Column #107 March 2004 by Jon Williams:

Measure High, Measure Low

I'm pretty sure I've made this confession before, but if I haven't, here goes: I'm a bit of nut when it comes to temperature. Let's just say that I have an exceptional temperature curiosity. I have thermometers of one sort or another spread from one end of my home to the other, and I seem to be checking them constantly. I even found a useful little travel clock with an atomic clock and a thermometer built in; I can keep track of the exact time when I travel and monitor the environment at the same time -- I like that.

In science and industry, one of the most popular methods of measuring temperature is with a thermocouple. They're inexpensive, fairly accurate, and easy to use. They're easy, but the process of using them properly is not necessarily very simple. Let's back up a bit ... a thermocouple is constructed by joining two dissimilar metal wires at one end. A voltage will be developed between the joined and open end that is proportional to the temperature difference between the two ends. This is called the Seebeck voltage, named after Thomas Seebeck who discovered the effect in 1821.

The trick is that the Seebeck voltage is very small; on the order of fractional- to low-millivolts, so we just can't pull out our trusty DMM and measure it. Another thing is establishing a reference at what is called the "cold junction" (the point where we measure the Seebeck voltage). This connection is called the cold junction because prior to electronic compensation, this connection point was placed in an ice bath to keep it at (or very near) zero degrees Celsius. If you look at a standard thermocouple table you'll see that the reference junction is specified at zero degrees Celsius.

Lucky for us, technology is on our side. Dallas Semiconductor makes a neat little chip called the DS2760 which was actually designed for monitoring Lithium-Ion batteries, but works very nicely as a thermocouple interface. To the best of my knowledge, the use of the DS2760 as a thermocouple interface was originally presented by Dan Awtry of Dallas Semiconductor. What we're going to do this month is create a program for the BS2pe (or BS2p) that will talk to the DS2760 and display the thermocouple temperature in Celsius and Fahrenheit.

Temperature on a Wire

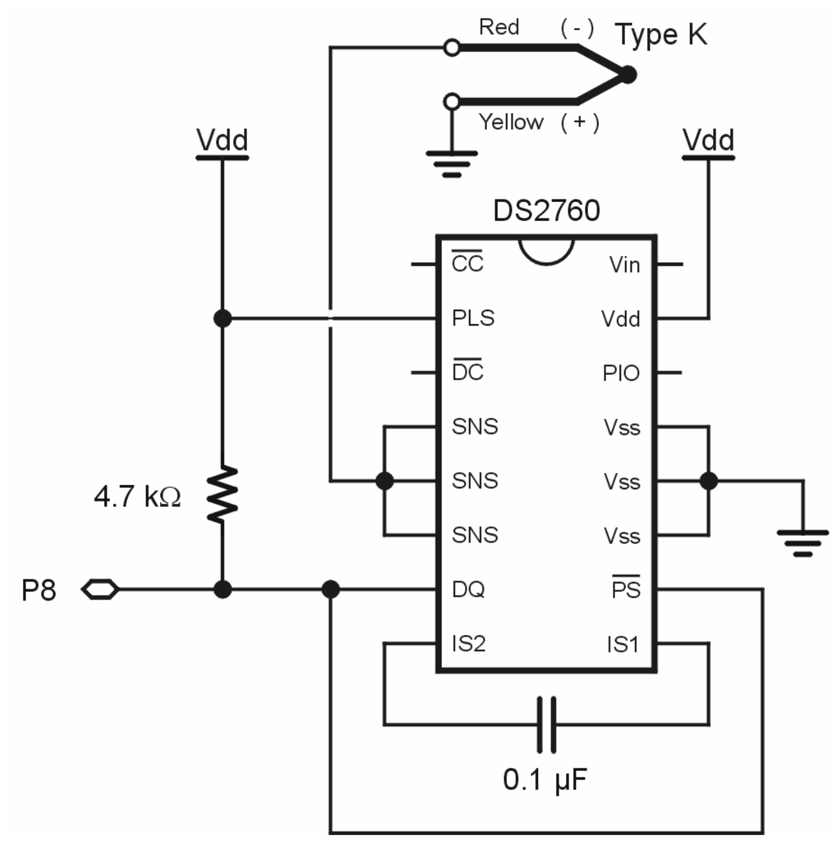
As you can see in Figure 107.1 the DS2760 is a one-chip solution for thermocouple interfacing. The BS2p/BS2pe makes talking to a 1-Wire device a snap; the rest is just assembling the code. Here's the plan.

- Measure the Cold Junction temperature
  (this comes from inside the DS2760)
- Measure the Seebeck voltage

- Find the thermocouple voltage that corresponds
  to the cold junction temperature
- Adjust the Seebeck voltage based
  on the cold junction temperature
- Look up the compensated temperature
  and display on LCD

Alright ... you know the drill: We've planned our work, now let's work out plan.

Figure 107.1: DS2760



Let's start at the top and make sure that we actually have a DS2760 connected. Note that this program is designed for just one sensor – it can be modified for multiple units but that's beyond the scope of what we're going to do here (You could, for example, put tables for various thermocouple types in different tables and select the type – you can download an example of this from Parallax). After checking to make sure that we're connected to a BS2p or BS2pe (required for 1-Wire communications), we initialize the 2x8 LCD and then retrieve the serial number from the 1-Wire device connected to P8.

```
Check_Device:
  OWOUT OW, %0001, [ReadNet]
  OWIN  OW, %0010, [SPSTR 8]
  GET 0, char
  IF (char <> $30) THEN
    LCDOUT E, LcdCls,   ["NO"]
    LCDOUT E, LcdLine2, ["  DS2760"]
    STOP
  ENDIF
```

We'll send the ReadNet ($33) command to the DS2760 using **OWOUT**, specifying a front-end reset (perform the reset process before sending data). ReadNet instructs the connected 1-Wire device to transmit its eight-byte serial number. Since we're not going to put the whole thing to use – but may want to display it later – we'll buffer it into the Scratchpad RAM using the SPSTR directive with **OWIN**. The first byte of the serial number string will be the device type; for the DS2760 this is $30. If that byte isn't $30 the program will put a message on the LCD and stop the program.

The reason we don't use **END** where **STOP** appears above is that **END** puts the Stamp into low-power mode. The Stamp's watchdog timer will interrupt the low-power mode every 18 milliseconds causing the pins to "glitch" (this is a known behavior). What I saw happen in testing is that the glitch on the LCD's E pin caused the display to be blanked, obliterating the message. **STOP** halts the program without placing the Stamp in low-power mode, so the IO pins remain in their current state without interruption.

Unless there's a problem we shouldn't get the "NO DS2760" message and we'll move right into the main loop that measures temperature using the process described earlier. The first step is to measure the cold junction temperature. This comes from inside the DS2760.

The temperature is read from registers $18 and $19. This value is 11 bits (10 bits plus sign), and interestingly, Bit10 (sign) is left-aligned with the MSB (Bit15) of our variable *tmpCJ*. Let's look at the code, and then go through it.

```
Read_CJ_Temp:
  OWOUT OW, %0001, [SkipNet, $69, $18]
  OWIN  OW, %0010, [tmpCJ.BYTE1, tmpCJ.BYTE0]
  IF (tmpCJ.BIT15) THEN
    tmpCJ = 0
    error = 1
  ELSE
    tmpCJ = tmpCJ.HIGHBYTE
    error = 0
  ENDIF
  RETURN
```

To retrieve the temperature we send the SkipNet ($CC) command (only one device is connected, so no serial number is required), followed by $69 (read), and then the register. Since 1-Wire devices work with bytes, our **OWIN** instruction breaks the *tmpCJ* variable into bytes using internal PBASIC aliases BYTE1 (high byte) and BYTE0 (low byte).

Remember that the temperature is left-aligned within *tmpCJ*, so the sign is currently sitting in Bit15. If this bit is one the temperature is negative. To keep things simple we will disallow negative cold junction temperatures (in theory it should be zero C, not lower) and set *tmpCJ* to zero and the error flag to one.

When the temperature is – as it will be in most cases – positive, we can convert the raw value to degrees by taking the high byte of the raw temperature. Yes, I know what you're thinking: "Huh?" Okay, here goes.... The raw value needs to be right shifted by five bits to correct the alignment. Okay, that's easy. Then we have to multiply by 0.125 to get whole degrees. As it turns out, 0.125 is a convenient fractional value because multiplying by 0.125 is the same as dividing by eight. And, as luck would have it, dividing by eight is the same thing as a right shift by three bits. So, in total, we have a right shift of eight bits which means that our whole degrees result is simply the high byte of the raw temperature value.

Let me interrupt this broadcast for a minute and talk about those "convenient fractional values." While the Stamp has operators (*/ and **) that can help with fractions there, are times when we don't need to take that route. In this case, for example, we could have used the */ operator with $40 to multiply by 0.125, but it's simpler to divide by eight. Now I admit, 0.125 is a common value and easy to recognize, but what about a value like 0.0769? Here's a tip: When in doubt about a fraction (that is less than one), enter it into your scientific calculator and then press the reciprocal [1/x] key. If the value is a whole number (or very very close), cha-ching! ... divide by the whole number. And if that value happens to be an even power of two (2, 4, 8, 16, 32 ...) then we can use the shift operator instead of divide since it's faster.

Okay, back to work. The next step is measuring the Seebeck voltage from the thermocouple. The process is identical to measuring temperature.

```
Read_TC_Volts:
  OWOUT OW, %0001, [SkipNet, $69, $0E]
  OWIN  OW, %0010, [tCuV.BYTE1, tCuV.BYTE0]
  signTC = tCuV.BIT15
  tCuV = tCuV >> 3
  IF signTC THEN
    tCuV = tCuV | $F000
  ENDIF
  tCuV = ABS tCuV */ 4000
  RETURN
```

The voltage is stored as a 13 bit (12 bits plus sign) value in the current registers ($0E and $0F) of the DS2760. The reason it's in the current register is that the DS2760 uses a shunt to convert a current to voltage for reading. In our application we're using the external sense resistor version of the DS2760 which lets us measure a voltage with a resolution of 15.625 microvolts per bit.

After retrieving the voltage into the variable *tCuV*, we save the sign by making a copy of bit 15. As with the temperature, the voltage is going to be left-aligned when in our word variable, and the sign bit is the MSB. After the sign is saved we correct the bit alignment in *tCuV* by shifting right three bits.

Now, if the sign bit is one, that means the voltage is negative and the value in *tCuV* is represented in two's-compliment format. Keep in mind that the shift process pads the opposite end with zeros (the high-end bits in this case), so we need to put ones in those positions to make the two's-compliment value of *tCuV* correct. This will let the **ABS** function return the right value.

The final step is to multiply by 15.625 to get microvolts. As the factor is fractional and greater than one, we'll uses the star-slash (*/) operator. The parameter for star-slash is calculated by multiplying 15.625 and 256.

Okay, we have the cold junction temperature and the Seebeck voltage; now it's time to do a bit of math and determine the actual thermocouple temperature.

Turning the Tables on Tough Math

A key element of this program is the use of large tables to hold the thermocouple data. The reason we use a table is that the thermocouple output voltage versus temperature is not linear and, in fact, would require a multi-order equation to maintain accuracy. One of my favorite features of the BS2p family is the ability to use **READ** and **WRITE** across program slots. This lets us put our code in slot zero, and our table(s) slots one and higher. The **STORE** instruction is used to select a table.

To compensate for a cold junction value above zero degrees Celsius, we'll determine the voltage that would be generated by that temperature for our thermocouple. This is simple; we point at our table with **STORE**, and then calculate the address within the table by multiplying our cold temperature value by two. This is necessary since we are using words (two bytes) to store the thermocouple output voltages.

```
    STORE PosTable
    READ (tmpCJ * 2), Word cjComp
```

Notice that we're taking advantage of a new PBASIC 2.5 feature: using the Word modifier with **READ**. The only caveat is that data must be stored in the table as low-byte, high-byte. This is not a problem for us as we're creating the table using the Word modifier. At this point, *cjComp* holds the cold junction compensation voltage for our thermocouple.

Now it's time to combine the compensation voltage with the Seebeck voltage. After we've done that, we can do a reverse lookup in the table to determine the thermocouple temperature.

```
    IF (signTC = 0) THEN
      ' TC is above cold junction
      cjComp = cjComp + tCuV
      STORE PosTable
      tblHi = 1023
      signC = 0
    ELSE
      ' TC is below cold junction
      IF (tCuV <= cjComp) THEN
        cjComp = cjComp - tCuV
        STORE PosTable
        tblHi = 1023
        signC = 0
      ELSE
        cjComp = tCuV - cjComp
        STORE NegTable
        tblHi = 270
        signC = 1
      ENDIF
    ENDIF
```

A bit of logic is used to do the combining for a couple of reasons. The first is that we've simplified other aspects of the code by maintaining a separate sign bit from the Seebeck voltage. The other reason is that we actually have two tables: one for positive temperatures (up to 1023 degrees C), one for negatives (down to -270 degrees C).

Things are easiest when the Seebeck voltage is positive. In this case we simply add the compensation voltage to the thermocouple voltage, and point to the positives table. We'll set the upper end of our table search to 1023 (this is the last Word in the table) and set the sign bit for Celsius degrees to zero since we know it's positive.

When the Seebeck voltage sign is one (voltage is negative) this indicates that the temperature is lower than the cold junction temperature, but we don't know if it is below zero Celsius, so we need to apply a bit of additional logic. If the Seebeck (absolute value) voltage is less than or equal to the compensation voltage, we can subtract it from the compensation voltage and point to the positive tables as before. When the Seebeck voltage is greater than the compensation voltage this means that the thermocouple temperature is below zero Celsius. We calculate the compensated voltage by subtracting the original compensation voltage from the Seebeck value, then pointing to the negatives table. Notice that the high end of the search for the negatives table is only 270. The reason for this, of course, is that absolute zero is 270 degrees Celsius. And since the temperature is negative we will set the Celsius sign bit accordingly.

Where In the Table is My Value?

With the compensated voltage (*cjComp*) in hand, all we have to do now is find that value – or it's closest match – in the table and that position will be our actual thermocouple temperature. Okay, how do we find it? One approach, the easiest, is to start at the bottom of the table and scan upward until we find a match or exceed our search value. The trouble with this method is that it can take a very long time to find a value that is in the high end of the table.

Searching large tables is nothing new, and we can borrow from computer science solutions. When the table is ordered, we can use what is called a binary search. This is a divide-and-conquer approach to searching a table (or array). To do a binary search we'll need three pointers: the low end of the search, the high end of the search, and the midpoint. We find the midpoint by adding low and high together, then dividing by two. Then we'll check the value at the midpoint against our search value (*cjComp*). If we find a match we jump right out of the search. If the midpoint value is not a match we will compare it against the search value. When the search value is lower than the midpoint value, we'll reset the high end of the table search to the midpoint. If the search value is higher than the midpoint table value, we'll reset the low end of the search to the midpoint. As you can see, we get rid of half of the available search values with every iteration of the search loop. This makes the binary search very fast and lets us find any value within 10 loop iterations.

```
TC_Lookup:
  tblLo = 0
  tempC = 22

  READ (tblHi * 2), Word testVal
  IF (cjComp > testVal) THEN
    error = 1
  ELSE
    DO
      eePntr = (tblLo + tblHi) / 2
      READ (eePntr * 2), Word testVal

      IF (cjComp = testVal) THEN
        EXIT
      ELSEIF (cjComp < testVal) THEN
        tblHi = eePntr
      ELSE
        tblLo = eePntr
      ENDIF

      IF ((tblHi - tblLo) < 2) THEN
        eePntr = tblLo
        EXIT
      ENDIF
    LOOP
    tempC = eePntr
  ENDIF
  RETURN
```

Our code is actually modified a bit from the traditional binary search. In typical application, the search will report the position or "not found." We want the closest position if the actual value is not in the table. This is accomplished by monitoring the span between the high and low pointers. When it falls to one or zero, we've searched the whole table and we will use the low pointer as our search result.

Now that we have the correct Celsius temperature, we can convert to Fahrenheit and send the values to an LCD.

```
Display_Temps:
  IF (tempC = 0) THEN
    signC = 0
  ENDIF

  tempF = tempC * 9 / 5
  IF (signC) THEN
    tempF = 32 - tempF
  ELSE
    tempF = tempF + 32
  ENDIF
  signF = tempF.BIT15
  tempF = ABS tempF

  LOOKDOWN tempC, >= [1000, 100, 10, 0], idx
  LCDOUT E, LcdLine1, [223, "C ", REP " "\idx,
                      signC * 13 + 32, DEC tempC]

  LOOKDOWN tempF, >= [1000, 100, 10, 0], idx
  LCDOUT E, LcdLine2, [223, "F ", REP " "\idx,
                      signF * 13 + 32, DEC tempF]
```

Before we do the conversion well fix the sign bit for Celsius if required. There may be times when the temperature is just a hair below zero and the sign bit will get set. It's an easy fix.

There's no magic in converting from Celsius to Fahrenheit; we use the formula F = C x 9 / 5 + 32. As our program uses absolute values with a separate sign bit, an **IF-THEN** structure will take care of the "+ 32" part of the equation. And this actually points to another reason for using absolute values: the divide operator (required in the Fahrenheit conversion) cannot be used with two's-compliment (negative) values.

To keep things on the LCD neat, I use Tracy Allen's right justification trick with REP (repeat) modifier for serial output instructions (**SEROUT**, **I2COUT**, **OWOUT**, and **LCDOUT** [even though it uses a parallel buss]). A **LOOKDOWN** table is used to determine the width of our value, and then the width is used to pad the display with spaces ahead of the printed value. The sign bit is used with a bit of math to print a space for positive values and a hyphen for negatives. The DEC modifier finishes the process.

Temperature Hunting

As you can see in the photos, I assembled my test unit on a standard BOE. By using a nine volt battery I was able to roam around and test temperatures. My first spot of interest was the hot water coming out of the tap in my hotel room (I'm visiting the California office as I write this). How hot is it? A whopping 140 degrees Fahrenheit! That's hot. But I've got access to hotter things, like that burner on the stove: over 800 degrees. And what about measuring cold temperatures? I picked up some dry ice at the supermarket and measured it at around minus 100 degrees Fahrenheit. Wow, that is cold.

Please ... before you go off on your own temperature hunting expeditions, be aware that you can be burned by extreme heat or extreme cold (like dry ice). I had a friend take the photos for me so that I could focus on not getting too close to the "danger zone" with my hands.

Even if your thing isn't thermocouples or temperature measuring, I do hope that you found the use of tables interesting. After finishing this project, I thought of a couple other projects that could be simplified with a table, and I would get better resolution than using integer math. You can use your favorite PC programming language to calculate values and output your table (as text that can be copied into the Stamp editor). I'm currently experimenting with a very interesting multi-platform language called Python. Check it out, you might find it interesting and useful too.

Until next time, Happy Stamping