Stamp Applications electronic version, vol. 2

This document is a collection of installments 8 through 13 of the *Stamp Applications* column published in *Nuts & Volts* magazine October 1995 through March 1996. The column provides hints, tips, and techniques for users of the popular Parallax BASIC Stamp® single-board computers and BS1 kit equivalent, the Counterfeit. The Counterfeit uses a genuine Parallax BASIC (PBASIC) interpreter chip and software licensed from Parallax. It's 100-percent compatible with the original BASIC Stamp (BS1).

I prepared this electronic version of *Stamp Applications* in response to many requests from Stamp users on Compuserve and the Internet. I plan to post additional volumes every six issues/months. Don't bother petitioning me to post more frequently; that would defeat one of the purposes of this document, which is to encourage Stamp users to subscribe to *Nuts & Volts*.

Contacting Us

Scott Edwards Electronics
PO Box 160
Sierra Vista, AZ 85636-0160 USA
ph: 520-459-4802; fax 520-459-0623
e-mail: 72037.2612@compuserve.com

Contacting Nuts & Volts magazine

T&L Publications Inc.
430 Princeland Court
Corona, CA 91719
ph: 909-371-8497; fax: 909-371-3052
subscription order line: 800-783-4624

Contacting Parallax

Parallax, Inc.
3805 Atherton Road, #102
Rocklin, CA 95765 USA
ph: 916-624-8333; fax 916-624-8003; BBS: 916-624-7101
e-mail: info@parallaxinc.com; file transfer via Internet: ftp.parallaxinc.com

Topics in Volume 2

Rotary Encoders and Header-Post Jumpers 8
Exterminating Common Bugs 9
Interfacing MAX7219 LED Driver 10
Crystal Controlled Precision Timer 11
Model Rocket with BS1-IC Instrumentation 12
Watchdogs and Error Recovery 13

Stamp Applications no. 8 (October '95):

Rotary Encoders Help You Program a Friendly Spin-and-Grin Interface

A Digital Dial plus Header-Post Jumpers, by Scott Edwards

A SAD consequence of the digitization of electronics has been the gradual disappearance of knobs. *Digital* equipment likes its input in terms of *digits*, not twists of the wrist.

Knobs are elegant and intuitive. Twist clockwise and the parameter being controlled increases; counter-clockwise and it decreases. Natural as breathing. Thankfully, the user-interface used by the largest number of people and requiring the most precise and reflexive control, the steering wheel of a car, is a big knob.

Knobs are battling back from the brink of extinction, though, now that digital versions called *rotary encoders* are being used in more designs. In this month's column, I'll show you a simple method for converting the output of a standard rotary encoder into data your Stamp can use. In the second half of the column, I'll introduce you to a technique for making tidy wiring harnesses that connect to those ubiquitous square header posts (like the pins on the Stamp board).

Rotary Encoders

The rotary encoders we're going to discuss today are properly known as "incremental rotary encoders." There are others known as "absolute" encoders. An absolute encoder outputs a binary value that's proportional to the angle of the shaft, much the same as an analog pot's resistance depends on its shaft angle. The resolution of an absolute encoder is a function of the number of output bits: An eight-bit encoder breaks a full rotation (360 degrees) into 256 parts; 10 bits, 1024 parts; 12 bits, 4096 parts.

An incremental encoder is different. It has only two output bits, regardless of its angular resolution. As the shaft rotates, the bits change in the sequence shown in figure 1. The encoder's resolution determines how far you must turn the encoder shaft before there is a change in one of the outputs.

Given the output bits shown in the figure, it's fairly easy for a controller to figure out the direction of the encoder shaft's rotation. Let's say that the bits are now 01 and they change to 00. Figure 1 shows that as clockwise rotation. If the bits start at 01 and change to 11, that's counterclockwise.

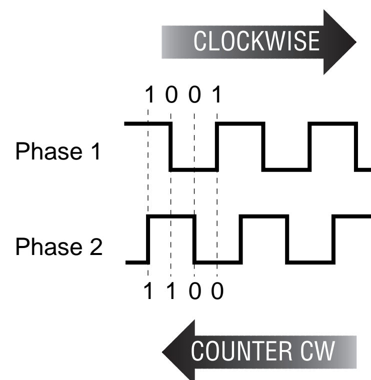


Figure 1. The sequence of bits appearing at the outputs of a rotary encoder tells the direction of rotation.

So, one way to interpret the output from an encoder would be to look up the sequence in a pair of tables. However, there's an efficient shortcut method using the exclusive-OR (XOR) operator (expressed as ^ in PBASIC). XOR's effect on bits can be stated as, "If bit A or bit B (but not both) = 1, then bit C = 1." In table form:

A XOR B	= C
0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

XOR has quite a few uses in programming. Any bit XORed with 1 is inverted; bits XORed with 0 are unchanged. If two bits are equal they XOR to 0; if not equal they XOR to 1.

In the case of the encoder sequence, it turns out that for any given sequence, XORing the righthand bit of the old value with the lefthand bit of the new value tells you the direction of rotation. For example, take the clcokwise sequence 01 00: 1 XOR 0 = 1. Now the counter-clockwise sequence 01 11: 1 XOR 1 = 0. This relationship holds for any pair of numbers in either direction.

Figure 2 and listing 1 demonstrate how a rotary encoder could be employed in a user-interface application. I got the idea for it from one of my customers. He used a Backpack-equipped LCD to simulate a radio "bandspread" tuning dial. Turn the encoder to the left and the display scrolls to the left; turn it right, the display scrolls right. In my customer's application, the Stamp also updated a frequency synthesizer chip. This gave him knob-controlled tuning with the advantages of digital control and display.

Figure 3 is a photo of my breadboard setup. I used an assembled Counterfeit kit (my Stamp-compatible controller; see Sources) running at four times the normal Stamp speed (16 MHz). This allowed the program to keep up with all but the fastest spins of the dial. If the controller isn't fast enough to track every transition of the encoder's outputs, "slippage" errors occur; the display gets out of sync with the knob. So speed is important.

Because the Counterfeit was running at 16 MHz, all of its instructions were proportionately accelerated. Although the program specifies 2400 baud for serial output to the Backpack-equipped LCD, the actual baud rate is four times that, 9600 baud. I installed a jumper on the Backpack's baud-rate header to set it for 9600 baud too.

Figure 3 also provides a nice introduction to this column's second topic, making jumper wires for convenient breadboarding with the Stamp, Counterfeit and accessories. Everything in the picture is hooked up with the kind of jumpers described in the next section.

Making Connections

The Stamp, its cousin the Counterfeit, and accessories like the Stretcher and Backpack, provide 0.025-inch square metal posts for making connections to the outside world. I've slowly become aware that most users aren't quite sure what to *do* with these connectors, known as header stakes. They usually end up wire-wrapping to make their connections.

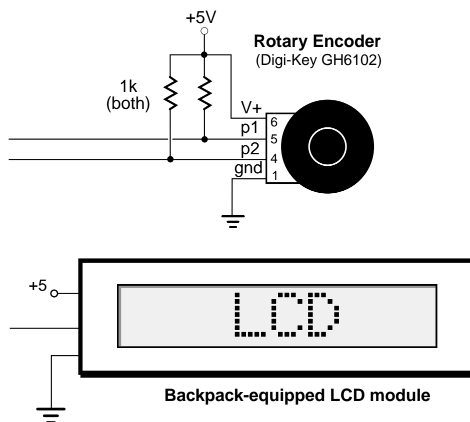


Figure 2. Connection diagram for the rotary-encoder demo.

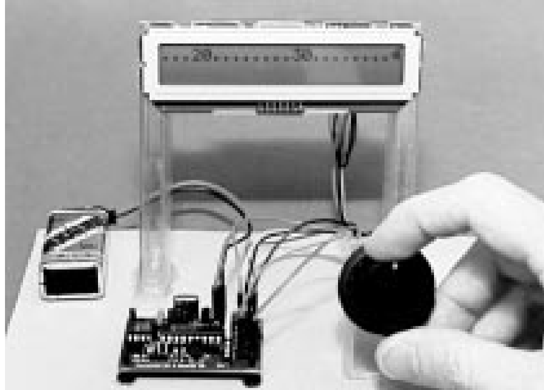


Figure 3. Photo of the rotary-encoder setup.

They also end up *un*wrapping old connections. And fixing bad connections.

There's a better way. If you're willing to invest $8 in a tool and a few bucks more in a supply of connectors, you can make slick, secure jumper wires and wiring harnesses.

Jameco (Sources) carries "female crimp pins" that are designed to fit 0.025-inch header stakes. The pins are stock number 100765 and cost a dime apiece in quantities of 10 or more. These pins are designed to fit into plastic housings to make tidy, detachable wiring harnesses of the sort you find inside PCs.

You can crimp these pin/sockets onto the ends of 22, 24 or 26-gauge stranded hookup wire with the help of Jameco's tool, stock number 99442 ($7.95). Here's how:

The crimp pins come attached to a ribbon of metal called a carrier strip. Cut a one-pin section from the end of this strip, leaving the pin attached to the piece of metal. This tab of carrier-strip metal makes a nice handle for the tiny pin. Strip 1/4 inch of insulation from the end of a piece of hookup wire. Hold the pin by the carrier tab and load the wire into the crimp pin as shown in figure 4. See the two sets of ears on the crimp pin? Line up the insulation with the back set and the bare wire with the front set.

Using the very end of the crimping tool, pinch the ears gently inward toward the wire. This ensures that it will fit into the concave crimping die. Next, position the back end of the crimp pin against the smallest tooth with the ears pointing into the die. Line the ears up with the funnel-shaped opening of the die, and squeeze the handles of the tool. When you open the crimp tool, you'll see that it has neatly wrapped one set of ears around the insulation, and the other around the bare wire. Pinch the bare-wire part of the crimp with the corners of the tips of the tool as final insurance of good contact.

You can use this connector as-is, or you can cover it with a short piece of heat-shrink tubing. If your wiring scheme permits, you can even use one of those neat plastic housings listed on the same page of the Jameco catalog as the pins. Just slip the pins into the housing until you hear a click. A latch inside the housing prevents the pin from slipping out.

One more benefit of header-socket pins: they fit perfectly on 22-gauge solid hookup wire for making temporary connections to breadboards and other circuits.

By the way, if you're a manufacturer of wire and cable goodies who could make the above-described jumper wires in the low thousands for a reasonable price, get in touch (see Sources for my contact information). I'd love to add prefabricated header-socket jumpers to my expanding line of Stamp goodies, but I haven't found anyone to make them for a decent price.

Sources

For more information on the BASIC Stamp, contact Parallax Inc., 3805 Atherton Road no. 102, Rocklin, CA 95765; phone 916-624-8333; fax 916-624-8003; BBS 916-624-7101; e-mail info@parallaxinc.com.

For female header pins and the crimping tool discussed in this column, get a catalog from Jameco Electronic Components, 1355 Shoreway Road, Belmont, CA 94002-4100; phone 1-800-831-4242.

The rotary encoder used in this application is available from Digi-Key, 701 Brooks Avenue South, PO Box 677, Thief River Falls, MN 56701-0677; phone 1-800-344-4539.

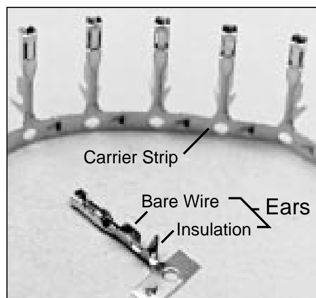
Send questions, suggestions, or requests for future Stamp Applications to: Scott Edwards Electronics, 964 Cactus Wren Lane, Sierra Vista, AZ 85635; phone 520-459-4802; fax 520-459-0623; e-mail (Compuserve) at 72037,2612; Internet 72037.2612@compuserve.com. Scott

offers Stamp-related kit goodies, including the following:

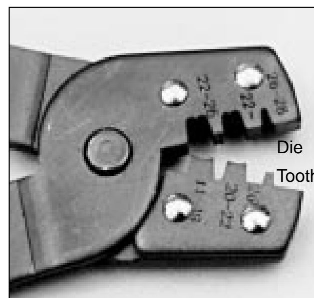
The Counterfeit controller, a kit alternative to the BASIC Stamp, is $29. Double- and quad-speed options are $2 and $4, respectively. The Counterfeit Development System, required to program Counterfeits (also for programming original BASIC Stamps, like the BS1-IC) is $69 and includes 150-page manual, downloading cable kit, Parallax software, and one Counterfeit controller kit. All Counterfeit hardware and software is 100-percent compatible with the BS1 and original BASIC Stamp.

The LCD Serial Backpack is a tiny daughterboard that attaches to 1- and 2-line LCDs to convert their fussy parallel interface to Stamp-compatible serial at 2400 or 9600 baud. The assembled Backpack is $29; with a preinstalled 16x1 LCD, $40; or with the 2x24 LCD shown in the photo, $50.

Prices are postpaid (express shipping and CODs extra). Visa, Mastercard, American Express accepted for phone/fax orders. POs accepted on approved credit. Personal checks and money orders are welcome for orders by mail.



0.025" Socket Crimp Pins



Crimping Tool

Grasp the wire and socket, using carrier-strip tab as a handle.

Use the end of the tool and gently pinch the ears to fit the die.

Put the connector into the tool with the ears facing into the die. Squeeze hard.

Done. Give the bare-wire ears a final crimp to ensure good contact, and break off the carrier-strip tab.

Figure 4. Making Stamp-compatible jumpers is easy with this step-by-step procedure.

' Program Listing Demonstrating Use Of a Rotary Encoder

```
' Program: Rotary.BAS (Read a rotary encoder and scroll a display)
' This program demonstrates the use of a digital rotary encoder
' as a unique user-interface control. Turning the knob scrolls a
' virtual tuning dial displayed on an LCD. To keep the code and
' hardware as simple as possible, the LCD is equipped with an
' LCD Serial Backpack, which interprets data and instructions sent
' to it serially.

SYMBOL old = b0               ' Previous state of encoder bits.
SYMBOL new = b1               ' Current state of encoder bits.
SYMBOL directn = bit0         ' Direction of encoder travel; 1=CW.
SYMBOL index1 = b2            ' For/Next counter variable.
SYMBOL index2 = b3            ' For/Next counter variable.
SYMBOL I = 254                ' Instruction-toggle for LCD.
SYMBOL LCD_cls = 1            ' Clear-screen instruction for LCD.
SYMBOL left = 24              ' Scroll-left instruction for LCD.
SYMBOL right = 28             ' Scroll-right instruction for LCD.

' The program starts by printing a scale on the LCD screen.
' The LCD's RAM can hold up to 80 characters and scroll them
' circularly across the display. The code below prints...
' 0........10........20........30........40........50..
' ...up to 70. Only the first 24 characters are initially
' visible on the display, but turning the encoder knob scrolls
' them into view, like an old-fashioned radio tuning dial.
pause 1000                    ' Let LCD initialize.
serout 0,n2400,(I,LCD_cls,I)  ' Clear LCD screen.
for index1 = 0 to 70 step 10  ' Scale: 0-70 (uses 80-char LCD RAM).
  serout 0,n2400,(#index1)    ' Print number on the screen.
    for index2 = 1 to 8
     serout 0,n2400,(".")     ' Print "........" between numbers.
    next
next

' Before entering the main loop, the program stores the beginning
' state of the encoder bits into the variable 'new.' It ANDs the
' pins with %11000000 in order to strip off all bits except for
' 6 and 7. (ANDing a bit with 0 always produces 0; ANDing with 1
' copies the state of the bit.)
let new  = pins & %11000000   ' Mask off all but bits 6 & 7.

start:
  let old = new & %11000000   ' Mask bits and copy new into old.
again:
  let new = pins & %11000000  ' Copy encoder bits to new.
  if new = old then again     ' If no change, try again.
  let directn = bit6 ^ bit15  ' XOR right bit of new w/ left bit of old.
  if directn = 1 then CW      ' If result=1, encoder turned clockwise.
  serout 0,n2400,(I,left,I)   ' If result=0, counterclock (scroll left).
goto start                    ' Do it again.
CW:
  serout 0,n2400,(I,right,I)  ' Clockwise (scroll right).
goto start                    ' Do it again.
```

Stamp Applications no. 9 (November '95):

Exterminating Common Bugs With Little-Known Stamp Info

Miscellaneous Tips and Techniques, by Scott Edwards

THE STAMP is so easy to use and program that I sometimes forget what a complex little computer it is. Fortunately, I have you, the readers of this column, to remind me. This month I'll present an assortment of bug fixes, explanations, hints, and tips based on questions and comments I've received by e-mail.

Programs without END

In figure 1, an LED is wired so that it lights when a low (0) appears on pin 0. Suppose you wrote a program that consisted of just the following line:

```
LOW 0 ' Light the LED.
```

What would happen? When the Stamp first turned on, all pins would be in input mode, and the LED would be dark. A fraction of a second later, the LOW instruction would execute, changing pin 0 to output/low. The LED would light.

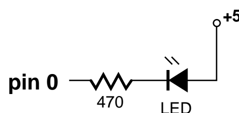


Figure 1. An implicit END instruction makes the LED blink.

After that, there are no more instructions for the Stamp to execute. With its work complete, the Stamp does the only sensible thing; it executes an END instruction and goes to sleep. During sleep, the LED stays on, but every 2.3 seconds it blinks off for a fraction of a second.

The Stamp resets every 2.3 seconds during sleep, and its pins go into input mode during reset. This reset takes about 18 milliseconds—long enough for you to see a visible blink in the LED.

This behavior can affect other devices as well. Suppose the Stamp were connected to a serial device. Every 2.3 seconds it would lose control of the pin being used for output. The glitch could cause "garbage characters" to appear at the serial receiver.

Problems caused by the implicit END instruction are rare in finished programs, since most are loops. But during casual testing it's quite common to type in a few lines to see what happens. If you want to avoid letting the Stamp execute that unwritten END, try this:

```
      LOW 0        ' Light the LED.
stop: GOTO stop    ' Freeze here.
```

Mighty Frustratin' Power Dangers

While we're on the subject of resets, let's talk about unwanted resets caused by an inadequate power supply.

The Stamp and Counterfeit both come with 9V battery clips. A built-in voltage regulator drops this input voltage to a steady 5-volt supply for the rest of the electronics.

The regulators are quite efficient. They only require an input of slightly over 5 volts in order

to maintain regulated 5 volts out. However, if the input voltage falls below their *dropout voltage*—the voltage at which they can no longer regulate the output—all bets are off.

This can happen more often than you might think, even with a battery that measures well over 5 volts. Although a battery is shown as a single component in schematic diagrams, it acts like two components, a voltage source and a series resistance. As the battery wears out, the voltage decreases *and* the series resistance increases. It's the increased series resistance that usually gets you.

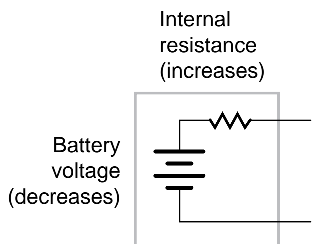


Figure 2. As a battery wears out, its internal resistance increases.

When a 9V battery is down to 7 volts, it's still well above the voltage regulator's dropout voltage, which may be as low as 5.1 volts. But the battery's internal resistance has increased to, say, 50 ohms. If your circuit draws a total of 50 mA, the voltage drop across that internal resistance becomes 0.05 x 50 = 2.5 volts. (Ohm's Law—current in amperes times resistance = voltage.) The battery delivers just 7 – 2.5 = 4.5 volts to the regulator's input. The result is that the supply is no longer in regulation, and a reset can occur.

I saw a particularly acute example of this in a recent on-line help message. A user had connected a radio-control servo (positioning motor with built-in electronics) to the Stamp, and was powering both from the same four-pack of AA penlight cells. The circuit was totally erratic. The cause was fairly obvious—whenever the batteries (only 6 volts total) tried to deliver large currents to move the servos, the large voltage drop across their internal resistance caused a brownout to the Stamp, which reset itself.

This problem is more severe with newer Stamps, which have a "brownout reset" IC on board. These reset the processor any time the regulated supply falls below 4 volts. Older Stamps had no such circuit, and could continue to operate at 2.5 volts or less. This sometimes caused problems, because different parts of the circuit shut down at different voltages, causing a kind of temporary insanity. (The Counterfeit also has a brownout circuit, which kicks in at approximately 3.4 volts.)

Here are some hints for tracking and fixing power-supply related problems:

- If a circuit behaves erratically and you suspect that varying loads are to blame, try removing those loads and rerunning the program. If it works OK without the load, you need to beef up the power supply, or use a separate supply.
- Don't draw more than 50 mA from the Stamp's built-in 5-volt supply, or 100 mA from the Counterfeit's supply.
- If your application involves motors or other high-current loads, use separate sets of batteries for the Stamp and the load. Just connect the grounds together.
- If you have multiple Stamps or other circuits running from a single power supply, don't daisy-chain the wiring; wire each module back to the power supply terminals separately.

Divide and Conquer

PBASIC can perform simple arithmetic on 16-bit positive integers. A 16-bit variable can range from 0 to 65535. What if your application involves calculations that generate numbers larger than 65535?

I recently answered an e-mail help request that illustrates the problem and a way to solve it. Dan DiLuzio was building a pH meter, a device that measures the acidity or alkalinity of a solution on a scale of 0 to 14. He had a pH probe and some analog circuitry that converted the pH to a 0- to 5-volt output, where 0 volts represented a pH of 14 and 5 volts a pH of 0.

An LTC1298 analog-to-digital converter (ADC) (described in *Stamp Applications* no. 4, June 1995) allowed the Stamp to read in the 0- to 5-volt signal as a 12-bit number from 0 to 4095. To

convert this number to a pH reading, Dan needed to use the following formula:

pH = 14000 – (ADC_result * 14000/4095)

This formula uses the number 14,000 to represent the full-scale reading of 14. Just moving the decimal point three places to the left (14.000) would scale the readings to real pH units. The trouble is that when the pH is 0 the ADC reads 4095, and the calculation ADC_result * 14000 equals 57,330,000. That's beyond PBASIC's maximum integer of 65,535.

Dan was starting to think that he'd have to abandon some of the precision offered by the 12-bit ADC, and content himself with readings of 0 to 14 or 0.0 to 14.0 at best. But I showed him how to chop the problem up into smaller pieces by factoring, preserve precision, *and* stay within the bounds of 16-bit math.

I started by factoring the constants 14,000 and 4095 into smaller integers. Since 4095 is close to 4096 (2^{12}) I cheated and changed it to the more convenient value. Here are the factors:

14,000 = 7*5*5*5*2*2*2*2
4096 = 2*2*2*2*2*2*2*2*2*2*2*2

I then rearranged Dan's formula into a series of smaller alternating multiplications and divisions. I eliminated cases in which the number would be multiplied by 2 then divided by 2, since these have no net effect. I wound up with:

ADC_result * 7 / 4 * 5 / 4 * 5 / 4 * 5 / 4

Trying this with the troublesome maximum value of 4095, I got an answer of 13,995—within a fraction of a percent of the correct answer of 14,000. And no calculation exceeded PBASIC's limit. I passed this info to Dan, along with the suggestion that he use a correction factor or calibrate the analog circuitry to eliminate the small bias introduced by fudging 4095 to 4096 and by the lost fractions due to integer arithmetic. He reported back: problem solved!

Timing NOT to Set Your Watch By

On the BS1-series Stamps, you can generate or measure pulses with 10-microsecond (µs) resolution. On the BS2, pulse resolution is 2 µs. This fine timing precision leads some users to believe that the Stamps are also very accurate. They're not, and the discussion that follows should interest both Stamp users and folks who never quite grasped the distinction between *precision* and *accuracy*.

First, you must understand that all Stamp timing is referenced to an internal oscillator whose frequency is set by a ceramic resonator. Resonators are similar in properties to quartz crystals, but are built to withstand rough treatment like shock and vibration. The tradeoff is that a mediocre crystal's actual frequency is usually within ±0.005 percent [50 parts per million (ppm)] of its nominal or rated frequency, while a good ceramic resonator may be within ±1 percent.

Putting that into perspective, a clock controlled by a 50-ppm crystal might be off by 4 seconds a day; with a 1-percent resonator the error could be close to 15 *minutes* a day!

On the BS1, the nominal 4-MHz resonator can operate at 3.96 to 4.04 MHz. A BS2's 20-MHz resonator can range from 19.8 to 20.2 MHz. (These ranges are based on the hypothetical 1-percent tolerance; some resonators are as sloppy as 3 percent.)

If a BS1 is used to measure a 200-millisecond (ms) pulse, the result should be 20,000 units of 10 µs apiece. But if the resonator is off by 1 percent, the actual reading could range from 19,800 to 20,200.

This resulted in a Stamp user calling me for help. He was trying to use a BS1 to produce pulses that were measured by a BS2 in order to simulate his final application involving motor-speed measurement. The two Stamp clocks were not only off frequency, but may have been off in opposite directions (above and below nominal), resulting in an error of close to 2 percent, and hundreds of pulse-timing units.

He was shocked, having expected that the fine *precision* of the pulse commands (units of 10 and 2 µs) also meant high *accuracy*, with results within a few units of right on the money. Nope; accuracy can't be any better than the timing reference, which is only good to ±1 percent. Bear this in mind when designing any time-critical application.

Bowlegs, Brackets and Bugs with the BS2

If you bought one of the early BS2s, you received the "preliminary documentation," a dump of just the essential information required to get started with the new Stamp. Because of some major improvements in PBASIC's structure, the BS2 dialect is not backwardly compatible with the BS1.

One change that's causing headaches for users who are accustomed to the BS1 is the new role of parentheses (like the ones surrounding this text). On the BS1, bowlegs enclose lists, such as data items for Serout, Lookup, Lookdown and Sound. The BS1 does not support the use of parentheses to change the order of math and logic operations. The BS2 does.

As a result, Parallax had to find a new way to enclose lists of data items, which might include expressions enclosed in bowlegs, and it chose square brackets [like these].

The change is sufficiently subtle that I didn't notice it on first reading of the new docs, but the host software brought it to my attention several times when I wrote my first BS2 program! I'm hoping this explanation will help you remember when to use bowlegs and when to use brackets.

Here are more tips for BS2 users:

• The serial commands now require baud rate expressed in terms of microseconds – 20. To convert a desired baud rate, divide it into 1, multiply the result by 1 million and round off, then subtract 20. Try 9600 baud: 1/9600 = 104.167×10^{-6}. Multiply by a million and round off: 104. Finally, subtract 20: 104 – 20 = 84. That's your baud-rate timing. For other baud-mode options (expressed as hex numbers in the documentation), just add them to the calculated timing value. For example, to invert the serial output at 9600 baud, use $4000 + 84.

• If you're planning to use the new X-10 remote control command Xout, or the synchronous-serial instructions Shiftin and Shiftout, make sure to get the new application notes from Parallax. And get the whole notes, not just the source code. The accompanying text and illustrations are probably more important than the programs to really understanding how these new features work.

• Having trouble downloading programs to a BS2? The problem may be that the host software can't figure out which serial port the BS2 is connected to. You can specify the port when you boot the software by adding the switch /1 for com 1 or /2 for com 2. For example:

```
STAMP2 /1
```

launches the host program for com port 1.

• The new BS2 method of reading and writing the I/O pins seems to be puzzling some people. It's probably because you no longer have predefined variables called "pin0, pin1...pin7." Under the new PBASIC, you have a pair of 16-bit variables called INS and OUTS which hold the input and output states of the pins.

To use the individual bits of these registers, you need to define bit variables, like so:

```
in_pin1  var  INS.bit1 ' Pin 1 input.
```

This may take some getting used to, but in the long run you'll appreciate the flexibility of the new approach.

• If you're using the BS2 to display data on the LCD Serial Backpack, there's a neat trick you should know. By using the new comparison feature of Lookdown and the decimal-digit function Dig, you can right-align a numeric display. See the listing for an example.

Back Issues of *Stamp Applications*

One of the biggest headaches associated with writing a monthly column is handling requests for reprints of previous installments. On one hand, it's flattering that readers who see a sample of the column want more. On the other, more practical hand, it's brutally expensive to make and mail paper copies.

Here's an information-age alternative: I have posted an electronic copy of the first seven *Stamp Applications* columns (March through September 1995) to locations on Compuserve and the Internet. You can download, read, and print these copies using your PC (under Windows) or Mac computer and free Adobe Acrobat software (version 2.0 or higher).

The file is called "ST_APPS1.PDF." It's roughly 250kB in size and contains the equivalent of 34 letter-sized pages, including all

text, drawings, and photos that appeared here in *N&V*. On Compuserve, it's located in the Hobby Electronics library of the Consumer Electronics Forum. On the Internet, it's on the Parallax ftp site, ftp.parallaxinc.com.

The free Acrobat reader is available from the Adobe forums on Compuserve; it's also on the Parallax ftp site.

For those of you who prefer the convenience of hard copy, call my order line (Sources) and plunk down $10 by credit card. We'll ship you the whole seven-issue collection printed on three-hole-punched paper for storage in a standard school binder.

I will prepare new compilations of the column and post them to those same on-line locations every six months. At the same time, I'll also make new hard copies available.

Of course, your best bet is to make sure that your subscription to *N&V* is always paid up…

Sources

For more information on the BASIC Stamp, contact Parallax Inc., 3805 Atherton Road no. 102, Rocklin, CA 95765; phone 916-624-8333; fax 916-624-8003; BBS 916-624-7101; e-mail info@parallaxinc.com.

Send questions, suggestions, or requests for future Stamp Applications to:

Scott Edwards Electronics, 964 Cactus Wren Lane, Sierra Vista, AZ 85635; phone 520-459-4802; fax 520-459-0623; e-mail (Compuserve) at 72037,2612; on the Internet 72037.2612 @ compuserve.com. Scott offers Stamp-related kits, including:

The Counterfeit controller, a kit alternative to the BASIC Stamp, is $29. Double- and quad-speed options are $2 and $4, respectively. The Counterfeit Development System, required to program Counterfeits (also for programming original BASIC Stamps, like the BS1-IC) is $69 and includes 150-page manual, downloading cable kit, Parallax software, and one Counterfeit controller kit.

The LCD Serial Backpack is a tiny daughterboard that attaches to 1- and 2-line LCDs to convert their fussy parallel interface to Stamp-compatible serial at 2400 or 9600 baud. The Backpack is $29; with a 16x1 LCD, $40.

A printed collection of the first seven installments of *Stamp Applications* is $10. See the text of the column for free, on-line sources of this material.

Prices are postpaid (express shipping and CODs extra). Visa, Mastercard, American Express accepted for phone/fax orders. POs accepted on approved credit. Personal checks and money orders are welcome for mail orders.

```
' Program: RJ_DEMO.BS2 (Right-justified printing with Stamp 2)
' This program demonstrates how to print numbers on the LCD Serial
' Backpack with right justification. This means that the ones place
' is always in the same location on the screen regardless of the
' number of digits in the number. This program uses a Lookdown table
' as a function that returns the number of decimal digits in a given
' 16-bit value.

I           con            254      ' Instruction toggle command.
ClrLCD      con            1        ' Clear-LCD instruction.
prn_at      con            140      ' Display RAM, address 13 (128+12).
j           var            word     ' 16-bit counter variable.
pos         var            byte     ' Cursor position to print at.
numDig      var            nib      ' Number of digits of number.
N96N        con            $4054    ' 9600 baud, inverted, no parity.

low 0                      ' Make the serial output low
pause 1000                 ' Let the LCD wake up.
serout 0,N96N,[I,ClrLCD,I] 'Clear the LCD.
serout 0,N96N,["Number: "] ' Print the fixed label.

' The loop below counts from 0 to 20000 and dislays the current count
' on the LCD Serial Backpack. The Lookdown table determines how many
' digits are in the current value of the count in order to position
' the printout aligned on the rightmost digit. It works by determining
' whether a given number is less than 10 (1 digit); between 10 and 100
' (2 digits), etc. The table is good for values from 0 to 65534.

Loop:
 for j = 0 to 20000        ' Count to 20,000.
  lookdown j,< [0,10,100,1000,10000,65535],numDig     ' Get # of digits.
  pos = prn_at - numDig    ' Adjust the screen position.

  serout 0,N96N,[I,pos,I,DEC j,"    "]    ' Print j at adjusted screen pos.
  pause 50                 ' Slow the count a little.
 next                      ' Keep going to 20,000.
goto Loop                  ' Do it again.
```

Stamp Applications no. 10 (December ’95):

Put Your Data Up in Lights Using an LED Display Chip

Interfacing the MAX7219 LED Driver And Part 1 of an Introduction to BASIC, by Scott Edwards

ALTHOUGH most consumer-electronic gear uses liquid-crystal displays (LCDs), military and industrial users are still in love with light-emitting-diode (LED) readouts. Balanced against LEDs’ brutally high current draw and poor contrast under bright lighting are their toughness, broad viewing angle, and wide operating temperature range. These are significant advantages that the goop-under-glass construction of LCDs cannot match.

Of course, you might prefer LEDs just for their snazzy appearance.

This month’s column will show you how to use an off-the-shelf LED driver chip to add an LED display to your Stamp projects with a minimum of hardware and software overhead.

And since so many of you have asked, I’m beginning a new feature this month—a column within a column—introducing the fundamentals of BASIC programming.

Meet the Max

The Maxim MAX7219 LED display driver is the key to creating a Stamp-friendly LED display. The chip’s features include:

- Drives up to eight 7-segment (plus decimal point) LED displays, or 64 discrete LEDs.
- Multiplexes display at high speed (more than 1200 Hz) to prevent visible flicker.
- Decodes binary-coded decimal (BCD) digits into patterns of LED segments.
- Controls current to the LEDs and provides software control over brightness.
- Permits software configuration of the display width from one to eight digits.

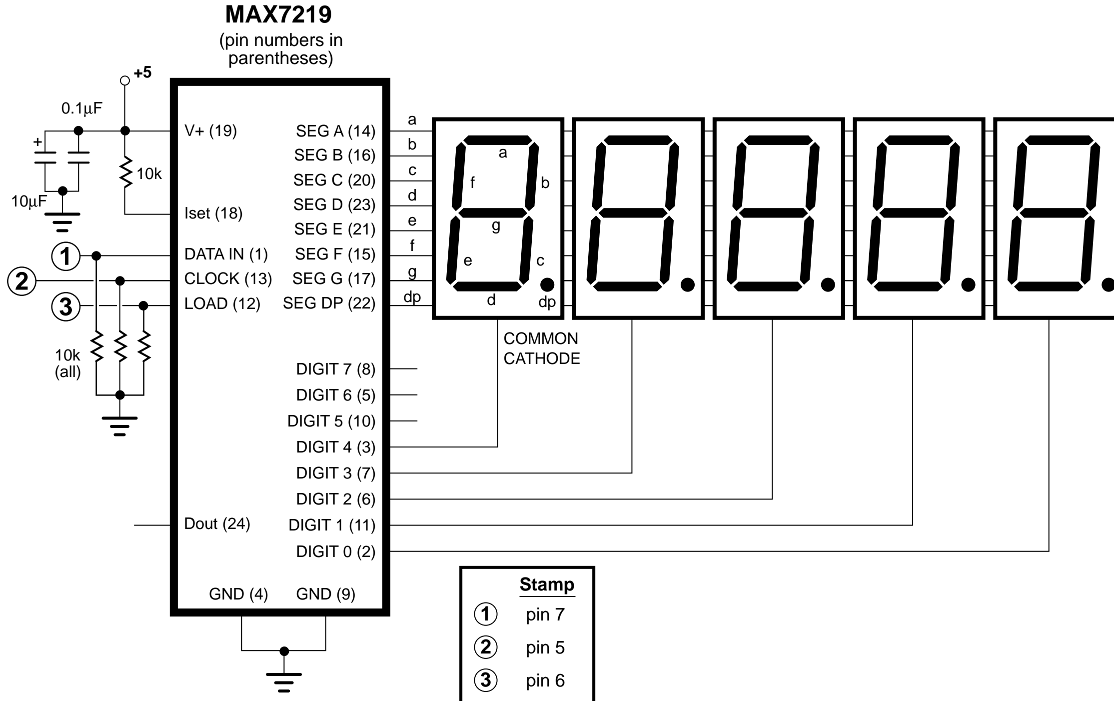
The figure and listing show how the hardware and software go together. Before you run off to build a display based on them, let’s discuss some operating principles of multiplexed LED displays.

Multiplexing Makes the Most of the Max

The MAX7219 uses a technique called multiplexing to drive 64 LEDs with just 16 output lines. How? Look at the figure. All of the displays’ individual segment anodes (the + connections of the LEDs) are connected in parallel. Within each display, all of the LED cathodes (the – connections) are tied together. This is called a “common cathode” configuration.

Imagine that the *a* segment line is connected to +5 volts, but only one display’s common-cathode lines is grounded. The top bar (segment *a*) of that one display would light up. The other displays would remain dark, lacking a complete path from +5 to ground.

Now, if you grounded a different display’s common-cathode line, its segment *a* would light.



Schematic diagram for five-digit display based on the MAX7219.

And if you switched the ground connection from one display to another 30 or more times a second, it would appear that all of the displays' *a* segments were lit at once.

It's not much of a stretch to see that a fast controller could create the impression of lighting all of the displays with different patterns of lights by rapidly switching the segment lines and scanning the digit lines. That's called *multiplexing*, and it's what the MAX7219 does.

In addition to multiplexing the displays, the MAX7219 incorporates tables that correlate the numbers 0 to 9 to their corresponding patterns of LEDs. For example, the number 3 is represented by lighting LED segments *a*, *b*, *c*, *d*, and *g*. It's normally the program's responsibility to convert digits into LED segments. However, the MAX7219 can perform this conversion for you, depending on a configuration setting. This feature saves at least a dozen bytes of PBASIC program memory in applications that use numeric displays.

Synchronous Serial Communication

The MAX7219 uses a three-wire synchronous-serial interface. We've seen these before with such peripherals as the LTC1298 analog-to-digital converter, DS1620 digital thermometer, and many others. This type of interface sends one bit at a time, just like RS-232 asynchronous serial. It differs in that it requires a separate clock pulse to tell the receiver when to grab the next data bit.

The listing shows how this process works on a BS1-type controller in the code labeled Max_out. The new BS2 controllers have a command called Shiftout that handles the whole process. In the BS2 version of the program (included with the AppKit; see Sources), the code within the For/Next loop is reduced to:

```
shiftout, Data_n,CLK,msbfirst,[temp]
```

...where *temp* is the variable containing data to be sent to the MAX7219.

Final Hardware Notes

As I mentioned at the beginning of the article, one of the reasons for preferring LCDs to LEDs is current draw. The MAX7219 provides a means of setting the segment current of the LEDs through the ISET pin. The smaller this resistor, the greater the current through each

LED segment. The value shown in the schematic—10k—sets the maximum segment current of 40 mA. If all eight segments of a particular display are lit, the current draw is 8 x 40 = 320 mA. I mention this because the voltage regulators on the Stamp products are limited to 50 mA; the Counterfeits to 100 mA. You can increase the value of the ISET resistor, but the display won't be as bright. At 60k the segment current will be approximately 10 mA, and combined maximum draw will drop to 80 mA. Depending on the LED displays you choose, this maybe bright enough.

Finally, you may be wondering about the 10k pulldown resistors on the interface between the Stamp and the MAX7219. What purpose do they serve? When a Stamp or Counterfeit resets, either when you first apply power or push the reset button, its pins are in input mode. They are effectively disconnected, so any digital inputs connected to them float. Such inputs frequently float high (logical 1, as though connected to +5V), but noise can cause them to change states at random.

During the time it takes the Stamp to come out of reset, noise on these inputs can put the MAX7219 into test mode, with all segments lit. The resulting current draw may overwhelm the voltage regulator, and prevent the Stamp from ever waking up. Less seriously, it may cause a bright, momentary flash on the display, making the user think that something's wrong with it. The resistors are cheap insurance against such embarrassments.

If it's so *BASIC*, how come I don't understand it?

I've received calls and e-mail recently from folks who read this column regularly and are intrigued by the applications they see here, but don't know anything about programming. They're eager to get started with the Stamp or Counterfeit, but unsure about learning BASIC.

This was news to me, because BASIC has been universally available and very popular since the dawn of the personal-computer era. Most of the early "home computers" had BASIC stored in read-only memory (ROM) right on the machine. Some form of BASIC has been bundled with every version of DOS, and it's starting to show up as a macro language for programs like spreadsheets and word processors.

On the other hand, bookstore shelves are no longer full to bursting with books on BASIC, and the popularity of programming as a leisure activity for computer users has fallen *waaay* behind *Doom* and Internet flame wars. And many schools are no longer teaching BASIC as an introduction to computing. Ivory-tower types have convinced them that anything as understandable as BASIC must cause irreparable damage to the mind. The bizarre C language—the "C" is for "cryptic"—is much more effective at convincing people to leave programming to professionals.

So it's time for a running tutorial on BASIC. From now on, the final section of this column will contain hints and information on programming for newbies. Next issue will kick things off with a discussion of what a program is, and how to set about writing one. Thereafter, we'll look at topics like memory, math, logic, decisions, loops, and subroutines in detail. Naturally, since this column is about the Stamp and workalikes (the Counterfeit), we'll concentrate on PBASIC, but a lot of the concepts will apply to other BASICs, and to programming in general.

A final thought—most programmers agree that there are really only two effective methods for learning to program; looking at someone else's programs and writing your own. So if you have QBASIC on your DOS machine, or own one of those old built-in-BASIC dinosaurs, fire it up and play around with programming. Many of the old manuals contain great tutorials; try 'em out. If you like it (and you will), take the plunge with a Stamp or Counterfeit. We learn by doing.

Sources

For more information on the BASIC Stamp, contact Parallax Inc., 3805 Atherton Road no. 102, Rocklin, CA 95765; phone 916-624-8333; fax 916-624-8003; BBS 916-624-7101; e-mail info@parallaxinc.com.

Send questions, suggestions, or requests for future Stamp Applications to:

Scott Edwards Electronics, 964 Cactus Wren Lane, Sierra Vista, AZ 85635; phone 520-459-4802; fax 520-459-0623; e-mail (Compuserve) at 72037,2612; on the Internet 72037.2612 @ compuserve.com. Scott offers Stamp-related kits, including:

The Counterfeit controller, a kit alternative to the BASIC Stamp, is $29. Double- and quad-speed options are $2 and $4, respectively. The Counterfeit Development System, required to program Counterfeits (also for programming BS1 Stamps) is $69 and includes a 150-page manual, downloading cable kit, Parallax software, and one Counterfeit controller kit.

The MAX7219 AppKit includes complete documentation, source code for Stamps I and II and PIC microcontrollers on disk, and one MAX7219 chip for $25.

Visa, Mastercard, and American Express accepted for phone/fax orders. POs accepted on approved credit. Personal checks and money orders are welcome for mail orders.

```
' Program Listing: MAX7219.BAS (Using the LED Display Driver with BS1)
' This program controls the MAX7219 LED display driver. It demonstrates
' the basics of communicating with the 7219, and shows a convenient
' method for storing setup data in tables. To demonstrate practical
' application of the 7219, the program drives a 5-digit display to
' show the current value of a 16-bit counter (0-65535). The subroutines
' are not specialized for counting; they can display _any_ 16-bit
' value on the LCDs. (A specialized counting routine would be faster,
' since it would only update the digits necessary to maintain the
' count; however, it wouldn't be usable for displaying arbitrary
' 16-bit values, like the results of Pot, Pulsin, or an A-to-D
' conversion).

' Hardware interface with the 7219:
SYMBOL DATA_n     = 7       ' Bits are shifted out this pin # to 7219.
SYMBOL DATA_p     = pin7    ' "    "   "        "        ".
SYMBOL CLK        = 5       ' Data valid on rising edge of this clock pin.
SYMBOL Load       = 6       ' Tells 7219 to transfer data to LEDs.

' Register addresses for the MAX7219. To control a given attribute
' of the display, for instance its brightness or the number shown
' in a particular digit, you write the register address followed
' by the data. For example, to set the brightness, you'd send
' 'brite' followed by a number from 0 (off) to 15 (100% bright).
SYMBOL dcd        = 9       ' Decode register; a 1 turns on BCD decoding.
SYMBOL brite      = 10      ' "        "    " intensity register.
SYMBOL scan       = 11      ' "        "    " scan-limit register.
SYMBOL switch     = 12      ' "        "    " on/off register.
SYMBOL test       = 15      ' Activates test mode (all digits on, 100% bright)

' Variables used in the program.
SYMBOL max_dat    = b11     ' Byte to be sent to MAX7219.
SYMBOL index      = b2      ' Index into setup table.
SYMBOL nonZ       = bit1    ' Flag used in blanking leading zeros.
SYMBOL clocks     = b3      ' Bit counter used in Max_out.
SYMBOL dispVal    = w2      ' Value to be displayed on the LEDs.
SYMBOL decade     = w3      ' Power-of-10 divisor used to get decimal digits.
SYMBOL counter    = w4      ' The value to be displayed by the demo.
```

```
' The program begins by setting up all pins to output low, matching
' the state established by the pulldown resistors.
let port = $FF00           ' Dirs = $FF (all outputs) and Pins = 0 (low).

' Next, it initializes the MAX7219. A lookup table is convenient way
' to organize the setup data; each register address is paired with
' its setting data. The table sets the scan limit to 4 (5 digits,
' numbered 0-4); brightness to 3; BCD decoding to the lower 5 digits
' (the only ones we're displaying), and switches the display on. The
' MAX7219 expects data in 16-bit packets, but our lookup table holds
' a series of 8-bit values. That's why the loop below is designed to
' pulse the Load line _every_other_ byte transmitted.
for index = 0 to 7            ' Retrieve 8 items from table.
  lookup index,(scan,4,brite,3,dcd,$1F,switch,1),max_dat
  gosub Max_out
  let bit0 = index & 1        ' Look at lowest bit of index.
  if bit0 = 0  then noLoad
  pulsout Load,1              ' If it's 1, pulse Load line.
NoLoad:                       ' Else, don't pulse.
next                          ' Get next item from table.

' ====================== MAIN PROGRAM LOOP ==========================
' Now that the MAX7219 is properly initialized, we're ready to send it
' data. The loop below increments a 16-bit counter and displays it on
' the LEDs connected to the MAX. Subroutines below handle the details
' of converting binary values to binary-coded decimal (BCD) digits and
' sending them to the MAX.
Loop:
  let dispVal = counter
  gosub MaxDisplay
  let counter = counter+1
goto loop

' ========================= SUBROUTINES ============================
' The MAX7219 won't accept a number like "2742" and display it on
' the LEDs. Instead, it expects the program to send it individual
' digits preceded by their position on the display. For example,
' "2742" on a five-digit display would be expressed as:
' "digit 5: blank; digit 4: 2; digit 3: 7; digit 2: 4; digit 1: 2"
' The routine MaxDisplay below does just that, separating a value
' into individual digits and sending them to the MAX7219. If the
' lefthand digits are zero (as in a number like "102") the
' routine sends blanks, not zeros until it encounters the first
' non-zero digit. This is called "leading-zero blanking."
MaxDisplay:
let decade = 10000            ' Start with highest digit first.
let nonZ = 0                  ' Reset non-zero digit flag.
for index = 5 to 1 step -1    ' Work from digit 5 to digit 1.
  let max_dat = index         ' Send the digit address.
  gosub Max_out
  let max_dat = dispVal/decade      ' Get the digit value (0-9).
  if max_dat = 0 then skip    ' If digit <> 0 then nonZ = 1.
  let nonZ = 1                ' If a non-zero digit has already
skip:                         ' ..come, or the current digit is not
```

```
  if nonZ = 1 OR max_dat <> 0 OR index = 1 then skip2 '..0, or the
  let max_dat = 15            '.._only_ digit is 0, send the digit,
skip2:                        '..else send a blank.
  gosub Max_out               ' Send the data in max_dat and
  pulsout Load,1              ' ..pulse the Load line.
  let dispVal = dispVal//decade     ' Get the remainder of value/decade.
  let decade = decade/10      ' And go to the next smaller digit.
 next                         ' Continue for all 5 digits.
return                        ' Done? Return.

' Here's the code responsible for sending data to the MAX7219. It
' sends one byte at a time of the 16 bits that the MAX expects. The
' program that uses this routine is responsible for pulsing the
' Load line when all 16 bits have been sent. To talk to the MAX7219,
' Max_out places the high bit (msb) of max_dat on DATA_p, the data pin,
' then pulses the clock line. It shifts the next bit into position by
' multiplying max_dat by 2. It repeats this process eight times.
' In order to avoid hogging the bit-addressable space of w0, the
' routine uses a roundabout way to read the high bit of max_dat: if
' max_dat < $80 (%10000000) then the high bit must be 0, so a 0
' appears on DATA_p. If max_dat >= to $80, then a 1 appears on DATA_p.
Max_out:
for clocks = 1 to 8           ' Send eight bits.
  let DATA_p = 0              ' If msb of max_dat = 1, then let
  IF max_dat < $80 then skip3 '..DATA_p = 1, else DATA_p = 0.
  let DATA_p = 1
skip3:
  pulsout CLK,1               ' Pulse the clock line.
  let max_dat = max_dat * 2   ' Shift max_dat one bit to the left.
next                          ' Continue for eight bits.
return                        ' Done? Return.
```

Stamp Applications no. 11 (January '96):

Crystal-Controlled Oscillator Is Heartbeat of 60-hour Timer

Precision Countdown Timer
And Part 2 of an Introduction to BASIC,
by Scott Edwards

STAMP and Counterfeit users have a wish list a mile long. They want displays, ADCs, DACs, more program memory, more variables, networking, keypads, motor drivers, wireless RF and infrared, faster serial communications, interrupts, and on, and on…

But the one thing that more of you seem to want more than anything in the world is a real-time clock.

I'm not going to give it to you.

Instead, I'll show you a technique for implementing precise timekeeping without the overhead of a real-time clock. I'll start by explaining why typical real-time clocks are not particularly Stamp-friendly peripherals, and then encourage you to abandon human timekeeping conventions.

In our BASIC-for-beginners department, I'll discuss programs as lists of things to do.

Clock watching

Many applications need to know the current time of day, day of the week, time elapsed since an event, etc. Unfortunately, PBASIC's built-in timing functions aren't up to the job. They are designed to make or measure pulses or pauses, then continue the program. In other words, PBASIC doesn't provide a background clock like the one in your PC.

Some users want to overcome this limitation by brute force; "Just tell me how many microseconds each instruction takes, and I'll figure it out from there." Tain't that simple. Aside from the fact that no one has compiled a list of PBASIC instructions and the time they take, all of the math functions take varying amounts of time depending on the numbers involved in the calculation. Add the relative inaccuracy of the built-in ceramic resonator (±1 percent) and you'll conclude that this approach is hopeless.

A bit more reasonable is the idea of connecting a real-time-clock (RTC) chip to the Stamp. If you're not familiar with these guys, they're sort of a pocket watch for computers. A low-power oscillator and some digital logic keep track of the current calendar date; day of the week; hour, minute, and second.

A problem with RTCs is that they generally provide their data in their data as binary-coded decimal (BCD) digits of four bits each. (In BCD, each four bits are used represent the numbers 0–9, the decimal digits, rather than 0–15 as in pure binary. The idea is to make it simpler to display the data in human-readable form.)

As BCD digits, the date and time might look like this:

```
95/11/13/1/15/04/31
```

meaning, “1995, 11th month, 13th day, day 1 of the week (Monday), 15 hours, 4 minutes, 31 seconds.” In the original PBASIC (BS1 flavor) there are no four-bit variables, so the 13 digits of RTC data would take 13 bytes—virtually all of the Stamp’s variable storage. Even if a program converted the data from BCD to binary as it came in, it’s still a lot of data. And don’t forget that our traditional timing units don’t follow normal rules of math—quick, add 16 hours 44 minutes to 3 days 10 hours and 52 minutes. See what I mean?

In microcontroller designs, it makes sense to look at what the controller really needs to *do* with the timing data, rather than trying to bend human time-keeping conventions to fit. For example, do you need the time of day to the nearest minute? Break the day into one-minute units of 0 to 1339, with 0 being midnight, 720 being noon, etc.

Need to collect data at intervals of 8 hours? Record the starting time/date and use the 8-hour offset between samples as an implicit time tag. That is, if the first sample is taken at the start time, the second is 8 hours later; the third at 16; the fourth at 24...

Need to record the time and date that some event occurred? Again, record the start time and tag each event with an offset in appropriate units. (Remember the NASA and military habit of reporting time relative to a reference mark, like “T minus 59 seconds” or “X-hour plus 18 hours.”)

Even if you do need to keep time in human-readable form in order to display it, it may still be simpler to use a timebase and a little math. That’s what our knob-driven countdown timer does.

Countdown Timer. The countdown timer application shown in figure 1 and listing 1 can be set to turn on an output after an interval ranging from 1 second to 59 hours, 59 minutes and 59 seconds with 1-second precision. It has a user-friendly interface consisting of a twist-knob for setting the timing, and an LCD to show the time set or current state of the countdown.

Its accuracy as tested was within ±1 second per 24-hour period. This could probably be improved by trimming the values of the ground capacitors on the crystal, but it’s respectable nonetheless. If you don’t like to fiddle with discrete components, you may substitute the canned oscillator shown on the schematic. These guys cost a couple of bucks, and draw more current, but are factory tuned for better than 100-parts-per-million accuracy. Hmmm... that’s a worst-case error of more than 8 seconds in a 24-hour period, so our homebrew version checked out better. Food for thought.

Modifications. The countdown timer could readily be changed into a duration timer by moving the instructions `high out_pin` to the beginning of the timing cycle and `high out_pin` to the end. The output would turn on when the timer started, and off when it finished.

You could also substitute some other timebase for the 4060 and 32,768-Hz crystal. For example, you might divide the 60-Hz powerline frequency by 10 or 100 with a digital counter to get manageable 6- or 0.6-Hz (50 pulse/minute) pulses. The powerline frequency is tightly controlled, so it’s a decent timebase—motorized electric clocks are directly synchronized to the 60-Hz coming out of the wall.

BASIC for Beginners. Last month, I announced this new section; this month we’ll get started learning the fundamentals of BASIC programming as they apply to the Stamp and Counterfeit. I plan to start at the very beginning by examining what a program *is*.

In its simplest form, a program is nothing more than a to-do list for a computer. In the case of the Stamp, suppose you wanted to turn on a light, wait 1 second, then turn on another light. We’ll assume that the lights are low-current LEDs connected to pins 0 and 1 such that they’re on when the pins are high (putting out +5V). The to-do list would read:

```
high 0        ' Turn on first light.
pause 1000    ' Wait 1000 milliseconds (1 sec)
high 1        ' Turn on second light.
```

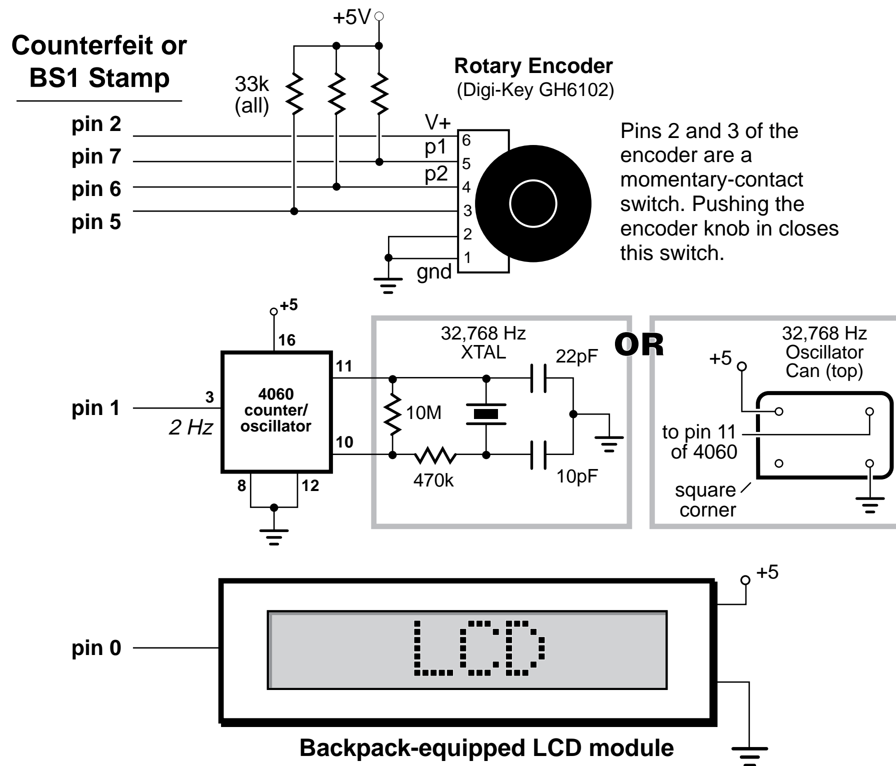


Figure 1. Schematic diagram of the 60-hour timer.
Timing intervals can be set with 1-second precision using a rotary dial.

Just like a to-do list, you can accomplish a lot by just figuring out a sequence of actions for the Stamp to perform, then converting that list into a program.

To-do lists have a couple of weaknesses and straight-line programs that imitate them share these limitations: They don't express repeated actions, and they don't adapt to changing conditions.

For example, a to-do list procedure for my mail-order business might read:

- Answer phone
- Get name of product ordered.
- Get credit-card data and shipping address.
- Hang up phone.

This procedure is terribly flawed. What if the call isn't an order, but a supplier, or a tech-support question? If it is an order, what if the customer wants more than one item? What if the credit-card and shipping information is already on file?

This is the kind of thinking that goes into programming—identifying decisions that must be made and working out responses for all of the possibilities. An improved mail-order program might begin:

- Answer phone.
- Get caller's request.
- If request is "order" then process order
- If request is "info" then tech support

...and so on. The person answering the phone gets a piece of information—the caller's request—that leads to a decision about which other to-do list to use.

The BASIC programming language lets you instruct a computer to make such simple IF/THEN decisions and act on them. Next time, we're going to write a few short programs that give IF/THEN a workout.

Sources

For more information on the BASIC Stamp, contact Parallax Inc., 3805 Atherton Road no. 102, Rocklin, CA 95765; phone 916-624-8333;

fax 916-624-8003; BBS 916-624-7101; e-mail info@parallaxinc.com.

Send questions, suggestions, or requests for future Stamp Applications to:
Scott Edwards Electronics, 964 Cactus Wren Lane, Sierra Vista, AZ 85635; phone 520-459-4802; fax 520-459-0623; e-mail (Compuserve) at 72037,2612; on the Internet 72037.2612 @ compuserve.com. Scott offers Stamp-related products and kits, including:

The Counterfeit controller, a kit alternative to the BASIC Stamp, is $29. Double- and quad-speed options are $2 and $4, respectively. The Counterfeit Development System, required to program Counterfeits is $69 and includes a 150-page manual, downloading cable kit, Parallax software, and one Counterfeit controller kit.

The LCD Serial Backpack is a daughterboard that attaches to LCDs, converting their fussy parallel interface to Stamp-compatible serial at 2400 or 9600 baud. The assembled Backpack is $29; with 16x1 LCD, $40; 16x2 LCD, $45; or backlit 20x4 LCD, $89.

Visa, Mastercard, and American Express accepted for phone/fax orders. Personal checks and money orders are welcome for mail orders.

```
' Program: ROT_TIME.BAS (Timer with rotary-encoder interface)
' This program implements a 60-hour countdown timer with a user-friendly
' rotary-encoder (twist-knob) interface and LCD Serial Backpack display.
' When first powered up, the display shows "00:00:00" and waits for
' the user to twist the knob to set the hours. Clockwise increases the
' setting, counter-clockwise reduces it. When the hours are set, the
' user pushes the knob in to set the minutes and seconds in the same
' way. Once the seconds are set, pushing the knob in one more time
' starts the timer. The display counts down to zero, then turns on the
' output.

' This application relies on an external timer as an accurate source of
' 2-Hz 'ticks.' Typical accuracy is within 2-3 seconds over the maximum
' timing period of 59:59:59 (almost 60 hours). Another interesting
' feature of the application is its control of the rotary-encoder power
' supply. Since the encoder's LEDs draw almost 20 mA of current, the
' program shuts them off when they're not needed and thereby conserves
' battery power.

' ================================
' Variables and constants.
' ================================

SYMBOL old = b0             ' Previous bit pattern of rotary encoder.
SYMBOL new = b1             ' Current  "   "       "  "      "
SYMBOL directn = bit0       ' Direction of knob rotation.
SYMBOL count = b2           ' Number dialed in by encoder.
SYMBOL hours = b3           ' Timer hours setting.
SYMBOL minutes = b4         ' Timer minutes setting.
SYMBOL seconds = b5         ' Timer seconds setting.
SYMBOL temp = b6            ' Temporary variable used by display routine.
SYMBOL prnPos = b7          ' Printing position on LCD screen.
SYMBOL btn = b8             ' Workspace variable for Button command.
SYMBOL case = b9            ' Offset for Branch command.
SYMBOL out_pin = 3          ' Output pin controlled by timer.
SYMBOL encoder = 2          ' Power to rotary encoder LEDs.
SYMBOL I = 254              ' LCD Backpack instruction prefix (see note).
SYMBOL cls = 1              ' LCD Backpack clear-screen instruction.
```

```
' NOTE: This program is written for the rev3A Backpack firmware,
' which uses an instruction prefix, rather than a toggle. The new
' firmware makes this latest Backpack "reset proof" since the
' controller can always put the LCD into a known state by clearing
' the screen (and optionally also turning the cursor on/off).

' ================================
' Main Program Start
' ================================
Begin:
  low out_pin                  ' Turn off the output pin.
  high encoder                 ' Turn on power to encoder LEDs.
  pause 1000                   ' Wait a sec for LCD initialization.
  serout 0,n2400,(I,cls)       ' Clear the LCD screen
  let new  = pins & $C0        ' Get initial state of encoder pins.
  let prnPos = 132             ' Set print position to 4 (128+4)
  gosub Display                ' Put 0s on the display.

' ================================
' User Setup of Time Duration
' ================================
Setup:
  gosub rotary                 ' Check the knob.
  serout 0,n2400,(I,prnPos)    ' Position cursor on the display.
  gosub showDigs               ' Display digits.
  button 5,0,255,0,btn,1,pushed     ' Check for knob push on pin 5.
goto Setup                     ' Loop.

' If the knob is pushed in, causing a low on pin 5, the program
' jumps from setup to here. It checks the current printing position
' to determine whether the user has been setting hours, minutes, or
' seconds and determine what to do next.
pushed:
  let case = prnPos-132/3       ' Convert position to 0-2.
  branch case,(setHours,setMins,setSecs) ' Branch based on 0-2)
setHours:
  let hours = count            ' Put the count into hours.
  goto continue                ' Continue setting timer.
setMins:
  let minutes = count          ' Put the count into minutes.
  goto continue                ' Continue setting timer.
setSecs:
  let seconds = count          ' Put the count into seconds.
  goto runTimer                ' And start the countdown.
continue:
  let count = 0                ' Continue: clear count for next.
  let prnPos = prnPos+3        ' Move to next screen position.
goto Setup                     ' Get more input from user.
```

```
' =================================
' Timing Countdown
' =================================
runTimer:
let old = 0                ' Initialize "old" to track ticks from timer.
low encoder                ' Turn off the encoder.

' This code counts changes in state from the external timer. Every
' fourth change (transition from 0-1 or 1-0) of the 2-Hz clock means
' that a second has passed. When that happens, the program subtracts
' 1 from the seconds, minutes and hours.
DoTiming:
  if pin1 = bit0 then DoTiming      ' No change? Loop.
  let old = old + 1            ' Changed: increment old.
  let new = old & %11          ' Look at bottom two bits of old.
  if new <> 3 then DoTiming    ' Loop is not 3 (4th count, 0,1,2,3)..
  let seconds = seconds - 1    ' Fourth count: decrement seconds.
  if seconds <> 255 then update     ' If not underflow (-1 = 255), update.
  let seconds = 59             ' Underflow: wrap around to 59 seconds.
  let minutes = minutes -1     ' Seconds underflowed: borrow 1 from mins.
  if minutes <> 255 then update     ' If not underflow (-1 = 255), update.
  let minutes = 59             ' Underflow: wrap to 59 minutes.
  let hours = hours - 1        ' Minutes underflowed: borrow 1 from hours.

update:
  gosub Display                ' Display new hours/mins/secs.
check:
  if hours <> 0 then DoTiming ' If not 00:00:00, continue timing.
  if minutes <> 0 then DoTiming
  if seconds <> 0 then DoTiming
  high out_pin                 ' Time's up: turn on the output.
hold: goto hold                ' Endless loop: reset to start again.


' =================================
' Subroutines
' =================================
' Check the rotary encoder. If it has moved, determine direction and
' adjust the value of the variable "count" accordingly.
rotary:
  let old = new & $C0          ' Make old = top two bits of new.
again:
  let new = pins & $C0         ' Make new = top two bits of pins.
  if new = old then done       ' No change? Done.
  let directn = bit6 ^ bit15   ' Change: determine direction.
  if directn = 1 then CW       ' Clockwise: goto routine below.
  let count = count - 1        ' Counterclockwise: decrement count.
  if count <> 255 then skip    ' If count < 0, then count = 59.
  let count = 59
skip:
return                     ' Return to main program.
```

```
CW:
  let count = count + 1     ' Clockwise: increment count.
  if count <> 60 then done ' If count = 60, wrap around to 0.
  let count = 0
done:
  return                    ' Return to main program.

' Display the hour:minute:second digits on the LCD screen.
Display:
  serout 0,n2400,(I,132)    ' Start at hours position.
  let count = hours         ' Show hours digits.
  gosub showDigs
  serout 0,n2400,(":")      ' Colon.
  let count = minutes       ' Now minutes.
  gosub showDigs
  serout 0,n2400,(":")      ' Colon.
  let count = seconds       ' Now seconds.
  gosub showDigs
return                      ' Return to main program.

' Display the two-digit value stored in count on the LCD.
showDigs:
  let temp = count/10       ' Get the tens-place digit.
  serout 0,n2400,(#temp)    ' Put it on the display.
  let temp = count//10      ' Get the ones-place digit.
  serout 0,n2400,(#temp)    ' Put it on the display.
return                      ' Return to main program.
```

Stamp Applications no. 12 (February '96):

Model Rocket Project Aims High With BS1-IC Instrumentation

Measuring Rocket Acceleration And Velocity And Truth and Consequences with IF/THEN, by Scott Edwards

MODEL rockets and Stamp microcontrollers appeal to educators for many of the same reasons. Both offer:

- Hands-on experience with high techology.
- A starting point for discussions of science and math fundamentals.
- An inexpensive, but thoroughly educational class project.
- Impressive, tangible accomplishments that *wow* school administrators and parents.
- Enough fun and excitement to turn slack-jawed Nintendo burnouts into enthusistic students!

You don't have to take my word for it. This month, I'm turning the first part of the column over to Dave Bodnar, Technology Coordinator for the Mount Lebanon School District of Pittsburgh, Pennsylvania. Last summer, Dave combined Stamp electronics with model-rocket pyrotechnics to create a couple of high-flying Stamp applications. Here's a description in Dave's own words:

Enclosed you will find the prototypes of the rocket sensors that I put together this summer. One senses the rocket's speed by measuring the speed of a propeller on the nose cone. The other measures acceleration.

The speed device uses an infrared photo transistor/emitter pair to detect the spinning of the propeller. A split wheel chops the light that normally goes between the emitter and detector. The two LEDs and buzzer are used to let the launch crew know when it is time to launch the rocket. The Stamp's internal memory is used to store the data that is collected. I have enclosed a disk with the results of the first few flights. Note that the power source for the speed sensor is a set of watch batteries in the base of the nose cone.

The system is operable and can be brought to life by connecting the sets of red/black wires. While making up the drawing of the circuit, I noticed that I had mislabeled the RESET and +5 pins and attached things incorrectly. Fortunately the RESET pin seems to supply enough current to run the circuit and pressing the reset button in the nose cone shorts the +5 and Ground to do a type of reset!

The acceleration sensor works on the same principle as the first device except that it senses the brightness of the light that falls on a CdS cell in the top of the hypodermic syringe. The bulb moves up and down the syringe based on the acceleration of the rocket. The syringe is normally covered with black tape to keep out light. The small switch on the top of the gold cells is the power switch. The cells are 2/3 of a 9-volt battery. They serve as a power source and a weight for the sensor.

The data is collected by connecting a small PC (I used an HP 95LX) via the serial port. The data is continuously output after the flight.

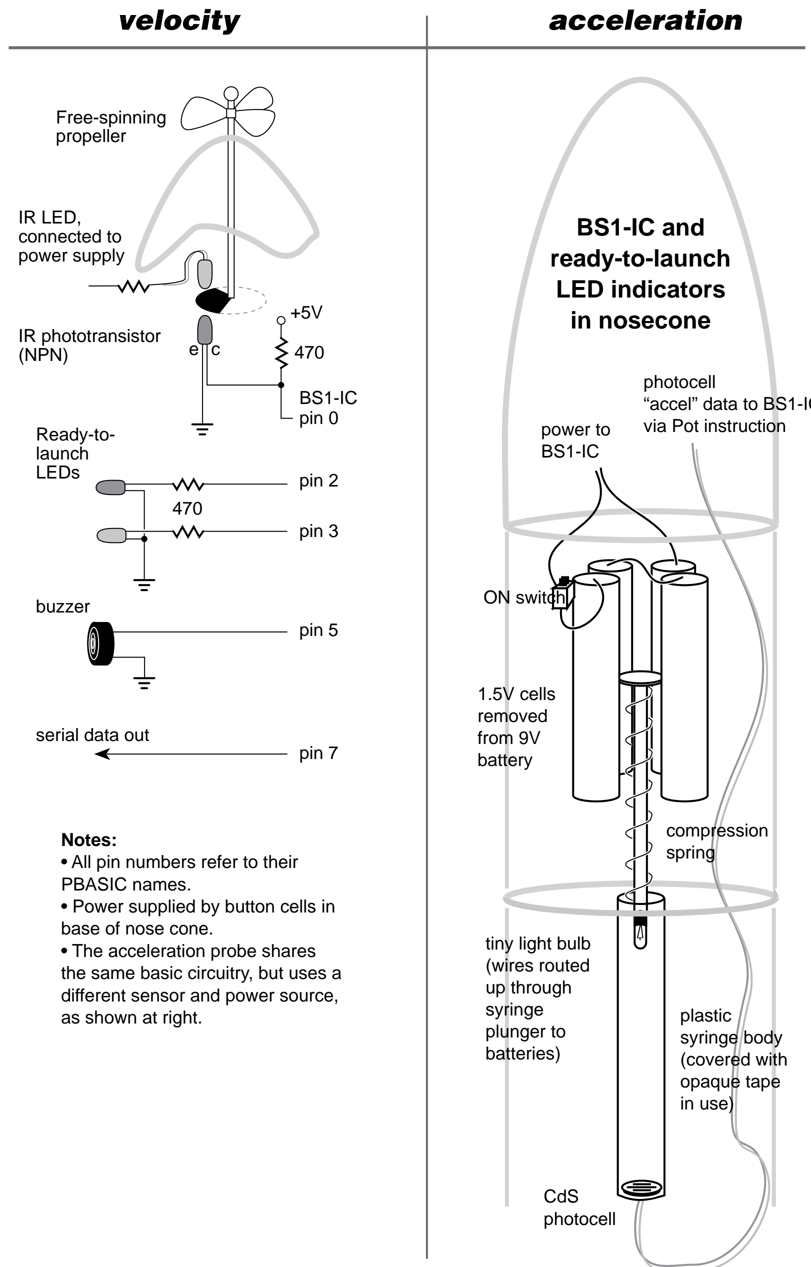


Figure 1. Diagrams of the velocity and acceleration probes.

Figure 1 shows the basic construction of Mr. Bodnar's instrumented model rockets, while listings 1 and 2 are the programs that run them.

BASIC for Beginners. Last month, I equated a BASIC program to a to-do list. The computer starts at the top of the program and performs the instructions in order of appearance.

However, the real power of programming is the ability to make decisions. The program is still a list of actions to do, but the order in which they're performed (and whether some of them are performed at all) can be based on conditions that exist at the time the program is run.

In BASIC, you describe decisions and their consequences with the IF/THEN instruction.

The *syntax* of this instruction—the correct way to use and express this instruction in your programs—is:

```
IF condition THEN label
```

In this syntax, *condition* is a comparison that can be true or false; for example 2 = 10 (read as *two is equal to ten*) is an example of a false condition. The comparison operators, also called relational operators, available in PBASIC are:

=	equal	<>	not equal
>	greater than	<	less than
>=	greater or equal	<=	less or equal

If you have trouble remembering which way the < should point, just use this mnemonic: the symbol is the mouth of a greedy alligator who always snaps up the larger of the two meals offered. So x > y reads *x is greater than y*.

When PBASIC processes an IF/THEN instruction, it determines whether the condition is true or false. If it's true, the next instruction the program executes will be the one specified by the label following THEN. If the condition is false, PBASIC will just go on to the next instruction in the program in normal to-do list fashion.

Let's put this information to work in a simple application. Suppose we're monitoring a burglar alarm system that's wired so that a 0 appears on pin1 when all doors and windows are shut, and a 1 when any are opened. When the system is turned on, opening a door or window should trigger the alarm. Here's a straightforward way to express that in PBASIC:

```
Monitor:
 IF pin1 = 1 THEN Alarm
GOTO Monitor

Alarm:
 SOUND 7,(90,100,120,100)
GOTO Alarm
```

In the section labeled Monitor, if pin1 is 0, meaning all is secure, the program just goes back to Monitor and checks the pin again. However, if pin1 is 1, meaning a door or window has been opened, the program goes to the instructions that start with the label Alarm. Here PBASIC generates a two-tone sound through pin7 to attract attention. (Don't worry about understanding the specifics of instructions like Sound—they're just fill-in-the blank forms for performing a simple action.)

Before I go on, I want to explain what labels like Monitor and Alarm are and how they work. You can see from the example that decisions and repeated actions depend on telling PBASIC to go to a specific point in the list of instructions that makes up a program. In old versions of BASIC each line began with a number, so you specified the "where" of a GOTO or IF/THEN instruction with the number of the destination line.

Line numbers don't say much about the "what" or "why" of these destinations, so modern BASICs like PBASIC use labels instead. A label is just a word that you, the programmer, have picked to mark a place in the program. To let PBASIC know that it's a label, you end the chosen word with a colon (:). There are other rules about labels, but we'll leave it at that for now.

I want to leave you with a further thought about IF/THEN: Suppose we wanted another condition that could trigger the alarm—say a panic button on pin2. It would normally read 0, but if you heard a strange noise in the bushes, you could push the button to put a 1 on pin2 to trigger the alarm. In other words, you want the alarm to go off if there's a door/window open OR the panic button is pressed. A small change to the IF/THEN instruction does the job:

```
Monitor:
 IF pin1 = 1 OR pin2 = 1 THEN Alarm
GOTO Monitor
```

Looky there: the idea of OR is expressed the same in PBASIC as in plain English! From this simple foundation, we'll examine the whole system of truth and consequences called Boolean logic in the next installment.

Sources

For more information on the BASIC Stamp, contact Parallax Inc., 3805 Atherton Road no. 102, Rocklin, CA 95765; phone 916-624-8333; fax 916-624-8003; BBS 916-624-7101; e-mail info@parallaxinc.com.

Send questions, suggestions, or requests for future Stamp Applications to:
Scott Edwards Electronics, 964 Cactus Wren Lane, Sierra Vista, AZ 85635; phone 520-459-4802; fax 520-459-0623; e-mail (Compuserve) at 72037,2612; on the Internet 72037.2612 @ compuserve.com. Scott offers Stamp-related products and kits, including:

The Counterfeit controller, a kit alternative to the BASIC Stamp, is $29. Double- and quad-speed options are $2 and $4, respectively. The Counterfeit Development System, required to program Counterfeits is $69 and includes a 150-page manual, downloading cable kit, Parallax software, and one Counterfeit controller kit.

The LCD Serial Backpack is a daughterboard that attaches to LCDs, converting their fussy parallel interface to Stamp-compatible serial at 2400 or 9600 baud. The assembled Backpack is $29; with 16x1 LCD, $40; 16x2 LCD, $45; or backlit 20x4 LCD, $89.

Visa, Mastercard, and American Express accepted for phone/fax orders. Personal checks and money orders are welcome for mail orders.

' Listing 1, Model Rocket Velocity Probe, by Dave Bodnar

```
REM Rocket telemetry program MINIMUM memory implementation
REM for use with PROPELLER/SPEED sensor only
REM uses internal STAMP memory for data storage
' RKT_SPD1.BAS (was RKT_MIN5.BAS) D. Bodnar  7-6-95 6:53

Symbol Prop = 0            'pin 0 for propellor
Symbol LED1 = 2            'pin to show ready for launch
Symbol LED2 = 3            'pin (another) to show ready for launch
Symbol Buzz = 5            'pin for Piezo buzzer
Symbol Ser_out = 7         'pin for serial output of data (orange wire)
Symbol Delay = 8000        ' better at about 8000 - 1000 for testing only
Symbol STOPmem=2

realstart:
read 255,b11               'memory end location for writing
'debug b11,cr
b11=b11-1
b10=b11                    'make copy for later

LOW buzz
high LED2:high LED1        'Both on FIRST
pause Delay                'RED only on (LED1)
HIGH buzz
low LED2                   'GREEN only on (LED2)
pause Delay                'LAUNCH ready
LOW buzz
high LED2:low LED1
pause Delay
Low LED2
HIGH buzz
```

```
waitforlaunch:              'stay here till LAUNCH detected
  high LED1                 'flash RED while waiting
  pulsin prop,1,w2
  low LED1
if w2=0 then waitforlaunch:
High LED2
LOW buzz
start:
   gosub GETnWRITEit:
if b11 >STOPmem then start:    'loop till RAM full
Low LED2
HIGH buzz
doneloop:
   b9=b10

'debug "start of end",cr
LOW buzz
   serout ser_out,n2400,("START",#b9,13,10)     'send "Start" & amount of RAM
  pause 2000
loop:
  HIGH buzz
  read b9, b7:b9=b9-1:read b9,b8
' debug #b9,"+1  hi=",#b7," lo=",#b8
  w2=b7*256 + b8
' debug " ",#w2,cr
  LOW buzz
  serout ser_out, n2400,(#w2,13,10)
' high LED1:Pause 20:low LED1:pause 20
  b9=b9-1
  if b9 > STOPmem then loop
  pause 5000
goto doneloop:

GETnWRITEit:
  HIGH buzz:PAUSE 40
  pulsin prop, 1,w2            'take reading
  b7=w2/256
  w1= b7*256
  b8=w2-w1
' debug #b11,"hi=",#b7," lo=",#b8, " 16bit=",#w2,cr
  Write b11,b7:b11=b11-1:write b11,b8:b11=b11-1
  LOW buzz:PAUSE 40
return
```

```
' Listing 2, Model Rocket Acceleration Probe, by Dave Bodnar
REM Rocket telemetry program MINIMUM memory implementation
REM uses internal STAMP memory for data storage
' RKT_ACC1.BAS (was RKT_MIN5.BAS) D. Bodnar  7-6-95 6:27

Symbol Accel = 1           'pin 1 for acceleration
Symbol LED1 = 2            'pin to show ready for launch
Symbol LED2 = 3            'pin (another) to show ready for launch
Symbol Ser_out = 7         'pin for serial output of data (orange wire)
Symbol Delay = 8000        ' better at about 8000 - 1000 for testing only

realstart:
read 255,b11               'memory end location for writing
b10=b11-1                  'make copy for later

high LED2:high LED1        'Both on FIRST
pause Delay                'RED only on (LED1)
low LED2                   'GREEN only on (LED2)
pause Delay                'LAUNCH ready
high LED2:low LED1
pause Delay
Low LED2


  gosub GETnWRITEit:
  b6=b2-3

waitforlaunch:             'stay here till LAUNCH detected
  high LED1                'flash RED while waiting
  pot accel,170,b2
  low LED1
if b2>b6 then waitforlaunch:
High LED2
start:
   gosub GETnWRITEit:
low LED2:pause 20:high LED2:pause 5
if b11 >1 then start:         'loop till RAM full
Low LED2

doneloop:
   b9=b10
   serout ser_out,n2400,("S",#b9,13,10)   'send "Start" & amount of RAM
   pause 2000
```

```
loop:
   read b9, b2:
   serout ser_out, n2400,(#b2,13,10)
   high LED1:Pause 20:low LED1:pause 20
   b9=b9-1
   if b9 > 1 then loop
   pause 5000
goto doneloop:

GETnWRITEit:
  pot accel,170,b2            'take reading
  Write b11,b2:b11=b11-1
return
```

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.

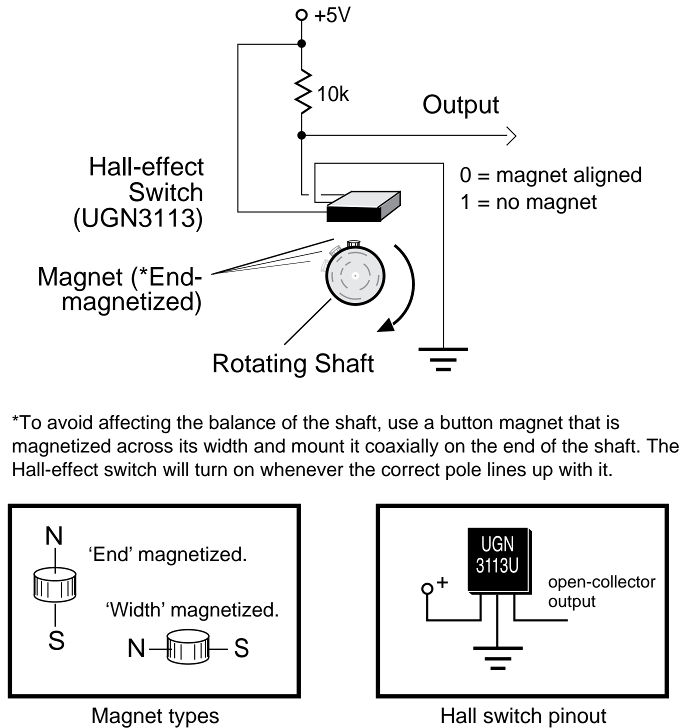


Figure 1. A PBASIC watchdog timer can monitor a motor via a Hall-effect switch like this.

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.

Sources

For more information on the BASIC Stamp, contact Parallax Inc., 3805 Atherton Road no. 102, Rocklin, CA 95765; phone 916-624-8333; fax 916-624-8003; BBS 916-624-7101; e-mail info@parallaxinc.com.

Send questions, suggestions, or requests for future Stamp Applications to:
Scott Edwards Electronics, 964 Cactus Wren Lane, Sierra Vista, AZ 85635; phone 520-459-4802; fax 520-459-0623; e-mail (Compuserve) at 72037,2612; on the Internet 72037.2612 @ compuserve.com. Scott offers Stamp-related products and kits, including:

The Counterfeit controller, a kit alternative to the BASIC Stamp, is $29. Double- and quad-speed options are $2 and $4, respectively. The

Counterfeit Development System, required to program Counterfeits is $69 and includes a 150-page manual, downloading cable kit, Parallax software, and one Counterfeit controller kit.

The Stamp Stretcher 1B adds 16 digital I/O lines, plus an 8-bit analog input to your Stamp I, Counterfeit, or Stamp II. It interfaces with these controllers via a 2400- or 9600-bps serial connections. The Stretcher 1B is $30 in kit form, $45 assembled.

Visa, Mastercard, and American Express accepted for phone/fax orders. Personal checks and money orders are welcome for mail orders.

The UGN3113U Hall-effect switch is available from Newark Electronics. To get the number of the Newark office nearest you, phone their national administrative office at 312-784-5100; Canada, 416-670-4187; overseas, 312-638-7652.

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.
```