3. Analog to digital converter of MC9S12DJ256

3.1 The purpose of the work

This paper aims to familiarize you to the concept of analog to digital converter in a microcontroller.

Skills expected at the end of this lab are:
- configuring the analog to digital converter according to data requirements;
- testing the proposed application using the methodology from the paper presented in the previous chapter.

3.2 Analog to Digital Converter

Most embedded systems are designed to cope applications involving certain non-electrical quantities: humidity, pressure, temperature, intensity, speed, weight and so on. These non-electric quantities are analogue in nature therefore need to be converted into digital form for processing.

An ADC can operate only with voltage levels. Any non-electric quantity is necessary to be converted into an electrical quantity using a particular type of transducer. As the output of such a is not suitable for binding one analogue-to-digital converter, a special circuit could be provided to translate the output of the transducer in a range of values appropriate for analog-digital conversion (Figure 24).

Figure 24 – Analog to digital conversion process (Huang,2010)

Figure 25 - The output characteristic of an A / D n-bit ideal converter

An ADC on n bits has 2 * n possible output values. The ideal characteristic of an ideal ADC on n bits is shown in Figure 25. The surface located above and below the dotted line is called *quantization error*. Using n-bit for an analog-to-digital conversion, the average error will be 2n + 1. A real ADC can have different *nonlinearity and monotony errors.*

An analog-to-digital conversion requires two types of voltage: low reference voltage (V_{RL}) and a high reference voltage (V_{RH}). Typically, V_{RL} is connected to 0 logic and V_{RH} to V_{DD}(5V). Typically, an analog input of 0 V (or V_{RL}) is converted to digital 0; an analog input V_{DD}(or V_{RH}) is converted to digital in $2^n – 1$, and a type k V input, in digital will be represented by a value of type: k □ $(2^n – 1)$ □ V_{DD}. The result of analog-to-digital conversion, k, can be converted back to an analog voltage V_k by the following equation:

$$V_K = V_{RL} + ((V_{RH} - V_{RL}) \times k) \div (2^n – 1).$$

Example:
You have an ADC on 10-bit with V_{RL} = 1 V and V_{RH} = 4V . Find the appropriate voltage for each of the following values obtained from the analog to digital conversion: 25, 80, 240, 500, 720, 800 and 900.
Solution:

$V(25) = 1\ V + (3 \times 25) \div (2^{10} - 1) = 1.07\ V$

$V(80) = 1\ V + (3 \times 80) \div (2^{10} - 1) = 1.23\ V$

$V(240) = 1\ V + (3 \times 240) \div (2^{10} - 1) = 1.70\ V$

$V(500) = 1\ V + (3 \times 500) \div (2^{10} - 1) = 2.47\ V$

$V(720) = 1\ V + (3 \times 720) \div (2^{10} - 1) = 3.11\ V$

$V(800) = 1\ V + (3 \times 800) \div (2^{10} - 1) = 3.35\ V$

$V(900) = 1\ V + (3 \times 900) \div (2^{10} - 1) = 3.64\ V.$

3.2.1 The case of the ADC of the MC9S12DJ256 microcontroller

80-pin version of the HCS12 microcontroller has an analog to digital converter, ATD_10B8C, on 8-channel and 10-bit resolution.

Analog-digital conversion can be started by writing to a control register or by an external trigger. The conversion is carried out by the method of successive approximations, the result can be aligned right unsigned, signed aligned to the left and unsigned aligned to the left.

ADC has 12 terminals. AN7/PAD7 terminal serves as analog input for channel 7, can be configured to provide external trigger for analog to digital conversion. AN[6:0]/PAD[6:0] terminals can be set as analog inputs to the respective channels. V_{RH} and V_{RL} terminals are voltage references for analog to digital conversion, V_{DDA} and V_{SSA} are the power terminals.

A / D converter is divided into analog and digital blocks, a block diagram of which is shown in Figure 26 Analog block consists of the following components:
Sample & Hold: supports analog and holds cubic form of capacitive voltage to a storage node.
Analog Input Multiplexer: connects the 8 analog input channels to block Sample & Hold.

A/D converter is divided into analog and digital blocks, the block diagram is shown in Figure 26. The analog block consists of the following components:

- **Sample & Hold**: accepts analog signals and retain them in the form of capacitive voltages in a storage node.

- **Analog Input Multiplexer**: connects one of the 8 analog input channels to block Sample & Hold.

Figure 26 - Block diagram of A/D converter (Freescale, 2005)

- **Sample Buffer Amplifier**: is used to retain the analog input signal for Sample & Hold block.
- **Module Analog-to-Digital**: Performs the actual conversion. Conversion resolution is programmable via register to 8 or 10 bits. It works by comparing the retained analog value with a number of potentials generated digitally.

Offset Address	Utilization	Access
$_00	ATD Control Register 0 (ATDCTL0)[1]	R
$_01	ATD Control Register 1 (ATDCTL1)[2]	R
$_02	ATD Control Register 2 (ATDCTL2)	R/W
$_03	ATD Control Register 3 (ATDCTL3)	R/W
$_04	ATD Control Register 4 (ATDCTL4)	R/W
$_05	ATD Control Register 5 (ATDCTL5)	R/W
$_06	ATD Status Register 0 (ATDSTAT0)	R/W
$_07	Unimplemented	
$_08	ATD Test Register 0 (ATDTEST0)[3]	R
$_09	ATD Test Register 1 (ATDTEST1)	R/W
$_0A	Unimplemented	
$_0B	ATD Status Register 1 (ATDSTAT1)	R

$_0C	Unimplemented	
$_0D	ATD Input Enable Register (ATDDIEN)	R/W
$_0E	Unimplemented	
$_0F	Port Data Register (PORTAD)	R
$_10, $_11	ATD Result Register 0 (ATDDR0H, ATDDR0L)	R/W
$_12, $_13	ATD Result Register 1 (ATDDR1H, ATDDR1L)	R/W
$_14, $_15	ATD Result Register 2 (ATDDR2H, ATDDR2L)	R/W
$_16, $_17	ATD Result Register 3 (ATDDR3H, ATDDR3L)	R/W
$_18, $_19	ATD Result Register 4 (ATDDR4H, ATDDR4L)	R/W
$_1A, $_1B	ATD Result Register 5 (ATDDR5H, ATDDR5L)	R/W
$_1C, $_1D	ATD Result Register 6 (ATDDR6H, ATDDR6L)	R/W
$_1E, $_1F	ATD Result Register 7 (ATDDR7H, ATDDR7L)	R/W

Table 5 – Registers of ADC

Digital block provides the following features:

Only analog input signals falling within VRL to VRH (reference potential of the A / D conversion) will provide digital output corresponding codes. To be completed in Worksheet.

- **External Trigger Input (ETRIG):** allows the user to synchronize ATD to external events.
- **General Purpose Digital Input Port Operation**: input terminals can be multiplexed between analog and digital transmission terminals; as analog inputs, they are multiplexed and sampled to supply signals to the A / D converter, and as digital inputs the data can be accessed through Digital Port Register PORTAD (set only as input).
- **Low power modes.**

The registers of the analog to digital convertes are presented in Table 5. Every ADC module has these registers:

Six control registers: ATDxCTL0 to ATDxCTL5, where ATDxCTL0 and ATDxCTL1 are used only for testing at manufacturer level.

- Two status registers: ATDxSTAT0 and ATDxSTAT1
- Two testing registers: ATDxTEST0 and ATDxTEST1
- A register for input activation: ATDxDIEN
- A register for ports data: PTADx
- Eight result registers on 16 bits: ATDxDR0 to ATDxDR7; where x = 0 or 1.

Input signal V_{RL} = 0 V V_{RH}= 5.12 V	Codes 8-bit signed	Codes 8-bit unsigned	Codes 10-bit signed	Codes 10-bit unsigned
5.120 Volts	7F	FF	7FC0	FFC0
5.100	7F	FF	7F00	FF00
5.080	7E	FE	7E00	FE00
2.580	01	81	0100	8100
2.560	00	80	0000	8000
2.540	FF	7F	FF00	7F00
0.020	81	01	8100	0100
0.000	80	00	8000	0000

Table 6 - Output values

Codes obtained at output depending on input voltage and settings for data to be signed or unsigned, to be aligned to the left are illustrated in Table 6.

3.3 Development of the work

1. Definition of work requirements

    a. The achievement of analog to digital conversion of the value introduced by PADD0 potentiometer.

    b. The conversion is performed both on 8-bit and 10-bit. See the result in PcanView.

    c. Read of the analogue channels.

2. Open the CodeWarrior project related to work number 3, namely .mcp file ending. The code for this work is presented in the following (main.c).

```
////////////////////////////////////////////////////////////////////////////
//
// Sample for SofTec Microsystems ZK-S12-B Starter Kit
//
// ---------------------------------------------------------------------------#include <hidef.h>
#include "mc9s12dj256.h"
#include "mscan.h"
#pragma LINK_INFO DERIVATIVE "mc9s12dj256b"

////////////////////////////////////////////////////////////////////////////
// Defines and variables
////////////////////////////////////////////////////////////////////////////
unsigned char potentiometer_value;
Bool can_send_enable;

////////////////////////////////////////////////////////////////////////////
```

```
// Variables
////////////////////////////////////////////////////////////////////////////////
extern unsigned char *can_periph[];

////////////////////////////////////////////////////////////////////////////////
// Peripheral Initialization
////////////////////////////////////////////////////////////////////////////////
#pragma CODE_SEG
#define ADC_8_BIT 1
// this value sets the conversion on 8-bit or 10-bit

void PeriphInit(void)
{

  //Configure PB[7..0] as output port
   PORTB = 0x00;
   DDRB = 0xFF;

   // Configure ADC periphery
#   if(ADC_8_BIT == 1)
   // ATD0: 8 conversions per sequence, 8 bit resolution, continuous conversion
     ATD0CTL3 = 0x; // 8 conversions per sequence
     ATD0CTL4 = 0x; // 8 bit resolution
     ATD0CTL2 = 0x; // normal operation
     ATD0CTL5 = 0x; // left justified; scan mode across channels
#   else
   // ATD0: 8 conversions per sequence, 10 bit resolution, continuous conversion
     ATD0CTL3 = 0x; // eight conversions per sequence.
     ATD0CTL4 = 0x; // 10 bit resolution
     ATD0CTL2 = 0x; // normal operation---------------
//
// This project has been written for CodeWarrior 3.1 for HC(S)12
// and uses Registers Files (MC9S12DJ256.h and MC9S12DJ256.c) v. 1.052
//
// ---------------------------------------------------------------------------------------
//
// Copyright (c) 2005 SofTec Microsystems
// http://www.softecmicro.com/
//
////////////////////////////////////////////////////////////////////////////////
  ATD0CTL5 = 0x; // right justified; scan mode across channels
#   endif
   CANInit(0);
   EnableInterrupts;
}

////////////////////////////////////////////////////////////////////////////////
// Entry point
////////////////////////////////////////////////////////////////////////////////

void main(void)
{   unsigned char i=0;
   struct can_msg msg_send;
   struct can_msg *msg;
   PeriphInit();
```

```
msg=msg_rdy=&msg_buff[0];

 asm("cli");
for(;;)
{
 while(!(ATD0STAT0 & 0x80));
 # if(ADC_8_BI T== 1)
 PORTB=ATD0DR0H;
 # else
 PORTB=ATD0DR0L;
 # endif
 // Resets SCF flag
 ATD0STAT0 = 0x80;

 if (msg!=msg_rdy){
  // message received on CAN BUS
  msg_send.id = (unsigned int)0x000F00|msg->id;

  if(msg->id==CAN_MSG_ID_CMD){
   //decode command message
   switch(msg->data[0]){
    case 0: { // COMMAND 00: software version
          memcpy(msg_send.data,"SI-SW2.0",8);
          msg_send.len = 8;
          break;
    }
    case 1:{ // COMMAND 01: digital read (read from the corresponding port)
          char *p;
          // message length is less than 3-digit hexadecimal
          if(msg->len<3){
           memcpy(msg_send.data,"bad-len",7); // sending an error message
           msg_send.len = 7;
           }
          //the readings have to do with A or P ports
          else if(msg->data[1]!='A'&&msg->data[1]!='P'){
           memcpy(msg_send.data,"bad-port",8); // sending an error message
           msg_send.len = 8;
           }
          // maximum 8 pins can be read
          else if(msg->data[2]>7){
           memcpy(msg_send.data,"bad-pin",7); // sending an error message
           msg_send.len = 7;
           }
          else{unsigned char i=1;
           p=msg->data[1]!='A'?&PTP:&PORTA;
           i<<=msg->data[2];
           msg_send.data[0]=(*p&i)>>msg->data[2];
           msg_send.len = 1;
           }
          break;
    }
    case 2:{ // COMANDA 02: digital write (write at the corresponding por)
           // message length is less than 4-digit hexadecimal
          if(msg->len<4){
           memcpy(msg_send.data,"bad-len",7); // sending an error message
```

```
          msg_send.len = 7;
          }
          // writing, it's about port B
         else if(msg->data[1]!='B'){
          memcpy(msg_send.data,"bad-port",8); // sending an error message
          msg_send.len = 8;
          }
          // maximum 8 pins
         else if(msg->data[2]>7){
          memcpy(msg_send.data,"bad-pin",7); // sending an error message
          msg_send.len = 7;
          }
         else{unsigned char i=1;
          i<<=msg->data[2];
          PORTB =msg->data[3]?PORTB|i:PORTB&(~i);
         // memcpy(msg_send.data,"out:",4);
          msg_send.data[0]=PORTB;
          msg_send.len = 1;
          }
         break;
     }

// ---------------------------------------------------------------------------------------
// Conversion COMMAND
// ---------------------------------------------------------------------------------------

     case 3:{ // COMMAND 03: analogic read
     // message length is less than 2-digit hexadecimal
     if(msg->len<2){
          memcpy(msg_send.data,"bad-len",7); // sending an error message
          msg_send.len = 7;
          }
         else if(msg->data[1]>7){
          memcpy(msg_send.data,"bad-ch",6); // sending an error message
          msg_send.len = 6;
          }
         else{unsigned char i=1;
          i<<=msg->data[1];
          # if(ADC_8_BIT!=1)
            // reset SCF flag
            //ATD0CTL5 right justified; scan mode across channels
            // Reads the ADC channel
           while(!(ATD0STAT0 & 0x80));
           switch(msg->data[1]){
            //result is the result registers, 10-bit conversion
            case 0:{
                 msg_send.data[0] = ATD0DR0H&0x03;
                    msg_send.data[1] = ATD0DR0L;
              break;
             }
            case 1:{
                 msg_send.data[0] = ATD0DR1H&0x03;
                    msg_send.data[1] = ATD0DR1L;
              break;
             }
```

```
  case 2:{
      msg_send.data[0] = ATD0DR2H&0x03;
         msg_send.data[1] = ATD0DR2L;
    break;
   }
  case 3:{
      msg_send.data[0] = ATD0DR3H&0x03;
         msg_send.data[1] = ATD0DR3L;
    break;
   }
  case 4:{
      msg_send.data[0] = ATD0DR4H&0x03;
         msg_send.data[1] = ATD0DR4L;
    break;
   }
  case 5:{
      msg_send.data[0] = ATD0DR5H&0x03;
         msg_send.data[1] = ATD0DR5L;
    break;
   }
  case 6:{
      msg_send.data[0] = ATD0DR6H&0x03;
         msg_send.data[1] = ATD0DR6L;
    break;
   }
  default:{
      msg_send.data[0] = ATD0DR7H&0x03;
         msg_send.data[1] = ATD0DR7L;
   }
  }
     msg_send.len=2;
     ATD0STAT0 = 0x80;
#else
// reset SCF flag
//ATD0CTL5 left justified; scan mode across channels
// Reads the ADC channel
while(!(ATD0STAT0 & 0x80));
switch(msg->data[1]){
 //the result is in only one result register, 8-bit conversion
   case 0:{
      msg_send.data[0] = ATD0DR0H;
    break;
   }
  case 1:{
      msg_send.data[0] = ATD0DR1H;
    break;
   }
  case 2:{
      msg_send.data[0] = ATD0DR2H;
    break;
   }
  case 3:{
      msg_send.data[0] = ATD0DR3H;
    break;
   }
```

```
                case 4:{
                    msg_send.data[0] = ATD0DR4H;
                  break;
                  }
                case 5:{
                    msg_send.data[0] = ATD0DR5H;
                  break;
                  }
                case 6:{
                    msg_send.data[0] = ATD0DR6H;
                  break;
                  }
                default:{
                    msg_send.data[0] = ATD0DR7H;
                  }
                }
               ATD0STAT0 = 0x80;
                    msg_send.len=1;
            #endif
            }
            break;
      }

      default: { // sending error message
        memcpy(msg_send.data,"bad_cmd",7);
        msg_send.len = 7;
      }
     }
    }
    else{ // sending error message
        memcpy(msg_send.data,"bad_msg",7);
        msg_send.len = 7;
    }

    // prepare for next message
    SendCANMessage(msg->dev, msg_send);
    msg=msg_rdy;
   }
  }
}

//////////////////////////////////////////////
#pragma CODE_SEG __NEAR_SEG NON_BANKED
interrupt void CAN0RcvISR(void){
  ReceiveCANMessage(0);
}

#pragma CODE_SEG DEFAULT
 //////////////////////////////////////////////
#pragma CODE_SEG __NEAR_SEG NON_BANKED

interrupt void CAN4RcvISR(void){
  ReceiveCANMessage(1);
}
#pragma CODE_SEG DEFAULT
```

3. Configure the ADC0 module in function *PeriphInit(void)* from the work code (see above) based on documentation S12ATD10B8CV2-ADC.pdf.

4. Using the signal generator and oscilloscope generate a DC signal with an amplitude of 2.4 V. Make the following connections on the development board:

- AN0 connects to AN3;
- AN1 connects to GND;
- On AN2 bring the signal from functions generator previously measured (generator ground goes to GND on the board);
- AN4 connects to V_{DDA};
- PADD0 potentiometer is on a random position, different from extremes.

5. The commands used in PcanView are (see the code and comments presented above):

d. **Command 00:** involves the returning as message of the software version used.

e. **Command 03:** read command of the ADC channels. Has the structure:

i. Byte 0: command [03]

ii. Byte 1: channel [00 (channel AN0), 01 (channel AN1),...]

Exemple:

03 01 introduced in PcanView, as test message, if the programming was done correctly, should return in PcanView, as the test response 0 as the channel was connected to GND (test development phase!).

6. Send on the board the code on 8-bit configuration, run it and give the following commands on USB2CAN (with extended ID in 500kbaud configuration, see the settings in the previous chapter):

6.a ID: 001h MSG: 00h

6.b ID: 001h MSG: 03h 00h

6.c ID: 001h MSG: 03h 01h

6.d ID: 001h MSG: 03h 02h

6.e ID: 001h MSG: 03h 03h

6.f ID: 001h MSG: 03h 04h

7. Change the presented code(see the value of **ADC_8_BIT**) so as we can send on the board the code with configuration on 10 bit, run it and give the following commands on USB2CAN ((with extended ID in 500kbaud configuration, see the settings in the previous chapter):

7.a ID: 001h MSG: 03h 00h

7.b ID: 001h MSG: 03h 01h

7.c ID: 001h MSG: 03h 02h

7.d ID: 001h MSG: 03h 03h

7.e ID: 001h MSG: 03h 04h

3.4 The abstract of the work

- During an analog to digital conversion are needed two types of voltages: *low reference voltage* (V_{RL}) and *high reference voltage* (V_{RH}).
- It can be seen that there are control registers, status registers, test, input activation, register PORTAD, and 8 result registers for analog to digital converter of the microcontroller used.
- Testing a portion of the implemented code is based on the hexadecimal system messages.
- Messages are introduced in PcanView. Depending on the requirement, the result of running the code can be seen throughout the PcanView, as the response message, in the OUTPUTS section of the board or at terminals.

3.5 Commented References

1. Datasheet for the family HCS12: 9S12DP256BDGV2-userguide.pdf (Freescale, 2005) presents the different functions of the microcontroller terminals.
2. Datasheet for the analog to digital converter of HCS12 microcontroller: S12ATD10B8CV2-ADC.pdf (Freescale, ATD_10B8C, 2005) presents the description of this module.
3. Deepening the ADC of HCS12 microcontroller can be done by referring to sections 12.1 to 12.11 in (Huang, 2010).
4. Mandatory consultation of (Popa, 2011).

3.6 Worksheet

Write your name:……

Semigroup:……..

1. Specify the content of the following registers in the required configuration (Section 3.3.3):

8 conversions per acquisition sequence, continuous conversion

ATD0CTL3 = 0x ; // eight conversions per sequence

ATD0CTL4 = 0x ; // 8 bit resolution

ATD0CTL4 = 0x ; // 10 bit resolution

ATD0CTL2 = 0x ; // normal operation

ATD0CTL5 = 0x ; // left justified; scan mode across channels

ATD0CTL5 = 0x ; // right justified; scan mode across channels

2. Specify in which bits from which registers can be found the result of conversion on 8-bit, aligned to the left (left justified) for channel ADC0 AN2 :.....

.....

3. Specify in which bits from which registers can be found the result of conversion on 10-bit, aligned to the right (right justified) for channel ADC0 AN4 : .....

.....

4. Provide responses to commands received on USB2CAN in section (Section 3.3.6 and **Section 3.3.7**):

6.a ID: MSG:

6.b ID: MSG:

6.c ID: MSG:

6.d ID: MSG:

6.e ID: MSG:

6.f ID: MSG:

7.a ID: MSG:

7.b ID: MSG:

7.c ID: MSG:

7.d ID: MSG:

7.e ID: MSG: