Introduction. This application note describes an inexpensive and accurate timebase for Stamp applications.

Background. The Stamp has remarkable timing functions for dealing with microseconds and milliseconds, but it stumbles a little when it comes to minutes, hours, and days.

The reason for this is twofold: First, the Stamp's ceramic resonator timebase is accurate to about ±1 percent, so the longer the timing interval, the larger the error. A clock that was off by 1 percent would gain or lose almost 15 minutes a day.

Second, Stamp instructions take varying amounts of time. For example, the *Pot* command reads resistance by measuring the length of time required to discharge a capacitor. The higher the resistance, the longer *Pot* takes. The math operators also take varying amounts of time depending on the values supplied to them.

The result is that even the most carefully constructed long-term timing programs end up being less accurate than a cheap clock.

An obvious cure for this might be to interface a real-time clock to the Stamp. Available units have all kinds of neat features, including calendars with leap-year compensation, alarms, etc. The trouble here is that once you write a program to handle their synchronous serial interfaces, acquire the time from the user, set the clock, read the time and convert

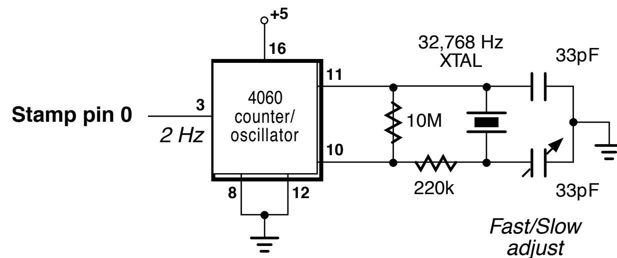

Figure 1. Schematic to accompany TIC_TOC.BAS

it to a usable form, you have pretty much filled the Stamp's EEPROM. A compromise approach is to provide the Stamp with a very accurate source of timing pulses, and let your program decide how to use them. The circuit and example program presented here do just that. For this demonstration, the Stamp counts the passing seconds and displays them using *debug*.

How it works. The circuit in figure 1 shows how to construct a crystal-controlled, 2-pulse-per-second timebase from a common digital part, the CD4060B. This part costs less than $1 from mail-order companies like the one listed at the end of this note. The 32,768-Hz crystal is also inexpensive, at just over 50 cents.

The 4060 is a 14-stage binary counter with an onboard oscillator. Although the oscillator can be used with a resistor/capacitor timing circuit, we're going for accuracy; hence the crystal. Why 32,768 Hz and not some other value, like 1 MHz? It just happens that $32,768 = 2^{15}$, so it's easy to use a binary counter like the 4060 to divide it down to easy fractions of one second. Since the 4060 is a 14-stage counter, the best it can do is divide by 2^{14}. The program further divides the resulting twice-a-second pulses to produce one count per second.

Take a look at the program listing. It consists of a main loop and a routine to increment the clock. In an actual application, the main loop would contain most of the program instructions. For accurate timing, the instructions within the main loop must take less than 250 milliseconds total. Even with the timing problems we've discussed, that's pretty easy to do.

Let's walk through the program's logic. In the main loop, the program compares the state of *pin0* to *bit0*. If they're equal (both 0 or both 1) it jumps to the *tick* routine.

In *tick*, the program toggles *bit0* by adding 1 to the byte it belongs to, *b0*. This makes sure that *bit0* is no longer equal to the state of *pin0*, so the program won't return to *tick* until *pin0* changes again.

B0 also serves as a counter. If it is less than 4, the program returns to the main loop. When *b0* reaches 4, *tick* clears it, adds 1 to the running total

of seconds, displays the number of seconds on the screen, and jumps back to the main loop.

This is pretty elementary programming, but there's one detail that may be bothering you: If we're using a 2-Hz timebase, why count to 4 before incrementing the seconds? The reason is that we're counting transitions—changes in the state of *pin0*—not cycles. Figure 2 shows the difference.

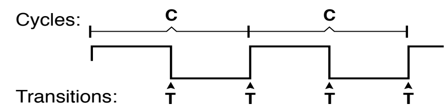

Figure 2.

This stems from our use of *bit0* to track changes in the timing pulses. As soon as *pin0* = *bit0*, we drop into *tick* and toggle the state of *bit0*. This keeps us from visiting *tick* more than once during the same pulse. The next time *pin0* changes—the next transition—*pin0* = *bit0*, and *tick* executes again. A side effect of this approach is that we increment the counter twice per cycle.

Construction notes. The circuit in figure 1 draws only about 0.5 mA, so you can power it from the Stamp's +5V supply without any problem. The resistor and capacitor values shown are a starting point, but you may have to adjust them somewhat for most reliable oscillator startup and best frequency stability. You may substitute a fixed capacitor for the adjustable one shown, but you'll have to determine the best value for accurate timing. The prototype was right on the money with a 19-pF capacitor, but your mileage may vary due to stray capacitance and parts tolerances.

Parts source. The CD4060B and crystal are available from Digi-Key (800-344-4539) as part numbers CD4060BE-ND and SE3201, respectively.

Program listing. This program may be downloaded from our Internet ftp site at ftp.parallaxinc.com. The ftp site may be reached directly or through our web site at http://www.parallaxinc.com.

```
' Program: TIC_TOC.BAS  (Increment a counter in response to a
' precision 2-Hz external clock.)

' The 2-Hz input is connected to pin0. Bit0 is the lowest bit of b0,
' so each time b0 is incremented (in tick), bit0 gets toggled. This
' ensures that tick gets executed only once per transition of pin0.

Main:
        if pin0 = bit0 then tick
          ' Other program activities--
          ' up to 250 ms worth--
          ' go here.
        goto Main

' Tick maintains a 16-bit counter to accumulate the number of seconds.
' The maximum time interval w1 can hold is 65535 seconds--a bit over
' 18 hours. If you want a minute count instead, change the second
' line of tick to read: "if b0 < 240 then Main". There are 1440 minutes
' in a day, so w1 can hold up to 65535/1440 = 45.5 days worth of to-the-
' minute timing information.

tick:
        let b0 = b0 + 1                 ' Increment b0 counter.
        if b0 < 4 then Main             ' If b0 hasn't reached 4, back to Main.
        let b0 = 0                      ' Else clear b0,
        let w1 = w1 + 1                 '  increment the seconds count,
        debug cls,#w1," sec."           '  and display the seconds.
        goto Main                       ' Do it again.
```