EE 3171 Lecture 12

D/A and A/D Conversion

Lecture 8 Concepts

- ADC Conversion
  - Why and how in general
  - How to do it on the Tiva C
    - The registers
    - The programming
    - The alphabet soup

Why Do We Need Conversions?

- Real world is analog, not digital.
  - Microcontroller must interface to real world.
  - Efficient DTA and ADC are important.
    - Digital to analog is easy.
    - Analog to digital is harder.

Digital-to-Analog

- DTA output is easy.
  - We output the digital value.
  - A low-pass filter smoothes output.

Analog-to-Digital

- ADC input is harder.
    - We must represent an analog value by one of a set of digital values.
- We are constrained by the *precision* of our converter.
    - Which, in turn, defines the *resolution* of the converter.

ATD Resolution

- ADCs have a fixed number of bits they put the converted value into.
  - E.g., 8, 9, 10, 12 and so on.

This is the precision.

- The entire range of voltages gets divided up among the possible range of binary values.
- Let:
  - V_{RH} = the highest reference voltage
  - V_{RL} = the lowest reference voltage
  - b be the number of bits the ADC uses to store results
- Then the *resolution* = $\boxed{(V_{RH}-V_{RL})/2^b.}$

Resolution Examples

- Let V_{RH} = +5, V_{RL} = -5 and b = 10.
  - Resolution = ?
- If $V_{sampled}$ = 1.3V, what is the converted value?
- If the converted value is `0x3F`, what is $V_{sampled}$?

Basic ADC Procedure

- Goal: Find a binary value proportional to an input voltage.
- Typical ATD Algorithm
  - Pass input voltage to analog voltage comparator.
  - Guess the number represented by the voltage.
  - Pass number to DTA converter to get assumed voltage.
  - Pass assumed voltage to other input of comparator.
  - If Comparator output = 0
    Then the number guessed was correct (output it)
    Else guess again

The “Guess Again” Step

- The key issues are:
    - What’s the first guess?
    - How do we guess the next value?
- Basic methods are
    - Up counter
    - Successive approximation

Up Counter

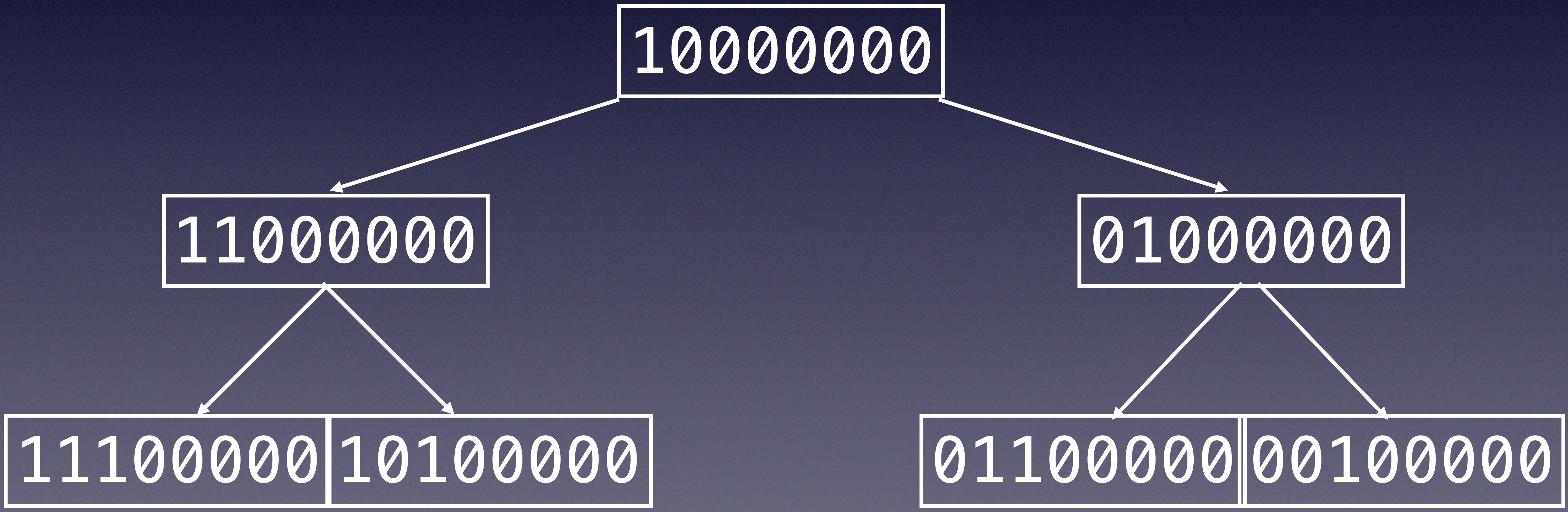
- Simplest method is an Up-counter
  - Done_Flag = 0
  - Number = 0
  - While difference ≠ 0
    - Increment number
  - Done_Flag = 1
  - Output the Number
- Problem: may have to count to `0xFFF` (or other maximum value).
  - Worst-case time = 2^b cycles where b is the number of bits.

Successive Approximation

- Classic binary search
  - Usually something around log b cycles
- Best Worst-Case Time
  - Done_Flag = 0
  - Number = $2^b/2$
  - While difference ≠ 0
    - If Guess is too low
      Then Number = Number + Number/2
      Else Number = Number - Number/2
  - Done_Flag = 1
  - Output the Number

Algorithm Progress

- Error is cut in half every step
- Compare against:



The Tiva ADC

- The TM4C123GH6PM microcontroller provides two ADC modules with each having the following features:
- 12 shared analog input channels
- 12-bit precision ADC
- On-chip internal temperature sensor
- Maximum sample rate of one million samples/second
- Four programmable sample conversion sequencers from one to eight entries long, with corresponding conversion result FIFOs
- Flexible trigger control
- Efficient transfers using Micro Direct Memory Access Controller (µDMA)

ADC Sample Sequencers

- Rather than a “one and done” conversion, the Tiva performs a sequence of conversions called a “sample sequence”.

- Each sample sequence is a fully programmed series of consecutive samples, allowing the ADC to collect data from multiple input sources without having to be re-configured or serviced by the processor.

- The programming of each sample includes parameters such as the input source and mode, interrupt generation and the indicator for the last sample in the sequence.

Other ADC Characteristics

- Interrupt Generation
  - Can alert the processor when conversions are done.
- DMA Integration
  - Can move the data to memory without processor intervention.
- Programmable Priority
  - There are four sample sequencers that have programmable adjustable priorities.
- Sampling Triggers
  - What causes a sequence to start.
- Hardware Averaging
  - Can automatically average up to 64 samples
- Differential Sampling
  - Instead of sampling against a reference voltage, it compares the difference between two inputs.

ADC Initialization

- The fairly straightforward initialization sequence for the ADC is as follows:
- 1. Enable the ADC clock using the `RCGCADC` register.
- 2. Enable the clock to the appropriate GPIO modules via the `RCGCGPIO` register.
- 3. Set the `GPIO AFSEL` bits for the ADC input pins.
- 4. Configure the `AINx` signals to be analog inputs by clearing the corresponding `DEN` bit in the GPIO Digital Enable (`GPIODEN`) register.
- 5. Disable the analog isolation circuit for all ADC input pins that are to be used by writing a `1` to the appropriate bits of the `GPIOAMSEL` register in the associated GPIO block.
- 6. If required by the application, reconfigure the sample sequencer priorities in the `ADCSSPRI` register. The default configuration has Sample Sequencer 0 with the highest priority and Sample Sequencer 3 as the lowest priority.

Sequencer Configuration

- The configuration for each sample sequencer should be as follows:
- 1. Ensure that the sample sequencer is *disabled* by clearing the corresponding `ASEN`*n* bit in the `ADCACTSS` register.
- 2. Configure the *trigger event* for the sample sequencer in the `ADCEMUX` register.
- 3. For each sample in the sample sequence, configure the corresponding input source in the `ADCSSMUXn` register.
- 4. For each sample in the sample sequence, configure the sample control bits in the corresponding nibble in the `ADCSSCTL`*n* register. When programming the last nibble, ensure that the `END` bit is set.
- 5. If interrupts are to be used, set the corresponding `MASK` bit in the `ADCIM` register.
- 6. Enable the sample sequencer logic by setting the corresponding `ASEN`*n* bit in the `ADCACTSS` register.

ADC Registers

- Let’s do a full initialization and configuration of an ADC sequencer to show how this works…
  - …examining each register as we go.
- Note that the TivaWare libraries make this *way* easier.

But you don’t always have helpful libraries like TivaWare.

ADC Clock

Analog-to-Digital Converter Run Mode Clock Gating Control (RCGCADC)

Base 0x400F.E000
Offset 0x638
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved														R1	R0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Writing a 1 to either of these bits turns on the clock.

Initialization Code:

```
SYSCTL_RCGCADC_R |= 0x01;
```

GPIO Clock

General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)

Base 0x400F.E000
Offset 0x608
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved										R5	R4	R3	R2	R1	R0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

This is the clock bit for Port E.

On the Tiva, ADC inputs share pins with GPIO Port E.

Initialization Code:

```
SYSCTL_RCGCGPIO_R |= 0x10;
```

Alternate Function Select

GPIO Alternate Function Select (GPIOAFSEL)

GPIO Port A (APB) base: 0x4000.4000
GPIO Port A (AHB) base: 0x4005.8000
GPIO Port B (APB) base: 0x4000.5000
GPIO Port B (AHB) base: 0x4005.9000
GPIO Port C (APB) base: 0x4000.6000
GPIO Port C (AHB) base: 0x4005.A000
GPIO Port D (APB) base: 0x4000.7000
GPIO Port D (AHB) base: [illegible]
GPIO Port E (APB) base: 0x4002.4000
GPIO Port E (AHB) base: 0x4005.C000
GPIO Port F (APB) base: [illegible]
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x420
Type RW, reset -

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	[illegible]	[illegible]	[illegible]	[illegible]	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	[illegible]	[illegible]	[illegible]	[illegible]	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								AFSEL							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-

The particular bits depends on the analog input you're using.

Initialization Code:

```
GPIO_PORTE_AFSEL_R |= 0x10;
```

This works for Ain9 and is NOT a general solution.

ADC Digital Enable

GPIO Digital Enable (GPIODEN)

GPIO Port A (APB) base: 0x4000.4000
GPIO Port A (AHB) base: 0x4005.8000
GPIO Port B (APB) base: 0x4000.5000
GPIO Port B (AHB) base: 0x4005.9000
GPIO Port C (APB) base: 0x4000.6000
GPIO Port C (AHB) base: 0x4005.A000
GPIO Port D (APB) base: 0x4000.7000
GPIO Port D (AHB) base: 0x4005.B000
GPIO Port E (APB) base: 0x4002.4000
GPIO Port E (AHB) base: 0x4005.C000
GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x51C
Type RW, reset -

	31	30	29	28	27	26	25	24	23	22					17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO					RO	RO
Reset	0	0	0	0	0	0	0	0	0	0					0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								DEN							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-

The particular bits depends on the analog input you're using.

Initialization Code:

```
GPIO_PORTE_DEN_R |= 0x00;
```

Remember, DEN means "Digital input ENable". Clearly we are using analog inputs!

Analog Mode Select

GPIO Analog Mode Select (GPIOAMSEL)

GPIO Port A (APB) base: 0x4000.4000
GPIO Port A (AHB) base: 0x4005.8000
GPIO Port B (APB) base: 0x4000.5000
GPIO Port B (AHB) base: 0x4005.9000
GPIO Port C (APB) base: 0x4000.6000
GPIO Port C (AHB) base: 0x4005.A000
GPIO Port D (APB) base: 0x4000.7000
GPIO Port D (AHB) base: 0x4005.B000
GPIO Port E (APB) base: 0x4002.4000
GPIO Port E (AHB) base: 0x4005.C000
GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x528
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	17	16
	reserved											
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0

The particular bits depends on the analog input you're using.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								GPIOAMSEL							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code:

```
GPIO_PORTE_AMSEL_R |= 0x10;
```

This works for Ain9 and is NOT a general solution.

Sequence Priority Select

ADC Sample Sequencer Priority (ADCSSPRI)

ADC0 base: 0x4003.8
ADC1 base: 0x4003.9
Offset 0x020
Type RW, reset 0x00

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	[illegible]	[illegible]	[illegible]	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	[illegible]	[illegible]	[illegible]	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved		SS3		reserved		SS2		reserved		SS1		reserved		SS0	
Type	RO	RO	RW	RW	RO	RO	RW	RW	RO	RO	RW	RW	RO	RO	RW	RW
Reset	0	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0

The higher the value in SS*n*, the lower the priority.

Initialization Code:

```
ADC0_SSPRI_R = 0x0123;
```

Suggested value from textbook. Inverts default priority.

Initialization Done

- So now the ADC is initialized. That was the easy part.
- Configuring the sample sequencer takes a lot more work, because every sequencer is completely programmable.

Disable the Sequencer

We do this during configuration to make sure there isn't an accidental trigger.

ADC Active Sample Sequencer (ADCACTSS)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x000
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															BUSY
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												ASEN3	ASEN2	ASEN1	ASEN0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code:

```
ADC0_ACTSS_R &= ~0x0008;
```

Weird, but safe, clearing.

Choose the Trigger

ADC Event Multiplexer Select (ADCEMUX)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x014
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	EM3				EM2				EM1				EM0			
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We put an appropriate value into the `EMx` field for whatever trigger we want.

Trigger Options

EMx Value	ADC Trigger
`0x0`	Processor (default)
`0x1`	Analog Comparator 0
`0x2`	Analog Comparator 1
`0x3`	Reserved
`0x4`	External (GPIO Pins)
`0x5`	Timer
`0x6`	PWM Generator 0
`0x7`	PWM Generator 1
`0x8`	PWM Generator 2
`0x9`	PWM Generator 3
`0xA–0xE`	Reserved
`0xF`	Always

Choose the Trigger

ADC Event Multiplexer Select (ADCEMUX)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x014
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	EM3				EM2				EM1				EM0			
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code:

```
ADC0_EMUX_R &= ~0xF000;
```

Enables software (processor) trigger for SS3.

Choose the Input Source

ADC Sample Sequence Input Multiplexer Select 0 (ADCSSMUX0)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x040
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	MUX7				MUX6				MUX5				MUX4			
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	MUX3				MUX2				MUX1				MUX0			
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

There are a bunch of these registers (ADCSSMUX*n*).

Initialization Code:

```
ADC0_SSMUX3_R = 0x09;
```

Again, this is an initialization for Ain9.

Configure Sequence Control

ADC Sample Sequence Control 0 (ADCSSCTL0)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x044
Type RW, reset 0x0000.0000

	[illegible]	[illegible]	[illegible]	[illegible]	27	[illegible]	[illegible]	[illegible]	[illegible]	22	21	20	19	18	17	16
	TS7	IE7	END7	D7	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	IE5	END5	D5	TS4	IE4	END4	D4
Type	[illegible]	[illegible]	[illegible]	[illegible]	RW	[illegible]	[illegible]	[illegible]	[illegible]	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	[illegible]	[illegible]	[illegible]	[illegible]	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

There are a bunch of these registers (ADCSSCTL*n*).

See the pattern?

So what do these four bits do?

Sequence Control Bits

- *n*th Sample Temp Sensor Select (`TS`*x*)
  - `0` The input pin specified by the `ADCSSMUX`*n* register is read during the *n*th sample of the sample sequence.
  - `1` The temperature sensor is read during the *n*th sample of the sample sequence.
- *n*th Sample Interrupt Enable (`IE`*x*)
  - `0` The raw interrupt is not asserted to the interrupt controller.
  - `1` The raw interrupt signal (`INR`*n* bit) is asserted at the end of the *n*th sample's conversion.
  - It is legal to have multiple samples within a sequence generate interrupts.

Sequence Control Bits

- *n*th Sample is End of Sequence (`END`*`x`*)
  - `0` Another sample in the sequence is the final sample.
  - `1` The *n*th sample is the last sample of the sequence.
  - It is possible to end the sequence on any sample position. Software *must* set an `END`*`x`* bit *somewhere* within the sequence.
- *n*th Sample Differential Input Select
  - `0` The analog inputs are not differentially sampled.
  - `1` The analog input is differentially sampled.
  - Because the temperature sensor does not have a differential option, this bit must not be set when the `TS`*`x`* bit is set.

Configure Sequence Control

ADC Sample Sequence Control 0 (ADCSSCTL0)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x044
Type RW, reset 0x0000.0000

There are a bunch of these registers (ADCSSCTL*n*).

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	TS7	IE7	END7	D7	TS6	IE6	END6	D6	TS5	IE5	END5	D5	TS4	IE4	END4	D4
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

On On

Initialization Code:

```
ADC0_SSCTL3_R = 0x0006;
```

Sequence Interrupt Enable

ADC Interrupt Mask (ADCIM)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x008
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved												DCONSS3	DCONSS2	DCONSS1	DCONSS0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												MASK3	MASK2	MASK1	MASK0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code:

```
ADC0_IM_R = 0x0000;
```

Enabler the Sequencer

ADC Active Sample Sequencer (ADCACTSS)

ADC0 base: 0x4003.8000
ADC1 base: 0x4003.9000
Offset 0x000
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															BUSY
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												ASEN3	ASEN2	ASEN1	ASEN0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code:

```
ADC0_ACTSS_R |= 0x0008;
```

Important!

- The initializations shown in this lecture are NOT universal solutions.
- You have to pick the specific set of initializations for your application.

TivaWare ADC Functions

- The ADC API is broken into three groups of functions: those that deal with the sample sequencers, those that deal with the processor trigger, and those that deal with interrupt handling.

- The sample sequencers are configured with `ADCSequenceConfigure()` and `ADCSequenceStepConfigure()`. They are enabled and disabled with `ADCSequenceEnable()` and `ADCSequenceDisable()`. The captured data is obtained with `ADCSequenceDataGet()`.

- The processor trigger is generated with `ADCProcessorTrigger()`.

- The interrupt handler for the ADC sample sequencer interrupts are managed with `ADCIntRegister()` and `ADCIntUnregister()`. The sample sequencer interrupt sources are managed with `ADCIntDisable()`, `ADCIntEnable()`, `ADCIntStatus()`, and `ADCIntClear()`.

ADCSequenceConfigure()

- Configures the trigger source and priority of a sample sequence.
- `void ADCSequenceConfigure(uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Trigger, uint32_t ui32Priority)`
  - `ui32Base` is the base address of the ADC module.
    `ui32SequenceNum` is the sample sequence number.
    `ui32Trigger` is the trigger source that initiates the sample sequence; must be one of the `ADC_TRIGGER_*` values.
    `ui32Priority` is the relative priority of the sample sequence with respect to the other sample sequences.
- This function configures the initiation criteria for a sample sequence. Valid sample sequencers range from zero to three. The trigger condition and priority (with respect to other sample sequencer execution) are set.
- We will almost always set the `ui32Trigger` parameter to `ADC_TRIGGER_PROCESSOR` to generate a trigger generated by processor via the `ADCProcessorTrigger()` function.
- The `ui32Priority` parameter is a value between 0 and 3, where 0 represents the highest priority and 3 the lowest. Note that when programming the priority among a set of sample sequences, each must have unique priority; it is up to the caller to guarantee the uniqueness of the priorities.

`ADCSequenceEnable()`

- Enables a sample sequence.
- `void ADCSequenceEnable(uint32_t ui32Base, uint32_t ui32SequenceNum)`
  - `ui32Base` is the base address of the ADC module. `ui32SequenceNum` is the sample sequence number.
- Allows the specified sample sequence to be captured when its trigger is detected.
- Remember: A sample sequence must be configured before it is enabled.

ADCSequenceDisable()

- Disables a sample sequence.
- `void ADCSequenceDisable(uint32_t ui32Base, uint32_t ui32SequenceNum)`
    - `ui32Base` is the base address of the ADC module. `ui32SequenceNum` is the sample sequence number.
- Prevents the specified sample sequence from being captured when its trigger is detected.
- Remember: A sample sequence should be disabled before it is configured.

ADCSequenceStepConfigure()

- Configure a step of the sample sequencer.
- `void ADCSequenceStepConfigure(uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t ui32Step, uint32_t ui32Config)`
    - `ui32Base` is the base address of the ADC module.
      `ui32SequenceNum` is the sample sequence number.
      `ui32Step` is the step to be configured.
      `ui32Config` is the configuration of this step;
    - `ui32Config` must be a logical OR of `ADC_CTL_TS`, `ADC_CTL_IE`, `ADC_CTL_END`, `ADC_CTL_D`, one of the input channel selects (`ADC_CTL_CH0` through `ADC_CTL_CH23`), and one of the digital comparator selects (`ADC_CTL_CMP0` through `ADC_CTL_CMP7`).

`ADCSequenceStepConfigure()`

- Bit Field Meanings:
  - `ADC_CTL_D` — Differential Mode
  - `ADC_CTL_CH0`-`ADC_CTL_CH23` — Select the channel to sample
  - `ADC_CTL_TS` — Use the internal temperature sensor
  - `ADC_CTL_END` — Configure this step to be the last step
  - `ADC_CTL_IE` — Enable the interrupt for this step
- The `ui32Step` parameter determines the order in which the samples are captured by the ADC when the trigger occurs.
  - <u>Valid values:</u>
  - First sequencer — 0-7
  - Second sequencer — 0-3
  - Third sequencer — 0-3
  - Fourth sequencer — 0

`ADCProcessorTrigger()`

- Causes a processor trigger for a sample sequence.
- `ADCProcessorTrigger(uint32_t ui32Base, uint32_t ui32SequenceNum)`
  - `ui32Base` is the base address of the ADC module. `ui32SequenceNum` is the sample sequence number.
- This function triggers a processor-initiated sample sequence if the sample sequence trigger is configured to `ADC_TRIGGER_PROCESSOR`.

ADCSequenceDataGet()

- Gets the captured data for a sample sequence.
- `int32_t ADCSequenceDataGet(uint32_t ui32Base, uint32_t ui32SequenceNum, uint32_t* pui32Buffer)`
  - `ui32Base` is the base address of the ADC module. `ui32SequenceNum` is the sample sequence number. `pui32Buffer` is the address where the data is stored.

Be clear: This is a pointer!

- This function copies data from the specified sample sequencer output FIFO to a memory resident buffer. The number of samples available in the hardware FIFO are copied into the buffer, which is assumed to be large enough to hold that many samples.
- Returns the number of samples copied to the buffer.

ADC Configuration Example

```c
uint32_t ui32Value;
//
// Enable the first sample sequencer to capture the value of chann
// the processor trigger occurs.
//
ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
                         ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
ADCSequenceEnable(ADC0_BASE, 0);
//
// Trigger the sample sequence.
//
ADCProcessorTrigger(ADC0_BASE, 0);
//
// Wait until the sample sequence has completed.
//
while(!ADCIntStatus(ADC0_BASE, 0, false))
{
}
//
// Read the value from the ADC.
//
ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

Configure ADC0, Sequencer 0 to be processor triggered with the highest priority.

ADC Configuration Example

```c
    uint32_t ui32Value;
    //
    // Enable the first sample sequencer to capture the value of channel 0 when
    // the processor trigger occurs.
    //
    ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
                             ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
    ADCSequenceEnable(ADC0_BASE, 0);
    //
    // Trigger the sample sequence.
    //
    ADCProcessorTrigger(ADC0_BASE, 0);
    //
    // Wait until the sample sequence has completed.
    //
    while(!ADCIntStatus(ADC0_BASE, 0, false))
    {
    }
    //
    // Read the value from the ADC.
    //
    ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

ADC0, sequencer 0, sample 0. Enable the interrupt, indicate this is the last sample and read from channel 0.

ADC Configuration Example

```
    uint32_t ui32Value;
    //
    // Enable the first sample sequencer to capture the value of channel 0 when
    // the processor trigger occurs.
    //
    ADCSequenceConfigure(ADC0_BASE, 0, AD
    ADCSequenceStepConfigure(ADC0_BASE, 0
                             ADC_CTL_IE |                            0);
    ADCSequenceEnable(ADC0_BASE, 0);
    //
    // Trigger the sample sequence.
    //
    ADCProcessorTrigger(ADC0_BASE, 0);
    //
    // Wait until the sample sequence has completed.
    //
    while(!ADCIntStatus(ADC0_BASE, 0, false))
    {
    }
    //
    // Read the value from the ADC.
    //
    ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

Enable the sequencer.

ADC Configuration Example

```
    uint32_t ui32Value;
    //
    // Enable the first sample sequencer to capture the value of channel 0 when
    // the processor trigger occurs.
    //
    ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
                             ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
    ADCSequenceEnable(ADC0_BASE, 0);
    //
    // Trigger the sample sequence.
    //
    ADCProcessorTrigger(ADC0_BASE, 0);
    //
    // Wait until the sample sequence has comp
    //
    while(!ADCIntStatus(ADC0_BASE, 0, false))
    {
    }
    //
    // Read the value from the ADC.
    //
    ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

Trigger the conversion.

ADC Configuration Example

```
uint32_t ui32Value;
//
// Enable the first sample sequencer to capture the value of channel 0 when
// the processor trigger occurs.
//
ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
                         ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
ADCSequenceEnable(ADC0_BASE, 0);
//
// Trigger the sample sequence.
//
ADCProcessorTrigger(ADC0_BASE, 0);
//
// Wait until the sample sequence has complete
//
while(!ADCIntStatus(ADC0_BASE, 0, false))
{
}
//
// Read the value from the ADC.
//
ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

Wait for the conversion to complete.

ADC Configuration Example

```c
uint32_t ui32Value;
//
// Enable the first sample sequencer to capture the value of channel 0 when
// the processor trigger occurs.
//
ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_PROCESSOR, 0);
ADCSequenceStepConfigure(ADC0_BASE, 0, 0,
                         ADC_CTL_IE | ADC_CTL_END | ADC_CTL_CH0);
ADCSequenceEnable(ADC0_BASE, 0);
//
// Trigger the sample sequence.
//
ADCProcessorTrigger(ADC0_BASE, 0);
//
// Wait until the sample sequence has completed.
//
while(!ADCIntStatus(ADC0_BASE, 0, false))
{
}
//
// Read the value from the ADC.
//
ADCSequenceDataGet(ADC0_BASE, 0, &ui32Value);
```

Get the result.

Definitely note how the data is collected.
We pass the address of `ui32Value` using the & operator.

Summary

- ATD is hard, expensive and necessary.
- The Tiva C has two 12-bit, 8-channel ATD converter units (at least most do).
- Remember, resolution = $(V_{RH}-V_{RL})/2^b$.