Tutorial 8: Now You Know Your ADCs

Prof. John McLeod

ECE3375, 2021 03 18

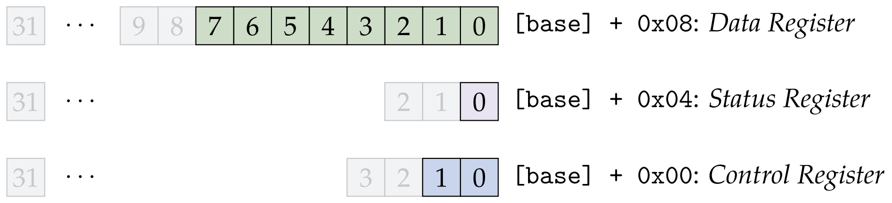
A model ADC was introduced in the lessons. It is a 4-channel, 8-bit ADC, with the following memory-mapped registers:

- A **control register** at the ADC base address. Writing a value between 0 and 3 to this register tells the ADC to start sampling the corresponding channel. Note that it is the "act of writing" that starts the ADC — at the start, this register may be filled with zeros, but if you want to read from channel 0 you still need to write `0x00` to this register.

- A **status register** at the ADC base address offset by `4`. The LSb in this register is normally 1, when it is cleared to 0 it means the conversion has finished.

- A **data register** at the ADC base address offset by `8`. This holds the 8-bit converted value from then channel selected by the **control register**.

Note that this ADC structure only allows one of the four channels to be sampled at a time, as there is only one data register to hold the result.

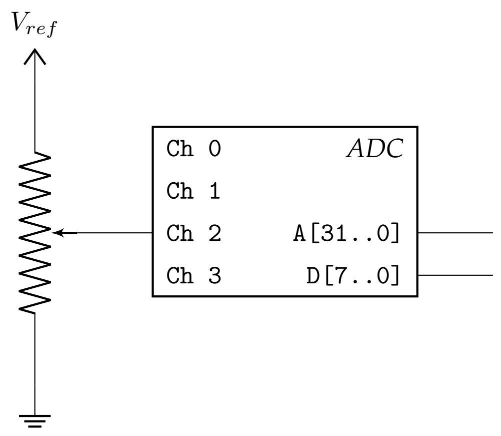

31
· · ·
9
8
7
6
5
4
3
2
1
0
[base] + 0x08: *Data Register*
31
· · ·
2
1
0
[base] + 0x04: *Status Register*
31
· · ·
3
2
1
0
[base] + 0x00: *Control Register*

Problem: Connect a potentiometer circuit to the an channel 2 of the ADC to provide a controlled analog voltage. Use this potentiometer to control the brightness of an LED. Assume the ADC is at base address `ADC_BASE` and the LED bank is at address `LED_BASE`. You may assume other typical model peripherals are available if you need them.

Solution: Hopefully all of you know about potentiometers (or "pots") from your previous courses on electrical circuits. In case you forgot:

- A potentiometer is a simple 3-terminal electromechanical device.
- The resistance between the two end terminals is fixed, while the resistance between an end terminal and the middle terminal (the "wiper") can be manually changed, usually by turning a dial.

For this problem, one of the end terminals of the pot should be connected to the high voltage reference, the other end terminal should be grounded, and the wiper terminal can be used as the input to the ADC. In this manner the pot acts as a manually controlled voltage divider, allowing the ADC to read any voltage between V_{ref} and 0 depending on the position of the dial on the pot.



To start solving this problem, we can define a data structure for the ADC.

```
typedef struct _t_ADC
{
    int data;
    int status;
    int control;
} t_ADC;

volatile t_ADC* const adc = (t_ADC*)ADC_BASE;
```

Note that I need to describe the data and status registers as integers even though only a few bits are actually used. I could define data as `uint8_t` and status as `bool` — but then I would have to align these variables in memory manually. That is definitely more trouble than simply using the LSbs of an integer.
By this point in the course, you should be reasonably proficient at working with peripherals.

- Our entire program should be wrapped in an endless loop.
- The ADC needs to be initialized to sample from channel 2 by writing `0b0x10` to the control register.
- We can then have another loop while we checking the status register until it is done converting.
- The data is then read from the ADC.

```
int main()
{
  int adc_done = 0;
  int adc_data = 0;

  // main loop
  while(1)
  {
    // tell ADC to start conversion
    adc->control = 2;

    // wait until ADC is done
    while(adc_done == 0)
    {
      adc_done = adc->status;
    }

    // read converted data
    adc_data = adc->data;

    // rest of program here...
  }
}
```

So far so good. But wait! How do we control the *brightness* of an LED? The LED brightness can be controlled by varying the voltage to the LED, but here we can only turn the LED on and off with the microcontroller. First, assuming the LED bank is at address LED_BASE, I can define the appropriate pointer.

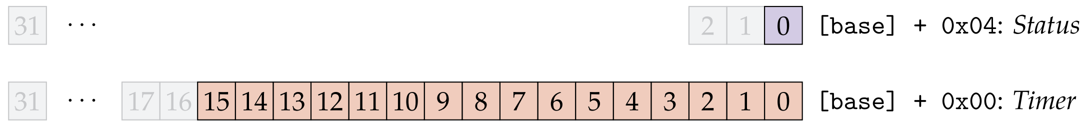
```
volatile int* const led = (int*)LED_BASE;
```

One way to do this is to have the LED flickering on/off, at a rate that is too fast for the human eye to see. Then by controlling the duty cycle of this flickering, the effective brightness of the LED can be adjusted. Implementing this is best done with a timer, so I will use the model timer discussed in the lessons.

The model timer described in the lessons has the following properties:

- This timer always counts **down**.
- The timer interval is written to the *timer* **data register**.
- The current counting time can also be read from this register.
- Bit 0 in the **status register** is set to 1 when the timer reaches zero.

For this problem, I will also assume the timer counts at 1 MHz and has 32-bit registers rather than 16-bit registers (to be consistent with the 32-bit register ADC).



Using a timer in this design requires rather significant adjustment in how the code is written: now we have *two* peripherals, each of which takes time to "complete", and we don't really know when (or in what order) each will finish. For those of you who have watched this week's video lessons, you will recognize this as an example of *event-driven programming*. First, I will define a data structure for the timer, assuming it is at memory address `TIMER_BASE`.

```
typedef struct _t_timer
{
  int counter;
  int status;
} t_timer;
```

```
volatile t_timer* const timer
                    = (t_timer*)TIMER_BASE;
```

There is more than one way to implement this design, of course, but here is one way:

- I will use the 8-bit ADC value to control the duty cycle of the LED: `0xFF` means the LED is on 100% of the time, and `0x00` means the LED is on 0% of the time. Any value in between will be a linear extrapolation from these two points. [1]
- To keep things as simple as possible, I will have the timer count a *tick* Δt, so that the LED will cycle on/off over a period of `0xFF`$\times \Delta t$.
- At timer tick, I will increment a counter. If the value on the counter is *less than or equal to* the current 8-bit value read from the ADC, the LED will be turned on. If the value is *greater than* the current 8-bit value, the LED will be turned off.
- This counter will be reset to 0 after it exceeds 255.
- I am fairly sure that the human eye can't see events faster than $100\,\mathrm{Hz}$, so setting each tick to be $\Delta t = 30\,\mu\mathrm{s}$ will make the LED cycle at $(256 \times \Delta t)^{-1} = (7.68\,\mathrm{ms})^{-1} = 130\,\mathrm{Hz}$, which should be too fast for the human eye to see any flickering.

To keep implementation simple, I will use only one loop for the main program, checking the timer and ADC status each cycle.

[1] It is unlikely that a linear duty cycle translates to linear *visual brightness*, but that is someone else's problem.

- First, we will start the timer and the ADC conversion, then enter the main loop.
- In the loop, first we check if the timer has completed a tick. If it has, increment the tick counter and restart the timer.
- Then check if the tick counter has gone over 255. [2]
- Next, check if the ADC is done. If it is, read in the data and restart the conversion.
- Finally, write a 1 to the LEDs if the current count is less than the ADC value, write a 0 to the LEDs otherwise.

[2] In principle, we could avoid this entirely by declaring the counter as an 8-bit integer (i.e., `uint8_t cycle_count = 0;`) instead of a 32-bit integer, but I like `int`s.

Note that this implementation writes to the LEDs once for each loop through the main program, however we expect each timer ticks to take multiple program cycles to complete, and the ADC may take even longer. Note also that because the ADC input is manually controlled, here it isn't a big problem if the ADC itself updates faster than the timer tick, as there is still no way a human can change the actual input that quickly so the updated ADC value will be basically the same as before.

```
int main()
{
  int cycle_count = 0;
  int timer_tick = 30;
  int adc_ch = 2;

  // start timer
```

```
timer->counter = timer_tick;
// start ADC
adc->control = adc_ch;

// main loop
while(1)
{
  // check if timer complete
  if (timer->status == 1)
  {
    // increment counter
    cycle_count++;
    // restart timer
    timer->counter = timer_tick;
  }

  // check if cycle should be reset
  if (cycle_count >= 256)
    cycle_count = 0;

  // check ADC
  if (adc->status == 1)
  {
    // get data
    adc_data = adc->data;
    // restart ADC
    adc->control = adc_ch;
  }
```

```
    // write to LEDs
    if (cycle_count <= adc_data)
    {
      *led = 1;
    }
    else
    {
      *led = 0;
    }
  }
}
```