SECTION SEVEN
TIMER/COUNTERS

SECTION OVERVIEW

- Reuse your code from the previous USART section.
- Learn how to configure and use the 8-bit and 16-bit Timer/Counters on the ATmega128, in both Normal mode and Fast PWM mode.

PRELAB

1. List the sequence of AVR assembly instructions needed to store the contents of registers R25:R24 into Timer/Counter1's 16-bit register, TCNT1. (Assume that the registers R25:R24 have already been initialized with a 16-bit value.)

2. List the sequence of AVR assembly instructions needed to load the contents of Timer/Counter1's 16-bit register, TCNT1, into registers R25:R24.

3. If the 8-bit Timer/Counter0 is configured in Normal mode with no prescaling, and the decimal value "127" is written into the TCNT0 register, how much time will elapse before the timer overflows and wraps around to 0? (Assume that the system clock is 16 MHz.)

PROCEDURE

Problem

TekToy Co. management was very impressed with the proof of concept remote & robot system that you developed in the previous lab. However, one common complaint was that there was no way to adjust the speed of the robot. Your new task is to revise your prototype so that the remote can be used to tell the robot to either increase or decrease its speed.

Specifications

After a bit of planning, you decide that your new prototype will adhere to the following guidelines:

1. Your revised prototype will continue to use the "Device ID + Action Code" command structure, so that a robot will only respond to its own remote control. As a reminder, refer to Figure 1 in the Lab 6 handout.

2. Your remote will still poll the Port D pushbutton switches to determine which action code to send to the robot. However, a generous coworker has provided you with a helper program (debounce.asm) that can improve the reliability of this approach. For more information, please refer to the "Additional Information" section of this document.

3. Since a baud rate of 2400 bps (with 8 data bits and 2 stops bits) worked well previously, you decide to reuse these same USART settings for the revised prototype.

4. The AVR board has two 8-bit Timer/Counters, TCNT0 & TCNT2. These timers, when configured for Fast PWM mode, control pins 7 and 4 of Port B – you recognize that these are the same pins used for Left Motor Enable and Right Motor Enable on your robot! So, you will **use the 8-bit Timer/Counters to control the duty cycle** of the signals applied to the Motor Enable pins, which in turn controls the speed of the robot.

5. Your robot still needs to have the minimal BumpBot intelligence included in the previous prototype. However, this time you decide to **implement the Wait function using the 16-bit Timer/Counter TCNT1 in Normal mode with polling**, instead of the old nested-loop Wait function seen in previous labs.

6. You decide that the robot will have 16 discrete speed levels, with the highest level being full speed, the lowest level being completely stopped, and all levels in between being equidistant. Refer to Table 1 for more details.

Speed Level	Speed
15	100% (255 / 255)
14	~93.3% (238 / 255)
13	~86.7% (221 / 255)
…	…
2	~13.3% (34 / 255)
1	~6.7% (17 / 255)
0	0% (0 / 255)

Table 1 - Equidistant Speed Levels

7. To visually assess the current speed setting of the robot, you decide to use Pins 3-0 of Port B to display a 4-bit indication of the current speed. For example, “0000” would represent Speed Level 0, and “1111” would represent Speed Level 15.

8. You will need to replace some of the action codes used in the previous prototype with the new Speed Up and Speed Down commands. Table 2 shows the revised action code formats.

Command	Action Code
Forward	0b10110000
Backward	0b10000000
Turn Right	0b10100000
Turn Left	0b10010000
Speed Up	0b10000010
Speed Down	0b10000001

Table 2 - Revised Action Codes

ADDITIONAL INFORMATION

Debouncing

MOTIVATION

By now, you have often written code to either poll the state of the pushbutton switches by reading directly from the PIND register, or by enabling some external interrupts which can trigger on certain changes to pushbutton state. However, as you may have noticed, there can occasionally be issues with this approach: for example, falling-edge triggered interrupts which sometimes trigger multiple times, despite the fact that you only pressed the switch once. This is due to a common physical characteristic of mechanical switches known as **bouncing**, where the logical state of a switch doesn't smoothly transition from high to low (1 -> 0) or low to high (0 -> 1); instead, it quickly bounces back and forth between states one or more times, before eventually settling into the correct logical state.

In previous labs, this bouncing was not detrimental enough to prevent the correct operation of your code. For example, bouncing may have caused you to accidentally send a "Move Forward" action code from your remote to your robot several times in a row. But, even though you meant to only send the code once, your robot is still going to do what you intended, which is move forward. For this lab however, you will need to address this bouncing problem in your transmit code. Then when your remote sends your robot a command, **especially a command to decrease or increase its speed**, the command is only sent once per button press. (Do not worry about solving this problem for the bumper pins on the robot.)

To address this bouncing problem, you will perform a technique called software **debouncing**, which uses several consecutive, frequent reads of PIND to guarantee a more smooth transition from high to low or low to high. **Use the provided file debounce.asm** to implement debouncing in your remote code. The rest of this section describes the proper use of debounce.asm.

DEBOUNCE_INIT SUBROUTINE

This subroutine initializes the Data Memory locations that are used to perform software debouncing of the pushbutton switches on Port D, and initializes the 16-bit Timer/Counter1 (TCNT1) to overflow (and generate the corresponding overflow interrupt) every 2 milliseconds. This subroutine requires no parameters, returns no values, and modifies none of the general-purpose registers nor the status register SREG. The following AVR instruction illustrates the proper usage of this subroutine:

```
rcall DEBOUNCE_INIT          ; initialize debouncing code
```

DEBOUNCE_D SUBROUTINE

This subroutine is actually an Interrupt Service Routine (ISR), which must be called from your remote code every time the Timer/Counter1 Overflow interrupt occurs. This subroutine returns one value, the debounced current state of Port D, via the memory location *DebounceResult*; it requires no parameters, and modifies none of the general-purpose registers nor the status register SREG.

Do not call this subroutine directly from the MAIN portion of your program. Instead, place a subroutine call at the interrupt vector for Timer1 OVF, as the following AVR code illustrates:

```
.org    $????                   ; TIMER1 OVF interrupt vector
        rcall DEBOUNCE_D        ; subroutine call to DEBOUNCE_D
        reti                    ; return from interrupts
```

USING DEBOUNCING

Once the debouncing code has been properly initialized by calling **DEBOUNCE_INIT**, and a call to **DEBOUNCE_D** has been placed at the proper interrupt vector, you can utilize software debouncing by following these guidelines:

1. Everywhere that you normally would have read from `PIND` to check the current state of the pushbuttons, you will instead read from memory location *DebounceResult*. **Also, once you detect and then take an action because one of the bits of the byte stored in *DebounceResult* has changed from 1 to 0, you must reset that bit back to 1**. Please see Table 3 below for an example of how to do this.
2. Make sure to include the debouncing code via an `.include` directive at the end of your program.
3. Make sure to enable interrupts globally.

Table 3 below gives two code samples which perform the same task (one without debouncing, and one with): continuously poll the 7th pin of Port D, and whenever a transition from 1 to 0 is detected, write $FF out to Port B.

Without Debouncing

```
MAIN:
    in    input, PIND ; read raw state
    mov   mpr, input  ; copy to temp register
    andi  mpr, (1<<7) ; mask out all bits except 7th
    brne  MAIN        ; branch to MAIN if not pressed
    ldi   mpr, $FF    ; if pressed, perform action
    out   PORTB, mpr
    rjmp  MAIN        ; jump to MAIN to resume polling
```

With Debouncing

```
MAIN:
    lds   input, DebounceResult ; get debounced state
    mov   mpr, input  ; copy to temp register
    andi  mpr, (1<<7) ; mask out all bits except 7th
    brne  MAIN        ; branch to MAIN if not pressed
    ldi   mpr, $FF    ; if pressed, perform action
    out   PORTB, mpr
    ori   input, (1<<7) ; reset bit 7 back to "1"
    sts   DebounceResult, input ; update state
    rjmp  MAIN        ; jump to MAIN to resume polling
```

Table 3 - Polling Pin 7, Without & With Debouncing

STUDY QUESTIONS / REPORT

Write up a short summary that details what you did and why, explains any problems you may have encountered, and answer the questions below. Be sure to explain the operation of your code in the document. Do not rely only on the comments in your code to describe the function of the code. NO LATE WORK IS ACCEPTED.

1. In this lab, you used the Fast PWM mode of both 8-bit timers to toggle Pins 7 and 4 of Port B, which is one of many possible ways to implement variable speed on a TekBot. Imagine instead that you used just one of the 8-bit timers in Normal mode, and every time it overflowed, it generated an interrupt. In that ISR, you manually toggled pins 7 and 4 of Port B. What are the advantages and disadvantages to this new approach, compared to the original PWM approach?

CHALLENGE

Implement **one** of the following improvements to your Lab 7 code.

1. Use Timer/Counter1 **with interrupts** to implement the 1-second Wait function used within HitRight and HitLeft on the robot. Your completed program must correctly perform a full HitRight/HitLeft routine (move backward for 1 second, then left/right for 1 second) without being interrupted by additional bumper hits or received USART data.
2. Use the LCD on the robot to show how many seconds have elapsed since either of the bumpers have been hit.