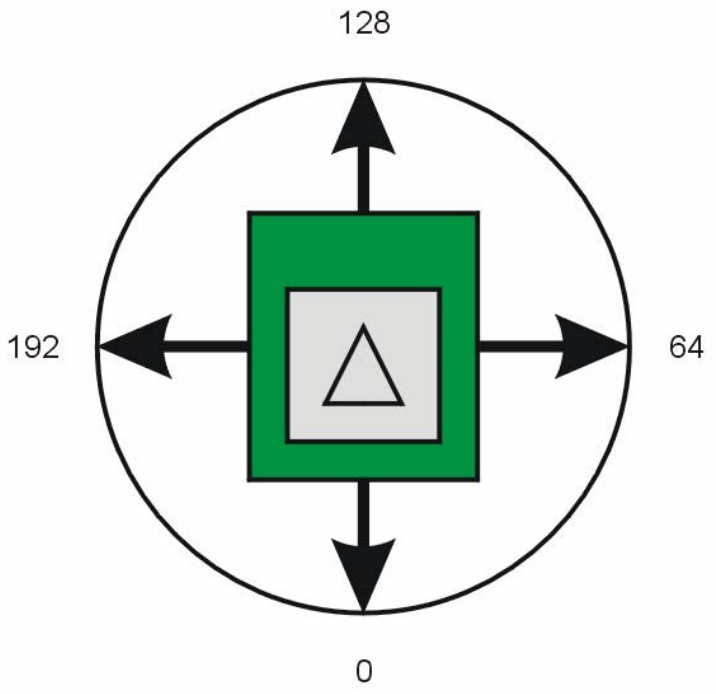

Figure 113.2: Tilting Sensor Simulates G-force

```
Main:
  GOSUB Get_Memsic
  DEBUG HOME,
        SDEC xG, CLREOL, CR,
        SDEC yG, CLREOL, CR,
        CR,
        DEC (xG ATN yG), CLREOL
  PAUSE 200
  GOTO Main
```

So under simulated acceleration we get zero; under breaking, 128; turning right, 192; and turning left, 64. Let's think about this for a moment. Wouldn't it be a bit more convenient if the output matched the number of steps of our stepper motor? And it might be easier to deal with mentally if the values increased in a clockwise direction as well.

I happened to grab a stepper that has a 3.6 degree step – this means that there are 100 steps in each revolution. Here's how we can do it:

```
  angle = 100 - ((xG ATN yG) */ 100) // 100
```

This may look a bit convoluted but I promise there is a reason for every bit of it – let's work from the inside out. By using */ (star-slash) with a parameter of 100 we scale the output of ATN from 0 – 255 to 0 – 99. Our next issue, then, is reversing the direction so that values increase going clockwise instead of counter-clockwise as they do now. By subtracting from 100 we're able to do that – with one tiny glitch: what was zero ends up being 100. Okay, then, we can use the modulus operator on the whole works and now our output is 0 to 99. Accelerating means we should point the coffee spout toward zero (front of car); when breaking, 50 (rear of car); turning right, 25; turning left, 75. Let's polish off our main loop:

```
Main:
  GOSUB Get_Memsic
  angle = 100 - ((xG ATN yG) */ 100) // 100
  IF (angle <> pos) THEN
    GOSUB Move_Spout
  ENDIF
  GOTO Main

  END
```

This is as simple as it gets: we read the new angle and compare it to the current spout position. If there's a difference then we reposition the spout, otherwise we go read the accelerometer again.

Before we can actually turn the spout we need to find the shortest path between the current position (pos) and the new position (at angle). This one actually took me a few minutes to figure out how to code cleanly. While it looks simple on paper with circles and dots, it's takes a bit of thinking. Let's look at the code that determines rotational direction and work through it.

```
Move_Spout:
  IF (angle > pos) THEN
    span = angle - pos
    IF (span < HalfRev) THEN
      stpDir = MCW
    ELSE
      stpDir = MCCW
      span = RevSteps - span
    ENDIF
  ELSE
    span = pos - angle
    IF (span > HalfRev) THEN
      stpDir = MCW
      span = RevSteps - span
    ELSE
      stpDir = MCCW
    ENDIF
  ENDIF
```

Here we go: if the new angle is greater than the current position then we get the span by subtracting the old position from the new. Next we check to see if that span is less than half a revolution. If it is then we set the stepper direction to clockwise, otherwise it will rotate counterclockwise. Notice that we have to make a correction in the span if we determine the shortest path is counter-clockwise. If we don’t' don't make this correction, the spout will overshoot the desired position. We can use a similar set of logic when the new angle is less than the current position, with adjustments, of course, to make sure that we turn the direction that provides the shortest path.

Okay, let's finish up and actually reposition the stepper motor.

```
  stpDelay = MoveTime / span

  DO
    stpIdx = stpIdx + stpDir // 4
    READ (Steps + stpIdx), Coils
    PAUSE stpDelay
    span = span - 1
  LOOP WHILE (span > 0)

  pos = angle

  RETURN
```

The rest of the subroutine starts by calculating the delay between steps. The value is set by dividing the total movement time (500 ms in our program) by the number of steps in our move. Note that we must have some delay – the stepper motor needs it. And we'll want to experiment with this a bit. The largest possible move is half a revolution. Setting the time for that move to 500 milliseconds seems reasonable; fast enough to get there swiftly, but no so fast as to sling coffee out of the spout.

A DO-LOOP is used to send the step data to the motor. We're back to our old tricks with the modulus operator – this allows us to set the direction with a variable. The coil data is pulled from a DATA table with READ and placed right on the pins. Of course, we cannot drive a stepper directly with BASIC Stamp IO pins; we need a high-current buffer to handle the coil load. Figure 113.3 shows how to use our old friend the ULN2003 to handle the load for us.

Finally, we need to update the pos variable with our new position and start the whole process again.

Short and Sweet Rules!

Okay, I know this project wasn't a big mental challenge, but I do think it's useful – even if we don't automate our car's cup-holder. I'm sure you'll figure out something fun to do with it, and when you do, don't hesitate to share your results with me.

Until next time, Happy Stamping.

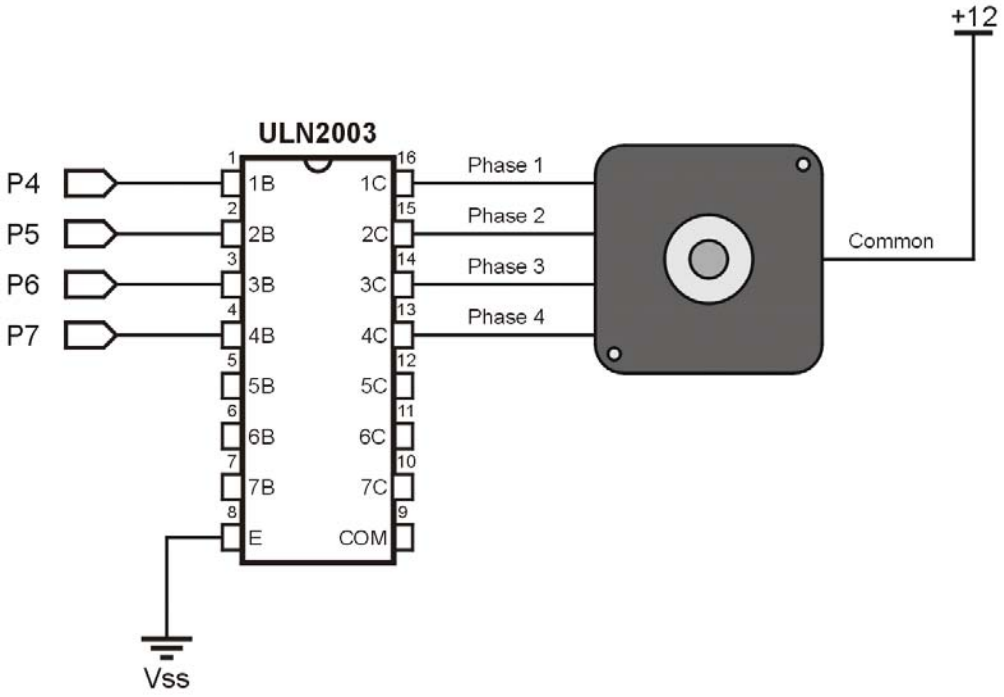

Figure 113.3: ULN2003 to Stepper Motor

```
' ================================================================================
'
'   File...... Coffee_Caddy.BS2
'   Purpose... To prevents coffee spills while driving
'   Author.... Parallax, Inc.
'   E-mail.... jwilliams@parallax.com
'   Started...
'   Updated... 15 JUL 2004
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' ================================================================================


' -----[ Program Description ]---------------------------------------------


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


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

Xpin            CON     0                       ' x output from Memsic
Ypin            CON     1                       ' y output

Coils           VAR     OUTB                    ' stepper coils


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

#SELECT $stamp
  #CASE BS2, BS2E
    Scale       CON     $200                    ' 2.0 us per unit
  #CASE BS2SX
    Scale       CON     $0CC                    ' 0.8 us per unit
  #CASE BS2P
    Scale       CON     $0C0                    ' 0.75 us per unit
  #CASE BS2PE
    Scale       CON     $1E1                    ' 1.88 us per unit
#ENDSELECT

RevSteps        CON     100                     ' steps per revolution
HalfRev         CON     RevSteps / 2            ' half revolution
MoveTime        CON     500                     ' move time (pos --> angle)

StepCycle       CON     4                       ' 4 full-steps in cycle
MCW             CON     1                       ' point to next table item
MCCW            CON     StepCycle - 1           ' point to previous
```

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

xG              VAR     Word                    ' x axis g-force input
yG              VAR     Word                    ' y axis g-force input
angle           VAR     Byte                    ' direct to point at
pos             VAR     Byte                    ' current spout pos
span            VAR     Byte                    ' ABS (
stpIdx          VAR     Nib                     ' pointer to phase table
stpDir          VAR     Nib                     ' direction of move
stpNum          VAR     Byte                    ' number of steps
stpDelay        VAR     Byte                    ' delay between steps


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

Steps           DATA    %0011, %0110, %1100, %1001


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

Reset:
  DIRB = %1111                                  ' stepper pins are outputs
  pos = 0
  PAUSE 100                                     ' let Memsic get ready


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

Main:
  GOSUB Get_Memsic                              ' read g-forces
  angle = 100 - ((xG ATN yG) */ 100) // 100     ' scale angle 0 - 99
  IF (angle <> pos) THEN                        ' angle change?
    GOSUB Move_Spout                            ' yes, adjust
  ENDIF
  GOTO Main

  END


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

' Read Memsic 2125
' -- scale output to -50 to +50

Get_Memsic:
  PULSIN XPin, 1, xG                            ' read Memsic 2125
  PULSIN YPin, 1, yG
  xG = xG */ Scale + 50 / 100 - 50
  yG = yG */ Scale + 50 / 100 - 50
  RETURN
```

```
' Determine shortest path to new angle
' -- move in "MoveTime" period

Move_Spout:
  IF (angle > pos) THEN
    span = angle - pos
    IF (span < HalfRev) THEN
      stpDir = MCW
    ELSE
      stpDir = MCCW
      span = RevSteps - span                    ' adjust for short route
    ENDIF
  ELSE
    span = pos - angle
    IF (span > HalfRev) THEN
      stpDir = MCW
      span = RevSteps - span
    ELSE
      stpDir = MCCW
    ENDIF
  ENDIF

  stpDelay = MoveTime / span                    ' delay between steps

  DO
    stpIdx = stpIdx + stpDir // 4               ' update step pointer
    READ (Steps + stpIdx), Coils                ' output new step data
    PAUSE stpDelay
    span = span - 1
  LOOP WHILE (span > 0)

  pos = angle                                   ' update position

  RETURN
```