SOFTWARE DEFINED RADIO

Larry Du
Joshua Yun

Dec 12th, 2023

ECE 395
UIUC

Abstract

We attempted to develop a software defined radio receiver that would receive radio signals over most of the amateur HF spectrum, with the goal of specifically demodulating amateur AM SSB from the 3.5 Mhz band to the 50 Mhz band. The radio used a direct Zero-IF homodyne architecture with quadrature mixing. This analog mixing scheme, along with using a microcontroller ADC sampling fast enough for the baseband AM signal allowed for many complex analog filters to be removed from the circuit design, while theoretically removing much of the electrical noise from the design, improving accuracy.

Table of Contents

Functional Overview

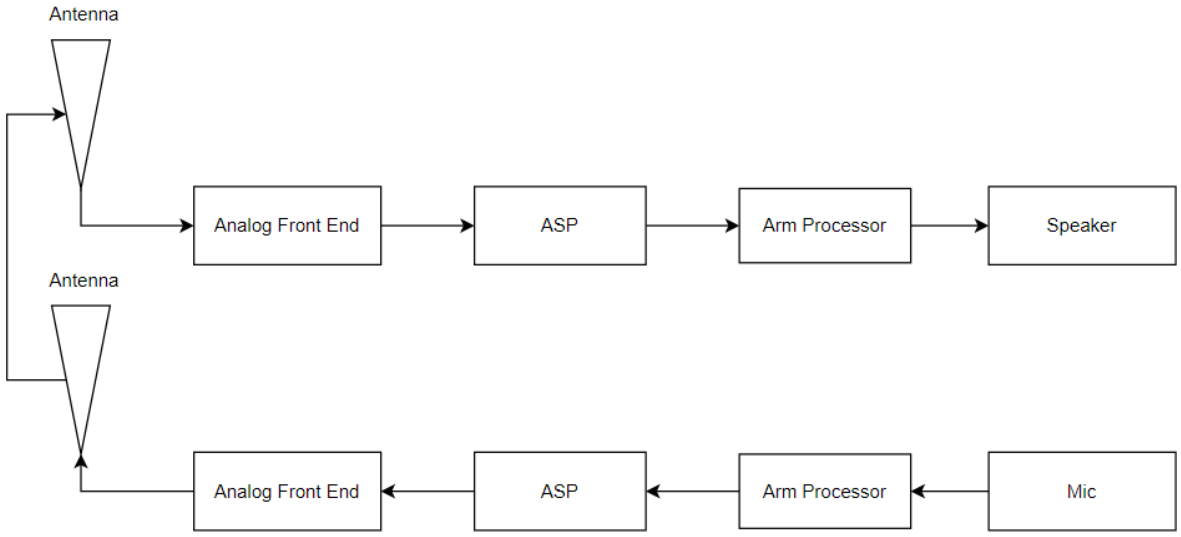
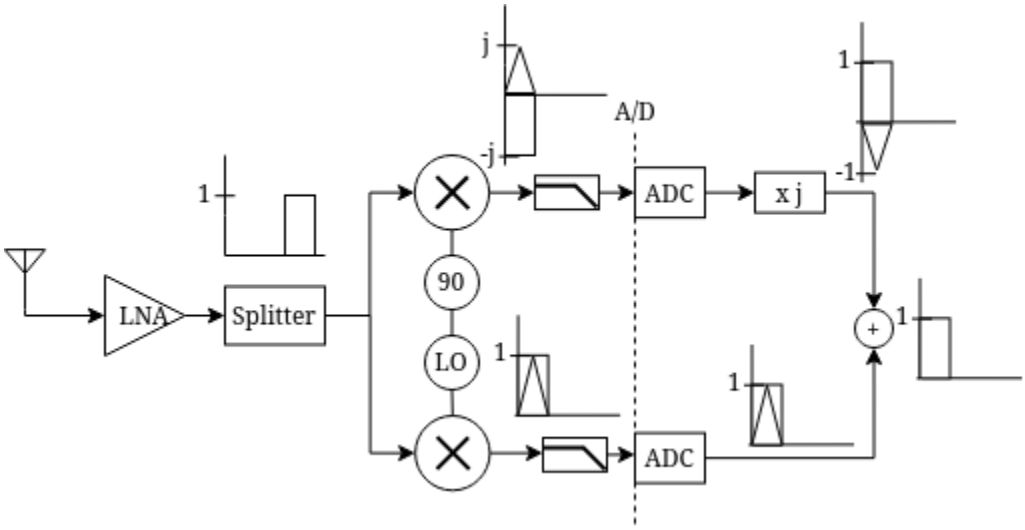


Figure 1. Block Diagram of Architecture.

The basic architecture is shown above, with the transmitter below and the receiver above. Due to time constraints, the transmitter was unable to be designed and made in time. The idea with the radio project was to have an antenna capture RF waves in the HF frequency regime, downconvert them and perform limited analog signal processing, before having the arm processor perform further signal processing on the resulting signal.

Analog Front End Design

Theory of Operation



The main architecture of the front end design was to initially boost the signal from the entire RF spectrum so that it can be further processed. These boosted frequencies are mixed with two cosine waves of the same frequency, with one being phase shifted 90 degrees relative to the other. As shown in the frequency graphs, multiplying by cosine shifts the frequency to baseband, while also shifting another negative frequency up to baseband, while sine does the same while multiplying the positive frequency by j and negative by -j. These resultant signals represent I and Q respectively which are then converted into the digital domain. The Q signal then gets multiplied by j using a digital hilbert filter, which changes the frequency domain to be real again, and adds it with I to eliminate the image station represented by the triangle.

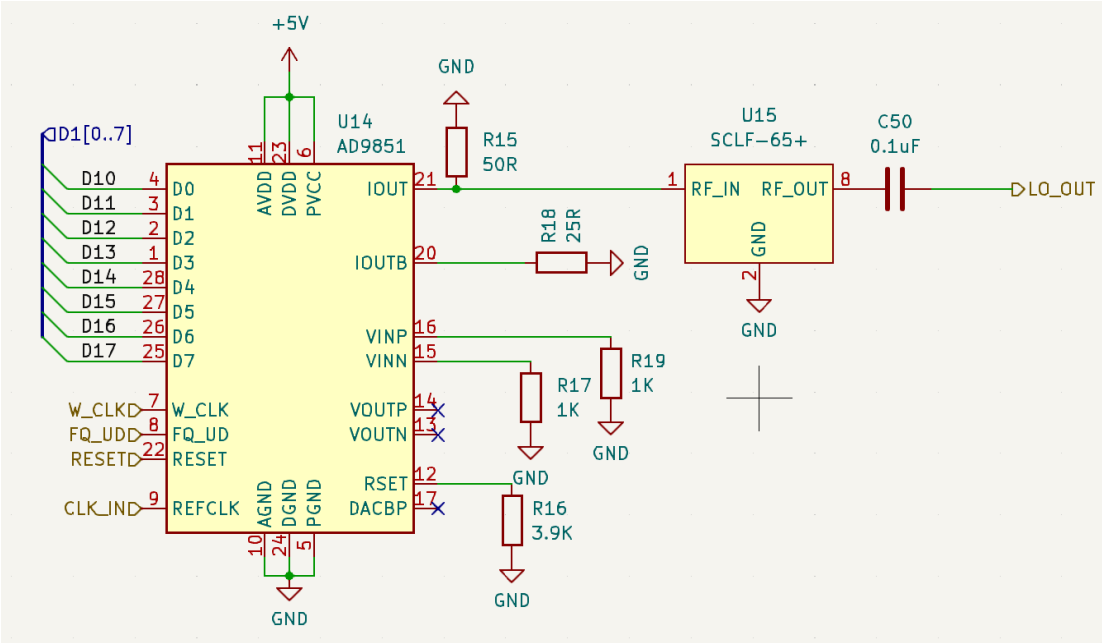
Specific Design Considerations

When doing normal schematic design, there were no special considerations other than the part selection itself, and whether it meets the requirements of the design.

The PCB had many more design considerations that were required for correct design. One of the larger considerations were impedance matching, where the trace widths were required to be a certain size consistently, so as to prevent signal reflections from occurring within the circuit. These impedance traces can be calculated using the JLCPCB manufacturer’s calculator, as well as the KiCAD impedance matching calculator.

Implementation

Local Oscillator

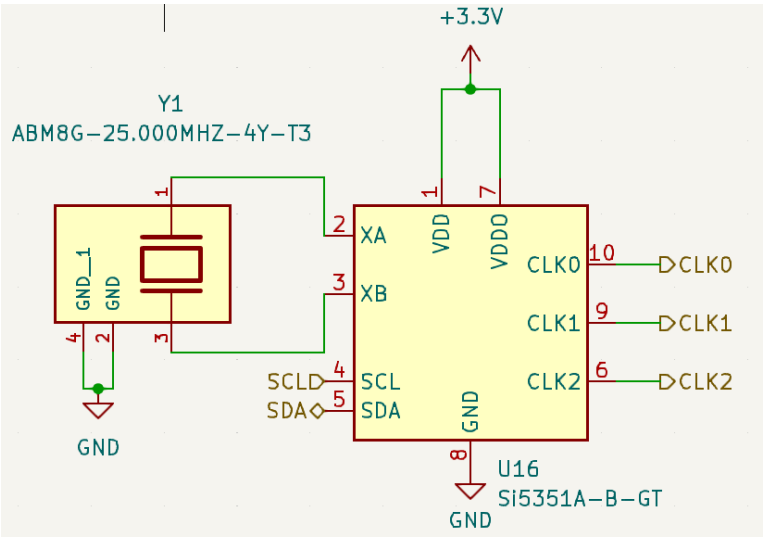


The local oscillator circuit was provided using the AD9851 direct digital synthesis (DDS) chip that was able to generate a sine wave using an input clock signal which drove a DAC at the specified frequency of the sine wave. Since the output waveform was a DAC, it was required that the output of the DAC went through a low pass filter (SCLF-65+) which removed the high frequency DAC artifacts. The DAC was also current driven, meaning that a 50 Ohm resistor needed to be placed connected to ground such that it could be impedance matched with all the other RF circuits. There are two of these chips found on the board itself, to generate the cosine and the sine wave, and since they are controlled by the STM32 and governed by the same clock, they can be programmed to be precisely 90 degrees phase offset from each other.

Mixing Circuit

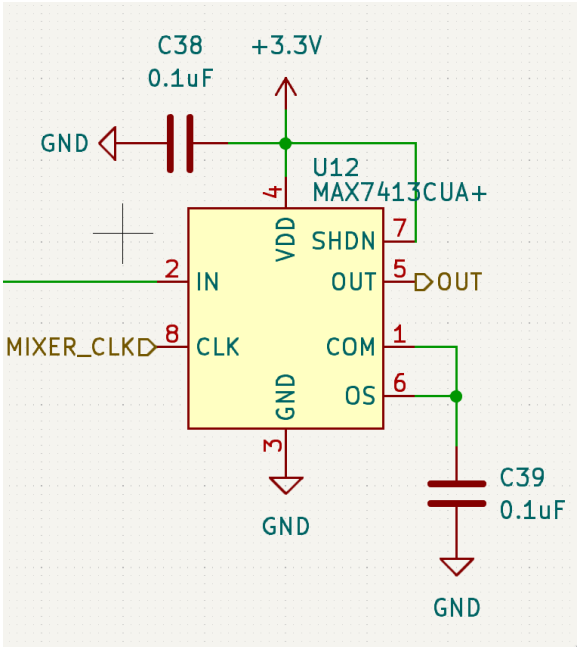
The Job of the mixing circuit was simply to take the frequencies generated by the local oscillator DDS chip and mix them with the signal received from the amplification and splitter stage that connects to the antenna. It is implemented using a double balanced mixer from Mini-circuits (ADE-1MHW+)

Clock Generator Circuit



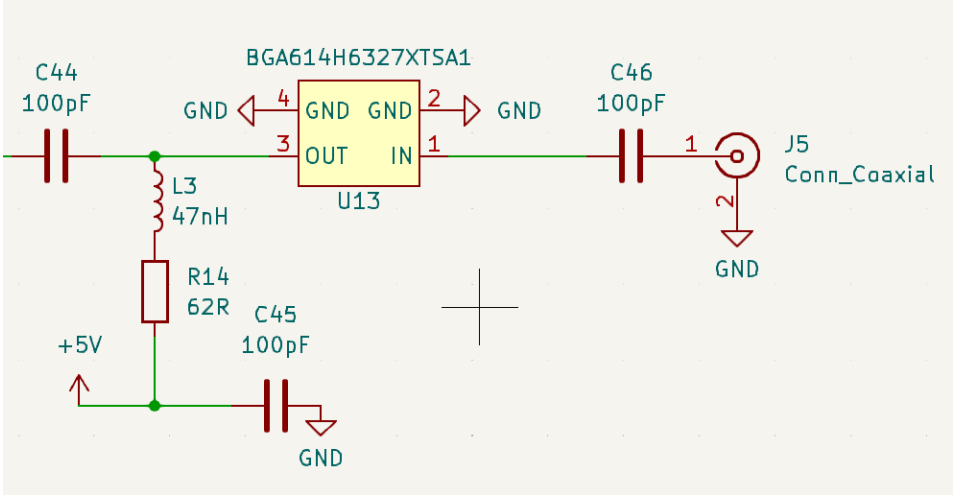
The clock generator was used to create the clocks for the corner frequencies of the low pass filters as well as the clocks for the DDS sine wave synthesizer. It was controlled using I^2C using a process explained later in the document.

Nyquist Low Pass Filters



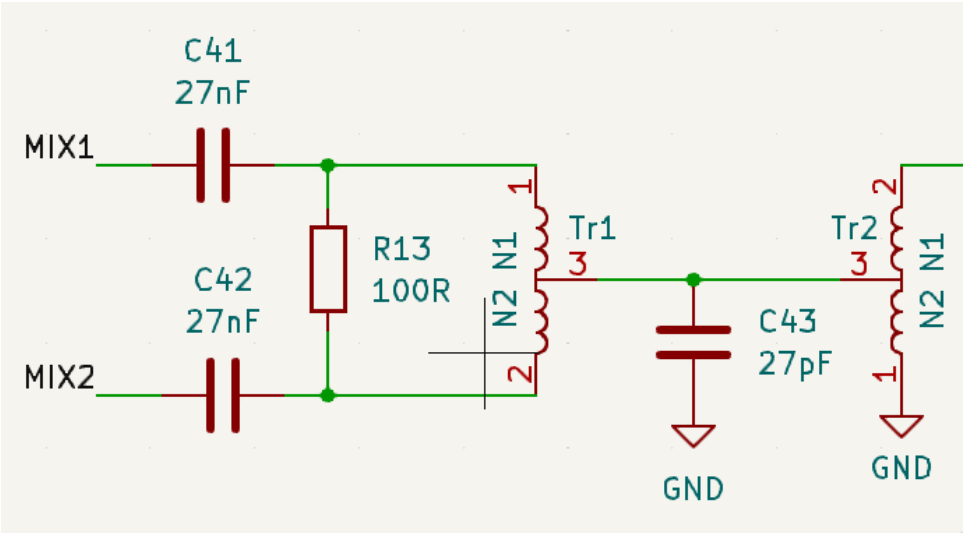
This filter was simply used to set the cutoff frequency of the input into the ADC such that its frequency did not exceed the Nyquist frequency of the ADC. The corner frequency of the low pass filter was controlled by a clock, which was the Si5351A clock generator chip

Amplification Circuit



This amplification circuit took the input from the coax, and amplified it in the form of AC ripples on a DC bias. C44 was used to remove the DC bias for further processing, while the resistor inductor circuit was used to set the output DC bias for amplification.

RF Splitter



This circuit is a common RF splitter circuit used in analog front ends. Tr2 is the input transformer which takes the input signal and makes its impedance 25 Ohm. Tr1 then is used to split the signal into two separate signals, which then goes into the DC decoupling capacitors. The 100 Ohm resistor is used such that signals from one mix cannot leak into the other mix. Each mix output is then connected into a 50 Ohm impedance matched circuit. Since there are two paths of 50 Ohm to ground, the impedance looking in is 25 Ohm, which necessitated the first transformer to lower the source impedance. C43 is used simply to improve performance at higher frequencies.

Digital Backend

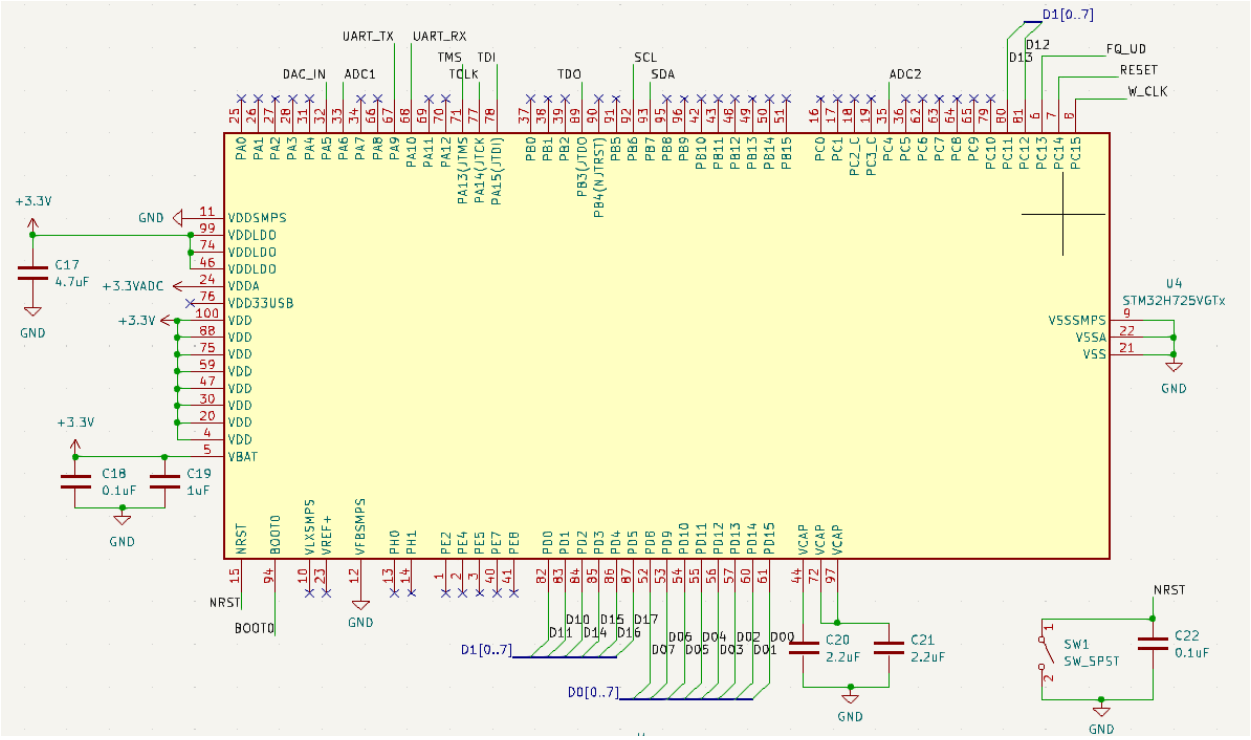
Theory of Operation

The operation is primarily meant to take in the IQ signals generated from the analog frontend, and perform filtering on them using FIR filters. One of the filters necessary would be the hilbert filter, which is used to phase shift every component frequency of a signal by 90 degrees. Which normally would be accomplished using complicated filter design, but in this case is accomplished using a digital FIR filter.

Hardware Implementation

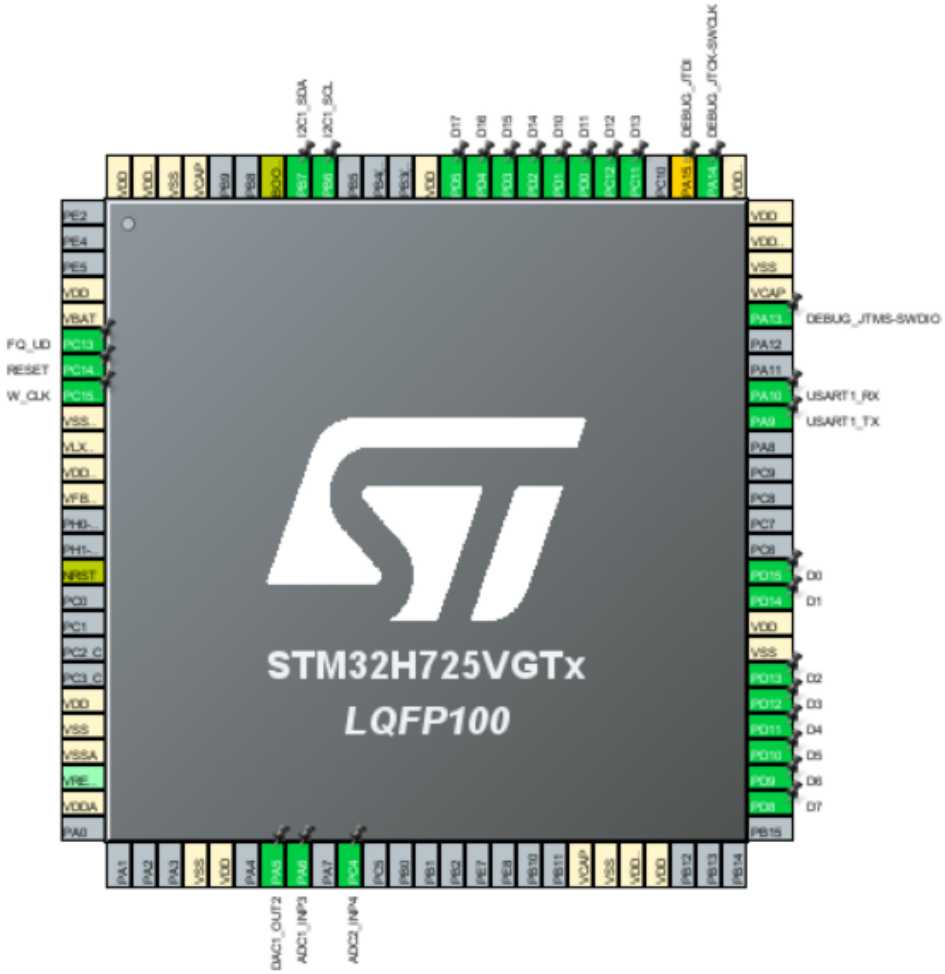
The digital signal processing was accomplished using a single high performance STM32 H725 part. Which has ADCs capable of sampling 4.5 MSPS, as well as the necessary processing power to run FIR filters and convolutions in real time for the signal processing applications. The STM32 was also responsible for controlling other components such as the clock generator and the DDS local oscillator.

This is the STM32 that is responsible for controlling many of the components on the board. Not shown are the power decoupling capacitors which are necessary for normal use.



Software Design Flow

We utilized STM32CubeMX to auto generate the extreme amounts of boilerplate code necessary to even begin programming the microcontroller.



Through CubeMX, we can set the appropriate pins and configure the microcontroller so that the code is generated with initialization of many of the necessary components. The code is generated with a Makefile toolchain, allowing us to build and flash code with a few simple commands.

From here, the auto generated code is opened in the Visual Studio Code IDE, so that we can utilize the “STM32 For VSCode” extension which packs the commands into convenient buttons for faster and easier design and debugging. Especially useful is its gdb ability, allowing us to step through code and debug with much finer granularity than otherwise would be allowed.

Microcontroller Interface Software

The microcontroller communicates with other hardware on the PCB through I2C, parallel GPIO, and ADC/DAC architecture.

I2C

I2C is used to control the SI5351A clock generator, which generates the 30 MHz local oscillator clock, used as the reference clock for the local oscillators, 10 kHz DAC and mixer low pass filter clock. To do so, we utilized the Skyworks ClockBuilder Pro software, which generated the necessary registers to write to with I2C. Then it was simply a matter of using STM32 I2C transmit functionality to write those registers.

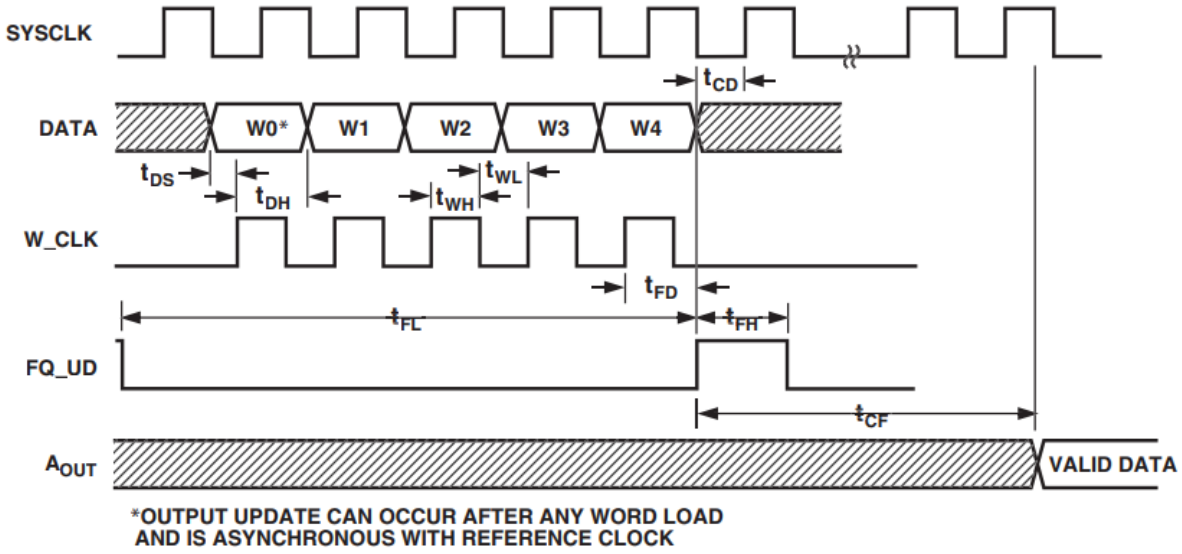
```
for (int i=0;i<SI5351A_REVB_REG_CONFIG_NUM_REGS;i++) {
    i2c_buf[0] = si5351a_revb_registers[i].address;
    i2c_buf[1] = si5351a_revb_registers[i].value;
    HAL_I2C_Master_Transmit(&hi2c1, 0x60 << 1, i2c_buf, 2, 1000);
    HAL_Delay(50);
  }
```

This code is executed during the configuration stage, and successfully writes the appropriate registers. The delay of 50 ms is definitely excessive, and should ideally be optimized through reception of an acknowledgement from the hardware, but since this is only run once at the beginning, it is negligible in a human time scale.

GPIO

16 of the microcontroller pins were dedicated to parallel GPIO output, 8 for configuring each of the AD9851 direct digital synthesizers (DDS), of which we utilize two. The AD9851 has its own

unique programming interface, which requires manual setting of I/O pins to configure.



To accomplish this, we wrote a custom function to correctly set the appropriate bits when necessary.

The frequency tuning word is calculated according to the function $f_{OUT} = (FTW * SYS\_CLK)/2^{32}$, where FTW is the 32 bit decimal value of the programmed tuning word, and SYS_CLK is 180 MHz, which is achieved by inputting a 30 MHz clock from the clock generator and setting the 6 x REFCLK bit high in the AD9851. With this equation, a desired 100 Hz output would require a FTW of 2386, which is the motivation for the seemingly opaque conversion factor seen in the code:

```
uint32_t tword = (freq/100) * 2386;
```

From this point, values of the 40-bit register are configured 8 at a time, using the AD9851 parallel programming mode. The code is quite malleable and allows for very easy configuration of any desired frequency, with phase offset between the two local oscillators hard coded to 90 degrees for IQ data manipulation. This phase offset can also easily be changed if desired with a little bit of extra code.

ADC/DAC

We utilize 2 ADCs and one DAC on the receiver architecture. The 2 ADCs sample simultaneously, but they sample data that has been IQ modulated, leading to IQ data entering the

microcontroller. In this time sensitive environment, the ADCs and DAC use DMA to quickly and efficiently write or read from memory buffers defined in the software. All three use a circular DMA buffer, meaning that rather than having to reset DMA pointers when the buffer is filled, the corresponding module will automatically begin reading or writing from the beginning of the buffer again. When the first and second half of this buffer are filled, an interrupt will be generated, allowing the software to manipulate only half of the buffer without accessing the other half, which is currently being written to or read from.

So what does the software actually do?

Currently, the code is written for singular ADC and singular DAC architecture, for the purpose of applying and testing FIR filters. It can easily be extended to differentiate between two different ADCs, but we ran into other issues that made that code unnecessary.

```
void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc) {
  //first half of adc buffer is full
  in_buf_ptr = &adc_val[0];
  out_buf_ptr = &dac_val[HALF_BUF_SIZE];// + HALF_BUF_SIZE;

  for (int i=0;i<HALF_BUF_SIZE;i++) {
    uint32_t test = fir(in_buf_ptr[i]);
    out_buf_ptr[i] = test;
  }

  flag=1;
}
```

The code above is executed via interrupt when half of an ADC buffer is full. We begin reading data from the first half of the ADC buffer, and prepare to output data to the second half of the DAC buffer via the in_buf_ptr and out_buf_ptr pointers. For every sample in the ADC buffer, we apply an FIR filter to that sample, then output the filtered input to the DAC buffer. The flag at the bottom is set for debugging purposes. The FIR filter parameters were generated through TFilter, an online tool for generating such parameters.

```
float firdata[FILTER_TAP_NUM];
int firptr[FILTER_TAP_NUM];
```

```
int fir_w_ptr = 0;

uint32_t fir(uint32_t in) {
  float in_f = (float)((int)in-2048);
  float fir_out = 0;
  for (int i=0;i<FILTER_TAP_NUM;i++) {
    fir_out += afilter_taps[firptr[i]] * firdata[i];
    firptr[i]++;
  }
  firdata[fir_w_ptr] = in_f;
  firptr[fir_w_ptr] = 0;
  fir_w_ptr++;
  if (fir_w_ptr == FILTER_TAP_NUM) fir_w_ptr = 0;

  return (uint32_t) (fir_out+2048);
}
```

FIR filtering is done through recognition that it is simply circular convolution of the input with the filter. The details of why are best left to a digital signal processing course, such as ECE 310. Said circular convolution is implemented above, with added buffers for memory of previously computed convolution outputs.

```
  float maxfir = 0;

  for (int i=0;i<FILTER_TAP_NUM;i++) {
    if (filter_taps[i] < 0) maxfir -= filter_taps[i];
    else maxfir += filter_taps[i];
  }

  for (int i=0;i<FILTER_TAP_NUM;i++) {
    afilter_taps[i] = filter_taps[i] / maxfir;
  }
```

Before the FIR filter is used, we scale it to prevent overflow of the DAC output data. The scaling is done with the worst case input in mind: if the input data just so happens to be its maximum negative value when multiplied by a negative filter coefficient and if it just so happens to be its maximum positive value when multiplied by a positive filter coefficient. Such a case is in

practice statistically impossible to observe, but is a good metric in order to completely prevent overflow. The result is a DAC output that generally observes a range of 0.3 to 3.0 V, as opposed to its maximum range of 0 to 3.3 V.

The current code demonstrates the simplest FIR filter, the all pass filter. Low pass, band pass, band stop, and high pass filters were additionally tested and work to an acceptable degree, although they limit the frequency at which we can sample at due to the increasing number of mathematical operations required to compute FIR filter response of these more complex filters.

Also included in the code is capability to apply the EMA filter, which is an incredibly bad low pass filter but is incredibly fast, featuring just two multiplications per sample. This was used early in development to ensure that the ADC and DAC DMA operations were working properly while having a function applied to the ADC input.

SDR Board Layout