C28x Analogue Digital Converter

Introduction

One of the most important peripheral units of an embedded controller is the Analogue to Digital Converter (ADC). This unit provides an important interface between the controller and the real world. Most physical signals such as temperature, humidity, pressure, current, speed and acceleration are analogue signals. Almost all of these can be represented as an electrical voltage between V_{min} and V_{max}, e.g. 0...3V, which is proportional to the original signal. The purpose of the ADC is to convert this analogue voltage in a digital number. The relationship between the analogue input voltage (V_{in}), the number of binary digits to represent the digital number (n) and the digital number (D) is given by:

$$V_{in} = \frac{D * (V_{REF+} - V_{REF-})}{2^n - 1} + V_{REF-}$$

V_{REF+} and V_{REF-} are reference voltages and are used to limit the analogue voltage range. Any input voltage beyond these reference voltages will deliver a saturated digital number. NOTE: Of course all voltages must stay inside the limits of the maximum ratings according to the data sheet.

In case of the C28x and especially with the eZdsp2812 and the Zwickau adapter board voltage V_{REF-} is set to 0V, V_{REF+} to +3.0V. The C28x internal ADC has a 12 Bit resolution (n =12) for the digital number D. This gives:

$$V_{in} = \frac{D * 3.0V}{4095}$$

Most applications require not only one analogue input signal to be converted into a digital value; their control loop usually needs several different sensor input signals. Therefore, the C28x is equipped with 16 dedicated input pins to measure analogue voltages. These 16 signals are multiplexed internally, that means they are processed sequentially. To do the conversion, the ADC has to make sure that during the conversion procedure there is no change of the analogue input voltage V_{in}. Otherwise the digital number would be totally wrong. An internal “sample and hold unit(s&h)” takes care of this. The C28x is equipped with two s&h-units, which can be used in parallel. This allows us to convert two input signals (e.g. two currents) at the same time.

But there’s more: The C28x ADC has an “auto-sequencer” capability of 16 stages. That means that the ADC can automatically continue with the conversion of the next input channel after the previous ones are finished. Thanks to this enhancement we do not have to fetch the digital results in the middle of a measurement sequence, one single interrupt service routine call at the end of the sequence will do it.

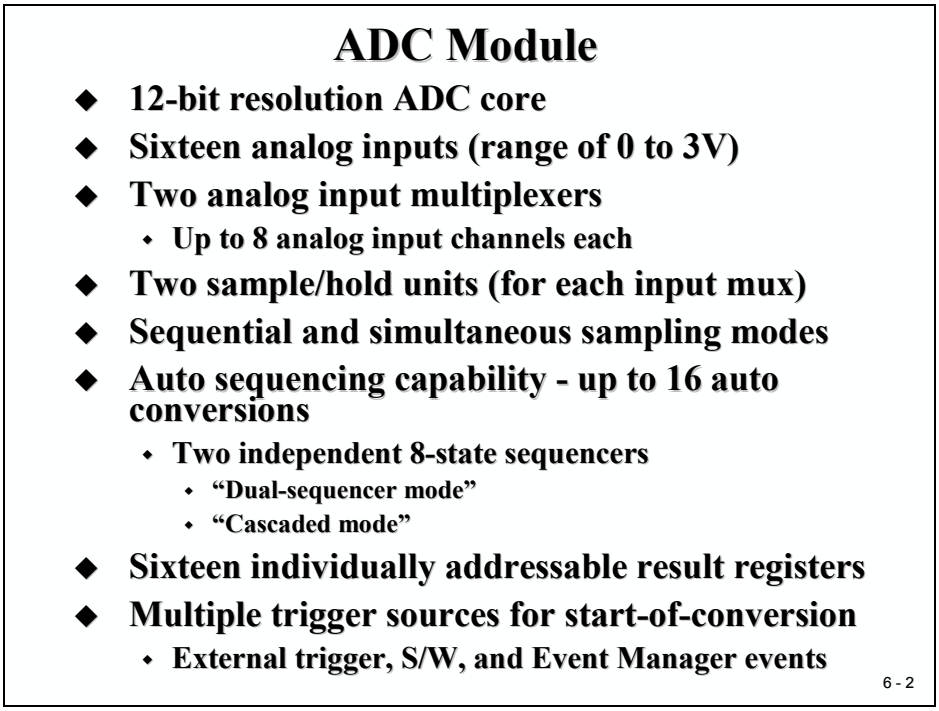
Module Topics

ADC Module Overview

Before we go into the details how to program the internal ADC let's summarize some details of the ADC Module. It was said that the digital resolution of the converted number is 12 bit. Assuming an input voltage range from 0...+3V we get a voltage resolution of 3.0V/4095 = 0.732mV per bit.

We have two s&h units, which can be used in parallel ("simultaneous sampling"). Each sample and hold is connected to 8 multiplexed input lines. The auto sequencer is a programmable state machine and is able to automatically convert up to 16 input signals. Each state of the auto sequencer puts a measurement into its own result register.

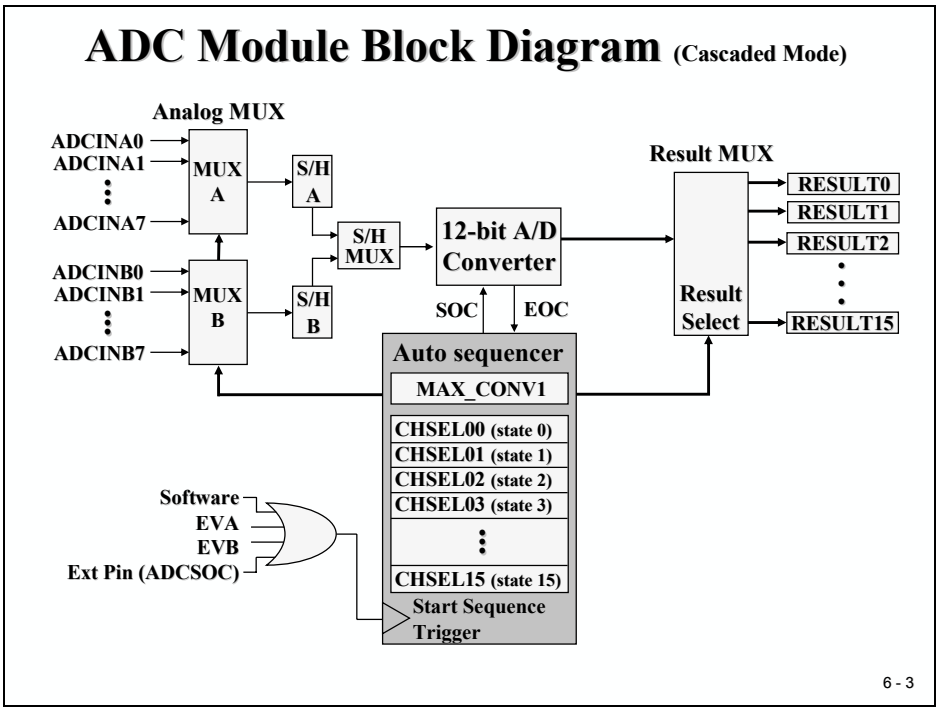
The fastest conversion time is 80ns per sample in a sequence and 160ns for the very first sample.



A start of a conversion sequence can be initiated from four sources:

- By software - just set a start bit to 1
- By an external signal "ADCSOC"
- By an event (period, compare, underflow) of Event Manager A
- By an event (period, compare, underflow) of Event Manager B

ADC in Cascaded Mode



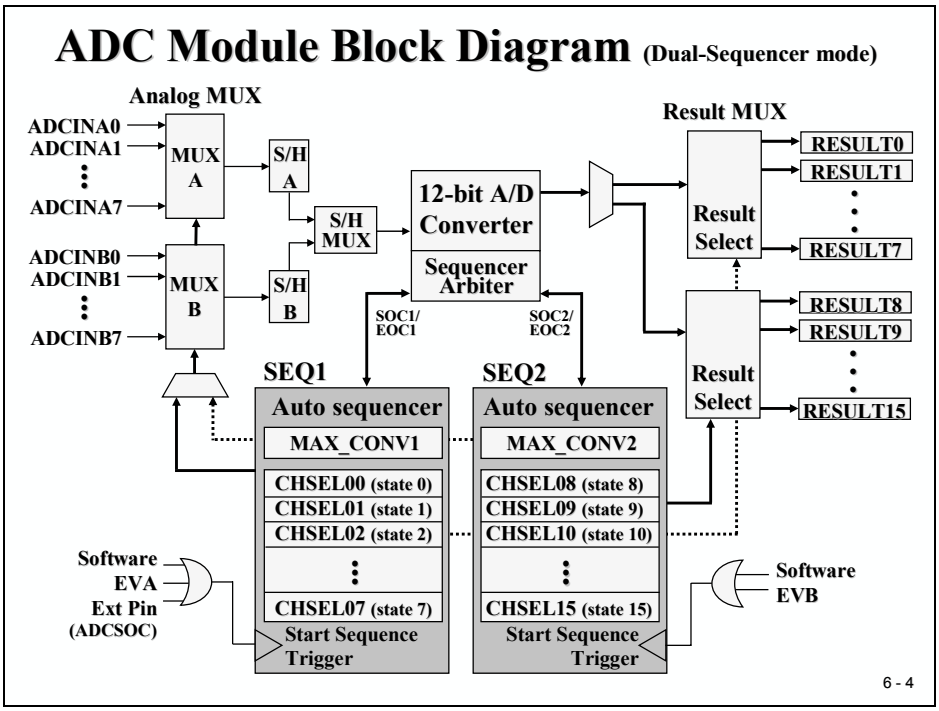
The slide shows the block diagram for the ADC operating in “cascaded mode”. One Auto sequencer controls the flow of the conversion. Before we can start a conversion, we have to setup the number of conversions (“MAX_CONV1”) and which input line should be converted in which stage (“CHSELxx”). The results are buffered in individual result registers (“RESULT0” to “RESULT15”) for every stage.

We can choose between two more options: “Simultaneous” and “Sequential” sampling. In the first case, both s&hs are used in parallel. Two input lines with the same input code (for example ADCINA3 and ADCINB3) are converted at the same time by stage CHSEL00. In “Sequential mode” the input lines can be connected to any of the states of the auto sequencer.

To trigger a conversion sequence we can use a software start by setting a particular bit. We also have three more start options using hardware events. Especially useful is the hard-wired output of a timer event, which leads to very precise sample periods. This is a necessity for correct operation of digital signal processing algorithms. No need to trigger an interrupt service (with its possible jitter due to interrupt response delays) to switch the input channel between subsequent conversions – the auto sequencer will do it.

We can use the ADC’s interrupt after the end of a sequence (or for some applications at the end of every other sequence) to read out the result register block.

ADC in Dual Sequencer Mode

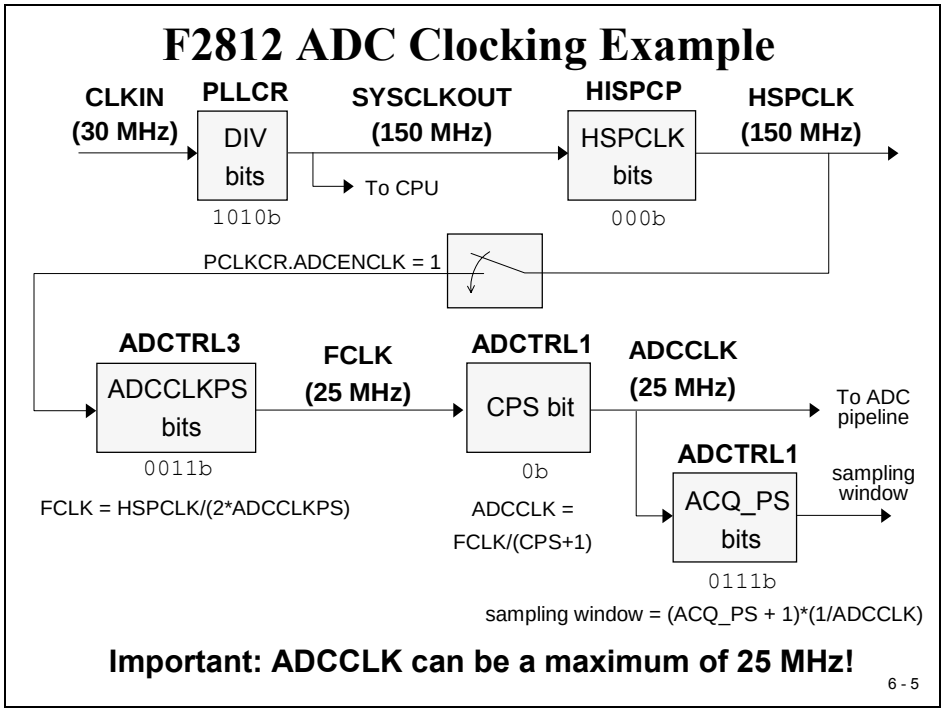


The second operating mode of the ADC "Dual Sequencer Mode" splits the auto sequencer into two independent state machines ("SEQ1" and "SEQ2"). This mode uses EVA as the hardware trigger for SEQ1 and EVB for SEQ2. To code the input channels for the individual states of the two sequencers we are free to select any of the 16 inputs for any of the 2x8 states. RESULT0 to RESULT7 cover the values from SEQ1 and RESULT8 to RESULT15 do it for SEQ2.

The reason for this split mode is to have two independent ADC's, triggered by their own hardware timer units, GP Timer 1 and 2 for SEQ1 and GP Timer 3 and 4 for SEQ2.

In case of a simultaneous start of SEQ1 and SEQ2 the Sequencer Arbiter takes care of this situation. In this event SEQ1 has higher priority; the start of SEQ2 will be delayed after the end of SEQ1.

ADC Conversion Time



There are some limitations for the set-up of the ADC conversion time. First, the basic clock source for the ADC is the internal clock HSPCLK – we cannot use any clock speed we like. This clock is derived from the external oscillator, multiplied by PLLCR and divided by HISPCP. We discussed these bit fields in earlier modules; so just in case you do not recall their meanings, look back.

The second limitation is the maximum frequency for "FCLK" as the internal input signal for the ADC unit. At the moment this signal is limited to 25MHz. To adjust this clock we have to initialise the bit field "ADCCLKPS" accordingly. Bit "CPS" gives the option for another divider by 2. The clock "ADCCLK" is the time base for the internal processing pipeline of the ADC.

A third limitation is the sampling window controlled by the field "ACQ_PS". This group of bits defines the length of the window that is used between the multiplexer switch and the time when we sample (or "freeze") the input voltage. This time depends on the line impedance of the input signal. So it is hardware dependent - we can't specify an optimal period for all applications. For our lab exercises in this chapter, it is a 'don't care' because we sample DC-voltages taken from two potentiometers of the Zwickau adapter board.

ADC Register Block

Three control registers “ADCTRL1 to 3” are used to set-up one of the various operating conditions of the ADC unit. Register “ADCST” covers the current status of the ADC.

Analog-to-Digital Converter Registers

Register	Address	Description
ADCTRL1	0x007100	ADC Control Register 1
ADCTRL2	0x007101	ADC Control Register 2
ADCMAXCONV	0x007102	ADC Maximum Conversion Channels Register
ADCCHSELSEQ1	0x007103	ADC Channel Select Sequencing Control Register 1
ADCCHSELSEQ2	0x007104	ADC Channel Select Sequencing Control Register 2
ADCCHSELSEQ3	0x007105	ADC Channel Select Sequencing Control Register 3
ADCCHSELSEQ4	0x007106	ADC Channel Select Sequencing Control Register 4
ADCASEQSR	0x007107	ADC Auto sequence Status Register
ADCRESULT0	0x007108	ADC Conversion Result Buffer Register 0
ADCRESULT1	0x007109	ADC Conversion Result Buffer Register 1
ADCRESULT2	0x00710A	ADC Conversion Result Buffer Register 2
: :	: :	: : : : : :
ADCRESULT14	0x007116	ADC Conversion Result Buffer Register 14
ADCRESULT15	0x007117	ADC Conversion Result Buffer Register 15
ADCTRL3	0x007118	ADC Control Register 3
ADCST	0x007119	ADC Status and Flag Register

6 - 6

ADC Control Register 1 - Upper Byte

ADCTRL1 @ 0x007100

ADC Module Reset
0 = no effect
1 = reset (set back to 0 by ADC logic)

Acquisition Time Prescale (S/H)
Value = (binary+1)
* Time dependent on the “Conversion Clock Prescale” bit (Bit 7 “CPS”)

15	14	13	12	11	10	9	8
reserved	RESET	SUSMOD1	SUSMOD0	ACQ_PS3	ACQ_PS2	ACQ_PS1	ACQ_PS0

Emulation Suspend Mode
00 = [Mode 0] free run (do not stop)
01 = [Mode 1] stop after current sequence
10 = [Mode 2] stop after current conversion
11 = [Mode 3] stop immediately

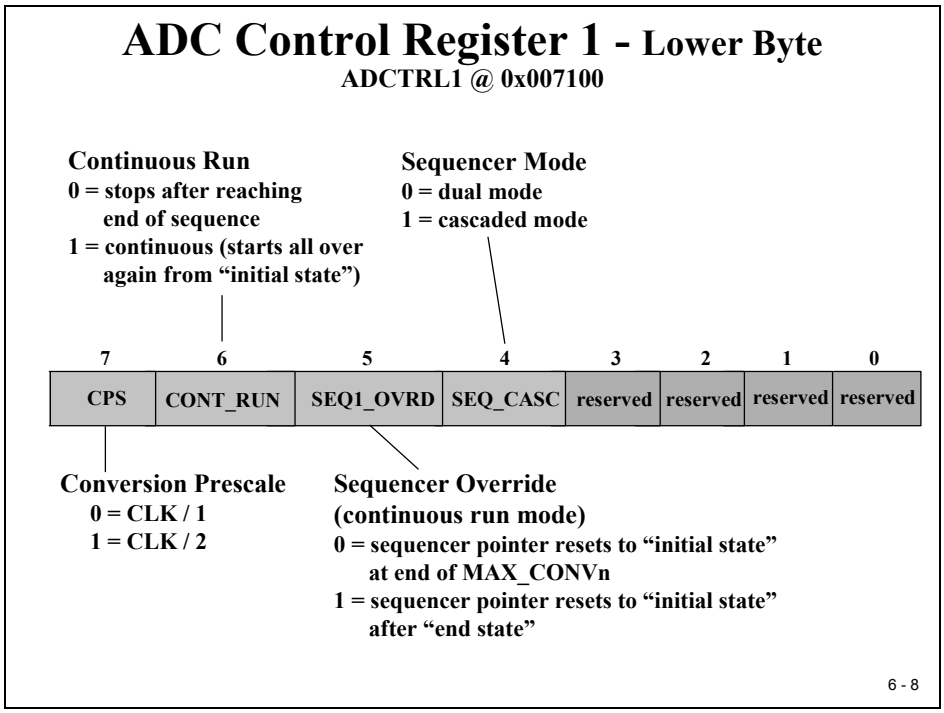
6 - 7

ADC Control Register 1

Bit 14 (“RESET”) can be used to reset the whole ADC unit into its initial state. It is always good practice to apply a RESET command before you initialise the ADC.

Bits 13 and 12 define the interaction between the ADC and an emulator command, similar to the behaviour that we discussed in the event manager module.

The next 4 bits define the length of the sample window.



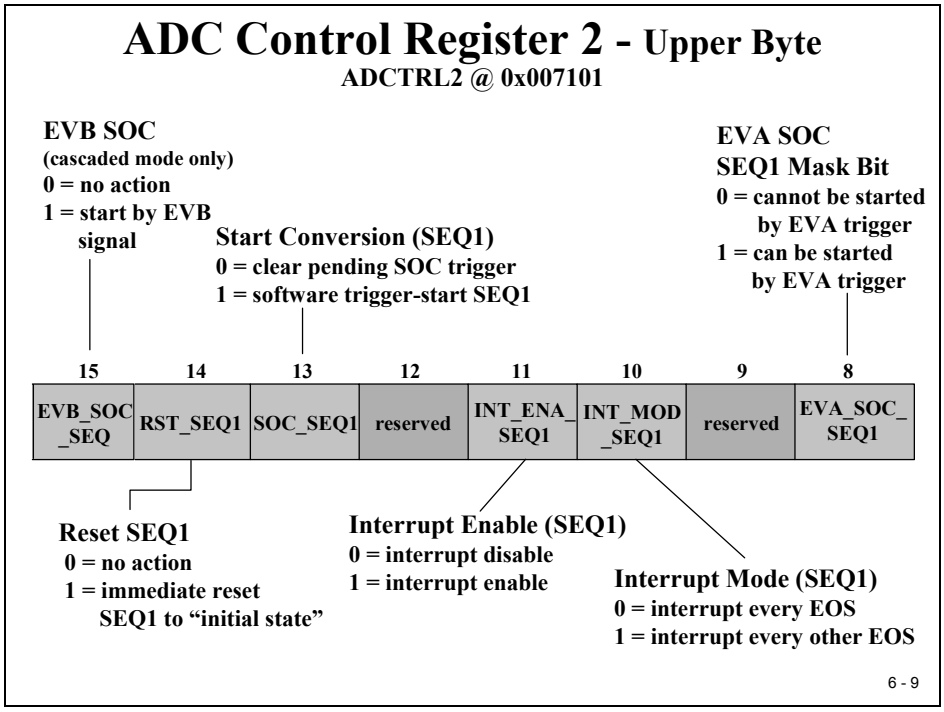
“CPS” is used to divide the input frequency by 1 or 2.

Bit 6 (“CONT_RUN”) defines if the auto sequencer starts at the end of a sequence (=0) and waits for another trigger or if the sequence should start all over again immediately (= 1).

Bit 5(“SEQ1_OVRD”) defines two different options for continuous mode. We will not use this mode during our labs, so it is a ‘don’t care’.

Finally Bit 4 defines the Sequencer Mode to be a state machine of 16 (=1) or to operate as two independent state machines of 8 states.

ADC Control Register 2



The upper half of register ADCTRL2 is responsible to control the operating mode of sequencer 1.

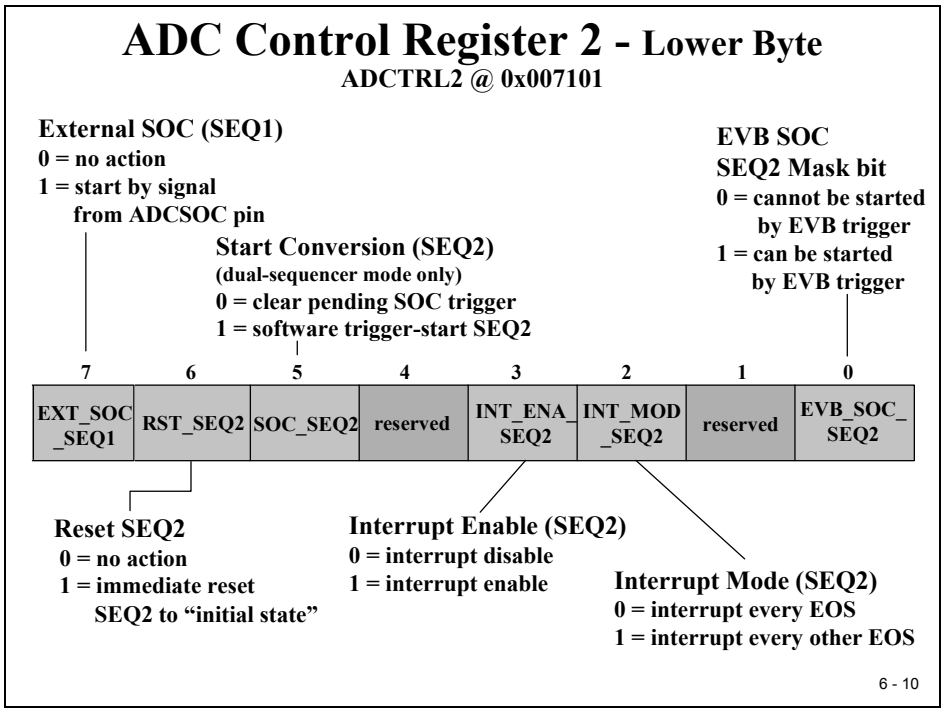
Bit 15 "EVB_SOC_SEQ" flags if Event Manager B has triggered the conversion. It is a 'read only' flag.

With Bit14 "RST_SEQ1" we can reset the state machine of SEQ1 to its initial state. That means that the next trigger will start again from CHSELSEQ1.

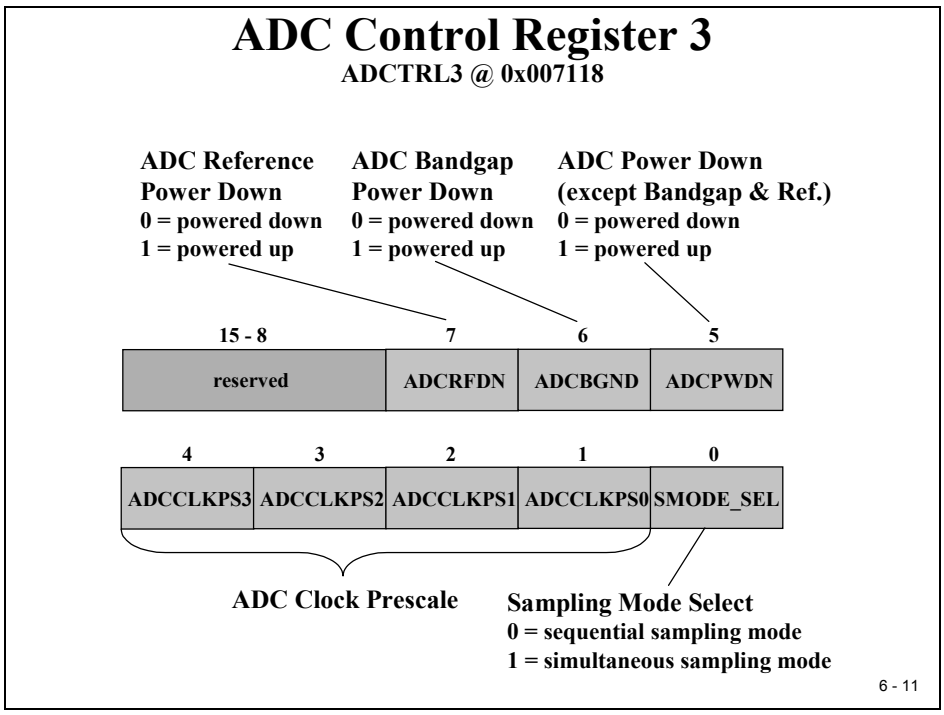
When we set Bit 13 "SOC_SEQ1" to 1 we perform a software start of the conversion.

Bits 11 and 10 define the interrupt mode of SEQ1. We can specify to whether we have an interrupt request for every "End of Sequence" (EOS) or every other (EOS).

Bit 8 "EVA_SOC_SEQ1" is the mask bit to enable or disable Event Manager A's ability to trigger a conversion. In Lab6 we will make use of this start feature, so please remember to enable this start option during the procedure of Lab6!



The lower byte of ADCTRL2 is similar to its upper half: it controls sequencer SEQ2. Bit 7 flags the event that the external pin "ADCSOC" has caused the conversion. The rest is identical to the upper half.



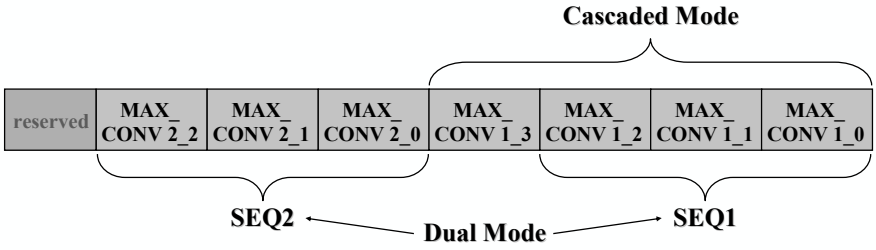
ADC MAXCONV Register

”MAXCONV” defines the number of states per trigger.

Maximum Conversion Channels Register

ADCMAXCONV @ 0x007102

- **Bit fields define the maximum number of auto conversions (binary+1)**



- **Auto conversion session always starts with the “initial state” and continues sequentially until the “end state”, if allowed**

	SEQ1	SEQ2	Cascaded
Initial state	CONV00	CONV08	CONV00
End state	CONV07	CONV15	CONV15

6 - 12

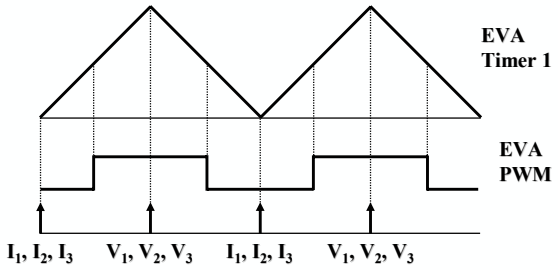
ADC Input Channel Select Sequencing Control Register

	Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0	
0x007103	CONV03	CONV02	CONV01	CONV00	ADCCHSELSEQ1
0x007104	CONV07	CONV06	CONV05	CONV04	ADCCHSELSEQ2
0x007105	CONV11	CONV10	CONV09	CONV08	ADCCHSELSEQ3
0x007106	CONV15	CONV14	CONV13	CONV12	ADCCHSELSEQ4

6 - 13

Example: 3 phase measurement

Example - Sequencer "Start/Stop" Operation



System Requirements:
- **Three auto conversions (I_1, I_2, I_3) off trigger 1 (Timer underflow)**
- **Three auto conversions (V_1, V_2, V_3) off trigger 2 (Timer period)**

Event Manager A (EVA) and SEQ1 are used for this example with sequential sampling mode

6 - 14

The two slides give a typical example of a 3-phase control system for digital motor control.

Example - Sequencer "Start/Stop" Operation
(Continued)

- **MAX_CONV1 is set to 2 and Channel Select Sequencing Control Registers are set to:**

Bits →	15-12	11-8	7-4	3-0	
0x007103	V_1	I_3	I_2	I_1	ADCCHSELSEQ1
0x007104	x	x	V_3	V_2	ADCCHSELSEQ2

- **Once reset and initialized, SEQ1 waits for a trigger**
- **First trigger three conversions performed: CONV00 (I_1), CONV01 (I_2), CONV02 (I_3)**
- **MAX_CONV1 value is reset to 2 (unless changed by software)**
- **SEQ1 waits for second trigger**
- **Second trigger three conversions performed: CONV03 (V_1), CONV04 (V_2), CONV05 (V_3)**
- **End of second auto conversion session, ADC Results registers have the following values:**

RESULT0	I_1	RESULT3	V_1
RESULT1	I_2	RESULT4	V_2
RESULT2	I_3	RESULT5	V_3

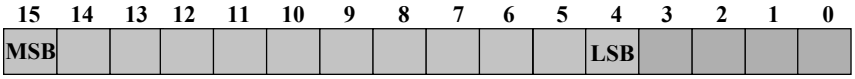
→ **User can reset SEQ1 by software to state CONV00 and repeat same trigger 1, 2 session**
- **SEQ1 keeps "waiting" at current state for another trigger**

6 - 15

ADC Result Register Set

ADC Conversion Result Buffer Register

ADCRESULT0 @ 0x007108 through ADCRESULT15 @ 0x007117
(Total of 16 Registers)



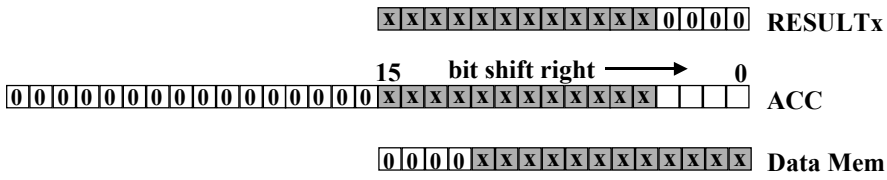
With analog input 0V to 3V, we have:

analog volts	converted value	RESULTx
3.0	FFFh	1111|1111|1111|0000
1.5	7FFh	0111|1111|1111|0000
0.00073	1h	0000|0000|0001|0000
0	0h	0000|0000|0000|0000

6 - 16

The 12 Bit digital results are left justified. Why is that? Answer: The 'native' numbering scheme of the C28x, which is fractional. We'll see how this makes scaling easier in Part II of this course.

How do we Read the Result? Integer format



Example: read RESULT0 register

```
#include "DSP281x_Device.h"

void main(void)
{
   Uint16 value;           // unsigned

   value = AdcRegs.ADCRESULT0 >> 4;
}
```

6 - 17

Lab 6: Two Potentiometer Voltages

Lab 6: Two Channel Analogue Conversion initiated by GP Timer 1

AIM :

- **AD-Conversion of ADCIN_A0 and ADCIN_B0 initiated by GPT1-period of 0.1 sec.**
- **ADCIN_A0 and ADCIN_B0 are connected to two potentiometers to control analogue input voltages between 0 and 3,0V.**
- **no GPT1-interrupt-service ➔ Auto-start of ADC with T1TOADC-bit !!**
- **Use ADC-Interrupt Service Routine to read out the ADC results**
- **Use main loop to show alternately the two results as light-beam on LED's (GPIO port B7..B0)**

6 - 18

Additional Registers to initialize Lab 6:

General Purpose Timer Control :	:	GPTCONA
Timer 1 Control	:	T1CON
Timer 1 Period	:	T1PR
Timer 1 Compare	:	T1CMPR
Timer 1 Counter	:	T1CNT
Interrupt Flag	:	IFR
Interrupt Enable ask	:	IER
ADC – Control 3	:	ADCTRL3
ADC – Control 2	:	ADCTRL2
ADC – Control 1	:	ADCTRL1
Channel Select Sequencer 1	:	CHSELSEQ1
Max. number of conversions	:	MAXCONV
ADC - Result 0	:	ADCRESULT0
ADC - Result 1	:	ADCRESULT1

6 - 19

Objective

The objective of this exercise is to practice using the integrated Analogue-Digital Converter of the C28x. The Zwickau Adapter board is equipped with 2 potentiometers at ADCIN_A0 and ADCIN_B0. The two voltages can be changed between 0 and 3.0Volt. The goal of this lab is to read the current status of the potentiometers and to show the voltages as 'Light-Beam' on 8 LED's (GPIO Port B0...B7).

GP Timer 1 generates the sample period of 100msec. The conversion is triggered automatically by a GP Timer 1 period event. The ADC interrupt service routine is the only interrupt needed in this example.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab6.pjt** in E:\C281x\Labs.
2. Open the file Lab5A.c from E:\C281x\Labs\Lab5A and save it as Lab6.c in E:\C281x\Labs\Lab6.
3. Add the source code file to your project:
   - **Lab6.c**
4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:
   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**
   - **DSP281x_Adc.c**
   - **DSP281x_usDelay.asm**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

5. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
   ..\include

6. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Modify Source Code

7. Open Lab6.c to edit: double click on "Lab6.c" inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the IQ-Math Library for this exercise:

   At the beginning of the code, delete the lines:

   #include "IQmathLib.h"
   #pragma DATA_SECTION(sine_table,"IQmathTables");
   _iq30 sine_table[512];

8. We do not need the interrupt service routine "T1_Compare_isr()"; instead, we need a new one for the ADC, called "adc_isr()".

   Change the prototype declaration for the interrupt service routine into:

   interrupt void adc_isr(void);

9. Next, just after the function prototype section include the definition of two global integer variables, called "Voltage_A0" and "Voltage_B0". The two variables will be used to pass the current measurement values from the ADC interrupt service routine to the main loop. Add:

   int Voltage_A0;

   int Voltage_B0;

10. Inside “main”, after the function call “InitPieVectTable();” add the following line to call the basic ADC initialization:

    InitAdc();

11. Change the three lines that re-map the PIE – entry and that enable the ADC interrupt into:

    PieVectTable.ADCINT = &adc_isr;

    PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

    IER = 1;

12. Although we initialized the ADC in step 10, we still have to configure its operating mode. Place the corresponding lines after the global interrupt enable instruction “ERTM”. Take into account the following setup:

    Dual Sequencer Mode:

    AdcRegs.ADCTRL1.bit.SEQ_CASC = ?

    No continuous run:

    AdcRegs.ADCTRL1.bit.CONT_RUN = ?

    Conversion Prescale = CLK/1:

    AdcRegs.ADCTRL1.bit.CPS = ?

    2 conversions (ADCIN0 and ADCINB0) out of a GP Timer 1 start:

    AdcRegs.ADCMAXCONV.all = ?

    Setup the channel sequencer to ADCIN0 and ADCINB0:

    AdcRegs.ADCSELSEQ1.bit.CONV00 = ?

    AdcRegs.ADCSELSEQ1.bit.CONV01 = ?

    Enable the Event Manager A to start the conversion:

    AdcRegs.ADCTRL2.bit.EVA_SOC_SEQ1 = ?

    Enable the ADC interrupt with every end of sequence:

    AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = ?

    We also have to initialize the speed of the ADC (see also slide 6-5). In function “InitSystem()” the high-speed clock prescaler HISPCP is set to ‘divide by 2’. Assuming a SYSCLOCKOUT of 150MHz we have an ADC input clock of 75MHz.

For 'FCLK' the maximum frequency is 25MHz, so we have to setup the ADC clock prescaler "ADCCLKPS" to '0010' ➔ FCLK = 18.75 MHz.

AdcRegs.ADCTRL3.bit.ADCCLKPS = 2;

13. We also have to adjust the Event Manager A configuration. We do not need to drive any output signal from the GP Timer 1. For Register "GPTCONA" we can disable the two outputs and set the polarity to "forced low":

    EvaRegs.GPTCONA.bit.TCMPOE = 0;

    EvaRegs.GPTCONA.bit.T1PIN = 0;

    To enable the auto start of the ADC with every timer period we have to enable this feature:

    EvaRegs.GPTCONA.bit.T1TOADC = 2;

14. Modify the setup for GP Timer 1(Register T1CON)! Take into account to setup:

    - "Continuous up count mode",
    - "Internal clock source",
    - "Stop on emulation suspend" and
    - "Disable Compare Operation".

15. Setup the GP Timer 1 Period:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

    To setup the timer for a sample period of 100 ms we calculate:

    - f_{CPU} = 150 MHz,
    - HSPCP = 2 and
    - f_{PWM} = 10Hz
    - TPS_{T1} = 128
    - T1PR = 58594

    EvaRegs.T1CON.bit.TPS = 7;

    EvaRegs.T1PR = 58594;

16. Delete the line "EvaRegs.EVAIMRA.bit.T1CINT = 1;" – we do not need a Timer Interrupt for this lab.

17. Inside function “Gpio_select()” do not enable T1PWM as pin function.

18. Delete the whole interrupt function “T1_Compare_isr()”.

19. Add a new interrupt service routine “adc_isr()” to your code. Inside this function, do:

    - Service the watchdog, part 1:

      ```
      EALLOW;
      SysCtrlRegs.WDKEY = 0x55;
      EDIS;
      ```

    - Read the two ADC result register and load the value into variables “Voltage_A0” and “Voltage_B0”:

      ```
      Voltage_A0 = AdcRegs.ADCRESULT0 >> 4;
      Voltage_B0 = AdcRegs.ADCRESULT1 >> 4;
      ```

    - Reset ADC Sequencer1 (Register ADCCTRL2):

      ```
      AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;
      ```

    - Clear Interrupt Flag ADC Sequencer 1 (Register ADCST)

      ```
      AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;
      ```

    - Acknowledge PIE Interrupt:

      ```
      PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
      ```

20. Inside function “InitSystem()” enable the clock system for the ADC:

    ```
    SysCtrlRegs.PCLKCR.bit.ADCENCLK = 1;
    ```

Build and Load

21. Click the “Rebuild All” button or perform:

    Project → Build

    and watch the tools run in the build window. If you get syntax errors or warnings debug as necessary.

22. Load the output file down to the DSP Click:

    File → Load Program and choose the desired output file.

Test

23. Reset the DSP by clicking on:

    Debug → Reset CPU followed by
    Debug → Restart and
    Debug → Go main.

24. Set a breakpoint into interrupt service routine "adc_isr()" at the last line of its code.

Run

25. When you've modified your code correctly and you execute a real time run, this breakpoint should be hit periodically. If not, you missed one or more steps in your procedure for this lab exercise. In this case try to review your modifications. If you do not spot a mistake immediately try to test systematically:

    - A good start is to temporarily disable the watchdog timer
    - Verify that GP Timer1 is counting (T1CNT)
    - Verify that the clock system is enabled (PCLKCR) for EVA and ADC
    - Inspect the Interrupt Registers (IER, PIEIER, INTM)
    - Inspect the ADC Register Set (ADCTRL1-3)

    If nothing helps, ask your instructor for advice. Please do not ask questions like "It is not working" or "I do not know what's wrong..." Instead, summarize your test strategy and show intermediate results for inspection.

26. After you verified that the interrupt service routine "adc_isr()" is called periodically, check the ADC results. Add variables "Voltage_A0" and "Voltage_B0" to your watch window. With the breakpoint still set, modify the analogue input voltages with the two potentiometers "R1" and "R2" of the Zwickau Adapter board. You should be able to get values between 0 and 4095 for the leftmost and rightmost positions of R1 and R2.

Add the display code (LED beam)

27. So far we verified that the GP Timer 1 every 100 ms triggers the ADC to convert the two input voltages periodically. Now we need to add a next portion of code to display the current status of Voltage_A0 and Voltage_B0. To display it, we have to use the 8 LEDs at GPIO – B0 to B7.

    A good point to add this code is the while(1) loop of main. After we served the watchdog we can easily add our display code. Question is: how do we display two values with 8 LED's only?

One option could be to alternate every 2 seconds from display “Voltage_A0” to “Voltage_B0”. Recall, the digital number is in the range 0 to 4095. The rule is simple: the bigger the number the more LED’s should be switched on.

To generate the 2 seconds alternation you can use a simple loop counter in main, or you could use GP Timer 1, now enabled for period interrupt, and count the number of 100ms periods up to 20 before you alternate the display value.

Try to finish this portion of Lab6 by yourself!

END of LAB 6

Optional Lab6A

Modify Lab-Exercise 4 (‘Knight-Rider’) :

- **use the Analogue Input ADCIN0 to change the frequency for the LED’s**
- **to add the ADC-setup use Lab6 as a start**
- **use a LED-frequency range between 50Hz and 1 Hz**
- **use (1) a linear or (2) a logarithm scale between F_{min} and F_{max}.**

6 - 20

Lab 6A: Speed Control of 'Knight Rider'

Objective

Now that we have exercised both with the ADC (Lab 6) and the CPU hardware Timer 0 (LAB 4) we can combine the two exercises. The objective is to control the speed step of the LED's sequence of Lab4 ("Knight Rider") with the ADC-Input ADCIN_A0 ➔ the higher the voltage ADCIN_A0 the higher the speed of the LED-sequence.

We will need two interrupts along with the main function, interrupt 1 for the ADC results and interrupt 2 for core CPU Timer 0.

Use your code from Lab4 and Lab6 as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab6A.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab6A.c in E:\C281x\Labs\Lab6A.
3. Add the source code file to your project:
   - **Lab6A.c**
4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:
   - **DSP281x_PieCtrl.c**
   - **DSP281x_PieVect.c**
   - **DSP281x_DefaultIsr.c**

- **DSP281x_Adc.c**
- **DSP281x_CpuTimers.c**
- **DSP281x_usDelay.asm**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

7. Setup the search path to include the peripheral register header files. Click:

   Project → Build Options

   Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

   C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;
   ..\include

8. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

   400

   Close the Build Options Menu by Clicking <**OK**>.

Modify Source Code

9. Open Lab6A.c to edit: double click on "Lab6A.c" inside the project window. Also open the file "Lab6.c" and copy the portions of code that are necessary to use the ADCIN_A0 input voltage to control the period of CPU core Timer 0. In detail:

   At begin of "Lab6A.c" add the prototype declaration for the interrupt service routine of the ADC:

   ```
   interrupt void adc_isr(void);
   ```

10. Add directly after the prototype section a global integer variable "Voltage_A0":

    ```
    int Voltage_A0;
    ```

11. Next, in function "main" after the line which calls "InitPieVectTable()" add the function call to initialize the ADC:

    ```
    InitAdc();
    ```

12. In function "main", after the re-map instruction for PieVectTable.TINT0 add the second re-map instruction:

    PieVectTable.ADCINT = &adc_isr;

13. Inside "main", after PIEIER1.bit.INTx7 is enabled, enable also INTx6 for the ADC-interrupt:

    PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

14. Before the line "CpuTimer0Regs.TCR.bit.TSS = 0" add the code from Lab6.c to configure the ADC. Take into account some minor modifications: Only 1 conversion (ADCIN_A0) needed, Channel Select Sequencer.CONV00 to ADCIN_A0.

15. Also before the line "CpuTimer0Regs.TCR.bit.TSS = 0" add another part from "Lab6.c" to initialize the GP Timer 1 (GPTCONA, T1CON, T1PR). No GP Timer 1 period interrupt is needed now. In case you have included this interrupt service in your last part of Lab6, delete these interrupt enable lines.

14. In function "InitSystem()" enable the clock distribution for EVA and ADC:

    SysCtrlRegs.PCLKCR.bit.EVAENCLK = 1;

    SysCtrlRegs.PCLKCR.bit.ADCENCLK = 1;

15. At the end of the code "Lab6A.c" add the interrupt service routine for the ADC "adc_isr" from Lab6. Delete the line: "Voltage_B0 = AdcRegs.ADCRESULT1 >>4;"

Build and Load

16. Click the "Rebuild All" button or perform:

    Project → Build

    and watch the tools run in the build window. If you get syntax errors or warnings debug as necessary.

17. Load the output file down to the DSP Click:

    File → Load Program and choose the desired output file.

Test

18. Reset the DSP by clicking on:

    Debug → Reset CPU followed by
    Debug → Restart and
    Debug → Go main.

Run

19. Run the code. The LED's should do the "Knight Rider".

20. So far we have reached the same result as in Lab4, a 200msec (50ms *4) period between the steps of the LED-sequence. But additionally now we have an active ADC in the background!

21. Open a watch window to watch variable "Voltage_A0". Click right mouse inside the watch window and select "Refresh". When you modify the potentiometer you should be able to see values between 0 and 4095, assuming you repeat the "Refresh" mouse click.

Modify the main loop

22. All we have to do now is to use variable "Voltage_A0" to control delay line that we so far used in the main loop:

    ```
    while(CpuTimer0.InterruptCount < 3);
    ```

    Now we can replace the constant '3' by a variable that is modified by "Voltage_A0". It is also recommended to include the watchdog service into this while-loop:

    ```
    while(CpuTimer0.InterruptCount < x)
    {
            EALLOW;
            SysCtrlRegs.WDKEY = 0xAA;
            EDIS;
    }
    ```

    All you have to do now is to find a useful construction to calculate an expression for x out of Voltage_A0. Hint: InterruptCount is a multiple of 50µs, so let's try to limit the value for x between 1 (= 50µs period) and 20(=1 sec period).

END of LAB 6A

This page was intentionally left blank.