Improving ADC Resolution by Oversampling and Averaging

1. Introduction

Many applications require measurements using an analog-to-digital converter (ADC). Such applications will have resolution requirements based in the signal's dynamic range, the smallest change in a parameter that must be measured, and the signal-to-noise ratio (SNR). For this reason, many systems employ a higher resolution off-chip ADC. However, there are techniques that can be used to achieve higher resolution measurements and SNR. This application note describes utilizing oversampling and averaging to increase the resolution and SNR of analog-to-digital conversions. Oversampling and averaging can increase the resolution of a measurement without resorting to the cost and complexity of using expensive off-chip ADCs.

This application note discusses how to increase the resolution of analog-to-digital (ADC) measurements by oversampling and averaging. Additionally, more in-depth analysis of ADC noise, types of ADC noise optimal for oversampling techniques, and example code utilizing oversampling and averaging is provided in appendices A, B, and C respectively at the end of this document.

2. Key Points

- Oversampling and averaging can be used to increase measurement resolution, eliminating the need to resort to expensive, off-chip ADCs.
- Oversampling and averaging will improve the SNR and measurement resolution at the cost of increased CPU utilization and reduced throughput.
- Oversampling and averaging will improve signal-to-noise ratio for "white" noise.

2.1. Sources of Data Converter Noise

Noise in ADC conversions can be introduced from many sources. Examples include: thermal noise, shot noise, variations in voltage supply, variation in the reference voltage, phase noise due to sampling clock jitter, and noise due to quantization error. The noise caused by quantization error is commonly referred to as *quantization noise*. Noise power from these sources can vary. Many techniques that may be utilized to reduce noise, such as thoughtful board layout and bypass capacitance on the reference voltage signal trace. However, ADCs will always have quantization noise, thus the best SNR of a data converter of a given number of bits is defined by the quantization noise with no oversampling. Under the correct conditions, oversampling and averaging will reduce noise and improve the

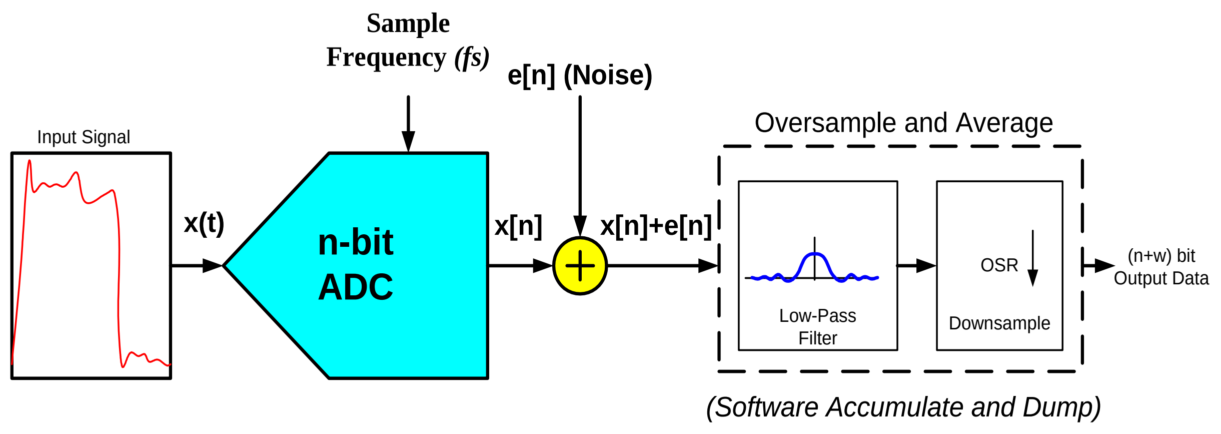


Figure 1. Oversampling and Averaging to Increase Measurement Resolution By "w" Bits

SNR. This will effectively increase the number of bits of a measurement's resolution. Such a system is shown in Figure 1 on page 1, and can be implemented with Silicon Lab's on-chip ADC and a software routine that takes a set of samples and averages (filters) them for the result.

Increasing the Resolution of an ADC Measurement

Many applications measure a large dynamic range of values, yet require fine resolution to measure small changes in a parameter. For example, an ADC may measure a large temperature range, yet still require the system to respond to changes of less than one degree. Such a system could require a measurement resolution of 16 bits. Rather than resorting to an expensive, off-chip 16-bit ADC, oversampling and averaging using Silicon Lab's on-chip, 12-bit ADC can measure a parameter with 16 bits of resolution.

Some applications will use an ADC to analyze a signal with higher frequency components. Such a system will also benefit from oversampling and averaging. The required sampling frequency in accordance with the Nyquist Theorem is the *Nyquist Frequency*:

$$f_n = 2 \cdot f_m$$

where f_m is the highest frequency component of interest in the input signal

Equation 1. Nyquist Frequency

Sampling frequencies (f_s) above f_n is *oversampling*, and will increase the resolution of a measurement. Please see **Appendix A** for a discussion of how this works.

Calculating the Oversampling Requirements To Increase Resolution

To increase the effective number of bits (ENOB), the signal is *oversampled*, or sampled by the ADC at a rate that is higher than the system's required sampling rate, f_s. The required sampling rate may be determined by how often a system requires a parameter be measured (output word rate), or it may be the Nyquist frequency, f_n.

For each additional bit of resolution, the signal must be oversampled by a factor of four:

$$f_{os} = 4^w \cdot f_s$$

*where **w** is the number of **additional** bits of resolution desired, $\boldsymbol{f_s}$ is the original sampling frequency requirement, and $\boldsymbol{f_{os}}$ is the oversampling frequency*

Equation 2. Oversampling Frequency To Add Measurement Resolution

A derivation of Equation 2 is presented in **Appendix A**.

Assume a system is using a 12-bit ADC to output a temperature value once every second (1 Hz). To increase the resolution of the measurement to 16-bits, we calculate the oversampling frequency as follows:

$$f_{os} = 4^4 \cdot 1(\text{Hz}) = 256\text{Hz}$$

Thus, if we oversample the temperature sensor at f_s=256 Hz, we will collect enough samples within the required sampling period to average them and can now use 16-bits of the output data for a 16-bit measurement. To do so, we accumulate (add 256 consecutive samples together), then divide the total by 16 (or right shift the total by 4-bits). Such a process is commonly referred to as *decimation*. This results in 16-bits of useful data. Such an operation is referred to as *accumulate and dump*. Once we calculate the result of 256 samples (in this example), we store or process the data and begin collecting data for the next output word.

Note: The memory location used to accumulate the oversampled data and perform the divide must have enough bytes to prevent overflow and truncation error.

An example of such oversampling and averaging is provided in **Appendix C**. In this example, Silicon Lab's on-chip temperature sensor is sampled using the on-chip 12-bit ADC to make a 16-bit measurement. For a more formal discussion of how oversampling affects noise and increases resolution, please see **Appendix A**.

Calculating the Oversampling Requirements To Increase SNR

The theoretical limit of the SNR of an ADC measurement is based on the *quantization noise* due to the quantization error inherent in the analog-to-digital conversion process when there is no oversampling and averaging. Because quantization error depends on the number of bits of resolution of the ADC (see Equation 5), the best case SNR is calculated as a function of the *Effective Number of Bits* of a data conversion as follows:

$$\mathrm{SNR(dB)} = (6.02 \cdot \mathrm{ENOB}) + 1.76$$

where ENOB is the effective number of bits of the measurement

Equation 3. SNR Calculation as a Function of ENOB

Note Equation 3 is valid for a *full-scale* input. That is, the dynamic range of the input signal must match the reference voltage of the ADC. If not, the SNR will be lower than that calculated using Equation 3.

If the ADC used to measure a parameter is 12-bits and not oversampled, then the best SNR (calculated using Equation 3) is 74 dB. If we desire a better SNR, then we could calculate the ENOB needed using Equation 3 for a specified SNR. Once we know the required ENOB, we can then use Equation 2 to calculate the oversampling requirements.

For example, if the required SNR for an application is 90 dB, then we will need at least 16-bits of resolution. Using and 12-bit ADC and Equation 2, we know we must oversample by a factor of 256.

When Oversampling and Averaging Will Work

The effectiveness of oversampling and averaging depends on the characteristics of the dominant noise sources. The key requirement is that the noise can be modeled as *white noise*. Please see **Appendix B** for a discussion on the characteristics of noise that will benefit from oversampling techniques. Key points to consider are [2] [3]:

- The noise must approximate *white noise* with uniform power spectral density over the frequency band of interest.
- The noise amplitude must be sufficient to cause the input signal to change randomly from sample to sample by amounts comparable to at least the distance between two adjacent codes (i.e., 1 LSB - please see Equation 5 in **Appendix A**).
- The input signal can be represented as a random variable that has equal probability of existing at any value between two adjacent ADC codes.

Note: Oversampling and averaging techniques will not compensate for ADC integral non-linearity (INL).

Noise that is correlated or cannot be modeled as white noise (such as noise in systems with feedback) will not benefit from oversampling techniques. Additionally, if the quantization noise power is greater than that of natural white noise (e.g., thermal noise), then oversampling and averaging will not be effective. This is often the case in lower resolution ADCs. The majority of applications using 12-bit ADCs can benefit from oversampling and averaging.

Please see **Appendix B** for a further discussion on this topic.

Example

An example that utilizes oversampling and averaging is provided in this application note in **Appendix C**. This code uses Silicon Lab’s on-chip, 100 ksps, 12-bit ADC to perform a 16-bit measurement of the on-chip temperature sensor, then outputs this data via the hardware UART.

Using Equation 2, the oversampling ratio is 256. The provided code (in “AN018_SW.c”) adds 256 consecutive ADC samples to the variable *accumulator*. After 256 samples have been added, it shifts *accumulator* right 4 bits and places the result in the variable *result*. This gives 16-bits of useful data. After the result is calculated, *accumulate* is then “dumped” (cleared) for the next calculation. The accumulation of the ADC samples are performed in an ADC end-of-conversion interrupt service routine (ADC_isr).

For more information concerning configuring and using the on-chip temperature sensor, please see application note “AN003 - Using the On-Chip Temperature Sensor.”

Resolution Improvement

We oversample and average the temperature sensor to increase the measurement resolution from 12-bits to 16-bits. Let’s compare the improvement in the temperature measurement.

The full-scale output of the on-chip temperature sensor is slightly less than 1 volt. Assuming a reference voltage (V_{ref}) of 2.4 volts, we can calculate the code width and temperature resolution (smallest measurable change in temperature) for both 12-bit and 16-bit measurements.

12-bit Temperature Resolution

Without oversampling, we will get a 12-bit result from the temperature measurement. The on-chip temperature sensor voltage will change *2.8 mV for each change in degrees Celsius*. The voltage resolution for a 2.4 volt V_{ref} and a PGA gain of 2 is (using Equation 5 in **Appendix A**):

$$\Delta = \frac{2.4}{2^{12} \cdot 2} = 293\mu V/°C$$

Δ is the code width as defined in Equation 5 on page 7. *The factor of 2 in the denominator is to account for a PGA gain of 2.*

Thus, the temperature resolution in a 12-bit measurement (the number a degrees C per ADC code) is:

$$_{res12} = \frac{293\mu V}{code} \cdot \frac{°C}{2.8mV} = 0.1046\ °C/cod$$

T_{res12} is the temperature resolution for a 12-bit measurement.

So for each ADC code, the minimum temperature change we may measure is 0.104 degrees C or above one-tenth of a degree. Perhaps we need better temperature resolution that will allow us to display closer to one-hundredth of a degree. We can achieve this resolution by using the same 12-bit ADC with oversampling and averaging.

16-Bit Temperature Resolution

Increasing the effective number of bits (ENOB) to 16-bits through oversampling and averaging, a new resolution is calculated as follows:

$$\Delta = \frac{1.2}{2^{16}} = 18.3\mu V/^{\circ}C$$

Thus, the smallest temperature change we can measure is:

$$_{s16} = \frac{18.3\mu V}{code} \cdot \frac{^{\circ}C}{2.8mV} = 0.0065\ ^{\circ}C/co$$

$_{res16}$ is the temperature resolution for a 16
it measurement.

We can now measure a 0.007 degree C change in temperature using the same, on-chip, 12-bit ADC with oversampling and averaging. This now allows us to measure temperature to an accuracy of better than one-hundredth of a degree.

Reduced Throughput

Throughput refers to the number of output data words we obtain per unit time. If an ADC has a maximum sample rate of 100 ksps, we would obtain a 100 ksps output word rate without oversampling and averaging. However, if we oversample and average (decimate) to achieve higher resolution, throughput will be reduced by a factor of the oversampling ratio, *OSR* (see Equation 7). Oversampling by a factor of 256 as we do in the provided example, our output word rate will be 100 ksps/256 = 390 samples per second (390 Hz). Thus, there is a trade-off between resolution and throughput for a given sampling rate. Another trade-off is the reduced CPU bandwidth during each sampling period ($1/f_S$) due to the additional sampling and computations required to achieve the additional resolution.

Summary

If ADC noise can be approximated as white noise, oversampling and averaging can be used to improve the SNR and effective resolution of the measurement. This can be done for static dc measurements and for input signals with higher frequency components. Equation 2 shows that each additional required bit of resolution can be achieved via oversampling by a factor of four, and each additional bit will add approximately 6 db of SNR (Equation 3) at the cost of reduced throughput and increased CPU bandwidth.

Appendix A - Theory of Noise and Oversampling

This section discusses how oversampling and averaging affects in-band noise, and how to calculate the oversampling requirements to obtain a desired SNR or measurement resolution.

How Oversampling and Averaging Improves Performance

Oversampling and averaging is done to accomplish two things: improve SNR and increase the effective resolution (i.e., increase the effective number of bits of the ADC measurement). Both of these are really the same entities. For example, if we have a 12-bit ADC and want to generate codes with 16-bits of resolution, then we can use oversampling and averaging to get the same SNR of a 16-bit ADC. This will increase the *effective number of bits* (ENOB) of the measured data, which is another measure of SNR. Producing a lower noise floor in the signal band, the oversampling and averaging filter allows us to realize 16-bit output words.

How Oversampling Affects In-Band Noise

A sampling frequency f_s will allow signals of interest to be reconstructed at one-half of the sampling frequency (Nyquist Theorem). Thus, if the sampling rate is 100 kHz, then signals below 50 kHz can be reconstructed and analyzed reliably. Along with the input signal, there will be a noise signal (present in all frequencies as white noise) that will *fold* or *alias* into the measured frequency band of interest (frequencies less than one-half of f_s)

$$E(f) = e_{rms} \cdot \left(\frac{2}{fs}\right)^{1/2}$$

where e_{rms} is the average noise power, f_s is the sampling frequency, and E(f) is the in-band ESD.

Equation 4. Energy Spectral Density of In-Band Noise

Equation 4 shows that the Energy Spectral Density (ESD), or *noise floor* of the sampled noise will decrease in the signal band as the sampling frequency is increased.[3]

The Relationship Between Oversampling and Increased Resolution

Given the fixed noise power due to quantization noise, we may calculate the amount of oversampling required to increase the effective resolution. For example, if we want to increase the effective number of bits of a parameter measured with a 12-bit ADC to a 16-bit measurement, then we will want to establish a relationship that allows us to calculate the oversampling requirement. To do so, we first define the characteristics of the noise.

Noise Analysis

To understand the effects of oversampling and averaging on noise, we must first define what the quantization noise will be.

The distance between adjacent ADC codes determines the quantization error. Because the ADC will round to the nearest quantization level, or ADC code:

$$\Delta = \frac{Vref}{2^N}$$

where ***N*** *is the number of bits in the ADC code and* ***Vref*** *is the reference voltage.*

Equation 5. Distance Between ADC codes, or the LSB

The quantization error (e_q) is:

$$e_q \le \frac{\Delta}{2}$$

Assuming the noise approximates *white noise*, the random variable representing the noise is equally distributed with zero mean between ADC codes. Thus, the *variance* is the average noise power calculated [3] :

$$e_{rms}^2 = \int_{-\Delta/2}^{\Delta/2} \left(\frac{e_q^2}{\Delta}\right) de = \frac{\Delta^2}{12}$$

Equation 6. Noise Power Due to Quantization in the ADC

A measure of the sampling frequency compared to the Nyquist frequency (see Equation 1) is the *oversampling ratio* (OSR). This is defined as follows:

$$OSR = \frac{fs}{2 \cdot fm}$$

where fs is the sampling frequency and fm is the highest frequency component of the input signal.

Equation 7. Oversampling Ratio

If the noise is white, then the in-band noise power **at the output of the low-pass filter** is (see Figure 1 on page 1):

$$_0^2 = \int_0^{fm} e_{rms}(f)^2 df = e_{rms}^2\left(\frac{2 \cdot fm}{fs}\right) = \frac{e_{rms}^2}{OSF}$$

vhere n_0 is the noise power output from the ilter.

Equation 8. In-Band Noise Power as a Function of the OSR

Equation 8 shows we can lower the in-band noise power by increasing the *OSR*. Oversampling and averaging does not affect the signal power [1]. Thus, we increase the SNR because oversampling lowers noise power and does not affect signal power.

From Equations 5, 6, and 8, we can derive the following expression relating the noise power to the oversampling ratio and resolution:

$$n_0^2 = \frac{1}{(12 \cdot OSR)}\left(\frac{Vref}{2^N}\right)^2$$

where OSR is the oversampling ratio, N is the number of ADC bits, and Vref is the reference voltage.

Equation 9. Noise Power As a Function [1] of OSR and Resolution

Conversely, given a fixed noise power, we can calculate the required number of bits. Solving Equation 9 for *N*, we obtain Equation 10 that shows how to calculate the number of effective bits given the reference voltage, in-band noise power, and oversampling ratio. [1].

$$N = -\frac{1}{2}\log(OSR)_2 - \frac{1}{2}\log(12)_2 - \frac{1}{2}\log(n_0^2)_2 + \log(Vref)_2$$

Equation 10. Number of Effective Bits As a Function of Reference Voltage, In-Band Noise Power, and Oversampling Ratio

From Equation 10, we observe:

Each doubling of the sampling frequency will lower the in-band noise by 3 dB, and increase the resolution of the measurement by 1/2 bit. [3]

In a practical sense, we measure a signal band-limited to less than 1/2**fs*, then oversample that signal with an oversampling ratio (*OSR*). The resulting samples are then averaged (or decimated) for the resulting output data. For each additional bit of resolution or 6dB of noise reduction, we oversample by a factor of four:

$$f_{os} = 4^w \cdot f_s$$

where ***w*** *is the number of additional bits of resolution desired,* $\boldsymbol{f_s}$ *is the original sampling frequency requirement, and* $\boldsymbol{f_{os}}$ *is the oversampling frequency*

Equation 11. Oversampling Frequency To Add Measurement Resolution

Equation 11 is Equation 2 presented at the beginning of this application note. If we are using the 12-bit on-chip ADC and wish to have the accuracy of a 16-bit ADC, we need an additional 4 bits of resolution. Four factors of four (using Equation 11) is 256. Thus, we need to oversample by a factor of 256 times the Nyquist rate. If the desired signal is band-limited to 60 Hz (f_m=60 Hz), then we must oversample at 120 Hz * 256 = 30.7 kHz. We improve the effective resolution by improving the SNR in our frequency band of interest.

Increasing the sampling rate, or *OSR*, lowers the noise floor in the signal band of interest (all frequencies less than 1/2 of *fs*). The frequency profiles of the quantization noise and input signal are shown in Figure 2. Note when oversampling occurs, less of the noise profile overlaps the input signal profile. Thus, a low-pass filter may be more selective without affecting the input signal, and filter more of the in-band noise. The noise power at the output of the filter is calculated using Equation 8. This is the noise level lowered due to the oversampling and averaging filter. This is depicted in Figure 3.

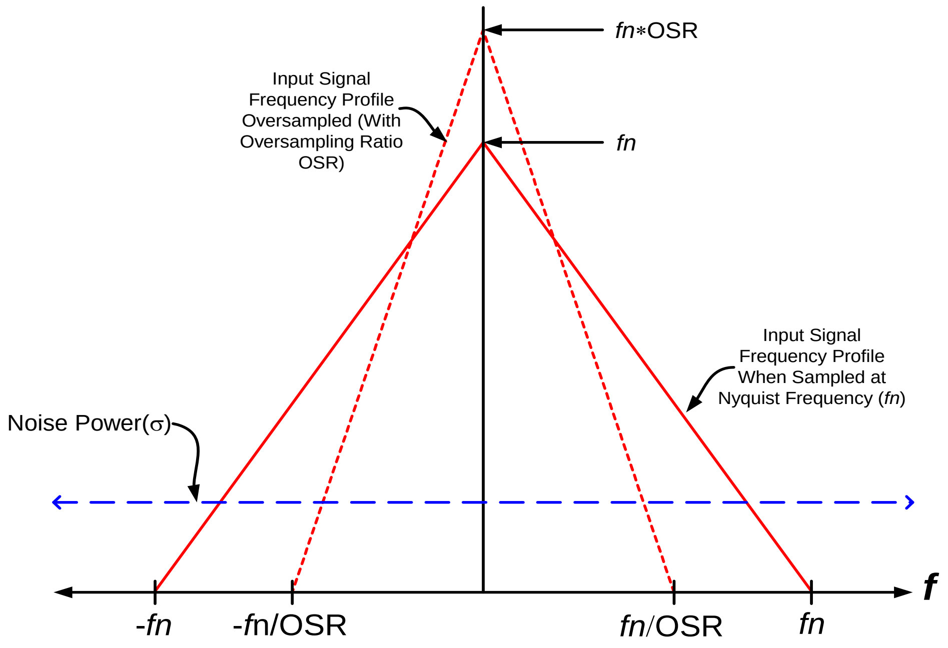

fn∗OSR
Input Signal Frequency Profile Oversampled (With Oversampling Ratio OSR)
fn
Input Signal Frequency Profile When Sampled at Nyquist Frequency (*fn*)
Noise Power(σ)
f
-*fn*
-*fn*/OSR
fn/OSR
fn

Figure 2. Frequency Profiles of Input Signals Sampled at Nyquist Frequency, Oversampled Frequency, and the Quantization Noise Floor

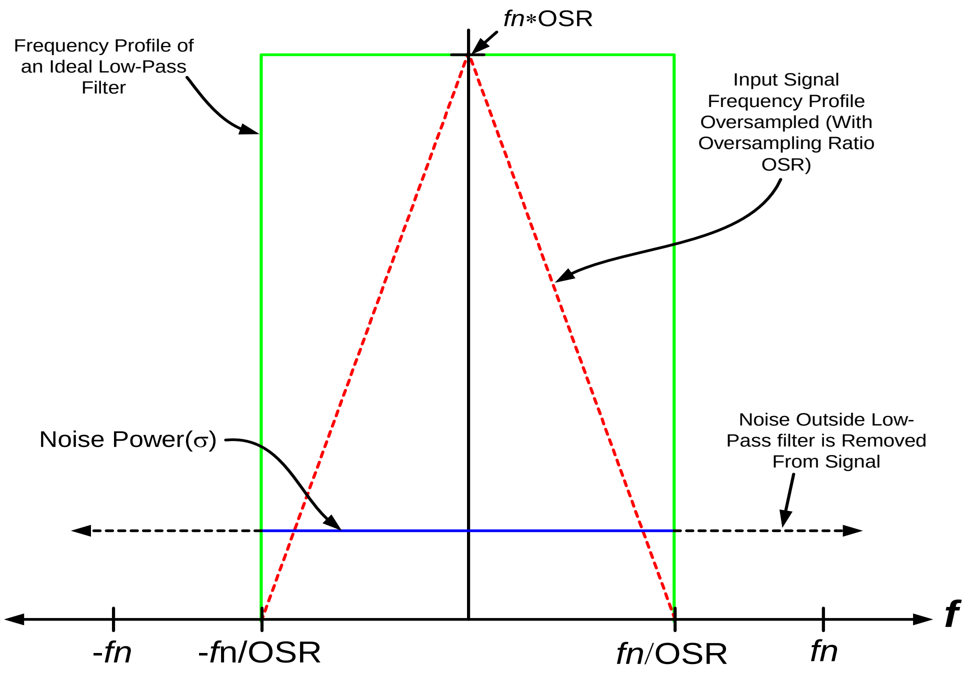

fn∗OSR
Frequency Profile of an Ideal Low-Pass Filter
Input Signal Frequency Profile Oversampled (With Oversampling Ratio OSR)
Noise Power(σ)
Noise Outside Low-Pass filter is Removed From Signal
f
-*fn*
-*fn*/OSR
fn/OSR
fn

Figure 3. Frequency Profile Of Oversampled Signal and an Ideal Low-Pass Filter Removing Noise

The noise that is filtered between *fm*, and *fm/OSR*. Without oversampling, the filter would not have removed this noise. The output is also downsampled (decimated) by a factor of the *OSR* (see Figure 1) to the original Nyquist frequency, *fn*. This will give the input signal its frequency profile as if sampled at the Nyquist frequency, and the noise profile a lower value (if filtered) of e_{rms}/OSR (see Figure 4).

Calculating Signal To Noise Ratio

Signal-to-noise ratio is defined as the ratio of the rms signal power to the rms noise power in decibels (dB). No matter how carefully we work to remove sources of ADC noise, quantization noise will always be present. Thus, ideal SNR is calculated based on quantization noise with no oversampling and averaging. Equation 5 shows that the higher the resolution of the ADC, the lower the quantization error and therefore, the lower the quantization noise. The more bits in the ADC, the better the SNR can be. As shown in the previous sections, oversampling and averaging lowers the in-band noise, improving the SNR and increasing the effective number of bits (ENOB). ENOB is another measure of SNR, and both can be calculated to determine specifications and oversampling requirements needed to meet these specifications.

In order to get the best case SNR, the dynamic range of the input signal must match the reference voltage (V_{ref}). If we assume the best case input signal to be a full-scale sine wave, then it's rms value as a function of V_{ref} will be:

$$V_{rms} = \frac{V_{ref}}{2\sqrt{2}}$$

Equation 12. Input Signal RMS Value as a Full-Scale Sine Wave

From the noise power calculation in Equation 9, we determine the rms noise power as a function of the number of bits, *N* (not oversampled) to be:

$$n_0 = \frac{V_{ref}}{2^N\sqrt{12}}$$

Equation 13. RMS Noise Power Value

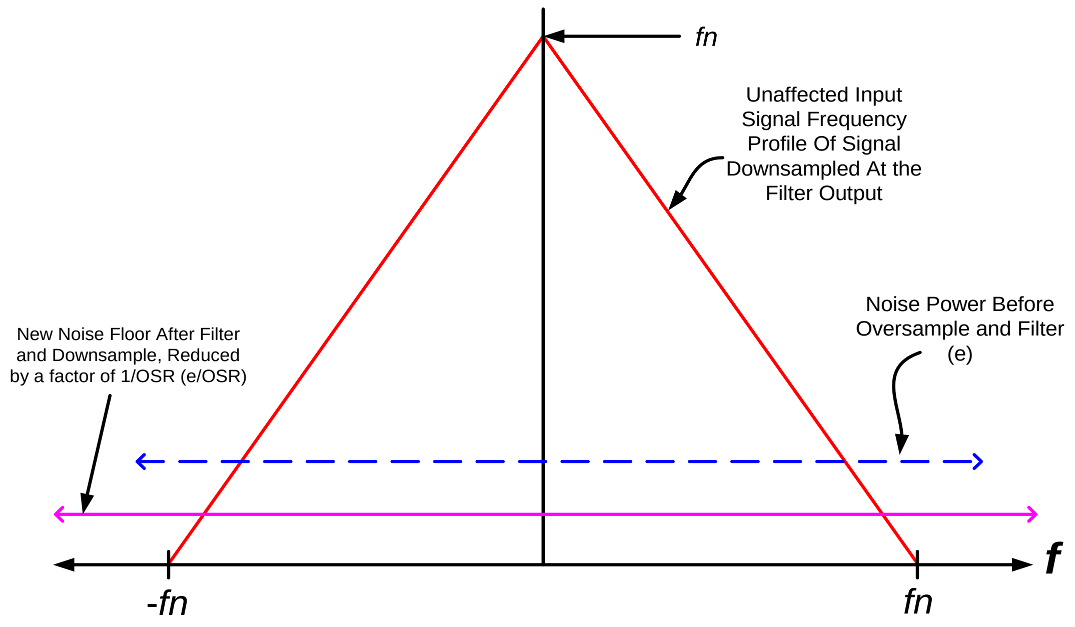

fn
Unaffected Input Signal Frequency Profile Of Signal Downsampled At the Filter Output
Noise Power Before Oversample and Filter (e)
New Noise Floor After Filter and Downsample, Reduced by a factor of 1/OSR (e/OSR)
f
-fn
fn

Figure 4. Oversampled Signal After Filter and Downsampled to the Nyquist Frequency Showing Lowered Noise Floor

The SNR in *dB* is then calculated as follows:

$$NR = 20 \cdot \log\left(\frac{V_{rms}}{n_0}\right) = 20 \cdot \log\left(\frac{2^N\sqrt{12}}{2\sqrt{2}}\right)$$

Equation 14. SNR as a Function of the Number of Bits, N

When oversampling, we may substitute the effective number of bits (*ENOB*) for *N* in Figure 14. Simplifying Equation 14 and substituting the term *ENOB* for *N* we obtain the well known result in decibels:

$$SNR(dB) = (6.02 \cdot ENOB) + 1.76$$

where ENOB is the effective number of bits of the measurement

Equation 15. SNR Calculation as a Function of ENOB

Averaging To Increase the Effective Resolution of a dc Measurement

Thus far, we have considered measuring signals within some frequency band of interest, f_m. However, our goal may be to measure a relatively static dc signal (such as a temperature or strain gauge output). If we wish to measure a signal that is relatively static, that is, the dominant frequency is near dc, we can still improve the effective resolution by oversampling and averaging [2].

Applications Measuring a Static Voltage

If a weigh scale must measure a wide range of weights, yet still be able to discern small changes in weight, then oversampling and averaging can increase the effective resolution of the measurement. As another example, if the ADC must measure the output of a temperature sensor, the temperature range may be large, yet the system application may have to respond to small changes.

Oversampling and Averaging as an Interpolative Filter

Averaging data from an ADC measurement is equivalent to a low-pass digital filter with subsequent downsampling (see Figure 1 on page 1). Digital signal processing that oversamples and low-pass filters a parameter is often referred to as *interpolation*. In this sense, we use oversampling to interpolate numbers between the 12-bit ADC codes. The higher the number of samples averaged, the more selective the low-pass filter will be, and the better the interpolation.

Appendix B - When Oversampling and Averaging Will Work

This section discusses guidelines to determine if oversampling and averaging will be effective for a given application.

The analog-to-digital data conversion process introduces noise. Oversampling and averaging can reduce certain types of noise, thereby increasing the SNR and effective resolution of the data conversion. Not all applications will benefit from oversampling and averaging. To understand which ADC measurements will benefit from oversampling, we must understand the type and characteristics of the noise present in a given system.

Noise Requirements For Effective Oversampling

Oversampling and averaging can improve the SNR and increase the effective resolution of the analog-to-digital data measurement. However, this will work only if the ADC noise can be approximated as *white noise* [2] [3]. If the input signal changes *randomly* from sample to sample, by *amounts* (amplitude) comparable to the code size (1 LSB), and the input signal has *equal probability* of being anywhere between two adjacent codes, then the noise can be modeled as approximating *white noise*. White noise is characterized as having a uniform power spectral density over the frequency band of interest. When the noise can be approximated as white noise, then oversampling and averaging can improve the SNR and increase the effective resolution of the data.

If the overall noise is not *stationary*, (e.g., systems that have some correlation due to feedback), then oversampling and averaging may not be effective. Additionally, if the quantization noise is comparable to sources of white noise (i.e., thermal and shot noise is small compared to the quantization noise), then oversampling and averaging may not be effective. This situation is typical when using lower resolution ADCs (e.g., 8-bit ADCs). In this case, the thermal noise does not have sufficient amplitude to cause the input signal to change randomly with equal probability between codes, because the code width Δ (Equation 5), is too large. Some applications will inject noise into the signal or process intentionally to overcome this effect. This is referred to as *dithering*.

Histogram Analysis

Most applications that measure a signal using a 12-bit ADC will benefit from oversampling and averaging techniques. A practical means of determining if the noise characteristics are appropriate is to analyze the ADC output data using a *histogram* (see Figure 5 below).[2] This histogram shows how many samples in a set from an ADC resulted in each ADC code. If the input signal is a constant dc voltage value, the histogram will approximate a gaussian probability distribution function (PDF) if the noise is white, as shown in Figure 5.[2] Due to the input voltage, the “bin” for code 1024 received the greatest number of samples, while surrounding codes received some samples due noise. Because the histogram approximates a Gaussian PDF (shown as a blue dotted line in Figure 5), the noise approximates white noise, and this system can benefit from oversampling and averaging techniques.

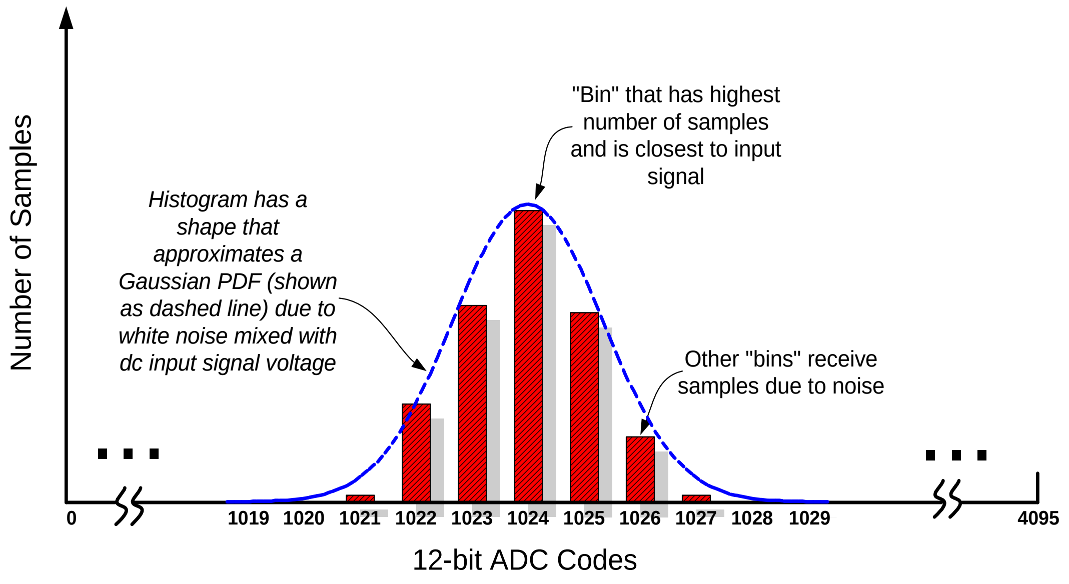


Figure 5. Histogram of ADC samples: dc Input With White Noise

A system with insufficient noise (besides the quantization noise) will result in a histogram with all samples going to only one bin, or code. Oversampling and averaging may not be helpful in such a system.

If the noise is correlated or the ADC's transfer function is non-linear (e.g., power supply noise, poor INL, etc.), the histogram may not approximate a Gaussian PDF, such as the one in Figure 6). In this case, oversampling and averaging may not helpful.

In summary, if the combined sources of noise in the resultant ADC codes approximates white noise, a histogram of the samples will approximate a Gaussian PDF, and oversampling and averaging will improve the SNR and increase the effective number of bits of the signal measurement.

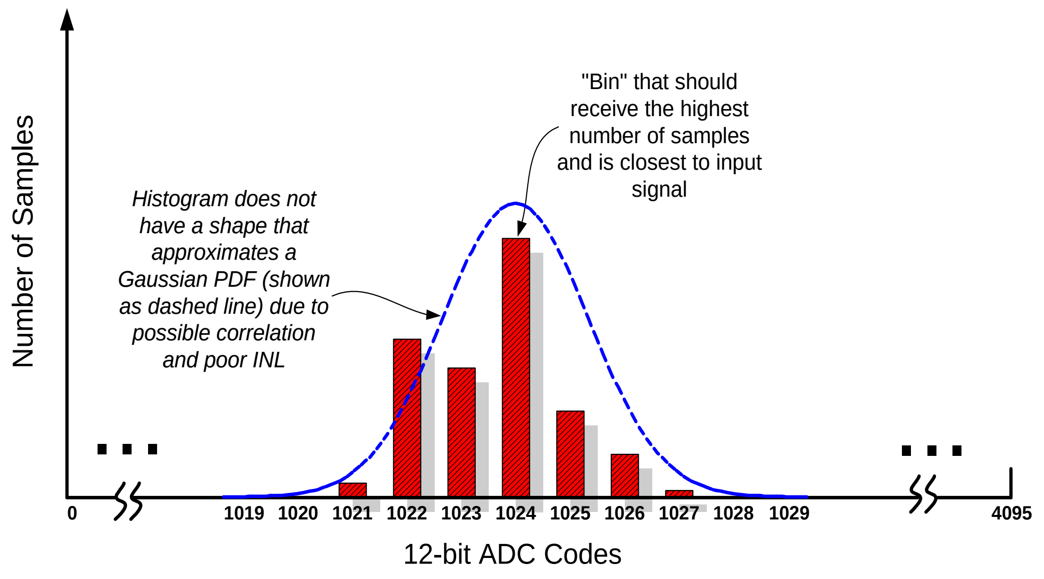


Figure 6. Histogram of ADC Samples Not Optimal For Oversampling and Averaging Techniques

Appendix C - Example Code

```
//-----------------------------------------------------------------------------
// AN018_SW.c
//-----------------------------------------------------------------------------
// Copyright 2001 Cygnal Integrated Products, Inc.
//
// AUTH: BW
//
// This program outputs the C8051Fxxx die temperature out the hardware
// UART at 115.2kbps. Assumes an 18.432MHz crystal is attached between
// XTAL1 and XTAL2.
//
// The ADC is configured to look at the on-chip temp sensor.  The sampling
// rate of the ADC is determined by the constant <SAMPLE_RATE>, which is given
// in Hz.  The maximum value of <SAMPLE_RATE> is limited to ~86kHz due to
// the choice of 18.432MHz crystal (SAR clock = SYSCLK / 16 = 1.152MHz. One
// conversion takes 16 SAR clocks --> 72kHz sampling rate).
//
// The ADC End of Conversion Interrupt Handler retrieves the sample
// from the ADC and adds it to a running accumulator.  Every 256
// samples, the ADC updates and stores its result in the global variable
// <result>.  The sampling technique of adding a set of values and
// decimating them (posting results every 256th sample) is called accumulate
// and dump.  It is easy to implement and requires very few resources.
//
// For each power of 4, you gain 1 bit of effective resolution.
// For a factor of 256, gain you 4 bits of resolution: 4^4 = 256.
// Also, to properly scale the result back to 16-bits, perform a right
// shift of 4 bits.
//
// Target: C8051F00x or C8051F01x
// Tool chain: KEIL C51 6.03 / KEIL C51 EVAL version
//

//-----------------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------------

#include <stdio.h>



#include <c8051f000.h>                 // SFR declarations

//-----------------------------------------------------------------------------
// 16-bit SFR Definitions for F00x, F01x
//-----------------------------------------------------------------------------

sfr16 DP       = 0x82;                 // data pointer
sfr16 TMR3RL   = 0x92;                 // Timer3 reload value
sfr16 TMR3     = 0x94;                 // Timer3 counter
sfr16 ADC0     = 0xbe;                 // ADC0 data
sfr16 ADC0GT   = 0xc4;                 // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                 // ADC0 less than window
sfr16 RCAP2    = 0xca;                 // Timer2 capture/reload
sfr16 T2       = 0xcc;                 // Timer2
sfr16 DAC0     = 0xd2;                 // DAC0 data
```

```
sfr16 DAC1     = 0xd5;                 // DAC1 data

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------

#define SYSCLK       18432000          // SYSCLK frequency in Hz
#define BAUDRATE     115200            // Baud rate of UART in bps
#define SAMPLE_RATE  100000            // Sample frequency in Hz

#define LED          P1.6              // LED=1 means ON


//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

void SYSCLK_Init (void);
void PORT_Init (void);
void UART_Init (void);
void ADC_Init (void);
void TIMER3_Init (int counts);
void ADC_ISR (void);

//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------

long result;                            // Output result from oversmapling and
                                       // averaging 256 samples from the ADC for
                                       // 16-bit measurement resolution

//-----------------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------------

void main (void) {
   long temp_copy;
   int temp_int;                       // integer portion of temperature
   int temp_frac;                      // fractional portion of temperature (in
                                       // hundredths of a degree)

   WDTCN = 0xde;                       // disable watchdog timer
   WDTCN = 0xad;

   SYSCLK_Init ();                     // initialize oscillator
   PORT_Init ();                       // initialize crossbar and GPIO
   UART_Init ();                       // initialize UART
   TIMER3_Init (SYSCLK/SAMPLE_RATE);   // initialize Timer3 to overflow at
                                       // sample rate
   ADC_Init ();                        // init ADC

    ADCEN = 1;                          // enable ADC

   result = 0L;                        // initialize temperature variable

   EA = 1;                             // Enable global interrupts

    while (1) {
      temp_copy = result;              // Get most recent sample to convert
                                       //  the ADC code to a temperature
      temp_copy -= 0xa381;             // correct offset to 0deg, 0V
      temp_copy *= 0x01a9;             // 2.86mV/degree C
      temp_copy *= 100;                // convert result to 100ths of a degree C
      temp_copy = temp_copy >> 16;     // divide by 2^16
      temp_int = temp_copy / 100;      // Seperate integer and fractional components
      temp_frac = temp_copy - (100 * temp_int);
       printf (“Temperature is %d.%d\n”, (int) temp_int, (int) temp_frac);
   }
}

//-----------------------------------------------------------------------------
// Initialization Subroutines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// SYSCLK_Init
//-----------------------------------------------------------------------------
//
// This routine initializes the system clock to use an 18.432MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
   int i;                              // delay counter

   OSCXCN = 0x67;                      // start external oscillator with
                                       // 18.432MHz crystal

   for (i=0; i < 256; i++) ;           // XTLVLD blanking interval (>1ms)

   while (!(OSCXCN & 0x80)) ;          // Wait for crystal osc. to settle

   OSCICN = 0x88;                      // select external oscillator as SYSCLK
                                       // source and enable missing clock
                                       // detector
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
//
// Configure the Crossbar and GPIO ports
//
void PORT_Init (void)
{
   XBR0    = 0x07;                     // Enable I2C, SPI, and UART
   XBR1    = 0x00;
   XBR2    = 0x40;                     // Enable crossbar and weak pull-ups
   PRT0CF |= 0xff;                     // enable all outputs on P0 as push-pull
                                       // push-pull; let xbar configure pins
                                       // as inputs as necessary
   PRT1CF |= 0x40;                     // enable P1.6 (LED) as push-pull output
}

//-----------------------------------------------------------------------------
// PORT_Init
//-----------------------------------------------------------------------------
```

```
//
// Configure the UART using Timer1, for <baudrate> and 8-N-1.
//
void UART_Init (void)
{
   SCON   = 0x50;                     // SCON: mode 1, 8-bit UART, enable RX
   TMOD   = 0x20;                     // TMOD: timer 1, mode 2, 8-bit reload
   TH1    = -(SYSCLK/BAUDRATE/16);    // set Timer1 reload value for baudrate
   TR1    = 1;                        // start Timer1
   CKCON |= 0x10;                     // Timer1 uses sysclk as time base
   PCON  |= 0x80;                     // SMOD = 1
   TI     = 1;                        // Indicate TX ready
}

//-----------------------------------------------------------------------------
// ADC_Init
//-----------------------------------------------------------------------------
//
// Configure A/D converter to use Timer3 overflows as conversion source, to
// generate an interrupt on conversion complete, and to use right-justified
// output mode.  Enables ADC end of conversion interrupt. Leaves ADC disabled.
//
void ADC_Init (void)
{
   ADC0CN = 0x04;                     // ADC disabled; normal tracking
                                      // mode; ADC conversions are initiated
                                      // on overflow of Timer3; ADC data is
                                      // right-justified
   REF0CN = 0x07;                     // enable temp sensor, on-chip VREF,
                                      // and VREF output buffer
   AMX0SL = 0x0f;                     // Select TEMP sens as ADC mux output
   ADC0CF = 0x61;                     // ADC conversion clock = sysclk/8

   EIE2 |= 0x02;                      // enable ADC interrupts
}

//-----------------------------------------------------------------------------
// TIMER3_Init
//-----------------------------------------------------------------------------
//
// Configure Timer3 to auto-reload at interval specified by <counts> (no
// interrupt generated) using SYSCLK as its time base.
//
void TIMER3_Init (int counts)
{
   TMR3CN = 0x02;                     // Stop Timer3; Clear TF3;
                                      // use SYSCLK as timebase
   TMR3RL  = -counts;                 // Init reload values
   TMR3    = 0xffff;                  // set to reload immediately
   EIE2   &= ~0x01;                   // disable Timer3 interrupts
   TMR3CN |= 0x04;                    // start Timer3
}

//-----------------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------
// ADC_ISR
```

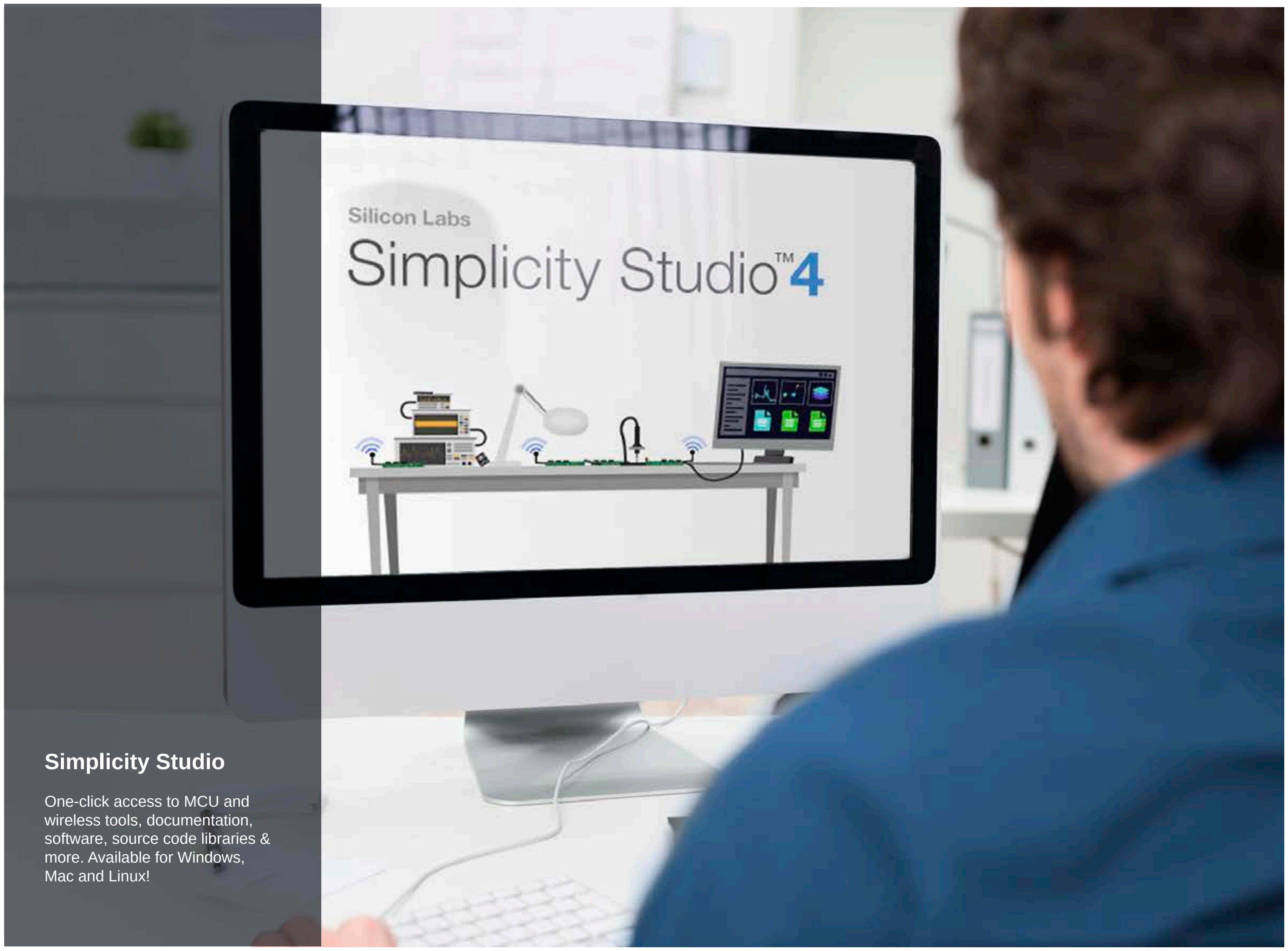
```
//-----------------------------------------------------------------------------
//
// ADC end-of-conversion ISR
// Here we take the ADC sample, add it to a running total <accumulator>, and
// decrement our local decimation counter <int_dec>.  When <int_dec> reaches
// zero, we calculate the new value of the global variable <result>,
// which stores the accumulated ADC result.
//
void ADC_isr (void) interrupt 15
{
   static unsigned int_dec=256;        // integrate/decimate counter
                                       // we post a new result when
                                       // int_dec = 0
   static long accumulator=0L;         // heres where we integrate the
                                       // ADC samples

   ADCINT = 0;                         // clear ADC conversion complete
                                       // indicator

    accumulator += ADC0;                // read ADC value and add to running
                                       // total
   int_dec--;                          // update decimation counter

   if (int_dec == 0) {                 // if zero, then decimate
      int_dec = 256;                   // reset counter
      result = accumulator >> 4;       // Shift to perform the divide operation
      accumulator = 0L;                // dump accumulator
   }
}
```

References

[1] A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing*, New Jersey: Prentice Hall, 1999 ed.

[2] J. Lis, *Noise Histogram Analysis*, Cirrus Logic Application Note AN37

[3] J.C. Candy and G.C. Temes, *Oversampling Methods for A/D and D/A Conversion*, IEEE Transactions on Circuits and Systems, June 1987 (Beginning discussion on the effects of oversampling on in-band noise).



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com