Column #71, March 2001 by Jon Williams:

Conversion Considerations

With five – yes, that's right, five – members of the BS2 family in circulation, it's very likely that we're going to start moving code from one to another; usually from a stock BS2 to one of the new units (BS2sx, BS2e or BS2p). The new Stamp manual from Parallax does a great job explaining the speed differences between the various commands. What we're going to do this month is demonstrate the differences and create portable code that's easy to update when we want to upgrade to a faster Stamp.

BS2 Family Review

Module	I/O Pins (1)	Programs	Scratch RAM (2)	Speed (3)	Current @ 5V
BS2	16 + 2	1 x 2K	N/A	1	8 mA
BS2sx	16 + 2	8 x 2K	63 bytes + 1	2.5	60 mA
BS2e	16 + 2	8 x 2K	63 bytes + 1	1	20 mA
BS2p-24	16 + 2	8 x 2K	127 bytes + 5	3	40 mA
BS2p-40	32 + 2	8 x 2K	127 bytes + 5	3	40 mA

Notes: Each Stamp has two dedicated serial I/O pins. These pins are used for programming and can be used (as pin 16) with SERIN and SEROUT. The BS2p scratchpad RAM locations 127 through 131 are read-only
Speed is shown relative to the BS2

More Power Means More Current

You can see by the chart that all of the multi-program Stamps consume more current than the stock BS2. This is due to the change from a Microchip PIC to the Ubicom SX micro (a change necessitated by the desire for more program and variable space). The BS2sx was the first multi-program Stamp. The BS2sx runs a full two-and-a-half times faster than the BS2, but at a significantly higher current. This is something to think about when using battery power for your applications.

Parallax responded to customer requests for a multi-program, lower-current Stamp with the BS2e. The BS2e uses the same micro as the BS2sx, but at a lower speed. The result is a multi-program Stamp that runs at the same speed as the stock BS2 and has the eight program slots of the BS2sx. This makes the BS2e a nice choice for updating when you run out of program space since there are no syntax changes required for the code to run properly.

More Memory

The BS2sx, BS2e and BS2p have multiple program slots which means, in theory, that we should be able to write longer programs. Right? Well, yes and no. The new Stamps actually have separate banks of program space and a common area called the scratchpad RAM. So what do we do?

The idea is easy: we simply put subroutines or little-used code in one of the other program banks and call it when we need it. There's the rub. You see, we can't do a GOTO or GOSUB across program banks and unless specific precautions are taken, our variables will get mangled when switching from one bank to another.

When using multiple program banks, I take advantage of the BRANCH function and design my main and secondary programs with a task-switcher approach. I generally prefer this design style since it eliminates a lot of IF *statement* THEN *addr* statements in the code. If conditions change than require a different code block to execute, the current task simply changes the task pointer variable. On the next pass through the switcher, the appropriate code will be run. This design is also helpful when using secondary program slots to store subroutines and other code.

For review, here's a simple task-switcher framework that will work with any Stamp:

```
' ----[ Title ]---------------------------------------------------------
'
' File...... TASKER.BS2
' Purpose... Task switcher framework
' Author.... Jon Williams
' E-mail.... jonwms@aol.com

' ----[ Program Description ]-------------------------------------------
'
' General purpose task-switcher for Stamp programs.

' ----[ Constants ]-----------------------------------------------------
'
NumTasks        CON    2                    ' number of tasks

' ----[ Variables ]-----------------------------------------------------
'
task            VAR    Byte                 ' current task

' ----[ Initialization ]------------------------------------------------
'
Initialize:
  task = 0                                  ' define first task

' ----[ Main Code ]-----------------------------------------------------
'
Main:
  BRANCH task,[Task0,Task1]                 ' run current task block
NextTask:
  task = task + 1 // NumTasks               ' point to next task
  GOTO Main                                 ' back to top

' ----[ Subroutines ]---------------------------------------------------
'
Task0:
  ' task code here
  GOTO NextTask

Task1:
  ' task code here
  GOTO NextTask
```

The point of this design is that the task variable keeps track of where we are in the program. We'll use this to our advantage when switching to another program slot. The current task gets saved so that when we come back we know where to start running.

Column #71: Conversion Considerations

The multi-program BS2sx introduced three new keywords to PBASIC:

> PUT *location,byte_data*
> GET *location,byte_data*
> RUN *program_slot*

PUT and GET work like WRITE and READ except that the data is written to or read from the Stamp's scratchpad RAM area. The BS2sx and BS2e each have 63 bytes (addressed as 0 to 62) of user RAM; the BS2p has127. The scratchpad RAM is common to all programs, so it's easy to use as a means of exchanging data between programs.

What about our main program variables? Well, if we're not careful they can be clobbered by our secondary program. Is there a way to prevent this? Yes, there is. The trick is to copy our variable definitions from the main program to the secondary program. When the RUN function is executed, the variable space is not cleared as it is on a reset. If our definitions are the same in both our main and secondary programs, we can happily switch between the two without worry about the variables. Of course, there will be those programs where this isn't possible. In those cases, we can PUT variables that we need to save into the scratchpad and then GET them in the secondary program or when we return. If the secondary program requires its own local variables, simply place the <u>after</u> the shared definitions.

Let's put our little task switcher to work in a simple program that demonstrates these concepts. Be sure to change your $Stamp definition to BS2sx or BS2p if you're not running a BS2e.

```
' ----[ Title ]-------------------------------------------------------
'
' File...... TASKER2.BSE
' Purpose... Task switcher demonstration with external subroutines
' Author.... Jon Williams
' E-mail.... jonwms@aol.com

' {$STAMP BS2e, SUBS.BSE}                    ' or BS2sx or BS2p

' ----[ Program Description ]-----------------------------------------
'
' Simple task switcher demonstration with external subroutines
' (in SUBS.BSE)

' ----[ Constants ]---------------------------------------------------
'
NumTasks        CON    3                     ' number of tasks
SubNum          CON    0                     ' RAM location of external
```

```
' ----[ Variables ]-------------------------------------------------------
'
task            VAR     Byte                    ' current task
count1          VAR     Byte                    ' byte counter
count2          VAR     Word                    ' word counter

' ----[ Main Code ]-------------------------------------------------------
'
Main:
  task = task + 1 // NumTasks                   ' point to next task
RunTask:
  BRANCH task,[Task0,Task1]                     ' run current task block
  GOTO Main                                     ' back to top

' ----[ Subroutines ]-----------------------------------------------------
'
Task0:
  PUT SubNum,0                                  ' define external subroutine
  RUN 1                                         ' run subs program

Task1:
  count2 = count2 + 10                          ' local update
  PUT SubNum,1                                  ' define external subroutine
  RUN 1                                         ' run subs program

Task2:
  DEBUG "(TASKER [2])",TAB,DEC count2,CR
  PAUSE 500
  GOTO NextTask
```

Notice that the first thing the program does is update the task variable. This is necessary since we'll be returning from an external program and need to update the task. This could be done externally, but I generally choose not to. Keep in mind that on reset, Task1 will be the first section of code that runs.

Okay, now for our subroutines program. Again, we must define any local variables after our shared definitions – this will keep our main program variables intact.

```
' ----[ Title ]-----------------------------------------------------------
'
' File...... SUBS.BSE
' Purpose... External subroutines for BASIC Stamp program
' Author.... Jon Williams
' E-mail.... jonwms@aol.com

' {$STAMP BS2e}                                 ' or BS2sx or BS2p

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

Column #71: Conversion Considerations

```
'
' Subroutines for TASKER.BSE

' ----[ Constants ]-----------------------------------------------
'
SubNum          CON    0                    ' RAM location of subroutine

' ----[ Variables ]-----------------------------------------------
'
' (shared)
task            VAR    Byte                 ' current task
count1          VAR    Byte                 ' byte counter
count2          VAR    Word                 ' word counter

' (local)
sub             VAR    Byte                 ' subroutine to run

' ----[ Main Code ]-----------------------------------------------
'
Main:
  GET SubNum,sub                            ' read sub to run from RAM
  BRANCH sub,[Sub0,Sub1]                    ' run the subroutine
  RUN 0                                     ' return if bad sub number

' ----[ Subroutines ]---------------------------------------------
'
Sub0:
  DEBUG "(SUBS [0])",TAB,DEC count1,CR       ' show count1
  count1 = count1 + 1                        ' external update of variable
  PAUSE 100
  RUN 0                                      ' return to main program

Sub1:
  DEBUG "(SUBS [1])",TAB,DEC count2,CR       ' show count2
  PAUSE 100
  RUN 0                                      ' return to main program
```

Using this approach, you can extend your programming space and keep things organized. Just remember that if you have to pass word-sized variables via the scratchpad, two PUT and GET functions are required as PUT and GET work only with bytes.

Speedy Updates

In addition to the added program space, the BS2sx and BS2p give us extra speed. This is great, especially when we're trying to get quite a bit done in a short period of time. What this means for us, however, that we'll have to make changes in our code to accommodate the higher speed of the new Stamps. Unfortunately, it's not a simple matter of factoring by 2.5 (BS2sx) or 3 (BS2p). For specifics, you need to consult the new (Version 2.0) Stamp manual. The new manual does a great job of detailing the speed differences for each function and it's available for download at no charge from Parallax.

The following functions will require updating when moving from the BS2 or BS2e to the BS2sx or BS2p:

> COUNT
> DTMFOUT
> FREQOUT
> PULSIN
> PULSOUT
> PWM
> RCTIME
> SERIN
> SEROUT

Let's take a look at one of these functions and what we can do to plan for changes with a faster Stamp. The following program reads the frequency from a simple pulse generator (Figure 71.1).

Figure 71.1: Simple pulse generator
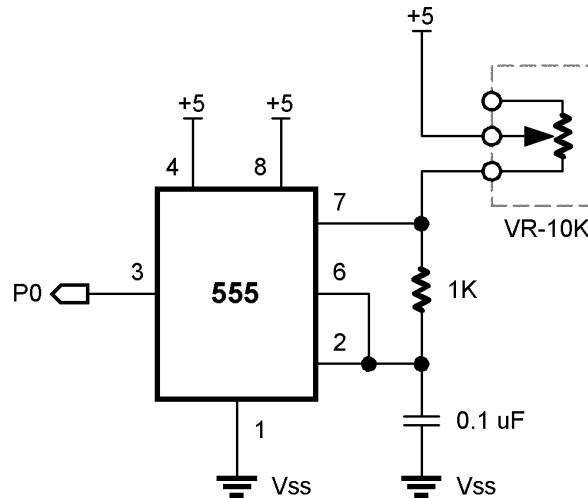

```
' ----[ Title ]------------------------------------------------------------
'
' File...... SHOWFREQ.BS2
' Purpose... Displays input frequency
' Author.... Jon Williams
' E-mail.... jonwms@aol.com


' {$STAMP BS2}                                ' or BS2e, BS2sx or BS2p


' ----[ Program Description ]----------------------------------------------
'
' This program reads an incoming square wave with PULSIN and displays the
' frequency in Hertz.

' ----[ I/O Definitions ]--------------------------------------------------
'
F_pin           CON    0                      ' frequency input pin

' ----[ Constants ]--------------------------------------------------------
'
FreqCon         CON    $0200                   ' conversion for BS2/BS2e
'FreqCon        CON    $00CC                   ' conversion for BS2sx (0.80)
'FreqCon        CON    $00BF                   ' conversion for BS2p (0.75)


' ----[ Variables ]--------------------------------------------------------
'
pHigh           VAR    Word                    ' high pulse width
pLow            VAR    Word                    ' low pulse width
period          VAR    Word                    ' cycle time (high + low)
```

```
freq            VAR    Word                    ' frequency

' -------------------------------------------------------------------

Main:
  PULSIN F_pin,0,pHigh                         ' get high portion of input
  PULSIN F_pin,1,pLow                          ' get low portion of input
  period = (pHigh + pLow) */ FreqCon           ' calculate cycle width
  freq = 50000 / period * 20                   ' calculate frequency

  ' display on DEBUG screen

  DEBUG Home
  DEBUG "Period...... ", DEC period, " uS    ", CR
  DEBUG "Frequency... ", DEC freq, " Hz     "

  GOTO Main                                    ' do it again
  END
```

I'm a big fan of using CONstants in my programs. I think it makes them easier to read and I know for a fact that it makes code easier to update. This program shows how we can use a constant for the frequency conversion (FreqCon) value in anticipation of an update to a faster Stamp.

We need to use a different value for each Stamp type because of the units value returned by PULSIN. Here's the breakdown

BS2/BS2e	2 us
BS2sx	0.8 us
BS2p	0.75 us

At first blush, you may get concerned over the units change from a whole number (2) to a fractional value. Thankfully, the */ (star slash) operator lets us multiply fractional values. If the low-order byte of our star slash value is zero, we're simply multiplying by the whole number in the high-order byte.

It would be nice if all conversions were this easy but the truth is that they aren't. That said, a little bit of planning will go a long way to simplifying your code conversions. PBASIC is surprisingly resilient and, in most cases, will support your update requirements without a lot of stress.

New Commands In the BS2p

There are new commands in the BS2p that will be of interest to current Stamp users, specifically the LCD commands, I2C commands and scanning inputs for changes.

Stamp users have been connecting standard parallel LCDs to Stamps since the days of the BS1. When porting your BS2 apps to the BS2p, you'll need to note the connection requirements dictated by the BS2p. Here are the options:

LCD	Option 1	Option 2
LCD.E	BS2p.0 or BS2p.1	BS2p.8 or BS2p.9
LCD.R/W	BS2p.2	BS2p.10
LCD.RS	BS2p.3	BS2p.11
LCD.DB4	BS2p.4	BS2p.12
LCD.DB5	BS2p.5	BS2p.13
LCD.DB6	BS2p.6	BS2p.14
LCD.DB7	BS2p.7	BS2p.15

Any change required is well worth the trouble as the LCDOUT command is a joy to work with – all of the nice modifiers of SEROUT work with LCDOUT. Here's a neat little routine to right justify a number in the LCD:

```
RJ_Print:                                    ' right justify
  digits = width
  LOOKDOWN temp,<[0,10,100,1000,65535],digits
  LCDOUT LCDpin,pos,[REP " "\(width-digits),DEC temp]
  RETURN
```

The routine prints the variable temp on the LCD at the location specified by pos. The width of the field is specified by the variable width. This routine works so nicely because we're able to use the REP and DEC modifiers with LCDOUT.

Many Stamp programmers have successfully connected I2C parts to the BS2, BS2sx and BS2e. This is possible due to the synchronous nature of the Philips I2C buss. The only drawback is the manual code required to do it.

If connecting I2C parts to your Stamp projects, the BS2p will be a real time and code save. You only need to be mindful of the BS2p connection requirements imposed for I2C parts. Here are your options:

I2C Bus	Option 1	Option 2
SDA	BSP.0	BSP.8
SCL	BSP.1	BSP.9

Once again, any changes necessitated by these connection requirements will be well worth the effort. Like SERIN and SEROUT, I2CIN and I2COUT are very easy to use and work with the new SPSTR serial modifier. With SPSTR you can buffer a large block of data from an I2C device to the scratchpad RAM area.

Another reason to consider the BS2p is if your current Stamp program is constantly scanning inputs for a change. By using the latching option of POLLMODE (mode value + 8), your inputs will be saved until the code has time to check them (and you can eliminate external latching hardware if you've been using it). By implementing the task-switcher design we discussed earlier, a task can be defined to check polled inputs by reading the interrupt pin states from scratchpad locations 128 through 131. The next task or any other action can be determined by the state of the inputs. Be sure to re-issue POLLMODE to clear inputs before the next scan.

Finally, adding Dallas 1-Wire devices to anything other than the BS2p requires external hardware. If you're interested in using 1-Wire technology, the BS2p is the only way to go.

Times Change

Products change. Luckily for us consumers, most product changes bring improvements and that is clearly the case with Parallax's BS2 series of controllers. One can only wonder what cool things are coming next. Whatever it is, I'm sure it will be worth the wait.

Happy Stamping.