ENEL260 CA9: Timer-Driven Peripherals

Ciaran Moore

"Time is of the essence"
– Unknown

Converting between logical values and voltages, as we looked at in the previous lecture, is only one part of the story when it comes to connecting a computer to the real world. In addition to physical quantities (like voltages), to connect to the real world we need to worry about *time*.

Computers operate in discrete steps, executing one instruction at a time. In contrast, the physical world and the phenomena in it experience a continuous flow of time. To connect our computers to the physical world we need some way of relating the discrete steps of the computer to the continuous flow of time in the physical world. One of the basic circuits used to perform that task is the humble *counter*.

Counters

A *counter* is a simple kind of Finite State Machine (FSM) that is designed (see Fig. 1) to output an incrementing (or decrementing, if we're talking about a "down-counter") sequence of binary numbers.

The state diagram for a counter, like the one in Fig. 1, tells us what sequence of numbers the counter will produce. Since the state diagram has no conditions on the transitions, we can assume that each transition from one state to the next simply happens whenever the clock signal ticks. As we saw a few lectures ago, we can design a multi-bit synchronous binary counter (e.g., Fig. 2) by creating *next-state logic* that causes the counter state to change by 1 with each clock edge.

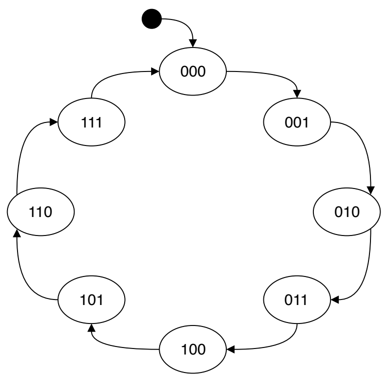


Figure 1: State diagram for a 3-bit binary counter

Note that the state diagram specifies that the initial state is 000. This would typically be achieved by sending a *reset* signal to all of the flip-flops on power-up. To avoid overcomplicating the diagram, Fig. 2 doesn't include the reset signal lines or pins (similarly, power and ground lines are implicit).

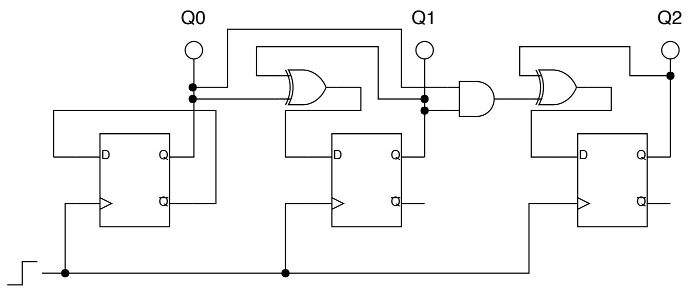


Figure 2: An example of a **synchronous** 3-bit counter.

Fundamentally, counter circuits *count* the number of pulses[1] that have occurred on the clock line. But perhaps the most common use of counters is to connect the clock line to a periodic clock signal that produces regular pulses at a known frequency. In that case, the

[1] For example, if a sensor that generates a pulse each time it detects some interesting event (e.g., a wheel revolution, the appearance of a stoat, or a customer entering a store) is connected to a counter circuit, then the counter will accumulate a count of the number of interesting events that have occurred.

counter will count *clock periods*. Which is another way of saying that the counter is counting the passage of *time*.

Timers

TIMERS ARE SPECIAL COUNTERS THAT ARE PROVIDED AS PERIPHERALS on microcontrollers. At its most basic, a hardware timer provides the software with a way to access time-related information. For example, if the timer clock frequency is known, then the time between events can be computed from the timer count corresponding to the occurrence of each event.

Beyond simply providing timing information, many hardware timers offer the ability to offload timing-related processing from the software onto the hardware. This not only frees up instruction cycles for other tasks, but usually results in much more precisely timed execution[2] of the timing-related tasks. Timers are used for setting up regular internal events, for measuring the timing of external events, and for generating waveforms. Special "watchdog timers" can be used to reset a processor that has become stuck doing one thing for too long.

[2] Hardware timers operate on well-defined clock cycles and the hardware has known and predicable timing characteristics. In contrast, the timing of software execution can be hard to predict–you may not know exactly what assembly-level instructions the compiler has generated, and the instructions themselves may execute in unpredictable ways depending on the architecture of the processor.

Example: Imagine that you want an LED to flash on and off at a particular frequency. Here are two possible ways to accomplish that task:

- *Method 1:* Write a C function that's designed to perform a short calculation a pre-selected number of times;[3] i.e., the function generates a time delay. Implement a `while`-loop that uses an I/O instruction to turn the LED on (via an I/O port), calls the delay function, turns the LED off, calls the delay function, and repeats. The microcontroller is thus directly and continuously working to keep the LED flashing.
- *Method 2:* Set up one of the timers to count up from 0 at a specific clock rate until it reaches a particular maximum number, N, then down until it reaches 0; the sequence then repeats (without the microcontroller's intervention). The count is constantly compared with the number $N/2$; if count $\geq N/2$, an output connected to the LED is set to 1 to turn the LED on, and if count $< N/2$, the output is 0 and the LED off. By initially choosing the clock rate and N, we can precisely set the cycling rate for the LED. The program executing on the microcontroller simply configures the timer and sets it to run.

[3] Performing the short calculation consumes a small amount of time. Repeating the calculation consumes multiples of that small amount of time.

Both these approaches will cause the LED to flash. The main difference is that adding extra processing to the program in Method 1 will require the delay function to be re-calibrated, and is unlikely to ever achieve a precise flashing rate, while Method 2 allows the controlling program to do other processing without affecting the LED flashing rate.

The AVR ATmega32U2 microcontroller has several timers. To quote from the ATmega32U2 datasheet, the microcontroller has:

- One 8-bit Timer/Counter with Separate Prescaler and Compare Mode (two 8-bit PWM channels).
- One 16-bit Timer/Counter with Separate Prescaler, Compare and Capture Mode (three 8-bit PWM channels).

The specific timer features mentioned on the ATmega32U2 datasheet provide various time-related functionality beyond simply counting the passage of time:

- The *'prescaler'* allows the timer to be clocked at a rate lower than the microcontroller itself.
- The *'compare'* mode allows the count to be compared to a number pre-loaded into a special register (useful for generating PWM waveforms).
- The *'capture'* mode allows the instant of an external event to be timed.
- *'PWM channels'* refers to generating timer-based digital waveforms.

Let's look at a couple of these features in more detail.

Pulse-Width Modulation (PWM)

Pulse Width Modulation, or PWM, refers to a method of changing (modulating) the width of repetitive pulses in a signal. The **duty cycle** of the resulting waveform is the ratio of the interval that the waveform is HIGH each pulse ("width") to the period of the waveform, expressed as a percentage.

Example:

- A signal that is HIGH for 10ms and has a 100ms period has a duty cycle of 10%.
- A signal that is HIGH for 200ns and has a 250ns period has a duty cycle of 80%.

Roles for a PWM generator include:

- representing a varying quantity (the duty cycle as a function of time);
- controlling the speed of a DC motor via an electronic switch; and
- generating an analog voltage (by passing the digital PWM waveform through an analog low pass filter, possibly as simple as a resistor and capacitor combination).

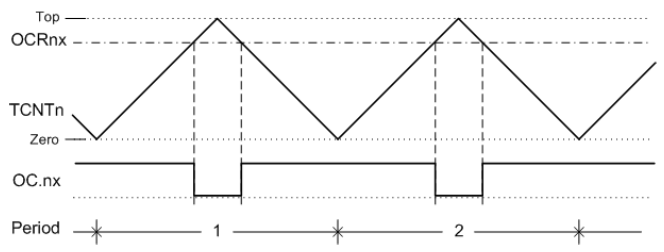


Figure 3: Generating a PWM waveform by means of a count-up/count-down counter and a digital comparator.

Fig. 3 (a simplified version of what appears in the ATmega32U2 datasheet) shows how a timer can be used to generate a PWM waveform.

In Fig. 3:

- `TCNTn` is the *Timer Count* register for timer `n` (`Timer0` and `Timer1`)
- `OCRnA` is the *Output Compare* register for the timer
- `OCRnx` is a second *Output Compare* register for the timer
- `OC.nx` is an output pin associated with the second *Output Compare* register

The timer is configured to count from the value `0` to the value `Top`, which is stored in `OCRnA`. When the timer reaches `Top`, it commences counting back down to zero. The up/down counting sequence then repeats. In PWM mode, the value in the `OCRnx` register is continuously compared with the value in `TCNTn`, thus[4]:

if (`TCNTn` $<$ `OCRnx`), `OC.nx = 1`, else `OC.nx = 0`

The period of the PWM waveform depends on the value of `Top`, and on the clock frequency to the timer. The *duty cycle* of the PWM waveform depends on the value of `Top` and the value loaded into `OCRnx`.

$$\begin{tikzcd} \bullet \arrow[r] & \text{UP} \arrow[r, bend left, "\text{TCNTn = TOP}"] & \text{DOWN} \arrow[l, bend left, "\text{TCNTn = 0}"] \end{tikzcd}$$

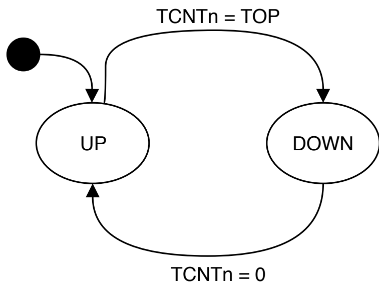


Figure 4: The up/down behaviour of the PWM generator can be viewed as an FSM.

[4] This describes a "digital comparator".

Example: If `Top = 0xC0` (decimal 192) is loaded into `OCRnA`, and the timer clock frequency is 10 kHz, it takes 192 clock cycles (where 1 clock cycle is $\frac{1}{10\times10^3} = 0.1$ msec) to reach `Top`, and another 192 clock cycles to get back to zero. Therefore the period of the output waveform is

$$P = 2 \times 192 \times 0.1 = 38.4 \text{ msec.}$$

When `Top = 0xC0`, and `0x30` (decimal 48) is loaded into `OCRnx`, `OC.nx` will be HIGH for 48 clock cycles during the up-count, and a further 48 clock cycles on the down-count, or 96 clock cycles over the period of the waveform. Therefore the duty cycle of `OC.nx` is

$$D = \frac{96}{384} = 25\%.$$

Input Capture

The *Input Capture* unit in the ATmega32U2 looks for a rising or falling edge in the signal on an "input capture pin". When an edge-event is detected, the Input Capture unit causes the current value of `TCNTn` to be stored into the `ICRn` input capture register and generates an interrupt. This provides a way to precisely timestamp an event.

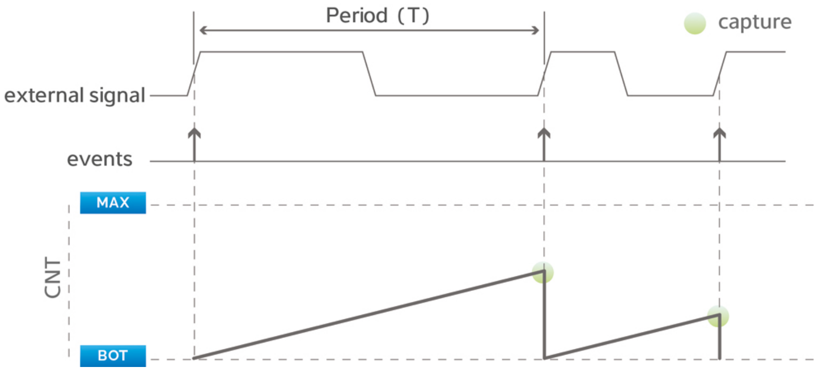


Figure 5: Using *input capture* with a timer to find the period of a repeating pulsed waveform

The value of `ICRn` can be read and stored by an interrupt handler. The stored timestamps can be used to create logs of events, as well as to compute duty cycles, periods or frequencies (Fig. 5), and other time-based signal features (Fig. 6).

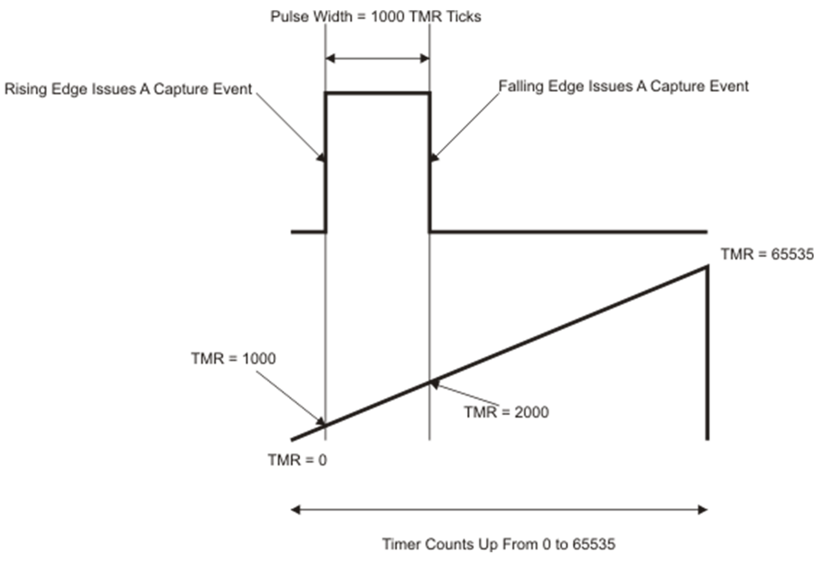

Pulse Width = 1000 TMR Ticks
Rising Edge Issues A Capture Event
Falling Edge Issues A Capture Event
TMR = 65535
TMR = 1000
TMR = 2000
TMR = 0
Timer Counts Up From 0 to 65535

Figure 6: Using *input capture* with a timer to measure the duration of a pulse

Analog-to-Digital Conversion (A-to-D or ADC)

ALL SIGNALS IN THE REAL WORLD ARE AT SOME POINT ANALOG: the amplitude of the sound waves propagating between a speaker and your ear, the intensity and colour of the light reaching your eye or camera lens, the torque you apply to turn the steering wheel of your car, and so on. However, the vast majority of the *processing* of those signals is now performed digitally. In order for that to happen each analog signal needs to be converted into a digital form.

In designing embedded systems, we use some form of sensor (e.g., microphone, photo-diode, or piezoelectric load cell) to convert a non-electrical analog signal (such as sound waves, light, or force) into an analog voltage. The role of the A-to-D Converter (ADC) is, as shown in Fig. 7, to converter the analog voltage into a *sampled discrete signal*.

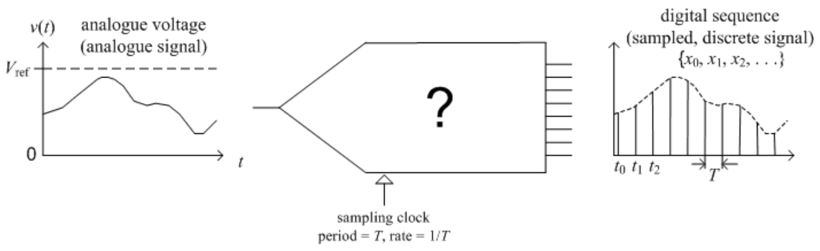


Figure 7: An ADC transforms an analog input into a sampled discrete signal output.

The ADC generates a digital signal by *sampling* the analog signal at regular intervals, and *quantizing* the sampled voltage into discrete values that can be represented by a finite number of bits. Each sample output by an ADC is a binary *code* that represents an approximation of the analog signal value at the time the signal was sampled. The greater the number of bits used to represent the ADC output, the better the approximation to the real signal value.

Example: When a movie camera is used to film a scene, it takes still "snapshots" (or samples) of the scene at regular intervals. The result is a sequence of images that, when played back, look a lot like the original scene.

Typically the time gap between individual frames, or *sampling interval*, is $T = \frac{1}{24}$ of a second. Thus the *sampling rate* (or *sampling frequency*) is $S = \frac{1}{T} = 24$ samples per second.

Example: An analog voltage can take on arbitrary real-numbered values (e.g., anything between 0V and 5V), but an N-bit binary code can only represent 2^N different values. So the sampled analog voltage is mapped to a binary code representing the number nearest to the real value.

- If we only had a 1-bit ADC, then voltages above 2.5V might be mapped to 1 and those below 2.5V to 0.
- If we had a 3-bit ADC, then we have $2^3 = 8$ different codes. So voltages between 0V and 0.625V might be mapped to 000, those between 0.625V and 1.25V to 001, and so on.

Conversion with the ATmega32U2

Source: Atmel application note AVR401.

Many micocontrollers include a built-in ADC peripheral. In some cases, there may be multiple ADCs. However, the ATmega32U2 (in common with a number of other microcontrollers in the AVR range) does not have an ADC peripheral. However, it *does* have an *analog comparator*: an electronic amplifier that outputs HIGH if the analog voltage at the + input is greater that the analog voltage at the – input, and otherwise the outputs LOW. The analog comparator can be used in conjunction with a timer to create an ADC (see Fig. 8).

The capacitor in Fig. 8 is discharged by configuring AIN0 as an output set to LOW. The reference is controlled by pins PD3 and PD4. Configured as outputs set to HIGH and LOW they provide a VCC/2 reference voltage through the resistor network to AIN0. Configured as inputs, they allow the input voltage to reach AIN0.

The analog comparator output can be used as a signal source for the Input Capture unit of one of the ATmega32U2 timers, allowing the times at which the rising capacitor voltage matches the reference or input voltage to be captured. The two measured times required to charge the capacitor (see Fig. 8) can be used to relate the reference voltage to the input voltage:

$$V_{in} = \frac{V_{ref} T_{in}}{T_{ref}}$$

The result of converting time to a voltage is a binary value (either integer or floating point, depending on how we've chosen to perform the calculation) that represents a quantized approximation of the input voltage. By using a timer to regularly trigger this sampling process, we can generate a sampled discrete signal.

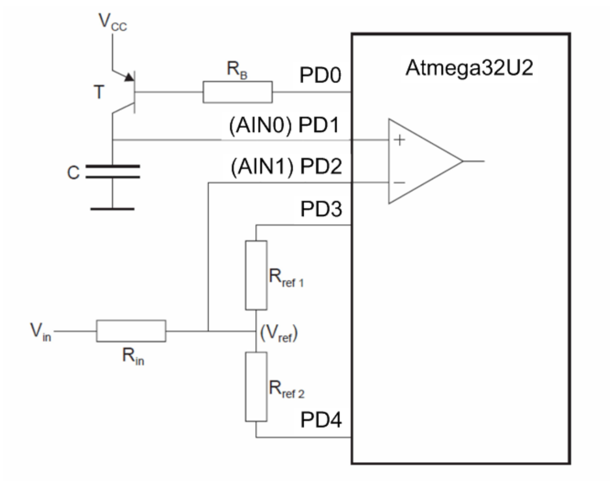


Figure 8: The ATmega32U2 can be connected to a few external components to form a "dual-slope" ADC

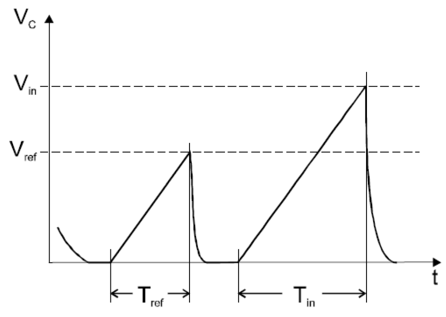
The algorithm used for the conversion is as follows:

1. Turn on the reference.
2. Charge the capacitor until the reference voltage is reached. Measure the time needed for this, T_{ref}.
3. Turn off the reference and discharge the capacitor
4. Charge the capacitor until the input voltage is reached. Measure the time needed for this, T_{in}.

The conversion cycle is shown in Figure 2.

The time measurement is performed by the Timer/Counter, which is expanded to nine bits by using the Timer/Counter Overflow Interrupt.

Figure 2. Conversion Cycle


V_C
V_{in}
V_{ref}
t
T_{ref}
T_{in}