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.
```

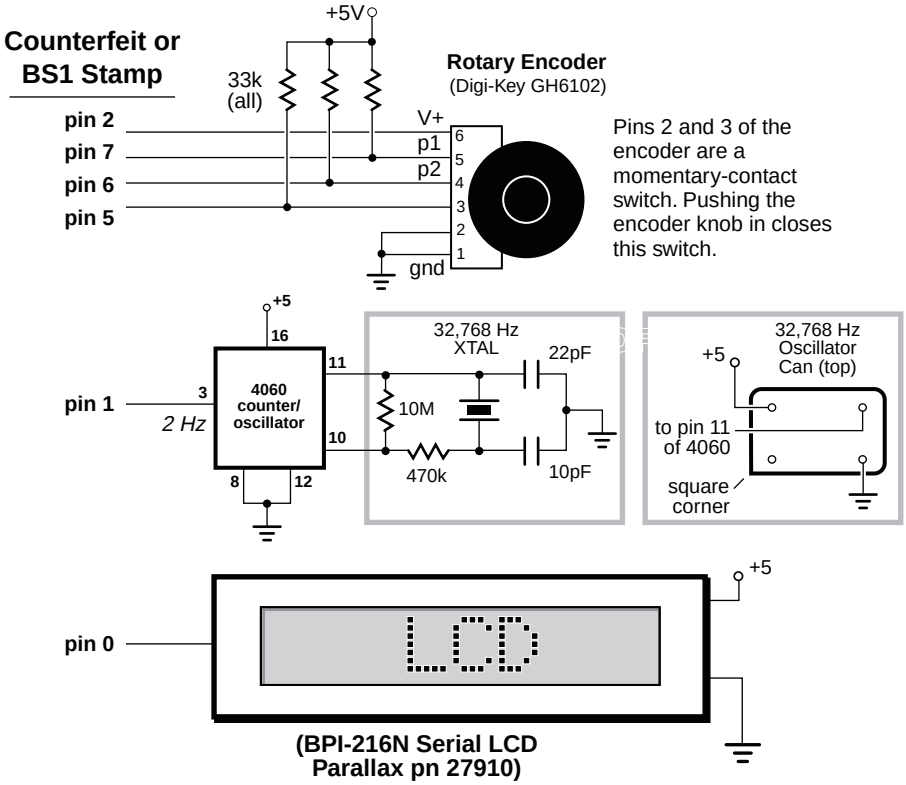

Figure 1. Schematic diagram of the 60-hour timer.
Timing intervals can be set with 1-second precision using a rotary dial.

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.

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

```
' 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.
```