



Column #100 August 2003 by Jon Williams:

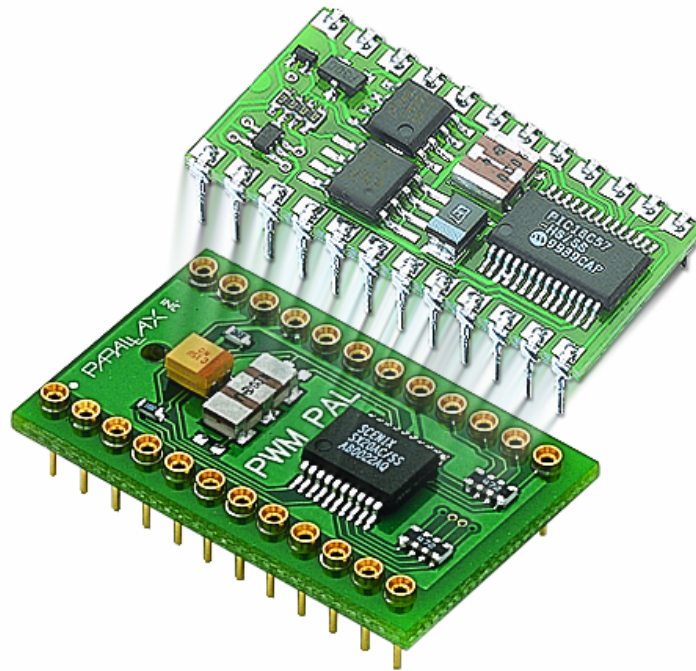
Get Your Motor Runnin'

Most people don't realize that the BASIC Stamp 2 has actually been around for quite a long time. Like the BASIC Stamp 1, it was designed to be a general-purpose embedded controller. And it's a darned good one; just look how many copy-cat products exist today.

Not long after the introduction of the first BASIC Stamp a new industry was created: the serial accessory industry. It all started with Scott Edwards' "Stamp Stretcher." After that, Scott created the serial LCD controller that we all take for granted (and has been copied by many) and a serial servo controller as well. With Scott's success, others jumped into the fray and there is a multitude of serial accessories that will work with the BASIC Stamp and other micros.

Even Parallax has created serial accessories for the BASIC Stamp; usually in the form of the "AppMod" that plugs into the expansion socket on the Board-of-Education. There's an LED terminal, a sound module, a compass – like I pointed out earlier, there's a tremendous variety of accessories devices for the BASIC Stamp.

Figure 100.1: PWM Pal Fits Between BASIC Stamp and Socket



PWM Made Painless

The latest from Parallax is the PWMPAL. It works very much like an AppMod (programmed through a serial connection and uses the AppMod protocol), but it is physically and mechanically different; it's configured as a "smart socket" (Figure 100.1). This has been done by other companies and the guys at Parallax like the idea: pop the Stamp out of its socket, drop in the PWMPAL, plug the Stamp in to it and away you go. By mounting the PWMPAL right under the Stamp so that they share pins, the Stamp is given new features. Neato.

So, you're wondering, "What is a PWMPAL, anyway?" Let me tell you. The PWMPAL is a four-channel PWM generator/controller and background counter. Any of the shared PWM pins [P12 – P15] that aren't used as PWM outputs can be used by the BASIC Stamp for any purpose. The control and counter pins [P8 – P11] can be used as inputs by the BASIC Stamp even when they're being used for PWM control or counter inputs. They can also be used by the BASIC Stamp as outputs to control a preset PWM channel that is running under hardware control. More on this later.

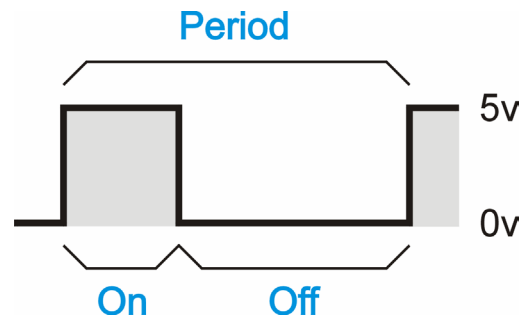
Like the BASIC Stamp, the PWMPAL is a general-purpose device. Instead of being dedicated to a single function (i.e., servo control, or DC motor control), the PWMPAL lets the programmer/engineer specify the "shape" of the PWM output. With this flexibility comes a bit of responsibility, but it's not difficult to do and the benefits of a little effort are certainly worth it.

Making Waves

Before we get to programming the PWMPAL, let's review a couple of small points about PWM waveforms: period, frequency, and duty-cycle.

Figure 100.2 shows the aspects of standard PWM output waveform. In the PWM cycle, the output will be on for some amount of time, then off for some amount of time (note that this is different than the Stamp's PWM function which is designed to synthesize sine waves through an RC filter).

Figure 100.2: Standard PWM Output Waveform



The total time for the waveform, on-time plus off-time, is the period. If we divide the period into one, we'll get the frequency (in Hertz) of the waveform. Finally, the duty-cycle of the wave is the ratio of on-time to period.

Here are the essential formulas (and their derivatives) that we'll work with when programming the PWMPAL:

$$\text{Period (s)} = \text{On-Time} + \text{Off-Time}$$

$$\text{Frequency (Hz)} = 1 / \text{Period}$$

$$\text{Duty Cycle (\%)} = \text{On-Time} / \text{Period} * 100$$

So how do we control these things with the PWMPAL? Each PWMPAL channel has two, 16-bit timers; one for the on-time, another for the off-time. For each count in these timers, we get 25 microseconds. Knowing this, we can calculate the proper values for a waveform we wish to create.

Let's say, for example, we want to control a servo and start with it centered. Knowing what we know about servos, we need the on-time to be 1.5 milliseconds and the off-time to be 20 milliseconds. The rest is simple math: 1.5 milliseconds divided by 0.025 milliseconds (25 microseconds) is 60. This will be our on-time count. Next we take 20 milliseconds and divide by 0.025 milliseconds to get 800.

The next step is to send these counts to the PWMPAL. Part of the PWMPAL design is the fixed serial connection that is on P0 of the BASIC Stamp. You'll have to keep this in mind as when you're designing the PWMPAL into your projects. What we're going to do, then, is send the counts using the PWMPAL's version of the Parallax AppMod protocol. The serial data string always starts with "!", is followed by the device identifier "PWM", then the command. The command for sending "motor" counts is going to be "M", followed by the motor number.

The PWMPAL's outputs correspond to Stamp pins P12 ("M1") through P15 ("M4"). If we want to connect the servo to P15, here's the command to send the centering counts:

```
SEROUT 0, 6, ["!PWMM4", 60, 0, 32, 3]
```

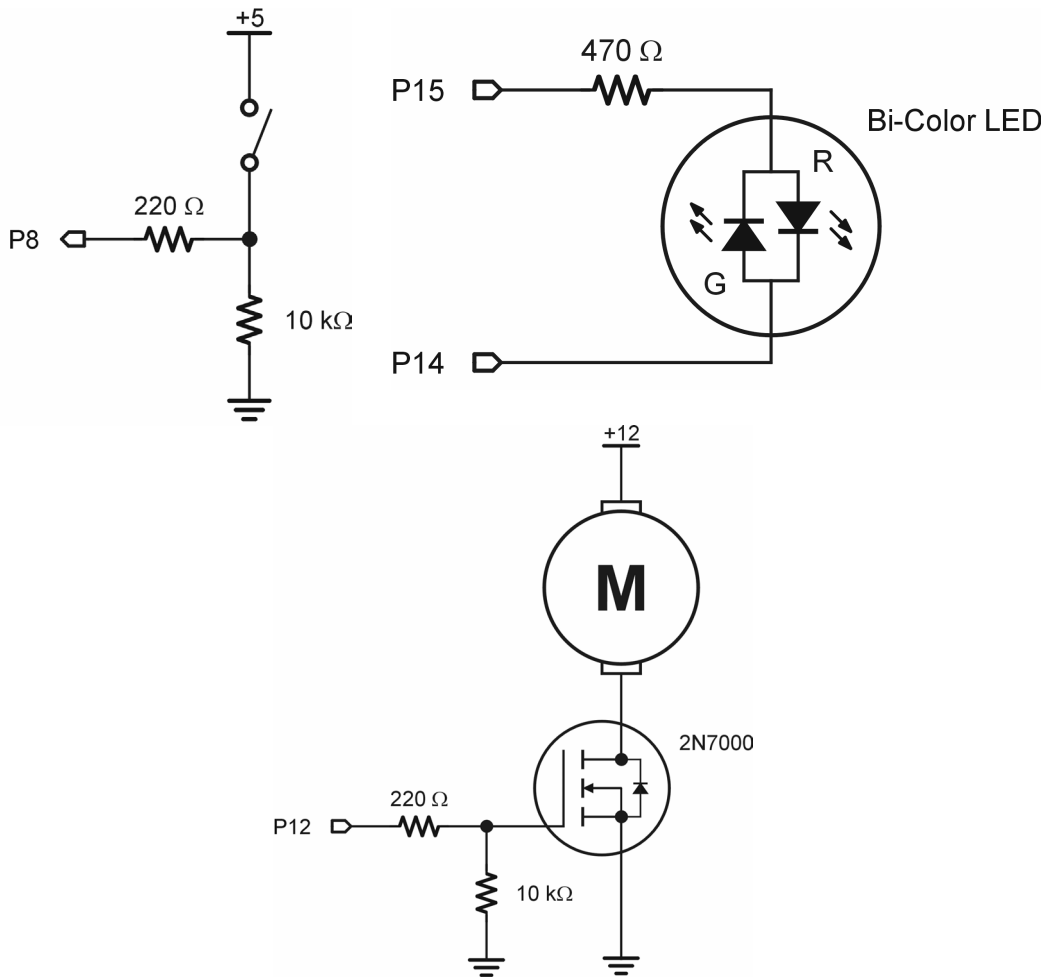
Huh? Why four bytes when we just have two counters? Remember, the counters are 16 bits wide and SEROUT works with bytes. Normally, what we're going to do is used Word variables for the counters and get some help with LOWBYTE/HIGHBYTE or BYTE0/BYTE1 modifiers. And before I forget, I need to mention that the PWMPAL will automatically detect baud rates of 9600, 19.2K and 38.4K.

Since we're a learn-by-doing crowd, let's actually hook it up and make it do something – our example will use fairly-simple parts and demonstrate nearly all the capabilities of the PWMPAL. After playing with this demo, you should be able to make the PWMPAL work for you in just about any application. What I've done is taken the separate projects from the PWMPAL docs (yes, I wrote them) and created a super project. The idea is to show you the possibilities and let your imagination run wild from there.

The circuits for this demo are pretty easy: an active-high push-button input, a buffer circuit (MOSFET) for a small DC motor, and a bi-color (red-green) LED. We'll demonstrate the PWMPAL by setting P8 as a control channel for the motor; when the switch is pressed, the motor will run – this is all under control of the PWMPAL and does not burden the BASIC Stamp. Since

P8 is an input, the BASIC Stamp program can monitor it and we'll do that to ramp the motor speed. Finally, we'll demonstrate the phase control of the PWMPAL that will let us send an AC waveform to the LED, making it appear yellow by switching quickly between red and green. This all sounds like quite a lot, but as you'll see, it's very easy to do.

Figure 100.3: Example PWM Pal Control Circuitry



Rev It Up

Author's Note: Please download the PWMPAL documentation (from the Parallax web site) for reference to the various commands as we work our way through this example.

This program requires a bit of setup before we get to the heart of it. The purpose of this setup section is to enable a counter channel on P8, clear that counter, then enable the motor output on P12 and set P8 as its control input. As you can see, P8 serves two purposes: 1) it controls the operation of the motor and, 2) it serves as a counter. What we'll be able to do, then, is to count the number of times we run the motor.

Here's the code:

```
Setup:
  SEROUT PpPin, PpBaud, ["!PWMSP", %00000001]
  SEROUT PpPin, PpBaud, ["!PWMX1"]
  SEROUT PpPin, PpBaud, ["!PWMSS", %00010001]
```

The first line enables the counter on channel 1 (P8). Later, we'll resend the phase/counter byte to enable the LEDs; for the moment we want them off. Next we'll make sure that the counter is clear by sending the "X1" command and, finally, we'll enable the motor under hardware control by setting the appropriate bits in the status/control byte. Note that any disabled "motor" outputs float so these pins can be used by the BASIC Stamp for other functions.

Since the main loop of the program is fairly large, we're going to break it up into logical sections. The first section monitors the button input and adjusts the motor speed while the button is being pressed. Remember, the PWMPAL actually activates the motor when this input is high; we're using the BASIC Stamp to update the motor's speed.

```
Main:
  DO
    IF (RunMotor = Yes) THEN
      IF (mSpeed < 100) THEN
        mSpeed = mSpeed + 1 MIN MinSpeed
        GOSUB Set Motor Speed
        update = Yes
      ENDIF
    ELSEIF (mSpeed > 0) THEN
      mSpeed = 0
      update = Yes
    ENDIF
```

As you can see, this first section is pretty straightforward; we monitor the switch input and when pressed, we compare the current speed value to 100. If the speed is less than 100, we'll increment

the speed and send the new value to the motor. We'll also set variable update so that the screen will reflect then new speed.

Updating the motor speed is a fairly simple matter of using the speed value as the on-time count and subtracting this count from 100 for the off-time.

```
Set_Motor_Speed:
  IF (mSpeed < 100) THEN
    tOn = mSpeed                ' set duty cycle
    tOff = 100 - mSpeed
  ELSE
    tOn = $FFFF                ' full on for 100%
    tOff = $0001
  ENDIF

  SEROUT PpPin, PpBaud, ["!PWMM1",
                        tOn.BYTE0, tOn.BYTE1,
                        tOff.BYTE0, tOff.BYTE1]

  RETURN
```

The exception, of course, is for 100% as we cannot have a zero off-time. What we'll do is "cheat" a bit and set the on-time value to the maximum (\$FFFF) and the off-time to the minimum (\$0001). What we actually end up with is a duty cycle of 99.9985% -- that should be close enough to 100% to run the motor at full speed.

Now, if the button isn't pressed and the current speed is greater than zero (as when the button is first released), we'll set the mSpeed value to zero and set the update variable. Notice that we don't have to send the zero speed to the PWMPAL. The reason is that we've enabled hardware control of the motor, so it will stop as soon as we release the button.

The next section of the main loop retrieves the counter channel from the PWMPAL. Like PWM control, counting happens without intervention of the Stamp; we simply need to retrieve the count when required.

```
SEROUT PpPin, PpBaud, ["!PWMC1"]
SERIN  PpPin, PpBaud, [runCount.BYTE0, runCount.BYTE1]
```

The first thing we have to do is tell the PWMPAL to send a counter value back to the BASIC Stamp. Immediately following this command, we'll use SERIN to capture the counter value – low byte, then high byte.

The final section will handle the update variable and serves two purposes: it will display the current motor speed and cycle count, and it will update the LED modulation based on the current motor speed.

```
IF (update) THEN
  DEBUG HOME,
    "Speed.... ", DEC mSpeed, CLREOL, CR,
    "Cycles... ", DEC runCount, CLREOL

  LOOKDOWN mSpeed, <=[24, 80, 95, 100], ledState
  IF (ledState <> lastLed) THEN
    ON ledState GOSUB Led_Off, Led_Green, Led_Yellow, Led_Red
    lastLed = ledState
  ENDIF
  update = No
ENDIF
PAUSE 100
LOOP
```

The DEBUG section needs no explanation. I will remind you, however, that with PBASIC 2.5 you can spread comma-delimited lists across multiple lines. This is an especially good idea when using DEBUG, SEROUT and SERIN as these functions are [internally] complicated and take a lot of code space. By using one DEBUG statement instead of three, we save EEPROM space that may come in handy later.

The final section of the main loop updates the LED. I like LEDs and I am particularly fond of bi-color LEDs because they can be so useful. By connecting the PWMPAL, we can actually get three colors out of a two-color LED. How does that happen?

The bi-color LED is actually a red LED and green LED that are connected back-to-back in the same package. When current flows one direction through the leads, the red LED will light. When the current is reversed, the green will light. If we can manage to switch the two back-and-forth very rapidly, our eyes will mix the colors and we'll perceive yellow. It's a cool trick.

One of the unique features of the PWMPAL comes into play to make this happen: the ability to set the phase of the outputs. What this is actually doing is allowing us to tell the PWMPAL to start on its high phase or its low phase. With careful programming, we can set two pins to run at opposite phase and the same frequency and duty cycle to create a TTL AC waveform. I'm going to use it to modulate the LED, but my buddy Chuck (who created the PWMPAL) has used this ability with suitable buffering to provide AC to a specialized sensor.

Back to the code. A LOOKDOWN table uses the current motor speed to set the value of ledState. If ledState has changed since the last loop through, we'll call the appropriate update routine. The first is simple, it extinguishes the LED by disabling both its PWM outputs.

```
Led Off:
  SEROUT PpPin, PpBaud, ["!PWMSS", %00010001]
  RETURN
```

The next routine works very much like the 100% motor update by setting the green LED on nearly all of the time and the red LED off nearly all of the time. The LED will look green – you'll never see the one cycle blip of red. Notice that the phase bits are set opposite of each other with the M3 output starting on its high cycle. The red LED control code is identical; the counter values and phase controls are simply reversed.

```
Led Green:
  SEROUT PpPin, PpBaud, ["!PWMM3", $FF, $FF, $01, $00]
  SEROUT PpPin, PpBaud, ["!PWMM4", $01, $00, $FF, $FF]
  SEROUT PpPin, PpBaud, ["!PWMSP", %01000001]
  SEROUT PpPin, PpBaud, ["!PWMSS", %11010001]
  RETURN
```

Making the LED look yellow is a not tough, but just a tad trickier than you might think at first. When I first wrote this program, I set the duty cycle to 50% -- it didn't look yellow at all; in fact, it looked down-right red. The reason is that red and green LEDs are made from different materials and given the same current, the red LED will light a bit brighter. I experimented until I found a satisfactory duty cycle that caused the LED to look yellow. It turned out to be about 18%.

```
Led Yellow:
  SEROUT PpPin, PpBaud, ["!PWMM3", $12, $00, $04, $00]
  SEROUT PpPin, PpBaud, ["!PWMM4", $04, $00, $12, $00]
  SEROUT PpPin, PpBaud, ["!PWMSP", %01000001]
  SEROUT PpPin, PpBaud, ["!PWMSS", %11010001]
  RETURN
```

Time To Get My Robots Runnin'

Well, that's about it. As you've just seen, the PWMPAL is pretty easy to deal with and at the same time offers up quite a bit of flexibility. Keep your eyes on the Parallax web site for application notes – like every new product, people will find interesting things to do with it over time and we'll certainly make that information available to everyone.

Finally, some of you may wonder why I did such a simple interface to the motor, and didn't go into H-Bridges and all that kind of circuitry. I didn't because my old pal Scott Edward already did – back in the January 1997 issue. Don't have it? No problem, you can order reprints of this column in "The Nuts & Volts of BASIC Stamps" book series from Nuts & Volts or Parallax. And if you really don't want to have a book handy (which would be silly, of course), you can still download electronic reprints of the column from the Parallax web site.

Have fun with the PWMPAL -- I've got a couple robots to update with it and I'd better get started. One of them uses a Sony PlayStation controller as a leash. I've you liked Aaron Dahlen's neat article on PSX controller interfacing in June, then be sure to tune in next month as I will be showing you some "inside tricks" that will help you get the most out of it.

Until then, Happy Stamping!

```

' =====
'
'   File..... PWM_Pal_Demo.BS2
'   Purpose.... PWM Pal Demo Program
'   Author..... Jon Williams
'   E-mail..... jwilliams@parallax.com
'   Started....
'   Updated.... 19 JUN 2003
'
'   {$STAMP BS2}
'   {$PBASIC 2.5}
'
' =====

' -----[ Program Description ]-----
'
' Demonstrates the features of the Parallax PWM Pal co-processor.
'
' NOTE: Do not connect DC motors directly to the Stamp/PWM Pal.  A buffer
'       (transistor, MOSFET, etc.) must be used to handle motor currents.
'
' -----[ Revision History ]-----
'
' -----[ I/O Definitions ]-----
'
PpPin          PIN      0              ' PWM Pal Serial I/O
RunMotor       PIN      8              ' motor run input
'
' -----[ Constants ]-----
'
T9600          CON      84
T19200         CON      32
T38400         CON      6
PpBaud         CON      T38400
'
Yes            CON      1              ' for active-high
No             CON      0
'
MinSpeed       CON      25              ' minimum DC to spin motor
'
' -----[ Variables ]-----
'
mSpeed         VAR      Byte           ' duty cycle (0 - 100%)
tOn            VAR      Word           ' pwm timing
tOff           VAR      Word           ' pwm timing

```

Column #100: Get Your Motor Runnin'

```
runCount      VAR    Word      ' motor run cycles
update        VAR    Bit       ' update DEBUG screen?
ledState      VAR    Nib       ' LED color
lastLed       VAR    Nib       ' last LED color

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

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

Setup:
  SEROUT PpPin, PpBaud, ["!PWMS", %00000001] ' enable counter 1
  SEROUT PpPin, PpBaud, ["!PWX1"]           ' clear counter 1
  SEROUT PpPin, PpBaud, ["!PWSS", %00010001] ' enable motor & ctrl

  update = Yes
  lastLed = 5

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

Main:
  DO
    ' check motor control
    '
    IF (RunMotor = Yes) THEN ' button pressed?
      IF (mSpeed < 100) THEN
        mSpeed = mSpeed + 1 MIN MinSpeed ' update speed
        GOSUB Set_Motor_Speed
        update = Yes
      ENDIF
    ELSEIF (mSpeed > 0) THEN
      mSpeed = 0
      update = Yes
    ENDIF

    ' get button counter
    '
    SEROUT PpPin, PpBaud, ["!PWC1"]
    SERIN  PpPin, PpBaud, [runCount.BYTE0, runCount.BYTE1]

    ' update screen
    '
    IF (update) THEN
      DEBUG HOME,
        "Speed.... ", DEC mSpeed, CLREOL, CR,
        "Cycles... ", DEC runCount, CLREOL

      ' modify LED for speed
      '

```

```

    LOOKDOWN mSpeed, <=[24, 80, 95, 100], ledState
    IF (ledState <> lastLed) THEN
        ON ledState GOSUB Led Off, Led Green, Led Yellow, Led Red
        lastLed = ledState
    ENDIF
    update = No
ENDIF
PAUSE 100                                ' ramp/scan delay
LOOP

```

```

END

```

```

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

```

```

Set Motor Speed:

```

```

    IF (mSpeed < 100) THEN
        tOn = mSpeed                                ' set duty cycle
        tOff = 100 - mSpeed
    ELSE
        tOn = $FFFF                                ' full on for 100%
        tOff = $0001
    ENDIF

```

```

    SEROUT PpPin, PpBaud, ["!PWMM1",
                           tOn.BYTE0, tOn.BYTE1,
                           tOff.BYTE0, tOff.BYTE1]
    RETURN

```

```

Led_Off:

```

```

    SEROUT PpPin, PpBaud, ["!PWMSS", %00010001]
    RETURN

```

```

Led_Green:

```

```

    SEROUT PpPin, PpBaud, ["!PWMM3", $FF, $FF, $01, $00]
    SEROUT PpPin, PpBaud, ["!PWMM4", $01, $00, $FF, $FF]
    SEROUT PpPin, PpBaud, ["!PWMSP", %01000001]
    SEROUT PpPin, PpBaud, ["!PWMSS", %11010001]
    RETURN

```

```

Led_Yellow:

```

```

    SEROUT PpPin, PpBaud, ["!PWMM3", $12, $00, $04, $00]
    SEROUT PpPin, PpBaud, ["!PWMM4", $04, $00, $12, $00]
    SEROUT PpPin, PpBaud, ["!PWMSP", %01000001]
    SEROUT PpPin, PpBaud, ["!PWMSS", %11010001]
    RETURN

```

Column #100: Get Your Motor Runnin'

```
Led_Red:
  SEROUT PpPin, PpBaud, ["!PWMM3", $01, $00, $FF, $FF]
  SEROUT PpPin, PpBaud, ["!PWMM4", $FF, $FF, $01, $00]
  SEROUT PpPin, PpBaud, ["!PWMSF", %10000001]
  SEROUT PpPin, PpBaud, ["!PWMS", %11010001]
  RETURN
```