INTERVALOMETER POWER CONSUMPTION

One of the key things I wanted from the design was to keep the power consumption low. By putting the PIC to sleep one can get ridiculously low power consumption but it's tricky then to keep the timer going.

In practice, when clocked at 32kHz and the supply voltage is 3V from a couple of AA batteries, the power consumption is about 60µA. If the AA based battery has a capacity of 1Ah it should last for nearly two years, which seems reasonable.

However there's another significant power drain: the LEDs and opto-couplers draw about 5mA each when on. Each exposure clocks up about 15LED seconds, or about 2×10^{-5} Ah. So a better model for battery life is to say that we'll be able to take about 50,000 photos.

Although not disasterous, it's obvious that this could be improved. The opto-isolators driving the camera might work with smaller currents, or could be replaced entirely with transistors: it's not quite clear that the isolation is needed.

Without the optos, it makes more sense to reduce the power drawn by the status LED: either lower current, a smaller duty-cycle, or both.

Accurate timing

I mentioned above that I was keen to get reasonably good accuracy from the intervalometer. One movie I wanted to make was a clock with the frames exactly one minute apart. Then, the second hand would stay still whilst the other hands moved. If such a movie were to last a day, then we'd need an accuracy of about 1 in 10^5.

We'll ignore the issue of frequency drift for now, and pretend that the only problem to overcome is that the crystal's frequency isn't exactly 32,768 Hz. I don't have a data sheet for the specific crystal I used, but an accuracy of ±20ppm seems to be fairly standard. That's about twice what we'd like to achieve.

One could try to fix this in the analogue realm changing the oscillator capacitors and thus its frequency. However, it's more sensible to handle the problem in the digital domain. We'll need to get some sense of scale though. Recall that each instruction takes about 122µs to execute, and that the time between exposures must be an integral number of instructions.

Now, if we want to change the interval between exposures by 1ppm we'll need to execute at least a million instructions. That will take about two minutes. In these days where so much software is written without much regard to the instruction

count (because CPUs are so fast), it's sobering to be in realm where we're concerned with a single extra instruction every two minutes!

Quite often we'll want the interval between exposures to be less than two minutes, so it's clear that to get the right average exposure, we'll have to vary the interval between exposures. For example, if the clock runs a bit fast so that we'd like (ideally) to have 512.3 ticks between exposures, we'll choose between 512 and 513.

Now, over how many Timer1 cycles should we do the averaging ? We know that to get 1ppm adjustments we'll need to wait about 2 minutes, which is roughly 2000 complete Timer1 cycles. Given that it's not going to be a particularly quick process I thought it worth waiting 4096 Timer1 cycles. That should take a pleasing 256s to complete.

Explicitly we'll have a tuning parameter between 0 and 4095, and implement a counter which counts from 0 to 4095. When the sum of the two is more than 4095 then we'll load the timer with the higher number.

There's one minor twist: rather than have a 12-bit counter ($2^{12} = 4096$) which increments one digit at a time, there's a 16-bit counter which counts up 16 at-a-time. Putting the four unused bits at the least-significant end means that we can specify the instruction count for the 256s cycle in a single 32-bit value.

In 256s we'll execute about two million instructions, so our precision will be about 0.5ppm. Happily, my bench timer claims an accuracy of 0.2ppm which gives us a good way to do the tuning. To drive the timer, we'll pulse one of the spare PORTC outputs every 256s.

There are a couple of details to consider. All the counters increment and things happen when they overflow. So, if we want n cycles in our 256s period the configuration datum will be 0x100000000 - n.

Loading a new value into Timer1 costs a couple of ticks, so actually the datum will be 0x100020000 - n.

Finally time, and instructions, will elapse between the interrupt being triggered and us reloading the timer. So, we should **add** a correction to the timer's LSB rather than setting it.

In practice this scheme works well enough. Rather lazily though, there's no convenient way to calibrate the intervalometer: instead one has to edit the constant in the source code, assemble, and upload it. Here's the relevant code:

```
;; higher number => shorter period
constant tmr1_dh    = 0xfe ; 0xfe00 -> 0x10000 = 512 => 16Hz fast
clock
constant tmr1_dl    = 0x01 ; these 2 cycles are lost when we reload
```

```
;; tweak setting (only the 12 most significant bits matter)
;; this is device/crystal specific
constant tmr1_adj_h = 0xff ;
constant tmr1_adj_l = 0x70 ;
;; increment to adjustment clock (0x80 => 32s cycle, 0x40 = 64s
cycle, ..)
constant adj_clk_inc = 0x10
```

We'll see later that the oscillator frequency depends a bit on the supply voltage, so irritatingly if we program the PIC at 5V but deploy the intervalometer at 3V we'll have to take this into account.

A helpful shuffle

Whilst the scheme above works, there's a snag. All of the (n+1) cycle periods are clumped together. This makes the deviation from the ideal behaviour worse than it need be.

Happily, it's easy to make a significant improvement. Recall that the heart of the problem is that the code for picking the Timer1 period is:

```
inc = (cycle + offset) > 4096 ? n : n + 1;
```

where cycle counts from 0 to 4095, and offset is fixed.

Suppose we change that to:

```
inc = (P(cycle) + offset) > 4096 ? n : n + 1;
```

where P(i) shuffles the numbers [0,4095]. Over a complete cycle the test will be true just as often, but it will be true at different times.

One could imagine all manner of clever definitions for P, but this doesn't do a bad job:

```
P(i) = i `xor` ((i && 0xff) << 8)
```

That is, just XOR the high byte with the low.

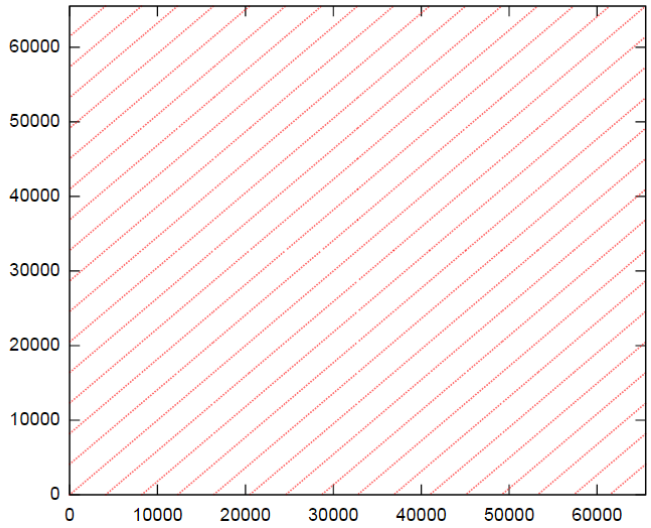
It's probably obvious that this just shuffles the elements, but if it's not here's a demonstration (in Haskell):

```
> :m Data.Bits Data.List
> let states = [ (h,l) | h <- [0..255], l <- [0,16..255] ]
> take 8 states
[(0,0),(0,16),(0,32),(0,48),(0,64),(0,80),(0,96),(0,112)]
> let states' = map (\(h,l) -> (h `xor` l, l)) states
> take 8 states'
[(0,0),(16,16),(32,32),(48,48),(64,64),(80,80),(96,96),(112,112)]
> sort states' == sort states
True
```

Or, if you prefer pictures, the plot below shows the shuffle. It might make more sense to think of the plot as a bit map in which every row and column has precisely

one cell filled. To see which cycles will enjoy the extra timer tick, mentally draw a horizontal line at the relevant level. Then regard the x-axis as time: if there's a dot below the line at that time, then we'll get an extra tick.

By contrast, the unpermuted code would simply have a 'y = x' line here: if you play the same game with the horizontal line, all the extra-tick times will be clumped at the left-side of the graph.



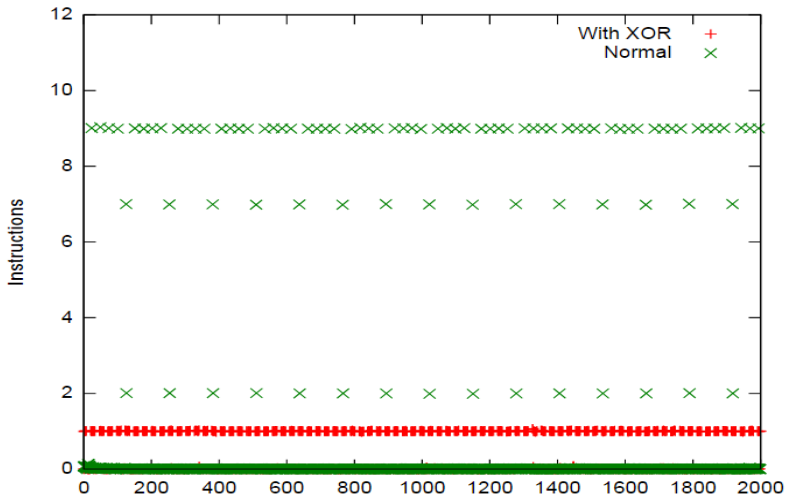
Ultimately of course we care about the effect on the interval between shutter triggers. The plots below show these, but some interpretation is needed.

Suppose we just plotted the interval over time. The interval's nominally 10s, and we're looking for changes on the order of 100µs: the time to execute an instruction. Clearly we're looking for a small effect!

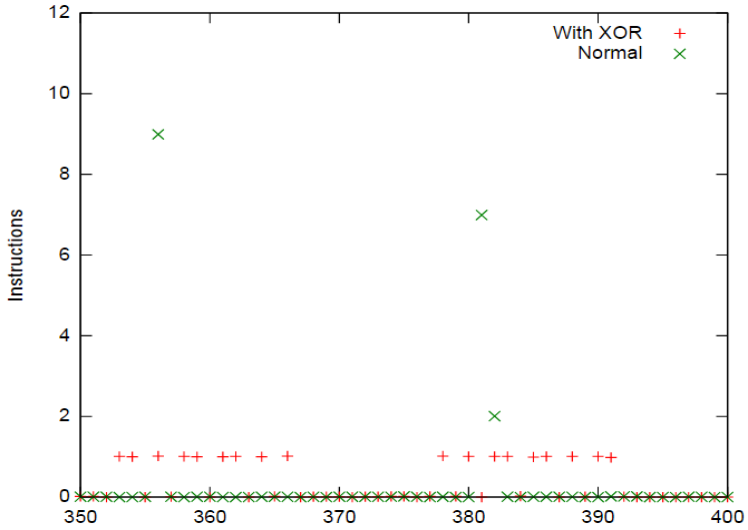
If there were no oscillator drift, we could simply pick the a suitable but sadly that's not the case. The oscillator does drift over time, so we pre-process the signal to remove this. Explictly we plot the difference between the measured time and the local (±3 samples) minimum. This should remove the drift in both the intervalometer's oscillator and the meter's timebase (the latter might be significant because I took the measurements with an Arduino).

Rather than plot the difference in seconds, we'll show it in clock ticks i.e. 2^{-13}s. Despite appearances to the contrary, there's no rounding to the nearest integer: if you look at the data you'll see variation at the ±0.03 instruction level.

It's immediately obvious that the XOR instruction improves the distribution: there are only two different intervals and they vary by a single instruction. By contrast the niave code sometimes generates an interval some nine ticks longer. Before we lose perspective though, that's about one millisecond!



The XOR code isn't perfect though. The plot below shows a small section of fifty intervals, and it's obvious that the longer intervals aren't quite evenly distributed over time. There are better solutions, but most need significantly more than one single instruction.



Oscillator drift

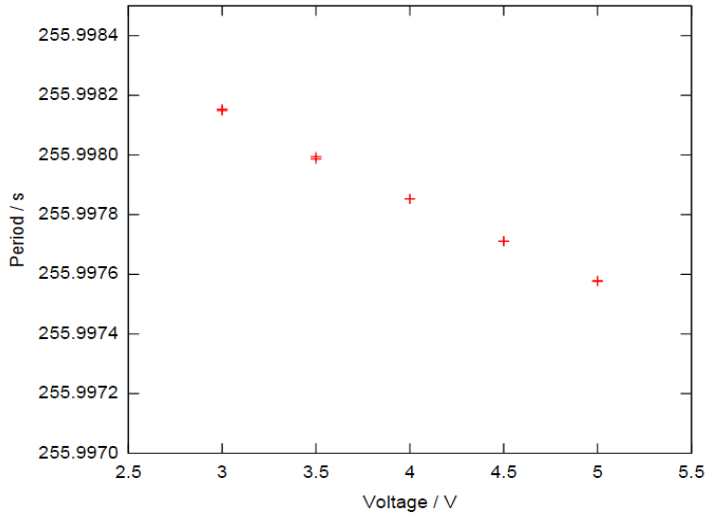
Although it would be nice to ignore it, in practice the oscillator frequency does depend on the environment.

Voltage dependence

It's easy to verify that the frequency depends on the intervalometer's supply voltage, and that higher voltages correspond to higher frequencies.

The graph below shows some experimental data covering the range 3–5 volts.

You'll see that the interval was measured twice at each voltage, in an attempt to isolate the voltage dependence from e.g. coincidentally correlated changes in temperature. Given that the difference between the two measurements is much smaller than the change between successive voltages, we seem fairly justified to claim that it's voltage driving this.



It's clear that the relationship is roughly linear, and a simple least-squares fit gives:

$$\tau(V)=255.99786(1-1.12\times10-6(V-4.0)).$$

The basic story though is that over this range of voltages, there's an approximately-linear fractional-change of about -1.12 × 10^{-6} per volt.

Thus as the battery discharges we'll see a drop of less than 0.5V, which translates to a fractional-change of about 5 × 10^{-7}, so we don't have to worry about it.

On the other hand, if we program the PIC at 5V but deploy at 3V we'll expect the period to rise by about 0.6ms. Accordingly we should tune for 255.9994s under 5V to see 256.0000s at 3V.

Temperature dependence

Quartz oscillators rely on the piezo-electric property of quartz to connect its electrical and mechanical properties: it turns a physical resonance into an electrical one. Accordingly we'd expect temperature, which changes the physical characteristics to change the electrical ones too.

Typically data sheets say that the resonant frequency change with temperature is well-modelled by,

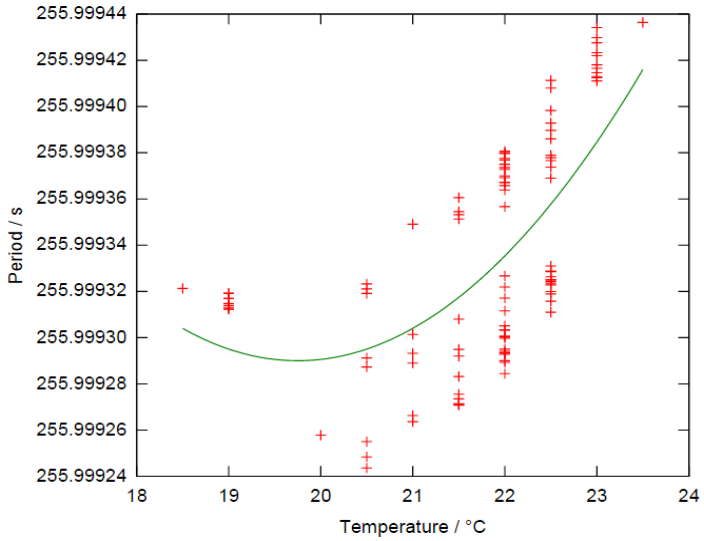
$$f_{res}(T)f_0 = 1 - \alpha(T - T_0)^2,$$

where $\alpha \approx 0.04 \times 10^{-6} C^{-2}$, $T_0 = 25^{\circ} C$.

Given that the effect is small, it's easy to convert this into an expression for the period:

$$\tau(T) = \tau_0(1 + \alpha(T - T_0)^2).$$

Sadly I don't have any sort of temperature controlled chamber to hand, so I just left the intervalometer on the bench for a while, logging the period and temperature automatically. Here's what I found:



The solid line is a parabola fitted to the data by eye. It has equation:

$$\tau(T) = 255.99929(1 + 3.5 \times 10^{-8}(T - 19.75)^2).$$

That seems broadly consistent with what we'd expect though the temperature of the extremum seems lower than I'd expected.

It seems foolish to infer too much of a quantitative nature from these data: they're just not good enough:

1. The temperature measurements are only precise to the nearest 0.5°C, and could easily have have a few degrees of systematic inaccuracy.

2. The time measurements appear to have an interesting likelihood structure which probably comes from the algorithm used by the counter (a TTi TF930). For example, one sees a gap of about 50µs between the clusters of points at given temperature. That's roughly the period of a 16kHz clock: two ticks of the intervalometer's master oscillator or about half an instruction. Neither of those seem a particularly good explanation.

Overall though, it seems reasonable to say that if we'd expect if the temperature remains within 20±5° C the clock won't vary by more than about 1ppm. In other words, providing we're not working outside, we can forget about the problem.

Source: http://www.mjoldfield.com/atelier/2011/08/intervalometer.html