EE443 - Embedded Systems

Experiment 6

Timers and Interrupts

Purpose: Use timers to generate a Pulse Width Modulation (**PWM**) waveform and to perform certain tasks in an Interrupt Service Routine (**ISR**) with precise timing.

Programming Hints

ATmega328 Timers

Timers in ATmega328 have two basic operation modes that are called "***Capture***" and "***Compare***" modes.

Compare Mode: Timer compares its counter outputs with the numbers previously stored in its registers (i.e. **OCR0A**, **OCR0B**, **OCR1A**, **OCR1B**) every time the counter is incremented. Timer can be programmed to perform an operation when a match occurs between the counter and the stored number:

- Timer can generate an interrupt so that the CPU can execute an ISR after a predefined time delay. Timer works as an alarm clock that tells CPU to perform the task programmed in the ISR.
- Timer can toggle an output pin generating pulses with a predefined interval and duration. Timer works as a signal generator to obtain a digital output, such as a PWM waveform.

Capture Mode: Timer waits for an external trigger signal while the counter increments. The counter outputs are copied or "***captured***" into a register (i.e. **`ICR1`**) when the trigger is received. This mode is used for time measurements on external signals.

If an external input signal is used as the timer clock in any of these modes, then the timer can be used for counting external events or for frequency measurements.

In this experiment two timers will be used. Timer-0 (an 8-bit timer) of ATmega328 will generate a PWM output duplicating the analog input waveform digitized by the ADC. Timer-1 (a 16-bit timer) will generate an interrupt once every **1 ms**. The ISR written for this interrupt will read an ADC sample and send it to Timer-0 to set the PWM output. Following is a list of SFRs that control timer operations.

Register	Description	bit-7	bit-6	bit-5	bit-4	bit-3	bit-2	bit-1	bit-0
PRR	Power Reduction Register	**PRTWI**	**PRTIM2**	**PRTIM0**	---	**PRTIM1**	**PRSPI**	**PRUSART0**	**PRADC**
TCCR0A	Timer/Counter Control Register A	**COM0A[1:0]**		**COM0B[1:0]**		---	---	**WGM0[1:0]**	
TCCR0B	Timer/Counter Control Register B	**FOC0A**	**FOC0B**	---	---	**WGM0[2]**	**CS0[2:0]**		
TCNT0	Timer/Counter 0 Register	**Timer/Counter0 (8-bit)**							
OCR0A	Output Compare Register A	**Timer/Counter0 Output Compare Register A**							
OCR0B	Output Compare Register B	**Timer/Counter0 Output Compare Register B**							
TIMSK0	Timer/Counter 0 Interrupt Mask Register	---	---	---	---	---	**OCIE0B**	**OCIE0A**	**TOIE0**
TIFR0	Timer/Counter 0 Interrupt Flag Register	---	---	---	---	---	**OCF0B**	**OCF0A**	**TOV0**

TCCR1A	Timer/Counter 1 Control Register A	**COM1A[1:0]**		**COM1B[1:0]**		---	---	**WGM1[1:0]**	
TCCR1B	Timer/Counter 1 Control Register B	**ICNC1**	**ICES1**	---	**WGM1[3:2]**		**CS1[2:0]**		
TCCR1C	Timer/Counter 1 Control Register C	**FOC1A**	**FOC1B**	---	---	---	---	---	---
TCNT1H TCNT1L	High & low parts of 16-bit Timer/Counter 1 Register	**Timer/Counter1 - Counter Register High Byte Timer/Counter1 - Counter Register Low Byte**							
OCR1AH OCR1AL	High & low parts of 16-bit Output Compare Register A	**Timer/Counter1 - Output Compare Register A High Byte Timer/Counter1 - Output Compare Register A Low Byte**							
OCR1BH OCR1BL	High & low parts of 16-bit Output Compare Register B	**Timer/Counter1 - Output Compare Register B High Byte Timer/Counter1 - Output Compare Register B Low Byte**							
ICR1H ICR1L	High & low parts of 16-bit Input Capture Register	**Timer/Counter1 - Input Capture Register High Byte Timer/Counter1 - Input Capture Register Low Byte**							
TIMSK1	Timer/Counter 1 Interrupt Mask Register	---	---	**ICIE1**	---	---	**OCIE1B**	**OCIE1A**	**TOIE1**
TIFR1	Timer/Counter 1 Interrupt Flag Register	---	---	**ICF1**	---	---	**OCF1B**	**OCF1A**	**TOV1**

Most of the control options are the same for the two timers. Timer control options are summarized below. The letter "**n**" in register names should be replaced by "**0**" or "**1**" to access the register in the corresponding timer.

PRR – Power Reduction Register

Setting a bit in this register turns off power to the corresponding peripheral module. Therefore, **PRTIM0**, **PRTIM1**, and **PRADC** bits must be cleared to turn on Timer-0, Timer-1 and ADC needed for this experiment. Initially all PRR bits are 0 after power up.

TCNTn – Timer/Counter Register

The counter value can be accessed through this register. It will not be used in this experiment.

OCRnA and OCRnB – Output Compare Register A and B

Numbers written to these registers determine the PWM duty cycle or the interrupt timing. There are two registers, **...A** and **...B**, for each timer that provide two PWM outputs with different duty cycle or two interrupts with different timing.

- In Timer-0, **OCR0A** will set the PWM duty cycle at **OC0A** output.
- In Timer-1, **OCR1A** will set the time between interrupts.
- **OCR0B** and **OCR1B** will not be used.

TCCRnA and TCCRnB – Timer/Counter Control Register A and B

COMnA[1:0] and **COMnB[1:0] :** These bits determine behavior of the output pins, **OCnA** and **OCnB**, when the counter starts and when it reaches the target settings.

- In Timer-0, set **COM0A[1:0] = 10** to obtain the regular PWM output. **OC0A** output will be **1** when counter restarts at 0x00 and it will be cleared when the counter reaches the value in **OCR0A** register. Set **COM0B[1:0] = 00** to disable **OC0B** output. Note that, **OC0A** output driver (pin-6 of port-D) should be enabled in the **DDRD** register.
- Both of the Timer-1 outputs should be disabled.

WGM0[2:0] and **WGM1[3:0] :** These bits select the waveform generation mode. They determine the maximum value timer counters can reach and how the timers restart to repeat operations.

- In Timer-0, set **WGM0[2:0] = 011** to select the **Fast PWM** mode. In this mode, Timer-0 repeats counting from **0x00** to **0xFF**. The PWM output pin (**OC0A**) toggles when the counter reaches the value in **OCR0A**, and when it restarts at **0x00** after **0xFF**.

- In Timer-1, set **WGM1[3:0] = 0100** to select the **Clear Timer on Compare Match** (**CTC**) mode. In **CTC** mode, Timer-1 automatically restarts the counter at **0x0000** after it reaches the number in the **OCR1A** register. An interrupt will be generated every time the counter restarts.

CSn[2:0] : These are the clock select bits that determine the timer clock frequency. Refer to the datasheet to see the clock divider factors.

- In Timer-0, set **CS0[2:0]** to obtain **31,250 Hz** PWM frequency. Timer-0 should count from **0x00** to **0xFF** (256 clock cycles) **31,250** times in one second.
- In Timer-1, set **CS1[2:0]** to adjust interrupt intervals with **1 µs** precision.

Assume that the main clock at the prescaler input is **8 MHz** as usual.

FOCnA, FOCnB : These bits can be used to force an immediate compare match at the timer outputs. They should be set to **0**.

TIMSKn – Timer/Counter Interrupt Mask Register

Writing **1** to a bit in this register enables the corresponding interrupt source.

- Timer-0 will not generate any interrupts.
- Timer-1 will generate an interrupt when the counter matches the number in **OCR1A** register. Therefore, **OCIE1A** bit should be set.

TIFRn – Timer/Counter Interrupt Flag Register

These flag bits indicate the interrupts waiting for service. Interrupt flags will be cleared by the hardware when the target ISR is called.

WinAVR compiler provides macros to access 16-bit settings of Timer-1 in a single statement. For example, the statement,

```
OCR1A = 0xAABB;
```

writes **0xAA** to **OCR1AH** (8 MSBs), and **0xBB** to **OCR1AL** (8 LSBs).

ATmega328 Interrupts

ATmega328 instruction set contains two instructions to enable and disable global interrupt functions:

SEI : Global Interrupt Enable
CLI : Global Interrupt Disable

WinAVR compiler provides two functions, **sei()** and **cli()** representing these instructions in the C code. Mask settings for individual interrupt sources can be found in the related control registers. Using interrupts involves the following steps in the C programs compiled with WinAVR:

1. Include the header file for interrupt function definitions:

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

2. Initialize the MCU settings related to the interrupt source. For example, if the interrupt is generated by Timer-1, then configure all SFRs that control Timer-1.

3. Enable the interrupt source in the related mask control register. For example, Output-Compare-A interrupt from Timer-1 can be enabled by setting **OCIE1A** bit of the **TIMSK1** register:

```
TIMSK1 |= _BV(OCIE1A);
```

4. Enable the global interrupts when the MCU is ready to handle the interrupts. The "**sei();**" statement should be used at the end of the initialization part of the main

program. Enabling interrupts before proper initialization may result in unexpected response of the system.

5. Write the interrupt service routine (ISR). ISRs work like normal C functions, but they cannot take any arguments. Instead, an ISR starts with specification of the interrupt vector which is the target of the ISR call operation related to the interrupt source.

```
ISR(TIMER1_COMPA_vect)
{
// Start with declarations as usual:
  static unsigned char  PulseCount = 0;
  ---
// Write the executable statements:
  PulseCount ++;
  ---
// No return statement is required.
}  // end of ISR
```

The ISR given above is invoked by the **Output-Compare-A** interrupt from **Timer-1**. A complete list of ATmega328 interrupt vector definitions can be found in the **iom328p.h** header file provided with WinAVR. This header file is located in the "<*WinAVR installation dir.*>\avr\include\avr" directory.

In addition to the user statements written in the ISR function, the compiler generates the background code required for handling the interrupt:

- A **`JUMP`** instruction is placed in the program memory reserved for the interrupt vectors to go to the beginning of the ISR function.
- ISR function is terminated with a **Return From Interrupt** (**`RETI`**) instruction.
- CPU registers used in the ISR are saved in the stack memory by using **`PUSH`** instructions at the beginning of the ISR. Register contents are restored with the **`POP`** instructions at the end.

Keeping Local Data in an ISR

Local variables declared in C functions and ISRs use temporary storage locations in the memory by default. The compiler allows other functions to share these temporary locations to save memory. As a result of this, a variable declared in an ISR can be overwritten by another function after the ISR returns. The local variables that are required to preserve their settings should be declared with the "**`static`**" attribute.

The declaration in the ISR example given above,

```
static unsigned char  PulseCount = 0;
```

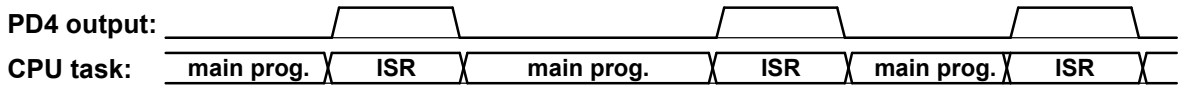
reserves permanent storage for the **`PulseCount`** variable. **`PulseCount`** is initialized to **0** after MCU power up or another master reset condition. It will be incremented to **1** during the first execution of the ISR. **`PulseCount`** will preserve its value until an interrupt invokes the ISR again. ISR will increment **`PulseCount`** by one in every interrupt call.

Using Time Markers in ISRs

It is a good practice to reserve a few general purpose I/O pins for debugging purposes and connect them to test points on the development hardware. These pins can be probed with an oscilloscope to obtain critical timing information while the program is running on the real hardware. PD4 output (pin-4 of port-D) is used to mark the beginning and end of ISR execution in the example given below:

```
ISR(TIMER1_COMPA_vect)
{
// Start with declarations as usual:
  static unsigned char  PulseCount = 0;
  ---
// Write the executable statements:
  PORTD |= _BV(PORTD4);  // Set ISR time marker
  PulseCount ++;
  ---
  ---
  PORTD &= ~_BV(PORTD4);  // Clear ISR time marker
}  // end of ISR
```

The **PD4** output is set right after the interrupt call, and it is cleared right before the CPU goes back to the main program as shown in the following timing diagram.



The marker outputs can also be used as trigger signals to easily capture other related events on the oscilloscope.

Preliminary Work

1. Modify the main program developed for Experiment 5 as follows.

- Keep the statements that initialize the LCD interface and the ADC module.
- Initialize the **DDRD** register to set the direction of port-D pins:
  Pin-0 and pin-1 are inputs (connected to SW0 and SW1 switches)
  Pin-4 and pin-5 are outputs (time marker signals from ISR)
  Pin-6 is output (PWM output from Timer-0)
- Write the necessary statements initializing Timer-0 SFRs to obtain a PWM output. Use **OC0A** output, and set PWM frequency to **31,250 Hz** referring to the Timer-0 clock selection information given in the datasheet.
- Write the necessary statements initializing Timer-1 SFRs to obtain an interrupt once every **1 ms**. Use the **Output Compare A** interrupt, and set Timer-1 prescaler to obtain **1 MHz** timer clock.
- Enable the interrupts right before the main loop and remove all statements from the main loop.

```
// Enable global interrupts:
sei();
// Empty loop (all tasks performed in the ISR):
while(1)
{ };
```

2. Write an interrupt service routine that performs the following operations that were done in the main program previously.

- Read the ADC result from ADC1 input.
- Write the 8 MSBs of the ADC result to the Timer-0 to set the PWM waveform duty cycle.
- Display the 8 MSBs of the ADC result as a 3-digit decimal number on the LCD screen.
- Set the pin-4 of port-D at the beginning of the ISR and clear the pin before the ISR returns.

Procedure

1. Create a new project directory and a new AVR project for Experiment 6. Use the main program and the ISR code you wrote in the preliminary work and add the files **LCDmodule.c** and **LCDmodule.h** to the new AVR project in Code::Blocks.

Compile your program and debug if necessary.

Open the extended listing (**.lss**) file in Code::Blocks, and locate the interrupt vector for the **Timer-1 Output Compare A** interrupt (i.e. **`<__vector_11>`**).

2. Open the design file for Experiment 5 in **ISIS**, and save it with a different name in the new project directory. Select the **.hex** file generated by the compiler for Experiment 6 in the **ATMEGA328P** properties window.

Delete the DAC connected to port-B.

Build an RC low-pass filter with **3 KHz** bandwidth (**-3 dB** corner frequency at **3 KHz**), and connect it to **OC0A** output (**PD6** at pin-12 of ATMEGA328P).

Apply a **50 Hz** sinusoidal waveform varying between **0 V** and **5 V** to the **ADC1** input.

Place an oscilloscope (select the **Virtual Instruments** list on the main toolbar) and probe the following nodes:

a) **ADC1** input of ATmega328
b) Filtered PWM signal at the RC low-pass filter output
c) **PD4** output of ATmega328 (time marker from ISR)
d) **PD5** output of ATmega328 (second time marker from ISR)

3. Test your program looking at the animated simulation results, and debug it, if necessary. Measure the following timing parameters on the oscilloscope window:

- Repetition time of ISR calls.
- PWM frequency.

Make the necessary corrections in your program if the measured values are not as expected.

4. Modify the ISR using pin-5 of port-D as a second marker output to measure the CPU time spent during the following operations:

- Waiting for ADC after starting a conversion.
- Displaying an ADC result on the LCD (i.e. **`LCD_MoveCursor(..)`** and **`LCD_WriteString(.)`** calls).

Question 1: Calculate these timing parameters based on the information provided in the ATmega328 datasheet and in LCDmodule.h and compare with the measured values.

Question 2: What is the other function call that takes significant time in the ISR? How much CPU time does it take?

5. Add an attenuation control variable (i.e. **Atten**) as described in the previous experiment:

Atten = **1**=>1/1, **2**=>3/4, **3**=>1/2, **4**=>3/8, **5**=>1/4, **6**=>3/16,
7=>1/8, **8**=>3/32, **9**=>1/16, **10**=>3/64, **11**=>1/32

Atten setting will be controlled by the two push-button switches:

- Decrement **Atten** (if Atten > 1) every time **SW0** is pressed.
- Increment **Atten** (if Atten < 11) every time **SW1** is pressed.

The ISR should check the switch inputs once in every interrupt call. Remember to use the "**`static`**" attribute when it is necessary for the declarations in the ISR.

Question 3: Suggest some solutions to prevent wasting of CPU time on idle tasks, such as waiting for ADC conversions and LCD operations.