LUMINARY MICRO®

Using the Stellaris® Microcontroller Analog-to-Digital Converter (ADC)

APPLICATION NOTE

AN01247-02

Legal Disclaimers and Trademark Information

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Table of Contents

Introduction

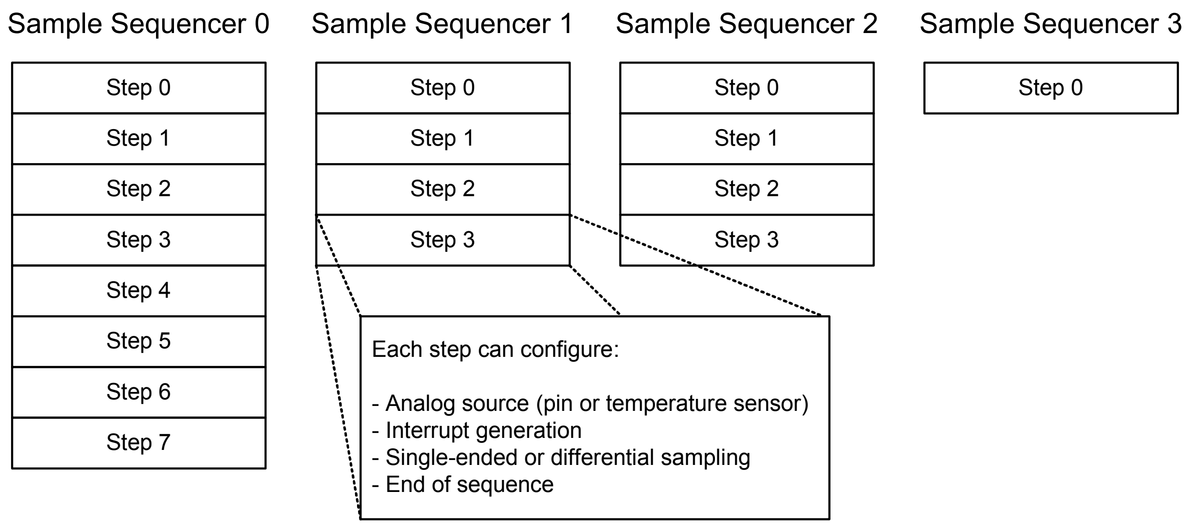
Luminary Micro Stellaris® microcontrollers that are equipped with an analog-to-digital converter (ADC), use an innovative sequence-based sampling architecture designed to be extremely flexible, yet easy to use. This application note describes the sampling architecture of the ADC. Since programmers can configure Stellaris microcontrollers either through the powerful Stellaris Family Driver Library or through direct writes to the device's control registers, this application note describes both methods. The information presented in this document is intended to complement the ADC chapter of the device datasheet, and assumes the reader has a basic understanding of how ADCs function.

Sample Sequencers

Most analog-to-digital converter implementations in 8-,16-, and 32-bit microcontrollers require processor intervention to configure each conversion when the analog input/channel is changed. Luminary Micro's sequence-based architecture gives software the ability to enable up to four separate sample sequences (encompassing all of the analog input channels) with a single series of configuration writes.

The ADC module has a total of four sample sequencers that allow sampling of 1, 4 (there are two 4-beat sequencers), or 8 analog sources with a single-trigger event (see Figure 1). Each sample sequencer has its own set of configuration registers making it completely independent from the other sequencers. All steps in a sample sequence are configurable, allowing software to select the analog input channel (including the temperature sensor), single-ended or differential mode sampling, and whether or not to generate an interrupt after the step completes. The sample sequences also have configurable priority to handle cases where multiple sequences are triggered by the same trigger source or trigger simultaneously.

Figure 1. Sample Sequencer Structure

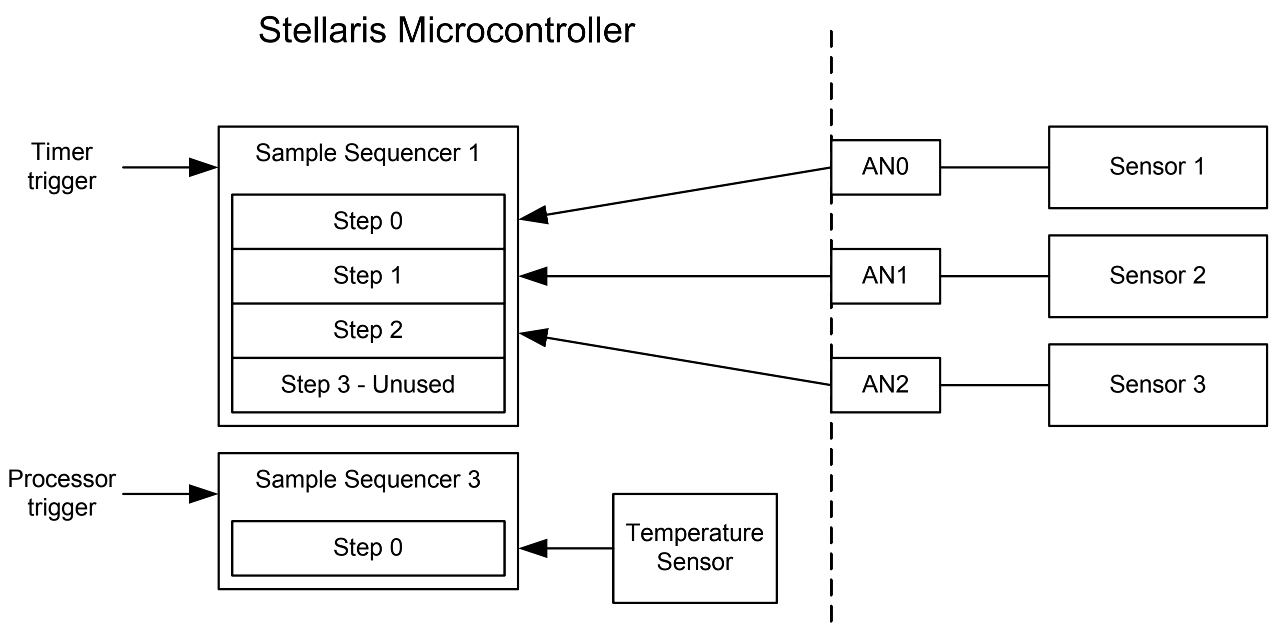


A sample sequence can be triggered by various sources, including the processor, timers, analog comparators, PWM unit or GPIO. For situations where multiple sequences have the same trigger source or are triggered simultaneously, the ADC arbitrates execution order based on the configured sequence priorities. When a sample sequence is triggered, it begins sampling at the programmed sampling rate (250K, 500K, or 1M samples/second, depending on the device), iterating through the steps of the sequence. Sampling continues until a step has its END bit set, indicating the end of the sequence. The END bit can be set for any step in the sequence, meaning a given sample sequence is not required to collect its maximum number of samples. When the sample sequence completes, the conversion results are stored in the sample sequence FIFO, and can be retrieved by the processor.

Module Configuration Example

To demonstrate the steps required to configure the ADC, consider the example shown in Figure 2. There are a total of three sensors being monitored, in addition to the on-chip temperature sensor. Since three analog inputs are used, this example assumes the specific Stellaris device has at least three analog inputs. Each sample sequence has its own FIFO with the number of slots equivalent to the size of the sequencer.

Figure 2. Example System Configuration



Notice how the analog inputs are mapped to the individual steps in the sample sequencer. Since there are three sensor inputs to monitor, one of the four step sequencers (in this case, sample sequence 1) is used. The temperature sensor is sampled using sample sequence 3 since it requires only one step and has a separate trigger source. If the temperature sensor was configured to have a timer trigger, it could be placed in the unused step of sequence 1.

Note: All code examples in the following sections show both direct register writes/reads and API calls to the Stellaris® Peripheral Driver Library. If attempting to reproduce the direct register access examples, the appropriate IC header file, for example “lm3s811.h” for an LM3S811 part, must be included. These header files, one for each member of the Stellaris family, can be found in the StellarisWare/inc directory in the installed software tree. To make use of the Stellaris Peripheral Driver Library instead of direct register access, header files hw_types.h, hw_memmap.h, and adc.h are required. These can be found in the StellarisWare and StellarisWare/DriverLib directories.

Module Initialization

Out of reset, all peripheral clocks are disabled to reduce power consumption and must be enabled for each peripheral module. Enabling a peripheral's clock is a simple task, requiring a write to one of the **Run-Mode Clock Gating Control (RCGCx)** registers in the System Control module. To enable the clock to the ADC, write a '1' to bit 16 (the `ADC` bit) of the **RCGC0** register (address 0x400FE100).

Table 1. Enabling the ADC Clock

Using Direct Register Write

```
//
// Enable the clock to the ADC module
//
// System Control RCGC0 register
//
SYSCTL_RCGC0_R |= SYSCTL_RCGC0_ADC;
```

Using DriverLib

```
//
// Enable the clock to the ADC module
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC);
```

Another aspect of the ADC that must be configured by the programmer is the sample rate. The default sample rate following a reset is 125K samples/second. Depending on the device, the ADC sample rate can be set to 250K, 500K, or 1M samples/second by writing bits 8-11 of the **Run-Mode Clock Gating Control 0 (RCGC0)** register. Table 3 shows the valid data range for the bit field and the available sample rates. You cannot set the sample rate higher than the maximum rate for the device. If software attempts to configure the ADC to a sample rate that is not supported by the device, the device remains at either the default value or the most-recently programmed value. Table 2 shows how to configure the ADC to sample at 500K samples/second.

Table 2. Setting the ADC Sample Rate

Using Direct Register Write

```
//
// Configure the ADC to sample at 500KSps
//
// System Control RCGC0 register
//
SYSCTL_RCGC0_R |= SYSCTL_RCGC0_ADCSPD500K;
```

Using DriverLib

```
//
// Configure the ADC to sample at 500KSps
//
SysCtlADCSpeedSet(SYSCTL_SET0_ADCSPEED_500KSPS)
```

Table 3. Valid ADC Sample Rates

Value	Sample Rate	Value	Sample Rate
0x0	125K samples/second	0x2	500K samples/second
0x1	250K samples/second	0x3	1M samples/second

Sample Sequence Configuration

Before changing configuration parameters in the ADC, it is a good practice to disable the targeted sample sequence. Disabling a sample sequence allows for safe modification of the configuration parameters without inadvertent triggers. To disable one or more sequences, set the corresponding bits in the **ADC Active Sample Sequencer (ADCACTSS)** register to '0'. For this example, sequences 1 and 3 should be disabled.

Table 4. Disabling the Sample Sequences

Using Direct Register Write

```
//
// Disable sample sequences 1 and 3
//
// ADC Active Sample Sequencer register
//
ADC_ACTSS_R &= ~(ADC_ACTSS_ASEN1 | ADC_ACTSS_ASEN3);
```

Using DriverLib

```
//
// Disable sample sequences 1 and 3
//
ADCSequenceDisable(ADC_BASE, 1);
ADCSequenceDisable(ADC_BASE, 3);
```

With the sequences disabled, it is now safe to load the new configuration parameters. Configure the priority of the sample sequences first. In a situation where multiple ADC triggers are active simultaneously, or multiple sequences are triggered by the same source, the ADC control logic has to decide which sample sequence runs first. Out of reset, the sample sequences are prioritized based on their number, meaning sequence 0 has the highest priority and sequence 3 has the lowest priority (the register bit fields range from 0-3, with 0 being the highest priority and 3 being the lowest). This example does not have a particular priority requirement, so sequence 3 is configured to have the highest priority.

In addition to the priority, the ADC trigger sources must be configured. The ADC offers a wide variety of trigger sources including the processor, analog comparators (if available), GPIO, PWM, and timers, but this example requires sequence 1 to have a timer trigger, and sequence 3 to have a processor trigger.

Table 5. Configuring the Sequence Priority and Trigger

Using Direct Register Write

```
//
// Configure sequence priority: order (highest to lowest)= 3, 1, 0, 2
//
// ADC Sample Sequencer Priority register
//
ADC_SSPRI_R = (ADC_SSPRI_SS3_1ST | ADC_SSPRI_SS1_2ND |
               ADC_SSPRI_SS0_3RD | ADC_SSPRI_SS2_4TH);

//
// Set up sequence triggers: sequence 1 = timer (encoding 0x5),
// sequence 3 = Processor (encoding 0x0)
//
// ADC Event Multiplexer Select register
//
ADC_EMUX_R = (ADC_EMUX_EM1_TIMER | ADC_EMUX_EM3_PROCESSOR);
```

Using DriverLib

```
//
// Configure sample sequence 1: timer trigger, priority = 1
//
ADCSequenceConfigure(ADC_BASE, 1, ADC_TRIGGER_TIMER, 1);

//
// Configure sample sequence 3: processor trigger, priority = 0
//
ADCSequenceConfigure(ADC_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
```

The next step in the configuration process is setting up the sequence steps. There are two registers that are responsible for the individual steps of the sample sequence: the **ADC Sample Sequence Input Multiplexer Select (ADCSSMUX)** and **ADC Sample Sequence Control (ADCSSCTL)** registers.

The **ADCSSMUX** register allows software to select the analog input source for each step in the sequence, and a step can select any one of the analog inputs. If a sequence step is sampling the internal temperature sensor, the corresponding value in the **ADCSSMUX** register is ignored by the hardware.

Control information such as sampling mode (single-ended or differential), temperature sensor sampling, interrupts, and end-of-sequence is stored in the **ADCSSCTL** register. Each step in a sample sequence has its own 4-bit nibble, allowing software to set any of the aforementioned configuration options. The first step in the sequence occupies the least significant nibble in the register, and so on. It is software's responsibility to set the `END` bit for the last step of a sequence. If the `END` bit is not set for a sequence, unpredictable behavior can occur.

Figure 3 shows the layout of the configuration nibble. The “n” next to each field is associated with the step number; for step 3, n is 3.

Figure 3. Sequence Step Configuration Nibble

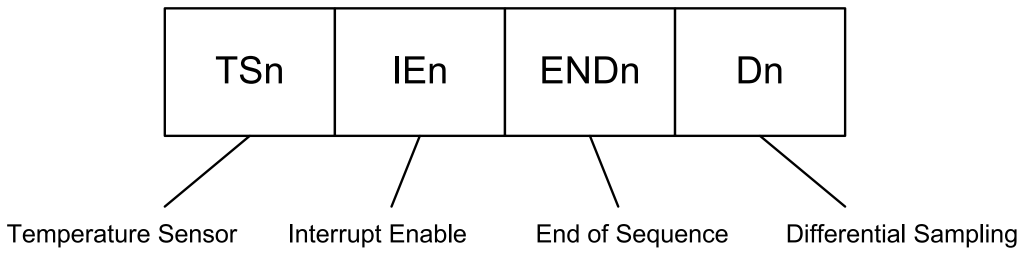


Table 6. Configuring the Sequence Steps

Using Direct Register Write

`//`
`// Configure sample sequence 1 input multiplexer:`
`//`
`// - Step 0: Analog Input 0`
`// - Step 1: Analog Input 1`
`// - Step 2: Analog Input 2`
`//`
`// ADC Sample Sequence Input Multiplexer Select 1 register`
`//`
`ADC_SSMUX1_R = ((0 << ADC_SSMUX1_MUX0_S) |`
`                 (1 << ADC_SSMUX1_MUX1_S) |`
`                 (2 << ADC_SSMUX1_MUX2_S));`
 	
`//`
`// Configure sample sequence 1 control`
`//`
`// - Step 0: Single-ended, No temp sensor, No interrupt`
`// - Step 1: Single-ended, No temp sensor, No interrupt`
`// - Step 2: Single-ended, No temp sensor, Interrupt, End of sequence`
`//`
`// ADC Sample Sequence Control 1 register`
`//`
`ADC_SSCTL1_R = (ADC_SSCTL1_END2 | ADC_SSCTL1_IE2);`
 	
`//`
`// Configure sample sequence 3 input multiplexer - not necessary since`
`// sequence 3 samples the temperature sensor`
`//`
 	
`//`
`// Configure sample sequence 3 control`
`//`
`// - Step 1: Single-ended, temp sensor, no interrupt, end of sequence`
`//`
`// ADC Sample Sequence Control 1 register`
`//`
`ADC_SSCTL3_R = (ADC_SSCTL3_TS0 | ADC_SSCTL3_END0);`

Using DriverLib

```
//
// Configure sample sequence 3 steps 0, 1 and 2
//
ADCSequenceStepConfigure(ADC_BASE, 1, 0, ADC_CTL_CH0);
ADCSequenceStepConfigure(ADC_BASE, 1, 1, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 1, 2, ADC_CTL_CH2 | ADC_CTL_IE | ADC_CTL_END)

//
// Configure sample sequence 3 step 0
//
ADCSequenceStepConfigure(ADC_BASE, 3, 0, ADC_CTL_TS | ADC_CTL_END);
```

Using ADC Interrupts

Luminary Micro's sequence-based architecture offers a vast amount of interrupt programming flexibility. All steps of a sample sequence have the capability to trigger an interrupt, allowing software to set indicators at any point in a sample sequence.

To enable interrupts, software must set the `MASK` bit for the sample sequence that requires the interrupt. For example, if sample sequence 1 is configured to trigger an interrupt, the `MASK1` bit of the **ADC Interrupt Mask (ADCIM)** register is set to '1'.

Table 7. Enabling the Sample Sequencer Interrupts

Using Direct Register Write

```
//
// Enable the interrupt for sample sequence 1
//
// ADC Interrupt Mask register
//
ADC_IM_R = ADC_IM_MASK1;
```

Using DriverLib

```
//
// Enable the interrupt for sample sequence 1
//
ADCIntEnable(ADC_BASE, 1);
```

Even with the interrupt mask bit set, software must set the step interrupt enable bit (`IE`) for one or more steps of a sample sequence for an interrupt to occur. The step interrupt enable bit is part of a step's configuration nibble in the **ADCSSCTL** register. In the example configuration being discussed, the `IE` bit for step 2 is set (see Table 6, "Configuring the Sequence Steps" on page 9).

Data Retrieval

Each sample sequence has its own FIFO with a depth equal to the number of steps the particular sequence supports (that is, sample sequence 0 has an 8-entry FIFO since it has 8 steps). Data retrieval from the FIFO is very straightforward; reading the **ADC Sample Sequence FIFO n (ADCSSFIFOn)** register returns a value from the FIFO.

The data returned during a FIFO read is a 32-bit value with the conversion result stored in the lower 10 bits. There are FIFO overflow and underflow flags available in the **ADC Overflow Status (ADCOSTAT)** and **ADC Underflow Status (ADCUSTAT)** registers, as well as FIFO empty, full, head, and tail pointer information in the **ADC Sample Sequence FIFO** registers.

Table 8. Retrieving Data from the FIFO

Using Direct Register Read

```
//
// Retrieve data from sample sequence 1 FIFO
//
// ADC Sample Sequence 1 FIFO register
//
ulSensor0Data = ADC_SSFIFO1_R;
ulSensor1Data = ADC_SSFIFO1_R;
ulSensor2Data = ADC_SSFIFO1_R;

//
// Retrieve data from sample sequence 3 FIFO
//
// ADC Sample Sequence 3 FIFO register
//
ulTempSensorData = ADC_SSFIFO3_R;
```

Using DriverLib

```
//
// Retrieve data from sample sequence 1 FIFO
//
ADCSequenceDataGet(ADC_BASE, 1, &ulSeq1DataBuffer);

//
// Retrieve data from sample sequence 3 FIFO
//
ADCSequenceDataGet(ADC_BASE, 3, &ulSeq3DataBuffer);
```

Differential Sampling

In addition to traditional single-ended sampling, the ADC module supports differential sampling of two analog input channels. To enable differential sampling, software must set the `D` bit in a step's configuration nibble (see Figure 3 on page 9).

When a sequence step is configured for differential sampling, its corresponding value in the **ADCSSMUX** register must be set to one of the four differential pairs, numbered 0-3. Differential pair 0 samples analog inputs 0 and 1; differential pair 1 samples analog inputs 2 and 3; and so on (see Table 9). The ADC does not support other differential pairings such as analog input 0 with analog input 3. The number of differential pairs supported is dependent on the number of analog inputs on the particular Stellaris microcontroller device.

Table 9. Differential Sampling Pairs

Differential Pair	Analog Inputs
0	0 and 1
1	2 and 3
2	4 and 5
3	6 and 7

The voltage sampled in differential mode is the difference between the odd and even channels:

ΔV (differential voltage) = V_0 (even channels) – V_1 (odd channels), therefore,

If $\Delta V = 0$, then the conversion result = 0x1FF
If $\Delta V > 0$, then the conversion result > 0x1FF (range is 0x1FF–0x3FF)
If $\Delta V < 0$, then the conversion result < 0x1FF (range is 0–0x1FF)

The differential pairs assign polarities to the analog inputs: the even-numbered input is always positive, and the odd-numbered input is always negative. In order for a valid conversion result to appear, the negative input must be in the range of ± 1.5 V of the positive input. If an analog input is greater than 3 V or less than 0 V (the valid range for analog inputs), the input voltage is clipped, meaning it appears as either 3 V or 0 V, respectively, to the ADC.

Figure 4 shows an example of the negative input centered at 1.5 V. In this configuration, the differential range spans from -1.5 V to 1.5 V. Figure 5 shows an example where the negative input is centered at -0.75 V meaning inputs on the positive input saturate past a differential voltage of -0.75 V since the input voltage is less than 0 V. Figure 6 shows an example of the negative input centered at 2.25 V, where inputs on the positive channel saturate past a differential voltage of 0.75 V since the input voltage would be greater than 3 V.

Figure 4. Differential Sampling Range, V_{in}(-) = 1.5 V

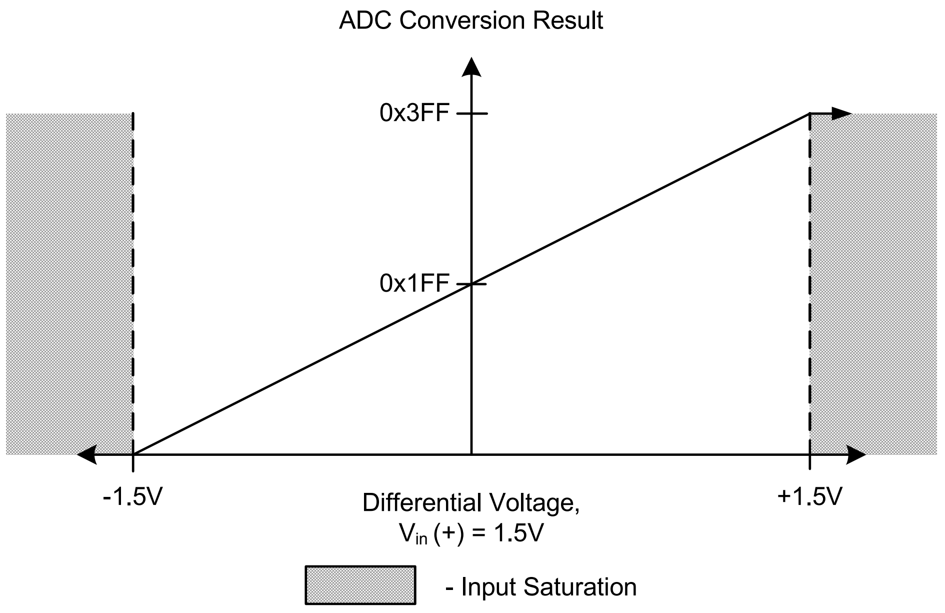


Figure 5. Differential Sampling Range, V_{in}(-) = 0.75 V

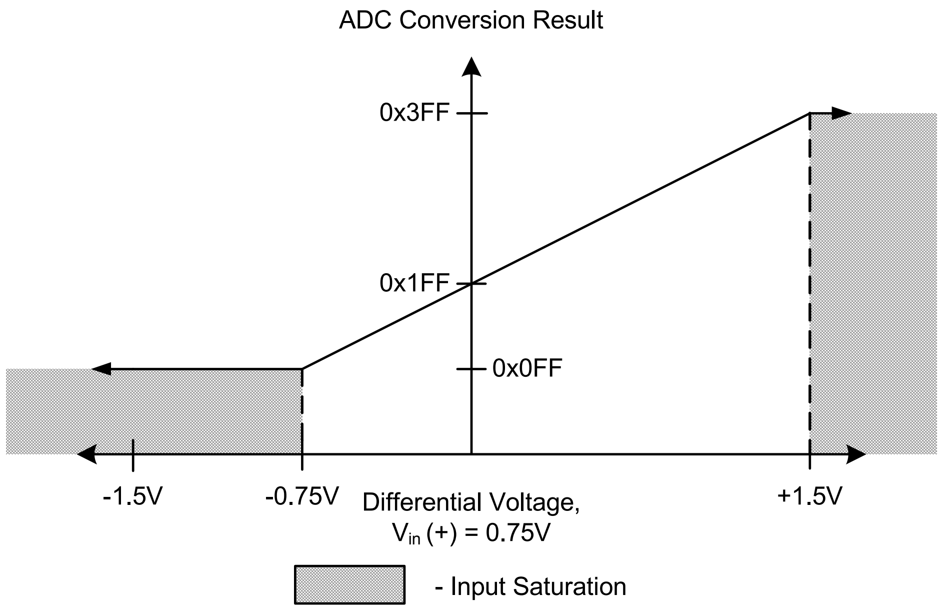


Figure 6. Differential Sampling Range, $V_{in}(-) = 2.25$ V

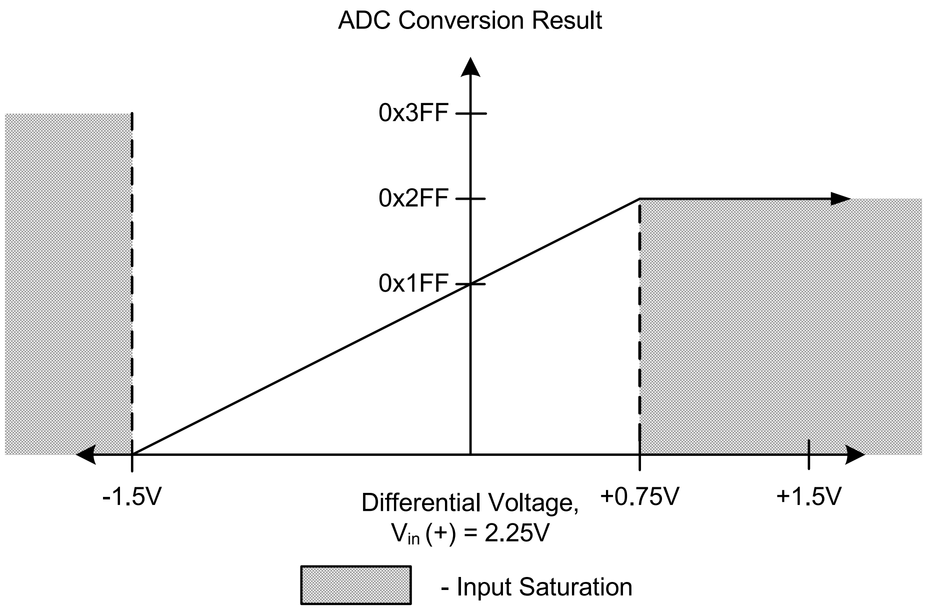


Table 10. Enabling Differential Sampling

Using Direct Register Write

```
//
// Configure sequence 3, step 0 for differential sampling
//
// ADC Sample Sequence Control 3 register
//
ADC_SSCTL3_R = (ADC_SSCTL3_D0 | ADC_SSCTL3_END0);

//
// Configure sample sequencer 3 input multiplexer to sample differential
// pair 1
//
// ADC Sample Sequence Input Multiplexer Select 3 register
//
ADC_SSMUX3_R = (1 << ADC_SSMUX3_MUX0_S);
```

Using DriverLib

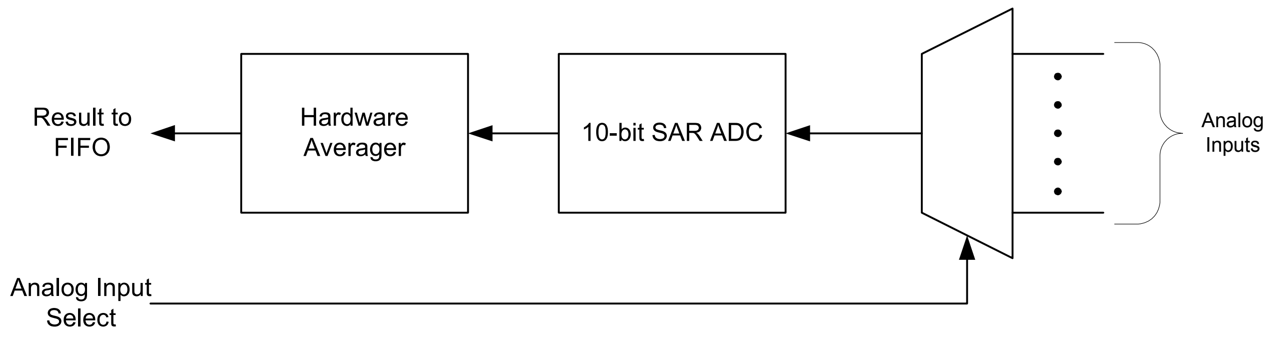
```
//
// Configure sequence 3, step 0 to sample differential pair 1
//
ADCSequenceStepConfigure(ADC_BASE, 3, 0, ADC_CTL_CH1 | ADC_CTL_D | ADC_CTL_END);
```

Hardware Averaging Circuit

Some applications require accuracy that exceeds the standard specifications of the ADC. To address this need, the ADC module contains a built-in hardware averaging circuit that can oversample the input source by up to 64 times.

When the hardware averaging circuit is enabled, all raw data collected by the ADC is oversampled by the same amount; the averager cannot be enabled/disabled on a step-by-step basis in a sample sequence. See Figure 7 for signal path details.

Figure 7. Hardware Averaging Circuit



There is also a bandwidth impact that must be considered before using the hardware averager. Whatever oversampling factor is selected for the averaging circuit also reduces the overall ADC throughput by the same amount. For example, oversampling by 8 reduces the throughput of a 500K samples/second ADC module to 62.5K sample/second since the ADC collects and averages 8 samples before returning a single conversion result to the FIFO.

To enable the hardware averaging circuit, software writes a value between 1 and 6 to the `AVG` field of the **ADC Sample Averaging Control (ADCSAC)** register. The amount of oversampling applied to the input is equivalent to 2^{AVG}. When the `AVG` bit is 0 (the default configuration), no averaging is applied to the input.

Table 11. Enabling 8x Hardware Averaging

Using Direct Register Write

`//`
`// Enable 8x hardware averaging`
`//`
`// ADC Sample Averaging Control register`
`//`
`ADC_SAC_R = ADC_SAC_AVG_8X;`

Using DriverLib

`//`
`// Enable 8x hardware averaging`
`//`
`ADCHardwareOversampleConfigure(ADC_BASE, 8);`

Conclusion

While architecturally different than many competitive ADC modules, the Luminary Micro ADC offers users more flexibility, control, and features without added processor overhead. Combining a straightforward programming interface and included Driver Library, many users will find the Luminary Micro ADC easy to integrate into their designs.

References

The following are available for download at www.luminarymicro.com:

- Stellaris® microcontroller data sheet, Publication Number DS-LM3S*nnn* (where *nnn* is the part number for that specific Stellaris family device)
- *Stellaris® Peripheral Driver Library User's Guide*, Publication Number SW-DRL-UG
- Stellaris® Peripheral Driver Library, Order Number SW-DRL

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext. 3