ECE3411 – Fall 2016

Lecture 5b.

ADC: Analog to Digital Conversion

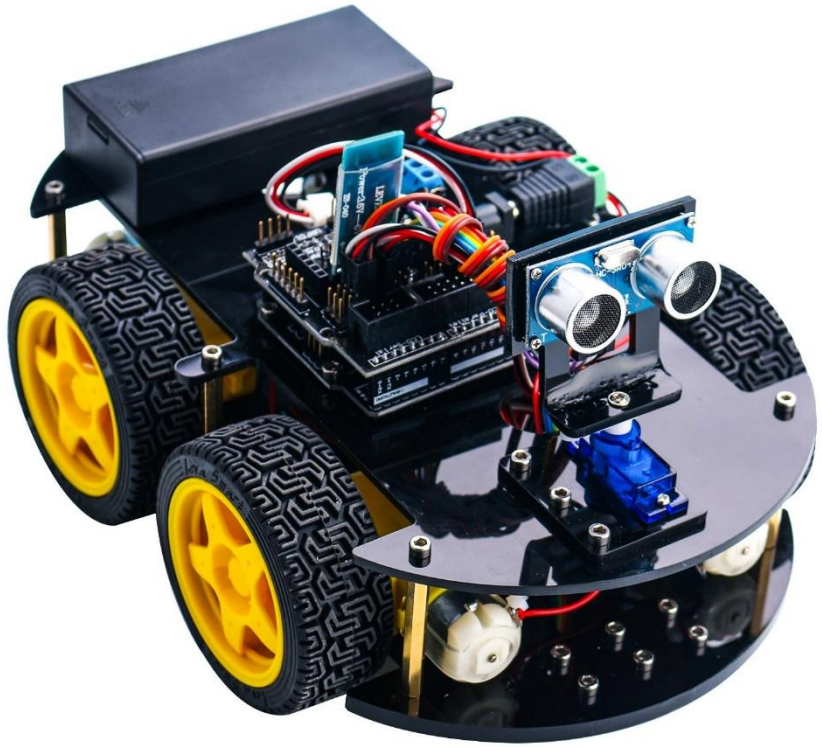
Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {marten.van_dijk , syed.haider}@uconn.edu

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

UCONN

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



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.

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 ..

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.

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)

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);
```

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;
}
```

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();
```

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;
}
```

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 ☺)

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;
}
```

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 ...
}
```

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?
```

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();
```

What is happening in the following code?

```
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 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.

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.

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)
- ?????

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)
- ?????

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()
- ?????

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.

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.

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.