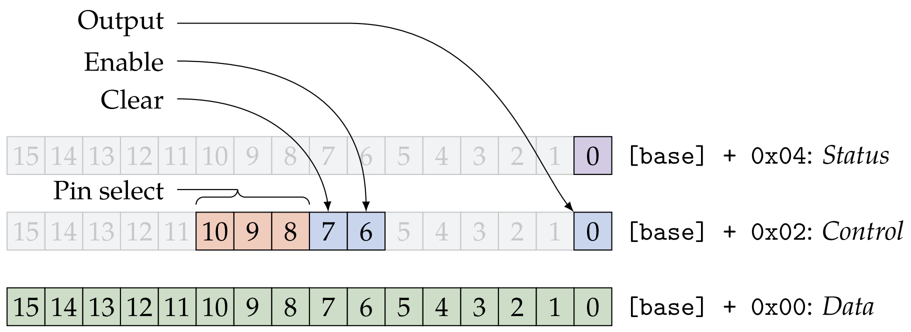
Tutorial 6: Time, Oh Give Me Time!

Prof. John McLeod

ECE3375, 2021 03 04

A model input capture peripheral was presented in the lesson, with the following structure.



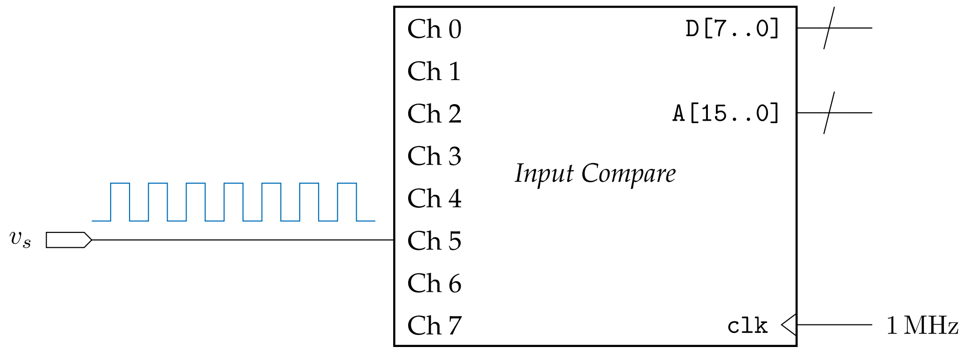
Problem: We have some digital input signal that is a square wave with a 50% duty cycle and a frequency between 1 kHz and 100 kHz. Design a program that uses the model input capture unit to measure the frequency of this signal. Assume the input capture unit uses a 1 MHz clock and is at address `BASE_INPUT_CAPTURE`.

Solution: A square wave with a 50% duty cycle means it spends the same amount of time at a high voltage as at zero voltage. [1] This greatly simplifies the problem, as we only need to test when the input goes high (or low), rather than measuring when it toggles from one state to another.

[1] Namely, a half period $T/2$.

The model input compare has 8 channels. Here I will use "lucky channel 5" to measure the input signal, for good reasons. [2]

[2] Sadly, I do not have time to explain these good reasons, but trust me — they are very good.



This kind of problem is admittedly a bit contrived: we don't have a working implementation of this input capture model, so really it is testing your ability to reason out an abstract approach to solving such a problem. As a potential exam question, *clarity* in your answer is more important than *syntax* — we can't actually run the code anyway! [3]

[3] Having said that, I don't anticipate the midterm exam having design problems like this. I want to keep it short and sweet and easy to mark.

I will begin by defining a data structure for the input capture unit. Here is an example of the ambiguity in this problem: if we had a C compiler implemented for this particular microcontroller architecture, it is quite possible that `int` would be restricted by the compiler to only 16-bit, as that matches the register size — so you would be within your rights to use `int` as the fundamental data unit, and add a note in your answer explaining this.

```
typedef struct _input_compare
{
  uint_16 data;
  uint_16 control;
  uint_16 status;
} input_compare;

volatile input_compare* const icu =
           (input_compare*) BASE_INPUT_COMPARE;
```

To configure this device, we need to write `0b101` (for channel 5) to bits 10,9,8; set bit 7 to enable the timer, and I will choose to set the output of bit 0 to tell the input compare to look for a high signal. The corresponding bit pattern for the control register is therefore `0b10101000001` = `0x541`.

```
//initialize and start ICU on ch5
//look for high input
icu->control = 0x541;
```

The main program can consist of simply looping and checking the status bit of the input compare unit, and reading the resulting time stamp from the data register. With a 16-bit timer and a $1\,\mathrm{MHz}$ clock, any period between $1\,\mu\mathrm{s}$ and $65.536\,\mathrm{ms}$ can be recorded. The expected input range of $10\,\mu\mathrm{s}$ to $1\,\mathrm{ms}$ is well within these limits. However there is one problem: this input compare unit does not detect *transitions*, only *levels*. If we are looking for an input level of 1,

and coincidentally turn on the input compare unit when the input signal is already high, the unit will only record a time interval of 1 μs.[4]
There are a few ways to deal with this, but one relatively simple way is to just expect that the frequency measurement will have some intrinsic errors during the first few cycles. If we are using the input compare unit to detect a high signal, than the half-period of a 50% duty cycle square wave will be the *maximum* elapsed time until the signal goes high. I will therefore define some variable for the period initialized to zero, and keep adjusting it if twice the recorded time interval is larger than the current value.

[4] Or maybe even 0 μs, depending on how the hardware is implemented.

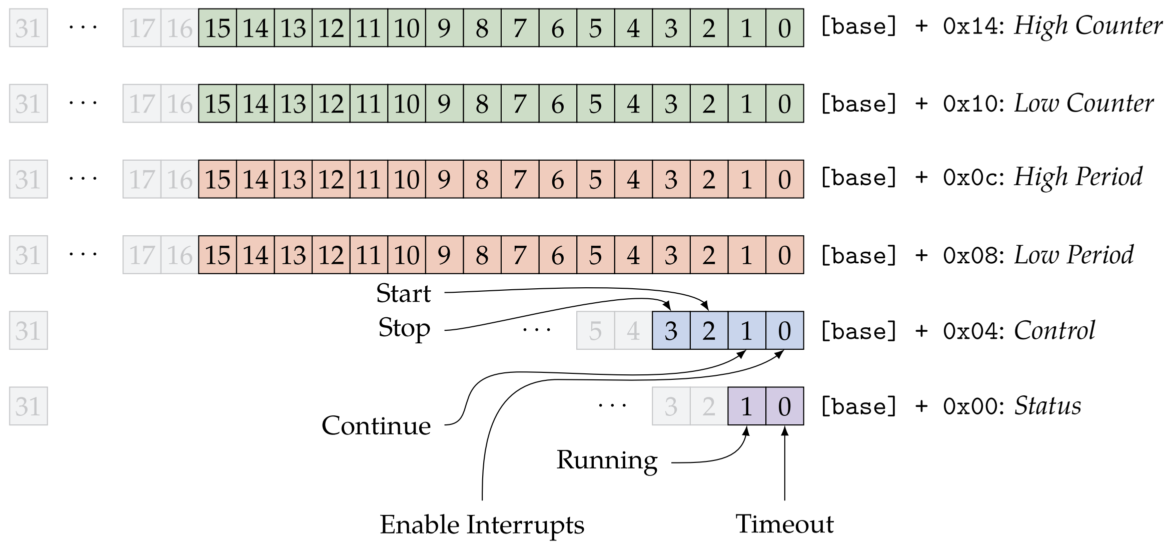
```
// signal period
// in clock cycles (1 ms)
int period = 1;
// signal frequency
// in MHz
float frequency = 0;

while (1)
{
  // check if input compare detects
  // high voltage
  if (icu->status == 1)
  {
    // check if interval is longer than
    // previous estimated period
    if (2*icu->data > period)
    {
      // overwrite old estimate
      period = 2*icu->data;
      frequency = 1/period;
    }

    // clear input compare
    icu->control = 0x80;
    // restart
    icu->control = 0x541;
  }
}
```

If we are unlucky, and start the input compare unit when the input signal is already at a high voltage, it may take several *CPU cycles* before a correct frequency is recorded. This is actually a common trade-off in embedded systems: to make writing the code as simple as possible, it is accepted that the output of the system may not be correct for the first few cycles. [5]

[5] And, actually, in this case this is a necessary inaccuracy. It is physically impossible to accurately measure the frequency of a signal before a single cycle has completed. The implementation shown here may take several *CPU cycles* before the correct frequency is measured, but it should take only one (or, at worst, two) *signal cycles* before the correct frequency is found.

The main problem with this implementation is that if the input signal is slowly tuned to a *higher frequency*, this will not be recorded as it would have a smaller period than previously recorded. A better implementation would be to report some rolling average frequency — but that is an exercise for the reader.

As discussed in the lessons, the DE1-SoC board has two *interval timers* with the following structure:



Problem: Configure the DE1-SoC board to blink LED 0 on/off every $0.5\,\mathrm{s}$, and blink LED 1 on/off every $1\,\mathrm{s}$.

Solution: There are two *interval timers*, so we could set one for a $0.5\,\mathrm{s}$ interval and the other for a $1\,\mathrm{s}$ interval, but that seems like overkill. Instead, I will just keep a running total of the $0.5\,\mathrm{s}$ intervals: the bit 0 will toggle LED 0 at $0.5\,\mathrm{s}$, and bit 1 will toggle LED 1 at $1\,\mathrm{s}$. First, I will use a data structure for the timer.

```
#define TIMER_1_BASE  0xFF202000
typedef struct _interval_timer
{
    int status;
    int control;
    int low_period;
    int high_period;
    int low_counter;
    int high_counter;
} interval_timer;

volatile interval_timer* const timer_1
         = (interval_timer*)TIMER_1_BASE;
```

Recall that these timers have a 100 MHz clock. A 0.5 s period is 50 000 000 cycles.

```
// 0.5 s interval
int interval = 50000000;
// write to low period
timer_1->low_period = interval;
// write to high period
timer_2->high_period = interval>>16;
```

Now I should set up the LEDs.

```
// pointer to LEDs
volatile int* const led = (int*)(0xFF200000);
```

I will start the timer counting continuously, and monitor the status bit. Each time the status bit flags high, I will increment the count accumulator and write the two LSbs to the LED panel. I need to remember to clear the timeout flag on the timer after each 0.5 s interval.

```
// count accumulator
int count = 0;
// mask for 2 LEDs
int led_mask = 3;

// start timer for continuous counting
timer_1->control = 6;

// main loop
while(1)
{
  //check if 0.5 s has passed
  if ( timer_1->status & 1 )
  {
    // increment accumulator
    count++;
    // clear timeout flag
    timer_1->status = 1;
    // write to LEDs
    *led = (count & led_mask);
  }
}
```