Module 9. Analog to Digital Conversion

Topics

- More on timers
- Starting Analog to Digital Conversion

Warmup: Analyzing a timer configuration

1. What is the period of the timer with the configuration below? (How often are interrupts generated?)

```
void runtimerA2(void)
{
    TA2CTL = TASSEL_2 | MC_1 | ID_3;
    TA2CCR0 = 32767;
    TA2CCTL0 = CCIE; // Enable timer A2 interrupt
}
```

2. The ISR for the timer above increments a counter called `timer` on each interrupt. If `timer = 2447`, how much time has elapsed since the timer was started?

Analog to Digital Converters (ADCs)

Analog to Digital Converters (ADCs), or A/D converters, have become ubiquitous in embedded applications.

- ADCs return a binary code to represent a measured voltage from within a fixed range of voltages
- Small voltages return "low valued" codes, greater voltages return "larger" codes

For example, within the range of 0–3V, a 10-bit ADC could return codes like these:

```
00 0000 0000b = 000h = 0d
01 1111 1111b = 1FFh = 512d
11 1111 1111b = 3FFh = 1023d
```

- It is **very** likely that you will use an ADC when you take ECE2799, do your MQP, or work on a robotics project!

ADC Concepts

When an analog signal is “read” by an ADC, the analog value of the signal is *sampled* to obtain a k-bit digital representation of that value at that time.

Realities of ADCs

No ADC is perfect–there will always be some error between the analog voltage and the one measured with the ADC. There are several reasons for this, including:

- Output codes are "quantized": the closest ADC code will differ somewhat from the analog voltage, depending on the resolution
- Our MSP430 uses a "sample and hold" type of ADC, which means the analog circuitry that samples the waveform can "hold" the analog value at a certain level—this means that it might miss certain changes in the waveform
- Transients from switching circuitry inside the ADC can affect the output code, which may introduce some non-linearity in the output values

When sampling at faster rates, these effects tend to get worse!

We will not deal with these issues much in this class, but it is important to know they exist.

ADCs on Microcontrollers

Small ADCs like the 12-bit "Sample and Hold" type ADC on our MSP430 often come standard on small microcontrollers.
Are they any good?

These ADCs are best suited to measuring from sensors with low to moderate data rates with a *fixed dynamic range*. Some examples:

These small ADCs are likely not suitable for applications with higher data rates or a larger dynamic range. Examples:

As always, however, the application will determine the type of ADC you need!
What is *dynamic range*, anyway?

Key Concepts for using Analog to Digital Converters
(or performing any measurements)

1. Full Scale Range (FSR): The maximum range of analog values that can be represented
This is defined as the total range of voltages between V_{REF+} and V_{REF-}

2. Resolution (for a single bit): The smallest change in value that can be measured
You can think of this as the "value of 1 bit" in an output code.

3. Dynamic range: Ratio of largest to smallest values that can be measured
The dynamic range is usually expressed in decibels (dB), and can be computed as follows:

Thinking about data representations

As an embedded systems engineer, you get to decide how to make your sensors interface with the ADC! Knowing how your external sensor works and how to "map" it to the ADC you're using is as critical as knowing how to make the MSP430 read the value!
Here’s a way to think about how sensor measurements are represented as digital values:

Example: Current sensor

You can make a simple digital current meter by measuring the voltage across a small sensing resistor.
Can we use the ADC12 on the MSP430 to measure current in the range 0–1A with 1mA accuracy? How about to 0.1mA accuracy?

For now, let's assume we have an FSR of 2.5V.

On the MSP430: Using the ADC12

Our MSP430 provides includes a 12-bit ADC, called the ADC12.

About the ADC12

- 16 channel, 12-bit sample-and-hold ADC
- Maximum sample rate of 200k samples/second
- 12 External analog inputs A0-A7, A12-A15; shared with Digital I/O ports 6 and 7
- You configure and use them by setting values in various control registers

Overall ADC operation

An ADC's job is to perform a *conversion* by sampling an analog voltage into a digital value.

The ADC12 has the following components:

- Inputs from analog input channels
- Core unit to perform conversions
- *Core configuration registers* that configure how the conversion happens
- Can define multiple *channels* to perform multiple conversions at once
    - *Memory control registers* that configure how each channel should be converted
    - *Memory registers* that store the conversion results for each channel

ADC12 Control and Data registers

You can find the ADC12 register definitions in the MSP430 User's Guide (Ch. 28).

Core configuration registers

The ADC12 conversion core is configured using ADC12CTL0 and ADC12CTL1.

ADC12CTL0 controls the following options:

- Sample and Hold Time (ADC12SHT1x, ADC12SHT0x): Controls sampling period
- Multiple sample conversion method (ADC12MSC)
- **Reference voltages (ADC12REF2_5V and ADC12REF_ON)**
- **ADC12ON bit**: Turns on the ADC12! (It's off by default!)
- **Enable conversions (ADC12ENC)**: Must be set to 1 before ADC will perform conversions! When set to 0, ADC can be configured.
- **Start conversion (ADC12SC)**: Starts a conversion!
- Overflow/conversion time interrupt enables (ADC12OVIE, ADC12TVIE)

ADC12CTL1 controls the following options:

- Conversion start address (ADC12STARTADDx)
- Sample and hold source select (ADC12SHS):
- Sample and hold pulse mode select (ADC12SHP): Always set this to 1
- Invert signal sample and hold (ADC12ISSH)
- ADC12 clock divider (ADC12DIVx): Typically use 1
- ADC12 clock source select (ADC12SSELx):
- Conversion mode select (ADC12CONSEQx): Can select single, multi-channel, or repeated conversions
- **ADC12 busy bit (ADC12BUSY)**

Results from each channel are stored in the low 12 bits of one 16 bit **Conversion Memory Register (ADC12MEMx)**.

Each memory register has a corresponding **Conversion Memory Control Register (ADC12MCTLx)**.

Each **ADC12MCTLx** controls one channel on which a conversion can occur. The conversion parameters for channel x is controlled by Memory Control Register x, and the result gets stored in memory register x.

Each ADC12MCTLx controls the following options:

- **Reference voltage select (ADC12SREFx)**: Important settings are as follows:

- **Analog input channel select (ADC12INCH_x)**:

- **End of Sequence (EOS)**: Set to 1 if this channel is the end of a sequence of channels. Used for multi-channel conversions.

So, as a programmer, what do you need to use the ADC12?

ADC configuration: Key steps

Step 0: Disable the ADC for configuration

- Before you can modify any ADC12 register settings, conversions must be disabled by setting ADC12ENC = 0.

Step 1: Select ADC core behavior (ADC12CTL0 and ADC12CTL1)

- Set clock source and divider
- Configure sample and hold behavior
- Select trigger source (ADC12SHS)

- **Reference voltages**

Step 2: Select conversion mode for your application

- Configure using ADC12CONSEQx in ADC12CTL1 register
- There are four possible conversion modes:

Table 28-2. Conversion Mode Summary

ADC12CONSEQx	Mode	Operation
00	Single-channel single-conversion	A single channel is converted once.
01	Sequence-of-channels (autoscan)	A sequence of channels is converted once.
10	Repeat-single-channel	A single channel is converted repeatedly.
11	Repeat-sequence-of-channels (repeated autoscan)	A sequence of channels is converted repeatedly.

Step 3: Select input channel(s)

- What analog inputs do we need to read?
- Configure using ADC12INCHx in ADC12MCTLx registers

- Analog inputs A0–A7 and A12–A15 are external analog inputs–these are **multiplexed** with Digital I/O pins on Port 6 and Port 7!
    - To use them, we need to configure the digital I/O pins for **function mode**!
      Ex. P6SEL | BIT7|BIT6;

- Analog inputs 8, 9, 11 are connected to the various on-chip reference voltages–you can use these to monitor the "health" of the microcontroller
- Input channel 10 is connected to an internal temperature sensor (ADC12INCH_10)

Step 4: Enable ADC interrupts, if desired (ADC12IE register)

- Using interrupts is NOT required, but useful if you are doing repeated measurements
- Also need to write ISR

Step 5: Enable ADC and start conversions

- Need to re-enable ADC so it will perform conversions (opposite of step 0)
- Start conversion process by setting ADC12SC.
- If not using interrupts, need to **poll** ADC12BUSY bit in ADC12CTL1 until conversion has finished!

Example: Current measurement sensor

You can make a simple digital current meter by measuring the voltage across a small sensing resistor. Can we use the ADC12 on the MSP430 to measure current in the range 0–1A with 1mA accuracy? (Yes!) How about to 0.1mA accuracy? (No!)
Assume we have an FSR of 2.5V and the analog voltage is connected to input A0.

What parameters do we need?

```
// Current sensor conversion example
void config_adc(void) {
  /* ***** Core configuration ***** */

  // Reset REFMSTR to enable control of reference voltages by ADC12
  REFCTL0 &= ~REFMSTR;

  /*
   * Initialize control register ADC12CTL0
   * STH0x   = 9 => 384 clock cycles; MSC = 0 => no multisample mode
   * REF2_5V = 1 => Reference is 2.5V, REFON = 1 => Use internel reference generator
   * ADC12ON = 1 => Turn on ADC12
   */
  ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12REF2_5V | ADC12ON;

  /*
   * Initialize control register ADC12CTL1
   * STARTADDx = 0 => Start conversion at ADC12MEM0
   * SHSx      = 0 => Conversion trigger:  Start when ADC12SC is set to 1
   * SHP       = 1 => SAMPCON sourced from sampling timer (default)
   * ISSH      = 0 => Input signal not inverted
   * SSEL      = 0 => ADC12clock = ADC12OSC (~5 MHz)
   * DIVx      = 0 => Divide ADC12CLK by 1
   * CONSEQx   = 0 => Single channel, single conversion mode
   */
  ADC12CTL1 = ADC12SHP;

  /* ***** Channel configuration ***** */

  // Set conversion memory control register ADC12MCTL0
  // SREF = 001b => Voltage refs:
  // EOS  =   0  => End of sequence not set (not a multi-channel conversion, so ig-
  nore)
  ADC12MCTL0 = ADC12SREF_1 | ADC12INCH_0;

  // Set P6.0 to FUNCTION mode
  // This connects the physical pin P6.0/A0 to the ADC input A0
  P6SEL |= BIT0;

  // Enable the ADC.  This means we are done configuring it,
  // so we can start the conversion.
  ADC12CTL0 |= ADC12ENC;
}
```

```
unsigned int read_adc(void) {
  // Input voltage has range 0-2.5V, which corresponds to 0 to 1A.
  unsigned int in_value;

  ADC12CTL0 &= ~ADC12SC;
  // Enable and start a single conversion
  ADC12CTL0 |= ADC12SC;


  // Wait for the conversion to finish by polling the busy bit
  // The busy bit is automatically set to 0 when the conversion is done
  while(ADC12CTL1 & ADC12BUSY) {
      __no_operation(); // Could also just leave the loop empty
  }

  // Now that the conversion has completed, we can read the result
  // from the memory register
  in_value = ADC12MEM0 & 0x0FFF; // Keep only the low 12 bits
  return in_value;
}
```

Now what do we do with the return value?

Example: The internal temperature sensor

To use any sensor, you need to understand how the sensor output (in this case, voltage) corresponds to the quantity it measures, which is documented by the designers.

Our MSP430 contains a built-in sensor to measure the internal chip temperature. It has a linear mapping from voltage to temperature:

A typical temperature sensor transfer function is shown in Figure 28-11 . The transfer function shown in Figure 28-11 is only an example—the device-specific data sheet contains the actual parameters for a given device. When using the temperature sensor, the sample period must be greater than 30 µs. The temperature sensor offset error can be large and may need to be calibrated for most applications. Temperature calibration values are available for use in the TLV descriptors (see the device-specific data sheet for locations).

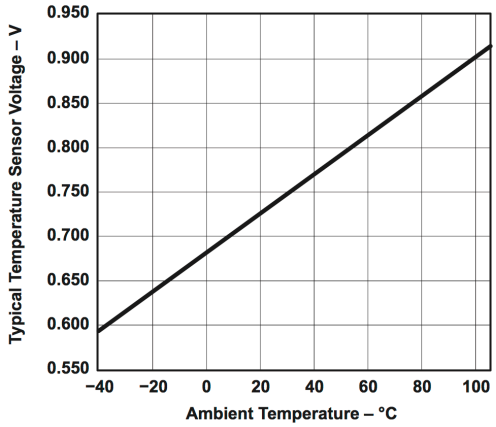

Figure 28-11. Typical Temperature Sensor Transfer Function

To use the sensor, we need to read some *calibration* data from the device, which we use in the computation for the resolution. This information is stored in the *Tag-Length-Value Table (TLV Table),* which is a portion of flash memory that contains some device-specific settings and constants—we need to read from the addresses specified in this table to get the calibration data. For more information on how this works, see p. 102 of the datasheet.

According to the datasheet, the calibration data provided is based on a 1.5V reference.

More ADC features

Multi-channel conversion

What if we wanted to read data from two sensors? Consider the following sensors:

- Our current sensor example from earlier (connected to input channel A0, 2.5V reference)
- A barometric pressure sensor (Input channel A4, 3.3V reference)

These sensors require different settings for reference voltages and inputs.
We *could* reconfigure the ADC every time we wanted to take a measurement, but this would be annoying. Instead, the ADC12 provides several different conversion modes to solve this problem. We will discuss the most straightforward: *multiple channel, single conversion* mode.

To perform readings from two sensors, we will need to use **two** ADC12MEMx registers, one for each channel.

Like the previous examples, we need to:

- Configure the ADC12 core, this time selecting *multiple-channel, single conversion* mode
- Configure one ADC12MEMx register for each reading we want to perform with the appropriate settings for each channel (ie, analog channel and reference voltage)

Configuration example

```
// Multi-sample conversion example:  current sensor + pressure sensor
// Input voltage has range 0-2.5V, which corresponds to 0 to 1A.
// The temperature sensor uses a 3.3V reference, using the following calibration
data


// Resolution for current sensor:  1.0A/4096
// (Here, we add the f suffix so the compiler will know this constant is a float)
#define MA_PER_BIT        (0.0244f)

void config_adc(void)
{
  unsigned int in_current, in_temp;
  /* ***** Core configuration ***** */

  // Reset REFMSTR to enable control of reference voltages
  REFCTL0 &= ~REFMSTR;

  /*
   * Initialize control register ADC12CTL0
   * This is similar to single-channel conversion, except we add the following:
   * MSC = 1 => Burst mode (convert multiple channels) * <- *NEW*
   * REF2_5V = 1 => Reference is 2.5V
   */
  ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12REF2_5V | ADC12ON | ADC12MSC;
  //                                                             ^----NEW!

  /*
  * Initialize control register ADC12CTL1
  * This is similar to the previous example,
  * except we need to consider the following:
  * STARTADDx = 0     => Start conversion at ADC12MEM0
  * CONSEQx   = **1** => ***Sequence of channels, converted once***
  */
  ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;
  //                       ^------NEW!

  /* ***** Channel configuration ***** */
  /* Here, we pick one MCTLx register per reading we need
   * Let's pick MCTL0 for the current sensor, and MCTL1 for the pressure sensor
   */

  // Current sensor:  Input channel A0, SREF_1 => Internal ref => 2.5V
  ADC12MCTL0 = ADC12SREF_1 + ADC12INCH_0;

  // Pressure sensor:  Input channel A4, SREF_0 => 3.3V Reference, ADC12EOS = 1
  ADC12MCTL1 = ADC12SREF_0 + ADC12INCH_4 + ADC12EOS;
  //                                       ^- NEW!
  // For the last channel in our sequence, we set the ADC12EOS bit.  This tells
  // the ADC12 that our sequence is finished!

  ADC12CTL0 |= ADC12ENC; // Done configuring, enable conversion
```

```
  /* ***** I/O configuration ***** */
  // Set P6.0 and P6.4 to FUNCTION mode
  // This connects the physical pins P6.0/A0 P6.4/A4 to the ADC inputs A0 and A4
  P6SEL |= BIT4|BIT0;
}

void read_adc(void)
{
  // Enable the ADC and start the conversion
  ADC12CTL0 |= ADC12ENC;
  ADC12CTL0 &= ~ADC12SC;
  ADC12CTL0 |= ADC12SC;

  // Wait for the conversion to finish by polling the busy bit
  while(ADC12CTL1 & ADC12BUSY) {
      __no_operation();
  }

  // Now that the conversion has completed, we can read the results
  // from the memory registers
  in_current = ADC12MEM0 & 0x0FFF; // Keep only the low 12 bits
  in_pressure    = ADC12MEM1 & 0x0FFF;

  // Finally, we would convert the results to current (amps)
  // and pressure (atm, kPa, mmHg. ...), respectively, and store them somewhere
}
```

ADC12 Interrupts

Our examples so far have used the `ADC12SC` bit to start conversions, and then we poll the `ADC12BUSY` bit to see when a conversion is complete:

```
ADC12CTL0 |= ADC12SC;       // Start the conversion

// Poll busy bit waiting for conversion to complete
while (ADC12CTL1 & ADC12BUSY) {
    __no_operation();       // Could just leave body of loop empty
}

in_value = ADC12MEM0 & 0x0FFF; // Read result
```

This is another form of *busy-waiting*, which is like `swDelay`:

- The CPU isn't doing any useful work–it's just sitting in a loop!
- While ADC conversions happen very quickly (<< 1 ms), the CPU executes faster, so we do need to wait for a result

The main purpose of on-chip peripherals like the Timer and ADC is to remove burdens from or provide services to the CPU.

How useful is a peripheral if the CPU is just waiting for it to finish?

How do we use interrupts on the ADC12?

- Configure control registers same as before (as if you were polling)
- Set the ADC12IE register bit corresponding to the **last** MEMx register used in the conversion. (If you are only doing single-channel conversions, this is ADC12MEM0.)

Here is an example configuration with two channels:

```
ADC12CTL0 = ADC12SHT0_9 | ADC12REFON | ADC12ON | ADC12MSC;
ADC12CTL1 = ADC12SHP | ADC12CONSEQ_1;

// Here, we are performing conversions for two channels
ADC12MCTL0 = ADC12SREF_0 + ADC12INCH_5;
ADC12MCTL1 = ADC12SREF_1 + ADC12INCH_6 + ADC12EOS;

// Because we are converting for two channels, we want the interrupt
// to occur after BOTH conversions are complete, so we enable the
// interrupt for MEM1.
ADC12IE = BIT1;

__enable_interrupt(); // Globally enable interrupts

ADC12CTL0 |= ADC12SC + ADC12ENC; // Enable ADC and start conversion
```

What should the ISR do? It's triggered when a conversion is finished, so it just needs to read the memory registers!

```
// Global variables for storing data
// (could also store into an array!)
volatile unsigned int in_value1, in_value2;

#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{




}
```

Timer-triggered ADC Measurements

It's also possible to have the ADC perform conversions automatically. An easier option is to trigger ADC conversions from a timer and use ADC interrupts to read the results.

For example, say we wanted to take measurements every 0.25 seconds:

```
void config_timerA2(void)
{




}
```

Inside the timer ISR, start the ADC12 conversion:

```
// NOTE:  this example assumes the ADC has already been configured.

// Global variables for storing data (could also store into an array!)
volatile unsigned int value1, value2;

// Timer A2 ISR
#pragma vector=TIMER2_A0_VECTOR
__interrupt void Timer_A2_ISR(void)
{
    timer++; // You probably still want to keep track of time


    ADC12CTL0 |= ADC12SC;
}

// ADC 12 ISR
#pragma vector=ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{

    // Move the result(s) into global variables
    value1 = ADC12MEM0 & 0x0FFF;
    // . . .
}

void main(void) {
    do_all_setup();
    while(1) {
        display(value1);
    }
}
```

Aside: Data logging

A common practice when working with sensors is to record past data. Since memory is finite resource, it is common to only keep the last N values.
For example: if we are building a sensor that logs temperature and wind speed, we may measure data every second and keep only the results from the last hour.

An efficient way to keep track of the "last N values" in an array is by using *circular indexing.* When used this way, the array is often called a *circular buffer*.

Circular Indexing: How it works

- Use a fixed-size array ("buffer") to store values
- Keep track of the *tail*, which is the "end" index of data in the array (also called the "write pointer")
  Using this definition, the tail always represents the location of the oldest element, which is also the next index to be filled with new data
- When adding a new element, add 1 to the index.
  When the index reaches the end of the array, wrap around to the beginning

How do we implement this?

For more information

- See "Data Logging Example" on the course website
- The Wikipedia entry on Circular Buffer has great animations!