Stamp Applications no. 13 (March '96):

When Good Luck is not Enough: Watchdogs and Error Recovery

Catching and Correcting Operating Errors And a few Bits of Boolean Logic,

by Scott Edwards

THIS is the thirteenth installment of *Stamp Applications*, so it's a perfect opportunity to talk about bad luck. In the microcontroller world, misfortune can take the form of garbled communications, power glitches, software bugs, hardware malfunctions, stuck motors, bad solder joints, electrostatic zaps, crazed users, and unread user's manuals. Some of these we can prevent; some we can control; and some we can neither prevent nor control, but only worry about.

We'll start our exploration of bad luck with a class of applications known as "watchdogs." These are circuits that monitor computers or processes for a telltale sign of proper operation. If they don't find it, they attempt to fix the problem, or to notify someone who can. A watchdog can restart a stuck PC, cut power to a stalled motor, or signal a technician for help.

I'll also show you how to use the BS2's serial timeout feature to resend data if a peripheral (or other BS2) doesn't respond within a reasonable amount of time. And since unexpected resets are often a symptom of hardware problems, I'll demonstrate a simple method of detecting them.

Watchdog Timer. Many microcontrollers, including the PBASIC interpreter chip used in the Stamp and Counterfeit, include a watchdog timer. This is a circuit that periodically increments (adds 1 to) a counter. When the counter is full and increments one more time, it overflows. This causes the PBASIC chip to reset, almost as if the reset button were pushed. (I say almost, because the chip can tell the difference between a watchdog reset and a hardware reset.)

The chip itself has no control over the watchdog timer. It cannot stop it from incrementing the counter. But it can clear that counter to 0, making the watchdog start all over in its march toward a reset.

If the chip resets the timer to 0 often enough, the watchdog reset never occurs.

What good is this? Let's take an example from desktop computers. With just a glance at the instructions, you install the latest software on your PC. You fire it up, and a program screen appears. So far, so good. Then... nothing. Press a key, move the mouse; nothing. Your PC is locked up. Sigh. Press CNTRL-ALT-DELETE to reset and try again.

In many control applications, there may not be a human standing by to decide when a program is locked up, so a watchdog timer serves the same purpose. The program is written in such a way that normal operation prevents the watchdog timer reset—but if the controller starts operating abnormally and forgets to clear the counter, the watchdog resets it.

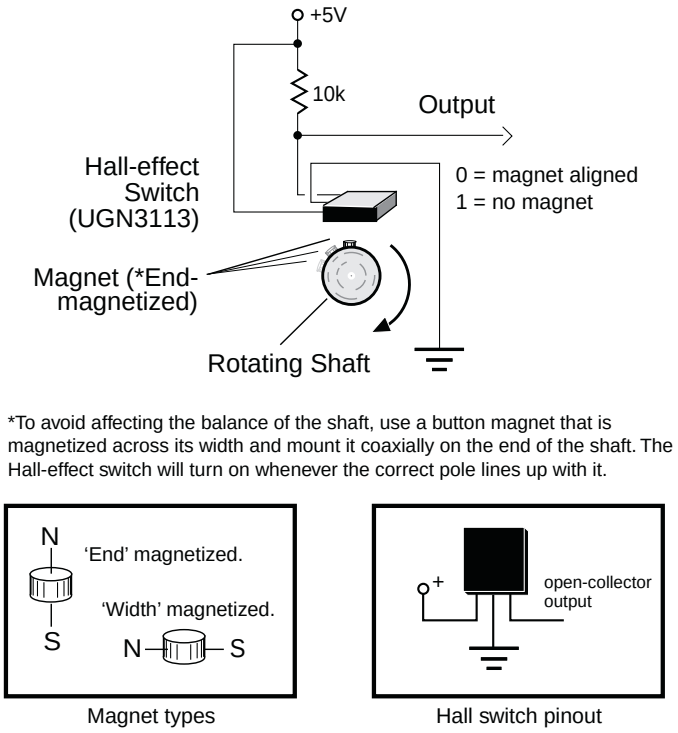

Figure 1. A PBASIC watchdog timer can monitor a motor via a Hall-effect switch like this.

PBASIC handles all the details of its own watchdog timer automatically, so you don't have to worry about it. But it can be useful to mimic the operation of the watchdog in your own programs.

For example, a reader asked me how to monitor a motor to ensure that it wasn't stalled. I suggested the watchdog approach illustrated in listing 1.

The program is set up as a big loop. Each trip through the loop PBASIC checks an input from a switch that is periodically pulsed by the rotation of the motor. Figure 1 shows a suitable magnetic-switch arrangement.

When the Button instruction detects a switch closure, the program clears the watchdog counter. Otherwise, the program increments the watchdog counter. If the watchdog counter exceeds a preset value (determined by trial runs), the program judges that the motor is stalled, and hollers for help. The program in listing 1 expects at least 1 pulse per second to prevent the watchdog from barking.

Other readers report using a similar method to monitor and reset PCs used in remote locations. They program the PC to periodically pulse an output line, such as one of the bits of the parallel port. Or they tap into the hard-drive activity light, if the application that's running uses the hard drive on a regular basis. The Stamp or Counterfeit monitors the pulse output. If no pulse arrives within before the watchdog counter reaches the maximum value set by the program, the Stamp or Counterfeit resets the PC, either by closing a relay across the reset button, or by opening a relay between the PC and its ac power source. (The latter approach seems a bit brutal to me, but reports from the field are that it's effective.)

Serial Retry. The BASIC Stamp 2 (BS2) has many new instructions and improvements to existing instructions. One such improvement is the addition of a timeout feature to the serial-input instruction Serin. It allows you to specify a number of milliseconds (up to 65535; over a minute) for the Stamp to wait for serial data. If the data doesn't arrive within that time, the program goes to a routine that you specify.

So, where the older Stamp could get stuck

waiting for serial input, the new one can unstick itself.

At first blush, I didn't see this feature as being terribly useful, since it addresses a symptom rather than the underlying problem. A corollary to Murphy's Law says that if you give up waiting for data now, the data you expected will arrive one millisecond later!

However, there's a common, real-world situation in which the serial timeout *can* be helpful. In two-way conversations between the BS2 and a serial peripheral or other Stamp, the timeout can provide a way to resend an instruction if the peripheral doesn't respond.

For example, take the Stamp Stretcher 1B. This board lets a BS1 or BS2 access 16 additional I/O lines through a 1-wire serial hookup at 2400 or 9600 baud. It also has a single-channel, eight-bit analog-to-digital converter (ADC).

To use the ADC, the Stamp sends the instruction "A" to the Stretcher with Serout, then waits for a response with Serin. Fine and dandy, unless the serial transmission got garbled and was received as something other than "A." The Stretcher is programmed to ignore characters that aren't in its command set, so the Stamp could end up waiting for a reply that isn't going to come.

If the garble was caused by some noise on the serial line, then it makes sense to try again by resending the instruction. Listing 2 shows how.

The retry loop of listing 2 offers all sorts of opportunities for customization. You could add a counter to the loop that would try three times, then flash an error light. Or continue the program without the analog data. Or connect to a modem and send a message to a human maintenance technician.

That last one brings up a good point—the serial-retry concept would be especially useful in dealing with error-prone communications over the phone line. If you wrote a program to interact with a remote computer, online service or bulletin-board system, chances are good that the connection would eventually fail. The serial timeout would allow the BS2 to try again later.

Reset Lockout. Many different hardware problems can cause unintended resets of the PBASIC chip. Any glitch that takes the 5-volt power supply lower than 4 volts, even for a fraction of a second, can cause a reset. Voltage transients that disrupt the relationship of ground to the +5-volt rail, or that put noise on the reset line, can also reset the chip. Of course, a curious finger poking the reset button will do it every time!

There are lots of applications in which you want to prevent the Stamp from resetting, because a reset would disrupt a process, reinitialize variables, or cause the loss of data. Elsewhere in this issue is my BS2 Data Collection Proto Board. I set it up to gather data on battery voltage and ambient temperature to get a general idea about battery longevity and generate sample data for the article. If the BS2 were allowed to reset during the data-gathering period, some or all of the earlier data would have been lost. So I devised a trapdoor approach that prevented the program from starting over if reset. See listing 3.

The idea is as simple as locking a door behind you. When the BS2 is first programmed, a Data directive writes 0 to address 0 of the EEPROM. The program reads this location, and, if it's 0, writes 255 to it and continues with the program. If the location does not contain 0, the program halts.

When the program is downloaded to the BS2, the door is unlocked, because the downloading process writes 0 to EEPROM address 0. When the program executes, it locks the door by writing 255 to it. If the BS2 resets, it will find 255 (locked) in the EEPROM address, and will stop.

This technique is also open to modification. In the data-logging application, I added a menu item that allowed the user to unlock the program manually. Another variation would be to use a switch instead of a byte of the EEPROM to lock out resets. The user would turn the device on, then move the switch from run to stop. The BS2 would check the run/stop switch at the beginning of the program, and continue only if it was in the run position.

In either case, a flashing light or buzzer could signal a reset that occurred after lock out.

BASIC for Beginners. Last month, we looked at the logic of IF/THEN instructions. We saw that PBASIC can look at a relationship such as "x < 10" and, if the current value of x makes that statement true, can go to a specific line in the program. In other words, PBASIC can make *decisions*.

We also touched on the fact that PBASIC can combine relationships using AND and OR to make decisions about more complicated matters. This month, we're going to look at the rules that govern this kind of logic. This logic not only works with IF/THEN decisions, it also lays the foundations for *bit manipulations*—efficient shortcuts for testing or changing the states of bits (1s and 0s).

Starting on familiar turf, let's take another look at a compound IF/THEN instruction.

```
IF pin1 = 1 OR pin2 = 1 THEN Alarm
```

So, if either of those comparisons— "pin1 = 1" or "pin2 = 1" is true, then the alarm goes off. Suppose we needed to make a table of all the conditions that could set off the alarm. It would look like this:

pin1 = 1	pin2 = 1	Alarm
FALSE	FALSE	OFF
FALSE	TRUE	ON
TRUE	FALSE	ON
TRUE	TRUE	ON

If those pins represent switches connected to doors and windows of a house, then that logic makes good sense. You want an alarm to go off in the event that bad guys are coming through the door, the window, or both. But suppose pin1 represented the state of the door switches, and pin2 was the state of the arming switch (0=off; 1=on). You'd want the alarm to sound only if the system were armed *and* a door opened. The new IF/THEN instruction, just like the plain-English description, uses AND instead of OR:

```
IF pin1 = 1 AND pin2 = 1 THEN Alarm
```

The new table expressing all possible states of the pins and alarm would be:

pin1 = 1	pin2 = 1	Alarm
FALSE	FALSE	OFF
FALSE	TRUE	OFF
TRUE	FALSE	OFF
TRUE	TRUE	ON

If AND and OR were useful only with IF/THEN instructions, they'd still be darn useful. But IF/THEN is only the tip of the iceberg. If you're just now encountering the power of logical operators like AND and OR for the first time, you're in about the same position as someone who's first encountered the arithmetic operators + and –. Not only are there more logical operators, but there's a whole system for understanding and applying them, called Boolean logic.

I want to leave you with a thought for next time, when we'll start digging around in the Boolean toolbox in earnest: AND and OR work on expressions that have two possible values, True and False. We could represent those states with individual bits, since they also have two possible states, 1 and 0. What if there were a way to apply logical operators directly to bits, or groups of bits in PBASIC? There *is*, and some of the most powerful techniques for writing efficient programs spring from this application of logic.

NOTE:

The Stamp Stretcher 1B is available from www.robotstore.com.

NOTE: This article was originally published in 1996. The Stamp Applications column continues with a changing roster of writers. See www.nutsvolts.com or www.parallaxinc.com for current Stamp-oriented information.

Listing 1. Watchdog Monitors 1-Hz Pulse (PBASIC 1)

```
' Program: WATCHDOG.BAS (PBASIC 1 detects motor stall)
' This program implements a watchdog timer--a device that monitors
' a pulsebeat and takes action if the pulse is absent for a
' predetermined amount of time. Applications for watchdogs include
' resetting a locked-up PC or cutting power to a stalled motor.

SYMBOL   state = bit0      ' Trigger state for button command.
SYMBOL   dog = w1          ' The watchdog counter.
SYMBOL   pulse_n = 0       ' Pin number for pulsebeat input.
SYMBOL   pulse_p = pin0    ' Pin name for pulsebeat input.
SYMBOL   btn = b4          ' Workspace for button command.
SYMBOL   timeout = 300     ' Max value of "dog" before alarm.

begin:
  let dog = 0              ' Clear watchdog variable to 0.
  state = pulse_p ^ 1      ' State = inverse of pulse pin.

' In the routine below, if the pulse input changes state, the OK
' routine shows us the count in variable "dog," then clears "dog"
' by looping back to the beginning of the program. Otherwise,
' it increments dog and, if dog exceeds the timeout value, shows
' the alaram message.
watchDog:
  button pulse_n,state,0,1,btn,1,OK
  let dog = dog + 1
  if dog > timeout then alarm
Goto watchDog

alarm:
  debug "alarm!",cr        ' Dog exceeded timeout.
  goto begin

OK:
  debug dog,cr             ' Show us how high "dog" got.
  goto begin               ' Then goto beginning to clear.
```

Listing 2. Serial Retry Overcomes Communication Glitches (PBASIC 2)

```
' Program: TIMEOUT.BS2 (Demonstrate serial timeout function of BS2)
' This program demonstrates how to use the serial-input timeout
' capability of the BS2. If the BS2, interfaced to a Stamp Stretcher 1B,
' does not receive a response to an analog-conversion request within
' 1 millisecond, it displays the message "Timeout" on the PC debug
' screen. If the Stretcher does return the data in time, the BS2
' does not execute the error code, but displays the ADC result on the
' debug screen. In a real program, the error handler would probably be
' more elaborate--tracking the number of retries, lighting a warning
' light, sounding a buzzer, reinitializing the Stretcher, etc. Since
' communication errors are relatively rare under normal circumstances,
' you can unhook the serial connection between the BS2 and Stretcher
' while the program is running to demonstrate the error routine.

N96N     con     $4054     ' Set 9600 baud, inverted, no parity.
result   var     byte      ' Store ADC result in this byte.
maxtime  con     1         ' Allow this many milliseconds for reply.
comPin   con     0         ' Connect this pin of BS2 to Stretcher "S" pin.

serout comPin,N96N,["***"] ' Reset the Stretcher.
again:                       ' Endless loop.
  pause 1000                 ' Wait a second between tries.
  serout comPin,N96N,["A"] ' Send (A)nalog request.

' The line below is the key to the program. It waits "maxtime" milli-
' seconds for serial start bit from the Stretcher. If the data doesn't
' arrive in time, it sends the program to the label "error." If the
' data does arrive, it stores it in the variable "result."

  serin comPin,N96N,maxtime,error,[result]      ' Get response.

  debug ? result           ' Display result.
goto again                   ' Do it again.

error:                     ' Serin timeout occurred: show error message.
  debug "Timeout",cr       ' Display "Timeout" on PC debug screen.
goto again                 ' Try again.
```

Listing 3. EEPROM Flag Locks Out Resets (PBASIC 2)

```
' Program: NO_RESET.BS2
' This program illustrates a method for letting a program detect the
' fact that the BS2 has reset since programming. This can be helpful
' in situations in which an unintended reset might cause loss of data,
' damage to equipment, etc. Note that an actual program should include
' some method for clearing the EEPROM reset flag other than just the
' data statement. Otherwise, the reset that occurs when a program is
' loaded, followed by the reset that occurs when the BS2 is disconnected
' from its programming cable, would trigger the reset-trapping routine.
' A button that causes the program to execute "write reset,0" would
' do the trick. To see the demo work, run the program. Watch the
' numbers go by on the debug screen, then press the reset button.
' The screen will display "Reset detected!"

x        var     byte      ' Variable for busy work in demo.
reset    data    @0,0      ' Write 0 to EEPROM address 0 as a flag to
                           ' indicate that the program has not reset.

Demo:
  read reset,x             ' Copy the value of reset into x
  if x = 0 then run        ' If x is 0, then the BS2 has not reset, so run.
  debug cls, "Reset detected!"
  END                      ' If x is not 0, then a reset occurred, so stop.

run:
  write reset,255          ' Record first startup of BS2

busy_work:                 ' Dummy program to show activity:
  debug ? x                ' Display value of x on PC screen.
  x = x+1                  ' Add 1 to x.
  pause 500                ' Wait a half second.
goto busy_work             ' Repeat endlessly.
```