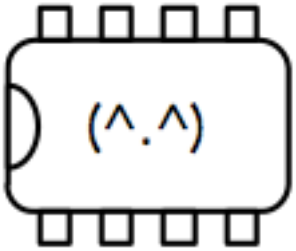
ZDK Saga ch2

Chronal Displacement Monitoring



www.ZENMCU.com

Draft 1 2023-04-05

Chronal Displacement Monitoring

This chapter is actually a reworked version of some code comments that I've removed, to preserve here, while cleaning up some of the ZDK timer code.

When I am writing application code and I want something done after X amount of time, or every so often, I generally couldn't care less about timers and clocks and ticks and prescalers. I just want to say something like: after(time, do_this); or every(interval, do_this); I don't want to deal with timer hardware and interrupts or any of that sort of thing.

But making actually good, useful, general purpose abstractions for these sorts of things is difficult. For example, consider something like:

```
after(t, fn);
```

This schedules a callback to occur in t ticks, and the rest of the code can continue to execute without delay. Of course, there is some machinery behind this. It keeps a linked list of time intervals sorted in order of least ticks remaining. When the timer interrupts it subtracts the time that just expired from all timers in the list. Then it removes any that hit zero, and invokes their callbacks. At that point the short timer is the first in the list, so it re starts the hardware timer for that amount. Lather, rinse repeat.

There could be more machinery to defer the callback onto a queue so that it can be invoked in the foreground rather than the interrupt context, but I have not built that. I'm trying to keep this simple, though I admit that the linked list is more complicated than just having one interval going at any point. However, I felt that was necessary since I often tend to want multiple intervals running concurrently, as different parts of the system do different things at different times.

One thing that I've toyed with is a way to leverage the preprocessor to get more of a Lisp like lambda thing working, something like:

```
after($t, $code);
```

I worked out a way to possibly do it using GCC's nested functions, but those can't be called once the outer function returns, so it doesn't feel like a good idea. Also, any actual code in the $code parameter cannot have commas in it or it confuses the outer macro expansion.

I went ahead and tried this for EVERY($t, $code) and it does work. I didn't use nested functions, but I did expand $code inline. And, if you avoid commas, you can put decent sizes chunks of code in there. Now you can do things like:

```
EVERY(20, after(r, r_off); after(g, g_off); after(b, b_off); );
```

And that sort of thing is, in some cases, quite useful. I even implemented PTEVERY() for use with PThreads. So I like having it, for when I need it, even if it is a bit tricky to use properly.

I did give quite a lot of thought to adding something periodic along the lines of:

```
TIMER* tmr =every(t, fn);
```

But that has baggage. First of all, some way to cancel an every timer is necessary. And if there's more than one running the cancel function needs a parameter to identify which one.

So every() needs to return one to use as a cancelling handle. And that puts the burden on the application to remember it. Not only that, but now the callback function needs a pointer to the timer as a parameter so that it can be cancelled in the callback. It is easy to see how this becomes somewhat baroque.

Therefore, I decided that every() is best implemented by making another call to after() in the callback. That, of course, isn't perfect since now the callback has to know the period and the callback. But it is simple, and reasonably light weight.

Of course, I did manage to do things I ended up regretting. While I was working out how I wanted to measure pulses, such as for timing how long it takes from now until a pin changes state, I wanted a timeout. Essentially, "time how long it takes for this pin to go low, up to some maximum timeout value".

Somehow I convinced myself that to do this I needed to be able to start the timeout timer, and loop watching the pin, and if it changed state, cancel the timer and ask it how much time had actually elapsed. And this became ... distasteful. I had to add a 'total' time to the timer struct, making each one 4 bytes larger. I made after() return a TIMER* and the callbacks accept a TIMER* and I implemented timer_elapsed() etc. And, ok, it did work and although I wan't terribly happy with it I managed to make use of it.

But it was bothering me that my carefully crafted simplicity had managed to mutate under my watchful eye. That, in fact, I had caused that mutation. So I went back to thoughtful mode and made the "hmmm..." sound a lot. What I came to realize was that my venerable software STOPWATCH implementation was very simple and would actually work for measuring pulses, except for the fact that it works off the millisecond timer.

```
typedef volatile MSECS STOPWATCH;
#define start_stopwatch($sw) $sw =msecs()
#define read_stopwatch($sw) (msecs() -($sw))
...
ensystick(); // enable the msec timer
STOPWATCH sw; // make a stopwatch
start_stopwatch(sw); // start it
... do stuff ...
MSECS elapsed =read_stopwatch(sw);
```

I mean, it's minimalist, simple, effective, and crystal clear what it does and how to use it. I personally think this is a solid abstraction. So I decided that I needed a microsecond resolution variant. Enter, the HWSTOPWATCH.

Its important to realize that the software stopwatches run off the msec timer, and so they can just all msecs() which accesses g_msecs which increments once every millisecond and roll over after about a minute with FCPU at 48MHz (its on the SysTick). So there can be a bunch of STOPWATCHes in play at the same time. But the HW stopwatch doesn't have a convenient variable updated by a timer interrupt.

Instead, it has a count register inside a hardware timer module. Since there is only one such module, THE stopwatch, it is only possible to measure one thing at a time with it. And that's ok, since generally you don't want other things holding off the measuring code and skewing the results.

So, now that I have a HWSTOPWATCH that works like the (in my opinion somewhat elegant) software STOPWATCHes, I no longer need the cancel_timer(), timer_elaspsed(), returning and passing TIMER*'s around, sort of nonsense that I had knowingly, though unhappily, added.

For my own amusement, here is an example of the sort of wrong-think I was getting up to:

> *I've had after() returning a TIMER*, along with cancel_timer() for a while. So my earlier concerns about avoiding this complexity are no longer valid. I mean, concern about complexity is valid, but in the end I needed this to get pulse measurement working the way I wanted it to work. So now I have it available, so I might as well make use of it for a version of every().*
>
> *If I make every() be like after() with a callback fn, and it just repeats until you cancel it, then I lose the ability to code the body of every() (the everybody?) inline in the call, which I am somewhat fond of. So I need a new name for this new thing. I am thinking of calling it after_every(ticks, fnptr) to avoid breaking all of my uses of every(). Also it needs a bit of state to tell it to repeat, which will eat more RAM, so I have not implemented it yet...*

I mean, yeah... I did have such thoughts. But now I'm thinking clearly again and putting things back to rights. I only have a few places where I actually used that nonsense, and I can consider most of it obsolete anyhow. So this change back to a kinded, gentler, thousand points of light sort of implementation shouldn't actually break any live code. WHich is a relief.

And that's it for this update, I think.

End