ECE3411 – Fall 2017
Lec 4a.

ADC: Analog to Digital Conversion

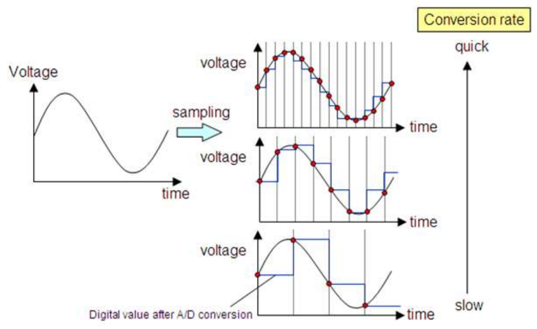
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lecture 5a, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

Introduction

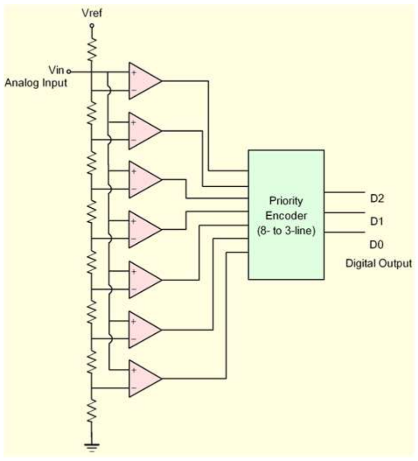
- Why do we need Analog-Digital Conversion?
  - Real world is Analog
  - Digital computers process Digital signals
  - ADC/DAC serve as interface between Computers and Real world!
- Analog Signals are "Continuous"
  - A "Discrete" version of the analog signal is created by "Sampling" the analog signal
  - ADC then maps each sample onto a quantized range of voltages which can be represented by binary values.



2

ADC Types: Flash ADC

- Parallel Design
  - A resistor divider network generates discrete voltage levels
  - Input voltage is compared against all the voltage levels at once
  - Priority Encoder considers the first "HIGH" input from the top as valid, and converts it to binary form.
- Advantage: Fast
  - Conversion takes just one cycle
- Disadvantage: A lot of components needed.
  - $2^n - 1$ comparators needed for n bit ADC

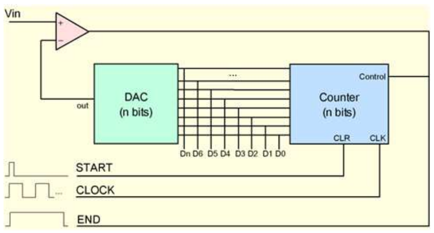


Picture Source: www.hardwaresecrets.com

3

ADC Types: Ramp ADC

- Sequential Design
  - A Counter counts from $0 \cdots 2^n$
  - A DAC generates discrete voltage levels corresponding to the digital values $0 \cdots 2^n$ (i.e. a voltage Ramp)
  - In each cycle, input voltage is compared against the current voltage level generated by DAC
  - The comparator generates a "HIGH" value as soon as the ramp crosses the input value. The corresponding counter value becomes the output.
- Advantage: Only a few components needed.
- Disadvantage: Very slow.
  - $2^n - 1$ cycles (in worst case) for n bit ADC conversion

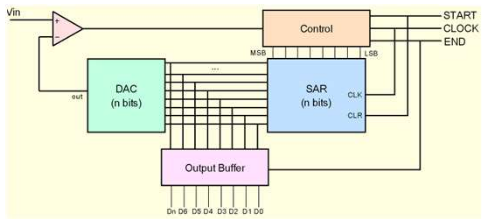


Picture Source: www.hardwaresecrets.com

4

ADC Types: Successive Approximation ADC

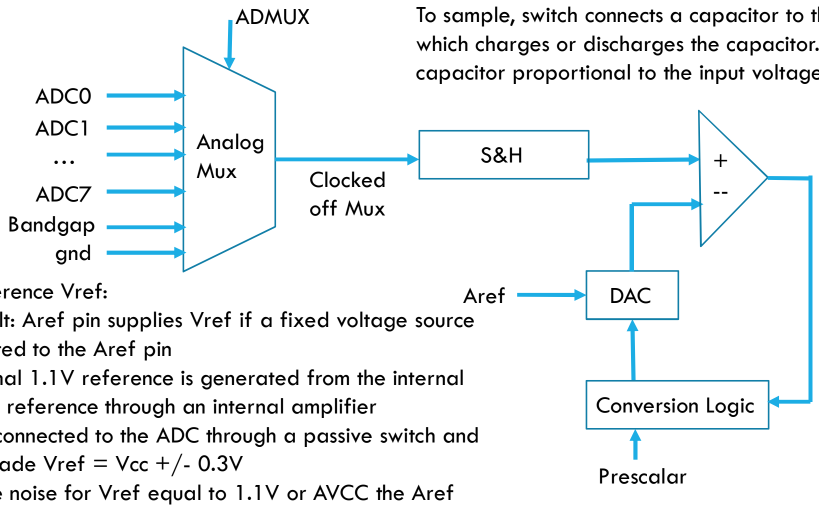
- Sequential Design
  - Closest digital value is approximated by “Binary Search”
  - First, the MSB of SAR is set to 1, and the comparator decides whether the input voltage is higher or lower than DAC voltage. The bit value is adjusted accordingly.
  - The process is repeated for each bit from MSB down to LSB
  - The final SAR value becomes the output.
- Most widely used ADC type.
- Advantages:
  - Only a few components needed.
  - Conversion takes just n cycles.



Picture Source: www.hardwaresecrets.com

ATMega328P ADC Diagram

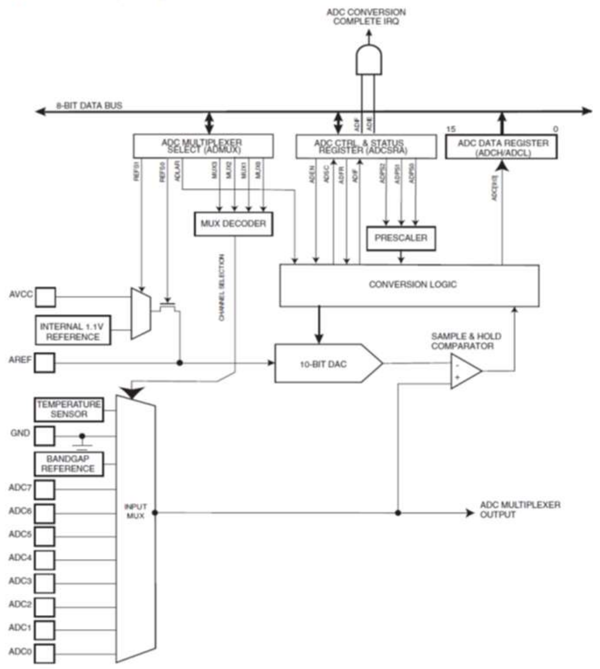
To sample, switch connects a capacitor to the output of a buffer amplifier, which charges or discharges the capacitor. This makes voltage across the capacitor proportional to the input voltage. To hold, the switch disconnects.



Conversion logic implements a successive approximation algorithm (a binary search; one bit per search):
- DAC takes as input the output of the conversion logic and converts it to an analog voltage where Aref sets the full range
- Analog comparator decides whether the DAC output or input voltage is the largest

Voltage reference Vref:
- By default: Aref pin supplies Vref if a fixed voltage source is connected to the Aref pin
- The internal 1.1V reference is generated from the internal bandgap reference through an internal amplifier
- AVCC is connected to the ADC through a passive switch and can be made Vref = Vcc +/- 0.3V
- To reduce noise for Vref equal to 1.1V or AVCC the Aref pin can be externally decoupled by a capacitor to ground

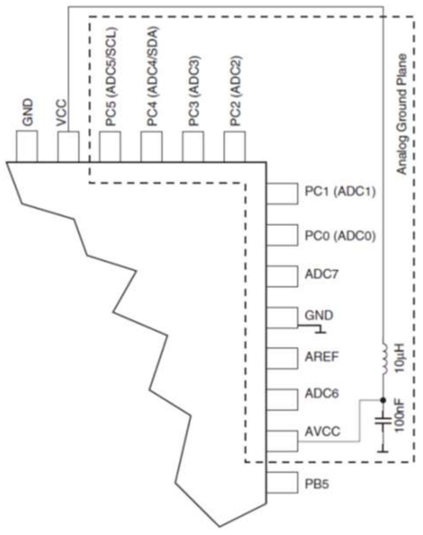
Figure 23-1. Analog to Digital Converter Block Schematic Operation,



7

Pin Assignment

Figure 23-9. ADC Power Connections

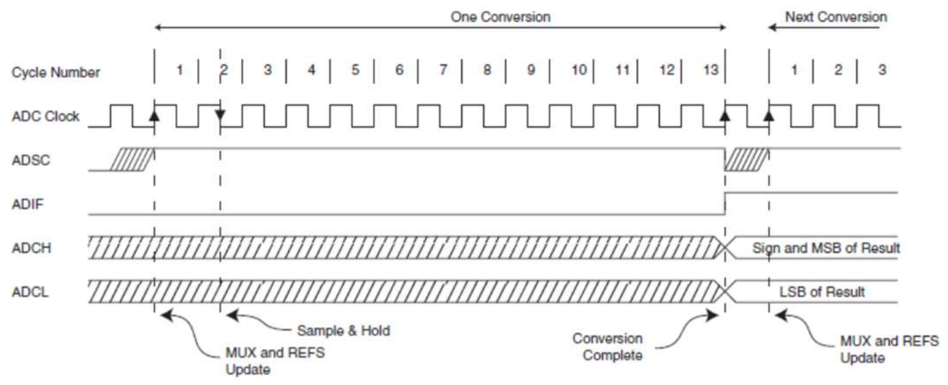


8

Normal Conversion

- Takes 13 cycles

Figure 23-5. ADC Timing Diagram, Single Conversion



Accuracy

- Capacitor in S&H leaks and can therefore not hold a value for too long
  - There exists a minimum sample speed/frequency
- Conversion logic takes time, so we cannot sample too fast
  - There exists a maximum sample speed/frequency
  - The faster you sample, you get a smaller number of accurate output bits (since the binary search cannot completely finish)

By default, the successive approximation circuitry requires an input clock frequency between 50 kHz and 200 kHz to get maximum resolution. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be higher than 200 kHz to get a higher sample rate.

- Noise: MCU produces up to 150mV line noise, there are other sources such as electrical field, etc.
  - Use capacitances close to the CPU to eliminate most of the inductance

Prescalar

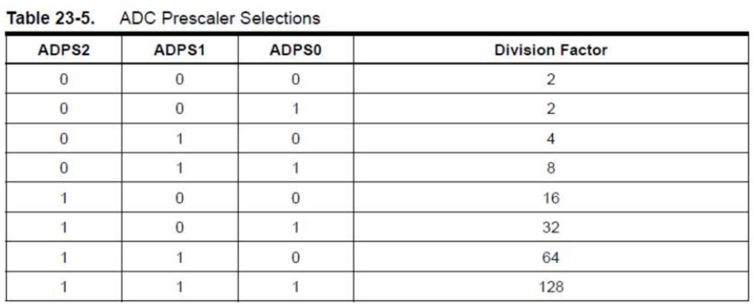


Table 23-5. ADC Prescaler Selections

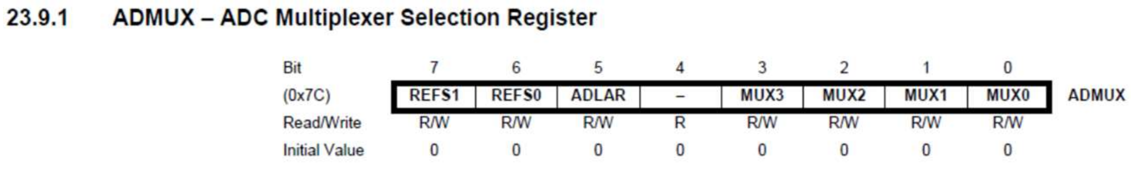
ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

- E.g., a prescalar of 128 gives 16MHz/128 = 125000 (between 50 and 200 kHz)
- To complete the binary search takes 13 cycles = 13/125000 = 104 micro seconds
- Gives 10 bits uncalibrated accuracy at a linear scale to Vref

- CPU clock is at least twice as fast as the ADC's acceptable frequency; therefore the smallest prescalar must be >=2

11

ADMUX Register



23.9.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

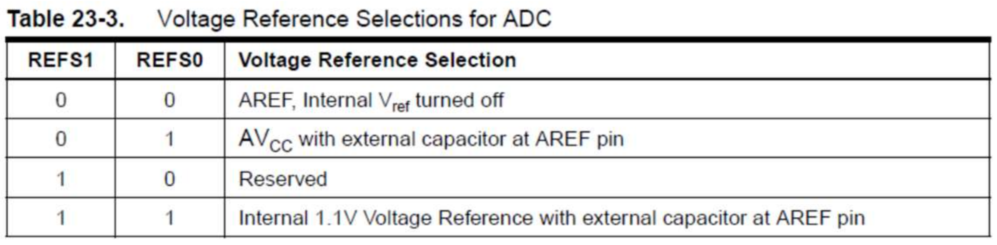
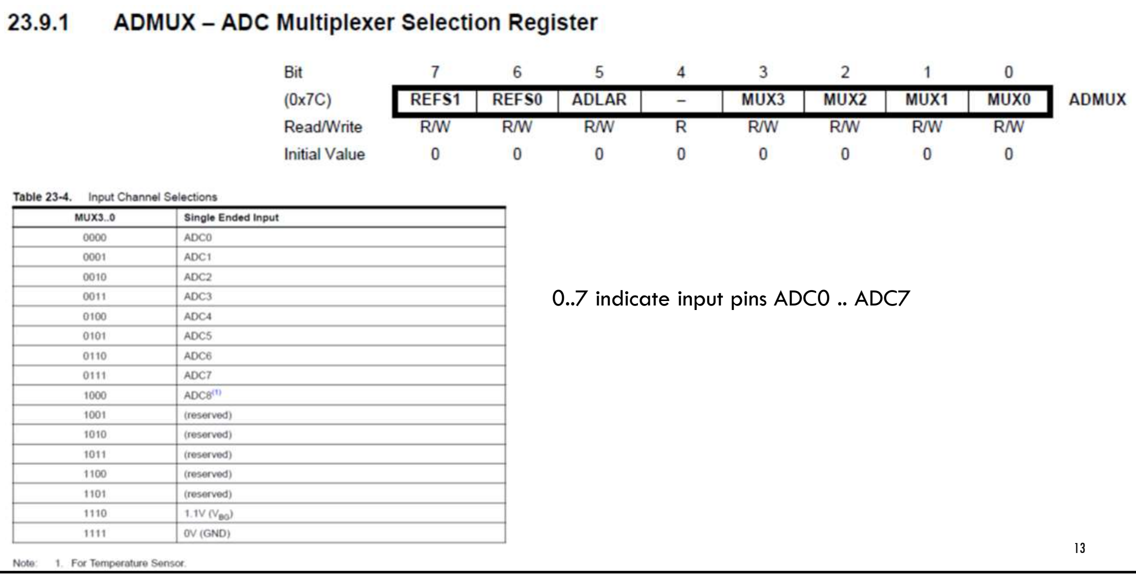


Table 23-3. Voltage Reference Selections for ADC

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal V_{ref} turned off
0	1	AV_{CC} with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

12

ADMUX Register



23.9.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 23-4. Input Channel Selections

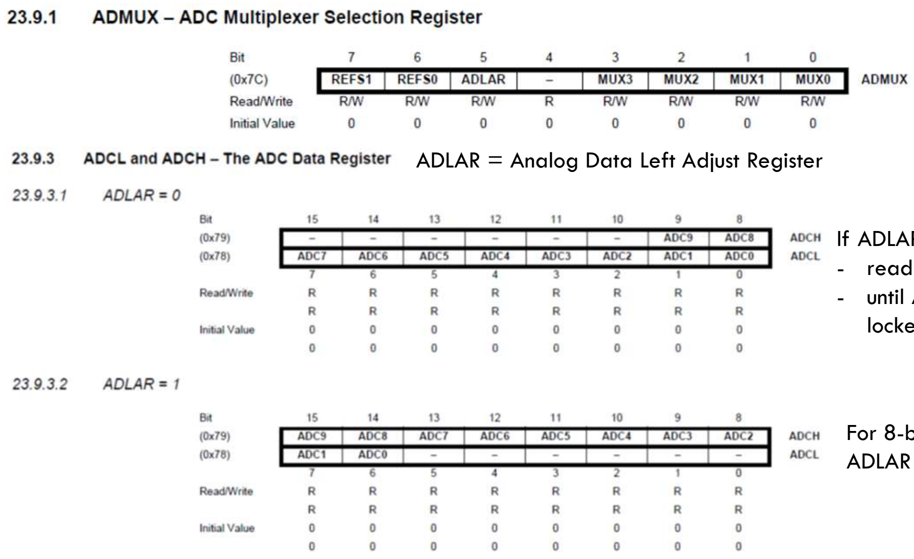
MUX3..0	Single Ended Input
0000	ADC0
0001	ADC1
0010	ADC2
0011	ADC3
0100	ADC4
0101	ADC5
0110	ADC6
0111	ADC7
1000	ADC8[1]
1001	(reserved)
1010	(reserved)
1011	(reserved)
1100	(reserved)
1101	(reserved)
1110	1.1V (V_{BG})
1111	0V (GND)

Note: 1. For Temperature Sensor.

0..7 indicate input pins ADC0 .. ADC7

13

ADCH/ADCL: ADC Data Registers



23.9.1 ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

23.9.3 ADCL and ADCH – The ADC Data Register

ADLAR = Analog Data Left Adjust Register

23.9.3.1 ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
(0x79)	–	–	–	–	–	–	ADC9	ADC8	ADCH
(0x78)	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

If ADLAR is set to 0,
- read ADCL for low order bits, and
- until ADCH is read the ADC is locked out

23.9.3.2 ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
(0x79)	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
(0x78)	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

For 8-bit conversion, set ADLAR to 1 and read ADCH

14

ADCSRA: ADC Status Register A

23.9.2 ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 7: ADEN – analog converter enable bit; set this bit to 1 if you want to do a conversion
- Bit 6 ADSC – AD start conversion; if it is set to 1, then a conversion is started for you and it is auto set back to 0 when done
  - You can poll this bit and as soon as it is 0, you know the conversion is done
  - Or you can poll the interrupt flag (or use the corresponding ISR if enabled):
- Bit 4: ADIF – AD interrupt flag; will be set when a conversion is done and will trigger an interrupt if ADIE is set
  - Warning: do not mess with this flag, e.g., use ADCSRA |= (1<<ADSC);

15

ADCSRA: ADC Status Register A

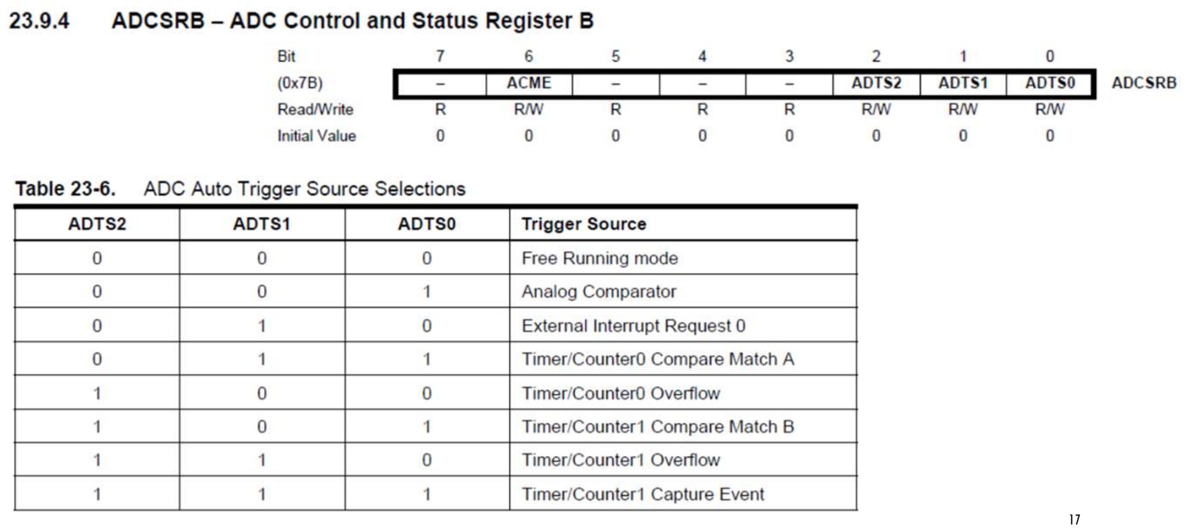
23.9.2 ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Bit 3: ADIE – AD interrupt enable; if turned on, write the ISR to handle what happens when conversion finishes
- Bit 5: ADATE – allows one out of 8 selected events to trigger the ADC converter when coupled with the ADCSRB register
- Bits 0,1,2: prescalar (see previous slide)

16

ADCSRB



23.9.4 ADCSRB – ADC Control and Status Register B

Bit	7	6	5	4	3	2	1	0	
(0x7B)	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0	ADCSRB
Read/Write	R	R/W	R	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 23-6. ADC Auto Trigger Source Selections

ADTS2	ADTS1	ADTS0	Trigger Source
0	0	0	Free Running mode
0	0	1	Analog Comparator
0	1	0	External Interrupt Request 0
0	1	1	Timer/Counter0 Compare Match A
1	0	0	Timer/Counter0 Overflow
1	0	1	Timer/Counter1 Compare Match B
1	1	0	Timer/Counter1 Overflow
1	1	1	Timer/Counter1 Capture Event

17

Example code ADC, no interrupt

```
// Borrowed from Bruce Land - Cornell University

// Performs single, left adjusted conversions and prints to UART

#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay.h>
#include <math.h>
#include "uart.h"

volatile int Ain, AinLow;
volatile float Voltage;
char VoltageBuffer[6];

FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);
```

18

Example code ADC, no interrupt

```
void main(void)
{
        DDRC &= 0x00;    // PC1 = ADC1 is set as input

        uart_init();
        stdout = stdin = stderr = &uart_str;

        // ADLAR set to 1 → left adjusted result in ADCH
        // MUX3:0 set to 0001 → input voltage at ADC1
        ADMUX = (1<<MUX0) | (1<<ADLAR);

        // ADEN set to 1 → enables the ADC circuitry
        // ADPS2:0 set to 111 → prescalar set to 128 (104us per conversion)
        ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS1) | (1<<ADPS0);

        // Start A to D conversion
        ADCSRA |= (1<<ADSC);
        fprintf(stdout,"\n\rStarting ADC demo...\n\r");
```

Takes more than 1ms, hence conversion will finish which takes 104us

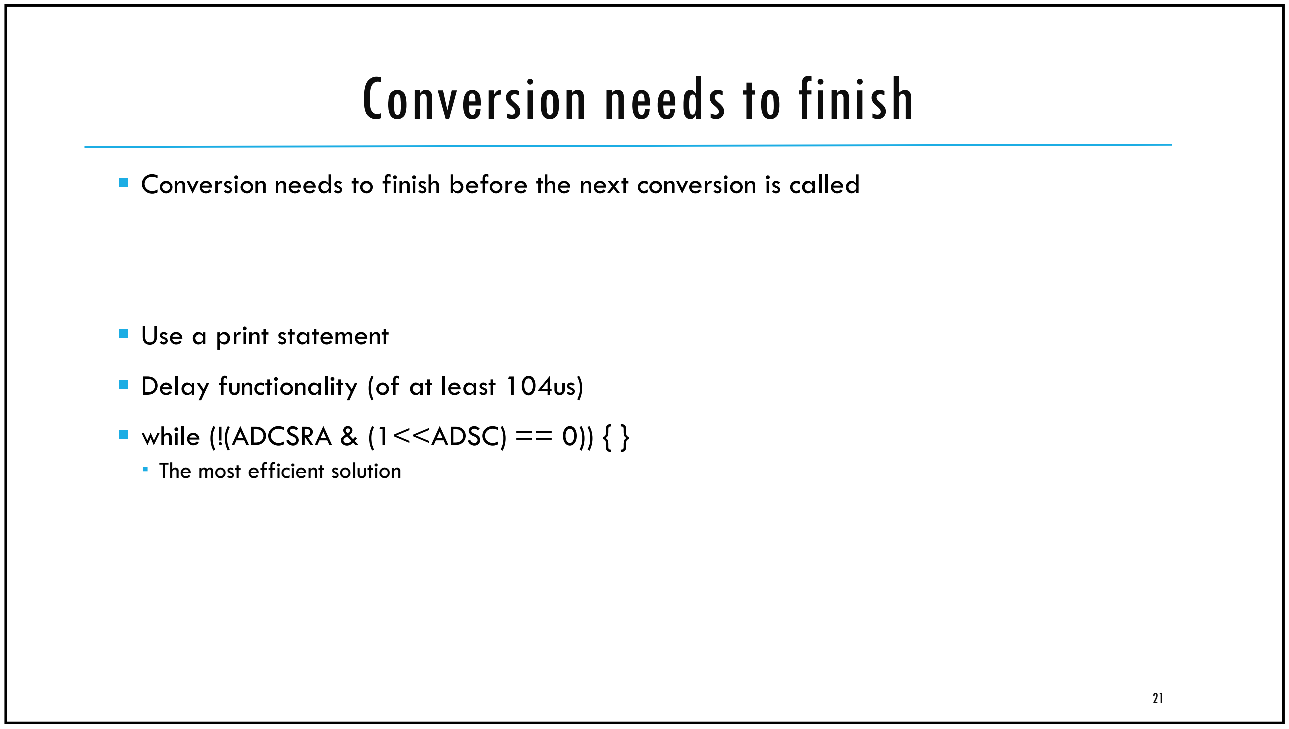
19

Example code ADC, no interrupt

```
        while (1)
        {
                // Read from ADCH to get the 8 MSBs of the 10 bit conversion
                Ain = ADCH;

                // Typecast the volatile integer into floating type data, divide by maximum 8-bit value, and
                // multiply by 5V for normalization
                Voltage = (float)Ain/256.00 * 5.00;

                //ADSC is cleared to 0 when a conversion completes.  Set ADSC to 1 to begin a conversion.
                ADCSRA |= (1<<ADSC);

                // Write Voltage to string format and print (3 char string + "." + 2 decimal places)
                dtostrf(Voltage, 3, 2, VoltageBuffer);
                fprintf(stdout,"%s\n\r",VoltageBuffer);
        }

        return 0;
}
```

Takes more than 1ms, hence conversion will finish which takes 104us

20



Conversion needs to finish

- Conversion needs to finish before the next conversion is called

- Use a print statement
- Delay functionality (of at least 104us)
- while (!(ADCSRA & (1<<ADSC) == 0)) { }
  - The most efficient solution

21

ECE3411 – Fall 2017

Lab 4a.

PWM: Pulse Width Modulation

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lab 5a, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

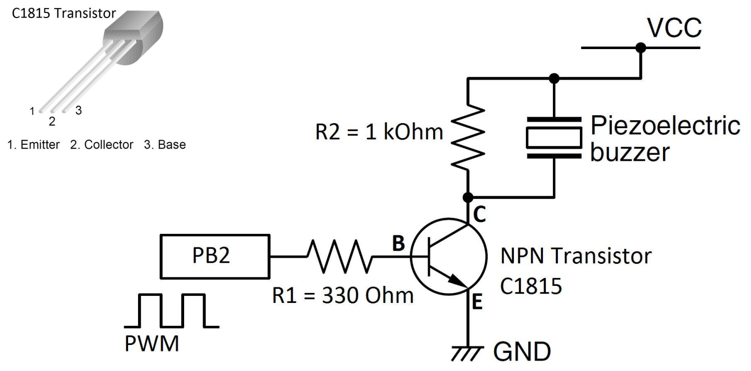
Using Timer1 for PWM

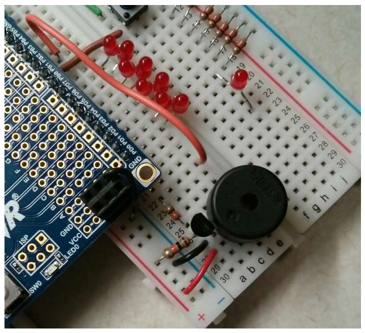
- We'll be using Timer1 in Fast PWM Mode with OCR1A as TOP
  - I.e. (WGM13, WGM12, WGM11, WGM10) = (1,1,1,1)
- PWM signal is output at pin OC1B which is PB2 on ATmega328P
- OCR1A controls the frequency of the resulting PWM signal
  - Every compare match A starts a new cycle of PWM waveform, i.e. PWM pin is 'set'.
- OCR1B controls the duty cycle of the resulting PWM signal
  - At every compare match B, the PWM signal is 'cleared' (non-inverting mode).
- Compare Match A ISR can be used to update OCR1A and OCR1B registers
  - OCR1A changes frequency and OCR1B changes duty cycle.
- If PWM signal is needed at additional pin(s):
  - Compare Match A ISR can be used to 'set' that pin(s).
  - Compare Match B ISR can be used to 'clear' that pin(s).

2

Connecting the Buzzer

- In this lab, we'll be using a buzzer that'll be driven by a PWM signal.
- Connect the buzzer according to the following schematic.





Task1: Low Frequency PWM Signal

Use Timer1 to generate a PWM signal on ***PB2 and PORTD*** such that:

- The PWM signal has a frequency of 1Hz
- While Switch 1 is pressed, the duty cycle of the PWM signal is gradually increased (say in steps of 5%) up to 100%
- While Switch 2 is pressed, the duty cycle of the PWM signal is gradually decreased (say in steps of 5%) down to 0
- Print the current duty cycle on LCD.

You don't need to debounce the switches.

Hint: Use Compare Match A and B ISRs to generate PWM on PORTD in software.

Connect a pair of a LED and a current limiting resister (330 Ohm) to PB2.

Now you should be able to observe the LEDs' blinking behavior with different duty cycles.

Task2: LED Brightness Control

Use Timer1 to generate a PWM signal on ***PB2 and PORTD*** such that:

- The PWM signal has a frequency of 1000Hz
- While Switch 1 is pressed, the duty cycle of the PWM signal is gradually increased (say in steps of 1%) up to 100%
- While Switch 2 is pressed, the duty cycle of the PWM signal is gradually decreased (say in steps of 1%) down to 0

You don't need to debounce the switches.

Hint: Use Compare Match A and B ISRs to generate PWM on PORTD in software.

Connect a pair of a LED and a current limiting resister (330 Ohm) to PB2.

Now you should be able to observe the LEDs' brightness behavior with different duty cycles.

5

Task3: The Ambulance Siren

Using Timer1 PWM generation, implement an Ambulance Siren.

In particular, implement the following:

- Using Timer1, generate a PWM signal at PB2 with 25% duty cycle.
- Using Timer0, gradually vary the frequency of the PWM signal from 1kHz to 4kHz, and then from 4kHz back to 1kHz and so on.
  - I.e. Timer 0 overflows every ~16ms → ~120 overflows in 2 seconds.
  - Each overflow changes the OCR1A value by 100 ticks.
- Notice that the duty cycle remains 25% for each frequency.
- Connect a Buzzer to the PWM signal as shown in earlier slides.

Now you should hear an ambulance siren.

Play with the frequency ranges and frequency update rates to transform the ambulance siren into a *Cop Car Siren* ☺

6

ECE3411 – Fall 2017
Lec 4b.

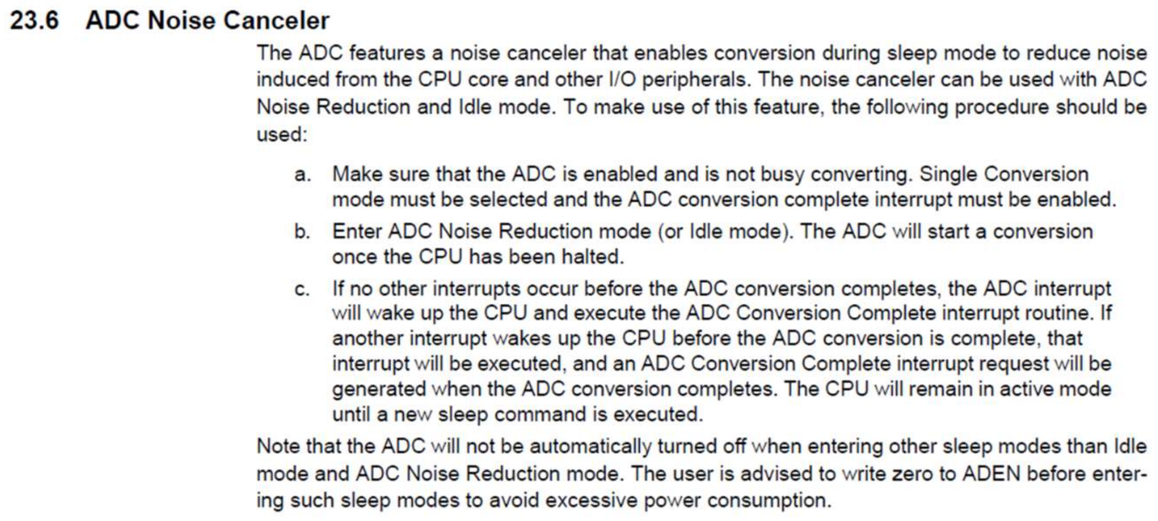
ADC: Analog to Digital Conversion

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lecture 5b, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

ADC Noise Canceler



23.6 ADC Noise Canceler

The ADC features a noise canceler that enables conversion during sleep mode to reduce noise induced from the CPU core and other I/O peripherals. The noise canceler can be used with ADC Noise Reduction and Idle mode. To make use of this feature, the following procedure should be used:

a. Make sure that the ADC is enabled and is not busy converting. Single Conversion mode must be selected and the ADC conversion complete interrupt must be enabled.
b. Enter ADC Noise Reduction mode (or Idle mode). The ADC will start a conversion once the CPU has been halted.
c. If no other interrupts occur before the ADC conversion completes, the ADC interrupt will wake up the CPU and execute the ADC Conversion Complete interrupt routine. If another interrupt wakes up the CPU before the ADC conversion is complete, that interrupt will be executed, and an ADC Conversion Complete interrupt request will be generated when the ADC conversion completes. The CPU will remain in active mode until a new sleep command is executed.

Note that the ADC will not be automatically turned off when entering other sleep modes than Idle mode and ADC Noise Reduction mode. The user is advised to write zero to ADEN before entering such sleep modes to avoid excessive power consumption.

2

ADC Noise Reduction Mode = ADC Sleep Mode

- Enable sleep mode;
- Start conversion by calling sleep_cpu();
  - MCU will be sleeping except for the conversion
- Set ADC interrupt and write ISR
  - All timers stop when you use ADC sleep; only ADC, timer 2, and interrupts stay running
- Do something wrong here and it may sleep forever
  - Always double check register settings and ISRs ..

3

Sleep Modes

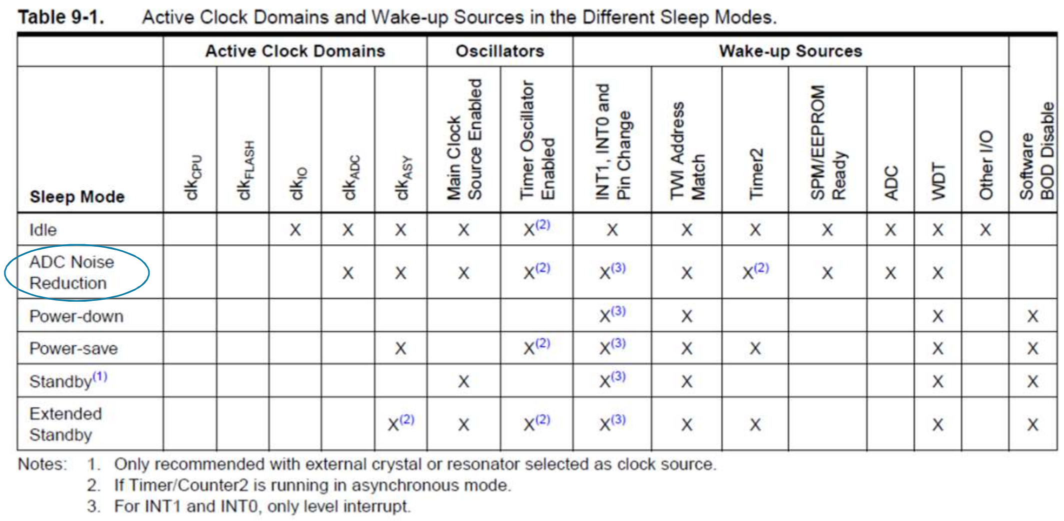
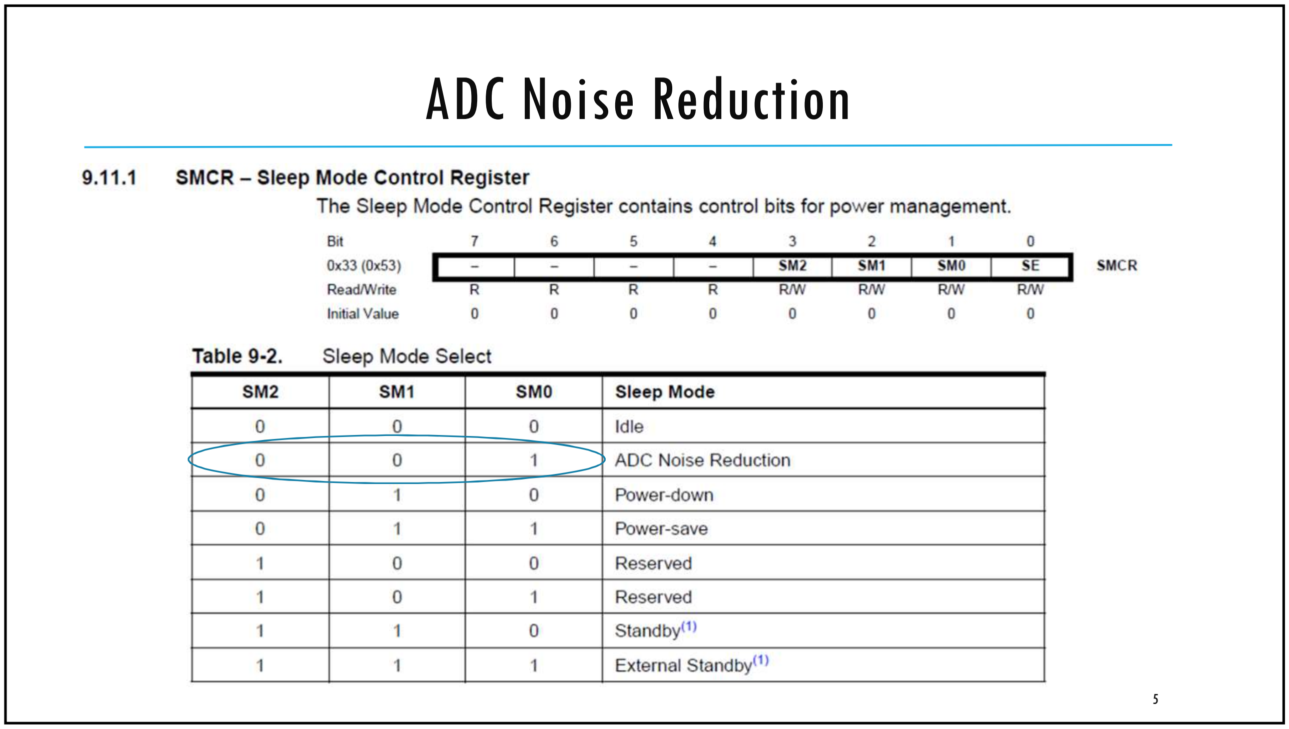


Table 9-1. Active Clock Domains and Wake-up Sources in the Different Sleep Modes.

	Active Clock Domains					Oscillators		Wake-up Sources								
Sleep Mode	clk_{CPU}	clk_{FLASH}	clk_{IO}	clk_{ADC}	clk_{ASY}	Main Clock Source Enabled	Timer Oscillator Enabled	INT1, INT0 and Pin Change	TWI Address Match	Timer2	SPM/EEPROM Ready	ADC	WDT	Other I/O	Software BOD Disable
Idle			X	X	X	X	X$^{(2)}$	X	X	X	X	X	X	X	
ADC Noise Reduction				X	X	X	X$^{(2)}$	X$^{(3)}$	X	X$^{(2)}$	X	X	X		
Power-down								X$^{(3)}$	X				X		X
Power-save					X		X$^{(2)}$	X$^{(3)}$	X	X			X		X
Standby$^{(1)}$						X		X$^{(3)}$	X				X		X
Extended Standby					X$^{(2)}$	X	X$^{(2)}$	X$^{(3)}$	X	X			X		X

Notes: 1. Only recommended with external crystal or resonator selected as clock source.
2. If Timer/Counter2 is running in asynchronous mode.
3. For INT1 and INT0, only level interrupt.

4



ADC Noise Reduction

9.11.1 SMCR – Sleep Mode Control Register

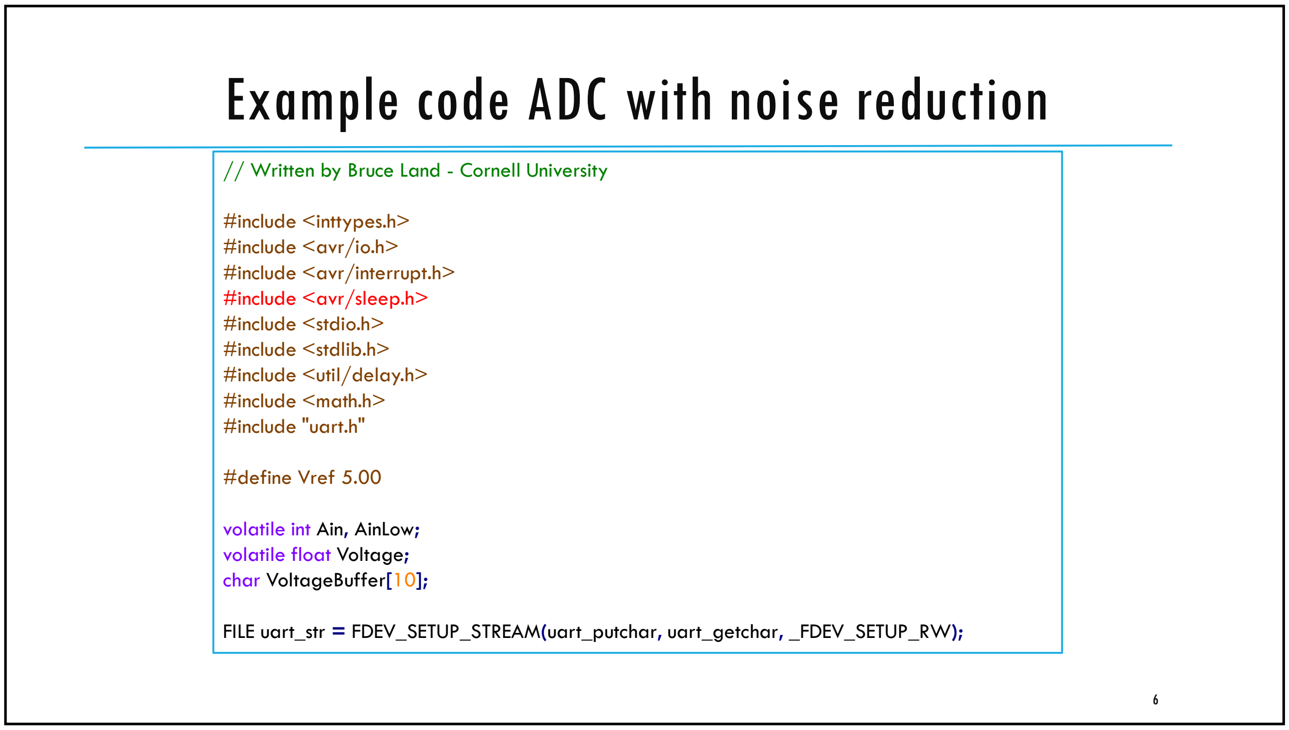
The Sleep Mode Control Register contains control bits for power management.

Bit	7	6	5	4	3	2	1	0	
0x33 (0x53)	–	–	–	–	SM2	SM1	SM0	SE	SMCR
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Table 9-2. Sleep Mode Select

SM2	SM1	SM0	Sleep Mode
0	0	0	Idle
0	0	1	ADC Noise Reduction
0	1	0	Power-down
0	1	1	Power-save
1	0	0	Reserved
1	0	1	Reserved
1	1	0	Standby(1)
1	1	1	External Standby(1)

5



Example code ADC with noise reduction

```
// Written by Bruce Land - Cornell University

#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/sleep.h>
#include <stdio.h>
#include <stdlib.h>
#include <util/delay.h>
#include <math.h>
#include "uart.h"

#define Vref 5.00

volatile int Ain, AinLow;
volatile float Voltage;
char VoltageBuffer[10];

FILE uart_str = FDEV_SETUP_STREAM(uart_putchar, uart_getchar, _FDEV_SETUP_RW);
```

6

Example code ADC with noise reduction

```
ISR (ADC_vect)
{
            // Program ONLY gets here when ADC done flag is set
            // When reading 10-bit values you MUST read the low byte first
            AinLow = (int)ADCL;
            Ain = (int)ADCH*256;
            Ain = Ain + AinLow;
}
```

7

Example code ADC with noise reduction

```
int main(void)
{
            //init the A to D converter
            ADMUX = 0b00000001;
            ADCSRA = (1<<ADEN) | (1<<ADIE) + 7 ;
            SMCR = (1<<SM0) ; // sleep -- choose ADC mode

            // init the UART -- uart_init() is in uart.c
            uart_init();
            stdout = stdin = stderr = &uart_str;
            fprintf(stdout,"\n\rStarting ADC ISR demo...\n\r");

            // Need the next two statements so that the USART finishes
            // BEFORE the cpu goes to sleep.
            while (!(UCSR0A & (1<<UDRE0))) ; // Is UART still doing stuff?
            _delay_ms(1); // enough time to empty the transmit buffer

            sleep_enable();
            sei();
```

8

Example code ADC with noise reduction

```
        while (1)
        {
                // Get the sample
                //The sleep statement lowers digital noise and starts the A/D conversion
                sleep_cpu();

                //program ONLY gets here after ADC ISR is done
                voltage = (float)Ain ;
                voltage = (voltage/1024.0)*Vref ; //(fraction of full scale)*Vref
                dtostrf(voltage, 6, 3, v_string);
                printf("%s", v_string);

                // Need the next two statements so that the USART finishes
                // BEFORE the cpu goes to sleep the next time thru the loop.
                while (!(UCSR0A & (1<<UDRE0))) ; // Is UART still doing stuff?
                _delay_ms(1); // enough time to empty the transmit buffer
        }
        return 0;
}
```

9

Exercises

- Can you get rid of the _delay_ms(1) instruction in the while loop by using a task based programming approach?
- This would be useful if other tasks would need to execute as well.
- Note that each char takes about 1ms to print:
  - Is a 1ms delay in the main while loop enough? Why?
  - How many ms does while (!(UCSR0A & (1<<UDRE0))); approximately wait in the main while loop?
  - In a task based approach would it be better to avoid while (!(UCSR0A & (1<<UDRE0))); ?
  - And how would you do this?
- Check the code in the slides (I changed an earlier version without double checking: you may figure out a bug here and there ☺)

10

Example Problem: What is happening in the following code?

```
... inclusion of packages ...
... declaration of global variables ...
... we assume a 20MHz crystal ...

ISR (TIMER0_COMPA_vect)
{
            //Update task timer
            if (taskADC_timer >0 ) {--taskADC_timer;}
}

ISR (ADC_vect)
{
            //Read a 10-bit conversion
            AinLow = (int)ADCL;
            Ain = (int)ADCH*256;
            Ain = Ain + AinLow;
}
```

11

What is happening in the following code?

```
void taskADC(void)
{
            //Reset task timer
            taskADC_timer = 400;

            //Convert Ain into a voltage
            voltage = ((1.0*Ain)/1024.0)*5.0;

            ... Some more computation: sometimes taking more and sometimes taking less time ...
            ... However, no matter how long taskADC() takes, its execution is always <= 200 ms ...
}
```

12

What is happening in the following code?

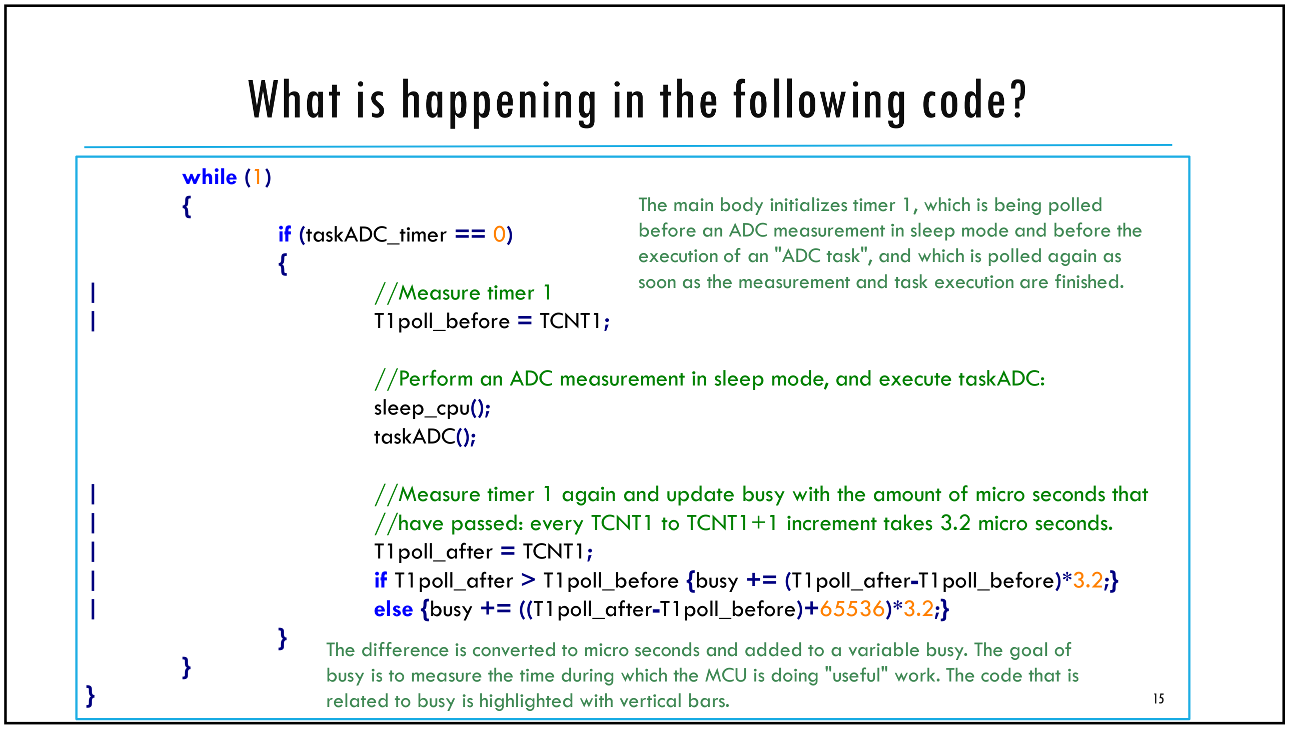
```
int main(void)
{
        ... initialization variables ...

|       //set up timer 1 for 3.2 micro second counter increments
|       TCCR1B = 3; //set prescalar to divide by 64

        //set up timer 0 such that ISR(TIMER0_COMPA_vect) is called every 1 milli second
        OCR0A = 77;             //Set the compare reg to 78 time ticks
        TIMSK0 = (1<<OCIE0A);   //Turn on timer 0 cmp match ISR
        TCCR0B = 4;             //Set prescalar to divide by 256
        TCCR0A = (1<<WGM01);    //Turn on clear-on-match
        //how accurate is this timer?
```

13

What is happening in the following code?

```
//initialize the A to D converter
DDRC &= 0xF0;                                //Set PORTC[3:0] as input for ADC
ADMUX = 0b00000001;                          //Indicate which pin should be measured
ADCSRA = (1<<ADEN) | (1<<ADIE) + 7 ;         //Enable AD converter, enable its interrupt,
                                             //set prescalar (notice that the ADSC bit is
                                             //not set, so no ADC conversion is started)
SMCR = (1<<SM0) ;                            //Choose ADC sleep mode

sleep_enable();
sei();
```

14



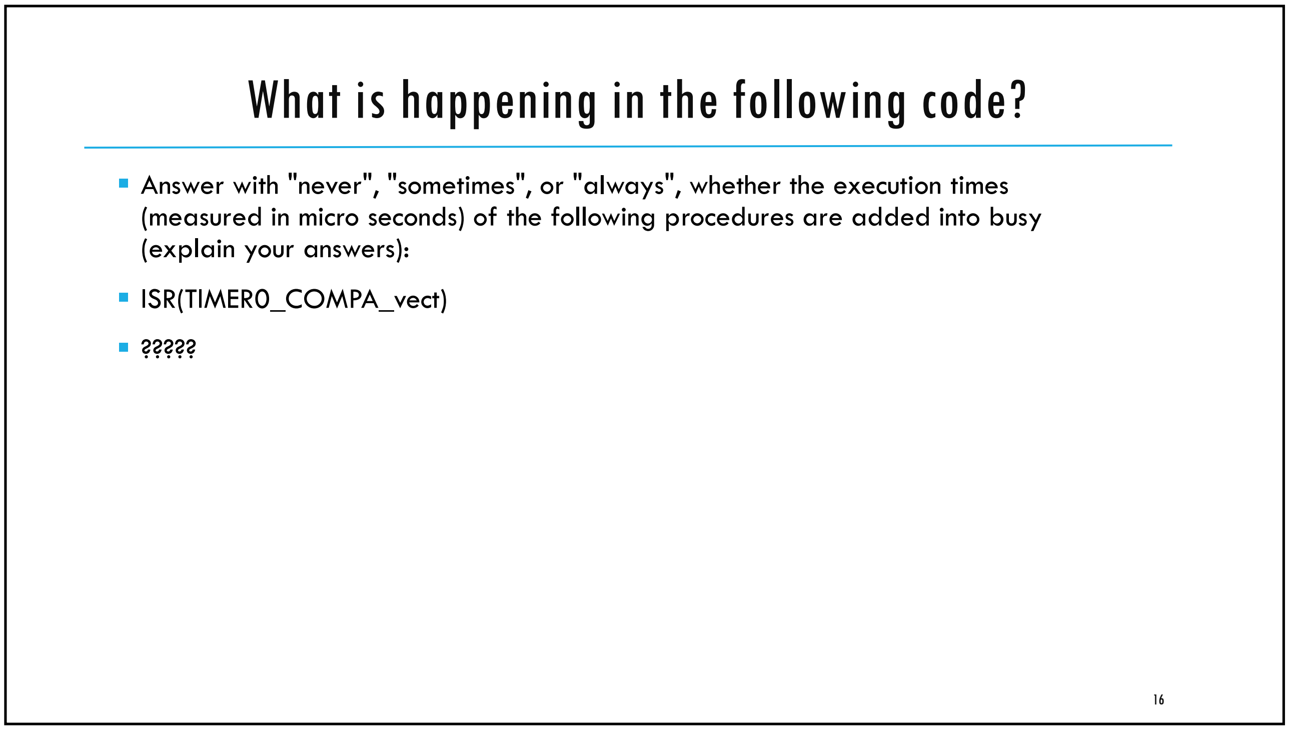
What is happening in the following code?

The main body initializes timer 1, which is being polled before an ADC measurement in sleep mode and before the execution of an "ADC task", and which is polled again as soon as the measurement and task execution are finished.

```
        while (1)
        {
                if (taskADC_timer == 0)
                {
|                       //Measure timer 1
|                       T1poll_before = TCNT1;

                        //Perform an ADC measurement in sleep mode, and execute taskADC:
                        sleep_cpu();
                        taskADC();

|                       //Measure timer 1 again and update busy with the amount of micro seconds that
|                       //have passed: every TCNT1 to TCNT1+1 increment takes 3.2 micro seconds.
|                       T1poll_after = TCNT1;
|                       if T1poll_after > T1poll_before {busy += (T1poll_after-T1poll_before)*3.2;}
|                       else {busy += ((T1poll_after-T1poll_before)+65536)*3.2;}
                }
        }
}
```

The difference is converted to micro seconds and added to a variable busy. The goal of busy is to measure the time during which the MCU is doing "useful" work. The code that is related to busy is highlighted with vertical bars.

15



What is happening in the following code?

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- ISR(TIMER0_COMPA_vect)
- ?????

16

What is happening in the following code?

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- ISR(ADC_vect)
- ?????

17

What is happening in the following code?

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- sleep_cpu()
- ?????

18

What is happening in the following code?

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- taskADC()
- ?????

What is happening in the following code?

- The program assumes that taskADC() always takes <=200 ms. Use this assumption to explain why the code

```
if T1poll_after > T1poll_before {busy += (T1poll_after-T1poll_before)*3.2;}
else {busy += ((T1poll_after-T1poll_before)+65536)*3.2;}
```

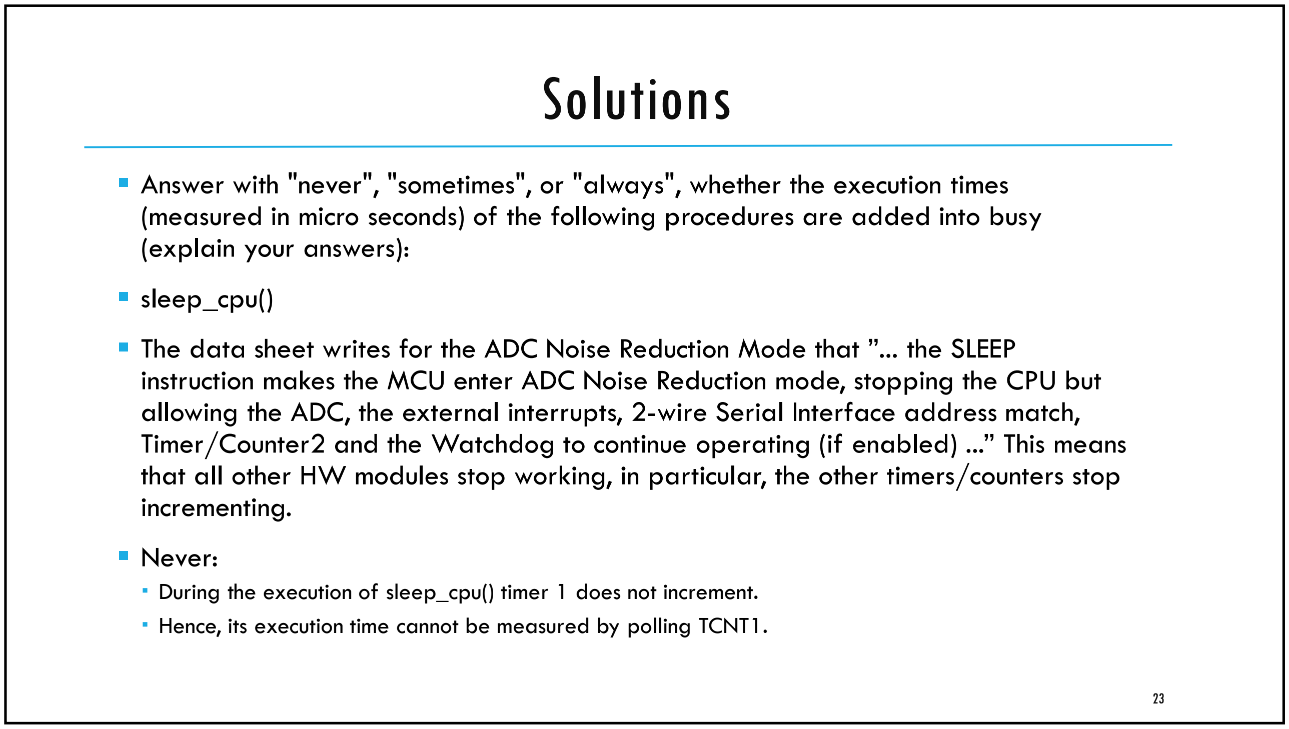
- correctly adds to busy the time in micro seconds that passed between the polling of T1poll_before and the polling of T1poll_after.
- Solution: ??????

Solutions

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- ISR(TIMER0_COMPA_vect)
- Sometimes:
  - The ADC task is executed approximately every 400 ms and executes in less than 200 ms.
  - So, there is always a significant number of ms during which the while loop does not execute the code within the if statement.
  - During this "idle" time the timer ISR is called every ms but its execution time is not added into busy.
  - During the time that the ADC task is executed the timer ISR will also be called and executed. These execution times do get added into busy.

Solutions

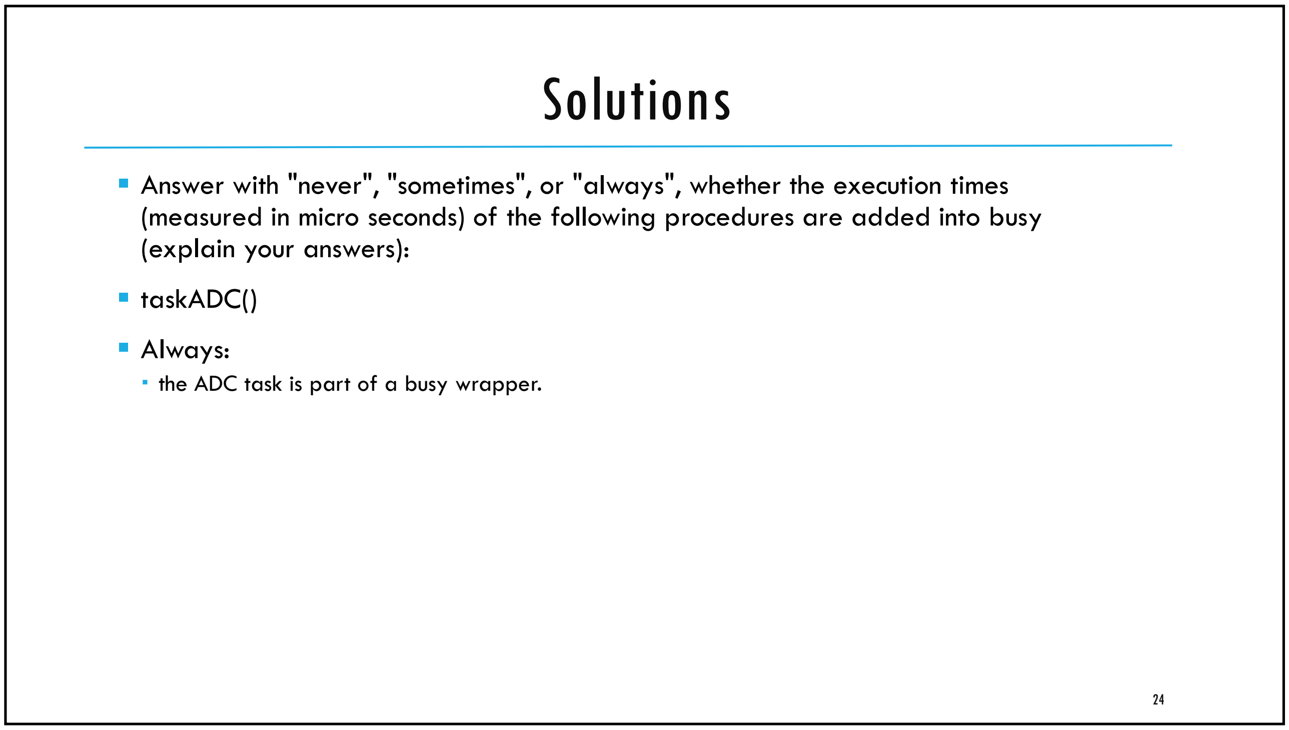
- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- ISR(ADC_vect)
- Always:
  - Right after sleep_cpu(), the ADC ISR is called.
  - Since sleep_cpu() is part of a busy wrapper, the execution time of each ADC ISR is part of busy's measurement.



Solutions

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- sleep_cpu()
- The data sheet writes for the ADC Noise Reduction Mode that ”... the SLEEP instruction makes the MCU enter ADC Noise Reduction mode, stopping the CPU but allowing the ADC, the external interrupts, 2-wire Serial Interface address match, Timer/Counter2 and the Watchdog to continue operating (if enabled) ...” This means that all other HW modules stop working, in particular, the other timers/counters stop incrementing.
- Never:
  - During the execution of sleep_cpu() timer 1 does not increment.
  - Hence, its execution time cannot be measured by polling TCNT1.

23



Solutions

- Answer with "never", "sometimes", or "always", whether the execution times (measured in micro seconds) of the following procedures are added into busy (explain your answers):
- taskADC()
- Always:
  - the ADC task is part of a busy wrapper.

24

Solutions

- The program assumes that taskADC() always takes <=200 ms. Use this assumption to explain why the code

```
if T1poll_after > T1poll_before {busy += (T1poll_after-T1poll_before)*3.2;}
else {busy += ((T1poll_after-T1poll_before)+65536)*3.2;}
```

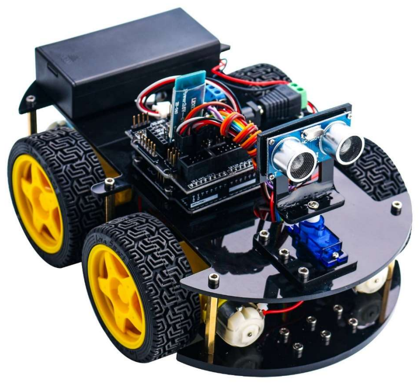
correctly adds to busy the time in micro seconds that passed between the polling of T1poll_before and the polling of T1poll_after.

- Solution:
  - Each task takes less than 200 ms, which is less than 2^{16} * 3.2 micro seconds (=209.7 ms),
  - which is the time it takes to increment TCNT1 from 0 to its maximum value.
  - So, TCNT1 may at most loop through *once*.
    - If TCNT1 does not loop through, then T1poll_after > T1poll_before and (T1poll_after-T1poll_before)*3.2} measures the amount of time that has lapsed in micro seconds.
    - If TCNT1 loops though once, then T1poll_after <= T1poll_before and ((T1poll_after - 0) + (2^{16} - T1poll_before))*3.2 measures the amount of time that has lapsed.

25

Spring 2017: Advanced MCU Applications Lab

- What?
  - Advanced course on Microcontrollers' Applications.
- Instructor
  - Marten van Dijk
- When?
  - Next semester: Spring 2017
- Who can join?
  - Everyone who has taken ECE3411
- What will be taught?
  - Parallel Bus interfaces (for external SRAM/other devices)
  - Controller Area Network (CAN Protocol)
  - Wireless Protocols (E.g. Bluetooth)
  - Analog Sensors Interfacing (E.g. Ultrasonic Sensors)
  - Motor Control (DC motors, Servo motors)
  - Real-time Operating Systems
  - And more…
  - Final Project: Collision Avoidance Robot



26

ECE3411 – Fall 2017
Lab 4b.

ADC: Analog to Digital Conversion

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

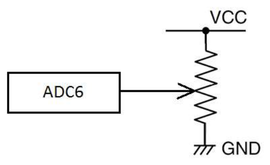
Copied from Lab 5b, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

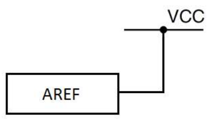
UCONN

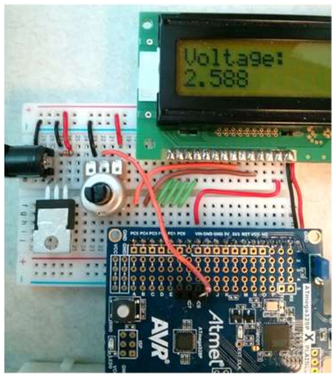
Hardware Changes for Task1

In this lab, we'll be measuring analog voltages using ADC

- Connect the potentiometer to ADC6 pin as shown below.
- Connect AREF pin to VCC → Reference voltage becomes 5V.







2

Task1: Simple Voltmeter

We are going to design a simple voltmeter that measures voltages between 0-5V with ~4mV resolution.

- Connect a potentiometer to produce variable voltage at ADC6 pin.
- Connect AREF pin to VCC
- Read the analog input voltage using ADC every 100ms
- Convert the ADC reading to voltage measurement and print the voltage on LCD.

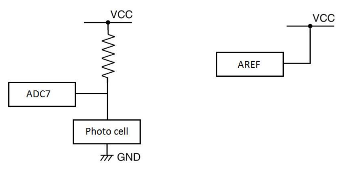
Note: Use the full 10-bit resolution of the ACD

Now you should be able to observe different voltage readings as you twist the potentiometer knob.

Task2: Light Sensor

We are going to control the brightness of LEDs based on the ambient light in the room.

- Replace the potentiometer with a resistor divider network with a photo cell (light dependent resistor) and a 10kOhm resistor to produce variable voltage at ADC7 pin.
- Connect AREF pin to VCC
- Read the analog input voltage using ADC every 100ms
- Change the duty cycle of the PWM signal for LEDs to control their brightness.
- When the Photo cell detects dark, increase the brightness and vice versa.



Task3: Voltage Measurement Statistics

Write a program with the following specifications:

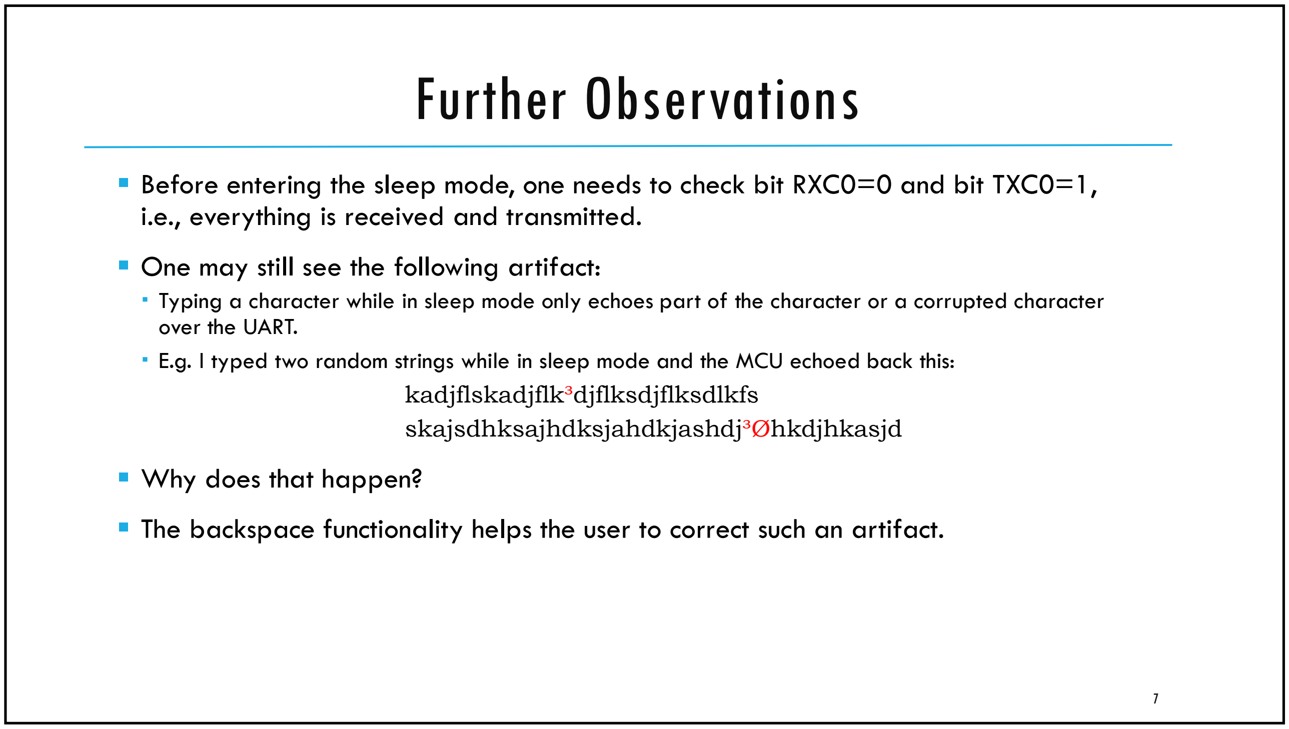
- The program allows two modes for measuring voltages: Normal mode and ADC sleep mode. The user can change mode through UART while the program is executing.
- The program measures with 10 bits accuracy the input voltage (against AREF with VREF=5V) every 1ms.
- The program keeps track of the average and standard deviation of your measurements over the last 50 ms.
- Every 150 ms display on the LCD screen on two separate lines the most recent measured average voltage with its standard deviation (also use words/symbols to make clear what is being displayed).

Open Questions in Task3

Task3 leaves some open questions:

- How often do you measure? And when do you measure? E.g., you may want to wait say 1 ms after each print statement to the terminal and LCD screen such that the print buffers are emptied before the next ADC conversion.
- How do you average if some of your ADC conversions are closer together in time than others?

Pick your solution and explain its accuracy and what you could do in future versions to improve the result.



Further Observations

- Before entering the sleep mode, one needs to check bit RXC0=0 and bit TXC0=1, i.e., everything is received and transmitted.
- One may still see the following artifact:
  - Typing a character while in sleep mode only echoes part of the character or a corrupted character over the UART.
  - E.g. I typed two random strings while in sleep mode and the MCU echoed back this:

```
kadjflskadjflk³djflksdjflksdlkfs
skajsdhksajhdksjahdkjashdj³Øhkdjhkasjd
```

- Why does that happen?
- The backspace functionality helps the user to correct such an artifact.

7

ECE3411 – Fall 2017
Lec 4c.

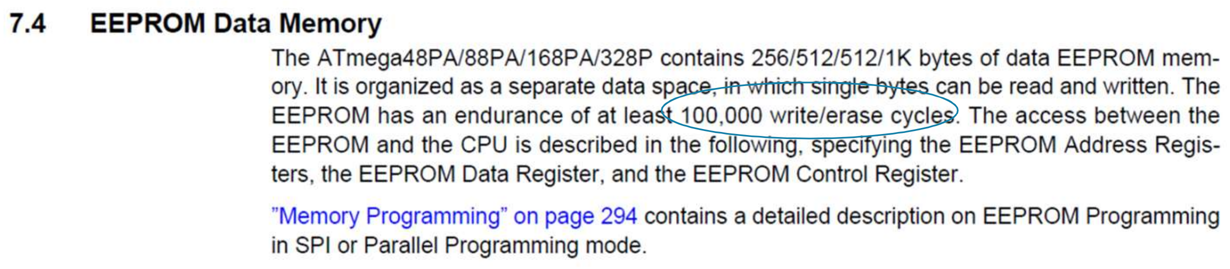
EEPROM
Watchdog Timer

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lecture 5c, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

EEPROM: Electrically Erasable Programmable ROM



7.4 EEPROM Data Memory

The ATmega48PA/88PA/168PA/328P contains 256/512/512/1K bytes of data EEPROM memory. It is organized as a separate data space, in which single bytes can be read and written. The EEPROM has an endurance of at least 100,000 write/erase cycles. The access between the EEPROM and the CPU is described in the following, specifying the EEPROM Address Registers, the EEPROM Data Register, and the EEPROM Control Register.

"Memory Programming" on page 294 contains a detailed description on EEPROM Programming in SPI or Parallel Programming mode.

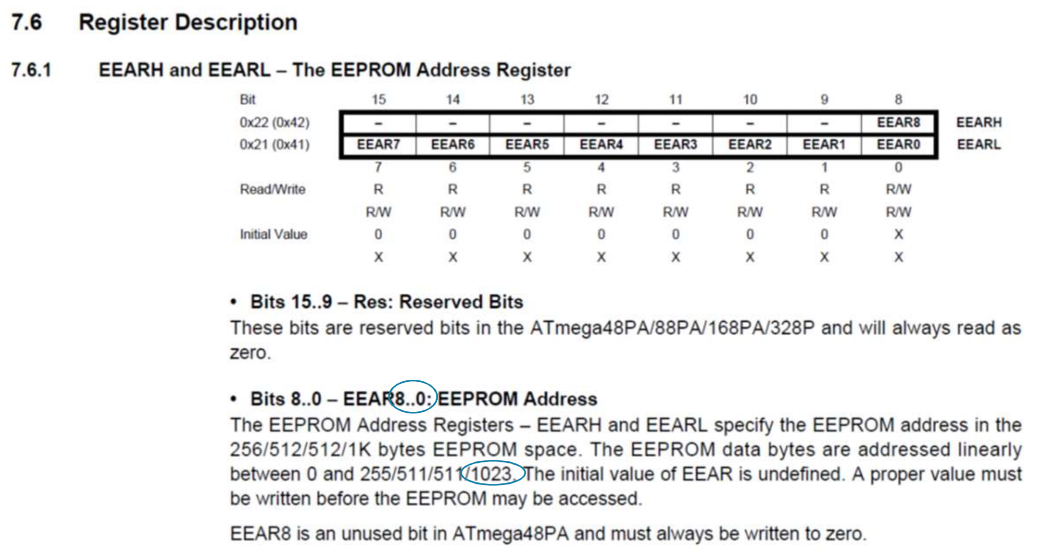
You should not access EEPROM in main in a loop, otherwise, it will not exist any more !

EEPROM reads in 2 cycles and writes in about 100 cycles
Flash (program + optional read only data): read in 2 cycles
RAM: read+write in 1 cycle each

Data in EEPROM remains even if you pull the chip out of the board or turn power on and off

2

EEPROM



7.6 Register Description

7.6.1 EEARH and EEARL – The EEPROM Address Register

Bit	15	14	13	12	11	10	9	8	
0x22 (0x42)	–	–	–	–	–	–	–	EEAR8	EEARH
0x21 (0x41)	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	X	
	X	X	X	X	X	X	X	X	

- **Bits 15..9 – Res: Reserved Bits**

These bits are reserved bits in the ATmega48PA/88PA/168PA/328P and will always read as zero.

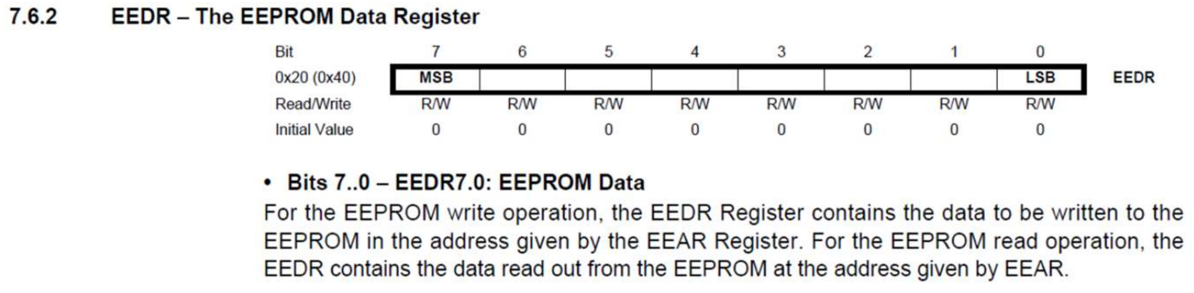
- **Bits 8..0 – EEAR8..0: EEPROM Address**

The EEPROM Address Registers – EEARH and EEARL specify the EEPROM address in the 256/512/512/1K bytes EEPROM space. The EEPROM data bytes are addressed linearly between 0 and 255/511/511/1023. The initial value of EEAR is undefined. A proper value must be written before the EEPROM may be accessed.

EEAR8 is an unused bit in ATmega48PA and must always be written to zero.

3

EEPROM



7.6.2 EEDR – The EEPROM Data Register

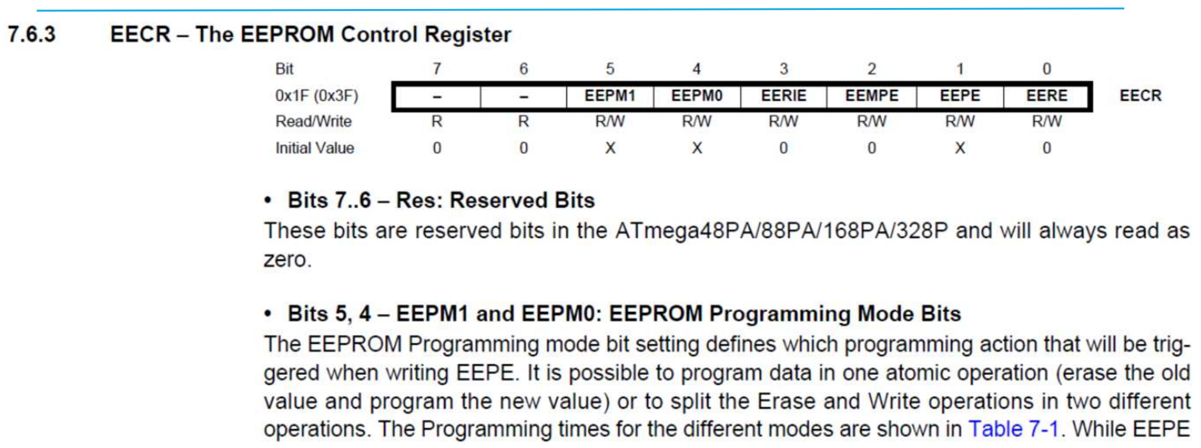
Bit	7	6	5	4	3	2	1	0	
0x20 (0x40)	MSB							LSB	EEDR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bits 7..0 – EEDR7.0: EEPROM Data**

For the EEPROM write operation, the EEDR Register contains the data to be written to the EEPROM in the address given by the EEAR Register. For the EEPROM read operation, the EEDR contains the data read out from the EEPROM at the address given by EEAR.

4

EEPROM



7.6.3 EECR – The EEPROM Control Register

Bit	7	6	5	4	3	2	1	0	
0x1F (0x3F)	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	X	X	0	0	X	0	

- **Bits 7..6 – Res: Reserved Bits**

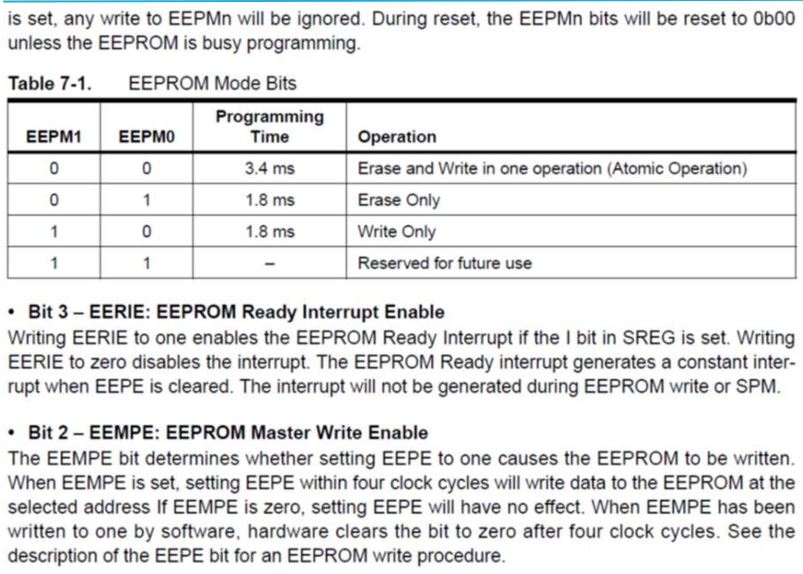
These bits are reserved bits in the ATmega48PA/88PA/168PA/328P and will always read as zero.

- **Bits 5, 4 – EEPM1 and EEPM0: EEPROM Programming Mode Bits**

The EEPROM Programming mode bit setting defines which programming action that will be triggered when writing EEPE. It is possible to program data in one atomic operation (erase the old value and program the new value) or to split the Erase and Write operations in two different operations. The Programming times for the different modes are shown in Table 7-1. While EEPE

5

EEPROM



is set, any write to EEPMn will be ignored. During reset, the EEPMn bits will be reset to 0b00 unless the EEPROM is busy programming.

Table 7-1. EEPROM Mode Bits

EEPM1	EEPM0	Programming Time	Operation
0	0	3.4 ms	Erase and Write in one operation (Atomic Operation)
0	1	1.8 ms	Erase Only
1	0	1.8 ms	Write Only
1	1	–	Reserved for future use

- **Bit 3 – EERIE: EEPROM Ready Interrupt Enable**

Writing EERIE to one enables the EEPROM Ready Interrupt if the I bit in SREG is set. Writing EERIE to zero disables the interrupt. The EEPROM Ready interrupt generates a constant interrupt when EEPE is cleared. The interrupt will not be generated during EEPROM write or SPM.

- **Bit 2 – EEMPE: EEPROM Master Write Enable**

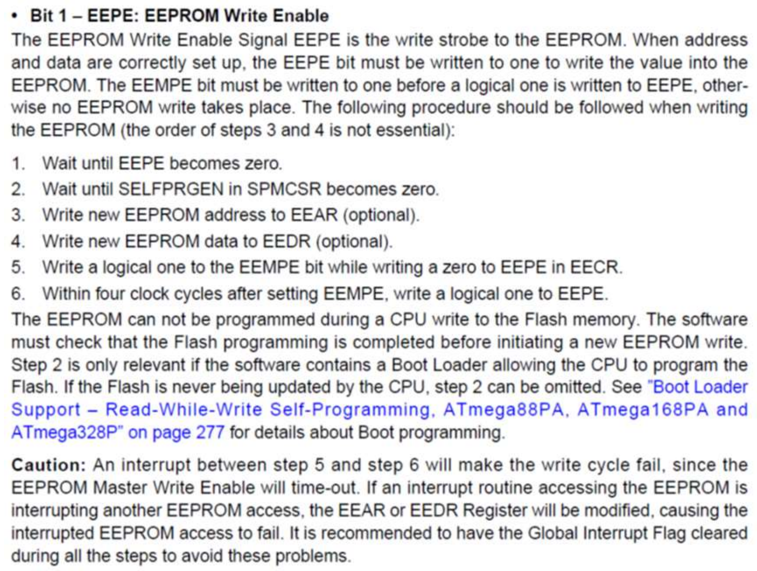
The EEMPE bit determines whether setting EEPE to one causes the EEPROM to be written. When EEMPE is set, setting EEPE within four clock cycles will write data to the EEPROM at the selected address If EEMPE is zero, setting EEPE will have no effect. When EEMPE has been written to one by software, hardware clears the bit to zero after four clock cycles. See the description of the EEPE bit for an EEPROM write procedure.

One can do sequential writes before an erasure:

- During a write one can only flip bits from 0 to 1
- If a bit needs to flip from a 1 to 0, an erasure is required before doing a write
- Hence, sequential writes may be possible if only 0 to 1 bit flips need to be written
- The lifetime of an eeprom bit is about 100.000 0 →1 write and 1→0 erasure cycles
- So, if sequential writes before an erasure are possible, the lifetime of eeprom is not unnecessarily shortened

6

EEPROM



• **Bit 1 – EEPE: EEPROM Write Enable**

The EEPROM Write Enable Signal EEPE is the write strobe to the EEPROM. When address and data are correctly set up, the EEPE bit must be written to one to write the value into the EEPROM. The EEMPE bit must be written to one before a logical one is written to EEPE, otherwise no EEPROM write takes place. The following procedure should be followed when writing the EEPROM (the order of steps 3 and 4 is not essential):

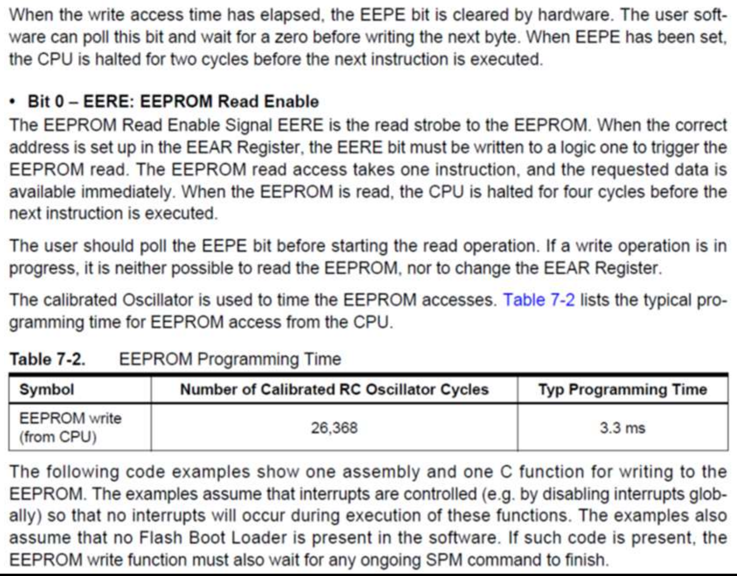
1. Wait until EEPE becomes zero.
2. Wait until SELFPRGEN in SPMCSR becomes zero.
3. Write new EEPROM address to EEAR (optional).
4. Write new EEPROM data to EEDR (optional).
5. Write a logical one to the EEMPE bit while writing a zero to EEPE in EECR.
6. Within four clock cycles after setting EEMPE, write a logical one to EEPE.

The EEPROM can not be programmed during a CPU write to the Flash memory. The software must check that the Flash programming is completed before initiating a new EEPROM write. Step 2 is only relevant if the software contains a Boot Loader allowing the CPU to program the Flash. If the Flash is never being updated by the CPU, step 2 can be omitted. See "Boot Loader Support – Read-While-Write Self-Programming, ATmega88PA, ATmega168PA and ATmega328P" on page 277 for details about Boot programming.

Caution: An interrupt between step 5 and step 6 will make the write cycle fail, since the EEPROM Master Write Enable will time-out. If an interrupt routine accessing the EEPROM is interrupting another EEPROM access, the EEAR or EEDR Register will be modified, causing the interrupted EEPROM access to fail. It is recommended to have the Global Interrupt Flag cleared during all the steps to avoid these problems.

7

EEPROM



When the write access time has elapsed, the EEPE bit is cleared by hardware. The user software can poll this bit and wait for a zero before writing the next byte. When EEPE has been set, the CPU is halted for two cycles before the next instruction is executed.

• **Bit 0 – EERE: EEPROM Read Enable**

The EEPROM Read Enable Signal EERE is the read strobe to the EEPROM. When the correct address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read. The EEPROM read access takes one instruction, and the requested data is available immediately. When the EEPROM is read, the CPU is halted for four cycles before the next instruction is executed.

The user should poll the EEPE bit before starting the read operation. If a write operation is in progress, it is neither possible to read the EEPROM, nor to change the EEAR Register.

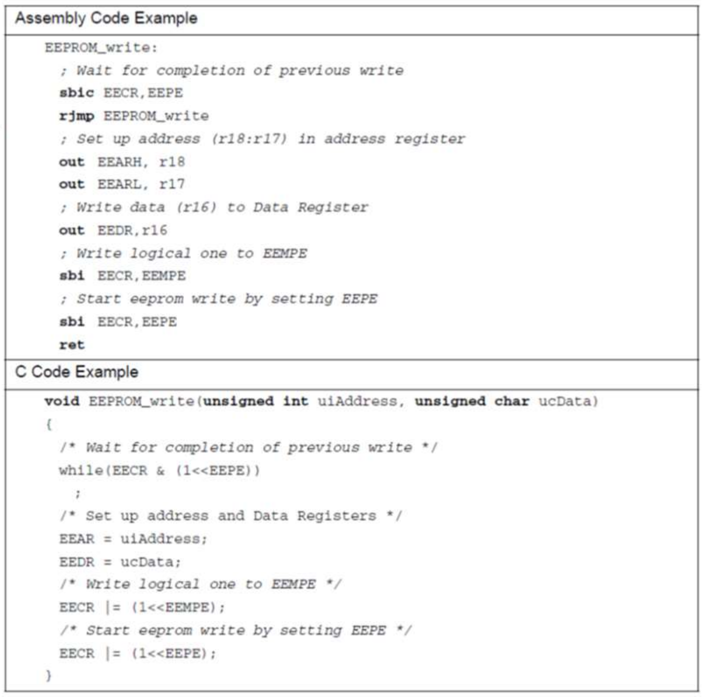
The calibrated Oscillator is used to time the EEPROM accesses. Table 7-2 lists the typical programming time for EEPROM access from the CPU.

Table 7-2. EEPROM Programming Time

Symbol	Number of Calibrated RC Oscillator Cycles	Typ Programming Time
EEPROM write (from CPU)	26,368	3.3 ms

The following code examples show one assembly and one C function for writing to the EEPROM. The examples assume that interrupts are controlled (e.g. by disabling interrupts globally) so that no interrupts will occur during execution of these functions. The examples also assume that no Flash Boot Loader is present in the software. If such code is present, the EEPROM write function must also wait for any ongoing SPM command to finish.

8



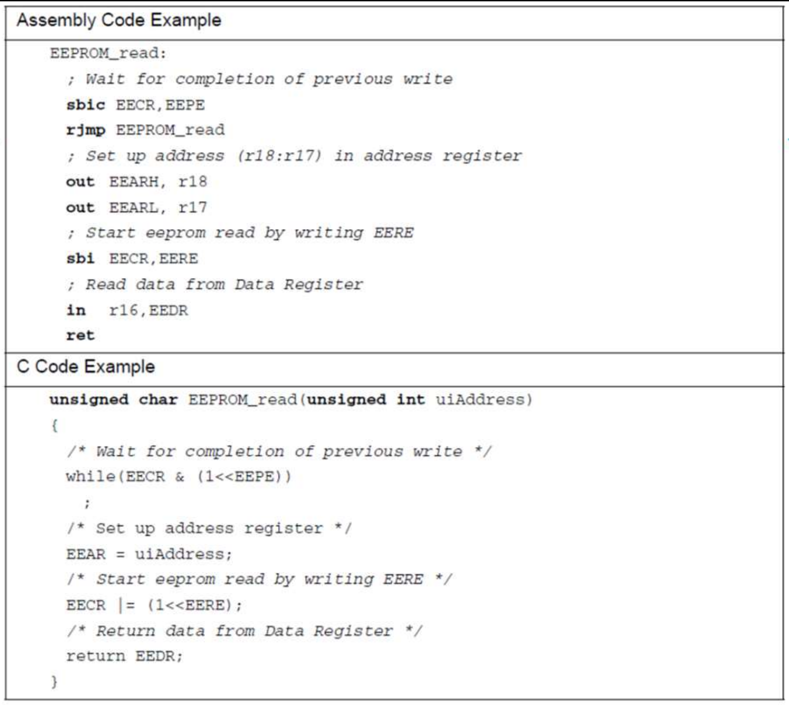
Assembly Code Example

```
EEPROM_write:
  ; Wait for completion of previous write
  sbic EECR,EEPE
  rjmp EEPROM_write
  ; Set up address (r18:r17) in address register
  out  EEARH, r18
  out  EEARL, r17
  ; Write data (r16) to Data Register
  out  EEDR,r16
  ; Write logical one to EEMPE
  sbi  EECR,EEMPE
  ; Start eeprom write by setting EEPE
  sbi  EECR,EEPE
  ret
```

C Code Example

```
void EEPROM_write(unsigned int uiAddress, unsigned char ucData)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEPE))
    ;
  /* Set up address and Data Registers */
  EEAR = uiAddress;
  EEDR = ucData;
  /* Write logical one to EEMPE */
  EECR |= (1<<EEMPE);
  /* Start eeprom write by setting EEPE */
  EECR |= (1<<EEPE);
}
```

9



Assembly Code Example

```
EEPROM_read:
  ; Wait for completion of previous write
  sbic EECR,EEPE
  rjmp EEPROM_read
  ; Set up address (r18:r17) in address register
  out  EEARH, r18
  out  EEARL, r17
  ; Start eeprom read by writing EERE
  sbi  EECR,EERE
  ; Read data from Data Register
  in   r16,EEDR
  ret
```

C Code Example

```
unsigned char EEPROM_read(unsigned int uiAddress)
{
  /* Wait for completion of previous write */
  while(EECR & (1<<EEPE))
    ;
  /* Set up address register */
  EEAR = uiAddress;
  /* Start eeprom read by writing EERE */
  EECR |= (1<<EERE);
  /* Return data from Data Register */
  return EEDR;
}
```

10

EEPROM

```
#include <avr/eeprom.h>
#define eeprom_true 0  //Suppose you want to store a flag at position 0
#define eeprom_data 1 //Suppose you want to store data at position 1

// Code snippet in e.g. an initialization
if (eeprom_read_byte((uint8_t*)eeprom_true) == 'T')
 {
            time = eeprom_read_byte((uint8_t*)eeprom_data);
 }
else
{
            time = 0; //Initialize time to 0 as this is the first time the code is running on the MCU
                      //before it has ever been reset
}
```

Use 'T' or something else from default 0 byte data values

(unint8_t*) is used to cast eeprom_data and eeprom_true into a byte pointer

11

EEPROM

```
// Code snippet in some task:
if (SW1_Pressed)
{
            if (eeprom_read_byte((uint8_t*)eeprom_true) != 'T')
             {
                        eeprom_write_byte((uint8_t*)eeprom_true,'T');
             }
            eeprom_write_byte((uint8_t*)eeprom_data,time);  //Write time to EEPROM
            fprintf(stdout,"button push at %d \n\r", time);        //Write time to UART
}
```

12

Watchdog Timer

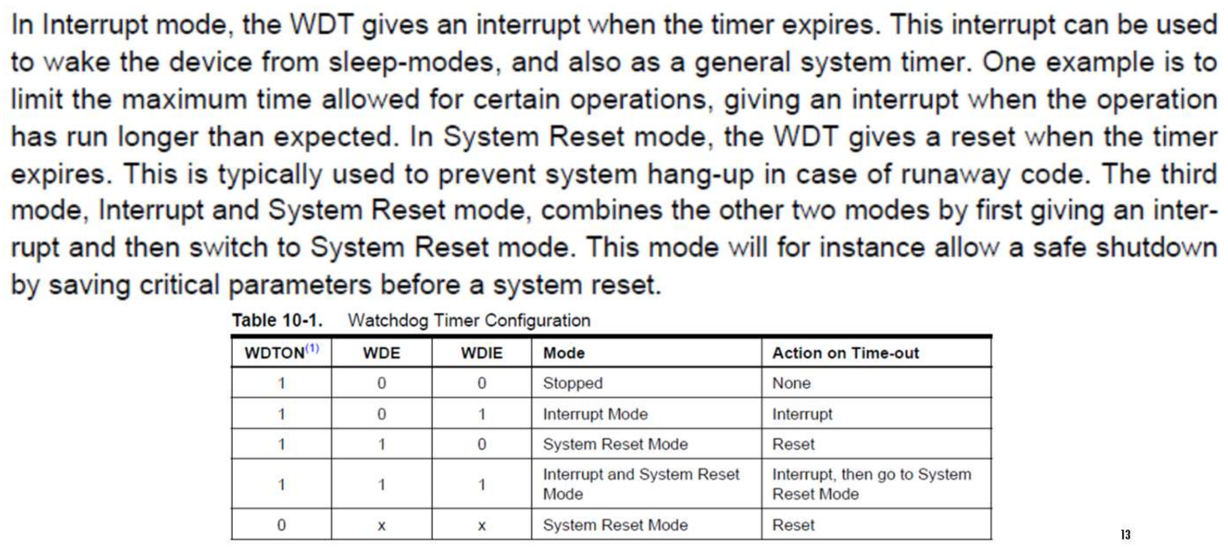

In Interrupt mode, the WDT gives an interrupt when the timer expires. This interrupt can be used to wake the device from sleep-modes, and also as a general system timer. One example is to limit the maximum time allowed for certain operations, giving an interrupt when the operation has run longer than expected. In System Reset mode, the WDT gives a reset when the timer expires. This is typically used to prevent system hang-up in case of runaway code. The third mode, Interrupt and System Reset mode, combines the other two modes by first giving an interrupt and then switch to System Reset mode. This mode will for instance allow a safe shutdown by saving critical parameters before a system reset.

Table 10-1. Watchdog Timer Configuration

WDTON[1]	WDE	WDIE	Mode	Action on Time-out
1	0	0	Stopped	None
1	0	1	Interrupt Mode	Interrupt
1	1	0	System Reset Mode	Reset
1	1	1	Interrupt and System Reset Mode	Interrupt, then go to System Reset Mode
0	x	x	System Reset Mode	Reset

13

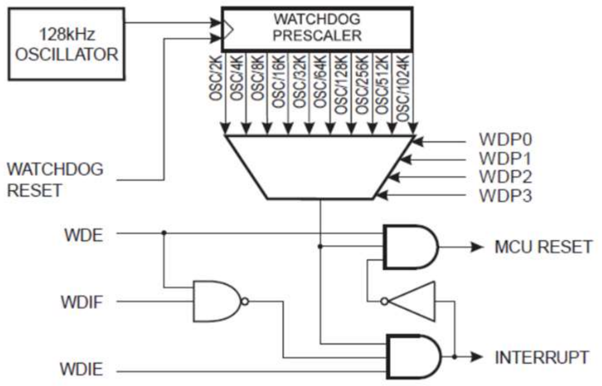
Watchdog Timer

- Suppose your application is heating a room
- Once the temperature is too high, the application should turn off
- Implement a watchdog timer:
  - An independent heat sensor causes an ISR (e.g., external interrupt) when the measured temperature is low enough
  - This ISR resets the watchdog and disables itself
  - The main program regularly enables the ISR
  - The watchdog will turn off the system if
    - The sensor breaks → no ISR will be called → the watchdog is not reset → the watchdog will count down to 0 and causes the system to be reset (with or without executing a watchdog ISR before reset)
    - The temperature is too high → no ISR will be called → etc.
- Safety: if sensor breaks or if temperature is to high, the system is reset
  - In SW one can always reset the system if the temperature is too high, but how does it know whether the sensor that measures the temperature is functioning correctly?

14

Watchdog Timer

Figure 10-7. Watchdog Timer



Watchdog reset restarts the counter before the time-out value is reached:

- #include <avr/wdt.h>
- wdt_reset();

Watchdog Timer

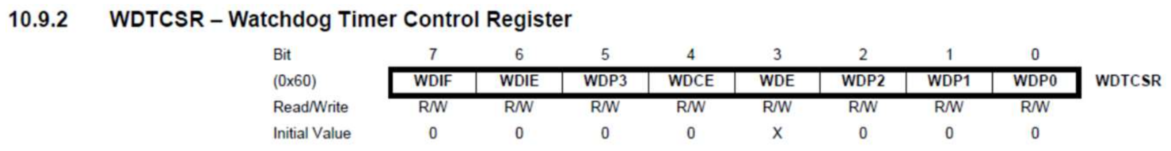
10.9.2 WDTCSR – Watchdog Timer Control Register

Bit	7	6	5	4	3	2	1	0	
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	WDTCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	X	0	0	0	

To further ensure program security, alterations to the Watchdog set-up must follow timed sequences. The sequence for clearing WDE and changing time-out configuration is as follows:

1. In the same operation, write a logic one to the Watchdog change enable bit (WDCE) and WDE. A logic one must be written to WDE regardless of the previous value of the WDE bit.
2. Within the next four clock cycles, write the WDE and Watchdog prescaler bits (WDP) as desired, but with the WDCE bit cleared. This must be done in one operation.

Watchdog Timer



10.9.2 WDTCSR – Watchdog Timer Control Register

Bit	7	6	5	4	3	2	1	0	
(0x60)	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0	WDTCSR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	X	0	0	0	

- WDTCSR |= (1<<WDCE) | (1<<WDE);
  - WDCE: Watchdog Change Enable allows to make changes during the next 4 cycles (= one operation)
  - WDE: Watchdog Enable
- WDTCSR = (1<<WDIE)|(1<<WDE)|(1<<WDP3);
  - This operation clears the WDCE bit as required by using = (not |=)
  - WDIE: Watchdog Interrupt Enable → E.g., we want an interrupt after 4.0 seconds
  - WDE: Watchdog Enable means a system reset is generated after 4.0 seconds
  - (1<<WDP3) sets a prescalar

17

Watchdog Timer

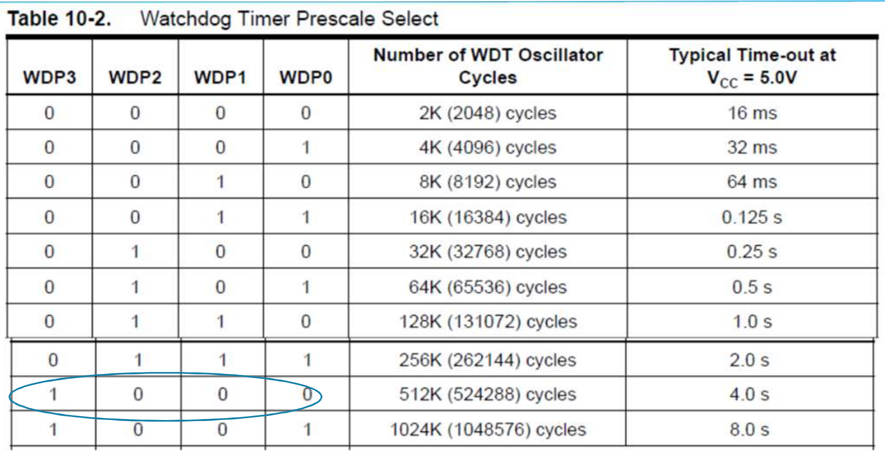
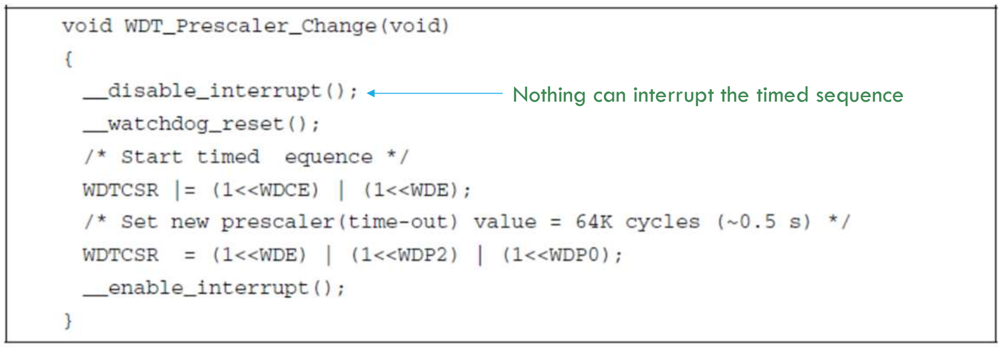


Table 10-2. Watchdog Timer Prescale Select

WDP3	WDP2	WDP1	WDP0	Number of WDT Oscillator Cycles	Typical Time-out at V_{CC} = 5.0V
0	0	0	0	2K (2048) cycles	16 ms
0	0	0	1	4K (4096) cycles	32 ms
0	0	1	0	8K (8192) cycles	64 ms
0	0	1	1	16K (16384) cycles	0.125 s
0	1	0	0	32K (32768) cycles	0.25 s
0	1	0	1	64K (65536) cycles	0.5 s
0	1	1	0	128K (131072) cycles	1.0 s
0	1	1	1	256K (262144) cycles	2.0 s
1	0	0	0	512K (524288) cycles	4.0 s
1	0	0	1	1024K (1048576) cycles	8.0 s

18

Watchdog Timer



```
void WDT_Prescaler_Change(void)
{
  __disable_interrupt();
  __watchdog_reset();
  /* Start timed  equence */
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  /* Set new prescaler(time-out) value = 64K cycles (~0.5 s) */
  WDTCSR  = (1<<WDE) | (1<<WDP2) | (1<<WDP0);
  __enable_interrupt();
}
```

Nothing can interrupt the timed sequence

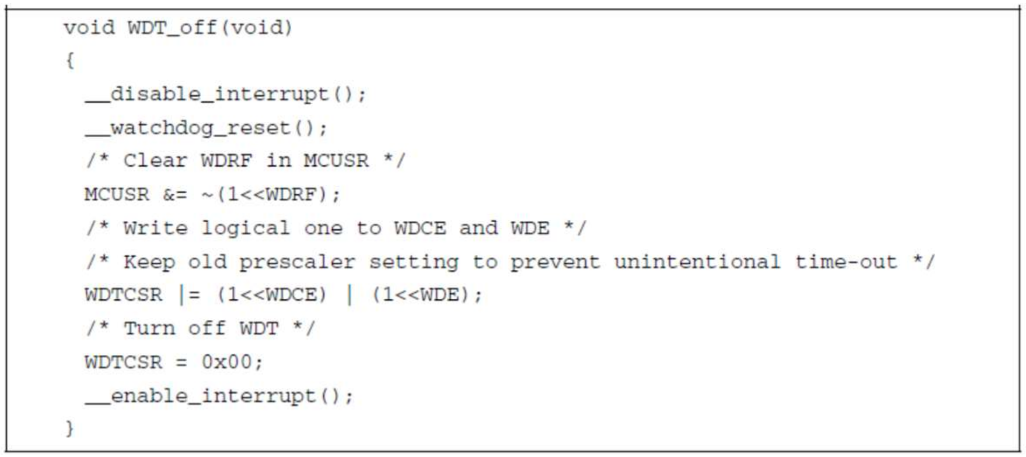
Another example from the datasheet without interrupt enable

Watchdog Timer

- If we have enabled the interrupt, an interrupt is created before the system is reset
- E.g., ISR(WDT_vect) { Store state in eeprom }
- After system reset the initialization can read the last state from eeprom

- If a reset occurs, it is good practice to turn of the watchdog as soon as the MCU starts
  - The register contents survive after restart: this means the watchdog is enabled (and reset)
  - If initialization takes too long, then the watchdog will time out and the MCU turns off: the MCU will never get through the initialization
  - So, turn off the watchdog at the start of your code, do the initialization, and turn on the watchdog before entering the main while(1){ …} loop

Watchdog Timer



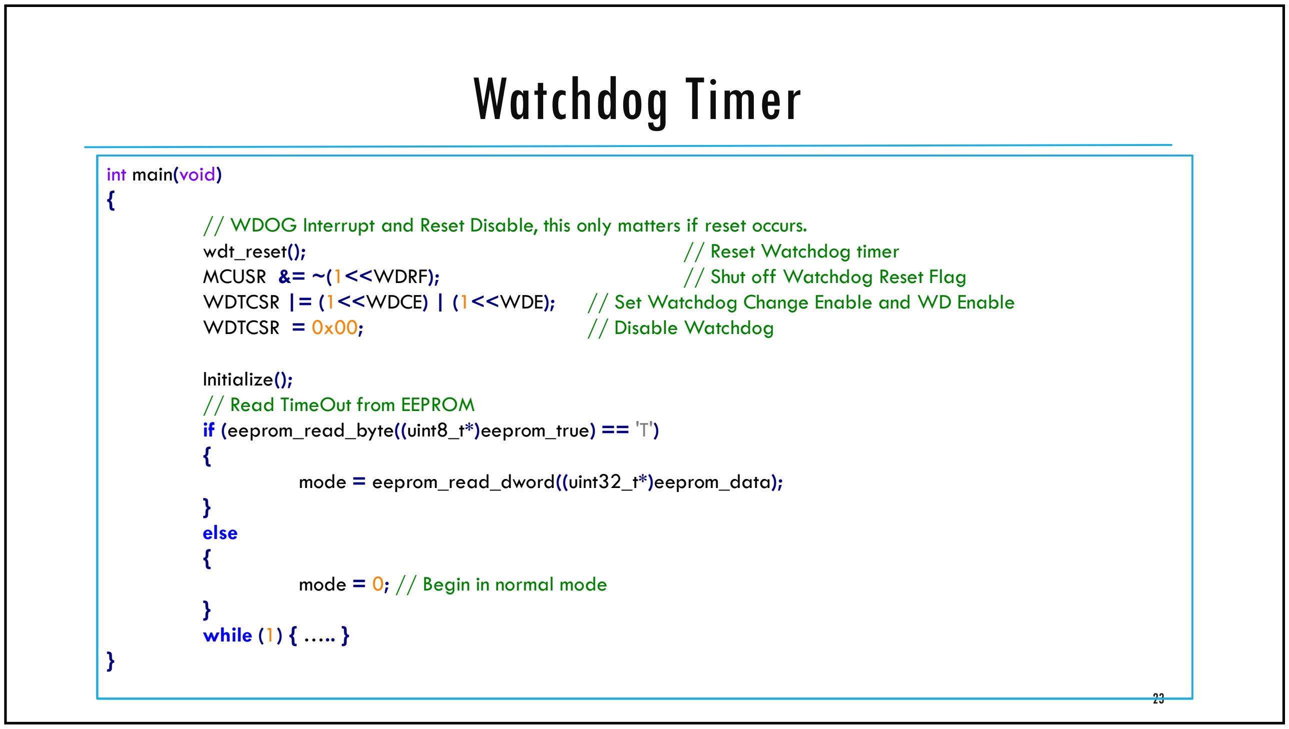
```
void WDT_off(void)
{
  __disable_interrupt();
  __watchdog_reset();
  /* Clear WDRF in MCUSR */
  MCUSR &= ~(1<<WDRF);
  /* Write logical one to WDCE and WDE */
  /* Keep old prescaler setting to prevent unintentional time-out */
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  /* Turn off WDT */
  WDTCSR = 0x00;
  __enable_interrupt();
}
```

21

Watchdog Timer

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

#include <avr/eeprom.h>
#define eeprom_true 0  //Suppose you want to store a flag at position 0
#define eeprom_data 1 //Suppose you want to store data at position 1

ISR (WDT_vect)
{
        eeprom_write_dword((uint32_t*)eeprom_data,mode);       //Write our current mode to EEPROM
        eeprom_write_byte((uint8_t*)eeprom_true, 'T');         //Set write flag TRUE
}

void Initialize(void)
{
        … all other initialization …
        WDTCSR |= (1<<WDCE) | (1<<WDE);  // Set Watchdog Condition Edit for four cycles
        WDTCSR = (1<<WDIE) | (1<<WDE) | (1<<WDP3);   // Set WDT Int and Reset; Prescalar at 4.0s.
}
```

22



Watchdog Timer

```
int main(void)
{
        // WDOG Interrupt and Reset Disable, this only matters if reset occurs.
        wdt_reset();                                          // Reset Watchdog timer
        MCUSR  &= ~(1<<WDRF);                                 // Shut off Watchdog Reset Flag
        WDTCSR |= (1<<WDCE) | (1<<WDE);               // Set Watchdog Change Enable and WD Enable
        WDTCSR  = 0x00;                               // Disable Watchdog

        Initialize();
        // Read TimeOut from EEPROM
        if (eeprom_read_byte((uint8_t*)eeprom_true) == 'T')
        {
                mode = eeprom_read_dword((uint32_t*)eeprom_data);
        }
        else
        {
                mode = 0; // Begin in normal mode
        }
        while (1) { ….. }
}
```

23

ECE3411 – Fall 2017
Lab 4c.

PWM: Signal Generation & Rectification

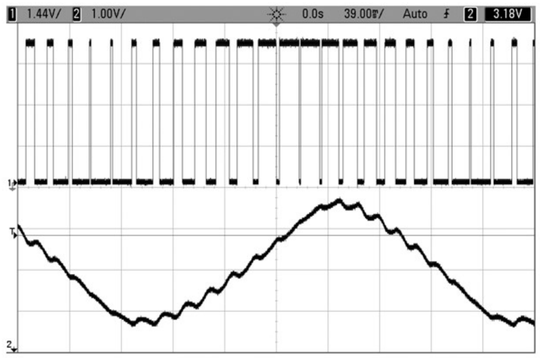
Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: marten.van_dijk@uconn.edu

Copied from Lab 6a, ECE3411 – Fall 2015, by
Marten van Dijk and Syed Kamran Haider

UCONN

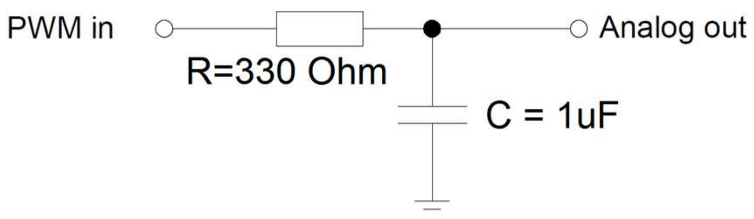
Task1: PWM – Rectified Sine Waveform

- In this task, you need to generate a \`rectified' 62.5Hz sine waveform using PWM.
- Generate a 64kHz PWM signal at PB2 using *Timer1* such that the duty cycle of the PWM is a function of a 62.5Hz sine wave i.e. for *f=62.5, duty_cycle* ~ sin2πft
- An example of such a PWM signal is shown below, where the duty cycle follows a sine function.



Connections

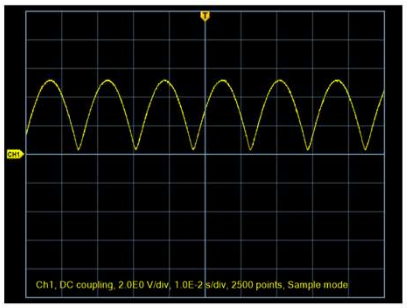
- Connect the PWM signal to a RC low pass filter as shown below.



3

Connections

- Connect the output of the low pass filter to an oscilloscope and observe the resulting waveform.
- This waveform should look like the one shown below, where the negative half cycles are also transformed into positive half cycles.



Ch1, DC coupling, 2.0E0 V/div, 1.0E-2 s/div, 2500 points, Sample mode

4

Task1a: Frequency Correction

- Correct frequency of the resulting sine wave.
- Hint 1: Include **math.h** library and use sin() function to compute the sine value.
- Hint 2: Use Timer0 to generate the argument to sin() function.

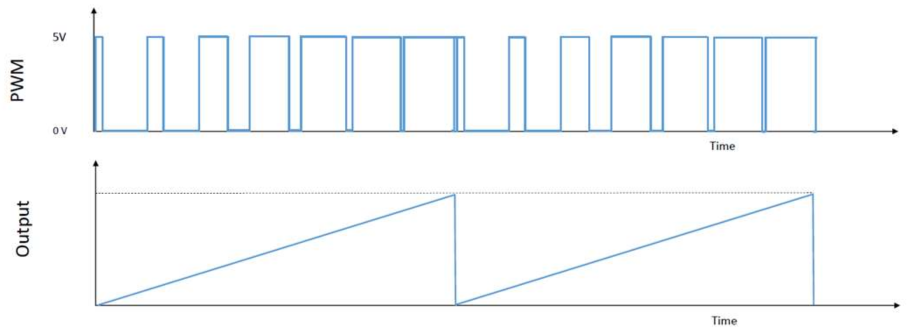
Task1b: Improving Signal Quality

- You are required to improve the quality of the resulting signal.
- Hint: Signal quality depends upon the number of steps taken from 0 to 2π to update duty cycle.

- **Notice that for this task, *_delay_ms() / _delay_us()* function calls are not allowed.**

Task2: PWM – Rectified Sawtooth Waveform

- In this task, you need to generate a saw tooth waveform using PWM.
- Generate a 64kHz PWM signal at PB2 using *Timer1* such that the duty cycle of the PWM is such that it results in a 10Hz saw tooth waveform.
- An example of such a PWM signal is shown below, where the duty cycle results in a saw tooth wave form.

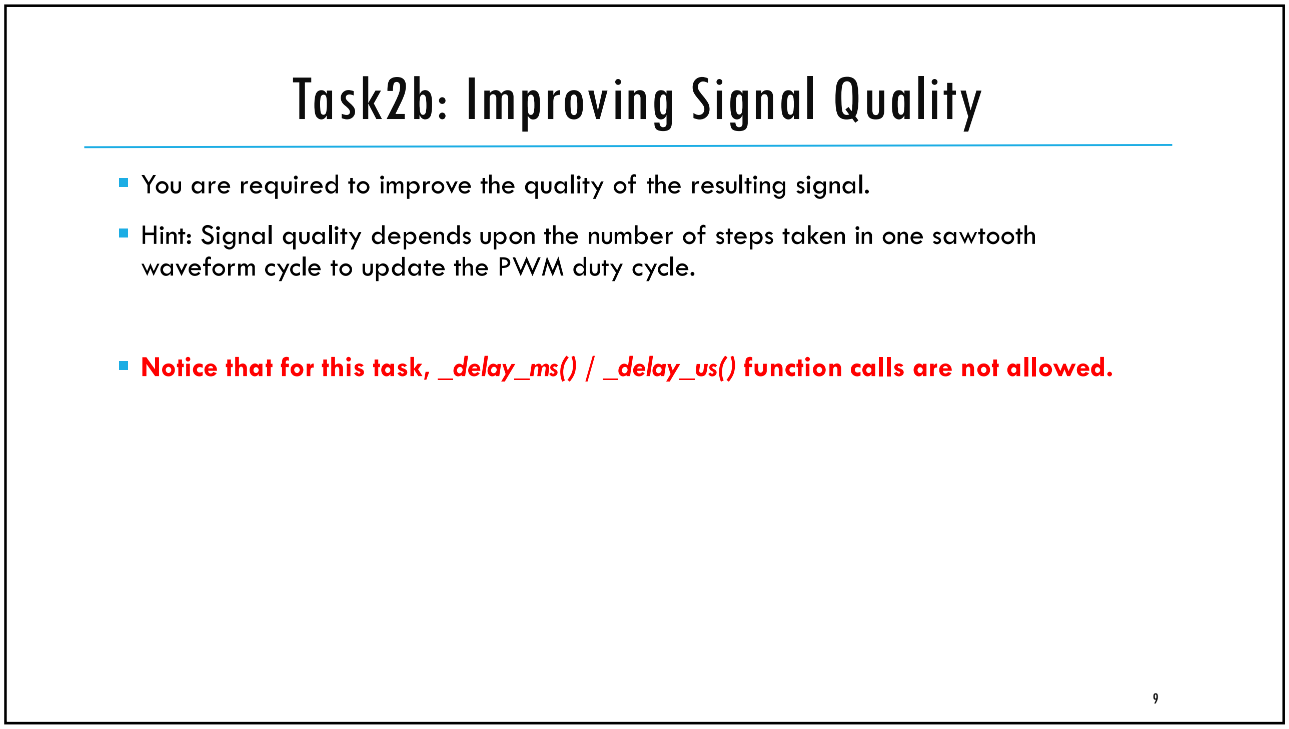


7

Task2a: Frequency Correction

- Similar to Task1a; Correct the frequency of the resulting saw tooth waveform.

8



Task2b: Improving Signal Quality

- You are required to improve the quality of the resulting signal.
- Hint: Signal quality depends upon the number of steps taken in one sawtooth waveform cycle to update the PWM duty cycle.

- **Notice that for this task, *_delay_ms()* / *_delay_us()* function calls are not allowed.**

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set P4

There are 5 questions in this problem set. Answer each question according to the instructions given in at least 3 sentences on own words.

If you find a question ambiguous, be sure to write down any assumptions you make.
Be neat and legible. If we can't understand your answer, we can't give you credit!

Any form of communication with other students is considered cheating and will merit an F as final grade in the course.

SUBMIT YOUR ANSWERS IN A HARDCOPY FORMAT.

Do not write in the box below

1 (x/24)	2 (x/20)	3 (x/30)	4 (x/16)	5 (x/10)	Total (xx/100)

Name:

Student ID:

1. [24 points]: Assume a clock frequency of $f_{clk} = 20\text{MHz}$ and the following initialization:

```
DDRD   = 0x10;
OCR1A  = 39062;
OCR1B  = 13020;
TCCR1A = 0b00110011;
TCCR1B = 0b00011101;
```

Answer the following questions:

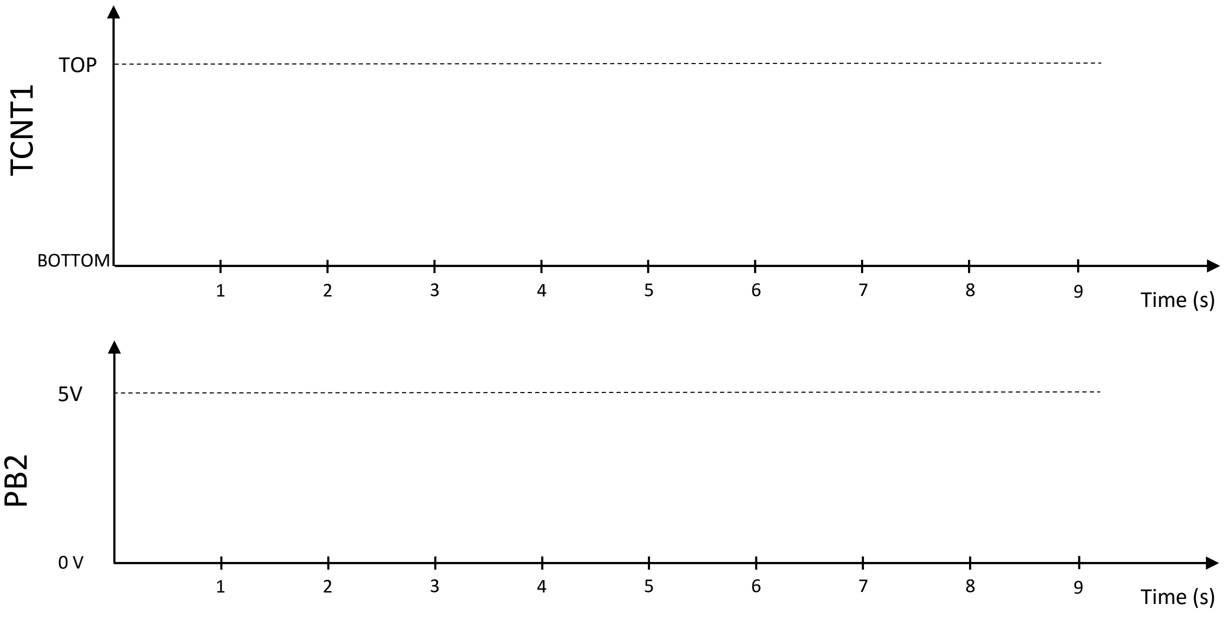
a. In which mode is `Timer1` running?

b. What is the numerical value of 'TOP' for `Timer1` in this mode?

c. How much time (in seconds) does it take for `Timer1` to complete one full cycle, i.e. going from BOTTOM $\rightarrow$ TOP $\rightarrow$ BOTTOM? Be as accurate as possible in your calculations.

Initials:

d. Starting from the moment of `Timer1`'s initialization, draw the waveforms of the `TCNT1` register value and the pin PB2 value w.r.t. time. Please draw the waveform strictly according to the timing scale shown on X-axis, otherwise no credit will be given.



e. In each full cycle of `Timer1`:

- For how much time (in seconds) is PB2 low? Be as accurate as possible in your calculations.

- For how much time (in seconds) is PB2 high? Be as accurate as possible in your calculations.

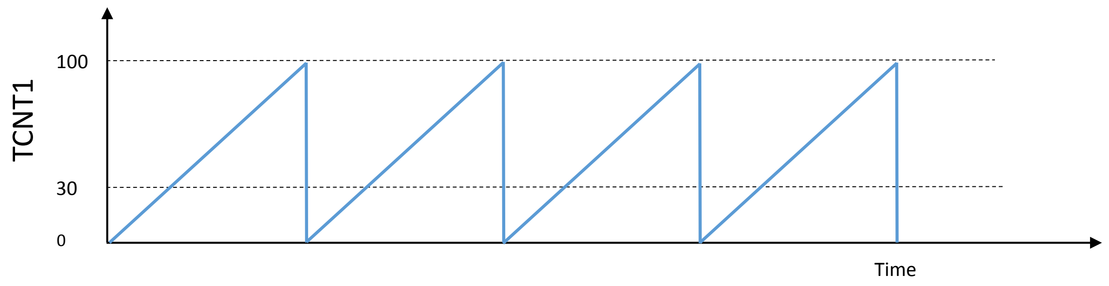
Initials:

2. [20 points]: The code given below generates PWM signal using Timer 1.

```
int main(void)
{
    DDRB = 0xFF;    // Setting Port B as output

    // Setting up Timer1
    OCR1A  = 100;
    OCR1B  = 30;
    TCCR1B |= (1<<WGM13) | (1<<WGM12);  // Turn on Fast PWM mode
    TCCR1A |= (1<<WGM11) | (1<<WGM10);  // Fast PWM mode with OCR1A as TOP
    TCCR1A |= (1<<COM1B1)|(1<<COM1B0);  // OC1B sets on compare match, clears at BOTTOM
    TCCR1B |= (1<<CS12);                // Set pre-scalar to divide by 256

    while(1);    // Nothing to do
}
```

a. At which pin of the microcontroller does this code produce the PWM signal? You may write the logical name of the pin if you don't remember the corresponding physical pin.

b. The figure below shows the TCNT1 register behavior over time. Draw the resulting PWM signal in the space provided below.



Initials:

3. [30 points]: You have designed a digital thermometer using the ATmega328P ADC and the temperature sensor. The ADC is running on full resolution and its reference voltage is set to $1.1V$.
If the temperature sensor produces $314mV$ at $25°C$ and its voltage sensitivity is $1mV/°C$ then answer the following questions:

a. *Theoretically*, what is the **maximum** and **minimum** value of temperature that you can measure using this thermometer?

b. What is the **smallest change** in temperature (in $°C$) that can be detected by this thermometer? Be as accurate as possible in your calculations.

c. If the ADC reference voltage is changed to $512mV$, then what would be the **smallest change** in temperature (in $°C$) that can be detected by this thermometer?

Initials:

4. [16 points]: The code measures voltage every $1ms$ through ADC in sleep mode. The user can also send characters to the MCU over UART. The program keeps track of the average and standard deviation of the measurements over the last $50ms$ using a circular buffer for storing voltage values.

```
void Task_ADCMeasure(void)
{
    int8_t flag = 1;
    while (flag !=0) { flag = (UCSR0A ^ (1<<TXC0)) & ((1<<RXC0) | (1<<TXC0)); }
    sleep_cpu();

    //statistics over last window samples
    volt_index = ((1.0*Ain)/1024.00)*5.00;    // 10 bit accuracy, AREF=5V
    volt = v_buffer[v_index];
    v_buffer[v_index] = volt_index;
    v_index++;
    v_index = (v_index % window);
    Sum1 = Sum1 - volt + volt_index;
    Sum2 = Sum2 - (volt*volt) + (volt_index * volt_index);

    // Computes Moving Average and Standard Deviation
}
```

a. Does the circular buffer approach shown in the code above produce accurate results for ADC noise measurement statistics? Explain your answer.

b. Can UART characters' reception be corrupted during the ADC operation shown above? Explain your answer.

Initials:

5. [10 points]: Can you shortly describe what you have learned and feel confident about using in the future?

End of Problem Set

Initials:

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set A4

There are 4 questions in this problem set. Answer each question according to the instructions given in at least 3 sentences on own words.

If you find a question ambiguous, be sure to write down any assumptions you make.
Be neat and legible. If we can't understand your answer, we can't give you credit!

Any form of communication with other students is considered cheating and will merit an F as final grade in the course.

SUBMIT YOUR ANSWERS IN A HARDCOPY FORMAT.

Do not write in the box below

1 (x/28)	**2 (x/30)**	**3 (x/20)**	**4 (x/22)**	**Total (xx/100)**

Name:

Student ID:

1. [28 points]: Given that clock frequency ($clk_{I/O}$) of ATmega328P is 8MHz.
Assume the following about the code snippet given below:

- Each one of `instruction_1`, `instruction_2`, ..., `instruction_52` takes 4 CPU cycles.
- Evaluating `while(1)` statement takes zero CPU cycles.
- Evaluating `if( !(ADCSRA & (1<<ADSC)) )` statement and executing its body take zero CPU cycles.

```
#define F_CPU 8000000UL
#include <avr/io.h>

/* Main Function */
int main(void)
{
    /* Configuring ADC Control and Status Register A */
    ADCSRA = 0x86;

    while(1)
    {
        instruction_1;
        instruction_2;
        instruction_3;
        ...
        ...
        instruction_52;
        if( !(ADCSRA & (1<<ADSC)) )
        {
            ADCSRA |= (1<<ADSC);    // Start A to D Conversion
        }

    } /* End of while(1) Loop */

} /* End of main() */
```

Answer the following questions about the code snippet.

a. Given that it takes 13 ADC cycles, how much time (in microseconds) does it take to complete one ADC conversion?

b. What prevents the condition "`if( !(ADCSRA & (1<<ADSC)) )`" from being satisfied?

c. How much time (in microseconds) does it take to complete one iteration of "`while(1)`" loop?

d. What is the percentage of `while(1)` loop iterations for which the body of "`if( !(ADCSRA & (1<<ADSC)) )`" condition is executed?

Initials:

2. [30 points]: Given that the clock frequency ($clk_{I/O}$) of ATmega328P is 16MHz, write a program that uses watchdog timer in interrupt and reset modes simultaneously. The detailed functionality of the program is as follows:

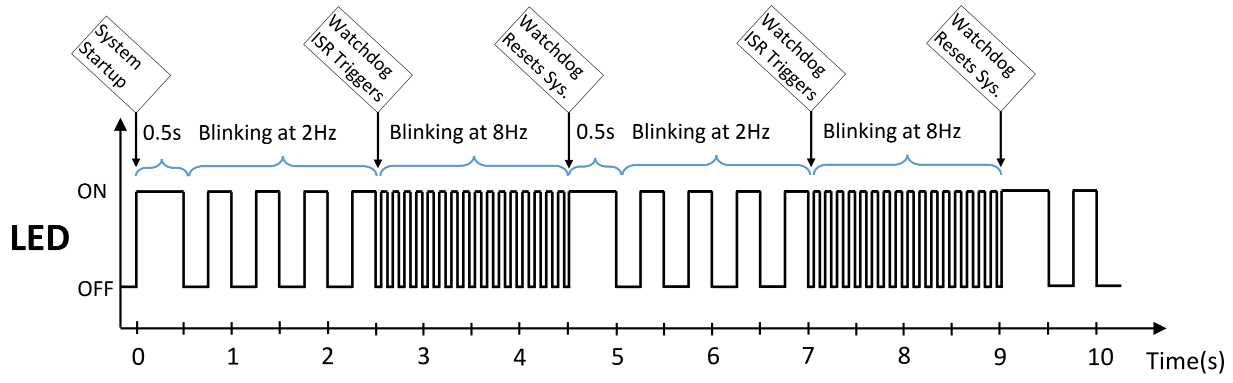
(a) Upon the system startup/reset, a LED connected to PB5 lights up for $0.5s$ and then turns off.

(b) After this, the main function starts blinking the LED at approximately $2Hz$.

(c) After 2 seconds, the watchdog interrupt occurs and it keeps blinking the LED at $8Hz$ until the system reset occurs.

To simplify the implementation, use `_delay_ms()` or `_delay_us()` routines inside `while(1)` loops to implement the LED blinking function.

The following figure shows the detailed timing of the LED for the desired system. Notice that, after the watchdog interrupt, it takes another watchdog timeout period for the system reset to occur.



Implement this system by filling the gaps in the provided code layouts of the subsections A, B and C. You may use the provided data sheet for your reference.

A. Initialization: (10 points)
Complete the function `initialize_all(void)` as instructed below:

```
// Define any variables here if needed




/* Initialization function */
void initialize_all(void)
{
    /* Configure the LED pin and implement the functionality of step (a) */










    /* Configure the Watchdog timer in Reset & Interrupt mode */
    /* Set a prescaler such that watchdog times out after 2 seconds */









    /* Any other initializations here if needed */








} /* End of initialize_all() */
```

Initials:

B. Watchdog timeout ISR Implementation: (10 points)
Write the ISR `ISR(WDT_vect)` to achieve the desired functionality.

```
/* Watchdog timeout ISR */
ISR(WDT_vect)
{
    /* Blink the LED at 8Hz using _delay_ms() or _delay_us() function */











} /* end of Watchdog timeout ISR */
```

Initials:

C. Main Function Implementation: (10 points)
Write the function `main()` to complete the system functionality.

```
/* Main Function */
int main(void)
{
    /* Cleanup any aftereffects of Watchdog timeout reset */









    initialize_all();     // Initialize everything
    sei();                // Enable Global Interrupts

    /* Event loop */
    while(1)
    {
        /* Blink the LED at 2Hz using _delay_ms() */









    }

} /* End of main() */
```

Initials:

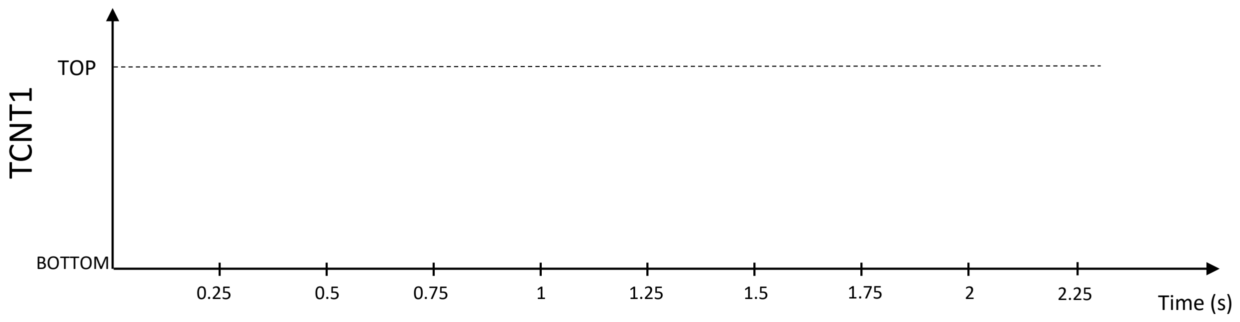
3. [20 points]: Assume a clock frequency of $f_{clk} = 16$MHz. Read the following initialization and ISRs:

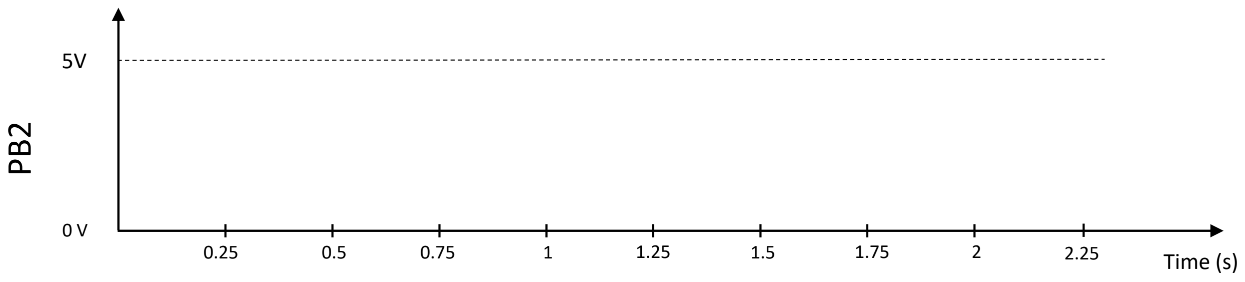
```
#define MIN_TICKS 15624
#define MAX_TICKS 62499

// PWM variables
volatile uint16_t duty_cycle;
volatile uint16_t time_period;
volatile uint8_t toggle_flag;
int percentage_duty_cycle;

void initialization ()
{

    DDRB |= (1<<DDB2);
    time_period = MAX_TICKS;
    duty_cycle = time_period/4;
    toggle_flag = 0;

    // Setup Timer1
    OCR1A = time_period;
    OCR1B = duty_cycle;
    TCCR1A |= (1<<WGM11) | (1<<WGM10);
    TCCR1B |= (1<<WGM13) | (1<<WGM12);
    TCCR1A |= (1<<COM1B1);
    TIMSK1 |= (1<<OCIE1A);
    TCCR1B |= (1<<CS12);
}
// Timer 1 Compare Match A ISR (TCNT1 = OCR1A)
ISR (TIMER1_COMPA_vect)
{
    if(toggle_flag)
    {
        if( time_period > MIN_TICKS)
        {
            time_period = time_period/2;
            duty_cycle = time_period/4;
        }
        else
        {
            toggle_flag ^= 1;
        }
    }

    else
    {
        if( time_period < MAX_TICKS)
```

Initials:

```
            {
                time_period = (time_period * 2) +1;
                duty_cycle = time_period/4;
            }
            else
            {
                toggle_flag ^= 1;
            }
        }

        OCR1A = time_period;
        OCR1B = duty_cycle;
    }
```

Starting from the moment of `Timer1`'s initialization, draw the waveforms of the `TCNT1` register value and the pin PB2 value w.r.t. time. Please draw the waveform strictly according to the timing scale shown on X-axis, otherwise no credit will be given.





Initials:

4. [22 points]: Given that the clock frequency ($clk_{I/O}$) of ATmega328P is 16MHz, you want to implement two pins (A and B) that output PWM signals.

a. The frequency of each PWM signal is 1MHz, and the initial duty cycle of these two PWM signals is 50%. Please write the initialization function for timer0 (signal A) and timer1 (signal B).

```
void initialization ()
{




}
```

Initials:

b. The duty cycle of A and B (duty_cycle_A and duty_cycle_B) should be updated in the ISRs according to the following rules:

(a) For A: If duty_cycle_B $> 90\%$, then duty_cycle_A = (1+duty_cycle_A)/2. If duty_cycle_B $< 10\%$, then duty_cycle_A = duty_cycle_A /2. In other cases, duty_cycle_A does not change.

(b) For B: If duty_cycle_A $< 50\%$, then duty_cycle_B = (1+duty_cycle_B)/2. If duty_cycle_A $\geq 50\%$, then duty_cycle_B = duty_cycle_B /2.

Please write the ISRs for these two timers.

```
ISR (TIMER0_COMPA_vect)
{



}


ISR (TIMER1_COMPA_vect)
{



}
```

End of Problem Set

Initials:

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Independent LAB4

There are 2 independent lab questions in LAB4.

You may not discuss independent labs in any way, shape, or form with anyone else and you are not allowed to lookup solutions from other sources.

Any form of communication with other students or looking up solutions is considered cheating and will merit an F as final grade in the course.

Name:

Student ID:

1. [Pass/Fail points]: In this task, you need to design a digital thermometer using the ADC and an external temperature sensor. The thermometer should display the room temperature in both Celsius and Fahrenheits down to $1/10th$ of a degree.
You may refer to the provided data sheet of the temperature sensor (MCP9701A).

a. First, connect a potentiometer to the ADC channel and sample the analog input voltage after every second.

- The potentiometer should generate a variable voltage between 0V and 5V.
- Print the current voltage (in millivolts) on LCD.

b. Now replace the potentiometer with a temperature sensor (MCP9701A) to read the temperature every second.

- Convert the input voltage from the temperature sensor to the equivalent temperature.
- Print the temperature reading on LCD in both Celsius and Fahrenheits down to $1/10th$ of a degree.
- Play with different resolutions of the ADC and different internal and external voltage reference values. What observations do you make?

Hint: The temperature sensor produces $400mV$ at 0 degree Celsius.

Initials:

2. [Pass/Fail points]: Complete the following tasks.

a. Your task is to extend the simple ADC voltage measurement code from Lab4b:Task1 with a watch-dog timer:

- The watch dog timer should be set up such that if ADC reads $\geq 4V$ continuously for a period of 4 seconds, only then a system reset must occur. Note that if ADC reads $< 4V$ at anytime after reading $\geq 4V$ but before the 4 seconds widnow has elapsed, then the system reset should not occur.
- Before entering the main loop print Starting . to the terminal (this allows you to see when a system reset actually occurs).
- Print a counter value on LCD where the counter is incremented every second.
- Before the system resets, the watch dog timer ISR should store in EEPROM the current value of the counter. This value should be loaded from EEPROM before entering the main loop and the counter should continue from this value onwards.

Initials:

b. Analyzing Assembly Code: Analyze the assembly of the provided C code that programs $Timer0$ in CTC mode to trigger Compare Match A ISR after every 1ms:

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <inttypes.h>
#include <avr/interrupt.h>

// variables
volatile uint16_t compare_matches;

//------------------------------------------------------------------------------
// All initializations
void initialize_all(void)
{
// Setup Timer0
TIMSK0 |= (1<<OCIE0A); // Enable Compare Match A Interrupt
OCR0A = 249; // 250 ticks
TCCR0A |= (1<<WGM01); // CTC Mode
TCCR0B |= (1<<CS01) | (1<<CS00); // Prescaler = 64 ==> Overflow every 1ms
}

//------------------------------------------------------------------------------
// Timer 0 Compare Match A ISR
ISR (TIMER0_COMPA_vect)
{
compare_matches++;
}

//-----------------------------------------------------------------------------
int main(void)
{
initialize_all();
sei(); // Enable global interrupts

while (1);
}
```

- Increment a global counter inside the Compare Match A ISR.
- Use Atmel Studio debugger to see the Assembly code of your program.
- By stepping through the Assembly instructions one by one in the debugger, explain the sequence of branch and jump instructions executed to call the Initialization function and TIMER0_COMPA_vect ISR.
- In particular, how does Interrupt Vector Table help in this regard?

Initials:

End of LAB4

Initials: