A Brief Time Out

Posted on **May 19, 2012**

In an embedded system, there are many occasions when a delay in execution is required. For most of these, I prefer to make use of the task sleep APIs of most RTOS packages. There are times however when no RTOS is present, or the time interval is too short for an RTOS. In those cases, a dedicated delay library comes in handy.

The quick and dirty approach is just to spin loop, like this:

```
#define DELAY_1MS 16000/5 // for 16MIPS
void DelayMs(WORD time)
{
   unsigned delay;
   while(time--)
      for(delay=0; delay<DELAY_1MS; delay++);
}
```

This code leaves a lot to be desired. For starters, it’s sensitive to compiler optimization, caching, pre-fetching or other system speed ups that can throw off the time or even eliminate the delay. Another defect is that is is hard to slip in other duties without throwing off the delay accuracy quite severely.

Better solutions usually revolve around the use of a timer peripheral. The PIC32 includes the MIPS 4K core timer. This 32 bit timer runs at the peripheral bus speed (typically 40MHz) and its value can be read. The are many other capabilities too, but for this delay library, reading the counter value will suffice.

To start our timer, we need to record the current value of the core timer. This is the starting point or “now” in our delay:

```
#define emStartDelay() ReadCoreTimer()
```

Not much here, just a macro around a read to the core timer. This returns a 32 bit value that is the starting point of the delay. Now as time goes by, the core timer will increment. The amount of time that has passed is simply: Current Timer Value – Start Timer Value. This holds true, even if the core timer should overflow. Now, if this elapsed time is greater than the desired delay, the timer has expired.

```
if ((ReadCoreTimer() - start) >= duration)
{
   // Stuff here
}
```

Note that a timer can be started and its end detected all without modifying the steady counting of the core timer, or using up any limited resources. Thus a large number of software timers can be created without any difficulty. This simple bit of code is the heart of the following delay library interface:

```
#define emStartDelay() ReadCoreTimer()  // Get a starting point.
B8 emIsFinishedRaw(U32 start, U32 duration);  // Finished yet?
void emFinishRaw(U32 start, U32 duration);  // Finish the delay!
void emDelayRaw(U32 duration); // Start and finish a delay!
```

The astute reader will have noted that most of these functions end with the word "raw"; why is that? Simply these functions deal with time only in terms of core timer ticks. For my API, I want to deal in abstract, standard time units. I chose nanoseconds because they are a standard time unit and second, they hint at the sorts of time scales I'm aiming for. To help with the transition from nanoseconds to ticks, a few macros help out:

```
#define NSPT_NUM (1)  // NSPT_NUM / NSPT_DEN equals the number
#define NSPT_DEN (25) // of PB tics per nanosecond.
```

These two integers form a rational number that represents tics per nanosecond. No floating point required or desired! At 40MHz, the core timer step period is 25ns, so each nanosecond is 1/25 of a tic. Nanoseconds are converted to tics in the next macro:

```
// Convert nanoseconds to core timer tics.
#if NSPT_NUM == (1)
#define CVT_NS_2_TICS(ns) ((ns)/NSPT_DEN)
#else
#define CVT_NS_2_TICS(ns) ((NSPT_NUM *(ns))/NSPT_DEN)
#endif
```

Note the slight optimization for the (common) case where the numerator is 1. This leads to the actual library interface:

```
#define emIsFinishedNS(start, ns) \
    (emIsFinishedRaw(start, CVT_NS_2_TICS(ns)))
#define emFinishNS(start, ns)      \
    (emFinishRaw(start, CVT_NS_2_TICS(ns)))
#define emDelayNS(ns)              \
    (emDelayRaw(CVT_NS_2_TICS(ns)))
```

Which is a flexible delay library using standardized units. Of course, Other units, like microseconds or milliseconds could be used and creating those routines is left as an exercise for the reader.

Now, let’s see this code in action. Example 1, a simple delay is needed in the code:

```
emDelay(200); // Delay 200ns
```

Example 2, a sequence of code must not complete before a minimum time:

```
start = emStartDelay();
// Other code goes here . . .
emFinishNS(1000); // Make sure at least 1000ns has elapsed.
// Something crucial and time sensitive here
```

Example 3, a state machine must remain in a state for a certain length of time:

```
switch (state)
{
    // other states removed
    case s_0010:
        timer = emStartDelay;
        state = s_0011;
        break;
    case s_0011:
        if (emIsFinishedNS(timer, 2000))
            state = s_0012;
        break;
    // other states removed
}
```

It’s assumed that the switch is executed frequently so that after 2 microseconds, state s_0011 will transition to state s_0012.

So there it is: a flexible delay library with just a core timer and a few scant lines of code. As always, comments and suggestions are welcomed.

Peter Camilleri (aka Squidly Jones)

SHARE THIS:

Print LinkedIn Facebook Twitter

LIKE THIS:

Loading...

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=990]** .