ADC Oversampling Techniques for Stellaris® Family Microcontrollers

Application Note

Copyright

Texas Instruments
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

Some members of the Stellaris® microcontroller family have an analog-to-digital converter (ADC) module. The hardware resolution of the ADC is 10 bits; however, due to noise and other accuracy-diminishing factors, the true accuracy is less than 10 bits. This application note provides a software-based oversampling technique, resulting in an improved Effective Number Of Bits (ENOB) in the conversion result. This document describes methods of oversampling an input signal, and the impact on precision and overall system performance.

Oversampling

As the name implies, *oversampling* gathers additional conversion data from an input signal. Standard convention for sampling an analog signal indicates that the sampling frequency f_s should be at least twice that of the highest frequency component f_H of the input signal. This is referred to as the Nyquist Theorem (see Equation 1).

Equation 1. Nyquist Theorem

$$f_S = 2\, f_H$$

Any sampling frequency selected above f_S is considered to be oversampling, and when combined with averaging techniques, improves the ENOB. This is possible because averaging the oversampled results also averages the quantization noise, thus improving the Signal-to-Noise Ratio (SNR), which has a direct effect on the ENOB.

For each bit of accuracy improvement, the signal must be oversampled by a factor of four, meaning that the relationship between the oversampling frequency f_{OS} and sampling frequency f_S is as shown in Equation 2.

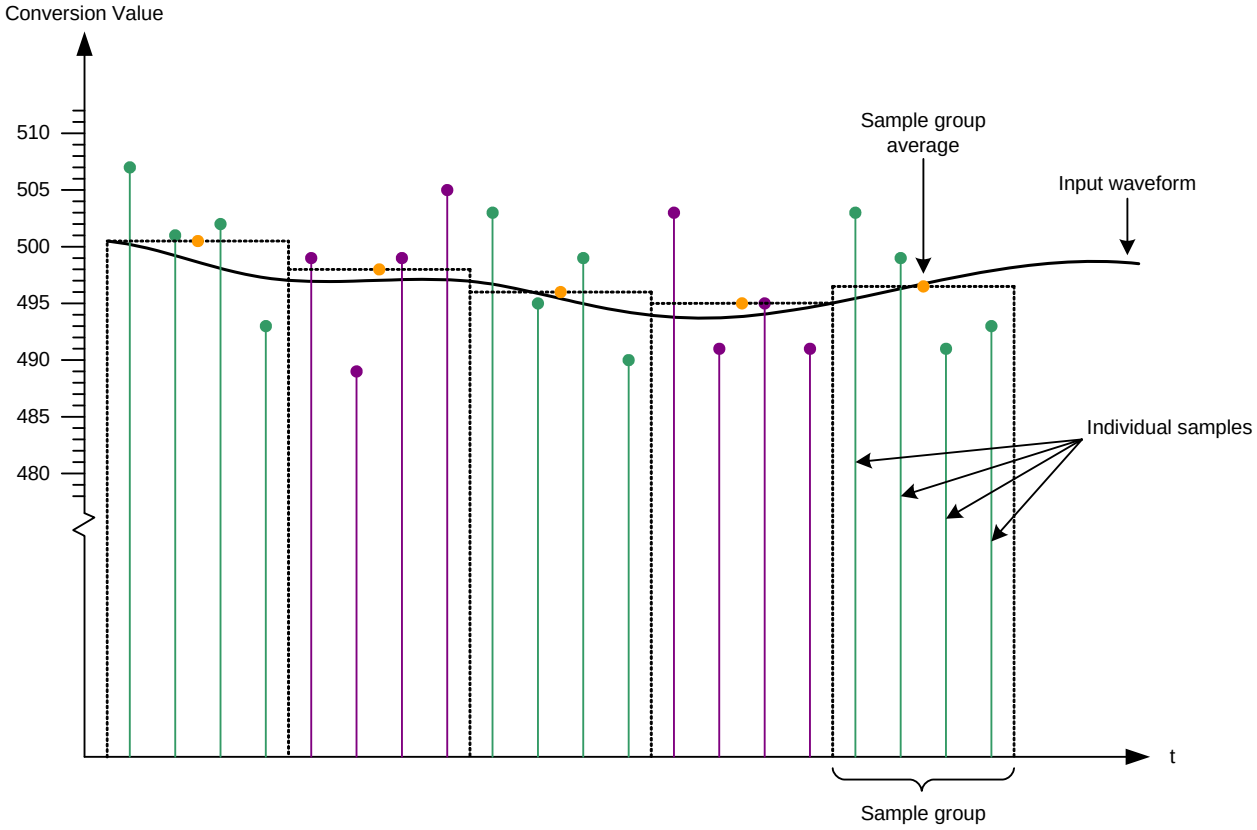
Equation 2. Oversampling Frequency

$$f_{OS} = 4^x * f_s$$

where x is the desired improvement on the ENOB (for example, for two bits of improvement, $x = 2$).

Figure 1 on page 5 shows how oversampling improves the accuracy of the conversion result. In this diagram, the input signal is oversampled by 4 (sample groups are shown in green and purple) and averaged. The sample points shown illustrate the difference between the raw, noisy signal and the average; the noise in this example affecting ± 3 bits of accuracy on an individual sample. Note that the averaged values (orange dots) are much closer to the ideal value than most of the single samples.

Figure 1. Averaged Conversion Results



Averaging

Averaging acts as a low-pass filter on the input signal, with the pass band of the filter narrowing as the sample size increases. When averaging conversion results, there are two approaches that can be taken: *normal average* or *rolling average*.

Normal Average

Taking *n* samples, adding them, and dividing the result by *n* is referred to as a *normal average*, and is shown in Figure 1. When using normal averaging in an oversampling scenario, after the technique is applied, the sample data used in the calculation is discarded. This process is repeated every time the application needs a new conversion result.

In an application, the normal averaging approach is ideally used in cases where the sampling frequency is low compared to the sampling rate of the ADC.

Important: When oversampling by *n* in a normal averaging scenario, the effective ADC sample rate is reduced by that same factor. For example, oversampling an input signal by 4 cuts the maximum effective ADC sample rate by 4, meaning that a 250K-samples/s ADC effectively becomes a 62.5K-samples/s ADC.

Figure 2 shows a situation where normal averaging is used to oversample an input source by 4. For this example, the application requires that a new conversion value be ready (averaging complete) at each step of t (*t0*, *t1*, *t2*, and so on).

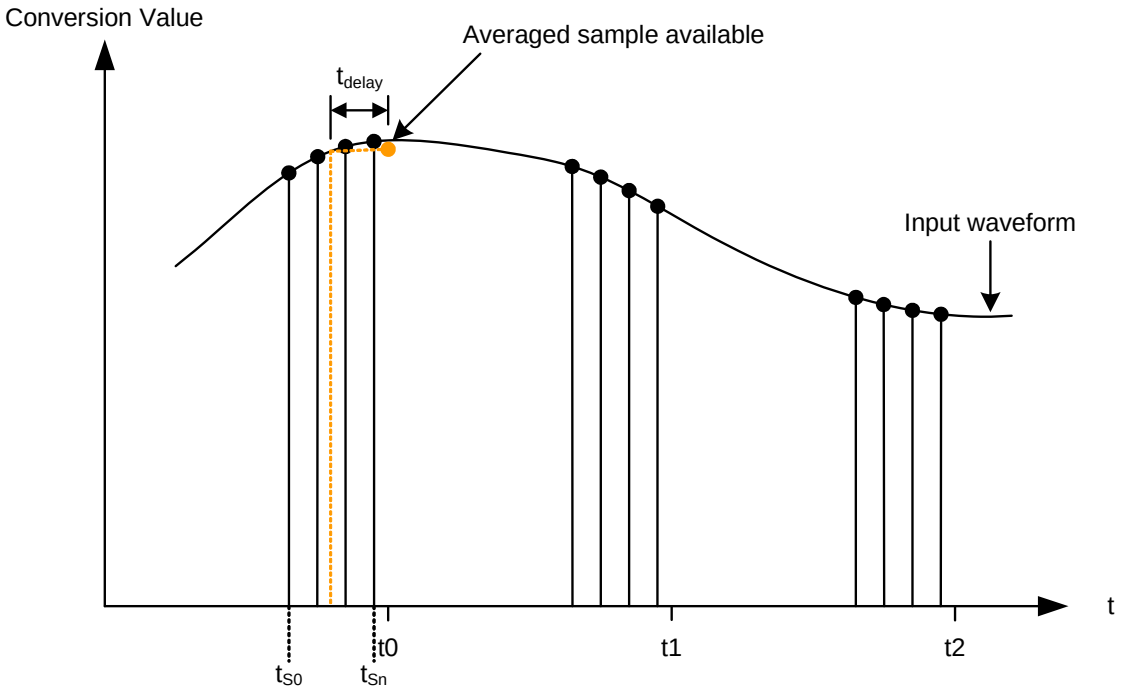
When using averaging techniques, there is a slight delay associated with the calculated conversion result since it corresponds to the average of the last *n* samples. The delay can be calculated using the formula shown in Equation 3.

Equation 3. Averaged Sample Delay

$$t_{delay} = (t_{Sn} - t_{S0}) / 2 + t_{process}$$

where t_{S0} is the time at which the first sample of the average occurs, and t_{Sn} is where the last sample occurs. The time required by the interrupt handler to process the sample data and calculate the average $t_{process}$ to supply to the application is also factored into the equation. In Figure 2, the delayed conversion result is highlighted in orange.

Figure 2. Normal Averaging



Rolling Average

A *rolling average* uses a sample buffer of the *n* most recent samples in the averaging calculation, allowing the ADC to sample at its *maximum rate* (the ADC sample rate is not reduced by *n* as in normal averaging), making it ideally suited for applications requiring oversampling and higher sample rates. The sample buffer can be prefilled with valid sample data (by taking *n–1* samples prior to the first “real” data point), or can be left in an unknown state, depending on the application. The disadvantage of not prefilling the buffer is that the first *n–1* samples contain invalid data and

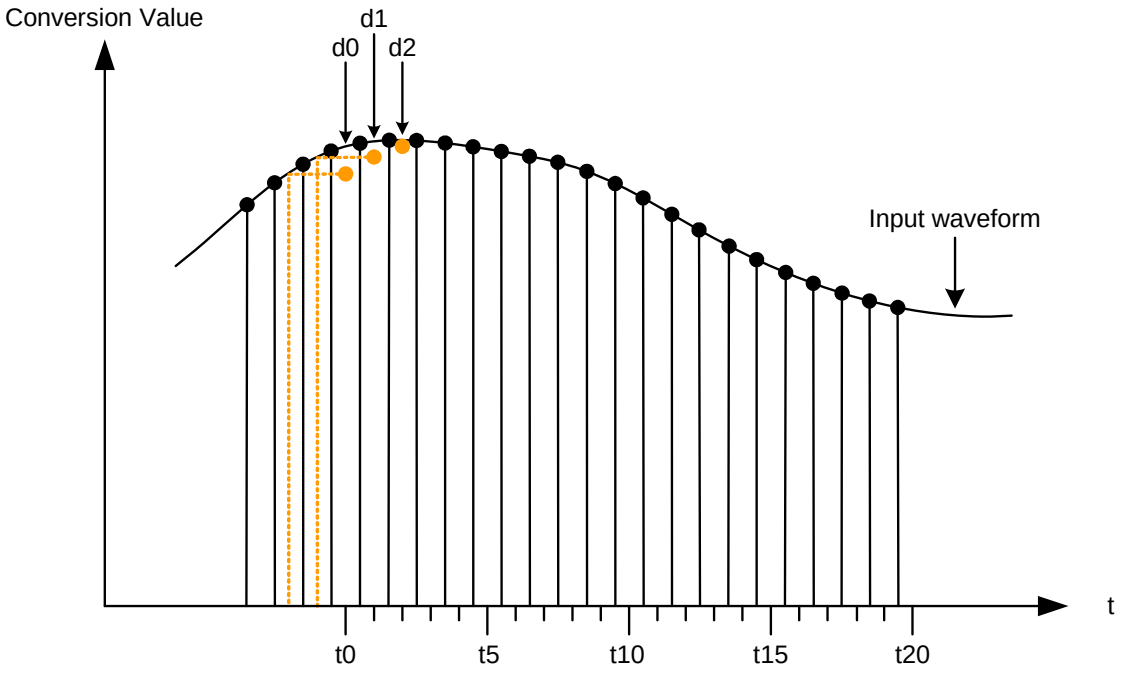
adversely affect the rolling average calculation. If acceptable by the application, buffer padding can be eliminated if the software can account for the possibility of the first *n–1* samples being skewed.

Figure 3 shows an example of oversampling with a rolling average. The diagram shows a case where the input signal is oversampled by 4, meaning that the sample buffer uses the 4 most recent samples to calculate the average. In this example, the application requires a new sample at each step of *t*. Before the first oversampled result is calculated at *t0*, the sample buffer collects three samples so that the first data supplied to the application is valid.

When using a rolling average, the same sample delay calculated by Equation 3 applies. In Figure 3, the delayed values for *t0*, *t1* and *t2* (shown as d0, d1 and d2, respectively) are highlighted in orange.

Important: Using a rolling average adds additional processing overhead due to the sample buffer manipulation that must be performed during each interrupt.

Figure 3. Rolling Average



Implementation

The sample sequencer architecture in the ADC makes oversampling simple by allowing for up to 17 unique samples (from any of the analog channels) to be collected using a single trigger. This allows for flexibility in software by providing the means for an application to oversample a number of channels at any given time.

This section shows various implementations of oversampling using the Stellaris microcontrollers. There are numerous methods that work using combinations of sample sequencer configurations, ADC triggers and interrupts, however, the examples shown here focus on techniques that are most likely to be used.

All the example code uses the Stellaris Peripheral Driver Library ADC functions. The source code for the Driver Library and the software examples shown in this application note can be found on the website at http://www.luminarymicro.com.

Oversampling Up to 8 Times Using the Driver Library Functions

The Stellaris Peripheral Driver Library has built-in functions that allow oversampling up to 8 times. For most applications, this level of oversampling is sufficient since the improvement on the ENOB is approximately 1.4 bits.

Using the Driver Library oversampling functions is the easiest way to oversample the input signal. The main difference between configuring a “typical” ADC conversion and an oversampled conversion is the function calls. The oversampling functions have an **ADCSoftwareOversample** prefix, and are easily distinguished from the standard ADC functions.

Once the parameters for the ADC conversion process are determined (sample frequency, trigger source, channel, and so on), writing the code is straight-forward. For example, the code to set up a 10-ms periodic conversion (triggered by a timer) that is oversampled by 8 consists of the code segments shown in Example 1.

Example 1. 8x Oversampling with the Driver Library Functions

Code Segment 1.a. ADC Configuration – Driver Library Functions

```
//
// Initialize the ADC to oversample channel 1 by 8x using sequencer 0.
// Sequencer will be triggered by one of the general-purpose timers.
//
ADCSequenceConfigure(ADC_BASE, 0, ADC_TRIGGER_TIMER, 0);
ADCSoftwareOversampleConfigure(ADC_BASE, 0, 8);
ADCSoftwareOversampleStepConfigure(ADC_BASE, 0, 0, (ADC_CTL_CH1 \
                                   | ADC_CTL_IE | ADC_CTL_END));

//
// Initialize Timer 0 to trigger an ADC conversion once every 10 milliseconds
//
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 100);
TimerControlTrigger(TIMER0_BASE, TIMER_A, true);
```

The ADC configuration shown in Code Segment 1.a dictates that an interrupt occur when sampling completes, meaning that an interrupt handler must be implemented (see Code Segment 1.b). Since the Driver Library oversampling functions automatically average the sampled data, the interrupt handler function is relatively basic. Keep in mind that having the average calculated during each interrupt adds computational overhead to the interrupt handler.

Code Segment 1.b. ADC Interrupt Handler

```
void
ADCIntHandler(void)
{
       long lStatus;

       //
       // Clear the ADC interrupt
       //
       ADCIntClear(ADC_BASE, 0);

       //
       // Get averaged data from the ADC
       //
       lStatus = ADCSoftwareOversampleDataGet(ADC_BASE, 0, &g_ulAverage);

       //
       // Placeholder for ADC processing code
       //
}
```

With the configuration steps and interrupt handler in place, the conversion process is initiated. Before the timer is turned on (begins counting), the ADC sequencer and interrupt must be enabled (see Code Segment 1.c).

Code Segment 1.c. Enabling the ADC and Interrupts

```
//
// Enable ADC sequencer 0 and its interrupt (in both the ADC and NVIC)
//
ADCSequenceEnable(ADC_BASE, 0);
ADCIntEnable(ADC_BASE, 0);
IntEnable(INT_ADC0);

//
// Enable the timer and start conversion process
//
TimerEnable(TIMER0_BASE, TIMER_A);
```

Oversampling More Than 8 Times Using Multiple Sequencers or a Timer

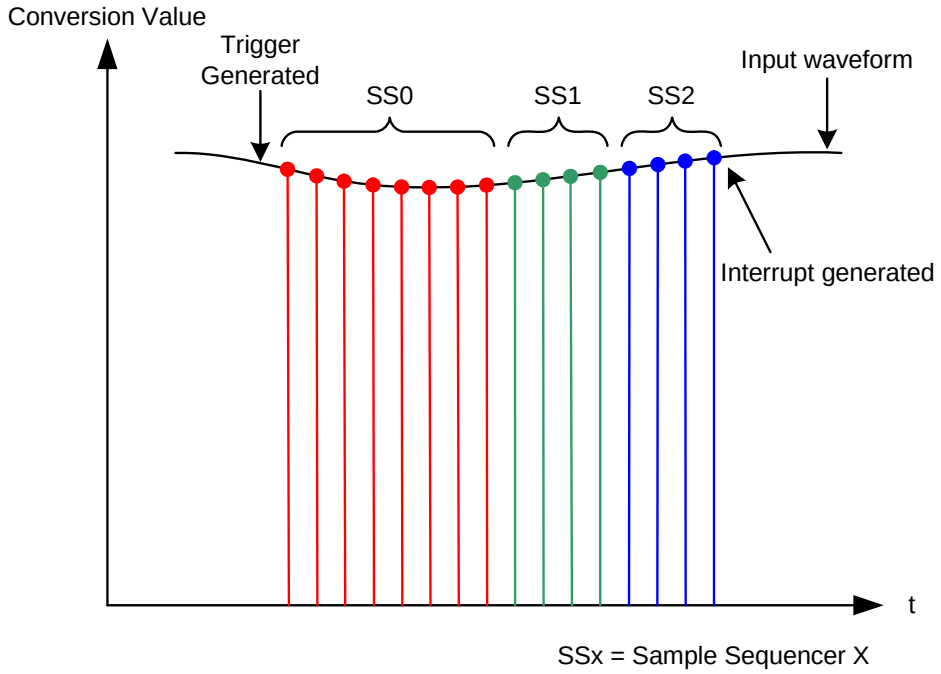
The Driver Library oversampling functions are limited to oversampling by 8 times (based on hardware limitations of the sample sequencers), meaning that applications requiring larger oversampling factors must use an alternative implementation. This section shows how to approach such a situation using two methods: multiple sample sequencers and a timer running at the oversampling frequency.

Example 2. 16x Oversampling Using Multiple Sample Sequencers

The flexibility of the sample sequencer allows a wide range of configurations. To oversample by 16, sample sequencers 0-2 can be used since their accumulated capacity is 16 samples (8 + 4 + 4). For this level of oversampling to work, all steps in the sequencers must be set to sample the same analog input, meaning the ability to sample multiple inputs using a single sequencer is lost.

Code Segment 2.a on page 11 configures a 10-ms periodic conversion using sequencers 0-2. A single timer trigger is used to initiate sampling on all 3 sequencers, eliminating the need for complex trigger configurations. To obtain the desired result, the sample sequencer priorities are configured such that sample sequencer 2 has the lowest priority (meaning it samples last), and the “end of conversion” interrupt is configured to assert after the last step of sample sequence 2 (as shown in Figure 4).

Figure 4. Oversampling by 16



Code Segment 2.a. ADC Configuration – Multiple Sample Sequencers

```
//
// Initialize the ADC for 16x oversampling on channel 1 using sequencers
// 0-2.  The conversion is triggered by a GPTM.
//
ADCSequenceConfigure(ADC_BASE, 0, ADC_TRIGGER_TIMER, 0);
ADCSequenceConfigure(ADC_BASE, 1, ADC_TRIGGER_TIMER, 1);
ADCSequenceConfigure(ADC_BASE, 2, ADC_TRIGGER_TIMER, 2);

//
// Configure sequence steps for sequencer 0
//
ADCSequenceStepConfigure(ADC_BASE, 0, 0, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 1, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 2, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 3, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 4, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 5, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 6, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 0, 7, (ADC_CTL_CH1 | ADC_CTL_END));

//
// Configure sequence steps for sequencer 1
//
ADCSequenceStepConfigure(ADC_BASE, 1, 0, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 1, 1, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 1, 2, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 1, 3, (ADC_CTL_CH1 | ADC_CTL_END));

//
// Configure sequence steps for sequencer 2
//
ADCSequenceStepConfigure(ADC_BASE, 2, 0, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 2, 1, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 2, 2, ADC_CTL_CH1);
ADCSequenceStepConfigure(ADC_BASE, 2, 3, (ADC_CTL_CH1 | ADC_CTL_IE \
                         | ADC_CTL_END));

//
// Initialize Timer 0 to trigger an ADC conversion once every 10 milliseconds
//
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 100);
TimerControlTrigger(TIMER0_BASE, TIMER_A, true);
```

In Code Segment 2.b, the interrupt handler must gather the data from the FIFOs and perform the averaging calculation. The **ADCSequenceDataGet** function is not used here since the desired results are obtained without having to deal with the function overhead, so direct register reads are used to empty the sequencer FIFOs. Using **ADCSequenceDataGet** requires the function to define an extra 8-entry sample buffer. Even with direct register reads, there is still computational overhead from the sum and average calculations performed in the interrupt handler.

Code Segment 2.b. ADC Interrupt Handler

```
void
ADCIntHandler(void)
{
       unsigned long ulIdx;
       unsigned long ulSum = 0;

       //
       // Clear the interrupt
       //
       ADCIntClear(ADC_BASE, 2);

       //
       // Get the data from sequencer 0
       //
       for(ulIdx = 8; ulIdx; ulIdx--)
       {
              ulSum += HWREG(ADC_BASE + ADC_O_SSFIFO0);
       }

       //
       // Get the data from sequencers 1 and 2
       //
       for(ulIdx = 4; ulIdx; ulIdx--)
       {
              ulSum += HWREG(ADC_BASE + ADC_O_SSFIFO1);
              ulSum += HWREG(ADC_BASE + ADC_O_SSFIFO2);
       }

       //
       // Average the oversampled data
       //
       g_ulAverage = ulSum >> 4;

       //
       // Placeholder for ADC processing code
       //
}
```

Before initiating the conversion process, the sample sequencers and interrupts are enabled (see Code Segment 2.c).

Code Segment 2.c. Enabling the ADC and Interrupts

```
//
// Enable the sequencers and interrupt
//
ADCSequenceEnable(ADC_BASE, 0);
ADCSequenceEnable(ADC_BASE, 1);
ADCSequenceEnable(ADC_BASE, 2);
ADCIntEnable(ADC_BASE, 2);
IntEnable(INT_ADC2);

//
// Enable the timer and start conversion process
//
TimerEnable(TIMER0_BASE, TIMER_A);
```

Example 3. 16x Oversampling Using a Timer Running at f_{OS}

Another way to oversample (without consuming a large portion of the ADC sequencer resources) is by using a periodic timer that runs at the oversampling frequency. For example, if a conversion must be returned to the main application every 10 ms and is to be oversampled by 16, a timer can be configured to take a single sample every 625 µs. Having the timer trigger a conversion at the oversampling frequency obviously generates additional ADC interrupt traffic, which must be accounted for in the application.

The code that configures the ADC and timer to operate like this is shown in Code Segment 3.a.

Code Segment 3.a. ADC Configuration – Timer Running at f_{OS}

```
//
// Initialize the ADC to take a single sample on channel 1, sequencer 3
// when a trigger is detected.
//
ADCSequenceConfigure(ADC_BASE, 3, ADC_TRIGGER_TIMER, 0);
ADCSequenceStepConfigure(ADC_BASE, 3, 0, (ADC_CTL_CH1 | ADC_CTL_IE \
                         | ADC_CTL_END));

//
// Initialize Timer 0 to trigger an ADC conversion once every 625 microseconds
//
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 1600);
TimerControlTrigger(TIMER0_BASE, TIMER_A, true);
```

Now that the ADC is sampling at the oversampling frequency, the interrupt handler must keep track of the number of samples taken and the overall sum (see Code Segment 3.b on page 14). When 16 conversions have been accumulated, the averaging is performed and the global sample count and sum variables are cleared.

Code Segment 3.b. ADC Interrupt Handler

```
void
ADCIntHandler(void)
{
    //
    // Clear the interrupt
    //
    ADCIntClear(ADC_BASE, 3);

    //
    // Add the new sample to the global sum
    //
    g_ulSum += HWREG(ADC_BASE + ADC_O_SSFIFO3);

    //
    // Increment g_ucOversampleCnt
    //
    g_ucOversampleCnt++;

    //
    // If 16 samples have accumulated, average them and reset globals
    //
    if(g_ucOversampleCnt == 16)
    {
        g_ulAverage = g_ulSum >> 4;
        g_ucOversampleCnt = 0;
        g_ulSum = 0;
    }

    //
    // Placeholder for ADC processing code
    //
}
```

Finally, before enabling the timer, sequencer 3 and its interrupt are enabled, and the global counter and sum variables are cleared (see Code Segment 3.c).

Code Segment 3.c. Enabling the ADC, Interrupts and Global Variables

```
//
// Enable the sequencer and interrupt
//
ADCSequenceEnable(ADC_BASE, 3);
ADCIntEnable(ADC_BASE, 3);
IntEnable(INT_ADC3);

//
// Zero the oversample counter and the sum
//
g_ucOversampleCnt = 0;
g_ulSum = 0;

//
// Enable the timer and start conversion process
//
TimerEnable(TIMER0_BASE, TIMER_A);
```

Oversampling Using a Rolling Average

The rolling average approach is useful in situations where the sampling frequency is closer to the maximum sample rate of the ADC. The main component of a rolling average application is the sample buffer, which drops/adds data each time a conversion completes.

In Example 4, the ADC is configured to sample once every 100 µs, and the sample buffer contains 16 entries. Note that the application does not prefill the sample buffer with valid data, so the first 16 samples must be handled accordingly by software. The ADC is configured to sample on a timer trigger, and interrupts the processor after every conversion.

Example 4. Oversampling Using a Rolling Average Every 100 Microseconds

Code Segment 4.a. ADC Configuration – Rolling Average

```
//
// Initialize the ADC to take a single sample on channel 1, sequencer 3
// when a trigger is detected.
//
ADCSequenceConfigure(ADC_BASE, 3, ADC_TRIGGER_TIMER, 0);
ADCSequenceStepConfigure(ADC_BASE, 3, 0, (ADC_CTL_CH1 | ADC_CTL_IE \
                         | ADC_CTL_END));

//
// Initialize Timer 0 to trigger an ADC conversion once every 100 microseconds
//
TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 10000);
TimerControlTrigger(TIMER0_BASE, TIMER_A, true);
```

The interrupt handler is responsible for updating the sample buffer and performing the averaging calculation (see Code Segment 4.b). For each ADC interrupt, the last element in the sample buffer is dropped, and the remaining data is shifted over one place in the buffer. The new conversion result is then placed at the beginning of the sample buffer before the average is computed. Again, the additional computations performed in the interrupt handler add overhead that must be taken into account.

Code Segment 4.b. ADC Interrupt Handler

```
void
ADCIntHandler(void)
{
    //
    // Clear the interrupt
    //
    ADCIntClear(ADC_BASE, 3);

    //
    // Check g_ucOversampleIdx to make sure that it is within range
    //
    if(g_ucOversampleIdx == 16)
    {
        g_ucOversampleIdx = 0;
    }
      //
      // Subtract the oldest value from the global sum
```

```
       //
    g_ulSum -= g_ulSampleBuffer[g_ucOversampleIdx];

    //
    // Replace the oldest value with the new sample value
    //
    g_ulSampleBuffer[g_ucOversampleIdx] = HWREG(ADC_BASE + ADC_O_SSFIFO3);

    //
    // Add the new sample to the overall sum
    //
    g_ulSum += g_ulSampleBuffer[g_ucOversampleIdx];

    //
    // Increment g_ucOversampleIdx
    //
    g_ucOversampleIdx++;

    //
    // Get the averaged value from the sample buffer data
    //
    g_ulAverage = g_ulSum >> 4;

    //
    // Placeholder for ADC processing code
    //
}
```

Again, before the timer is started, the sequencer and its interrupt are enabled (see Code Segment 4.c).

Code Segment 4.c. Enabling the ADC and Interrupts

```
//
// Enable the sequencer and the interrupt
//
ADCSequenceEnable(ADC_BASE, 3);
ADCIntEnable(ADC_BASE, 3);
IntEnable(INT_ADC3);

//
// Enable the timer and start conversion process
//
TimerEnable(TIMER0_BASE, TIMER_A);
```

Issues to Consider

The oversampling techniques described in this document have an obvious impact on overall system performance since they require additional code to perform the averaging calculations, additional interrupts, and/or most of the sample sequencer resources. Choosing the technique that best suits the application requires analyzing the trade-offs between increased interrupt traffic and bulkier interrupt handlers.

Conclusion

The sample sequencer architecture offers a wide range of options for implementing oversampling techniques. When combined with software averaging, the architecture enables system designers to effectively make the engineering trade-offs between sampling frequency, system performance and sampling resolution.

References

The following documents 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)
- StellarisWare® Driver Library
- *StellarisWare® Driver Library User's Manual*, publication number SW-DRL-UG

In addition, the following document may be useful:

- Nyquist Theorem, Wikipedia.org, http://en.wikipedia.org/wiki/Nyquist_Theorem

Important Notice

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265