ENGR 290

Controller programming hints

related to the sample code

1. **Working with ADC.**
   ADC values are stored in the global structure ADC_data;
   The ADC is running in 8-bit mode, which is more than sufficient for the purpose of the course.
   You can access the desired channel by using variable that corresponds to the channel number, e.g. for channel4: ADC_data.ADC4.
   ADC channels numbers correspond to port C pin numbers, e.g. ADC3 is connected to PC3.
   Note that the structure gets updated automatically and asynchronously to the main() process and your code. It is driven by hardware interrupts that are transparent to you and your code. Therefore, if you wish to use the same data more than once in the your formula or function, you have to either create your local variable and store in it the current ADC value OR pass the current ADC **value** to the function through the function call, but you MUST NOT use a pointer.
2. **Servo motor control.**
   Because of specifics of servo-motor control signal, it can be connected to PWM0 and PWM1 channels only. Default is PWM0. So, if you wish to use a servo motor, just use PWM0.
   16-bit timer1, which controls those channels, overflows every 20ms. Timer “top” is 2500.
   Global array “const uint16_t Servo_angle [256]” contains values that you have to write to OCR1A in order to change the position of the servo-motor. Storing in OCR1A value of Servo_angle [0] will turn the servo-motor a bit further than -90 degrees, Servo_angle [127] – middle point, Servo_angle [255] – a bit past +90 degrees. Initially, the table was created to move the servo-motor according to the position of the slider that was connected to an ADC: 256 values cover the entire range of 8-bit ADC. There are 16 steps, so the array works as a kind of “deadband filter”.
   If you wish to calculate the value for OCR1A yourself, use the formula:
   OCR1A=a*2500/20=(a*250)>>1, where a=[1.00;2.00] a=1.00 will bring the servo to -90 degree position, a=1.50 – to the middle, a=2.00 – to +90 degrees. Don’t forget to cast the variables properly – OCR1A is a 16-bit (unsigned integer) register and “a” is a “float”.
   If you wish to avoid using floating point variables (excellent idea!) you can set the range of a=[256, 512] and “>>8” the result.
   OCR1B is connected to P15/2. It can be used for servo if PWM0 is used for power control. Note that a 4to3 pin adapter is required for servo.
   Writing a value [0, 2500] to OCR1B will change the duty cycle form 0% to 100% linearly. There is a macro D1B(x) that converts x=[0, 255] to the appropriate range of duty cycle values.

3. **PWM fan control:**
    a. **PWM1 channel:**
    PWM1 uses channel A of 8-bit timer0. So, writing a value [0; 255] to OCR0A will change the duty cycle from 0% to 100%
    b. **PWM2 channel:**
    PWM2 uses channel B of 8-bit timer0. So, writing a value [0; 255] to OCR0B will change the duty cycle from 0% to 100%.
    c. **On/Off channel 0:**
    The control can turn ON and OFF a fan. It cannot control its speed, but it has higher efficiency than PWM channels because there is virtually no voltage drop across the switch (a relay) vs. voltage drop across Rds(on) of the MOSFETS that control PWM channels. It is connected to PD7, so to turn it on:
    PORTD|=1<<PD7;
    To turn it off:
    PORTD&=~(1<<PD7);
    d. **On/Off channel 1:**
    It is similar to the channel 0, but it is connected to PD4. I let you figure out how to control it.
4. **Using ultrasonic sensors (USS) in PWM mode**.
Unlike the infrared sensors (IRS), the some types USS (e.g. MaxSonar) can be connected through either ADC or a digital pin (PWM mode). While ADC mode is exactly the same as for IRS, PWM mode can be used in different ways. The use of PWM mode for the range measurement requires understanding of real time systems, interrupts, 16-bit values update mechanism on 8-bit systems, etc. Other type of sensors (e.g. HC-SR04) work with force triggering. The triggering can be done by bit-banging the pin or (better way) PWM. Force triggering is a good thing as it helps to avoid crosstalk between two sensors.

HC-SR04

This USS has two data pins: Trigger and Echo.
There are three connectors on the controller that can be used to interface the USS.
P6: INT0 and OC2A – possible to trigger the USS with PWM and capture the echo with interrupt.
P10: ICP and PB4 – bit-bang to trigger and input capture feature of Timer1. NOTE: In the current f/w release input capture register (ICR1) is used for storing the TOP for PWM for servo-motor control. Can ICP feature be used without changing the mode of operation of Timer1?
P13: INT1 and PB5.
The provided harness connects Echo to INT0, ICP or INT1, and Trigger to OC2A, PB4 or PB5.
In general, there are two ways to measure the length of a pulse:

- polling the pin;
- using an interrupt (preferably input capture one).

Both methods require a timer running at the background. Let's see how we can work with an USS connected to PD2 pin (GPIO or INT0) of the extension header.

Polling the pin (GPIO mode)

Check the USS' datasheet for the details on the pulses timing.

1) Trigger the USS (see the datasheet for details).
2) To start the time count we have to detect the rising edge of the signal:

while (PORTD&(1<<PD2)); //just in case, waiting for the signal to go LOW

while (!(PORTD&(1<<PD2))); //waiting for the signal to go HIGH – catching the raising edge – start of the PW pulse.
time_start=TCNT1; //stored the current value of timer1
while (PORTD&(1<<PD2)); // catching the falling edge.
length=TCNT1-time_start; // calculating the number of timer ticks. 1 tick=20ms/2500
Easy, right? Not quite, if we take into account a few things:

- Take a look at DELAY_ms(x) macro. Global 16-bit variable "delay_ms", which gets modified by Timer1 overflow ISR, is initialased when Timer1 overflow interrupt is disabled. **When you read/write TCNT1, the Timer1 overflow interrupt must be disabled!** Otherwise, expect "surprises", when the interrupt happens while you read the TCNT1 lower byte.
- Timer1 "top" is set to 2500, and it overflows every 20 ms. Note that the max PW pulse length is 38ms (no obstacle), i.e. almost double the timer's period. Think how you would address that. Hint: global variable "time" counts the timer1 overflow events.
- It could happen that "time_start" is higher that the value of TCNT1 at the falling edge. Remember, the USS timing is asynchronous to the uCU clock. Thus, the rising edge could happen towards the end of timer1 period (e.g. TCNT1=2300). Then timer1 overflows (TCNT1=0) and starts counting again. Two situations are possible at the falling edge of USS PW output, which arrives after the overflow:
  - TCNT1<2300 (the number is taken just for example)
  - TCNT1>2300

Think how you should deal with those situations.
In addition, measuring PW pulse length in polling mode will "freeze" the system while the function waits for the PW pulse to start and to end.

Hardware interrupts

Using hardware interrupts can solve that issue. You no longer need to wait for PW pulse to go high or low – it's the ISR that will take care of catching those events while your main() code is still running. What you need to include in INT0 or INT1 ISR:

- Check which edge caused the IRQ (read the GPIO pin status).
- If it is low – it was the falling edge, if high – rising edge.
- On rising edge store the value of TCNT1 and … in global volatile variables.
- On falling edge perform the calculations of the PW pulse length. Take into account what I wrote above for the polling mode.

Also, you have to be aware of the situation when if at the start of the system the falling edge comes first, you won't have the start values of TCNT1 and … .

LV-MaxSonar-EZ

Let's see how we can work with the USS in PW pulse mode. In general, there are two ways to measure the length of a pulse:

- polling the pin;
- using an interrupt (preferably input capture one).

Both methods require a timer running at the background. Let's see how we can work with an USS connected to PD2 pin (GPIO or INT0) of the extension header.

Polling the pin (GPIO mode):

According to the datasheet: "LV-MaxSonar-EZ sends the transmit burst, after which the pulse width pin (PW) is set high. When a target is detected the PW pin is pulled low. The PW pin is high for up to 37.5mS if no target is detected."

Thus, to start the time count we have to detect the rising edge of the signal:

```
while (PORTD&(1<<PD2)); //waiting for the signal to go LOW if the function is executed in the middle of acquisition time.
while (!(PORTD&(1<<PD2))); //waiting for the signal to go HIGH – catching the raising edge – start of the PW pulse.
time_start=TCNT1; //stored the current value of timer1
while (PORTD&(1<<PD2)); // catching the falling edge.
length=TCNT1-time_start; // calculating the number of timer ticks. 1 tick=20ms/2500
```

Easy, right? Not quite if we take into account a few things:

- Take a look at DELAY_ms(x) macro. Global 16-bit variable "delay_ms", which gets modified by timer1 overflow ISR, is initialased when timer1 overflow interrupt is disabled. When you read it, the timer1 overflow interrupt **must be disabled**!
- Timer1 "top" is set to 2500, and it overflows every 20 ms. Note that the max PW pulse length is 37.5ms, i.e. almost double the timer's period. Think how you would address that. Hint: global variable "time" counts the timer1 overflow events.
- It could happen that "time_start" is higher that the value of TCNT1 at the falling edge. Remember, the USS timing is asynchronous to the uCU clock. Thus, the rising edge could happen towards the end of timer1 period (e.g. TCNT1=2300). Then timer1 overflows (TCNT1=0) and starts counting again. Two situations are possible at the falling edge of USS PW output, which arrives after the overflow:
    - TCNT1<2300 (the number is taken just for example)
    - TCNT1>2300

  Think how you should deal with those situations.
  Also, you will have to schedule the PW pulse length reading function calls every 49-50 ms as reading the PW signal more often won't bring you any extra data; the same applies to the ADC mode. Moreover, measuring PW pulse length in polling mode will "freeze" the system while the function waits for the PW pulse to start and to end.
  Using hardware interrupts will have the same advantages as for HC-SR04.

  In order to use the interrupt, you have to configure and enable it and write your ISR. The standard names of ISRs can be found here:

  https://www.nongnu.org/avr-libc//user-manual/group__avr__interrupts.html

  Sample code for INT0 (PD2 pin) initialisation:

```
DDRD&=~(1<<PD2); // making sure that PD2 is an input.
PORTD&=~(1<<PD2); // making sure that internal pull-up resistor is not active.
EICRA|=(1<< ISC01)|(1<<ISC00); // Configuring the interrupt: any logical change on INT0 generates an interrupt request.
EIMSK|=(1<<INT0); // enabling INT0 interrupt
```

  The code must be placed in main() before the infinite while() loop.

  The ISR must be placed before the main():

```
ISR (INT0_vect) {
// place your ISR code here
}
```

5. End.

References:

1) LV-MaxSonar®-EZ™ Series High Performance Sonar Range Finder MB1000, MB1010, MB1020, MB1030, MB10402 datasheet (PD11832g).
2) Atmel ATmega328/P [DATASHEET] Atmel-42735B-ATmega328/P_Datasheet_Complete-11/2016.