F2833x 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 interface between the controller and the real world. Most physical signals such as temperature, humidity, pressure, current, speed and acceleration are analogue signals. With the aid of the appropriate transducer, 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 to 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:

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

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 produce a saturated digital number. NOTE: Of course, all voltages must remain within the limits of their maximum ratings, as specified in the data sheet.

In the case of the F2833x, the voltage V_{REF-} is fixed at 0V and V_{REF+} is connected to +3.0V. The F2833x internal ADC has a 12-bit resolution (n =12) for the digital number D. This gives a simplified equation:

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

Most applications require not only one analogue input signal to be converted into a digital value, but their control loop usually needs several different sensor input signals. Therefore, the F2833x is equipped with 16 dedicated input pins to measure analogue voltages. These 16 signals are multiplexed internally, which means they are processed sequentially. To perform a conversion, the ADC has to ensure that during the conversion procedure there is no change of the analogue input voltage V_{in}, otherwise the digital number would be erroneous. An internal “sample and hold unit (s&h)” takes care of this. The F2833x 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 of a 3-phase system) at the same time.

In addition, the F2833x ADC has an “auto-sequencer” capability of 16 stages. This means that the ADC can automatically continue with the conversion of the next input channels after the previous channels are completed. Thanks to this enhancement, we do not have to fetch the digital results in the middle of a measurement sequence, the task being carried out by a single interrupt service routine at the end of the sequence.

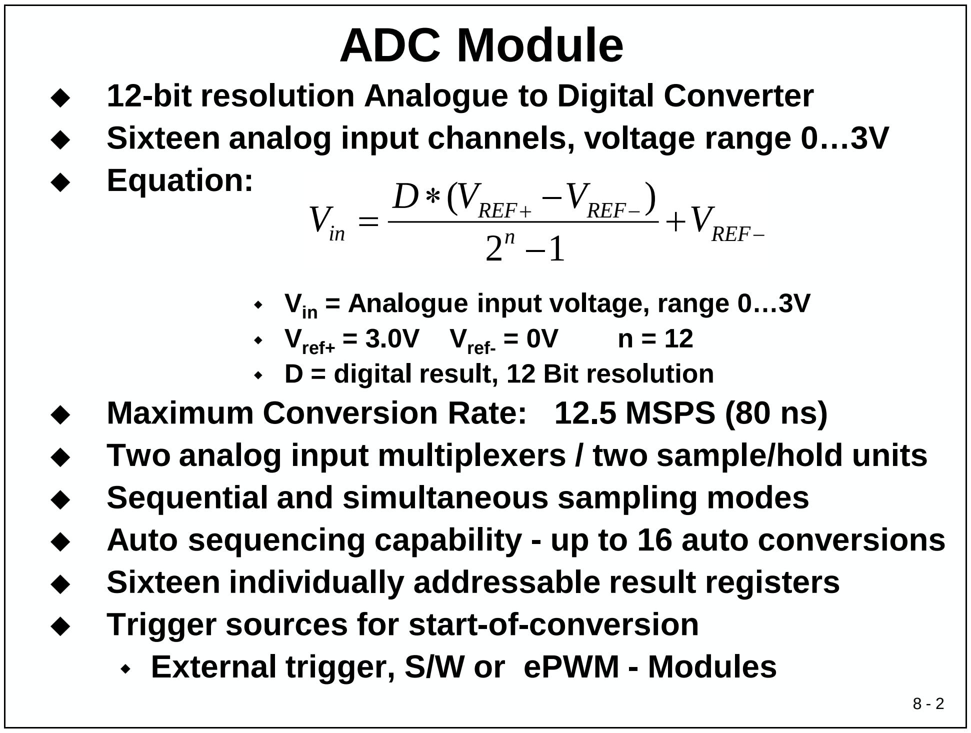
Module Topics

ADC Module Overview

Before we go into the details of how to program the internals of the ADC, let us first summarize some of the features of the ADC Module. It was stated earlier that the digital resolution of the converted number is 12 bits. Assuming an input voltage range from 0...+3V, we obtain a voltage resolution of 3.0V/4095 = 0.732mV per bit.

We have two sample and hold units, which can be used in parallel; the corresponding operating mode is called “simultaneous sampling”. Each sample and hold unit is connected to 8 multiplexed input lines. There is also an auto sequencer, which is a programmable state machine that is capable of automatically converting up to 16 input signals. Each state of the auto sequencer stores a measurement in its own dedicated result register.

The fastest conversion time is 80ns per sample in a sequence and 160ns for the very first sample. Of course we will have to adapt this conversion rate to the signal system that is actually used.



ADC Module

- 12-bit resolution Analogue to Digital Converter
- Sixteen analog input channels, voltage range 0…3V
- Equation:

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

  - V_{in} = Analogue input voltage, range 0…3V
  - V_{ref+} = 3.0V V_{ref-} = 0V   n = 12
  - D = digital result, 12 Bit resolution
- Maximum Conversion Rate: 12.5 MSPS (80 ns)
- Two analog input multiplexers / two sample/hold units
- Sequential and simultaneous sampling modes
- Auto sequencing capability - up to 16 auto conversions
- Sixteen individually addressable result registers
- Trigger sources for start-of-conversion
  - External trigger, S/W or ePWM - Modules

8 - 2

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

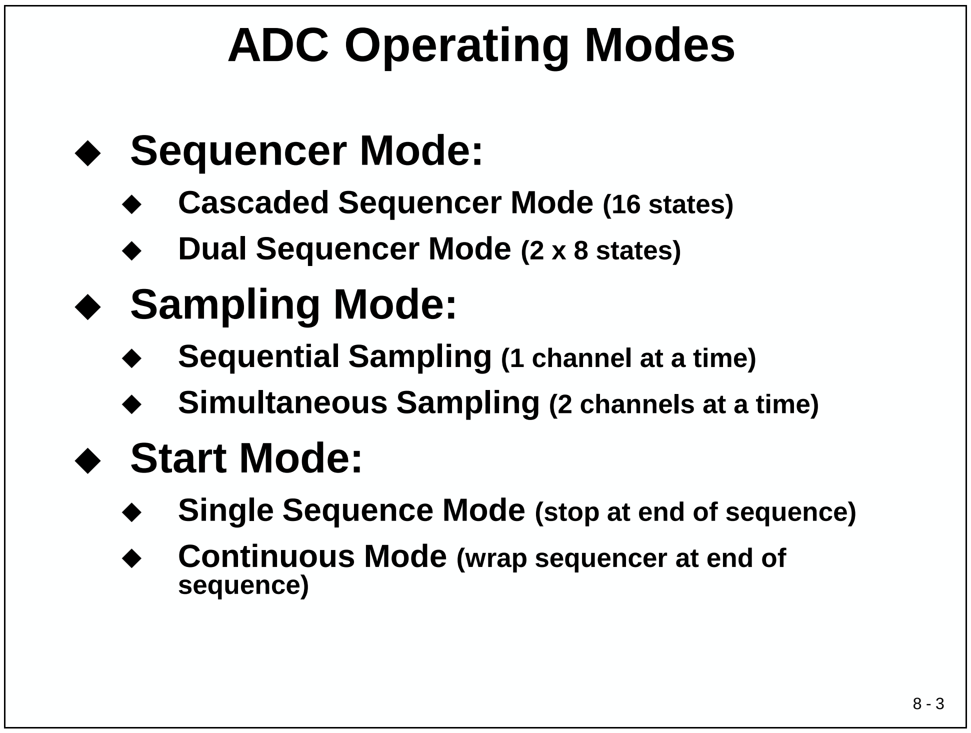
- By software - just set a start bit to 1
- By an external signal on pin “GPIO/XINT2_ADCSOC”
- By an event (period, compare or underflow) of one of the PWM-units ePWM1 to ePWM6

ADC operating modes

The ADC module can operate in different setups. An operating mode is always a combination of the three different basic selections:

- Sequencer Mode
- Sampling Mode
- Start Mode

Not all of the 8 possible combinations do actually make sense, so be careful what you select. The Sequencer Mode selects whether we use state machine of the Auto sequencer as a single 16 stage state machine ("Cascaded Mode") or as a pair of two independent 8-stage measurement units ("Dual Sequencer Mode"). By selecting "Simultaneous Sampling" for the sampling mode we convert 2 analogue input signals at one time. If we choose "Sequential Sampling" only one multiplexed input channel is converted at one time. Finally by selecting "Single Sequence Mode" (or "Start/Halt - Mode") the Auto sequencer starts at the first input trigger signal, performs the predefined number of conversions and stops at the end of this conversion sequence - then to wait for a second trigger. In continuous mode the Auto sequencer starts all over again at the end of the first conversion sequence without waiting for another trigger input signal.



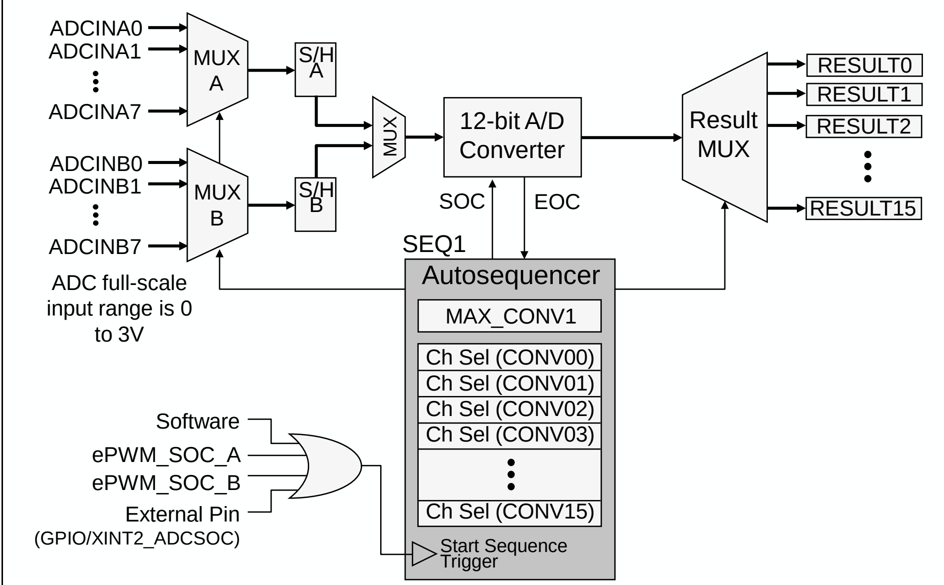
ADC Operating Modes

- ◆ **Sequencer Mode:**
  - ◆ **Cascaded Sequencer Mode (16 states)**
  - ◆ **Dual Sequencer Mode (2 x 8 states)**
- ◆ **Sampling Mode:**
  - ◆ **Sequential Sampling (1 channel at a time)**
  - ◆ **Simultaneous Sampling (2 channels at a time)**
- ◆ **Start Mode:**
  - ◆ **Single Sequence Mode (stop at end of sequence)**
  - ◆ **Continuous Mode (wrap sequencer at end of sequence)**

8 - 3

ADC in Cascaded Mode

ADC Sequencer in Cascaded Mode



8 - 4

The slide above (Slide 8-4) 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 each stage.

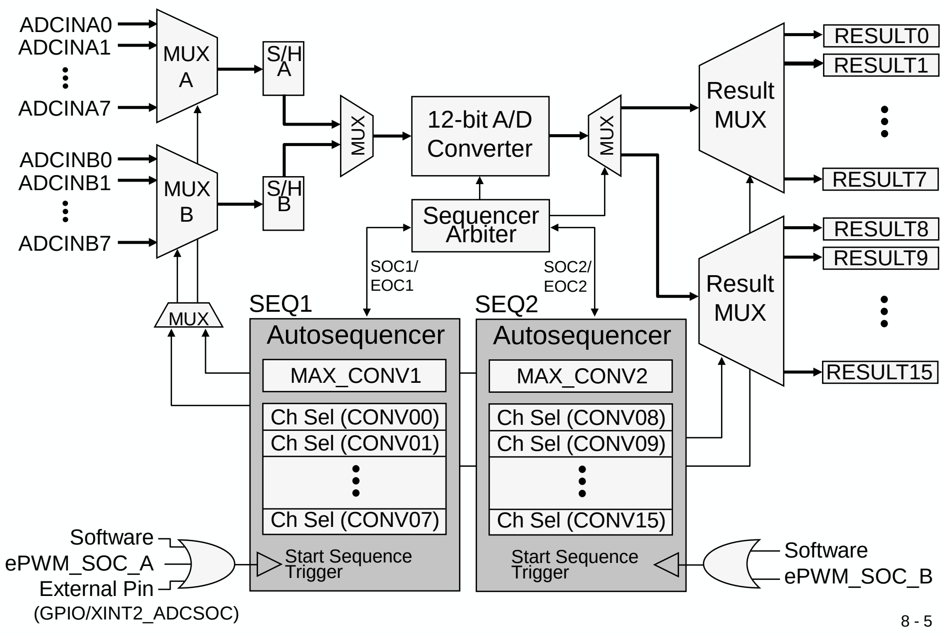
We can choose between two more options: “Simultaneous” and “Sequential” sampling. In the case of simultaneous sampling, both sample and hold units are used in parallel. Two input lines with the same input code (for example ADCINA3 and ADCINB3) are converted at the same time by stage CONV00. In “Sequential 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 an ePWM event, which leads to very precise sample periods. This is a necessity for correct operation of digital signal processing algorithms. There is no need to trigger an interrupt service (with its possible jitter due to interrupt response delays) to switch the input channel between subsequent conversions because the auto-sequencer will do that.

We can use the ADC 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

ADC Sequencer in Dual - Sequencer Mode

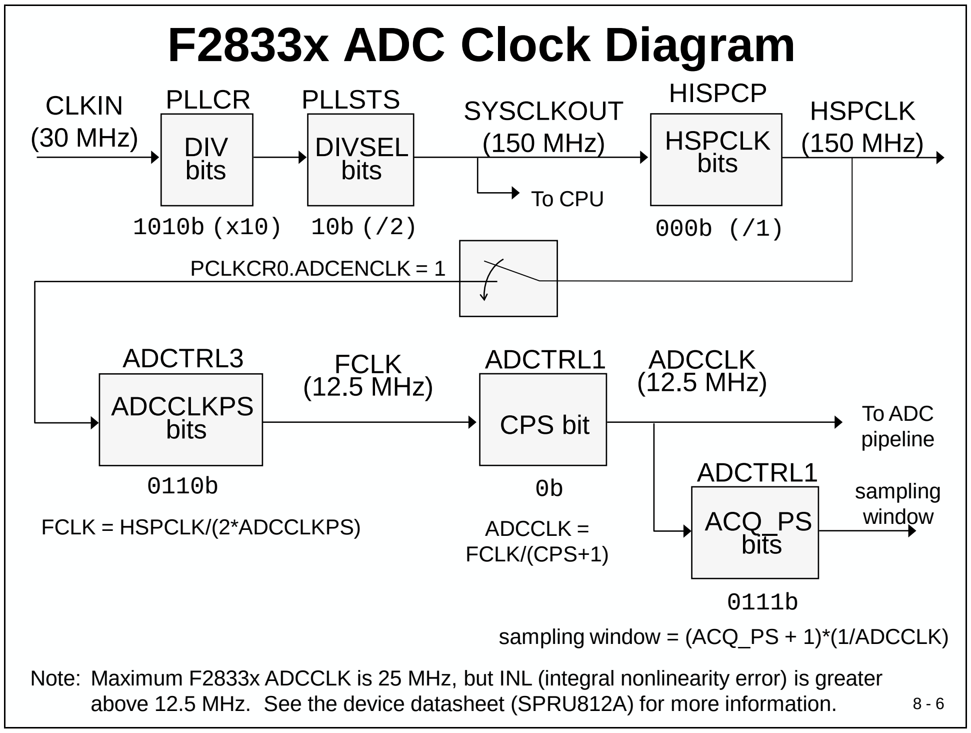


The second operating mode of the ADC is called “Dual Sequencer Mode”, which splits the Auto-Sequencer into two independent state machines (“SEQ1” and “SEQ2”). This mode uses the signal ePWM_SOC_A (“Start Of Conversion A”) as the hardware trigger for SEQ1 and ePWM_SOC_B for SEQ2. To code the input channels for the individual states of the two sequencers, we are free to select any of the 16 inputs for any of the 2x8 states. The registers RESULT0 to RESULT7 contain the values from SEQ1 and registers RESULT8 to RESULT15 for SEQ2.

The reason for this split mode is to have two independent ADCs, triggered by their own control time base for SEQ1 and SEQ2. In the ePWM chapter you will learn that we can generate ePWM_SOC_A and ePWM_SOC_B by various time events in any of the ePWM units. As an example you can use ePWM1-3 as the control system for a first 3-phase motor control unit and ePWM4-6 for a second one. In such a scenario SEQ1 will be the measurement unit for motor 1 and SEQ for motor 2.

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 until the end of the SEQ1 conversion sequence.

ADC Conversion Time



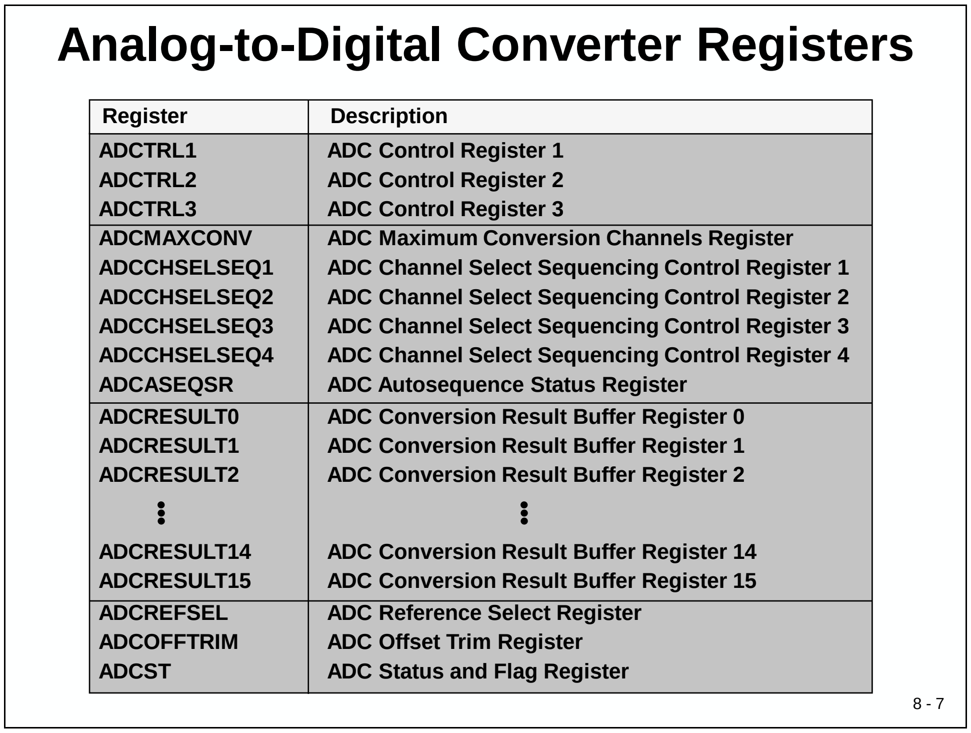
There are some limitations when setting up the ADC conversion time. First, the basic clock source for the ADC is the internal clock HSPCLK - we cannot use any clock speed we like. This clock is derived from the external oscillator, multiplied by PLLCR and divided by HISPCP. We discussed these bit fields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

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. However, when we use this maximum frequency we get a rising nonlinearity error for the results. In cases where we do not need that high conversion rate, it is better to limit FCLK to 12.5 MHz. To setup FCLK we have to initialise the bit field “ADCCLKPS” accordingly. Bit “CPS” gives the option of another divide by 2. The “ADCCLK” clock provides 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 cannot 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 variable resistors of the Peripheral Explorer Board.

ADC Register Block

Three control registers “ADCTRL1 to ADCTRL3” are used to set up one of the various operating conditions of the ADC unit. Register “ADCST” covers the current status of the ADC.

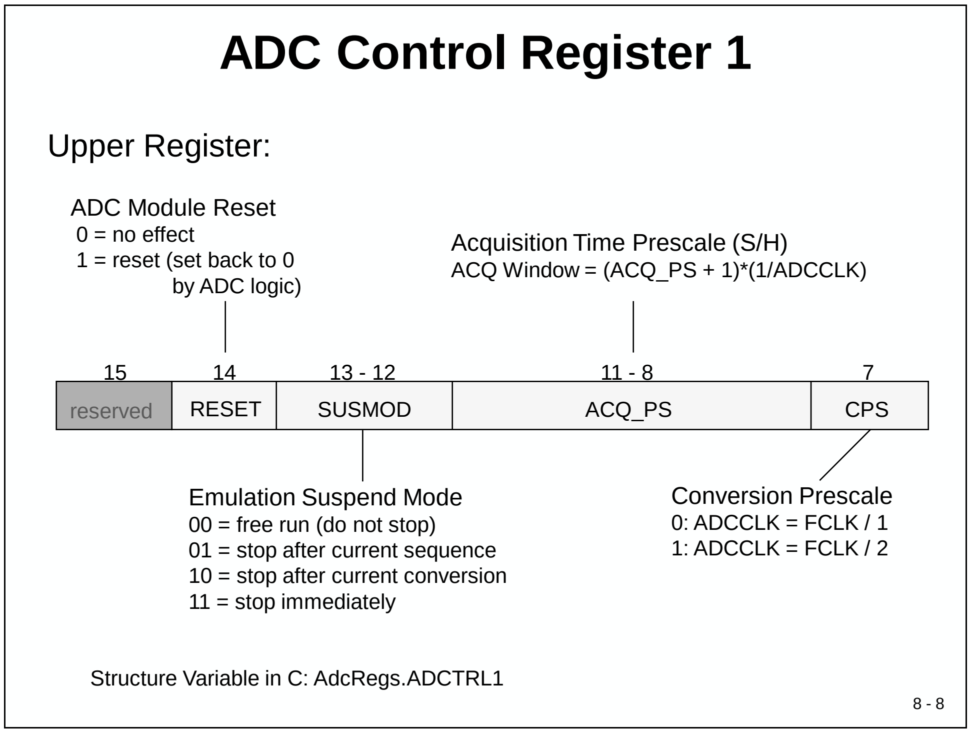


Analog-to-Digital Converter Registers

Register	Description
ADCTRL1	ADC Control Register 1
ADCTRL2	ADC Control Register 2
ADCTRL3	ADC Control Register 3
ADCMAXCONV	ADC Maximum Conversion Channels Register
ADCCHSELSEQ1	ADC Channel Select Sequencing Control Register 1
ADCCHSELSEQ2	ADC Channel Select Sequencing Control Register 2
ADCCHSELSEQ3	ADC Channel Select Sequencing Control Register 3
ADCCHSELSEQ4	ADC Channel Select Sequencing Control Register 4
ADCASEQSR	ADC Autosequence Status Register
ADCRESULT0	ADC Conversion Result Buffer Register 0
ADCRESULT1	ADC Conversion Result Buffer Register 1
ADCRESULT2	ADC Conversion Result Buffer Register 2
⋮	⋮
ADCRESULT14	ADC Conversion Result Buffer Register 14
ADCRESULT15	ADC Conversion Result Buffer Register 15
ADCREFSEL	ADC Reference Select Register
ADCOFFTRIM	ADC Offset Trim Register
ADCST	ADC Status and Flag Register

8 - 7

ADC Control Register 1


Upper Register:
ADC Module Reset
0 = no effect
1 = reset (set back to 0 by ADC logic)
Acquisition Time Prescale (S/H)
ACQ Window = (ACQ_PS + 1)*(1/ADCCLK)
15
14
13 - 12
11 - 8
7
reserved
RESET
SUSMOD
ACQ_PS
CPS
Emulation Suspend Mode
00 = free run (do not stop)
01 = stop after current sequence
10 = stop after current conversion
11 = stop immediately
Conversion Prescale
0: ADCCLK = FCLK / 1
1: ADCCLK = FCLK / 2

Structure Variable in C: AdcRegs.ADCTRL1

8 - 8

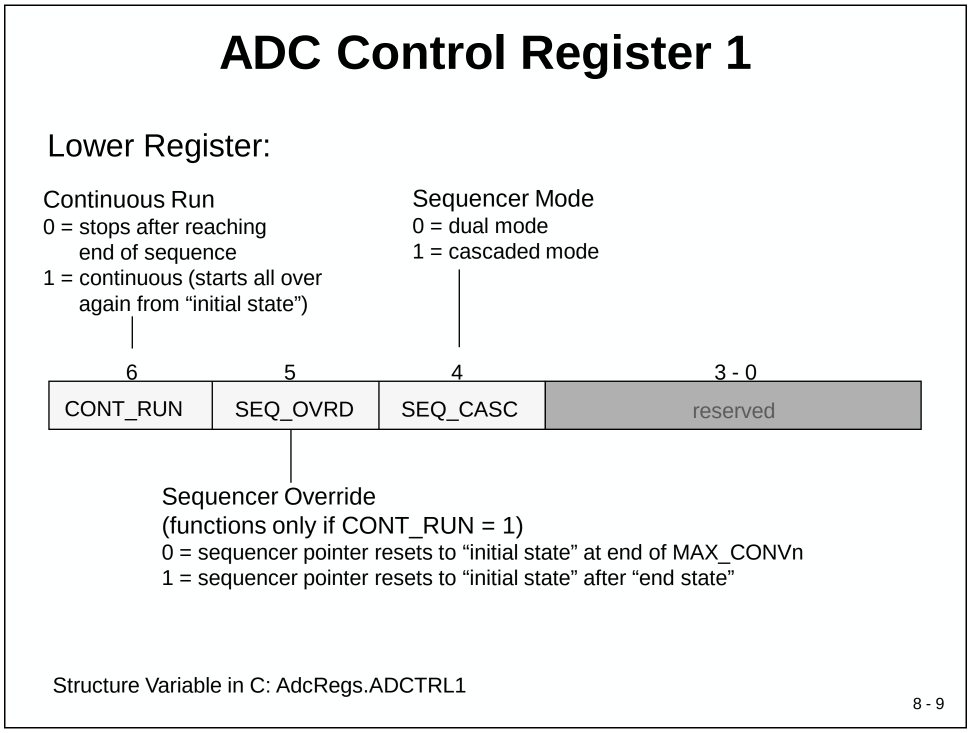
ADC Control Register 1

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. Please note that you cannot initialize the rest of this register in the same instruction, where you reset the ADC - so use a follow-up instruction to initialize ADCTRL1.

Bits 13 and 12 ("SUSMOD") define the interaction between the ADC and an emulator command, similar to the behaviour that we already discussed in chapter 7 (ePWM-module).

Bits 11 to 8 ("ACQ_PS") define the length of the sample window.

Bit 7 ("CPS") is used to divide the input frequency by 1 or 2.

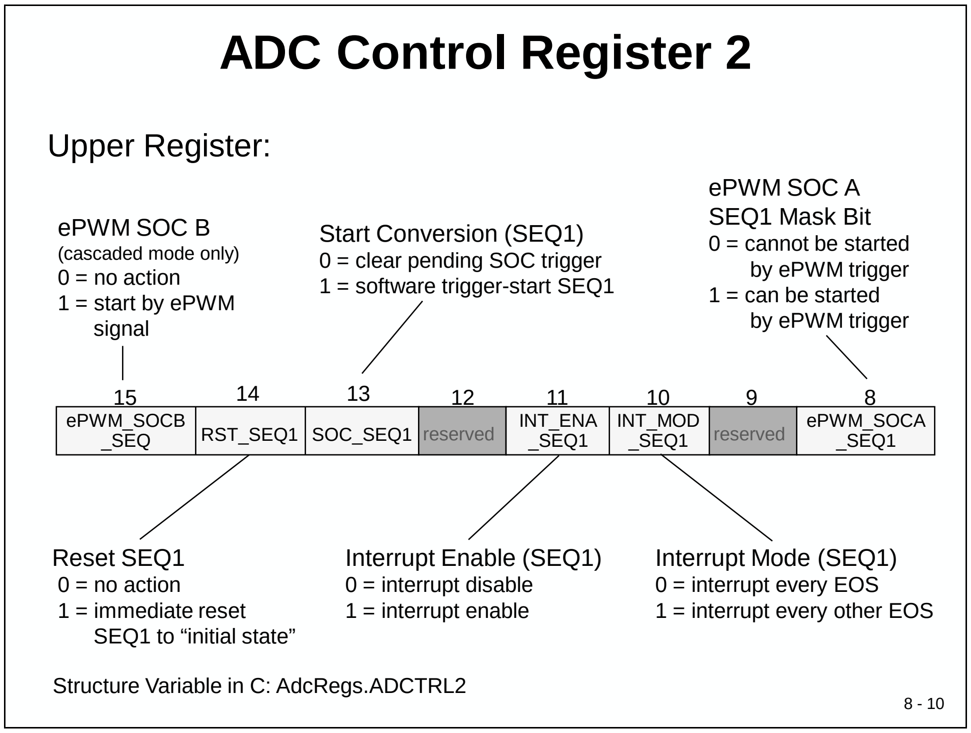

ADC Control Register 1
Lower Register:
Continuous Run
0 = stops after reaching end of sequence
1 = continuous (starts all over again from "initial state")
Sequencer Mode
0 = dual mode
1 = cascaded mode
6
5
4
3 - 0
CONT_RUN
SEQ_OVRD
SEQ_CASC
reserved
Sequencer Override
(functions only if CONT_RUN = 1)
0 = sequencer pointer resets to "initial state" at end of MAX_CONVn
1 = sequencer pointer resets to "initial state" after "end state"
Structure Variable in C: AdcRegs.ADCTRL1
8 - 9

Bit 6 ("CONT_RUN") defines whether 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("SEQ_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 ("SEQ_CASC") is the sequence/cascade bit. It defines the Sequencer Mode to be a state machine with 16 states (SEQCASC = 1), or to operate as two independent state machines, each having 8 states (SEQ_CASC = 0).

ADC Control Register 2


ADC Control Register 2

Upper Register:

ePWM SOC B
(cascaded mode only)
0 = no action
1 = start by ePWM signal

Start Conversion (SEQ1)
0 = clear pending SOC trigger
1 = software trigger-start SEQ1

ePWM SOC A SEQ1 Mask Bit
0 = cannot be started by ePWM trigger
1 = can be started by ePWM trigger

15	14	13	12	11	10	9	8
ePWM_SOCB _SEQ	RST_SEQ1	SOC_SEQ1	reserved	INT_ENA _SEQ1	INT_MOD _SEQ1	reserved	ePWM_SOCA _SEQ1

Reset SEQ1
0 = no action
1 = immediate reset SEQ1 to “initial state”

Interrupt Enable (SEQ1)
0 = interrupt disable
1 = interrupt enable

Interrupt Mode (SEQ1)
0 = interrupt every EOS
1 = interrupt every other EOS

Structure Variable in C: AdcRegs.ADCTRL2

8 - 10

The upper half of the ADCTRL2 register is responsible for controlling the operating mode of sequencer SEQ1.

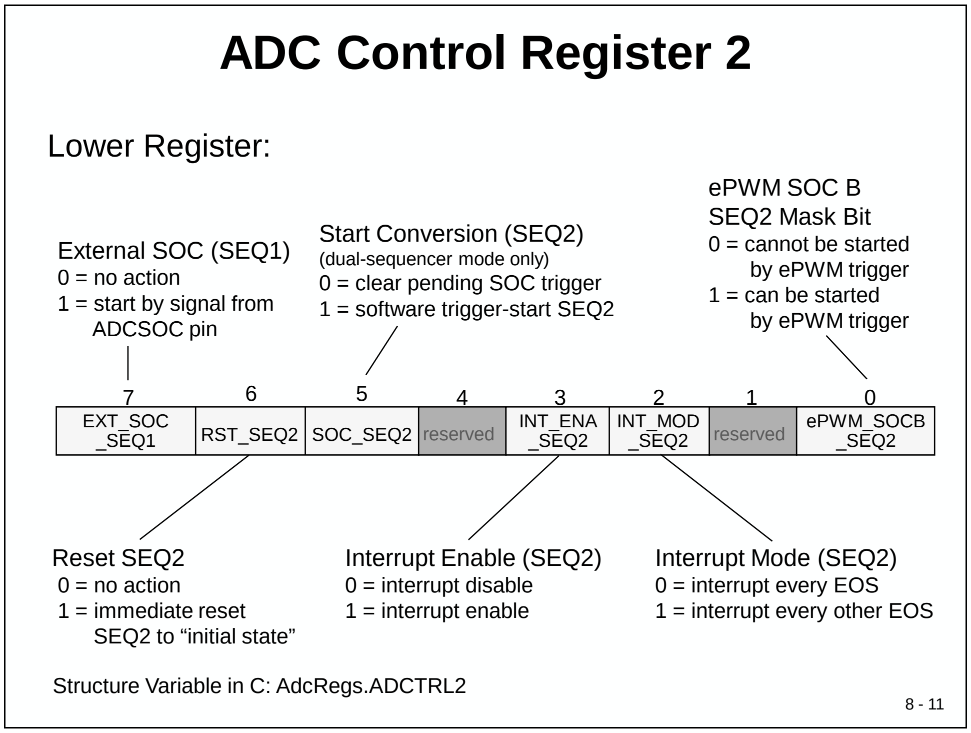
Setting Bit 15 (“ePWM_SOCB_SEQ”) allows the cascaded sequencer to be started by an ePWM SOCB signal. The bit is not working in “Dual Sequencer Mode” (see bit 0).

Using Bit 14 (“RST_SEQ1”), we can reset the state machine of SEQ1 to its initial state. This means that the next trigger will start a new conversion of the channel defined in CONV00.

When we set Bit 13 (“SOC_SEQ1”) to 1, we perform an immediate start of the conversion under software control.

Bits 11 (“INT_ENA_SEQ1”) and 10 (“INT_MOD_SEQ1”) define the interrupt mode of SEQ1. We can specify whether we have an interrupt request every “End of Sequence” (EOS) or every other (EOS).

Bit 8 (“ePWM_SOCA_SEQ1”) is the mask bit to allow the ePWM-signal “SOCA” to be used as the trigger for a conversion. In Lab8_1 we will use of this start feature, so please remember to set this bit in the initialization part for Lab8_1!



ADC Control Register 2

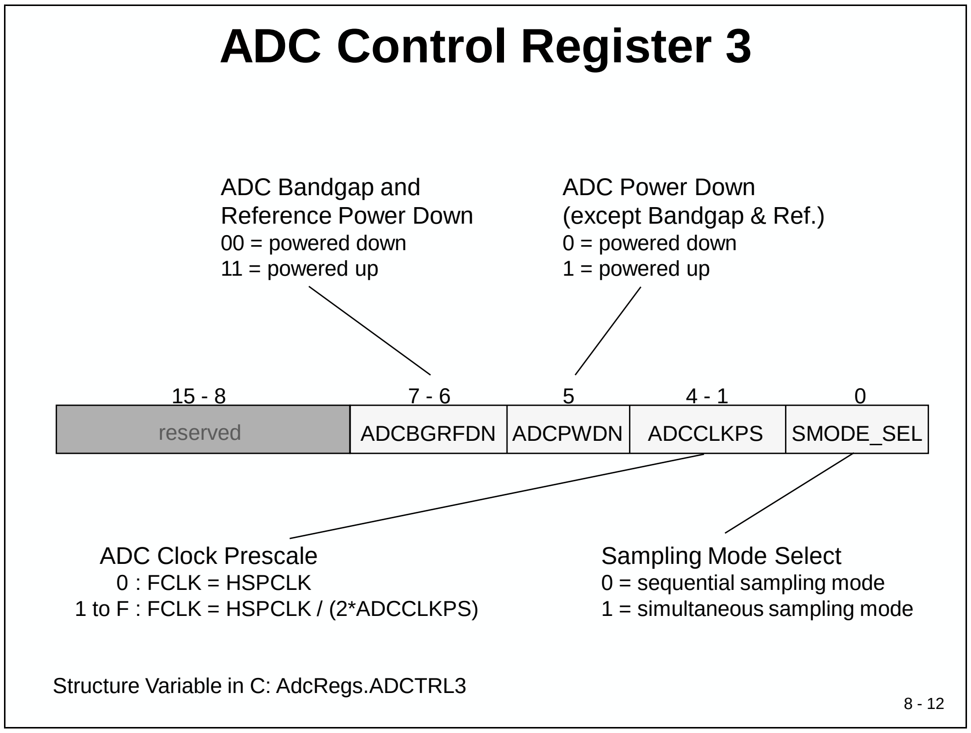
Lower Register:

The lower byte of ADCTRL2 is similar to its upper half: it controls sequencer SEQ2.

Setting Bit 7 enables an ADC auto conversion sequence to be started by a signal from a GPIO Port A pin (GPIO31-0) configured as XINT2 in the GPIOXINT2SEL register.

Bit 6 to Bit 0: The remaining part of ADCTRL2 is similar to Bits 14…8 in the upper half of the register. However they are used to initialize the operating mode of SEQ2. If we do not use sequencer 2 because we are in "Cascaded Mode", these bits are don't cares.

ADC Control Register 3


ADC Control Register 3
ADC Bandgap and Reference Power Down
00 = powered down
11 = powered up
ADC Power Down (except Bandgap & Ref.)
0 = powered down
1 = powered up
15 - 8
7 - 6
5
4 - 1
0
reserved
ADCBGRFDN
ADCPWDN
ADCCLKPS
SMODE_SEL
ADC Clock Prescale
0 : FCLK = HSPCLK
1 to F : FCLK = HSPCLK / (2*ADCCLKPS)
Sampling Mode Select
0 = sequential sampling mode
1 = simultaneous sampling mode
Structure Variable in C: AdcRegs.ADCTRL3
8 - 12

Bit 0 selects the sampling mode to be sequential or simultaneous. Recall that in simultaneous mode two analogue input signals are converted in parallel.

- Example: Let us assume that you would like to convert signals ADCINA4 and ADCINB4 in parallel. All you have to do is to initialize:
  - o SMODE_SEL = 1 // simultaneous sampling
  - o MAXCONV = 0 // 1 conversion; actually 2, because of SMODE_SEL = 1
  - o CONV00 = 4 // channel number for ADCINA4

  After the conversion is complete, register RESULT0 will contain the value for ADCINA4 and register RESULT1 the value for ADCINB4

Bit 4-1 are used to initialize the FCLK as basic clock of the ADC module (see also Slide 8-6).

Bit 5 is the main power switch for the analog circuitry inside the device. By setting this bit we power up the ADC except the band gap and reference circuitry.

Bits 7-6 control the ADC band gap and reference voltage power down sequence of the internal reference voltage system.

- Bits 7-6 = 00: The band gap and reference circuitry is powered down.
- Bits 7-6 = 11: The band gap and reference circuitry is powered up.

Note: *If we use the internal reference circuitry, we first have to set bits 7-6 to 11 followed by the set of bit 5.*

ADC MAXCONV Register

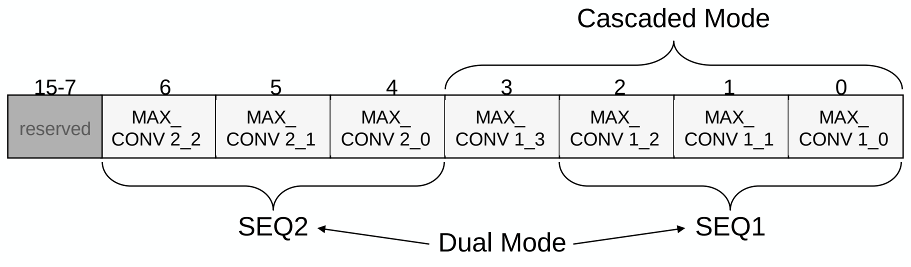
”MAXCONV” defines the number of conversion stages of the Auto sequencer. After a valid trigger signal, the Auto sequencer will convert the predefined number of channels automatically.

Please note that the number in the register bit fields corresponds to the number of conversions minus 1.

- MAXCONV = 4 // means 5 conversions, with input channel numbers coded // in bit fields CONV00 to CONV04 of register // ADCCHSELSEQ1 and ADCCHSELSEQ2

Maximum Conversion Channels Register

♦ Bit fields define the number of conversions per trigger (binary+1)



♦ Each sequencer starts at the “initial state” and advances sequentially
♦ Each will wrap at the “end state” unless software resets it sooner

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

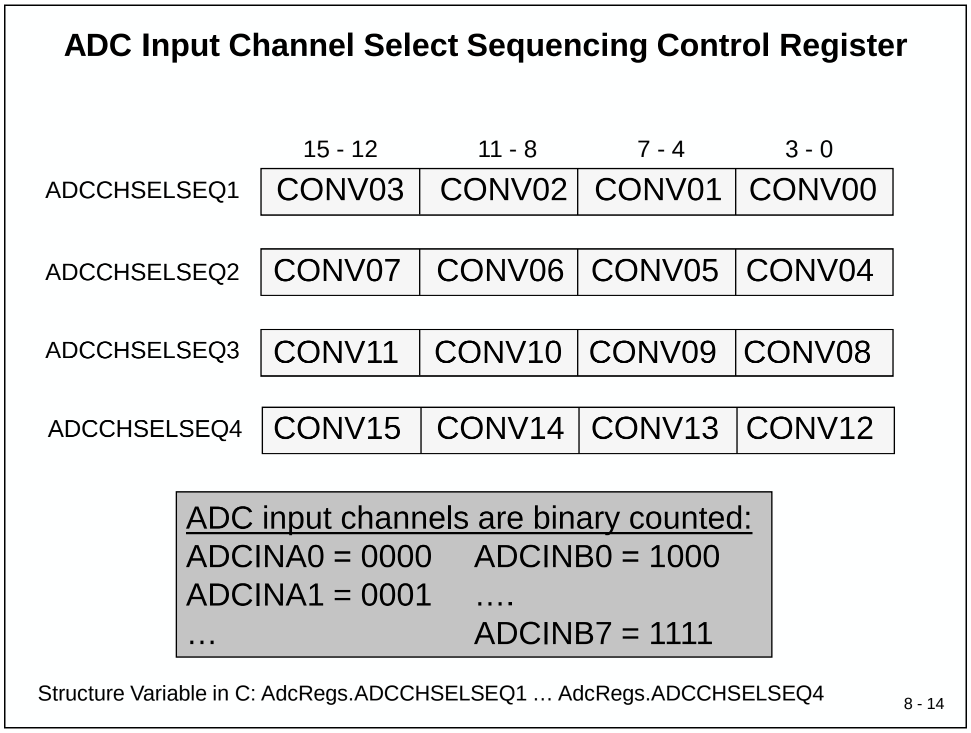
Structure Variable in C: AdcRegs.ADCMAXCONV

8 - 13

If we would use “Dual Sequencer Mode” the interpretation of register MAXCONV changes slightly. In this mode bits 0 to 2 are used to specify the number of conversions in sequencer SEQ1 and bits 4 to 6 are used for SEQ2. Recall that in this mode each sequencer has a maximum number of 8 conversions, hence the limitation to 3 bits in MAXCONV.

The Auto sequencer operates as a state machine that starts with an initial state and progresses after each conversion to the next one. This principle continues until the end state or until we reset the state machine pointer back to init state (Bit 14 and Bit 6 of ADCTRL2). If we do not reset and the state machine has reached the end state, it will wrap back to state zero automatically.

ADC Input Channel Select Registers



ADC Input Channel Select Sequencing Control Register

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

ADC input channels are binary counted:

ADCINA0 = 0000	ADCINB0 = 1000
ADCINA1 = 0001	….
…	ADCINB7 = 1111

Structure Variable in C: AdcRegs.ADCCHSELSEQ1 … AdcRegs.ADCCHSELSEQ4

8 - 14

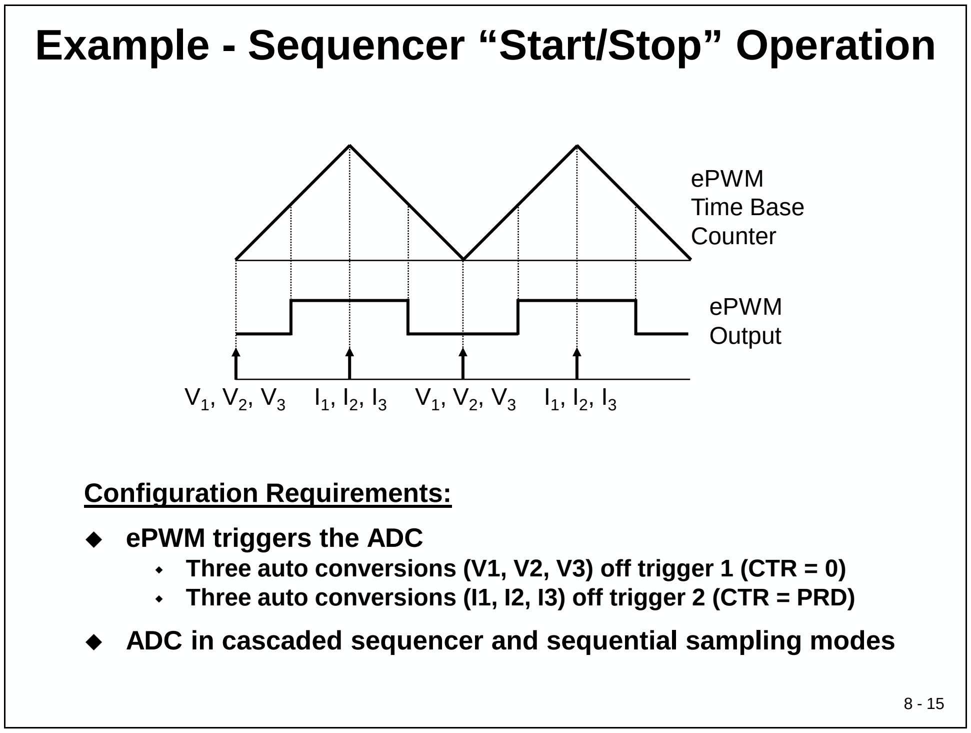
The group of four registers ADCCHSELSEQ1…4 is used to specify the binary number of the input channel ADCINA0…ADCINB7 by means of sixteen 4-bit -groups CONV00…CONV15.

Recall that we can use up to 16 stages in the Auto sequencer. These stages correspond to CONV00 to CONV15. All we have to do is to fill in the correct numbers for the analogue input channels (see Slide 8-14).

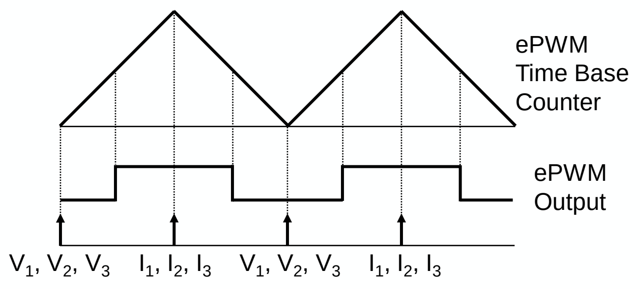
Example:

- Conversion of 5 channels in a sequence:
- ADCINA6, ADCINB1, ADCINA2, ADCINA0 and ADCINB6
  - o CONV00 = 6
  - o CONV01 = 9
  - o CONV02 = 2
  - o CONV03 = 0
  - o CONV04 = 14

Example: 3 phase control system & measurement



Example - Sequencer "Start/Stop" Operation

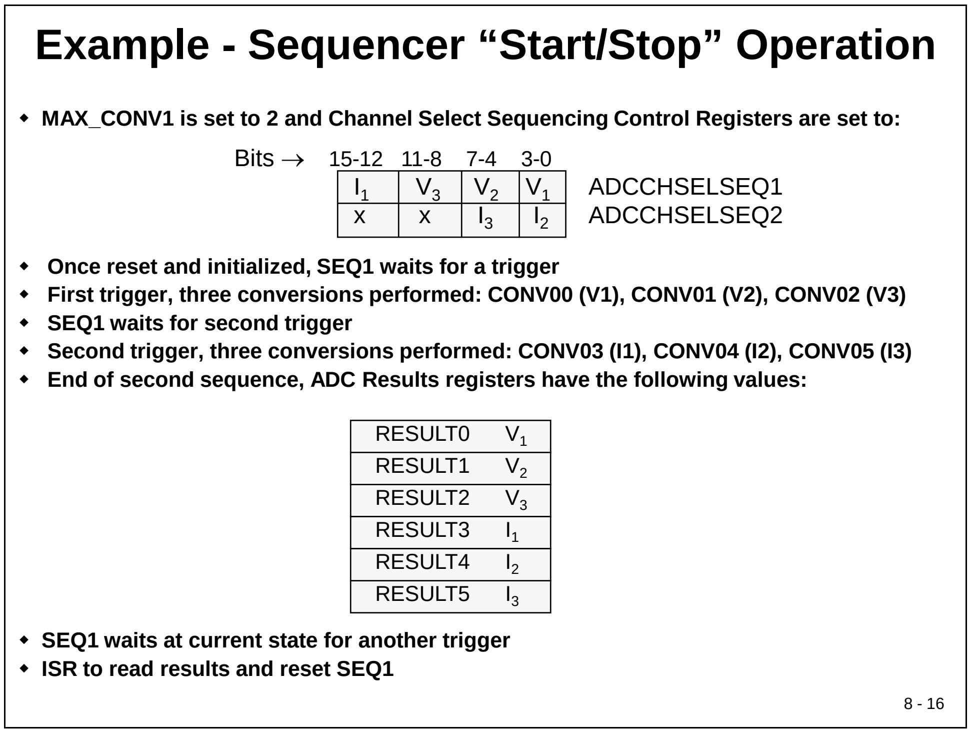


Configuration Requirements:

- **ePWM triggers the ADC**
  - **Three auto conversions (V1, V2, V3) off trigger 1 (CTR = 0)**
  - **Three auto conversions (I1, I2, I3) off trigger 2 (CTR = PRD)**
- **ADC in cascaded sequencer and sequential sampling modes**

8 - 15

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



Example - Sequencer "Start/Stop" Operation

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

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

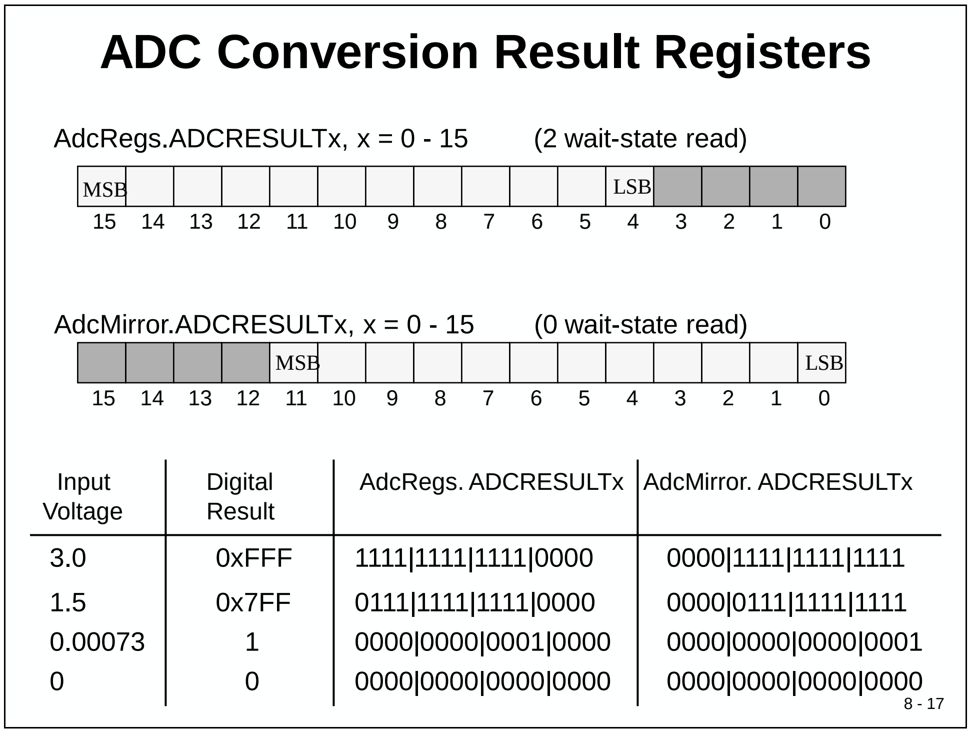
- **Once reset and initialized, SEQ1 waits for a trigger**
- **First trigger, three conversions performed: CONV00 (V1), CONV01 (V2), CONV02 (V3)**
- **SEQ1 waits for second trigger**
- **Second trigger, three conversions performed: CONV03 (I1), CONV04 (I2), CONV05 (I3)**
- **End of second sequence, ADC Results registers have the following values:**

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

- **SEQ1 waits at current state for another trigger**
- **ISR to read results and reset SEQ1**

8 - 16

ADC Result Register Set



ADC Conversion Result Registers

AdcRegs.ADCRESULTx, x = 0 - 15 (2 wait-state read)

MSB											LSB	▨	▨	▨	▨
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

AdcMirror.ADCRESULTx, x = 0 - 15 (0 wait-state read)

▨	▨	▨	▨	MSB											LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

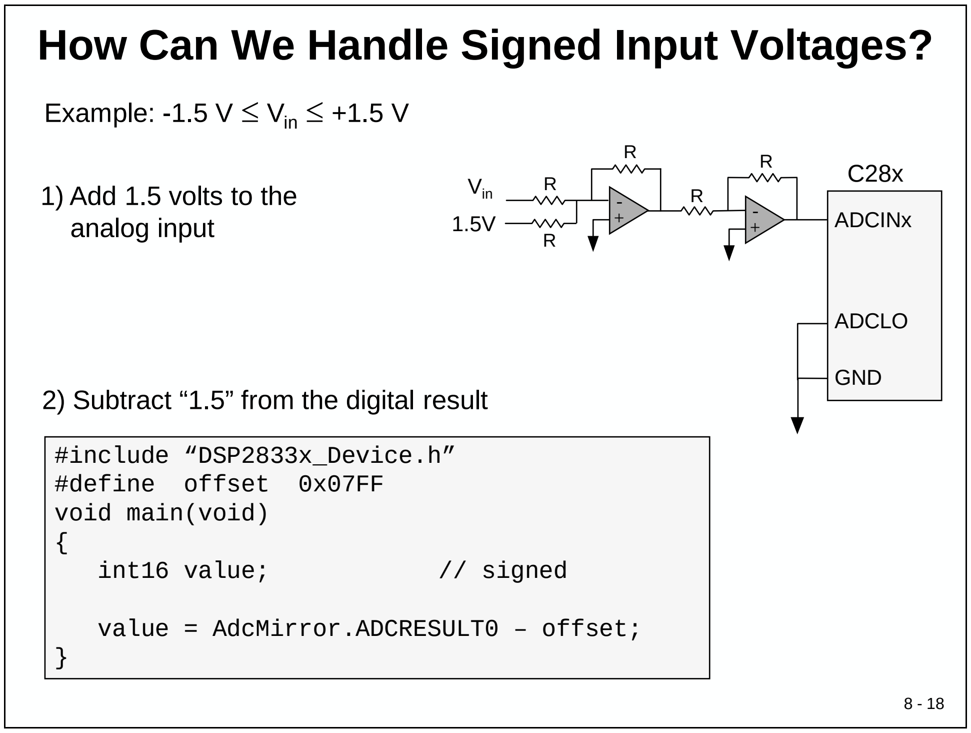
Input Voltage	Digital Result	AdcRegs. ADCRESULTx	AdcMirror. ADCRESULTx
3.0	0xFFF	1111|1111|1111|0000	0000|1111|1111|1111
1.5	0x7FF	0111|1111|1111|0000	0000|0111|1111|1111
0.00073	1	0000|0000|0001|0000	0000|0000|0000|0001
0	0	0000|0000|0000|0000	0000|0000|0000|0000

8 - 17

The 12-bit digital results are available in two different memory sections.

The ADCRESULTn registers are left justified when read from Peripheral Frame 2 (0x7108-0x7117; global C- variable "AdcRegs") with two wait states and right justified when read from Peripheral Frame 0 (0x0B00-0x0B0F; global C-variable "AdcMirror") with zero wait states.

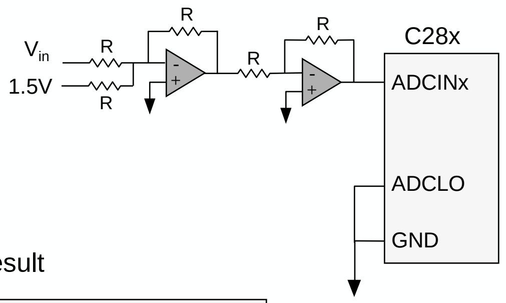
Left justified results are advantageous when a control system operates on "fractional" numbers. We will see how this makes scaling easier in a later chapter of this course.



How Can We Handle Signed Input Voltages?

Example: -1.5 V ≤ V_{in} ≤ +1.5 V

1) Add 1.5 volts to the analog input



2) Subtract “1.5” from the digital result

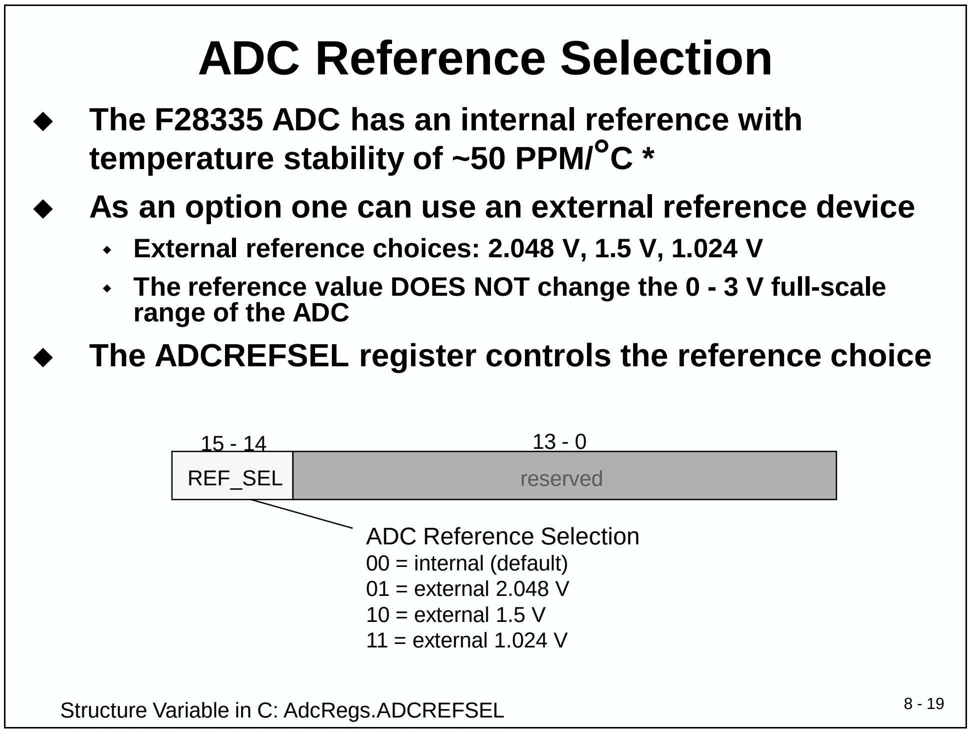
```
#include “DSP2833x_Device.h”
#define  offset  0x07FF
void main(void)
{
   int16 value;            // signed

   value = AdcMirror.ADCRESULT0 – offset;
}
```

8 - 18

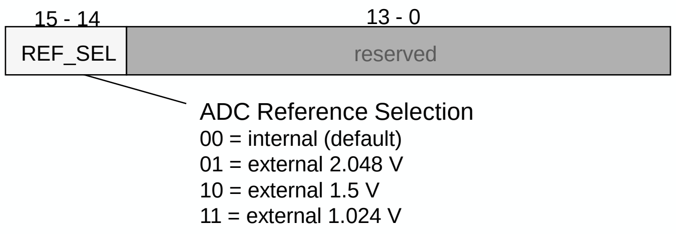
ADCREFSEL Register

To switch between internal or external ADC reference voltages, we could use register ADCREFSEL. For our next lab experiments using the Peripheral Explorer Board we will stay with the internal voltage source, which is selected by default.



ADC Reference Selection

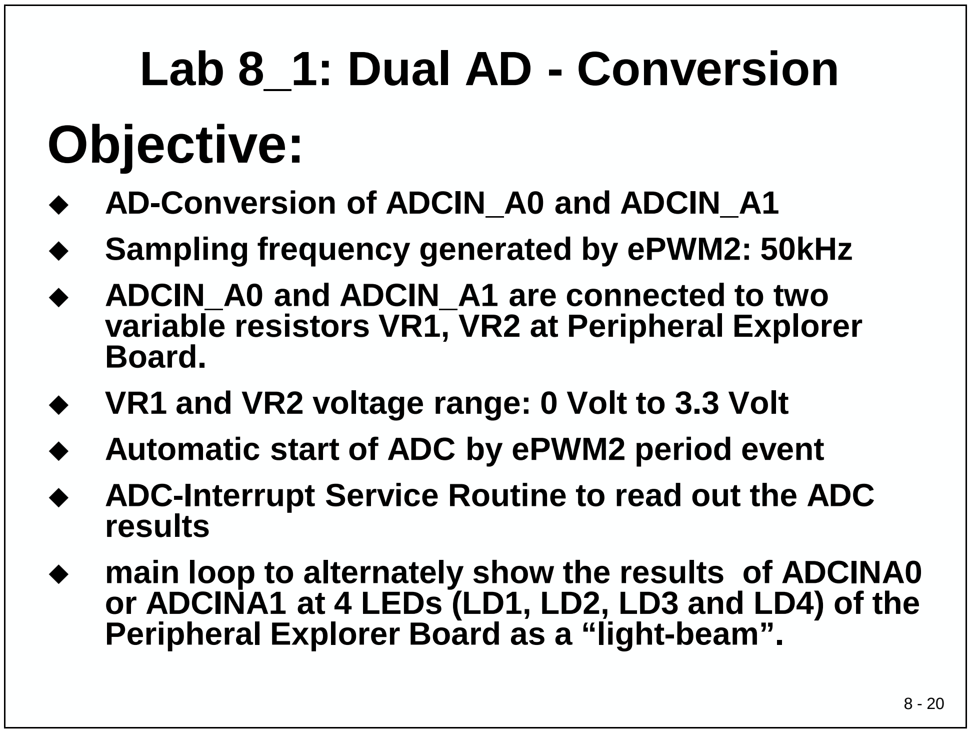
- **The F28335 ADC has an internal reference with temperature stability of ~50 PPM/°C ***
- **As an option one can use an external reference device**
  - **External reference choices: 2.048 V, 1.5 V, 1.024 V**
  - **The reference value DOES NOT change the 0 - 3 V full-scale range of the ADC**
- **The ADCREFSEL register controls the reference choice**



Structure Variable in C: AdcRegs.ADCREFSEL

8 - 19

Lab 8_1: Two Potentiometer Voltages

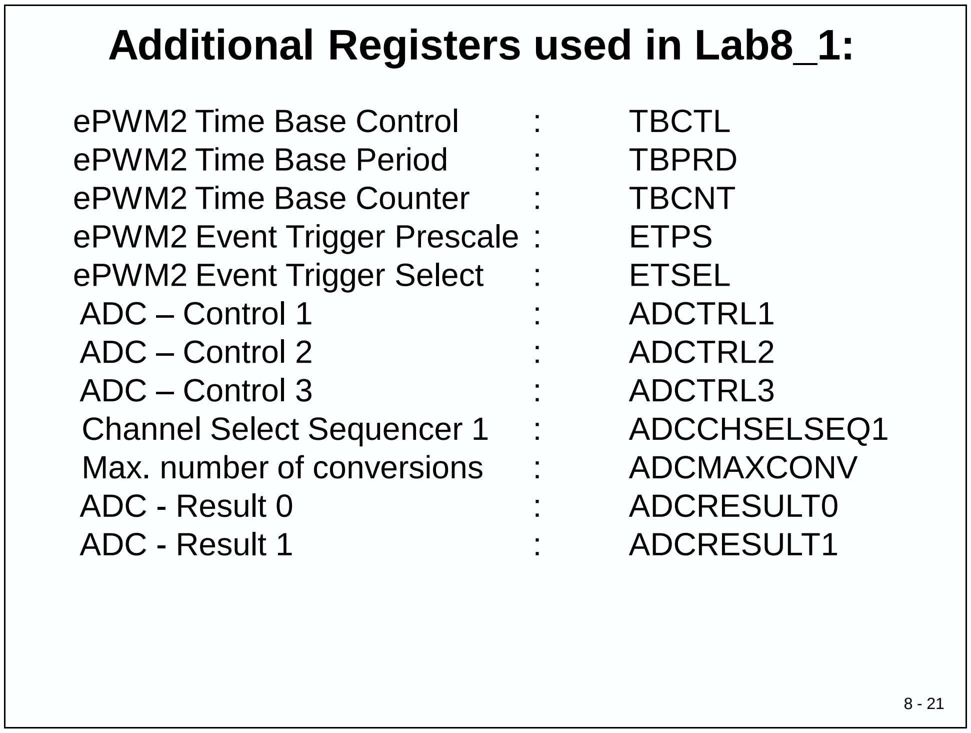


Lab 8_1: Dual AD - Conversion

Objective:

- **AD-Conversion of ADCIN_A0 and ADCIN_A1**
- **Sampling frequency generated by ePWM2: 50kHz**
- **ADCIN_A0 and ADCIN_A1 are connected to two variable resistors VR1, VR2 at Peripheral Explorer Board.**
- **VR1 and VR2 voltage range: 0 Volt to 3.3 Volt**
- **Automatic start of ADC by ePWM2 period event**
- **ADC-Interrupt Service Routine to read out the ADC results**
- **main loop to alternately show the results of ADCINA0 or ADCINA1 at 4 LEDs (LD1, LD2, LD3 and LD4) of the Peripheral Explorer Board as a “light-beam”.**

8 - 20



Additional Registers used in Lab8_1:

ePWM2 Time Base Control	:	TBCTL
ePWM2 Time Base Period	:	TBPRD
ePWM2 Time Base Counter	:	TBCNT
ePWM2 Event Trigger Prescale	:	ETPS
ePWM2 Event Trigger Select	:	ETSEL
ADC – Control 1	:	ADCTRL1
ADC – Control 2	:	ADCTRL2
ADC – Control 3	:	ADCTRL3
Channel Select Sequencer 1	:	ADCCHSELSEQ1
Max. number of conversions	:	ADCMAXCONV
ADC - Result 0	:	ADCRESULT0
ADC - Result 1	:	ADCRESULT1

8 - 21

Objective

The objective of this lab is to practice using the integrated Analogue-Digital Converter of the F2833x. The Peripheral Explorer Board is equipped with 2 variable resistors VR1 and VR2, which are connected to the analogue input lines ADCIN_A0 and ADCIN_A1. The two input voltages can be adjusted between 0 and 3.0 volts. In this lab we will read the current status of the potentiometers and display the converted voltages on LEDs (LD1 to LD4) of the Peripheral Explorer Board (GPIO9, GPIO11, GPIO34 and GPIO49) in form of a “light-beam” .

The ePWM2 unit will generate the sampling frequency of 50 kHz (or sampling period of 20µs). The conversion is triggered automatically by signal “SOCA” at the period match of ePWM2. The ADC interrupt service routine will be used to copy the 12-bit results into two global variables “Voltage_VR1” and “Voltage_VR2”.

CPU Timer 0 will be used to generate a time base for the monitoring part of this lab exercise. It will be initialized to run at a period of 100 milliseconds. The interrupt service routine will increment a global variable “CpuTimer0.InterruptCount”. Based on the value in this variable we can establish an alternation in the display between VR1 and VR2 every 0.5 seconds.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8.pjt** in C:\DSP2833x\Labs.

2. Open file Lab6.c from C:\DSP2833x\Labs \Lab6 and save it as Lab8_1.c in C:\DSP2833x\Labs \Lab8.

3. Add the source code file to your project:
   - **Lab8_1.c**

4. Add the provided source code file “Display_ADC.c” to your project. This file, enclosed in the file “labs_08.zip”, includes a function “display_ADC()”, which converts a 12 bit unsigned integer number into a “light-beam” at the four LEDs. The term “light-beam” means that the bigger the input value is, the more LEDs are switched on.

5. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\source* add:
   - **DSP2833x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\cmd* add:
   - **28335_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd* add:
   - **DSP2833x_Headers_nonBIOS.cmd**

   From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* add to project:

- **DSP2833x_PieCtrl.c**
- **DSP2833x_PieVect.c**
- **DSP2833x_DefaultIsr.c**
- **DSP2833x_Adc.c**
- **DSP2833x_SysCtrl.c**
- **DSP2833x_CpuTimers.c**
- **DSP2833x_usDelay.asm**
- **DSP2833x_ADC_cal.asm**

From *C:\CCStudio_v3.3\c2000\cgtools\lib* add:

- **rts2800_fpu32.lib**

Project Build Options

6. Set up 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\dsp2833x\v131\DSP2833x_headers\include;
   C:\tidcs\C28\dsp2833x\v131\DSP2833x_common\include

7. Set up the floating-point support for the C-compiler. Inside Build Options select the Compiler tab. In the "Advanced" category set "Floating Point Support" to

   fpu32

8. Set up 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 Lab8_1.c to edit: double click on “Lab8_1.c” inside the project window. First delete the local variable “counter” from main, including the definition at the beginning of “main()” and the use of “counter” in the endless-loop at the end of “main()”.

10. Add two new global unsigned integer variables “Voltage_VR1” and “Voltage_VR2”.

11. In the endless loop of “main()”, change the wait construction based on the variable “CpuTimer0.InterruptCount” to wait until it has value 5. Recall that this variable is incremented every 100 milliseconds by CPU Timer 0 ISR. This way we can include a time delay of 500 milliseconds. Also recall, that the maximum

overflow period for the watchdog unit is less than 500 milliseconds. While the second clear instruction for the watchdog unit is part of CPU Timer 0 ISR, we have to embed the 0x55 - instruction into our while-wait loop.

12. After this wait-loop, add a call or function “display_ADC()”:

    display_ADC(Voltage_VR1);

13. Next, add a similar wait-loop like in procedure step 11 and wait until variable “CpuTimer0.InterruptCount” has a value of 10. This will give us another interval of 500 milliseconds.

14. Now call function “display_ADC()” for variable “Voltage_VR2”.

15. Right after step 14 clear variable “CpuTimer0.InterruptCount” to zero.

16. The provided function “display_ADC()” has been defined in an external file. To be able to use this function, we have to add an external prototype at the beginning of “Lab8_1.c”:

    extern void display_ADC(unsigned int);

Build, Load and Test

17. Although we haven’t initialized the ADC so far, it might make sense to perform a preliminary test. Do:

    - **Project → Build**
    - **File → Load Program** and choose the desired output file.
    - **Debug → Reset CPU** followed by
    - **Debug → Restart** and
    - **Debug → Go main.**

    Open a Watch-Window and add the two variables “Voltage_VR1” and “Voltage_VR2” to it. In Watch-Window change the values of “Voltage_VR1” to 0 and “Voltage_VR2” to 4000.

    - **Run (F5).**

    Result: All 4 LEDs should blink at a rate of 0.5 seconds on and off period.

Add ADC Initialization

18. Inside “main()”, after the function call “InitPieVectTable();” add the following line to call the basic ADC calibration and internal reference enabling function:

    InitAdc();

    Also add an external function prototype at the beginning of “Lab8_1.c”:

    extern void InitAdc(void);

19. In “main()”, straight after the function call of “InitAdc()”, add code to initialize the ADC register. Refer to the reference section or to the slides shown earlier with this presentation to complete the following register settings:

    For register ADCTRL1:

    - **AdcRegs.ADCTRL1.bit.SEQ_CASC = ?;** // Dual Sequencer Mode
    - **AdcRegs.ADCTRL1.bit.CONT_RUN = ?;** // Single Run Mode
    - **AdcRegs.ADCTRL1.bit.ACQ_PS = ?;** // 8 x ADC-Clock
    - **AdcRegs.ADCTRL1.bit.CPS = ?;** // divide by 1

    For register ADCTRL2:

    - **AdcRegs.ADCTRL2.bit.EPWM_SOCA_SEQ1 = ?;** // ePWM_SOCA trigger
    - **AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = ?;** // enable ADC interrupt from sequencer 1
    - **AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = ?;** // interrupt after every EOS

    For register ADCTRL3:

    - **AdcRegs.ADCTRL3.bit.ADCCLKPS = ?;** // set FCLK to 12.5 MHz

    Note: do NOT modify other bit fields of ADCTRL3 than “ADCCLKPS”. All other bits have been initialized by the function call “InitAdc()”. Use slide 8-6 to calculate the value for ADCCLKPS. If your F28335ControlCard is equipped with a 30MHz external clock, it runs at a SYSCLKOUT-Frequency of 150 MHz.

    For register MAXCONV:

    - **AdcRegs.ADCMAXCONV.all = ?**; // 2 conversions

    For register ADCCHSELSEQ1:

    - **AdcRegs.ADCCHSELSEQ1.bit.CONV00 = ?;** // 1st channel ADCINA0
    - **AdcRegs.ADCCHSELSEQ1.bit.CONV01 = ?;** // 2nd channel ADCINA1

Add ePWM2 Initialization

20. ePWM2 will be the clock base for the sampling frequency. We will setup this unit to run at 50 kHz and trigger a SOCA start of the ADC automatically at the end of a period. Right after the ADC-register initialization add code:

    For register EPwm2Regs.TBCTL, add code to:

    - Ignore emulation suspend
    - CLKDIV = HSPCLK/1
    - HSPCLK = SYSCLKOUT/1

- no SWFSYNC
- SYNC-Out disabled
- no PHSEN
- Reload TBPRD on TBCTR = 0
- CTRMODE = count up mode

For register TBPRD:

- TPPRD +1 = T_{PWM} / (HSPCLKDIV * CLKDIV * T_{SYSCLK})
  = 20 µs / 6.667 ns

For register ETPS:

- SOCAPRD : generate SOCA-signal on first event
- Clear all remaining bits of ETPS

For register ETSEL:

- SOCAEN: enable SOCA-signal
- SOCASEL: generate SOCA-signal on PRD event

Add ADC-Interrupt system

21. In “main()”, search for the re-map instruction of the PIE-table entry for “PieVectTable.TINT0 = &cpu_timer0_isr;”, which is embedded between “EALLOW” and “EDIS”. Also between these two instructions add:

    PieVectTable.ADCINT = &adc_isr;

22. Also add a line to enable the PIE-Interrupt for the ADC:

    PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

23. At the end of “Lab8_1.c” add a new interrupt service routine “adc_isr()” to your code. Inside this function, add the following:

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

      Voltage_A0 = AdcMirror.ADCRESULT0;

      Voltage_B0 = AdcMirror.ADCRESULT1;

    - 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;

24. At the beginning of “Lab8_1” add a function prototype for the ADC interrupt service routine:

interrupt void adc_isr(void);

Build, Load and Test

25. Now we can test the final version:

Project	**→ Build**	
File	**→ Load Program**	and choose the desired output file.
Debug	**→ Reset CPU**	followed by
Debug	**→ Restart**	and
Debug	**→ Go main.**	

26. Set a breakpoint in the interrupt service routine "adc_isr()" at the last line of code.

Run

27. When you have modified your code correctly and you execute running in real-time, 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 ePWM2 is counting (TBCTR)
    - Verify that the clock system is enabled (PCLKCR) for EPWM2 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.

28. After you have verified that the interrupt service routine "adc_isr()" is called periodically, check the ADC results. Inspect the variables "Voltage_VR1" and "Voltage_VR2" in your watch window. With the breakpoint still set, modify the analogue input voltages with the two potentiometers "VR1" and "VR2" of the Peripheral Explorer Board. You should obtain values between 0 and 4095 for the leftmost and rightmost positions of VR1 and VR2 respectively.

29. Now remove all breakpoints and run the code. LEDs LD1 to LD4 should display the values for "Voltage_VR1" and "Voltage_VR2" every 0.5 seconds.

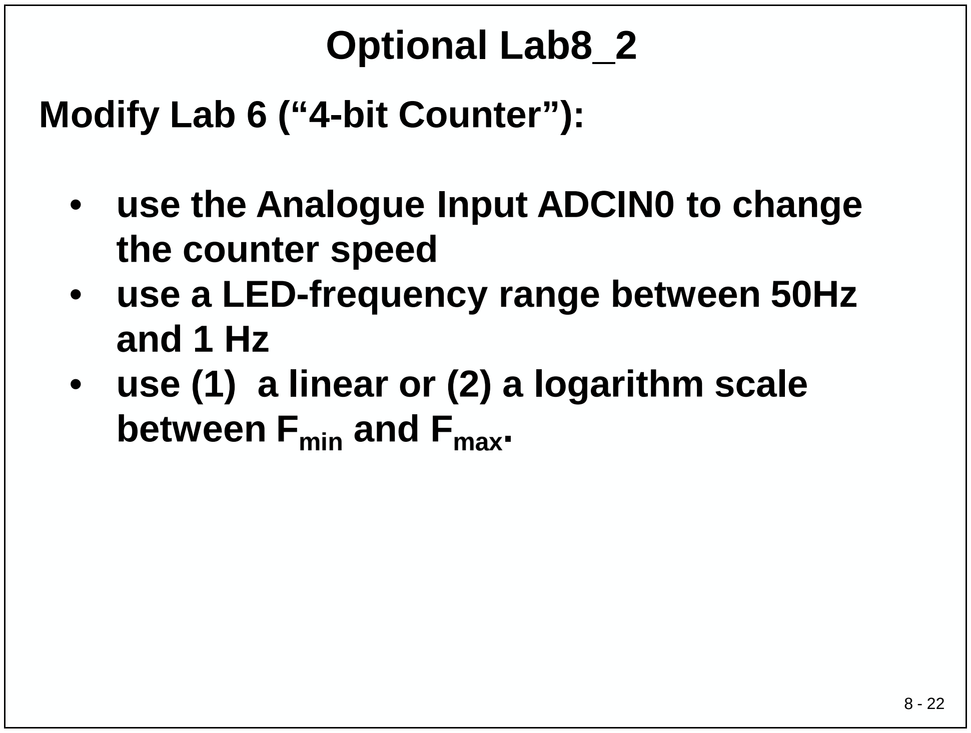
END of LAB 8_1

Lab 8_2: Analogue Control of "LED- counter"

Objective

Now that we have performed an exercise both with the ADC (Lab 8_1) and the CPU Timer 0 based binary counter in Lab6, we can combine the two exercises. The objective is to control the speed step of the binary LED-counter from Lab6 by a voltage taken from ADC-Input ADCIN_A0. The control law should be: The higher the voltage ADCIN_A0, the higher the speed of the LED-counter.

Use your code from Lab8_1 and Lab6 as the starting point.



Optional Lab8_2

Modify Lab 6 ("4-bit Counter"):

- **use the Analogue Input ADCIN0 to change the counter speed**
- **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}.**

8 - 22

Procedure

Open Project

1. If not still open from Lab8_1, re-open project **Lab8.pjt.**
2. Open the file "Lab8_1.c" and save it as "Lab8_2.c"
3. Remove file "Lab8_1.c" from project and add "Lab8_2.c" to it. Note: optionally you can also keep "Lab8_1.c" but exclude it from build. Use a right mouse click on file "Lab8_1.c", select "File Specific Options"; in category "General" enable "Exclude from Build".

Modify Source Code

4. Edit "Lab8_2.c". In "main()", remove the whole contents of the endless while(1) -loop and replace it with the contents of the while(1)-loop of file "Lab6.c"

5. At the beginning of "Lab8_2.c", remove the global variable "Voltage_VR2" and add an integer variable "counter"; initialize this counter to zero.

6. Change the ADC initialization to convert channel ADCINA0 only. Since we do not need VR2 in this exercise, also remove the read instruction for ADCRESULT1 from function "adc_isr()".

Build, Load and Test

7. Time for a preliminary test:

Project	**→ Build**	
File	**→ Load Program** and choose the desired output file.	
Debug	**→ Reset CPU**	followed by
Debug	**→ Restart**	and
Debug	**→ Go main.**	
Debug	**→ Run**	

Run the code. The LEDs should show the binary counter on LEDs LD1 to LD4. The counter period is still fixed to 100 milliseconds.

In the watch window, the variable "Voltage_VR1" should have a value between 0 and 4095. Modify the position of VR1, right click into Watch Window, select "Refresh" and verify that the value of "Voltage_VR1" changes accordingly.

So far we have reached the same result as in Lab6, a 100 millisecond period between the steps of the counter. However, now we have additionally an active ADC in the background!

Modify the main loop

8. All we have to do now is to use variable "Voltage_VR1" to control the period of CPU-Timer 0. Since we can exceed the Watchdog overflow period by the CPU Timer 0 period, it makes sense to include both watchdog clear instructions into the wait loop:

```
while(CpuTimer0.InterruptCount == 0)
    {
            EALLOW;
            SysCtrlRegs.WDKEY = 0x55;
            SysCtrlRegs.WDKEY = 0xAA;
            EDIS;
    }
```

Now we have to modify the period of CPU Timer0. Recall that the task is to generate a period between 20,000µs (50Hz) and 1,000,000µs (1Hz). And that CPU Timer 0 can be initialized by a function call as follows:

```
ConfigCpuTimer(&CpuTimer0,100,x);
```

Add such a function call directly after the wait-loop! Also, after this function call, re-enable CPU-Timer 0:

```
ConfigCpuTimer(&CpuTimer0,100,x);     // calculate x before calling
CpuTimer0Regs.TCR.bit.TSS = 0;        // restart timer0
```

Parameter x in this function call is a floating-point variable and gives the period in microseconds. What we have to do is to call this function with a value for x, which is calculated based on "Voltage_VR1" (0…4095). Recall that x should be between 20,000 µs (50Hz) and 1,000,000 µs (1Hz).

END of Lab 8_2

This page has been left blank intentionally.