Unit C – Timers and Counters

Counter/Timers Overview

ATmega328P has two 8-bit and one 16-bit counter/timers.
- Can count at some rate up to a value, generate an interrupt and start over counting from 0.
- Useful for performing operations at specific time intervals.
- Can be used for other tasks such as pulse-width modulation or counting external events.

Counter/Timers Overview

Basic idea of counter/timer operation:
- Decide how long an interval is required between interrupts (1 sec, 523 ms, 800 us, etc.).
- Determine a counter frequency, and a counter modulus that will make the counter take that long to count from 0 to the modulus value.
- Configure registers.
- Write an ISR.
- Start the timer.

Counter/Timers Overview

But we already have delay() functions ... why do we need timers?
- Delay functions tie up the processor while executing.
- Better to let a timer measure the delay, and generate an interrupt when complete.
- We can do other useful work while we are waiting for time to elapse!
General Overview of Timer HW

- System Clock (16MHz Arduino)
- 16-bit Counter (TCNTx): Increments every prescaled "clock"
- Modulus A (OCRxA): 0000 0010 0000 0000
- Modulus B (OCRxB): 0000 1010 0110 1100

Start Over @ 0?

We’ll just use the modulus A register so you can ignore B for our class.

Counter/Timer Registers

- Control Register A (TCCR1A)
- Control Register B (TCCR1B)
- Control Register C (TCCR1C)
- Timer/Counter Register (TCNT1H & TCNT1L):
- Output Compare Register A (OCR1AH & OCR1AL):
- Output Compare Register B (OCR1BH & OCR1BL):
- Input Capture Register (ICR1H & ICR1L):
- Interrupt Mask Register (TIMSK1):
- Interrupt Flag Register (TIFR1):

- Interrupt if equal
- Prescalar (1, 8, 256, 1024)

Counter/Timer Registers

- Good News: Can ignore most for simple timing

- Bad News: Lots of register bits to deal with

Computing the Desired Cycle Delay

- Primary step: calculate how many processor clock cycles are required for your desired delay
  - Desired clock cycles = clock frequency \times delay time
  - Arduino UNO clock is fixed at 16 MHz

- Example: 0.25 second delay with a 16 MHz clock
  - Desired clock cycles = 16,000,000 c/s \times 0.25s = 4,000,000 cycles

- Problem: The desired value you calculate must fit in at most a 16-bit register (i.e. max 65,535)
  - If the number is bigger than 65,535 then a prescalar must be used to reduce the clock frequency to the counter from 16MHz to something slower
Calculating the Prescalar

- The counter prescaler divides the processor clock down to a lower frequency so the counter is counting slower.
- Can divide the processor clock by four different powers of two: 8, 64, 256, or 1024.
- Try prescalar options until the cycle count fits in 16-bits:
  - $4,000,000 / 8 = 500,000$ ← too big
  - $4,000,000 / 64 = 62,500$ ← OK
  - $4,000,000 / 256 = 15,625$ ← OK, but not an integer
  - $4,000,000 / 1024 = 3906.25$ ← OK, but not an integer
- In this example, either of the last three could work but since we can only store integers in our timer count registers the last one would not yield exactly 0.25s (more like 0.249984s)

Counter/Timer Initialization 1

- Set the mode for “Clear Timer on Compare” (CTC)
  - WGM13 = 0, WGM12 = 1
  - This tells the hardware to start over at 0 once the counter is reaches your desired value
- Enable “Output Compare A Match Interrupt”
  - OCIE1A = 1
- Load the 16-bit counter modulus into OCR1A
  - This is the value the counter will count up to and then generate an interrupt.
  - The counter then clears to zero and starts counting up again.
  - In C, the register can be accessed as...
    - A 16-bit value “OCR1A”
    - Or as two eight bit values “OCR1AH” and OCR1AL”.

```c
#include <avr/io.h>
#include <avr/interrupt.h>

unsigned char qsecs = 0;

void init_timer1(unsigned short m) {
    TCCR1B |= (1 << WGM12);    // Set to CTC mode
    TIMSK1 |= (1 << OCIE1A);    // Enable Timer Interrupt
    OCR1A = m;                 // Load the MAX count
    TCCR1B |= (1 << CS12);     // Load the MAX count
    TIMSK1 |= (1 << OCIE1A);    // Assuming prescalar=256
    OCR1A = 15625;             // counting to 15625 =
    // 0.25s w/ 16 MHz clock
    // Set prescalar = 256
    // and start counter
    TCCR1B |= (1 << CS12);     // Enable interrupts
    sei();
}

ISR(TIMER1_COMPA_vect) {
    // increments every 0.25s
    qsecs++;          // do something w/ qsecs
}
```

Counter/Timer Initialization 2

- Select the prescalar value with bits: CS12, CS11, CS10 in TCCR1B reg.
  - 000 = stop    Timer starts when prescaler set to non-zero
  - 001 = clock/1
  - 010 = clock/8
  - 011 = clock/64
  - 100 = clock/256
  - 101 = clock/1024
- Enable global interrupts

```c
#include <avr/io.h>
#include <avr/interrupt.h>

unsigned char qsecs = 0;

void init_timer1(unsigned short m) {
    TCCR1B |= (1 << WGM12);    // Set to CTC mode
    TIMSK1 |= (1 << OCIE1A);    // Enable Timer Interrupt
    OCR1A = m;                 // Load the MAX count
    TCCR1B |= (1 << CS12);     // Load the MAX count
    TIMSK1 |= (1 << OCIE1A);    // Assuming prescalar=256
    OCR1A = 15625;             // counting to 15625 =
    // 0.25s w/ 16 MHz clock
    // Set prescalar = 256
    // and start counter
    TCCR1B |= (1 << CS12);     // Enable interrupts
    sei();
}

ISR(TIMER1_COMPA_vect) {
    // increments every 0.25s
    qsecs++;          // do something w/ qsecs
}
```

Counter/Timer Initialization 3

- Make sure you have an appropriate ISR function defined
  - Using name ISR(TIMER1_COMPA_vect)
8-bit Counter/Timers

- The other two counters are similar but only 8-bits.
- Same principle: find the count modulus that fits in an 8-bit value.

---

8-bit Timers (Timer 0 & Timer 2)

- Timer0 (Timer2) of the Arduino only have an 8-bit timer and max count value (thus we can only count up to 255)
- Set WGM01 (WGM21) bit to CTC
- Enable interrupt via OCIE0A (OCIE2A) bit in TIMSK0 (TIMSK2) register
- Load the OCR0A (OCR2A) Register
- Start timer when desired by setting appropriate prescalar

---

ISR Names

- In CTC mode, an "Output Compare A Match Interrupt" will vector to an ISR with these names:
  
  ```
  - ISR(TIMER0_COMPA_vect) {} /* 8-bit Timer 0 */
  - ISR(TIMER1_COMPA_vect) {} /* 16-bit Timer 1 */
  - ISR(TIMER2_COMPA_vect) {} /* 8-bit Timer 2 */
  ```

---

USING STATE MACHINES TO SIMPLIFY & ORGANIZE DESIGNS

Stopwatch Lab
An Example

Let's design a stopwatch (0.1s units)

What are the inputs and outputs

Inputs
- Buttons for start/stop
- Timer

Outputs
- LCD [SS.Tenths time format]

Question:
- What do I need state for in this design?

Answer:
- Anytime you provide the same input and different outputs/actions occur, there is state inside
- Different actions for same button press

LCD Shield Buttons

The LCD shield has 5 buttons
- However, they do not produce 5 individual signals like you are used to from previous labs
- They are configured in such a way such that they sum together to produce a single analog voltage which the shield connects to the A0 input of the Arduino
- If the voltage is in certain range we can infer that a particular button is being pressed

Why Use State Machines

It can be very hard/difficult to design a system where all the inputs can affect each of the outputs (i.e. an all-to-all relationship)
- If n-inputs & m-outputs then all-to-all => m*n cases to account for

<table>
<thead>
<tr>
<th>Button</th>
<th>Volts (V)</th>
<th>Avg. 8-bit Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right</td>
<td>0 V</td>
<td>0</td>
</tr>
<tr>
<td>Up</td>
<td>1.0 V</td>
<td>52</td>
</tr>
<tr>
<td>Down</td>
<td>2.0 V</td>
<td>104</td>
</tr>
<tr>
<td>Left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why Use State Machines

- Easier to decouple relationship between input and output
- Let inputs update state, then examine the state to decide what outputs should be or do => \(m+n\) cases to account for
- Similar to the popular MVC GUI & Web app design approach
  - Model->View->Controller (MVC design)
  - Model (State), View (Output), Controller (Input)

![State Machine Diagram]

Stopwatch Application

- What states do we need to differentiate button presses

![Stopwatch State Diagram]

- When timer interrupt occurs examine the state to decide how to update the display (or just leave current displayed time)
- What else in this design is technically "state"?
  - Time: SS.Tenths
  - Every time the timer interrupts check to see if time needs to update & increment the time if necessary

Suggested Guidelines

- Use a timer to generate an interrupt every 0.1s
- Use the timer ISR to perform time updates and set a flag if display needs updating
- Use state machine approach, polling in main() to detect input button presses and update state and display only if necessary

```c
// Necessary declarations
int main()
{
    // be sure to init. state
    unsigned char state = 0;
    // init and start timer
    // used to check inputs
    // and perform state updates &
    // update display
    while(1)
    {
        // Use a state machine to poll
        // inputs and update state &
        // display, if necessary
        return 0;
    }
    // Use to perform time update
    ISR(TIMER1_COMPA_vect)
    {
        // update time based on state
        // set flag to tell main to
        // update display based on state
    }
```