Ben Sawtelle

Professor Xiaoguang Liu

EEC 134

26 March 2017

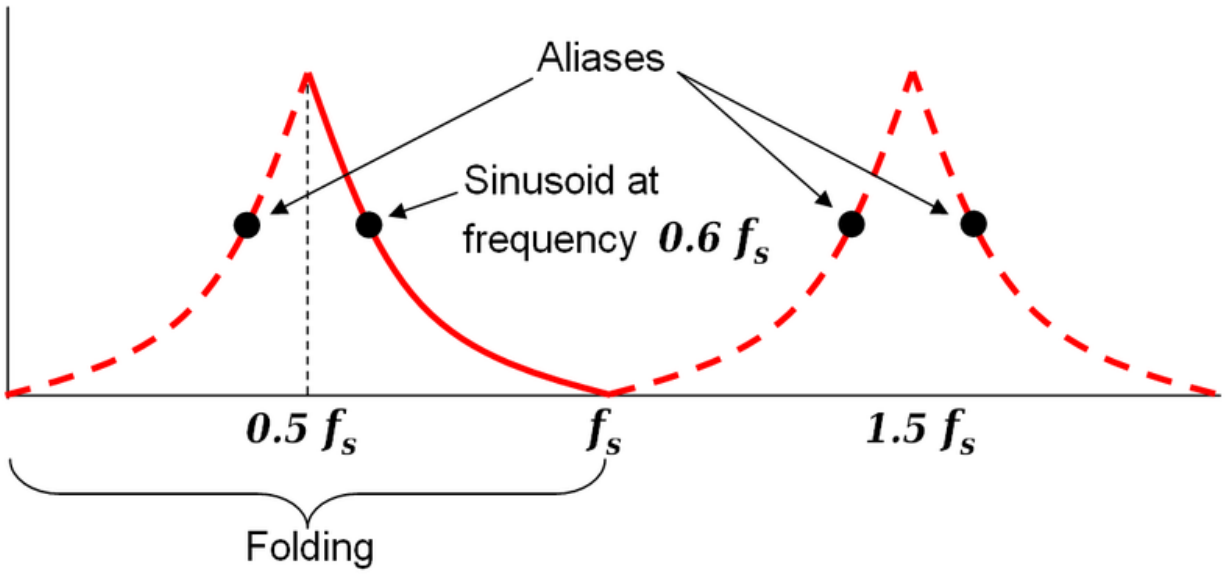
Basics of ADCs and FFTs

Introduction

This application note will cover a few key points within the digital signal processing side of the radar design including data acquisition with ADCs and FFTs. These concepts are complex in themselves, so the intimate details will be left out in place of a general overview that should be enough to create a simple radar output. The goal is to mostly highlight a few notes that I think would have been helpful to know at the start of the project as well as a few basic things that were already in the course slides.

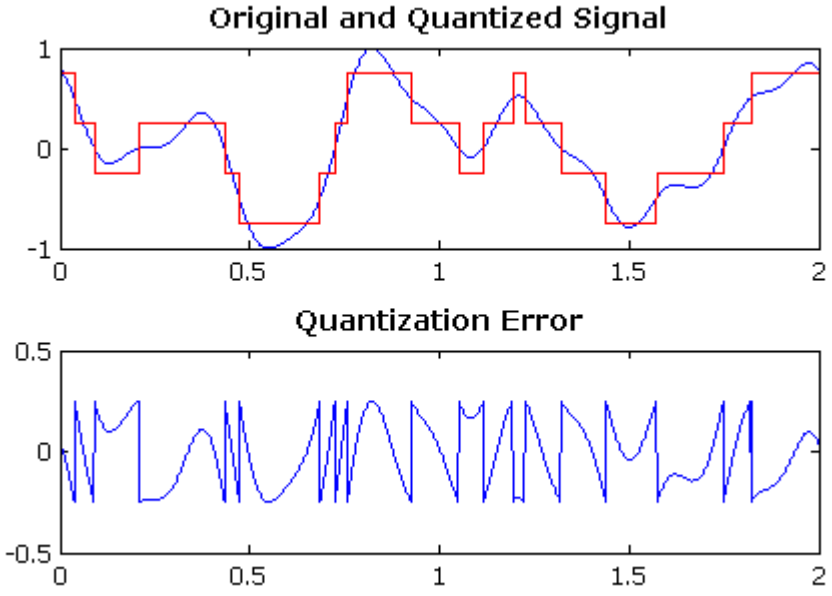
ADC Bits and Quantization Error

The first requirement for successful data processing is to find the voltage range that the signal can be read and analyzed at. The first detail is that the frequency that the ADC samples at must be at least two times the frequency that is being sampled (at least without doing any tricks for periodic signals). If this criterion isn't fulfilled, aliasing can occur which will add noise and strange frequency readings to the system. This happens in the form of the frequencies above the sampling rate or above half the sampling rate "folding" around the sampling frequency.



https://en.wikipedia.org/wiki/Aliasing

The voltage of the signal is important as well. The ADC can often be adjusted to set a max voltage reading to allow the signal to be broken into as many useful bits as possible e.g. if the max signal voltage is 0.5V, then the max ADC voltage may be able to be set to 0.5V to make sure that the ADC can break the signal into as many useful bits as possible. The number of bits used reduces the quantization error and making sure the voltage is correct ensures the error is as low as possible for the ADC.



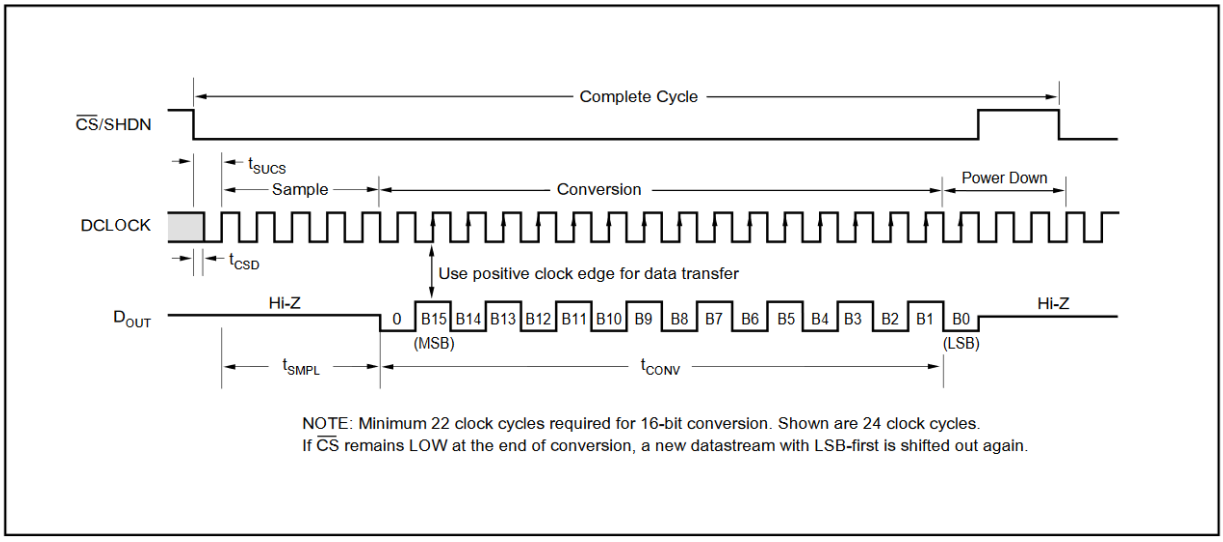
https://en.wikipedia.org/wiki/Quantization_(signal_processing)

This diagram shows the error introduced from recreating the signal using 4 voltage levels (2-bits). The graph below is the difference between the recreated signal and the original signal.

Communicating with the ADC

ADCs can have varying methods of outputting the data. The quickest way to transfer data is using a “flash ADC” which is also known as a “direct-conversion ADC”. These ADCs outputs the bit data in parallel, meaning there are physical pins for each bit output e.g. an 8-bit ADC will have eight output pins. Other methods are through serial communication such as SPI and I2C. Serial communications generally have a clock pin and a data pin (or pins) that allow the devices to send data one bit after the other on one data bus, so reading an 8-bit ADC using SPI or I2C would use two or three pins rather than eight https://learn.sparkfun.com/tutorials/serial-communication. These methods can take a few more clock cycles, but they use significantly less pins. The time for a transfer is generally minimal, since most microcontrollers have fast clock

speeds. An example of a 16-bit ADC SPI timing diagram can be found below.



Each line represents the logic level of a pin. This diagram has a chip select (CS), a clock (DCLOCK), and a data pin (Dout). The chip select is switched low to begin the transfer, and the clock pin is toggled six times before the data can be read. The data can be read when the clock goes from low to high (a positive clock edge). The chip select can then be set high and the clock pulsed four times to turn the ADC off until another read will take place. The transfer cycle is a bit odd compared to some more simple SPI transfers, but the pins can be toggled directly to follow the timing diagram. The code on the next page is an example of directly toggling pins in Arduino rather than using a hardware based SPI method. There are specific timings that may have to be accounted for (t_{sucs}), but the Arduino write cycles are slow enough that no additional time needed to pass. These timings are specified in tables on the datasheet and allow the max transfer rate of the ADC to be calculated.

```
void readADC(void) {

  bool dataBit;
  uint16_t data;
  digitalWrite(clock_pin, HIGH);
  digitalWrite(cs_pin, LOW);
  digitalWrite(clock_pin, LOW);

  for (int x = 0; x < 6; x++) {
    digitalWrite(clock_pin, HIGH);
    //delay(1);
```

```
  digitalWrite(clock_pin, LOW);
 }

 for (int x = 0; x < 16; x++) {
  digitalWrite(clock_pin, HIGH);
  dataBit = digitalRead(data_pin);
  digitalWrite(clock_pin, LOW);
  data = (data | (dataBit << (15 - x)));
 }

 digitalWrite(cs_pin, HIGH);
 for (int x = 0; x < 3; x++) {
  digitalWrite(clock_pin, HIGH);
  digitalWrite(clock_pin, LOW);
 }

//part of an interrupt routine, turns itself off after a specific number of points have been collected
 realTime[ADCCount] = data;
 ADCCount++;
 if(ADCCount >= FFTPoints){
  noInterrupts();
  ADCCount = 0;
  ADCFull = true;
 }
 return;
}
```

Fast Fourier Transforms

After the data has been sampled and retrieved, the data needs to be converted into a frequency spectrum. This can be done fairly simply if the data is a power of two https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm. A very detailed explanation of FFTs as well as sampling can be found at http://www.peteronion.org.uk/FFT/FastFourier.html, but this will go over some of the basics that are important for the FMCW radar application. An FFT algorithm stands for a fast Fourier transform and allows the data to be converted into the frequency domain, which will be used to

find the distance that the radar traveled. The number of points that are transformed determines the resolution of the FFT. The formula for the resolution is

$$Resolution = \frac{Fs}{N}$$

where F_s is the sampling frequency of the signal and N is the number of points. A signal sampled at 1024 samples/s transformed using 256 points will have a frequency resolution of 4 Hz in the frequency domain. This means that you will end up with a 256 point spectrum from 0 Hz to 1024 Hz with a spacing of 4Hz between each point. Anything above 512 Hz would be discarded, assuming this is data from an ADC, since it would be within the range of aliasing. To increase the resolution the sampling frequency should be decreased or the number of points increased, but both solutions will lengthen the amount of time taken for the signal to be sampled. The sampling time is expressed as the inverse of the resolution.

$$Sampling\ time = \frac{N}{Fs}$$

This means that the better resolution you have, the longer the signal takes to be sampled. The 4Hz sample example from before will take 250ms to sample. There is a tradeoff between the bandwidth that can be sampled, the number of points, the sampling time, and the resolution.

Memory Requirements for FFts

The data acquired must be stored somewhere and it will add up very quickly. A 1024 point FFT is comprised of a real portion and an imaginary portion, which if they are stored as two 1024 point float variable types (often 4 bytes) in the microcontroller, this will be 8192 bytes of information. This is already more data than something like an Arduino can store, so the deciding the resolution required for your FFT is important for deciding which microcontroller to choose. The amount of processing time can be lengthy for larger calculations as well.