Column #39, May 1998 by Jon Williams:

Building A Custom Digital Thermometer

I've always wanted to build a stand-alone, graphing thermometer but, until recently, I was not able to do it with the BASIC Stamp. Once again, serial peripherals — coupled with the strength of the PBASIC language — allow the seemingly impossible to come true.

Not that this project wasn't without its challenges; believe me, it was. Due to the low pin count, I decided to build it in a Stamp 1. Wow, was it a lot of work! There were several moments during the software development that I nearly gave up and punted to the Stamp 2. The struggle, however, was worth it in the end. I managed to squeeze ("crowbar" would be the more accurate term here) all the features I wanted into the BS1. There are a couple of danger points along the way. I will explain the choices I made as we work through the program so there are no surprises should you decide to build your own. Okay, let's jump into it. Strap on your seatbelts — this could be a wild ride.

The Project

My goal was to build a digital thermometer that displays the current temperature, recorded high, recorded low, and a graph of the last hour's worth of readings. For the display, I chose the Scott Edwards Electronics G12864 graphical LCD (refer to the March '98 edition of this column for details on the G12864).

This LCD lets me display text and graphics, which meets my needs for the digital display and the graph of recent temperatures.

To measure temperature, I chose the Dallas Semiconductor DS1620 digital thermometer — a thermometer on a chip. Scott wrote about the DS1620 in this column back in April '95, so I won't go into all of the details. Even if you don't have that issue, analysis of this program and my notes should help you understand this chip.

The last part I needed was some external memory — even new users are aware that there is not enough RAM for the Stamp to store 90 minutes worth of temperatures. Thankfully, Solutions Cubed builds a neat little device called the RAMPack B (herein refered to as RAMPack). The RAMPack can store up to 32K of data and needs only a single pin to communicate with the Stamp. The base unit comes with a socketed 8K RAM chip, which is plenty for this project, as I ultimately needed to store just 90 minutes of temperatures. The serial peripherals make the hardware side of this project very simple; only five of the Stamp's pins are used (refer to Figure 39.1). As I've already mentioned, it was the software that was a real bear. What I'm about to take you through is my design ideas and how I actually managed to make it all work.

Program Analysis

Based on my goals, there were several discrete areas that I could block out to build the framework of the program. Among them:

- Initialization
- Get the current temperature
- Record high/low
- Display digital temperatures
- Record historical
- Graph historical temperatures

All of this was built around a synthesized clock. I had no need for a real-time clock, and yet, I wanted my graph to have a time-based orientation. I decided that I would read and display the temperatures approximately every second, then draw the temperature graph every minute. The graph scrolls from right to left, that is, the most recent temperature is displayed on the right edge. When the graph is updated, the left-most value is lost due to the circular nature of the external memory buffer (more on this later).

In the beginning, there are the SYMBOLs. I've commented many times about using them and I'll do so again. Use them! End of commentary. Even in simple programs, SYMBOLs will make your program easier to read and maintain. This program would be a nightmare to analyze without the use of SYMBOLs.

You'll notice that Pin0 gets two definitions: one called DQ and the other called DQpin. This is necessary to change the direction on this pin, which is used to communicate data to and from the DS1620. The definition DQpin allows us to set the direction of the pin (input or output). The definition DQ allows us to send and receive bits of data.

The rest of the definitions are very straightforward and intended to make the program easier to read and maintain. You beginners might be wondering how I knew what SYMBOLs I'd need before I actually started into the heart of the program. I didn't — no one does. Programming is an iterative process. As your program takes shape, review and update your SYMBOLs list to make sure that you don't end up with mystery code. As a final review, print your listing and mark it up with a red pen. Make sure that your SYMBOLs and comments are appropriate. I promise you, the short amount of time you spend doing this editing will be made up the next time you come back to the program.

There is nothing worse than trying to figure out one's own logic ...

The initialization section of the code is pretty simple. First, we start with Dirs and Pins. You'll notice that Pin7 (used to talk to the RAMPack) is set as an output and initially set HIGH. This is a requirement of the RAMPack.

Our next step is to set up the DS1620 digital thermometer. Since the BS1 does not have SHIFTIN and SHIFTOUT in its vocabulary, we have to synthesize them. That is the purpose of the subroutines called CmdOut and TempIn. CmdOut sends an eight-bit command to the DS1620. In our case, we're going to configure it for use with a CPU. TempIn is used to retrieve the nine-bit temperature value from the DS1620.

Now for our first of many choices. The DS1620 has a temperature range of –55C to +125C (-67F to +257F). Since I'm using my thermometer indoors, I'm going to ignore the sign bit and limit my range to positive Centigrade values. At zero degrees Centigrade, this means that my lowest recordable temperature is 32 degrees Fahrenheit. No problem; if it ever gets that cold in my home, I won't be here! I limit the high temperature to 120 degrees Fahrenheit. This keeps it in line with my graphic scale and, once again, it is well beyond expected norms.

With the DS1620 initialized, it's time to clear our external memory buffer. If we don't, we might end up with trash in the display until we've had 90 readings. You might be wondering why the RAMPack doesn't clear the RAM during its initialization. The reason is that it's compatible with NV (non-volatile) RAM modules. If you were using NVRAM in a project, you certainly wouldn't want to lose that data every time you started your program. Note that when writing values to the RAMPack, you've got to pause just a bit to allow the RAMPack's processor to save your data.

Sending data to the RAMPack is very simple with the SEROUT command. The first byte sent is always $55. This allows the RAMPack to detect the Stamp's baud rate. After the synchronization byte, we'll send the command ($00 to write a byte, $01 to read), then the high and low bytes of the address for our data. Since our buffer is only 90 bytes, our high address byte is set to $00. When using the RAMPack to store more that 256 bytes of data, you'll need to make use of the addrHi variable. The write command requires one more parameter: the data. Since we're clearing the buffer at this point, we simply write a zero. The last thing that we need to do in the initialization section is preset the low temperature value, in this case 255 (for a byte-sized variable). This is necessary because the program logic will only record the low temperature if it is less than the last known low. If we didn't take this step, the Stamp's internal initialization would cause this value to be zero and we'd never see anything lower.

Okay, now the going gets a little rougher. We're into the core. Actually, the first part is pretty easy; we retrieve the temperature from the DS1620. Remember that the temperature is returned in Centigrade and, living in the United States, I want to display Fahrenheit.

If you look at the conversion code, it probably looks familiar, but not quite right. It deserves a little explaining. The general formula for converting Centigrade to Fahrenheit is F = C * 9 / 5 + 32. "Okay, so where do the 325 and 10 (in the source code) come from?," you ask. The DS1620 has a resolution of 0.5 C (0.9 F). I considered this enough to worry about, so I do my conversion in tenths, then divide by 10 to get the final value. Bit0 of the raw temperature is equal to 0.5 C. Since the LSB of the whole part of the temperature is in Bit1, the whole value has been multiplied by two. Multiplying by the raw value by five, then, produces a value that is in tenths-of-a-degree Centigrade. "So where did that five go?" Continuing with the conversion equation, you'll see that we're eventually going to divide by five. This five (the divisor) cancels the first five that was used to convert to tenths.

Since we're working in tenths, we have to change the constant 32 to 320. You may be wondering, then, why the code says 325. The reason is rounding. Since the Stamp uses integer math that truncates numbers (no rounding), we need to do the rounding manually. This is an old programmers' trick for rounding up: Convert your value to tenths, add five, then divide by 10. Now you know where the 10 came from. The last part — MAX 120 — limits the value to our usable range.

After we've converted our temperature reading, we update the high and low values. This is very easy. If the new temperature is higher than the highest recorded value, record a new high. If the temperature is lower than the lowest recorded value, record a new low. Now you can see why we initially set the recorded low to a large value.

With all the temperatures recorded, we send them in text form to the LCD. Notice that I used two lines of code to send the temperatures. I started with three (for neatness) but, during the "crowbarring" stage of the program I was forced to consolidate the last two commands to save code space.

The next step is to pad the loop with a PAUSE so that it runs about once per second. I use the constant, OneSec, which is something less than 1000 (the PAUSE value for one second). This is for readability. Even a casual programmer will figure out that I want this loop to run once per second. If I use the value of 1000, however, the loop will be longer because of all the statements that need to be executed.

How did I arrive at the final value of OneSec? With a stopwatch. I started with an initial value of 1000 and ran the program. The timing was accomplished by inserting a temporary PULSOUT command to blip an unused output (monitored with a logic probe). I measured the actual time for 20 of these "blips," then scaled OneSec accordingly. Once this was done, I removed the PULSOUT command as it was no longer needed.

Moving on, we process our synthesized clock. The first thing we do is update the seconds value. Remember the purpose of the modulus (//) operator: it returns the remainder of a division and, when used like this, limits the range of a value. In our case, the seconds will be limited from 0 to 59. When the seconds value is zero, we've had a new minute, so we update minutes in the same fashion. The minutes, however, are allowed a range of zero to 89. This was the largest practical value I could display graphically on the LCD.

When the minute is updated, we put the last temperature reading into the RAMPack. Earlier, I referred to the external memory as circular. The reason for this is that the minutes value is used as the address pointer for temperature storage.

Since the minutes wrap from 89 to zero, we have a circular buffer (it wraps back onto itself) that stores the last 90 minutes worth of temperatures.

With the new temperature recorded, it's time to update our graph. Remember that we want to display the last recorded temperature on the far right. The easiest way to do this is to draw our graph from right-to-left, starting with the last recorded temperature and working backwards.

We start by setting our memory pointer to the current minutes value. Next, we loop through our X-axis, going backwards so that our graph is drawn from right to left. The first part of the loop clears the current X value by drawing a clear line the full span of the graph. Then we get a temperature reading from the RAMPack. With the temperature in hand, we need to calculate the top of the line.

The size of the G12864 allowed me to display temperatures at a resolution of two degrees per pixel. The bottom of the line is at 61. The nature of Stamp 1 mathematics (strictly left-to-right evaluation) caused me to rearrange the formula. I had to multiply 61 by two since the pixel-scaling division would affect it. Don't forget that we add 64 to numeric values when sending them to the G12864 so that we can send them as one byte. Now that we know the location for the top of the line, we change the LCD ink color back to black and draw the line.

The last step in the loop is to update our address pointer. Once again, we take advantage of the wrap-around nature of the modulus operator. This technique allows us to display the last 90 temperature readings in one loop.

Back to the main body, we have to make up for our time away to keep our clock in sync. Drawing lines on the G12864 takes a fair amount of time. Since the graph draws so slowly, I used my stopwatch to time it, and updated the seconds value accordingly. With the graph complete, we start the whole process over.

Wrap-Up

This project bit me - bit me hard. Honestly, I went in a little cavalier figuring it would take me just a couple of hours to write the code. It actually took about 12. Why? Well, the program barely fits, that's why. And that was after a lot of work. When a program is tight, we're sometimes forced to resort to all sorts of experiments. I found, for example, that a simple rearrangement of the main loop saved several bytes of code with no functional change. This is where science becomes art and we find new tricks for our bag.

I used STMPSIZE (see last month's column) every step of the way to monitor my progress.

For you newcomers, don't be frightened by my last statements. As my friend Roger says, sometimes "it's harder than it looks." Most Stamp 1 programs will NOT be this hard, I promise. When things do get tough, don't despair. Just break down your program into the smallest practical chunks and attack them one at a time. Perseverance pays off — I now have my graphing thermometer!

RAMTEMP.ZIP is included on the CD-ROM in this book. It contains the source code and graphics files for the G12864. Happy Stamping.

```
' Program Listing 39.1
' Nuts & Volts: Stamp Applications, May 1998

' ----[ Title ]-----------------------------------------------------------
'
' File...... RAMTEMP.BAS
' Purpose... Measure, store and display 90 temperature readings
' Author.... Jon Williams
' E-mail.... jonwms@aol.com
' WWW....... http://members.aol.com/jonwms
' Started... 28 MAR 98
' Updated...  5 APR 98

' ----[ Program Description ]---------------------------------------------
'
' Digital thermometer displays current, high and low temperatures, and
' a graph of the last 90 temperature readings (~one per minute)
'
' Temperature read from a Dallas Semiconduction DS1620
' Temperatures stored in a Solutions Cubed RAMPack B
' Temperatures and graph displayed on a SEETRON G12854 graphic LCD
'
'
' G12864 Configuration Switch Settings:
' 1 : Off      (Run)
' 2 : Off      (2400 baud)
' 3 : On       (BL On) - do not power from Stamp (use seperate supply)
' 4 : Off      (Esc)
' 5 : Off      (Protect EE - must be On to download custom graphics)
' 6 : On       (Screen 2)

' ----[ Revision History ]------------------------------------------------
'
'  5 MAR 98 : Version 1.0 complete

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

```
'
'  I/O pins
'
SYMBOL  DQ      = Pin0              ' needs to match DQPin definition
SYMBOL  DQpin   = 0                 ' DS1620 data I/O
SYMBOL  CLK     = 1                 ' DS1620 Clock input
SYMBOL  RST     = 2                 ' DS1620 Reset input
SYMBOL  LCDpin  = 6                 ' serial connection to G12864
SYMBOL  RAMpin  = 7                 ' serial connection to RAMPack B

' DS1620 commands
'
SYMBOL  TEMPR   = $AA               ' Read temperature
SYMBOL  THW     = $01               ' Write TH (high temp register)
SYMBOL  TLW     = $02               ' Write TL (low temp register)
SYMBOL  THR     = $A1               ' Read TH
SYMBOL  TLR     = $A2               ' Read TL
SYMBOL  START   = $EE               ' Start temperature conversion
SYMBOL  STOP    = $22               ' Stop temperature conversion
SYMBOL  CFGW    = $0C               ' Write configuration register
SYMBOL  CFGR    = $AC               ' Read configuration register

' RAMPack commands
'
SYMBOL  WrByte  = $00
SYMBOL  RdByte  = $01

' G12864 controls
'
SYMBOL  Esc     =  27
SYMBOL  ClrLCD  =  12               ' Clear LCD text screen
SYMBOL  PosCmd  =  16               ' Position cursor
SYMBOL  TposC   =  65               ' temperature locations in G12864
SYMBOL  TPosH   =  97
SYMBOL  TPosL   = 113
SYMBOL  X1      = 101               ' X1 = 37
SYMBOL  X2      = 190               ' X2 = 126
SYMBOL  White   =  64               ' Ink colors
SYMBOL  Black   =  65

SYMBOL  OneSec  = 895               ' *tuned* PAUSE value


' ----[ Variables ]------------------------------------------------------
'
SYMBOL  cmdByte = B0                ' command sent to DS1620
SYMBOL  loTmpC  = B0                ' lo byte of temperature
SYMBOL  hiTmpC  = B1                ' high byte (sign) of temperature
SYMBOL  tempC   = W0                ' temperature (needs 9 bits)
SYMBOL  tempF   = B2                ' temperature in Fahrenheit
SYMBOL  tHigh   = B4                ' highest temperature reading
```

```
SYMBOL  tLow    = B5                    ' lowest temperature reading
SYMBOL  secs    = B6                    ' seconds counter
SYMBOL  mins    = B7                    ' minutes counter
SYMBOL  shift   = B3                    ' loop counter
SYMBOL  x       = B3
SYMBOL  addrLo  = B8                    ' RAM address
SYMBOL  addrHi  = B9
SYMBOL  y       = B10                   ' graphing coordinate

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


' ----[ Initialization ]---------------------------------------------------
'
Init:   Dirs = %10000111                ' setup I/O pins
        Pins = %10000000

        ' initialize the DS1620
        '
        HIGH RST                        ' alert the DS1620
        cmdByte = CFGW                  ' prepare to write config
        GOSUB CmdOut
        cmdByte = %00000010             ' use with CPU; free run mode
        GOSUB CmdOut
        LOW RST
        PAUSE 10                        ' wait 10 ms for EEPROM write
        HIGH RST                        ' alert the DS1620
        cmdByte = START
        GOSUB CmdOut                    ' start continuous conversion
        LOW RST                         ' end DS1620 commo

        ' clear RamPack storage area
        PAUSE 500
        FOR addrLo = 0 TO 89
          SEROUT RAMpin,T2400,($55,WrByte,$00,addrLo,$00)
          PAUSE 10
        NEXT addrLo

        tLow = 255                      ' initialize low temp


' ----[ Main Code ]--------------------------------------------------------
'
Main:   HIGH RST                        ' alert the DS1620
        cmdByte = TEMPR                 ' prep for temperature read
        GOSUB CmdOut
        GOSUB TempIn                    ' get temp from the DS1620
        LOW RST                         ' end DS1620 commo

        ' convert to Fahrenheit
```

```
        ' (32 to 120 deg F)
        tempF = loTmpC * 9 + 325 / 10 MAX 120

        IF tempF <= tHigh THEN ChkLow
        tHigh = tempF                ' update new high temp
ChkLow: IF tempF >= tLow THEN ShoTmp
        tLow = tempF                 ' update new low temp

ShoTmp: SEROUT LCDpin,N2400,(PosCmd,TPosC,#tempF)
        SEROUT LCDpin,N2400,(PosCmd,tPosH,#tHigh,PosCmd,tPosL,#tLow)

        PAUSE OneSec

        secs = secs + 1 // 60        ' update seconds
        IF secs > 0 THEN Main        ' check for minutes update

Main2:  mins = mins + 1 // 90        ' update minutes
        ' record the latest temp
        SEROUT RAMpin,T2400,($55,WrByte,$00,mins,tempF)
        PAUSE 10
        GOSUB ShGrph                 ' update temperature graph
        secs = secs + 10             ' make-up for graphing delay
        GOTO Main


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

' Send command to DS1620
' 8-bit command is sent LSB -> MSB
'
CmdOut: OUTPUT DQpin                 ' make DQ an output
        FOR shift = 1 TO 8           ' shift eight bits
          LOW CLK
          DQ = Bit0                  ' output LSB of cmdByte
          HIGH CLK
          cmdByte = cmdByte / 2      ' shift cmd for next bit
        NEXT shift
        RETURN


' Retrieve temperature from DS1620
' Data comes in LSB -> MSB in 9 bits
' - 1 bit for sign
' - 8 bits for temp (in 2's compliment format if negative)
'
TempIn: INPUT DQpin                  ' make DQ an input
        FOR shift = 1 TO 9
          tempC = tempC / 2          ' shift temp bits
          LOW CLK
          Bit8 = DQ                  ' get data bit
```

```
          HIGH CLK
        NEXT shift
        RETURN

' Get last 90 readings from RamPack
' Draw temperature graph on G12864
'
ShGrph: addrLo = mins                ' current reading on far right
        FOR x = X2 TO X1 STEP -1

          ' clear the current display line
          SEROUT LCDpin,N2400,(Esc,"I",White,Esc,"L",x,125,x,66)

          ' get reading from RAMPack
          SEROUT RAMpin,T2400,($55,RdByte,$00,addrLo)
          SERIN  RAMpin,T2400,tempF

          IF tempF = 0 THEN nxtAddr
          ' calculate top of line and graph the temperature
          y = 122 - tempF / 2 + 64
          SEROUT LCDpin,N2400,(Esc,"I",Black,Esc,"L",x,125,x,y)

          ' update RAM address pointer (with wrap-around)
nxtAddr:  addrLo = addrLo + 89 // 90
        NEXT x
        RETURN
```