

# Agile Radio Frequency Source for Hyperfine Manipulation of Ultra-Cold Atoms

A thesis submitted in partial fulfillment of the requirement for  
the degree of Bachelor of Science with Honors in Physics from  
The College of William and Mary

by

Justin Winkler

Accepted for: \_\_\_\_\_  
(Honors)

\_\_\_\_\_  
Adviser: Seth Aubin

\_\_\_\_\_  
Jan Chaloupka

\_\_\_\_\_  
Chi-Kwong Li

\_\_\_\_\_  
Gina Hoatson

Williamsburg, VA  
May 2009

## **Abstract**

This thesis describes the design, construction, and testing of a radio frequency signal source for generating rapidly changing frequencies in a repeatable and phase continuous manner. The radio frequency (RF) signal is generated by a Direct Digital Synthesizer (DDS) which is controlled by a triggerable microprocessor unit which is itself remotely programmed through an Ethernet connection. Waveforms generated by the RF source can be used for direct RF evaporation or RF manipulation of magnetic trapping potentials for ultra-cold atoms on an atom chip. The ultimate objective of the project is to mix the RF signal with higher frequency carrier waves to generate microwaves for the manipulation of hyperfine states of ultra-cold atoms of potassium and rubidium.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Atomic Structure . . . . .	3
1.2	Oscillating Magnetic Fields . . . . .	4
1.3	Theory . . . . .	5
1.4	Microwave Beamsplitting . . . . .	7
1.5	Evaporation . . . . .	7
1.6	Engineering Approach . . . . .	8
<b>2</b>	<b>Design</b>	<b>9</b>
2.1	Radio Frequency Source . . . . .	9
2.2	DDS Setup . . . . .	11
2.3	General DDS Operation . . . . .	16
2.4	Sweep Programming . . . . .	18
2.5	Microprocessor . . . . .	21
2.6	Optocouplers . . . . .	22
2.7	Register Board . . . . .	23
2.8	Microwave Mixing . . . . .	26
<b>3</b>	<b>Results</b>	<b>28</b>
3.1	Spectrum Analysis . . . . .	31
3.2	Line Width . . . . .	36
<b>4</b>	<b>Conclusion</b>	<b>41</b>
<b>5</b>	<b>Acknowledgements</b>	<b>42</b>

# 1 Introduction

My research project is the design and construction of a radio frequency source for manipulating the internal and external states of ultra cold atoms. The source is to be used to drive transitions between hyperfine levels and sublevels of  $^{39}\text{K}$ ,  $^{40}\text{K}$ ,  $^{41}\text{K}$  and  $^{87}\text{Rb}$ . These atoms would be positioned on an atom chip for an experiment, at close enough distances to current carrying wires that near-field RF and microwave magnetic fields could be used to manipulate the hyperfine states of these atoms, such as for RF and microwave evaporation, microwave beam splitting, and interferometry. Atomic interferometry allows for precision measurements of forces, while the use of an atom chip offers the prospect of miniaturization and enhancements of RF and microwaves in the near field.

## 1.1 Atomic Structure

For microwave beam splitting, a particular superposition of internal spin states is desired. One energy splitting of interest is the hyperfine splitting of the F-levels caused by the magnetic interaction between the spin of the nucleus and the spin of the valence electron. When placed in a static magnetic field, the sub-levels  $|F, m_F\rangle$  will experience a Zeeman shift, which is the other energy splitting of interest. Neglecting the magnetic moment of the nucleus, the Hamiltonian for the interaction of an atom with an external magnetic field is:

$$H = -\vec{\mu} \cdot \vec{B} \quad (1)$$

where  $\vec{\mu} = \frac{eg_e}{2m_e} \vec{S}$  is the magnetic moment of the electron, with  $e$  being the electron charge ( $e = 1.60218 \times 10^{-19}$  C),  $g_e$  is the gyromagnetic ratio of the electron ( $g_e = 2.00232$ ),  $m_e$  is the mass of an electron ( $m_e = 9.10938 \times 10^{-31}$  kg), and  $\vec{S}$  is the spin operator for the electron.

The relevant atomic structure of  $^{40}\text{K}$  is illustrated in Figure 1. Depending on the hyperfine level and sublevel an atom occupies, the atom will either seek high magnetic fields or low magnetic fields. When in a magnetic trap, the atomic state will be anti-trapped or trapped, respectively.

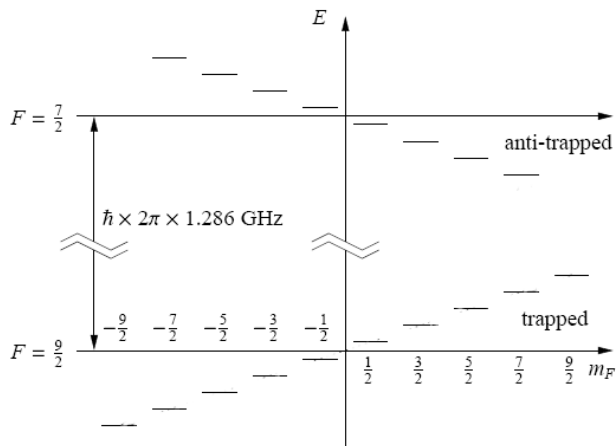


Figure 1: The atomic structure of  $^{40}\text{K}$  in a weak magnetic field. Whether or not an atom is trapped depends on the atom's spin configuration. The energy difference between hyperfine manifolds is  $\hbar \times 2\pi \times 1.286 \text{ GHz}$ . When driving transitions between two trapped spin states in the same hyperfine manifold, we want to use two different microwave magnetic fields. One transition would be from a trapped to an anti-trapped state, and another from the anti-trapped state to a different trapped state. This would effectively prevent atoms from becoming anti-trapped because they would immediately transition to a trapped state whenever they occupied the anti-trapped state. Figure adapted from [1].

## 1.2 Oscillating Magnetic Fields

An oscillating magnetic field can be used to drive transitions between states. Unfortunately, driving transitions between two specific  $m_F$  states has the complication that all  $m_F$  sublevels within a single  $F$  level are evenly spaced. Thus, using a magnetic field oscillating at radio frequencies to drive transitions between  $m_F$  sublevels will drive any  $\Delta m_F = \pm 1$  transition within a hyperfine level. As such, a specific transition, say from  $m_F = 9/2$  to  $m_F = 7/2$  within a single hyperfine manifold, cannot be made without some atoms also occupying other  $m_F$  sublevels.

To create a specific superposition of internal spin states, we need to drive transitions between hyperfine manifolds. The benefit of such transitions is that the energy spacing between Zeeman sublevels in different hyperfine levels is different for every possible transition. Thus, coupling only occurs for desired levels when transitioning between  $F$  levels. To drive these transitions, microwave frequencies are needed instead of radio frequencies, as

the hyperfine splitting is 462 MHz for  $^{39}\text{K}$ , 1286 MHz for  $^{40}\text{K}$ , 254 MHz for  $^{41}\text{K}$ , and 6835 MHz for  $^{87}\text{Rb}$ .

### 1.3 Theory

An atom in a magnetic field has a Hamiltonian described by equation 1. When the magnetic field is oscillating with linear polarization along the quantization axis (which we set as the z-axis), and with magnitude  $B_z$  and angular frequency  $\omega$ , then the Hamiltonian becomes

$$H = \frac{eg_e}{2m_e} B_z \cos(\omega t) \hat{S}_z \quad (2)$$

Perturbation theory yields the energy between spins states. It turns out that the first-order energy computed from perturbation theory is a very good approximation of the true energy in the low magnetic field limit. The transition amplitude  $T_{F,m_F \leftrightarrow F',m'_F}$  between two spin states  $|F, m_F\rangle$  and  $|F', m'_F\rangle$  is

$$T_{F,m_F \leftrightarrow F',m'_F} = \langle F, m_F | H | F', m'_F \rangle = \frac{eg_e}{2m_e} B_z \cos(\omega t) \langle F, m_F | \hat{S}_z | F', m'_F \rangle \quad (3)$$

The effect of  $\hat{S}_z$  on a state  $|F, m_f\rangle$  can be determined by expressing  $|F, m_f\rangle$  in terms of  $|I, m_I\rangle$  and  $|S, m_S\rangle$  basis states, which can be accomplished using Clebsch-Gordan coefficients.  $\hat{S}_z |S, m_S\rangle = \hbar m_S |S, m_S\rangle$ , so expressing  $|F, m_f\rangle$  in this way allows us to obtain an answer.

Taking  $F = F' = 2$ , we can calculate the transition amplitudes  $T_{m_F \leftrightarrow m'_F}$  from states  $m_F$  to  $m'_F$  within the  $F = 2$  hyperfine manifold. For the case of linear polarization, however, all transitions are found to be zero, with the only non-zero elements of  $\langle F, m_F | H | F', m'_F \rangle$  being found when  $m_F = m'_F$ . These do not refer to transitions, but rather to the energy differences between states that results from the static Zeeman effect i.e. the Zeeman energy level shifts just oscillate at the frequency of the oscillating magnetic field:

$$\langle F = 2, m_F | H_{linear} | F' = 2, m'_F \rangle = \begin{cases} E_{m_F=2} = \hbar \frac{eg_e}{2m_e} B_z \cos(\omega t) \\ E_{m_F=1} = \frac{\hbar}{2} \frac{eg_e}{2m_e} B_z \cos(\omega t) \\ E_{m_F=0} = 0 \\ E_{m_F=-1} = -\frac{\hbar}{2} \frac{eg_e}{2m_e} B_z \cos(\omega t) \\ E_{m_F=-2} = -\hbar \frac{eg_e}{2m_e} B_z \cos(\omega t) \end{cases} \quad (4)$$

If the magnetic field is circularly polarized, then  $\vec{B}_\pm = \frac{B_x \hat{x} \pm i B_y \hat{y}}{\sqrt{2}} \cos(\omega t)$ . Since this is a circularly polarized magnetic field,  $B_x = B_y$ . Let  $B_\pm \equiv B_x = B_y$ . Under these conditions, then, the transition amplitudes between two spin states  $|F, m_F\rangle$  and  $|F', m'_F\rangle$  is

$$\begin{aligned} T_{F=2, m_F \leftrightarrow F'=2, m'_F} &= \langle F, m_F | H | F', m'_F \rangle \\ &= \frac{1}{\sqrt{2}} \frac{eg_e}{2m_e} B_\pm \cos(\omega t) \langle F, m_F | \hat{S}_x \pm i \hat{S}_y | F', m'_F \rangle \\ &= \frac{1}{\sqrt{2}} \frac{eg_e}{2m_e} B_\pm \cos(\omega t) \langle F, m_F | \hat{S}_\pm | F', m'_F \rangle \end{aligned} \quad (5)$$

The transitions that are possible depend upon the direction of polarization. Finding these transition amplitudes for  $B_+$  and  $F = F' = 2$  using the same techniques as before yields the following non-zero transition amplitudes:

$$T_{F, m_F \leftrightarrow F', m'_F} = \begin{cases} T_{2 \rightarrow 1} = \frac{\hbar}{4} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \\ T_{1 \rightarrow 0} = \frac{\hbar \sqrt{6}}{8} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \\ T_{0 \rightarrow -1} = \frac{\hbar \sqrt{6}}{8} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \\ T_{-1 \rightarrow -2} = \frac{\hbar}{4} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \end{cases} \quad (6)$$

For the  $B_-$  case, the transition directions are reversed:

$$T_{F,m_F \leftrightarrow F',m'_F} = \begin{cases} T_{1 \rightarrow 2} = \frac{\hbar}{4} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \\ T_{0 \rightarrow 1} = \frac{\hbar\sqrt{6}}{8} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \\ T_{-1 \rightarrow 0} = \frac{\hbar\sqrt{6}}{8} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \\ T_{-2 \rightarrow -1} = \frac{\hbar}{4} \frac{eg_e}{2m_e} B_{x,y} \cos(\omega t) \end{cases} \quad (7)$$

These calculations refer only to radio frequency transitions within the  $F = 2$  hyperfine manifold. Calculating microwave transitions between hyperfine manifolds, however, is accomplished using the same techniques. From the transition amplitudes, we can easily determine the Rabi frequency that our oscillating magnetic field needs to be to drive transitions between these states. Furthermore, by application of Ampere's law, we can determine the necessary current through a wire in an atom chip needed to drive a transition for an atom some variable distance away. This is necessary in understanding the amount of power and amplification needed for the RF signal.

## 1.4 Microwave Beamsplitting

Once we have a microwave frequency source capable of creating various superpositions of internal spin states, a useful application is to then use this source for microwave beam-splitting. The source could be used to create a superposition of atoms with opposite spins. An intensity gradient of a far-detuned microwave magnetic field can then be used to drive these coherent atomic states apart spatially. If the process is driven in reverse after, then the scheme behaves as a Mach-Zender atom interferometer.

## 1.5 Evaporation

Another important application for the RF source includes usage for evaporative cooling of the atoms. Evaporative cooling is the final step to cooling atoms to quantum degeneracy. The spin state of an atom determines whether the atom will seek to be in higher or lower magnetic fields. Atoms in low-field seeking spin state will be held within a magnetic trap. The cooling process works by removing the higher energy atoms. The hotter atoms climb higher along the walls of the trap potential, to where their Zeeman energy shift is resonant



with a superimposed oscillating magnetic field. This causes only the higher energy atoms to spin flip to a high-field seeking state, removing them from the trap. The remaining atoms in the trap then rethermalize, and the process is repeated. [2]

## 1.6 Engineering Approach

The physics behind the various manipulation techniques imposes technical requirements when engineering the source. Rapid frequency sweeps are necessary when manipulating the internal and external state of the atoms, so the source needs to be sufficiently agile. Furthermore, changes in frequency need to be phase-continuous to prevent white noise that could disrupt the state of the atoms. It would also be beneficial if the signals are repeatable and exactly reproducible. Multiple phase-locked RF frequency sources will be needed in order drive more than one transition between hyperfine levels at a time and can also be used to generate circularly polarized oscillating magnetic fields.

One difficulty is that driving transitions between hyperfine manifolds requires a range span of about 100 MHz at moderate magnetic field strengths, but at microwave frequencies. It is difficult to engineer an agile microwave source with this operational range. To achieve these frequencies, both an agile RF source and a fixed microwave source need to be combined.

The rest of the thesis will concern the device design and the results of analyzing the output of the constructed device. Section 2 will explain the design of the RF source in broad terms, then discuss in detail the construction and utilization of the constituent components of the RF source, which includes a direct digital synthesizer, microprocessor, and additional circuitry. Section 2 will also discuss mixing the RF source output with a fixed microwave signal to produce the desired microwave signal. Section 3 considers the performance of the device by frequency analysis of the output signal.

## 2 Design

To perform frequency sweeps in the range required for microwave manipulation, it was necessary to create a radio frequency source that can operate within the needed frequency range. The output of this source can then be mixed with a constant microwave frequency source such that adjusting the frequency of the RF signal will adjust the frequency of the output signal from the mixing apparatus. The resulting signal can therefore be produced and altered within the required range of frequencies. The creation of a precise and agile variable frequency RF source is much simpler than the creation of a variable microwave frequency source, which is why this mixing setup will be employed. Alternatively, the RF frequency source can be multiplied up into the microwave range.

### 2.1 Radio Frequency Source

The desired radio frequencies are generated using direct digital synthesis (DDS). DDS devices can output analog sine wave signals with designated frequencies and many such products include the ability to have the signal ramp through a designated range of frequencies. The benefits of DDS include high precision and stability as well as phase-continuous frequency changes. Because DDS works by first creating signals digitally before being converted to an analog signal, the behavior of the DDS output is highly reproducible. Frequency sweeps can be set to occur over almost any given time frame, and the DDS can potentially be swiftly reprogrammed. Using a DDS to generate an RF signal is significantly cheaper than existing commercial alternatives, and possibly performs better. Some DDS devices, including ones in the lab, also allow the phase and amplitude of the signal to be manipulated.

The lab has five Analog Devices DDS devices: three part number AD9854 and two part number AD9910. In fact, over the summer as part of the REU program, I have already set up an AD9854 as a prototype radio frequency source. The AD9854 is programmed through a microprocessor controller, with sweep parameters sent to the microprocessor from a computer through an Ethernet connection. The microprocessor is set up to program sweeps to the DDS after receiving a trigger. The trigger will be sent from a sequencer that will be used to generate all the timing signals in the experiments. This design concept is

outlined in Figure 2.

Unfortunately, the AD9854 can only be clocked up to 300 MSPS (Mega Samples Per Second) and as such cannot output frequencies above the Nyquist frequency of 150 MHz. In fact, due to the digitization of the signal, using the AD9854 becomes unreliable for many applications for frequencies above a few tens of MHz. Because of its relatively small range of useful frequencies, the prototype is not particularly suitable for microwave manipulation, though it could still be used for other applications such as evaporative cooling or as a generic variable signal source.

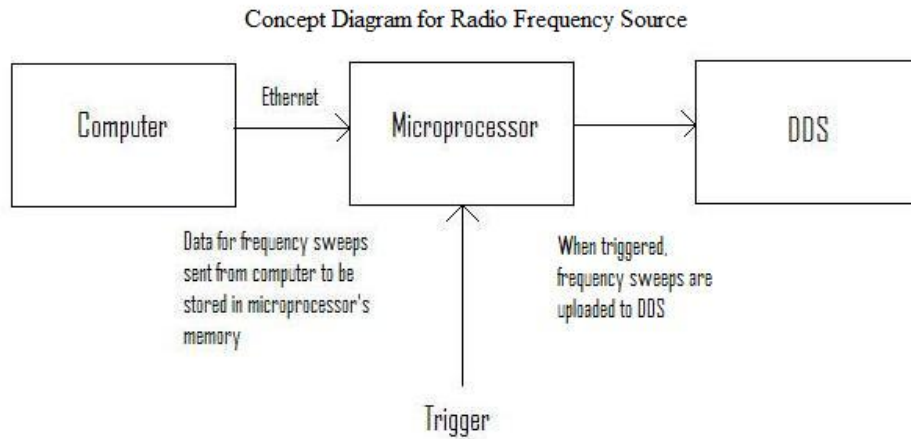


Figure 2: Concept diagram outlining design for RF source. A computer programs the sweep specifications into the microprocessor over an Ethernet connection. When triggered, the microprocessor programs the sweep into the DDS.

The newer AD9910 model is capable of being clocked up to 1 GSPS and as such has a more suitable output frequency range (from 0 Hz up to 400 MHz after filtering) for driving transitions between hyperfine levels when combined with a microwave carrier. Work this past year has as such consisted of setting up a radio frequency source employing the AD9910, but in a similar fashion to the AD9854 prototype. Again, users will send sweep data to the microprocessor over an Ethernet connection which will in turn be sent to the DDS based on an external TTL trigger.

The microprocessor used is a RCM 3200 RabbitCore module from Rabbit Semiconductors. The RabbitCore is programmed in a language called Dynamic C, which is similar to

C but has additional functions, including functions for digital I/O and Ethernet communication. The microprocessor programs the DDS using digital I/O lines.

One advantage of using the microprocessor is the Ethernet connection. Once multiple sources have been created, a user will be able to program them all from a single computer by connecting through the network. This is far more convenient than programming each source individually.

Some additional circuitry is used to connect the microprocessor and the DDS. For example, all digital lines have been optically isolated from the DDS to prevent ground loop noise causing uncontrolled effects on the experimental sample. Furthermore, construction has begun of circuitry consisting of registers and counters that will store the instructions sent from the microprocessor to help prevent timing jitter caused by frequency and phase mismatch of the RabbitCore clock with the trigger and the DDS sampling clock. The register circuitry could also be used to speed up the time it takes to program the DDS after a trigger is received: unaided, the Rabbit takes just over 2 ms to program the DDS, but this same programming process would take about 15  $\mu$ s if a 10 MHz signal was used to clock the serial data transfer.

## 2.2 DDS Setup

The lab's AD9910 was already mounted on an evaluation board by Analog Devices, so relatively little setup was required. Nonetheless, the board still required a power source and a reference clock. Also, additional modification of the evaluation board proved necessary to use the DDS's phase-locked loop (PLL) multiplier and the evaluation board's filters.

Powering the DDS requires both 1.8 V and 3.3 V power sources. Analog and digital components have separate power lines and grounds to suppress noise on the analog lines. Additionally, the evaluation board has optional USB interface circuitry that requires an extra 3.3 V power line and ground. As such, the evaluation board has 5 separate power lines and 3 ground lines: 1.8 V digital power, 3.3 V digital power, 1.8 V analog power, 3.3 V analog power, 3.3 V USB power, and separate grounds for the digital, analog, and USB circuitry. Thus, it was necessary to construct regulator circuits to supply all of these power lines, with each power line having its own devoted regulator. Figure 3 shows a diagram of

the regulator circuits.

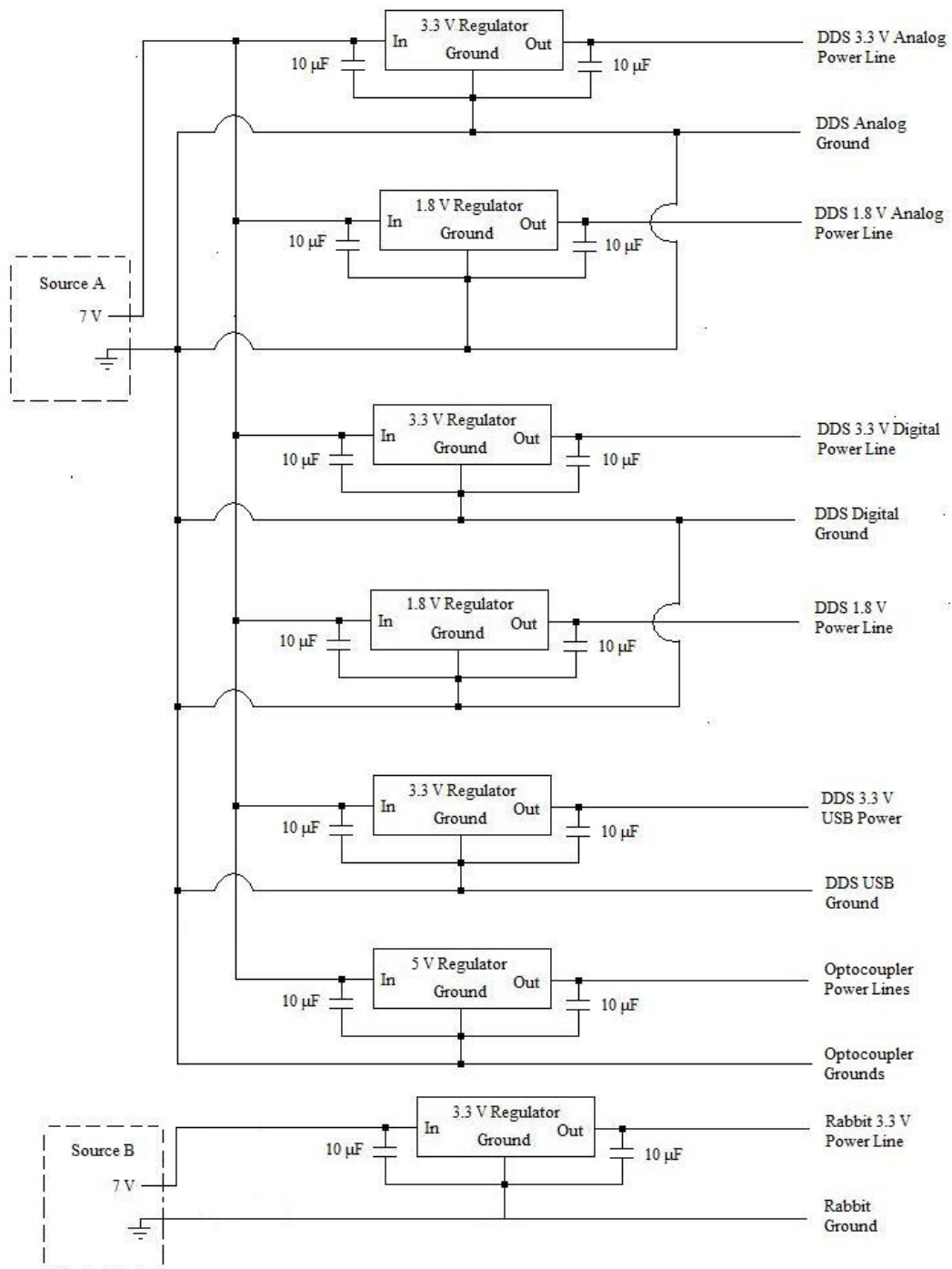


Figure 3: A diagram of all of the regulator circuits. The regulators are powered using 7 V DC sources. Two separate sources are used (marked A and B in the diagram) so that the Rabbit microprocessor and the DDS have separate grounds.

The reference clock for the DDS can be supplied using a 25 MHz crystal oscillator attached to the evaluation board or can be supplied from an external source (depending on the setting of jumper W7 of the evaluation board). Additionally, an external reference clock can be multiplied by the DDS to obtain a higher system clock frequency. Any externally provided reference clock signal should have a peak-to-peak amplitude between 50 and 1000 mV and it was found that the DDS did not operate reliably at system clock frequencies below 50 MHz.

For early testing, we used a system clock of 50 MHz provided directly from a DE2 FPGA development kit by Altera Corporation. For practical usage, the DDS will use a 1 GHz system clock frequency, provided via a stable 10 MHz clock signal multiplied by a factor of 100 using an onboard phase-lock loop (PLL) multiplier. Later testing used the latter setup.

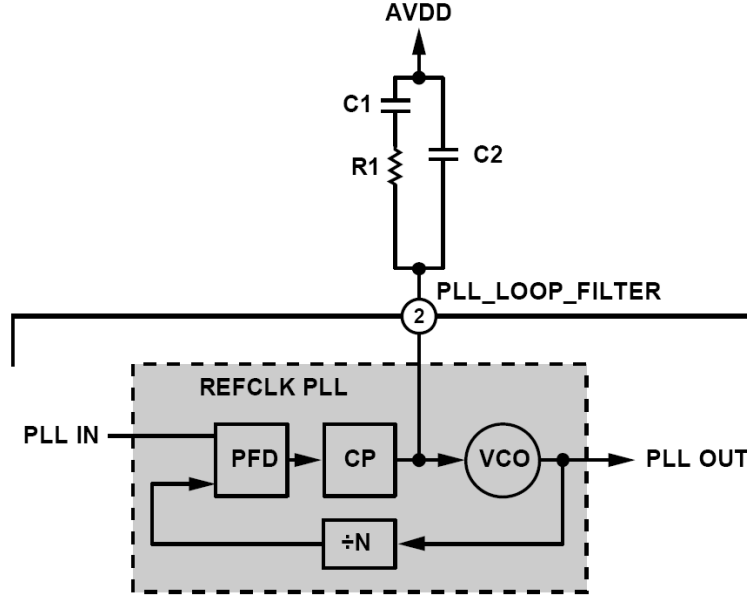


Figure 4: Although the AD9910 included a phase-lock loop multiplier, an external loop filter is required to make use of the PLL, connected to the AD9910 through the PLL\_LOOP\_FILTER pin. The values of R1, C1, and C2 are determined by the desired open-loop bandwidth and phase margin of the PLL, as well as the multiplication factor N, the gain of the VCO, and the programmed value of the charge pump current. Figure adapted from [3].

Use of the DDS's PLL multiplier requires an external loop filter, the design of which is shown in Figure 4. The capacitance and resistance values in this loop are determined by the

desired open-loop bandwidth and phase margin of the PLL, with equations for these values provided in the DDS manual. Although there is a location on the evaluation board for these capacitors and resistor, these parts were not included by Analog Devices to give the end-user more flexibility. As such, the board needed to be modified by soldering the required surface-mount parts to the board. With a multiplication factor of 100, capacitances of  $C1 = 2.2 \text{ nF}$  and  $C2 = 330 \text{ pF}$  and a resistance of  $R1 = 680 \text{ } \Omega$  were used, which approximately corresponds to an open-loop bandwidth of 300 MHz and a phase margin of  $50^\circ$ .



#### 400 MHz Low Pass Filter

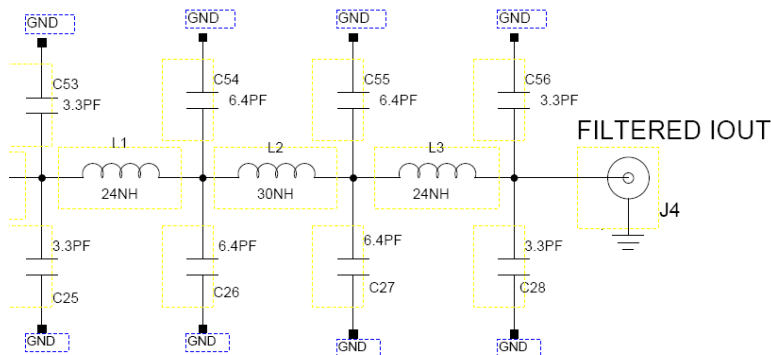


Figure 5: The AD9910 evaluation board includes a 400 MHz low pass filter, the design of which is shown in the diagram above. A user of the DDS can obtain the output signal from either the filtered output jumper (shown above), or can obtain the unfiltered signal from a different jumper on the board (not shown in the above diagram). Figure adapted from [4].

A 400 MHz 4<sup>th</sup> order LC low pass filter is included on the evaluation board. A sample of the evaluation board schematic is shown in Figure 5 to show the design of the low pass filter the board employs. This circuitry was initially not properly connected on the evaluation board and a single wire connection also had to be soldered to the board to make use of the filter.

### 2.3 General DDS Operation

The DDS can be operated using 20 and 40 pin headers mounted on the evaluation board. The pins needed to make the DDS perform sweeps are listed in Table 1, along with their functions. Of particular importance are the SDIO and SCLK pins, which are used for serial communication with the DDS. While the AD9910 does have a parallel I/O feature, this cannot be used with the digital ramp generator and is as such not useful for this project. Thus, the DDS is programmed using serial I/O.

Serial I/O consists of setting a logic value to SDIO and then toggling SCLK to store the logic value of SDIO in the DDS. Multiple bits and bytes can be stored in the DDS over multiple ticks on SCLK. As a default, the DDS expects to receive the most significant bit of a byte first, although this can be reprogrammed. Programming the DDS consists

of storing binary values in the various internal registers of the chip. The values in these registers set various control or output parameters, such as the output signal's frequency or the multiplication factor of the PLL.

The first step in programming a register is to serially send that register's address to the DDS. The address consists of a single byte, and the hexadecimal values of a particular register's address can be looked up in the AD9910 manual. After an address is sent to the DDS, the DDS then expects to receive a particular number of bytes to be stored in the selected register. The number of bytes the DDS expects depends on the register, and this number can again be looked up in the AD9910. Thus, a communication cycle with the DDS consists of sending a register address to the DDS, then sending the DDS the values you want to store in that register. After all expected bytes are received by the DDS, the communication cycle starts again and the DDS expects to next receive an address byte. An example of a communication cycle is shown in Figure 6.

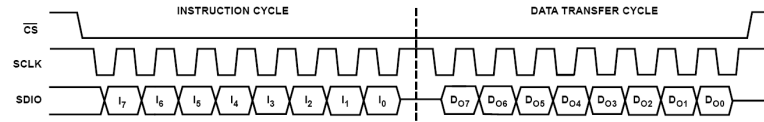


Figure 6: An example of a DDS communication cycle. During serial communication, on a rising edge of SCLK, the DDS stores the value of SDIO into I/O buffers. The first part of a communication cycle, the instruction cycle, consists of sending a byte containing a register address. The second part, the data transfer cycle, consists of sending bits to be stored in the register selected during the instruction cycle. Once the number of bytes expected for the chosen register is sent to the DDS, the DDS resets to the instruction cycle and waits to receive another address byte. The chip select pin (CS or CSB) must be set to logic low for any serial communication to occur. Figure adapted from [3].

A few other pins are used to program the AD9910's registers. The chip select pin CSB must always be set to logic low for any serial communication to occur. In case a data transfer fails, the communication cycle can be aborted and reset by toggling the I/O reset pin. It's also worth noting that using SDIO and SCLK will only store values in I/O buffers. Toggling the I/O update pin or changing the selected profile with the profile select pins (P\_0, P\_1, and P\_2) will transfer the contents of the I/O buffers into the internal registers,

in effect finalizing the programming process.

Pin Name	Description
SDIO	The logic value on this pin is used for serial I/O. A rising edge on SCLK transfers the logic value of this pin to the I/O buffers.
SCLK	This pin provides the serial I/O clock. A rising edge on this pin stores the logic value of SDIO in the I/O buffers.
IO_UPDATE	A logic high transfers the contents of the I/O buffers to the internal registers.
IO_RESET	Restarts the serial I/O communication cycle in case of failure.
CSB	The chip select pin. A logic low allows the DDS to detect serial input, whereas a logic high causes the DDS to ignore serial input.
DRHOLD	A logic high causes the digital ramp generator to stall in its present state until logic low is asserted.
DR_CTL	Controls the slope polarity of the digital ramp generator. A logic low will cause a downwards sweep, and a logic high causes an upward sweep.
OSK	A logic high enables output shift keying, which can sweep or switch the output amplitude between zero and the programmed amplitude scale.
RESET	The master reset pin. A logic high resets all memory elements and sets registers to default values.
EXT_PWR_DWN	A logic high initiates the programmed power down mode.
P_0, P_1, P_2	The profile select pins. The logic values of these three pins select one of the eight phase/frequency profiles. Furthermore, a change of logic values on these pins transfers the contents of the I/O buffers to the internal registers, like a logic high on IO_UPDATE.

Table 1: A list of useful pins and their descriptions. This table is reproduced from reference [3].

## 2.4 Sweep Programming

Having covered the general operation of the DDS, it is time to discuss how to actually use the DDS to produce frequency sweeps. Before getting into the specifics, though, when powered up, the DDS needs to have a few settings altered. The ramp limit registers (address 0x0B) do not have default values and appear to retain their previously programmed values even when the DDS is powered down. As such, the ramp limit registers should first be zeroed out to prevent the old limits being used when the DDS is switched to digital ramp

mode.

It may also be preferable to set the digital ramp rate registers (address 0x0D) at this point. The digital ramp rate registers control how often the digital ramp generator's accumulator is updated, where updates to the accumulator alter the output frequency of the DDS by a set amount. Smaller ramp rates decreasing the time between updates. Selecting the smallest possible ramp rates will maximize the time resolution of the sweep. The exact time between accumulator updates is given by the following formula from the manual:

$$\Delta t = \frac{4P}{f_{sysclk}} \quad (8)$$

where P is the value stored in the register and  $f_{sysclk}$  is the frequency of the reference clock. Additionally, the control function registers (addresses 0x00, 0x01, and 0x02) need to be changed from their default values. In particular, the bits of these control registers should be set so that digital ramp mode is enabled and the input divider is bypassed. If using the PLL multiplier, then it needs to be enabled, and the appropriate VCO needs to be selected (VCO5 for multiplication up to 1 GHz), and the multiplying factor N needs to be set. All of the parameters relevant to the PLL are controlled by bits in the third control function register.

There are three signal parameters that the AD9910 can ramp: frequency, phase, and amplitude. Which parameter is being ramped is controlled by the digital ramp destination bits in control function register 2. By default, frequency is ramped (although ramping amplitude could be a useful future function). Testing has always involved only frequency ramps, but programming amplitude sweeps is just as easy as programming frequency sweeps. The only major difference is that the ramp limits when using amplitude are less precise than with frequency and so extra dummy bits need to be sent to the registers when performing amplitude sweeps.

The parameter selected by the digital destination bits is controlled by the digital ramp generator when digital ramp mode is enabled. The other two parameters would be controlled by the selected single tone profile. Therefore, when performing a frequency sweep, a profile should be selected and programmed so that it gives the desired phase and amplitude scale

to the output signal. Again, this profile should ideally be programmed when the DDS is powered up (I typically select and program single tone profile 0, address 0x0E). Note that the amplitude scale has a 14-bit resolution and can be found using the equation

$$\text{Amplitude Scale} = \frac{\text{ASF}}{2^{14}} \quad (9)$$

which gives the fraction of the full scale amplitude that will be output, and where ASF refers to the 14-bit programmable amplitude scale factor. The phase adjustment can be found using

$$\Delta\theta = 2\pi \frac{\text{POW}}{2^{16}} \quad (10)$$

which gives the phase adjustment in radians as a function of the 16-bit programmable phase offset word (POW).

Once these initial settings have been assigned, the DDS is ready to perform a sweep. Only two registers now need to be set to do this. First, the digital ramp step size register (address 0x0D) needs to be set. The digital ramp step size register controls the amount the output frequency is increased each time the accumulator updates. In the direction of the sweep, the value of the ramp size is based upon the amount of time the end-user wants the sweep to take. In the direction opposite the sweep, the value of the ramp size is set to maximum, which should cause the digital ramp to snap to the ramp limit in a single tick of the DDS clock and prevent ramping through undesired frequencies in the wrong direction. The size of a frequency step based on the contents of the digital ramp step size register is determined by the following formula:

$$\text{Frequency Step} = \frac{M}{2^{32}} \times f_{sysclk} \quad (11)$$

Where M is the value stored in the register and  $f_{sysclk}$  is the frequency of the DDS system clock.

After the step size is sent to the DDS, the digital ramp limit register (address 0x0B) needs to be set. This register sets the upper and lower bounds of a ramp. Thus, this

register is used to determine the start and finish of a sweep. When frequency is the digital destination, the values programmed into this register should be the frequency tuning words (FTW) of the limit frequencies. An FTW is determined using the following equation:

$$\text{Output Frequency} = \frac{\text{FTW}}{2^{32}} \times f_{sysclk} \quad (12)$$

An FTW of  $2^{31}$  is effectively the maximum FTW. If the FTW is set above  $2^{31}$ , the DDS will not produce a coherent signal.

After the step size and limit registers have been programmed, the IO\_UPDATE pin should be toggled so that these changes take effect. The DRCTL pin should then be set to create a sweep from one limit to the other, in the appropriate direction.

## 2.5 Microprocessor

The algorithm to program the DDS is described above. This algorithm is performed by the RCM 3200 Rabbitcore microprocessor to program a single sweep when it receives a trigger. The program run by the Rabbit microprocessor for sweep generation was adapted from an older program used for the same purpose with the AD9854. The functions concerned with communicating with the DDS had to be revised significantly from the older version, whereas the algorithms that managed Ethernet communication and data storage were relatively unaltered.

I will now outline the sweep generation program run by the microprocessor. I should note that I have only written programs that perform frequency sweeps or generate constant frequencies as well as various small test programs. It should, however, be easy to expand these programs to perform more tasks. Since it is of critical importance to my project, I will limit my programming discussion to the frequency sweep program.

The binary values that set up a sweep on the DDS are to be determined by a computer side program and then sent to the microprocessor over an Ethernet connection. These sweep parameters are queued up in a static array that is emptied as triggers are received. Note that the number of sweeps that can be queued is limited, as such. The microprocessor program was written so that the Rabbit could only listen for the trigger or the Ethernet

connection and not both at once. As such, currently a user needs to program desired sweeps into the Rabbit, tell the Rabbit to listen for sweeps, and then wait until all sweeps queued in the Rabbit have been programmed into the DDS.

Data sent to the Rabbit over the Ethernet connection needs to follow a specific format. A byte sent to the Rabbit is interpreted as a command which, depending on the byte's value, tells the Rabbit to receive and store data for a sweep, to clear all current sweeps from its storage array, to forcibly restart the DDS and restore hard-coded default settings, or to switch to listening for the trigger. More commands can be written as necessary though these four have proven sufficient.

When the Rabbit receives a command to store another sweep, the Rabbit will attempt to receive another 13 bytes over the Ethernet connection. These bytes need to be sent in a specific order, with the first byte indicating sweep direction (a zero for a sweep from low to high, otherwise a sweep from high to low), the next four need to be the FTW for the higher frequency limit, the next four are the FTW of the lower frequency, and the next four are the digital ramp step size.

It is important that any computer-side program that communicates with the Rabbit perform all tuning word calculations prior to sending the data to the Rabbit. The primary reason for this is that the Rabbit's precision is relatively limited. A simple GUI for this purpose has been written. The GUI takes the frequency sweep limits and the time duration of the sweep and sends the necessary binary values to the Rabbit. This GUI, made using LabWindows, is as simple a program as possible and was meant to be used primarily for testing. Further expanding the functions of the GUI and the Rabbit program should be straight forward for a competent programmer. At present, the code does allow the Rabbit to program the DDS to generate a variety of sweeps, as required.

## **2.6 Optocouplers**

While I have discussed the programming of the Rabbit microprocessor, I have not yet discussed any circuitry. Like the DDS, the Rabbit needs a power source. Specifically, the Rabbit requires a 3.3 V power source, which is provided through a corresponding regulator (again, refer to Figure 3 for the circuit diagram).

Other circuitry beyond the power regulators usually try to mitigate problems associated with using the Rabbit. One such problem is the possibility of ground loop noise, as mentioned previously. Atoms on an atom chip are very sensitive to any electrical noise such as in a ground loop, so isolating the ground of the DDS is vital. Since there is not a convenient way to optically isolate Ethernet lines, the Rabbit has to be directly connected to a PC with which it shares a ground. The voltage of the PC is not something we can control, so the DDS must be optically isolated from the microprocessor. Digital I/O with the microprocessor occurs through a variety of pins on the microprocessor, but each pin used on the microprocessor is connected to an optocoupler which is in turn connected to the pins on the AD9910. The optocouplers work by converting the digital signal from the microprocessor into a light pulse that is received by a photodiode and converted back to a digital signal. As such, the microprocessor and DDS are not directly connected and do not share grounds or a common power supply, preventing the possibility of ground-loop noise. Figure 7 shows the circuit layout of a single optocoupler. The only other element that might introduce ground loop noise is the reference clock. For flexibility, the reference clock is not hard wired into the rest of the device but is rather supplied externally. If the reference clock is powered by the same power source as the DDS, then it should not be a source of ground loop noise. If, however, the reference clock requires a different power source, then additional optocoupling may be necessary. While optocouplers are easily fast enough to transmit data under the Rabbit's clock, the optocouplers can only transmit a signal up to about 10 MHz. As such, if the external clock is optocoupled, then the performance of the optocouplers will likely set an upper limit on the usable clock speeds.

## 2.7 Register Board

Another problem associated with the Rabbit microprocessor is its relative slowness. The time required to program the DDS by the Rabbit is on the order of a couple of milliseconds, and as such there is a relatively large programming delay between when the Rabbit is triggered and when the DDS performs a sweep. Furthermore, there is significant trigger jitter when triggering the Rabbit that makes operating the DDS signal timing less repeatable. Ideally the source would be programmed after the trigger with minimal delay and jitter.



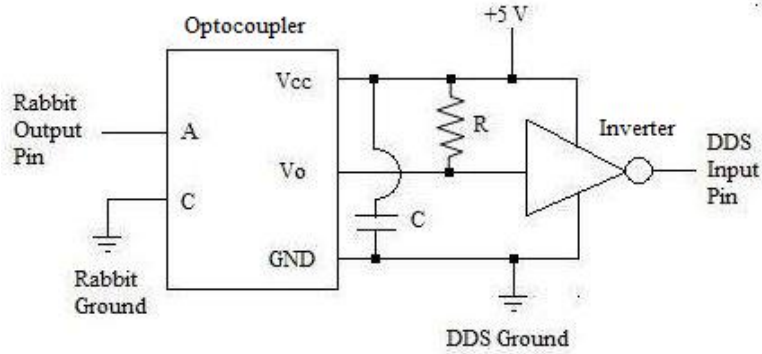


Figure 7: A diagram of the circuit layout of an optocoupler connecting a Rabbit output pin to a DDS input pin. The optocoupler accepts an input signal from the Rabbit microprocessor through the optocoupler’s anode, A, and transmits this as a light pulse to a photodiode, and the signal is then output from  $V_O$ , inverted. The optocoupler’s cathode, C, needs to be connected to the same ground as the Rabbit. The optocoupler uses a 5 V voltage supply, with a pull-up resistor of resistance  $R = 200\ \Omega$  connecting the output  $V_O$  of the optocoupler to the voltage supply. There is also a bypass capacitor of capacitance  $C = 0.1\ \mu\text{F}$  attached between the supply voltage pin and ground pin of the optocoupler. The output signal is put through an inverter before being sent to the DDS so that programming of the Rabbit need not be done using inverted logic.

The reason for the delay and jitter is that microprocessors usually require multiple clock cycles to perform an instruction. Thus, bits are sent to the DDS at a rate significantly below the microprocessor’s clock speed, while extra time is needed for the Rabbit to process that a sweep has occurred. A simple digital circuit, on the other hand, can be made to send a bit in a single clock tick, and can respond almost immediately to triggering.

To address the jitter and programming delay problems, construction has begun of a “register board” that holds sweep data in various registers. The data in the registers can be clocked in by the Rabbit prior to a trigger. When a trigger is received, the contents of the registers are clocked into the DDS using the fast and reliable DDS clock, rather than the relatively slow and unreliable Rabbit. The currently planned setup is displayed in Figure 8. Unfortunately, the register board has not been fully built, the plans are included here as they will still prove useful if someone else needs to finish the construction.

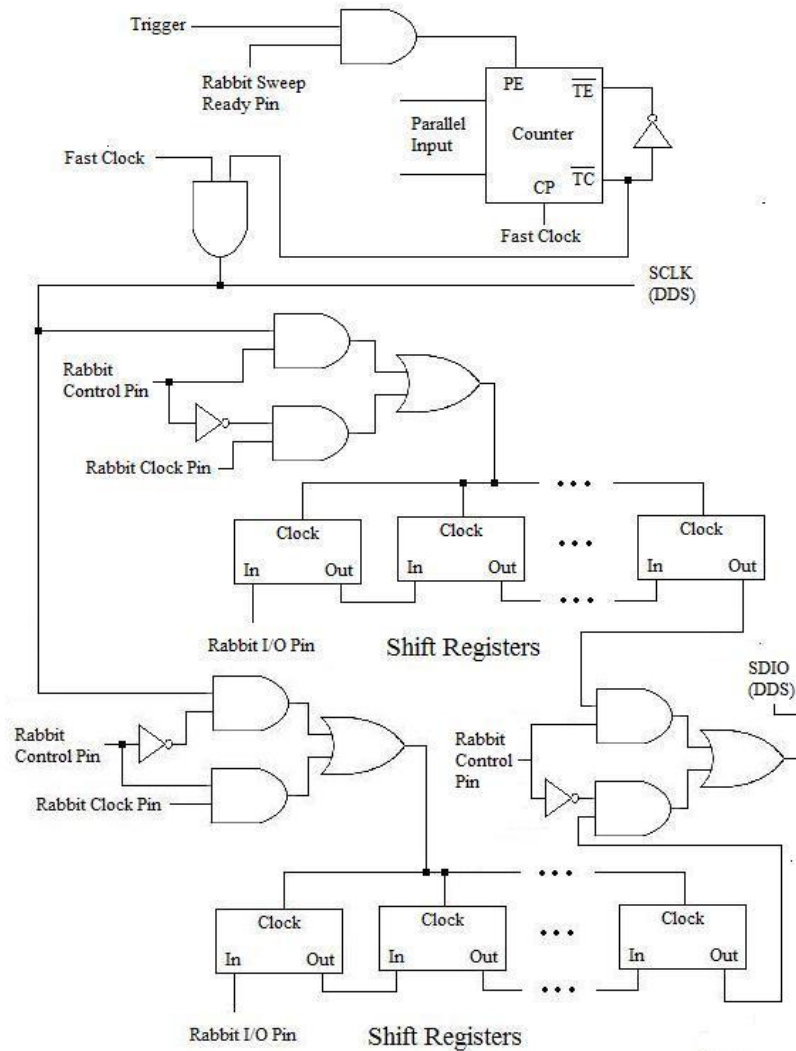


Figure 8: A diagram of the planned register circuitry. The Rabbit will use a control pin with digital logic circuitry to switch between clocking sweep data into one set of shift registers or another, while the other register can be clocked into the DDS using a “fast clock,” which will likely be a 10 MHz oscillator. The fast clock is only used when a trigger is sent to the board, resetting the counter to a value hardwired onto the parallel inputs (probably 144 since programming a DDS sweep requires 144 bits). The counter will then allow the fast clock to clock in a sweep and then cut off the fast clock’s connection to the SCLK pin and the register boards after the hardwired number of clock ticks have passed. The Rabbit can detect when a sweep has been programmed into the DDS and can switch which register board the fast clock is clocking by using the control pin once the Rabbit has finished programming the next sweep into a register set. Note that this design may change as various components are constructed and tested.

Note that minor changes to the design in Figure 8 may prove necessary. In the current plan, the counter cuts off the fast clock's connection to the SCLK pin when the counter hits zero. Instead, it may be necessary that a 5 (for example) or lower on the counter will cause control to be cut off, with the last few ticks before the counter hits zero being used to manipulate the I/O update pin and the DRCTL pins on the DDS. This would be done by using logic gates to determine the behavior of these pins based on the counter output. It's not clear whether or not this change will be needed, but it's a relatively minor difference in design from the plan displayed in Figure 8. As the register board is still being constructed, the final design may vary in minor ways from the initial plans, especially as individual components of the circuitry are tested.

To make use of the register board, the Rabbit microprocessor will need to be reprogrammed, though the changes should not be too significant. A few more optocouplers may be needed for extra connections from the Rabbit. It's currently planned to set up the Rabbit so that a user can easily switch between using the register board or not using the register board, where the benefit in not using the register board would be that programming would be slightly easier and older programs could still be used.

## 2.8 Microwave Mixing

As mentioned previously, this radio frequency source will in turn be mixed with a stable microwave frequency source. The microwave source will output a constant frequency near 460 MHz, 1.3 GHz, or 6.8 GHz depending on the atom being used (although we may just frequency double to get 460 MHz). Conventional mixing works by multiplying the two input signals together. A single sideband modulator mixer can mix signals while suppressing unwanted harmonics that occur from the signal multiplication. This would leave the mixed output having a primary frequency component that is the sum of the RF and microwave frequencies. The resulting mixed signal will be filtered further of undesired frequency components, amplified, and then sent to the atom chip where the atoms are trapped. Figure 9 illustrates the design of the microwave mixing apparatus using a block diagram.

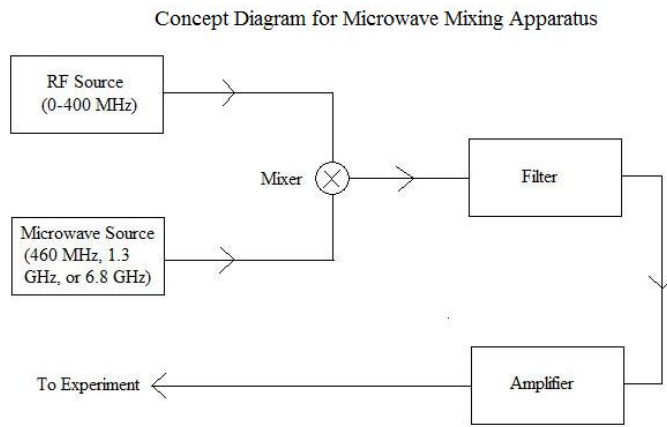


Figure 9: Concept Diagram Outlining Design for Microwave Mixing Apparatus. The output signal of the RF source is mixed with a fixed microwave frequency source. Potential fixed microwave frequencies include 460 MHz, 1.3 GHz, and 6.8 GHz. After mixing, the signal is filtered and amplified, then sent to the atom chip.

### 3 Results

Over these past two semesters, I have built the circuitry that connects the RabbitCore to the AD9910, such as the circuitry that optically isolates the digital lines and the power regulators. I have started the construction of register circuitry to improve the device's response to triggering. Indeed, apart from the register circuitry and microwave mixing, all circuitry associated with the DDS has been completed, debugged, and tested. Figure 10 shows the finished circuits.

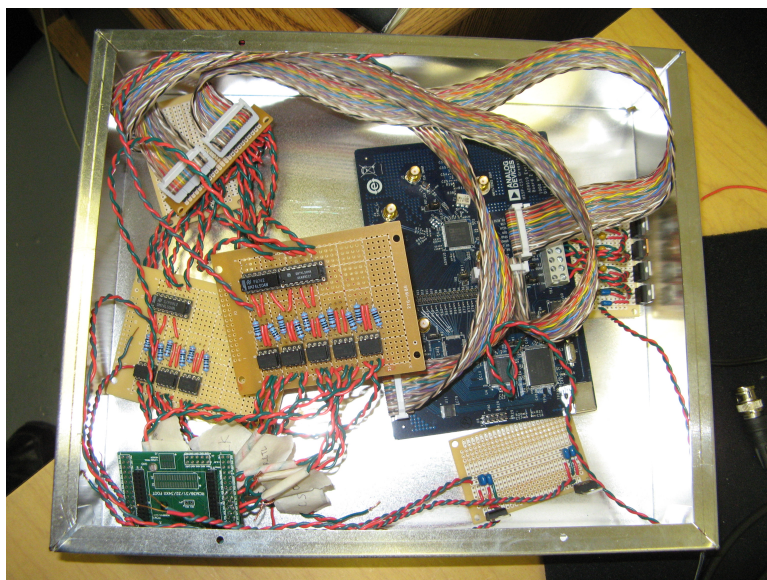


Figure 10: A picture of circuitry that has been built, excluding the unfinished register circuits.

I have also reprogrammed the sweep generation programs to work with the AD9910. The final program has no known problems associated with Ethernet communication or sweep storage. Testing suggests that the program can generate consecutive sweeps in any direction from any reasonable limits with no problems yet noticed for this version.

It is important to characterize the output signal generated by the DDS. It is not easy to analyze the quality of the sweeps. At best one can check with an oscilloscope that sweeps are performed over the right range of frequencies and over the right amount of time. An example of a sweep is shown in Figure 11. A static frequency, however, can be easily analyzed by making use of the fast Fourier transform function available on oscilloscopes. Of

particular interest are the higher output frequencies since the digitization effects are more pronounced.

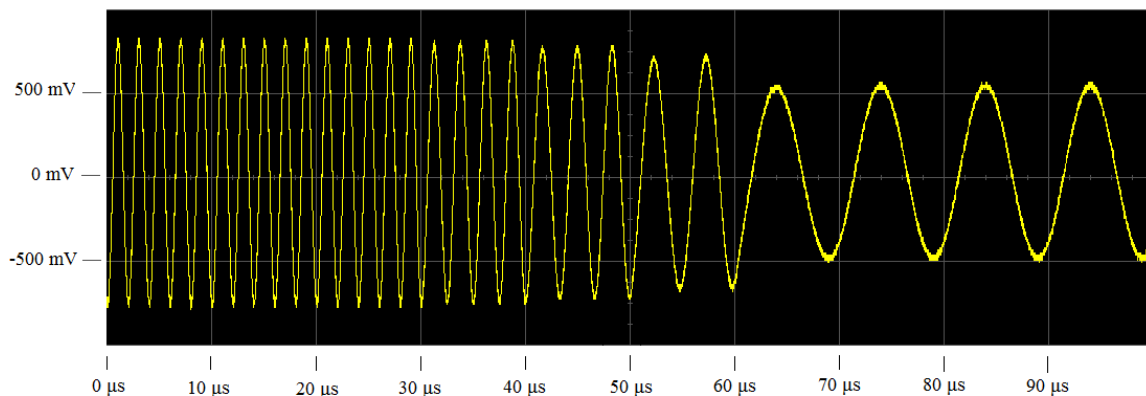


Figure 11: Oscilloscope picture of a sweep from 500 kHz to 100 kHz. To illustrate the sweep process, the step frequency was set at 100 MHz and the time step between accumulator updates was set to  $10\ \mu\text{s}$ . Thus, the signal starts a ramp at 500 kHz, jumps down to 400 kHz, and after  $10\ \mu\text{s}$  jumps down to 300 kHz, and so forth until the output is at 100 kHz. Usually, individual frequency steps cannot be discerned because the time step between accumulator updates is minimal. This figure is meant to illustrate how the DDS digitally produces sweeps. In practice, changes in frequency will be much smaller than 100 kHz, probably on the order of a few Hertz, making the sweep far closer to continuous.

Figure 12 shows an example of a highly digitized signal. We want to know how high the output frequency can go while still being pure enough to be useful. Also, while the evaluation board includes filtering circuitry, we want to evaluate whether more filtering is necessary.

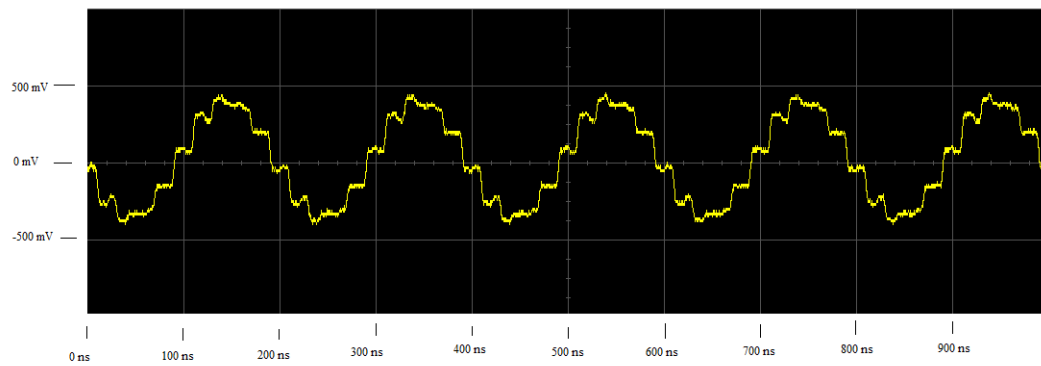


Figure 12: Oscilloscope picture of a noticeably digitized signal. The closer the frequency of the output signal is to the reference frequency of the DDS, the more pronounced the digitization of the output signal is, revealing the piecewise construction of the sine waves. This picture was taken using a 50 MHz reference frequency with an output frequency of 5 MHz, which is why a period consists of 10 voltage steps. This particular output signal was taken from the unfiltered output of the evaluation board and so does not characterize the filter performance. In practice, output signals from the DDS will not have this pronounced digitization because the reference clock will be faster and filtering will be used.

### 3.1 Spectrum Analysis

I used a spectrum analyzer (Agilent model E4405B) to characterize the output signals with frequencies of 10 MHz, 30 MHz, 100 MHz, 200 MHz, 300 MHz and 400 MHz. The data from the spectrum analyzer are displayed in Figures 13, 14, 15, 16, 17, and 18. While it seems that there are some additional frequency components to these output signals, these extra components are relatively small and hopefully will not require additional filtering. Few extra harmonics exceed -60 dBm. The most noticeable exception to this is that the 400 MHz signal has a -40 dBm peak at 600 MHz, suggesting that the filtering is less effective when the output is increased up to 400 MHz. This is not too surprising, as the filter is a 400 MHz lowpass filter. Overall, the filter performs fairly well at frequencies significantly below 400 MHz.

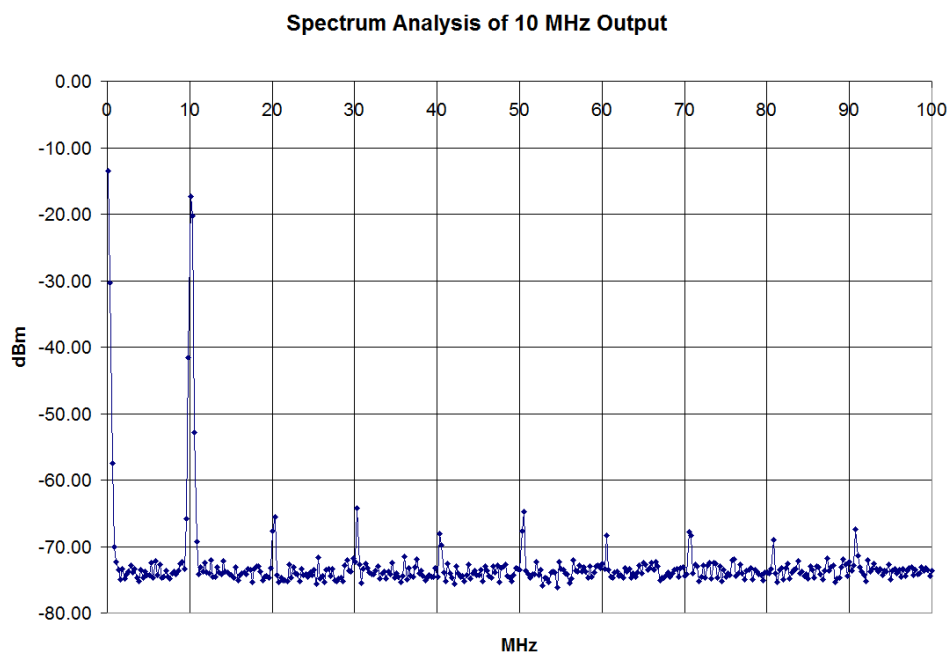


Figure 13: Spectrum analysis of 10 MHz output signal. Extra harmonics are observed at 10 MHz intervals, but are heavily suppressed. Peaks at zero occur because an infinite amount of time would be needed to exclude possible low frequency components of the signal. The zero peaks seen in this figure and others are not, as such, indicative of any DC artifact.



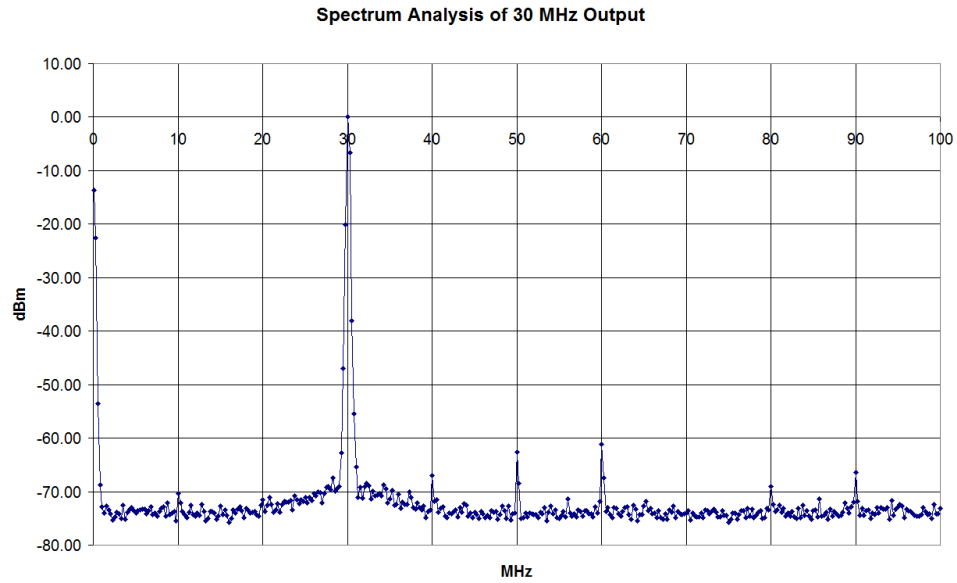


Figure 14: Spectrum analysis of 30 MHz output signal. Like with the 10 MHz signal, there are extra harmonics found with the 30 MHz at frequencies that are multiples of 10 MHz. Again, these never exceed -50 dBm.

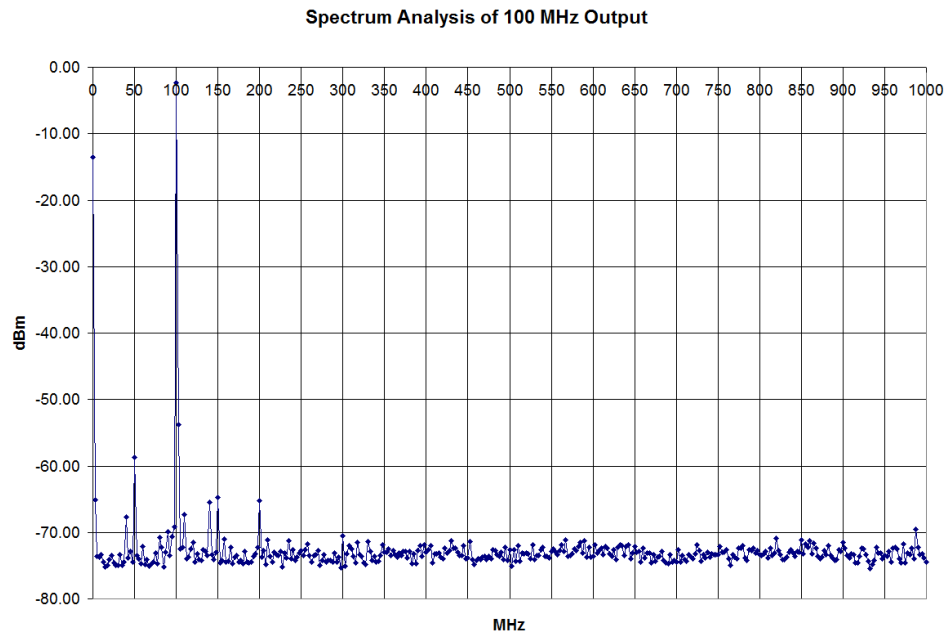


Figure 15: Spectrum analysis of 100 MHz output signal. While all noise is still heavily suppressed, the signal seems to have more noise in the 0 to 200 MHz range, with noticeable sidebands at 50, 150, and 200 MHz.

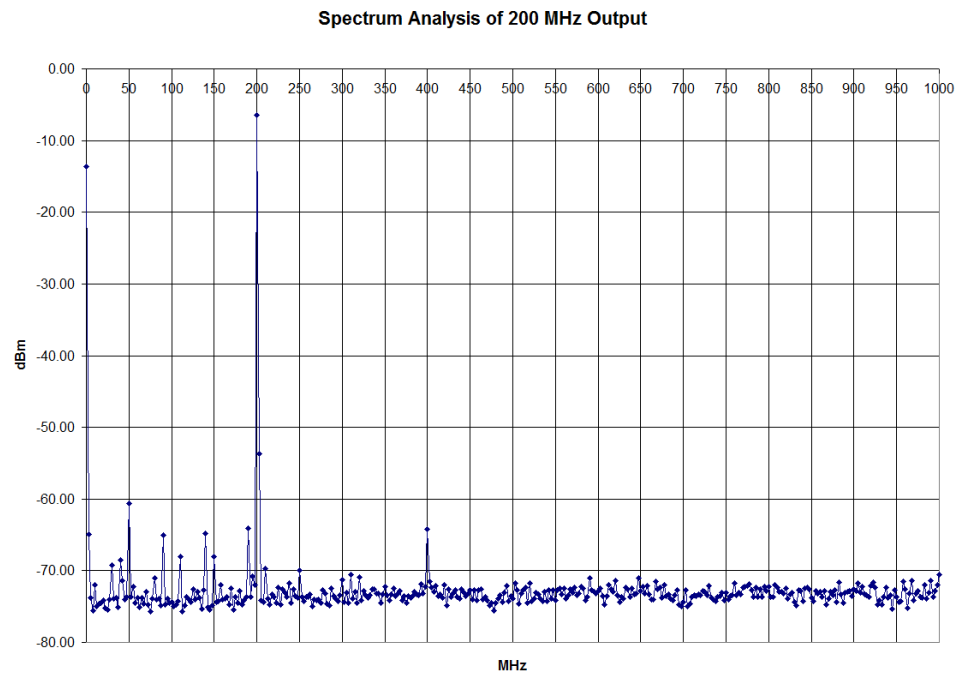


Figure 16: Spectrum analysis of 200 MHz output signal. While not exceptional, there is noise in the 0 to 200 MHz range and to a lesser extent in the 200 MHz to 400 MHz range. The most notable sidebands are at 100 MHz and 400 MHz.

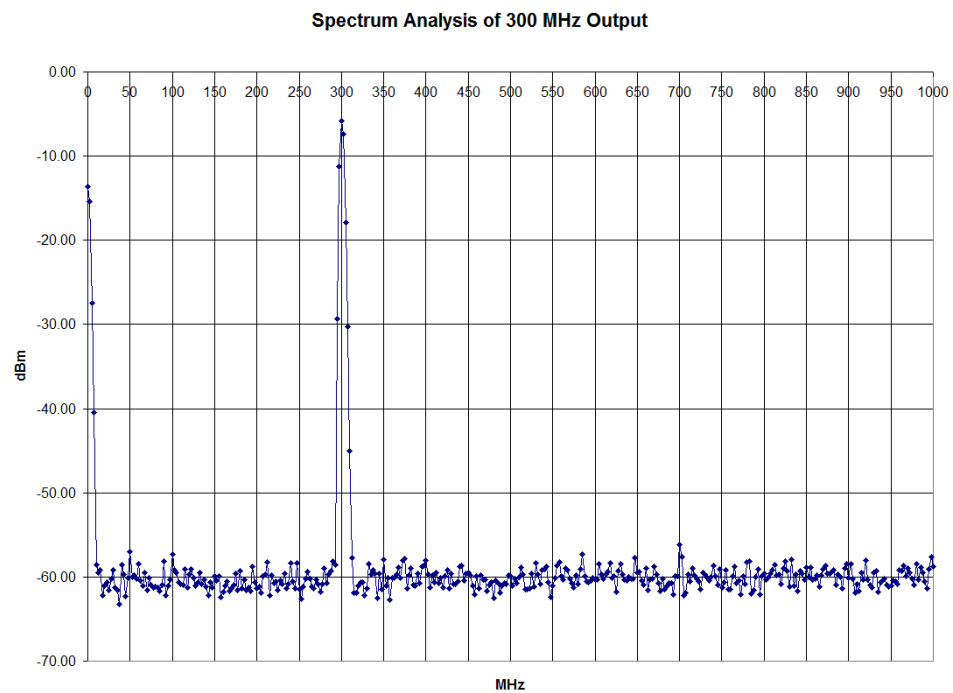


Figure 17: Spectrum analysis of 300 MHz output signal. There do not appear to be any exceptional examples of noise , with noise appearing level at -60 dBm.

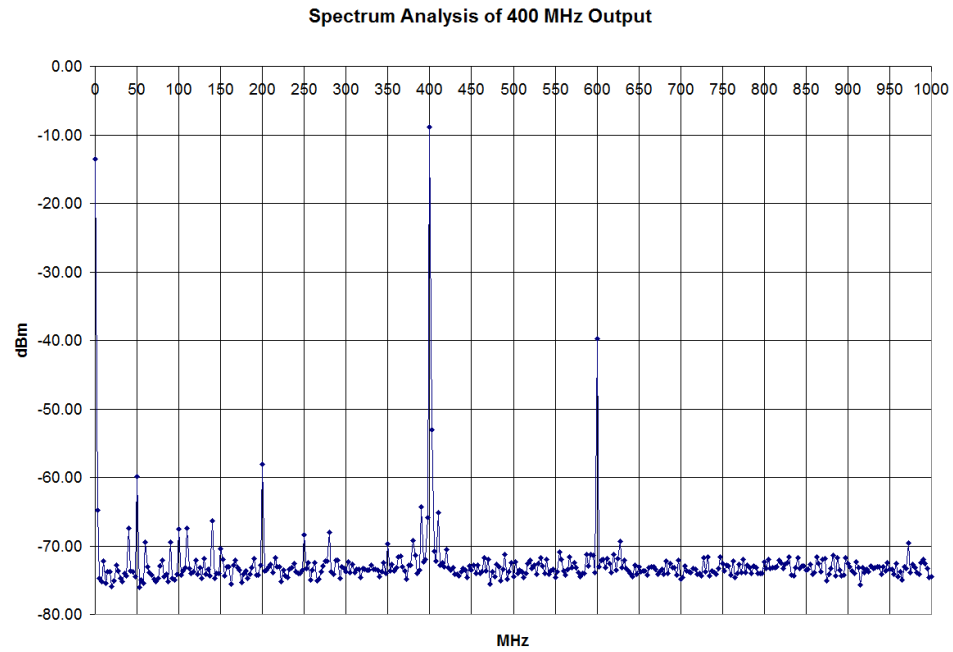


Figure 18: Spectrum analysis of 400 MHz output signal. This is the noisiest signal that was analyzed, with noticeable background noise around the 400 MHz peak. There is also noise scattered in the 0 to 400 MHz range, with the notable sideband at 200 MHz. The signal also has a 600 MHz component that is the least suppressed extra harmonic of all the signals analyzed.

### 3.2 Line Width

Data was also taken concerning the line width of a peak. The spectrum analyzer was used to take data while systematically zooming in on the main peak of both a 10 MHz signal and 300 MHz signal. The results are displayed in Figures 19, 20, 21, 22, 23, 24, 25, 26, and 27. These plots suggest that the line width is thin for both low and high frequency outputs.

Analysis of the 10 MHz signal is meant to characterize the line width at lower frequencies. Background noise is noticeable but relatively low. The background noise observed in the 9.95 to 10.05 MHz range has about 70 dBm of suppression. At the 9.99 to 10.01 MHz range, the background noise is much more significant, though with still about 60 dBm of suppression.

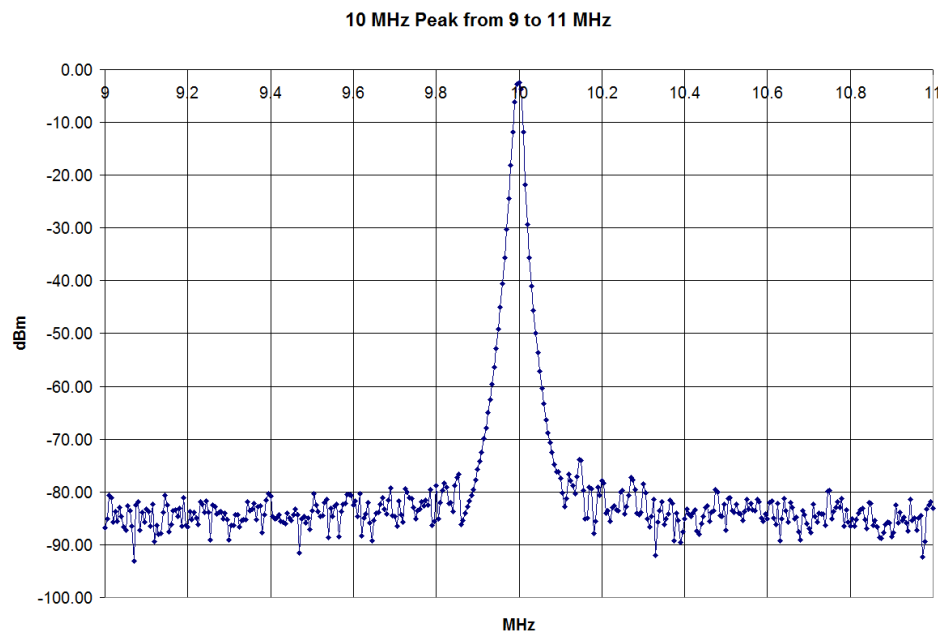


Figure 19: 10 MHz peak viewed from 9 MHz to 11 MHz. There's not background noise that's noticeable at this range. The background noise that is observable ranges from about 9.8 to 10.2 MHz and is suppressed by about -75 dBm.

Analysis of the 300 MHz signal is meant to characterize the line width at higher frequencies. Background noise is more prevalent at this higher frequency, but it is still suppressed by approximately 50 dBm or more.

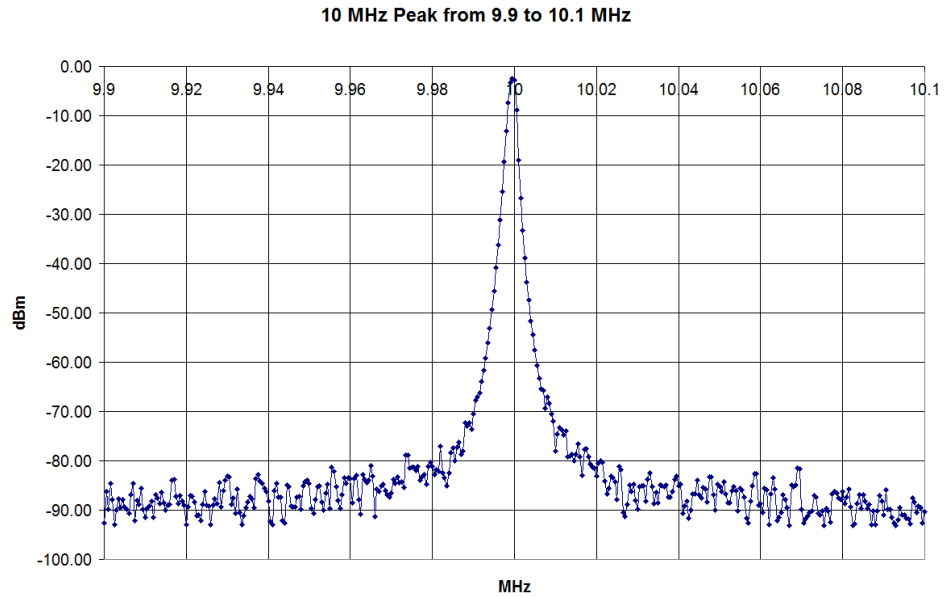


Figure 20: 10 MHz peak viewed from 9.9 MHz to 10.1 MHz. Background noise is slightly more pronounced, particularly in the range from 9.96 to 10.04 MHz, though still sees at least 70 dBm of suppression.

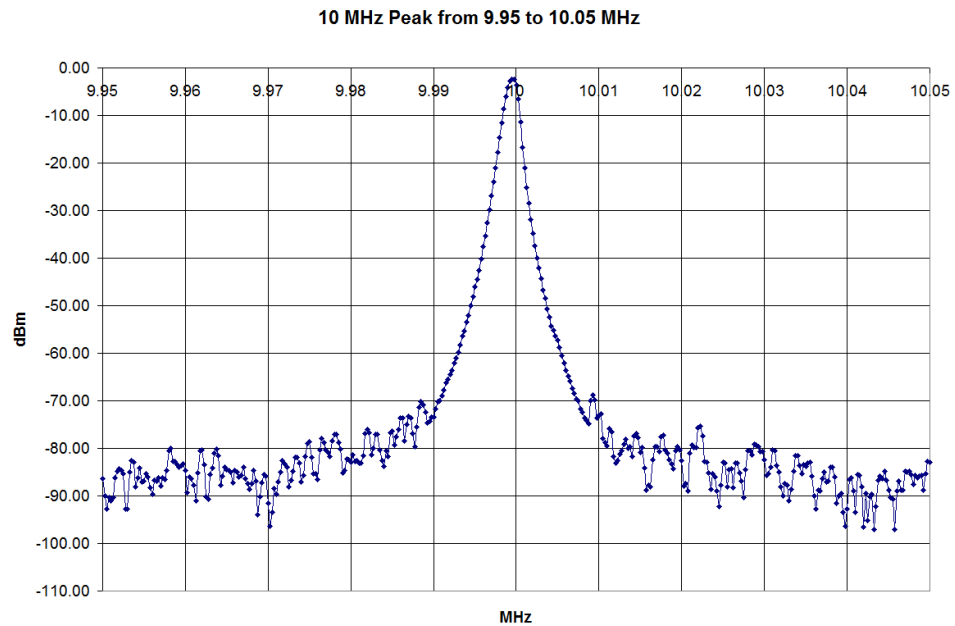


Figure 21: 10 MHz peak viewed from 9.95 MHz to 10.05 MHz. While still seeing suppression of 70 dBm or more, the effect of background noise is clearly visible in the 9.97 to 10.03 range.

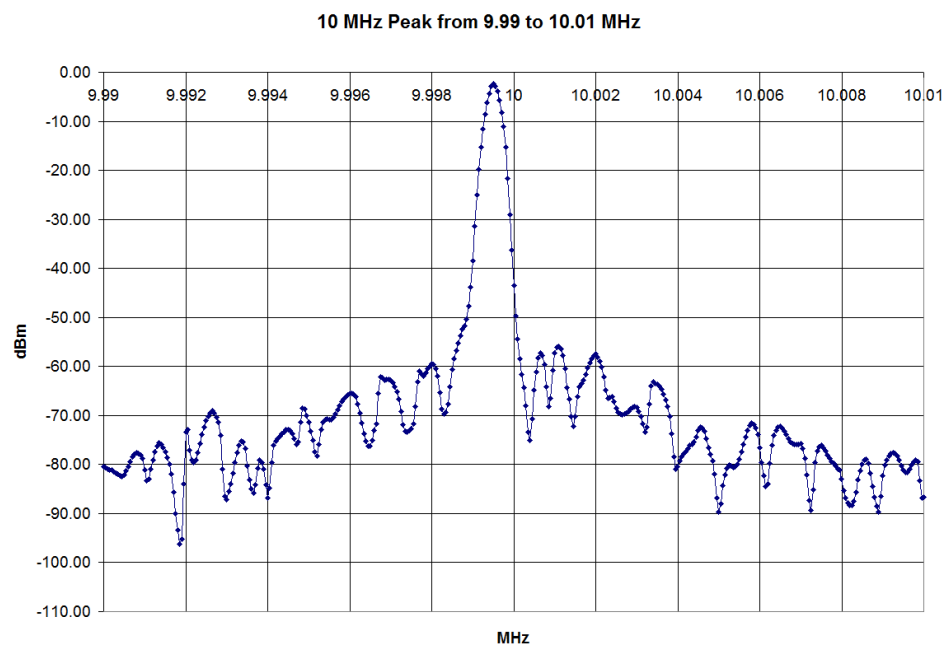


Figure 22: 10 MHz peak viewed from 9.99 MHz to 10.01 MHz. At this relatively close range, background noise is only seen to be suppressed by about 60 dBm.

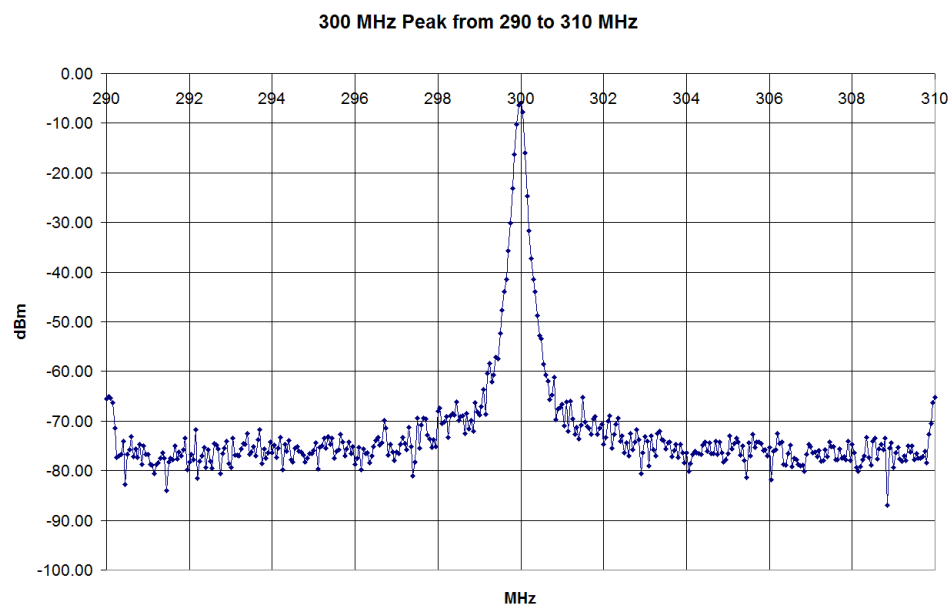


Figure 23: 300 MHz peak viewed from 290 MHz to 310 MHz. There appears to be a little bit of extra noise from 290 to 300 MHz, though not much above -65 dBm.

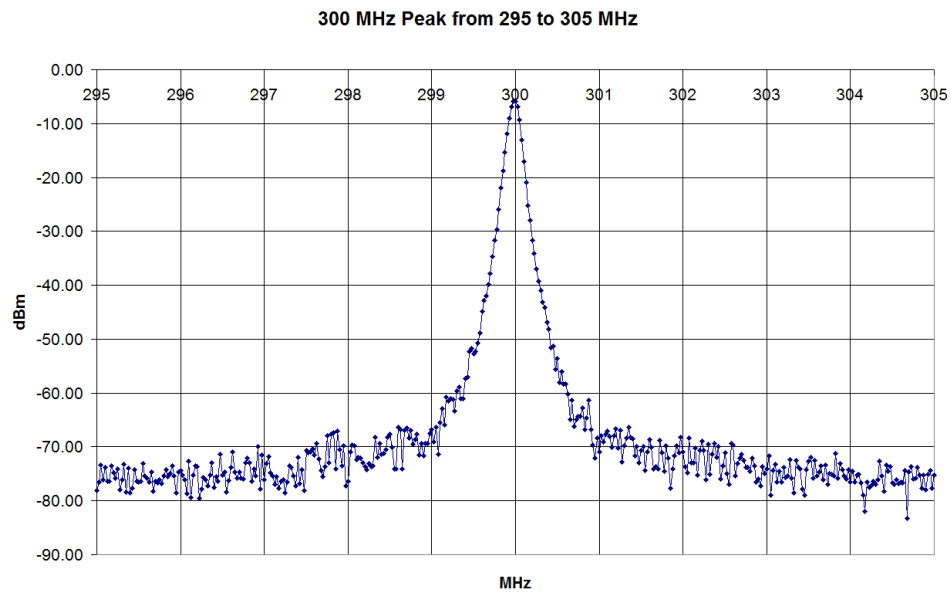


Figure 24: 300 MHz peak viewed from 295 MHz to 305 MHz. At this range, there seems to be some background noise from about 298 to 302 MHz, though it's not very significant.

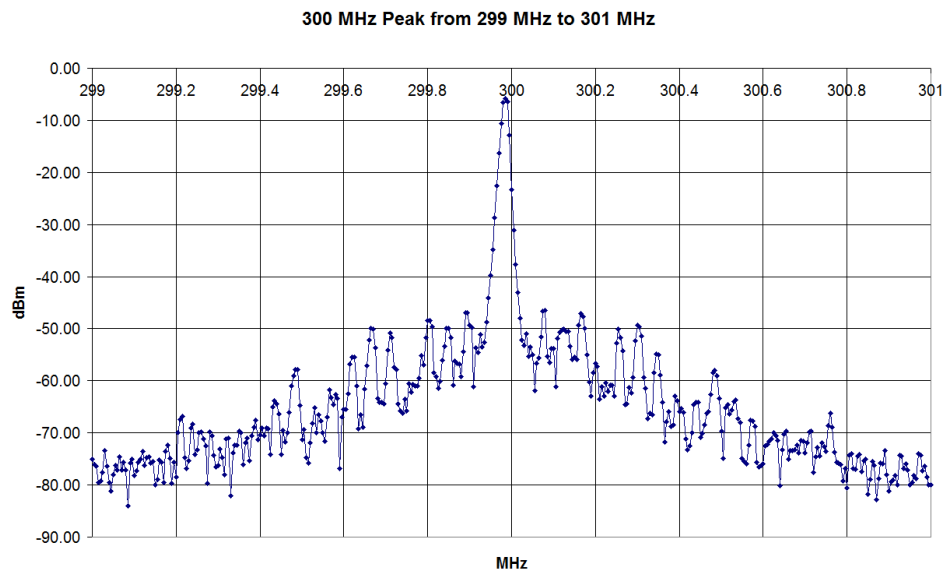


Figure 25: 300 MHz peak viewed from 299 MHz to 301 MHz. Unlike with the previous peak plots, there is a very noticeable background noise at 299.5 to 300.5, suppressed by only about 50 dBm.



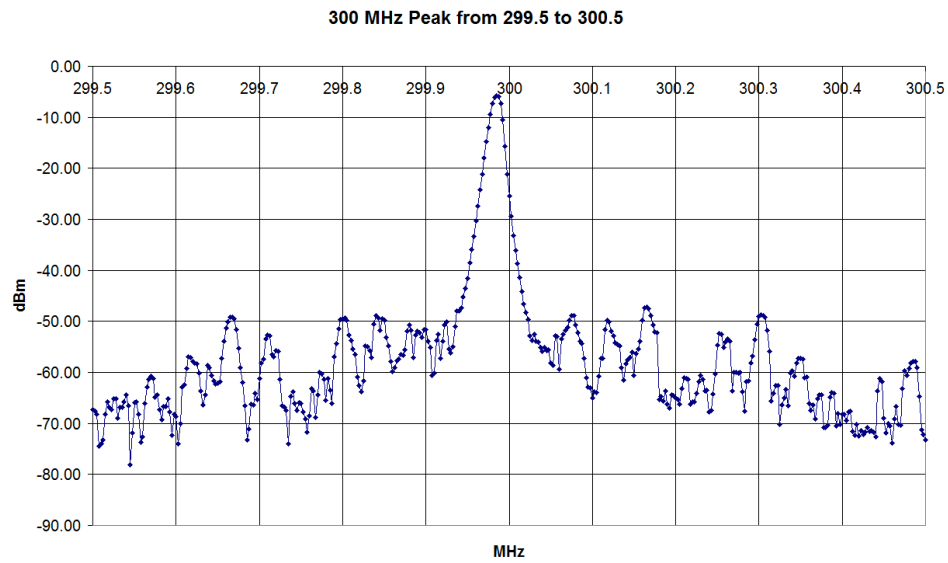


Figure 26: 300 MHz peak viewed from 299.5 MHz to 300.5 MHz. Again, the background noise is at about -50 dBm except at the very edges of the plot range, and the noise is present throughout the entire range of the plot.

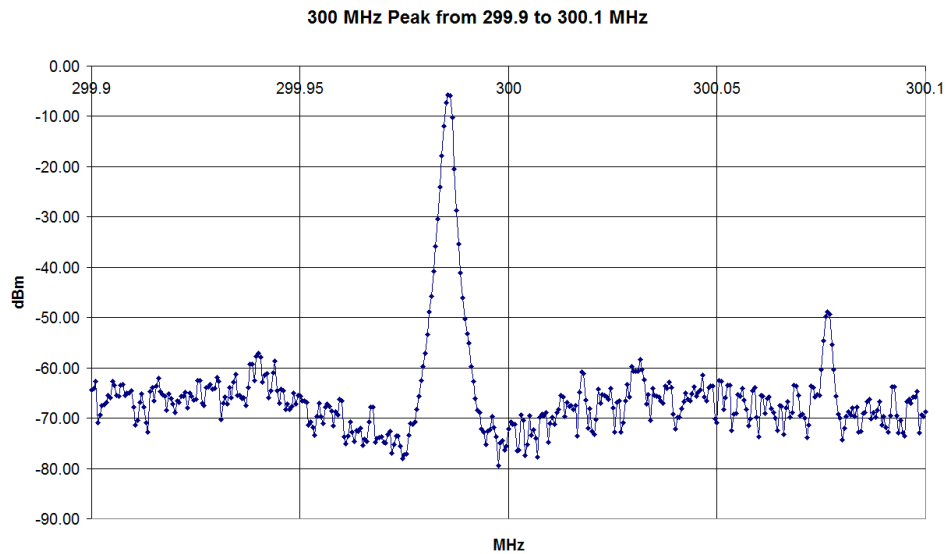


Figure 27: 300 MHz peak viewed from 299.9 MHz to 300.1 MHz. Background noise still peaks at about -50 dBm.

## 4 Conclusion

The creation of a radio frequency source for ultra-cold atom manipulation has been largely completed and can rapidly perform sweeps in the range of 0 to 400 MHz. A circuit composed of logic gates and shift registers has been designed, which is meant to improve the timing performance of the device, though unfortunately construction of this circuitry has only just begun. Nonetheless, the device should in theory be able to be used for driving transitions between hyperfine levels, with applications for interferometry. Other applications of the device include use for RF evaporation and as a generic signal source.

Further work on this project includes the completion of the registry circuitry. Reducing the time delay and jitter associated with the trigger is important for repeatable experiments. Programming could also be expanded, as manipulating the frequency of output signals was the primary focus of this project. Of particular interest are features such as output shift keying and amplitude sweeps. Most importantly, the device needs to be thoroughly tested with a single sideband modulator mixer to test whether the device will be useful for microwave manipulation of atoms.

## 5 Acknowledgements

I'd like to thank Professor Novikova and her lab group for allowing us to use their spectrum analyzer.

## References

- [1] Picture from Ian Leroux, “Manipulation of Ultra-Cold Atoms Using Radio-Frequency and Microwave Radiation”, B.A. Sc. Thesis, University of Toronto, April 2005.)
- [2] S. Aubin, M. H. T. Extavour, S. Myrskog, L. J. LeBlanc, J. Estve, S. Singh, P. Scrutton, D. McKay, R. McKenzie, I. Leroux, A. Stummer, and J. H. Thywissen. Trapping fermionic 40K and bosonic 87Rb on a chip. *Low Temp. Phys.* 140, 377 (2005)
- [3] Analog Devices, Inc. “AD9910: 1 GSPS 14-Bit, 3.3 V CMOS Direct Digital Synthesizer Data Sheet (Rev. B)”, [http://www.analog.com/static/imported-files/data\\_sheets/AD9910.pdf](http://www.analog.com/static/imported-files/data_sheets/AD9910.pdf) (December, 2008)
- [4] Analog Devices, Inc. “AD9910 Evaluation Tools: Evaluation Board Schematics, Rev. G”, [http://www.analog.com/static/imported-files/eval\\_boards/AD9910\\_Schematics.pdf](http://www.analog.com/static/imported-files/eval_boards/AD9910_Schematics.pdf) (December 10, 2007)

## Appendix A: Comments and Concerns

Thus far, the only problem encountered that might damage the AD9910 is having the DDS output at voltages above 0.5 V amplitude. All code for the AD9910 has been altered to decrease the output current of the DDS and the output load should be 50  $\Omega$ .

Presumably incorrectly connecting the power lines could also damage the AD9910 so care should be shown when connecting the power lines. As of this writing, the regulators are not labelled, although the voltages of each regulator can be determined from the part name printed on each regulator. Labelling the lines may prove more convenient, although I note that lines are connected to the AD9910's evaluation board in a fairly intuitive manner.

Operation of the device requires two power sources. The source powering the Rabbit microprocessor needs to be 5.3 V minimum, while the source powering the DDS needs to be 7 V minimum. Since the power source is connected through regulators, the power sources can be set to higher voltages with the heat of the regulators being the limiting factor. During testing, I used two 7 V sources to power the device without any problems.

The evaluation board does require, as noted in the thesis, a few modifications. The parts for the PLL loop filter have been left off by Analog Devices and need to be mounted. Additionally, a piece of wire was used to connect the filtered output on the older AD9910. The unfiltered output of the newer AD9910 is similarly disconnected, though the filtered output did come connected. Further connections will be necessary to utilize advanced features of the DDS. The outputs and inputs related to synchronizing multiple DDSs are disconnected from the DDS chip, for example.

The jumper configuration on the evaluation board is very simple. The jumper W7 should be set to REF\_CLK mode. Otherwise, jumpers should be set to the disable mode or open when disable is not an option.

The most time consuming problem when I set up the DDS was getting the reference clock to work properly. Problems related to the reference clock have now hopefully all been identified. As of this writing, both AD9910s are set up to receive a 10 MHz clock signal and multiply this to 1 GHz as use for the reference clock. Not all of the code is set up this way, however, as some older code is set up to work with a 50 MHz clock. Either clock signal setup is acceptable, although the PLL needs to be turned off in code if the 10 MHz signal is not used. Note that the clock signal should be 1 V p-p maximum and that the DDS tends not to work with unmultiplied clock signals below 50 MHz. Any unmultiplied clock signal above 50 MHz up to 1 GHz should work with the DDS, but the PLL is only setup to take a 10 MHz and multiply it up to 1 GHz at a multiplication factor of 100. Again, new AD9910s need to have the parts for the PLL loop filter mounted to the evaluation board before the PLL multiplier can be used.

I have not encountered heat problems with the AD9910, even when using the PLL multiplier for some time (on the order of 20 minutes). Still, I encourage heat sinking the AD9910 as an appropriate precaution.

The only other major time sink I encountered related to incorrectly programming the control registers. I have tried to make comments in my code when setting the control registers somewhat helpful, but setting the control registers remain one of the more obscure programming aspects. When altering the code that sets the control registers of the DDS, referring to the DDS's manual section on the control registers is usually necessary. Indeed, the register map section and subsequent register description section are the most useful reference sections in the DDS manual.

I have explored some of the amplitude control features. For some time, I was under the impression that the AD9910 will take amplitude parameters from the selected profile during a sweep, but this behavior was not observed. I found that the amplitude scale of sweeps

could be altered during a sweep by setting the DDS to OSK mode, and programming the ASF register. The ASF register is a register separate from the profile registers that controls the amplitude scale of the output when the DDS is operating in OSK mode (and amplitude is not the sweep parameter).

On the subject of OSK mode, this mode employs the OSK pin. When the OSK pin is low, the DDS outputs a 0 amplitude. When the OSK pin is high, the DDS outputs at the amplitude scale programmed into the ASF register. When the OSK pin switches from logic high to logic low, or vice versa, the DDS will either sweep between the programmed amplitude scale and zero at a preprogrammed rate, or the DDS will switch between the two instantly. Whether or not the DDS sweeps or switches instantly is determined by the setting of the OSK mode (whether it is set to manual or auto mode) in the control registers. If OSK mode is not enabled, then the OSK pin has no effect on the DDS.

Phase control is also a matter of interest. All I have found is that the DDS is not apparently phase locked to the clock signal provided to it, so I was unable to test phase change. Presumably, if multiple AD9910s are synchronized using the synchronization features, then the effect of phase changes from the DDS will be readily observable. I would advise testing this.

## Appendix B: Rabbit Sweep Generator Code Written For AD9910



```

/*****
* AD9910_Sweep_Waystation.c
* By Justin Winkler
*
* This program stores bytes necessary for programming the AD9910
* to perform frequency sweeps.
*
* COMMENTS NEED UPDATING FOR NEW VERSION OF CODE
*
* Bytes are received from an Ethernet connection and placed in
* an array. When triggered, the stored bytes are sent from the
* Rabbit to the AD9854, programming the DDS to perform
* sweeps between two frequencies with a desired delta frequency.
*
* This program will usually simply listen for a TCP connection.
* Once a connection is established, the Rabbit can be sent bytes which indicate
* particular operations for it to perform. As of this version, there are 4
* operations that can be performed.
* The first is to receive data from the Ethernet connection
* that will be used to program a sweep in the DDS, storing that
* data in an array. The second operation is to clear out the array of
* data for programming the DDS (this would be used if the end user had made
* a typo and sent incorrect data). The third operation is to master reset the
* DDS and program in some default settings to the DDS. The fourth operation
* stops the Rabbit from listening for TCP connections, and instead the Rabbit
* starts listening for a trigger. When triggered, the Rabbit programs the DDS
* to perform the stored sweeps, in the order that the Rabbit was sent. While
* waiting for a trigger, the Rabbit cannot establish a TCP connection, and
* this will continue until the Rabbit runs out of stored sweeps to perform.
*
* Once the appropriate indicator byte is sent to this program to
* store bytes for a sweep, this program expects a set number of
* bytes sent to it from an Ethernet connection, and it expects to receive
* the information in a particular order.
* The first byte of data is just a number indicating the direction
* of the sweep: 1 (or non-zero, really) to indicate a sweep from
* a higher frequency to a lower frequency, or 0 to indicate a
* sweep from low to high. The next 6 bytes should be the first
* frequency tuning word (FTW 1), which should be the lower of the
* two frequencies. The 6 bytes after that should be the second
* frequency tuning word (FTW 2), which is the higher frequency.
* The final 6 bytes should be the delta frequency word.
* Naturally, this read in setup leaves the burden of calculating
* tuning words to the program sending bytes to the Rabbit.
*
* As of this version, the DDS can be programmed to perform
* sweeps in any direction. The ramp rate that the DDS
* is set to is kept static, but sweeps can be performed over
* varying amounts of time by adjusting the delta frequency
* sent to this program. Tweaks have been made from version
* to version in the attempt to decrease the amount of time
* it takes the Rabbit to setup a sweep after being triggered.
*****/

/*****
* Pin Configuration
* List of connections between the Rabbit and the DDS
*
* Port E
*
* PE0 -> IO_RESET
* PE1 -> CSB (keep tied down for serial data transfer)
* PE3 -> SCLK
* PE4 -> SDO (planned, not currently connected,
* probably won't be used)
* PE5 -> SDIO
* PE6 -> DRHOLD
* PE7 -> DR_CTL

```

```

*
* Port F
*
* PF0 -> P_0
* PF1 -> P_1
* PF2 -> P_2
* PF3 -> DR_OVR (planned, not currently connected)
* PF4 -> OSK
* PF5 -> IO_UPDATE
* PF6 -> EXT_PWR_DWN
* PF7 -> RESET
*
* Port G
*
* PGO -> External Trigger
*****/

#class auto

/*****
* TCP Connection Macros and Constants
*****/

#define TCPCONFIG 1 // Macro used to setup the Rabbit for TCP communication

// More TCP macros
#define MY_IP_ADDRESS "10.10.6.101" // IP Address used by Rabbit
#define MY_NETMASK "255.255.255.0" // Netmask used by Rabbit
#define MY_GATEWAY "10.10.6.19" // Gateway used by Rabbit

// TCP relevant constants
#define PORT 1111 // Port to be used by the Rabbit when listening for a connection

/*****
* AD9910 Register Layout Constants
*
* List and definition of the serial
* addresses of each register and the
* number of bytes to be sent to each
* register. The default register
* settings are also defined.
*
* I'll leave notes with some general
* information about certain registers.
*
* Check the AD9910 manual for more
* information about the registers. Rev. A
* of the manual includes a register map
* and bit descriptions starting on page 49.
*
* Not all of these definitions are used,
* but I've gone ahead and defined all of
* them in case they prove useful later.
*****/

/*****
* General Control Register Information
*
* There are 3 control registers, each
* 4 bytes long. These registers determine
* which features of the AD9910 are in use,
* each bit or group of bits often serving
* as a way of selecting specific features.
* Full details can found in the manual,
* Rev. A, starting on page 54. I'll write
* a summary of the bits that control
* features of particular interest. If
* I haven't mentioned a control bit, then

```

```

* the default is what we want set for
* that bit at all times, at least for the
* purposes of this sweep program.
*****/

/*****
* Control Function Register 1 Information
* Default settings for CFR1 are
* <31:24> -> 0x00
* <23:16> -> 0x00
* <15:8> -> 0x00
* <7:0> -> 0x00
*
* CFR1<23> determines whether OSK is
* externally controlled. I am not yet
* sure whether we will be using OSK,
* but this could be potentially useful
* (remember, PE3 connects to the OSK
* pin on the evaluation board).
* Default setting is 0, which sets
* the OSK pin as inoperative.
*
* CFR1<16> determines whether the DDS
* outputs a cosine of sine wave (the
* default is 0 for cosine). This could
* potentially be a handy way of creating
* a 90 degree phase offset. Needs
* investigation.
*
* CFR1<15:10> all affect ways that
* accumulators or timers reset. Setting
* one of these bits high usually causes
* a particular register to reset on an
* I/O_UPDATE or PROFILE<2:0> change.
* I suspect most of these can be kept
* at 0 (default) for normal operation,
* but activating one or two of
* these may prove convenient.
* Testing is needed.
*
* CFR1<9:8> are OSK enable and auto OSK
* control bits (0 is disabled).
* I still need to see if we'll be
* using OSK.
*
* CFR1<7:3> activates or disables clock
* signals to certain parts of the DDS,
* allowing unused parts to be powered
* down to reduce current drawn by the
* DDS. This is a feature that we will
* probably want to utilize, eventually.
*****/

// CFR1 - Control Function Register 1
// Serial Address: 0x00
// Bytes Expected: 4
#define CFR1_ADDRESS 0x00
#define CFR1_BYTES_EXPECTED 4

/*****
* Default settings for CFR2 are
* <31:24> -> 0x00
* <23:16> -> 0x40 (SYNC_CLK enabled)
* <15:8> -> 0x08 (PDCLK enabled)
* <7:0> -> 0x20 (Sync timing validation
* disabled)
*
* CFR2<25> controls whether the DROVER pin

```

```

* is in use. Set this to 1 and the DROVER
* pin will go high after a sweep completes.
* This could be used to trigger sweeps
* immediately after previous sweeps have
* ended. This has potential use for testing,
* at the very least. May also see later use
* to help time experiments, though I don't
* know the details about how timing will work
* just yet.
*
* CFR2<21:20> effectively determines the type
* of ramp that the DRG will perform. If these
* two bits are set to 00, then the DDS signal
* control parameter that will be ramped is
* frequency. Likewise, 01 corresponds to phase
* and 1x (10 or 11) corresponds to amplitude.
* Initially, we'll just want to ramp frequency,
* but setting up phase and amplitude ramps
* are good features to add at some point.
* By the way, note that this control
* scheme means that only one of these
* control parameters may be ramped at a time
* (the DDS cannot simultaneously perform
* a frequency sweep and an amplitude sweep).
*
* CFR2<19> is the digital ramp enable bit.
* Set this to 1 at almost all times (only
* disable if switching to single tone mode
* proves necessary).
*
* CFR2<11> controls whether the PDCLK pin is
* enabled or not (0 = disabled, 1 = enabled).
* PDCLK is meant to serve as a data clock
* for the parallel data port, which we will not
* be using. PDCLK is by default enabled and
* may as well be turned off, so set this bit
* to 0.
*
* CFR2<7> is the matched latency enable bit,
* which determines whether or not simultaneous
* changes to amplitude, phase, and frequency
* arrive at the output in the order listed or
* simultaneously. Default is that the changes
* arrive at the output in the order listed.
* Turning on matched latency enable might
* prove useful.
*****/

// CFR2 - Control Function Register 2
// Serial Address: 0x01
// Bytes Expected: 4
#define CFR2_ADDRESS 0x01
#define CFR2_BYTES_EXPECTED 4

/*****
* Default settings for CFR3 are
* <31:24> -> 0x1F (VCO SEL set to 111
* meaning PLL is bypassed)
* <23:16> -> 0x3F (I_cp set to 111 meaning
* the charge pump current
* is set to 387 micro Amps)
* <15:8> -> 0x40 (REFCLK input divider is
* selected and operating normally)
* <7:0> -> 0x00
*
* CFR3<26:24> is the VCO SEL bit. Appears to affect
* the frequency range the PLL can achieve. Will
* probably need to be set to 101 (VCO5) to achieve

```

```

* the desired 1 GHz sysclock frequency, although the
* manual suggests that this varies between devices.
* Testing of the PLL should be done very carefully.
*
* CFR3<21:19> selects the charge pump current in
* the REFCLK PLL. I am not at all sure if we want
* to use this or not.
*
* CFR3<8> is the PLL enable bit. Default is 0 to bypass
* the REFCLK PLL. This will need to be enabled to
* reach the desired frequencies, probably.
*
* CFR3<7:1> are called N and the 7-bit number
* is the divide modulus of the REFCLK PLL feedback
* divider. I assume this sets the amount that
* the system frequency is multiplied by
* but I should double check this with
* Professor Aubin.
*****/

// CFR3 - Control Function Register 3
// Serial Address: 0x02
// Bytes Expected: 4
#define CFR3_ADDRESS 0x02
#define CFR3_BYTES_EXPECTED 4

/*****
* Auxiliary DAC Control Register Information
*
* Default Value is 0x7F (127 in decimal)
*
* Although there are 4 bytes assigned to the Auxiliary
* DAC Control register, 3 bytes are left open.
* The 8-bits that are used control the "full-scale
* output current of the main DAC." In other words,
* this register can alter the output current according
* to an equation in the manual (page 23 or Rev. A).
* May be useful in later phases, but not for
* early testing.
*****/

// Auxillary DAC Control
// Serial Address: 0x03
// Bytes Expected: 4
#define AUX_DAC_CONTROL_ADDRESS 0x03
#define AUX_DAC_CONTROL_BYTES_EXPECTED 4

/*****
* I/O Update Rate Register informationn
*
* Default value is 0xFFFFFFFF.
*
* This register controls the rate of I/O updates if
* I/O updates are being internally generated. Since
* I/O updates being externally controlled (CFR2<23> = 0)
* for this version of code, this register will not be used.
*****/

// I/O Update Rate
// Serial Address: 0x04
// Bytes Expected: 4
#define IO_UPDATE_RATE_ADDRESS 0x04
#define IO_UPDATE_RATE_BYTES_EXPECTED 4

/*****
* Frequency Tuning Word (FTW) Register Information
*
* This is a 32-bit register that controls the output

```

```

* frequency of the DDS.
*
* From the manual, the equation that describes the
* relationship between the FTW and the output frequency
* is
*
*  $f_{out} = (FTW * sysclock\_frequency) / (2^{32})$ 
*
* The manual suggests this register is only
* used during RAM mode, so this register will probably
* not be used.
*****/

// FTW - Frequency Tuning Word
// Serial Address: 0x07
// Bytes Expected: 4
#define FTW_ADDRESS 0x07
#define FTW_BYTES_EXPECTED 4

/*****
* Phase Offset Word (POW) Register Information
*
* This is a 16-bit register that controls the phase
* of the output signal.
*
* From the manual, the phase offset generated by the
* DDS is
*
*  $Phase\ Offset\ (in\ Radians) = (2 * \pi * POW) / (2^{16})$ 
*
* The manual suggests this register is only
* used during RAM mode, so this register will probably
* not be used.
*****/

// POW - Phase Offset Word
// Serial Address: 0x08
// Bytes Expected: 2
#define POW_ADDRESS 0x08
#define POW_BYTES_EXPECTED 2

/*****
* Amplitude Scale Factor (ASF) Register Information
*
* This is a 32-bit register that controls the amplitude
* of the output signal.
*
* The register has 3 parts.
*
* ASF<31:16> control the ramp rate of the amplitude,
* which is used if Output Shift Keying (OSK) is enabled
* and being automatically controlled.
*
* ASF<15:2> is the 14 bit amplitude scale factor.
* The relevant equation is
*
*  $Amplitude\ Scale = ASF / (2^{14})$ 
*
* Amplitude scale being the amplitude as a fraction of the
* full scale amplitude. This amplitude scale factor is the
* most important part of this register.
*
* ASF<1:0> controls the amplitude step size and like the
* ramp rate control is only used if OSK is enabled and
* being automatically controlled.
*
* The manual suggests this register is only
* used for RAM mode, so this register will probably

```

```

* not be used.
*****/

// ASF - Amplitude Scale Factor
// Serial Address: 0x09
// Bytes Expected: 4
#define ASF_ADDRESS 0x09
#define ASF_BYTES_EXPECTED 4

/*****
* Multichip Sync Register Information
*
* This is a register that controls various functions
* relating to synchronizing multiple devices. If I
* understand the manual, the sunchronization capabilities
* of the AD9910 are meant for when the parallel ports are
* in use so I do not believe that this register is useful.
*****/

// Multichip Sync
// Serial Address: 0x0A
// Bytes Expected: 4
#define MULTICHIP_SYNC_ADDRESS 0x0A
#define MULTICHIP_SYNC_BYTES_EXPECTED 4

/*****
* Digital Ramp Limit Register Information
*
* This 64-bit register sets the upper and lower limits of a
* digital ramp.
*
* Bits <63:32> describe the value of the upper limit
* of the ramp.
*
* Bits <31:0> describe the value of the lower limit of
* the ramp.
*
* Presumably the values set to this register
* should be the FTW, POW, or ASF that corresponds to the
* desired frequency, phase offset, or amplitude scale limits.
* Naturally, choosing whether to send a FTW, POW or ASF
* depends on the type of ramp the DDS will perform as
* determined by CFR2<21:20>.
*****/

// Digital Ramp Limit
// Serial Address: 0x0B
// Bytes Expected: 8
#define DR_LIMIT_ADDRESS 0x0B
#define DR_LIMIT_BYTES_EXPECTED 8

/*****
* Digital Ramp Step Size Register Information
*
* This 64-bit register sets the increment and decrement
* step size of a ramp.
*
* Bits <63:32> describe the value of the decrement step size.
*
* Bits <31:0> describe the value of the increment step size.
*
* During a ramp, after a set amount of time has passed,
* the output parameter is increased or decreased by a set
* amount. This register describes the amount the output
* parameter will be decreased during a downward sweep
* and increased during an upwards sweep.
*
* The step sizes can be calculated using the following

```

```

* equations from the manual:
*
* Frequency Step = (M * sysclock_frequency) / (2^32)
*
* Phase Step (in radians) = (pi * M) / (2^31)
*
* Amplitude Step = M * I_FS / 2^32
*
* Where M is the magnitude of the decrement/increment
* and I_FS is the full-scale output current of the DAC
* (presumably as controlled by the Auxiliary DAC Control
* Register).
*****/

// Digital Ramp Step
// Serial Address: 0x0C
// Bytes Expected: 8
#define DR_STEP_ADDRESS 0x0C
#define DR_STEP_BYTES_EXPECTED 8

/*****
* Digital Ramp Rate Register Information
*
* This 32-bit register sets the time interval between
* increment and decrement values.
*
* Bits <31:16> describe the negative slope value that
* defines the time interval between decrement values.
*
* The negative slope step interval is found
* using the following equation:
*
* -dt = 4N / sysclock_frequency
*
* where N is the 16-bit value stored in the ramp rate
* register and -dt is the negative slope step interval.
*
* Bits <15:0> describe the positive slope value that
* defines the time interval between increment values.
*
* The positive slope step interval is found
* using the following equation:
*
* +dt = 4P / sysclock_frequency
*
* where P is the 16-bit value stored in the ramp rate
* register and +dt is the positive slope step interval.
*****/

// Digital Ramp Rate
// Serial Address: 0x0D
// Bytes Expected: 4
#define DR_RATE_ADDRESS 0x0D
#define DR_RATE_BYTES_EXPECTED 4

/*****
* Profile Register Information
*
* The single tone and RAM profiles
* share serial addresses. The manual notes that
* RAM profiles are in effect when CFR1<31> = 1
* and the single tone profiles are in effect when
* CFR1<31> = 0, CFR2<19> = 0, and CFR2<4> = 0.
* See AD9910 manual, Rev. A, page 60 for details.
*
* My understanding is that the profiles can be
* easily switched between using the P0 - P2 pins
* and allow a user of the DDS to preprogram

```



```

* waveforms using the single tone or RAM mode.
*
* According to the manual, when a data parameter
* (frequency, phase, or amplitude) is selected
* by the DRG to be ramped, the other two parameters
* are supplied by the registers of the active profile.
* This means the profiles must be utilized.
*
* A profile is 64-bits. The bit descriptions are
* as follows:
*
* <63:62> are open.
*
* <61:48> designate the amplitude scale factor (ASF).
*
* As stated above,
*
* Amplitude Scale = ASF / (214)
*
* Amplitude scale being the amplitude as a fraction of the
* full scale amplitude.
*
* <47:32> designate the phase offset word (POW).
*
* As stated above, the phase offset generated by the
* DDS is
*
* Phase Offset (in Radians) = (2*pi*POW) / (216)
*
* <31:0> designate the frequency tuning word (FTW).
*
* As stated above, the equation that describes the
* relationship between the FTW and the output frequency
* is
*
* f_out = (FTW*sysclock_frequency) / (232)
*****/

// Single Tone Profile 0
// Serial Address: 0x0E
// Bytes Expected: 8
#define ST_PROFILE_0_ADDRESS 0x0E
#define ST_PROFILE_0_BYTES_EXPECTED 8

// RAM Profile 0
// Serial Address: 0x0E
// Bytes Expected: 8
#define RAM_PROFILE_0_ADDRESS 0x0E
#define RAN_PROFILE_0_BYTES_EXPECTED 8

// Single Tone Profile 1
// Serial Address: 0x0F
// Bytes Expected: 8
#define ST_PROFILE_1_ADDRESS 0x0F
#define ST_PROFILE_1_BYTES_EXPECTED 8

// RAM Profile 1
// Serial Address: 0x0F
// Bytes Expected: 8
#define RAM_PROFILE_1_ADDRESS 0x0F
#define RAN_PROFILE_1_BYTES_EXPECTED 8

// Single Tone Profile 2
// Serial Address: 0x10
// Bytes Expected: 8
#define ST_PROFILE_2_ADDRESS 0x10
#define ST_PROFILE_2_BYTES_EXPECTED 8

```

```

// RAM Profile 2
// Serial Address: 0x10
// Bytes Expected: 8
#define RAM_PROFILE_2_ADDRESS 0x10
#define RAN_PROFILE_2_BYTES_EXPECTED 8

// Single Tone Profile 3
// Serial Address: 0x11
// Bytes Expected: 8
#define ST_PROFILE_3_ADDRESS 0x11
#define ST_PROFILE_3_BYTES_EXPECTED 8

// RAM Profile 3
// Serial Address: 0x11
// Bytes Expected: 8
#define RAM_PROFILE_3_ADDRESS 0x11
#define RAN_PROFILE_3_BYTES_EXPECTED 8

// Single Tone Profile 4
// Serial Address: 0x12
// Bytes Expected: 8
#define ST_PROFILE_4_ADDRESS 0x12
#define ST_PROFILE_4_BYTES_EXPECTED 8

// RAM Profile 4
// Serial Address: 0x12
// Bytes Expected: 8
#define RAM_PROFILE_4_ADDRESS 0x12
#define RAN_PROFILE_4_BYTES_EXPECTED 8

// Single Tone Profile 5
// Serial Address: 0x13
// Bytes Expected: 8
#define ST_PROFILE_5_ADDRESS 0x13
#define ST_PROFILE_5_BYTES_EXPECTED 8

// RAM Profile 5
// Serial Address: 0x13
// Bytes Expected: 8
#define RAM_PROFILE_5_ADDRESS 0x13
#define RAN_PROFILE_5_BYTES_EXPECTED 8

// Single Tone Profile 6
// Serial Address: 0x14
// Bytes Expected: 8
#define ST_PROFILE_6_ADDRESS 0x14
#define ST_PROFILE_6_BYTES_EXPECTED 8

// RAM Profile 6
// Serial Address: 0x14
// Bytes Expected: 8
#define RAM_PROFILE_6_ADDRESS 0x14
#define RAN_PROFILE_6_BYTES_EXPECTED 8

// Single Tone Profile 7
// Serial Address: 0x15
// Bytes Expected: 8
#define ST_PROFILE_7_ADDRESS 0x15
#define ST_PROFILE_7_BYTES_EXPECTED 8

// RAM Profile 7
// Serial Address: 0x15
// Bytes Expected: 8
#define RAM_PROFILE_7_ADDRESS 0x15
#define RAN_PROFILE_7_BYTES_EXPECTED 8

/*****
* Miscellaneous macros and definitions

```

```

*****/

#include xmem
#include dcrtcp.lib

// Row and column length of the 2-D command list.
// I've simply opted to use a static array for storing
// data in the Rabbit, putting a limit on how many sweeps
// can be queued in the Rabbit.
#define CL_COLUMN_LEN 20 // Number of events that can be triggered
#define CL_ROW_LEN 13 // Amount of characters stored per event

// Some command list formatting numbers
// These are meant for navigating the command list with a for loop.

// This version doesn't use these much. They aren't, strictly speaking, necessary,
// though do make for good style when using them to navigate the
// command list with a for loop. To widdle down the time needed for
// programming a sweep, the for loops were removed in favor of
// hardcoding the array indices accessed, a technique which is slightly faster.
#define DIR_INDEX 0 // Indicates index of command list where the direction should be stored
#define FTW_1_START 1 // Indicates index of command list where the 6 bytes of FTW 1 start
#define FTW_2_START 5 // Indicates index of command list where the 6 bytes of FTW 2 start
#define DFW_START 9 // Indicates index of command list where the 6 bytes of delta frequency word start

// When receiving data from the Ethernet connection, if these bytes are received
// they indicate an operation to be performed by the Rabbit.
#define COMMAND_RECEIVE 0x00 // Indicates that the next CL_ROW_LEN worth of bytes from the connection
// are to be stored for programming frequency sweeps
#define COMMAND_CLEAR 0x01 // Empties the command list of any stored data.
#define COMMAND_SETUP_DDS 0x02 // Master restarts the DDS and programs some initial settings for the DDS
#define COMMAND_FINALIZE 0xff // Switch from listening for a connection to listening for a trigger.

char commandlist[CL_COLUMN_LEN][CL_ROW_LEN]; // Where all bytes used to program a sweep will be stored
int CLused; // Value of how many sweeps have been stored
int CLwrInd; // When writing to the command list, indicates which column to write to
int CLrdInd; // When reading from the command list, indicates which column to read from

tcp_socket commandsock; // Socket used for connections

/*****
 * sendByte
 *
 * Sends a single byte to the DDS.
 *
 * Return value: None
 *
 * Parameters:
 * oneByte - A character that has the value
 * of the byte to be sent to the DDS.
 *
 * Function Details:
 * The unsigned character oneByte is read using
 * bitwise operations and used to set the SDIO
 * pin to each bit value of oneByte (the pin
 * is set high or low). The rising edge of
 * the SCLK pin causes the DDS to register the
 * bit value currently set for SDIO, and doing
 * this 8 times sends a byte.
 *
 * Default operation of the AD9910 reads bytes
 * MSB-first(big endian) and this function is
 * written assuming the AD9910 is setup to
 * read MSB-first.
 *****/
void sendByte(unsigned char oneByte)
{

```

```

int c;

// I've written this routine partially using assembler, for the sake of speed.
// WARNING: This routine does not update the shadow registers for port E or F.

// For whatever reason, the following assembler routine did not work after mounting
// the DDS and the Rabbit to the box. The C version of this function worked fine,
// so that will be used for now.

// Regardless, I'm not very familiar with assembler, so the assembler version needs to be
// improved anyways.

/*
for(c = 7; c >= 0; c--) // Read through all 8 bits of oneByte, starting at the most significant byte
{
    if((oneByte >> c) & 01) // If the c-th bit of oneByte is 1, set SDIO to 1...
    {
        #asm
            set 5,a ; set bit 0 only (SDIO)
ioi ld (PEDR),a ; write data to port f
#endasm
    }
    else // ...otherwise set SDIO to 0
    {
        #asm
            res 5,a ; clear bit 0 only (SDIO)
ioi ld (PEDR),a ; write data to port f
#endasm
    }
    // Toggle SCLK so the DDS registers the value of SDIO
    #asm
        set 3,a ; set bit 6 only (SCLK)
ioi ld (PEDR),a ; write data to port e

        res 3,a ; clear bit 6 only (SCLK)
ioi ld (PEDR),a ; write data to port e
#endasm
    }
}
*/

// The below code performs the same operation, but uses pure
// C (slower), rather than assembler (faster)

// Note that the print commands should only ever be used for debugging,
// as they slow down the following routine even more than usual

//Print Hex Value of Sent Byte
// printf("Sending Byte, Hex Value: %x\n", oneByte);

for(c = 7; c >= 0; c--)
{
    // SDIO is connected to PE5
    BitWrPortI(PEDR, &PEDRShadow, (oneByte >> c) & 01, 5); // Input to SDIO

    // Print Binary Value of Sent Byte
    // printf("%d", (oneByte >> c) & 01);

    // SCLK is connected to PE3
    BitWrPortI(PEDR, &PEDRShadow, 1, 3); // Toggle SCLK
    BitWrPortI(PEDR, &PEDRShadow, 0, 3);
}

// printf("\n");
}

```

```

/*****
* boardSetup
*
* Sets up the value and operation of the Rabbit's pin outs.
*
* Return value: None
*
* Parameters: None
*
* Function Details: This is a function that is called once,
* at the beginning of the program. It's simply used
* for initial pin settings.
*
* As of this version, only Ports E, F, and potentially G are
* used, so those are the only ports that need to be configured.
* All pins except pin 3 of port F and pin 0 of port G are set
* to output, and all pins are set low initially
* (not that this effects the input pins PF3 and PG0).
*****/
void boardSetup()
{
    // Configure Port E
    WrtPortI(PECR, &PECRShadow, 0x00); // clear all bits to pclk/2
    WrtPortI(PEFR, &PEFRShadow, 0x00); // clear all bits to normal function
    WrtPortI(PEDDR, &PEDDRShadow, 0xFF); // set all bits to output
    WrtPortI(PEDR, &PEDRShadow, 0x00); // set all bits output low

    // Configure Port F
    WrtPortI(PFCR, &PFCRShadow, 0x00); // clear all bits for pclk/2
    WrtPortI(PFFR, &PFFRShadow, 0x00); // clear all bits for normal function
    WrtPortI(PFDCR, &PFDCRShadow, 0x00); // set all bits to totem-pole output
    WrtPortI(PFDDR, &PFDDRShadow, 0xFB); // set all bits but 3 to output
    // set bit 3 to input
    WrtPortI(PFDR, &PFDRShadow, 0x00); // set all bits low

    // Configure Port G
    WrtPortI(PGCR, &PGCRShadow, 0x00); // clear all bits for pclk/2
    WrtPortI(PGFR, &PGFRShadow, 0x00); // clear all bits for normal function
    WrtPortI(PGDCR, &PGDCRShadow, 0x00); // set all bits to totem-pole output
    WrtPortI(PGDDR, &PGDDRShadow, 0xFE); // set all bits but 0 to output
    // set bit 0 to input
    WrtPortI(PGDR, &PGDRShadow, 0x00); // set all bits low

    // printf("Board Setup\n");
}

/*****
* ioReset
*
* Toggles the IO RESET pin on the DDS.
*
* Return value: None
*
* Parameters: None
*
* Function Details: A simple helper function. This
* toggles PEO high then low. PEO is connected to the
* IO_RESET pin of the DDS. Calling this function
* thus restarts the input cycle.
*****/
void ioReset()
{
    BitWrtPortI(PEDR, &PEDRShadow, 1, 0);
    BitWrtPortI(PEDR, &PEDRShadow, 0, 0);

    // printf("IO Reset Toggled\n");
}

```

```

}

/*****
 * masterReset
 *
 * Toggles the MRESET pin on the DDS.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: A simple helper function. This
 * toggles PF7 high then low. PF7 is connected to the
 * RESET pin of the DDS (the master reset control).
 * Calling this function thus restarts the DDS,
 * clearing all memory elements and resetting all registers
 * to their default values.
 *****/

void masterReset()
{
    BitWrPortI(PFDR, &PFDRShadow, 1, 7);
    BitWrPortI(PFDR, &PFDRShadow, 0, 7);

    //    printf("Master Reset Toggled\n");
}

/*****
 * toggleUpdateClock
 *
 * Toggles the I/O UD pin on the DDS.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: A simple helper function. This
 * toggles PF5 high then low. PF5 is connected to the
 * IO_UPDATE pin of the DDS. This function thus controls
 * the update clock when the update clock is set to
 * be externally generated (which currently is the case for
 * this program). A rising edge of the update clock
 * transfers all buffered binary input to the
 * registers of the DDS.
 *****/

void toggleUpdateClock()
{
    BitWrPortI(PFDR, &PFDRShadow, 1, 5);
    BitWrPortI(PFDR, &PFDRShadow, 0, 5);

    //    printf("Update Clock Toggled\n");
}

/*****
 * DDSInitialSetup
 *
 * MAJOR REWRITE NEEDED FOR NEW VERSION
 *
 * Restarts the DDS and sets the control register and
 * various other registers to desired starting values.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: This function is called whenever the user
 * wishes to reset the DDS. The function resets the DDS and

```

```

* then programs the control register, FTW 1 register, and
* ramp rate to useful initial settings.
*
* This function turns off the AND gate providing the
* reference clock to the DDS. This is done because
* when the DDS is reset, it starts generating the
* update clock internally. Thus, the reference clock
* needs to be disconnected to set up the Rabbit to
* provide update clock as an external source.
* When the reference clock is disconnected, the update clock
* doesn't tick, preventing data transfers from being
* prematurely sent to the registers, which would cause
* data to be only partially transferred.
* When the bytes to set the control register are sent to the DDS,
* the bytes are all buffered while the DDS is unlocked.
* Only once the reference clock is reconnected are these
* buffered bytes then transferred to the register,
* which then puts the control of the update clock
* in the hands of the Rabbit.
*****/

void DDSInitialSetup()
{
    masterReset();
    ioReset();

    //printf("Setting initial control register...\n");

    // printf("Setting hardcoded ramp rate...\n");
    // Hardcoded ramp rate
    // Ramp rate is set to 1 for both directions
    sendByte(DR_RATE_ADDRESS);
    sendByte(0x00);
    sendByte(0x01);
    sendByte(0x00);
    sendByte(0x01);

    // Clear contents of ramp limit registers
    sendByte(DR_LIMIT_ADDRESS);
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x00);

    sendByte(CFR1_ADDRESS); // Address of CFR1
    sendByte(0x00); // AD9910 default
    sendByte(0x00); // AD9910 default
    sendByte(0x40); // Autoclear digital ramp accumulator
    sendByte(0x00); // AD9910 default

    // sendByte(CFR2_ADDRESS); // Address of CFR2
    // sendByte(0x00); // AD9910 default
    // sendByte(0x40); // AD9910 default
    // sendByte(0x00); // Disable PDCLK
    // sendByte(0x20); // AD9910 default

    sendByte(CFR2_ADDRESS); // Address of CFR2
    sendByte(0x00); // AD9910 default
    sendByte(0x48); // Enable digital ramp
    // Set to ramp frequency
    sendByte(0x00); // Disable PDCLK
    sendByte(0x20); // AD9910 default

    // Setting for control register 3 without

```

```

        // using multiplier

/*
sendByte(CFR3_ADDRESS); // Address of CFR3
sendByte(0x1F); // AD9910 default
sendByte(0x3F); // AD9910 default
sendByte(0xC0); // Bypass Input Divider
sendByte(0x00); // AD9910 default
*/

// Setting for control register 3
// using multiplier

sendByte(0x02); // Address of CFR3
sendByte(0x35); // DRV0 set so that REFCLK_OUT outputs a high current
// Use VCO5 for PLL output range of 820-1150 MHz
sendByte(0x3F); // AD9910 default
sendByte(0xC1); // Enable PLL, bypass input divider
sendByte(0xC8); // Set N to 100

toggleUpdateClock(); // Transfer all of these commands to the registers of
// the DDS

BitWrPortI(PEDR, &PEDRShadow, 1, 7);
}

main() {
char bytein; // Used for reading bytes from the Ethernet connection
int status; // Integer used for TCP functions
    int listen; // Used to indicate whether the Rabbit is listening for an Ethernet connection or not
    int c, r; // Used for for loops

listen = 1; // Start off listening for an Ethernet connection

    boardSetup();

    // Clear the command list to all zeros
    for(c = 0; c < CL_COLUMN_LEN; c++)
    {
        for(r = 0; r < CL_ROW_LEN; r++)
        {
            commandlist[c][r] = 0;
        }
    }

    CLused = 0; // The command list is considered empty
    CLwrInd = 0; // Start writing at index zero
    CLrdInd = 0; // Start reading from index zero

DDSInitialSetup();

    sock_init();

    while(1)
    {
        while(listen)
        {
            // Functions used for listening for an Ethernet connection

            printf("Listening for a connection\n");
            tcp_listen(&commandsock, PORT, 0, 0, NULL, 0);
            sock_wait_established(&commandsock, 0, NULL, &status);

            printf("Receiving incoming connection\n");
            sock_mode(&commandsock, TCP_MODE_ASCII);

            while(tcp_tick(&commandsock))
        {

```



```

    sock_wait_input(&commandsock,0,NULL,&status); // Wait for input

bytein = sock_getc(&commandsock); // Get a character from the socket

    printf("Received byte: %x\n", bytein);

    // Process bytein
    switch(bytein)
    {
    case COMMAND_RECEIVE: // Fills current row with bytes sent through Ethernet
        if(CLused < CL_COLUMN_LEN) // Only read in bytes if the command list is not full
        {
            printf("Filling Command List Row\n");

            // Get CL_ROW_LEN characters from the socket
            for(r = 0; r < CL_ROW_LEN; r++)
            {
                sock_wait_input(&commandsock,0,NULL,&status); // Wait for input
                commandlist[CLwrInd][r] = sock_getc(&commandsock); // Get a character from the socket
                printf("Received byte: %x\n", commandlist[CLwrInd][r]);
            }

            CLused++; // Indicate that another row of the socket has been filled

            // Have command list write index indicator circle through command list
            if(CLwrInd == CL_COLUMN_LEN - 1)
            {
                CLwrInd = 0;
            }
            else
            {
                CLwrInd++;
            }
        }
        else
        {
            printf("Command List is Full\n");

            // Get the next CL_ROW_LEN characters from the socket,
            // but do nothing with them.
            // This prevents the switch statement from processing
            // bytes meant to be stored in the command list.
            for(r = 0; r < CL_ROW_LEN; r++)
            {
                sock_wait_input(&commandsock, 0, NULL, &status);
                sock_getc(&commandsock);
            }
        }
        break;

    case COMMAND_CLEAR: // Clear command list
        printf("Clearing Command List\n");

        // Clears the command list and all command list
        // related variables to zero
        for(c = 0; c < CL_COLUMN_LEN; c++)
        {
            for(r = 0; r < CL_ROW_LEN; r++)
            {
                commandlist[c][r] = 0;
            }
        }

        CLused = 0;
        CLwrInd = 0;
        CLrdInd = 0;
        break;
    case COMMAND_SETUP_DDS: // Effectively restart the DDS and program a few initial settings

```

```

        DDSInitialSetup();
        break;
        case COMMAND_FINALIZE: // Switch the DDS over to listening for a digital trigger,
        // rather than an Ethernet connection
            listen = 0;
        default: // Do nothing for unspecified cases
            break;
    }
}

sock_err:
    switch(status) // Handle errors
    {
        case 1: /* foreign host closed */
            printf("User closed session\n");
            break;
        case -1: /* time-out */
            printf("\nConnection timed out\n");
            break;
    }
    while(!listen)
    {
        costate
        {
            //printf("Current logic value of Port G, pin 0: %d\n", BitRdPortI(PGDR, 0));
            // if (!BitRdPortI(PGDR, 0)) // Wait for trigger (read PG0)
            // abort;
            // waitfor(DelayMs(1)); // Waits 50 milliseconds to see if the trigger is released.
            // This sort of wait will be removed when triggered by the ADWIN,
            // but I'm currently triggering this manually for development.
            if (BitRdPortI(PGDR, 0))
            {
                //printf("Trigger received...\n");
                if(CLused >= 1)
                {
                    /*
                    // Print out command list row to be sent, for debugging purposes
                    for(c = 0; c < 19; c++)
                    {
                        if(c == 0)
                        {
                            if(commandlist[CLrdInd][c])
                            {
                                printf("Sweep from high to low");
                            }
                        }
                        else
                        {
                            printf("Sweep from low to high");
                        }
                    }
                    else
                    {
                        if(c == FTW_1_START)
                        {
                            printf("\nFTW 1\n");
                        }
                        if(c == FTW_2_START)
                        {
                            printf("\nFTW 2\n");
                        }
                        if(c == DFW_START)
                        {
                            printf("\nDFW\n");
                        }
                        printf("%x ", commandlist[CLrdInd][c]);
                    }
                }
            }
        }
    }

```

```

}
printf("\n\n");

    /*

    if(commandlist[CLrdInd][0])
    {
        // Handle sweep from high to low

        // Send step size to DDS
        // Will set decrement size
        // as designated by index 9 through 12
        // Increment size will be
        // as large as possible
        sendByte(DR_STEP_ADDRESS); // Address for step size
        // Set decrement step size
        sendByte(commandlist[CLrdInd][9]);
        sendByte(commandlist[CLrdInd][10]);
        sendByte(commandlist[CLrdInd][11]);
        sendByte(commandlist[CLrdInd][12]);
        // Set increment step size
        sendByte(0xFF);
        sendByte(0xFF);
        sendByte(0xFF);
        sendByte(0xFF);

        /*
        // Set both ramp limits to starting frequency
        sendByte(DR_LIMIT_ADDRESS); // Address for ramp limits
        // Set upper limit to higher frequency
        sendByte(commandlist[CLrdInd][1]);
        sendByte(commandlist[CLrdInd][2]);
        sendByte(commandlist[CLrdInd][3]);
        sendByte(commandlist[CLrdInd][4]);
        // Set lower limit to higher frequency
        sendByte(commandlist[CLrdInd][1]);
        sendByte(commandlist[CLrdInd][2]);
        sendByte(commandlist[CLrdInd][3]);
        sendByte(commandlist[CLrdInd][4]);
        */

        //toggleUpdateClock();

        // Set both ramp limits to appropriate limit frequencies
        sendByte(DR_LIMIT_ADDRESS); // Address for ramp limits
        // Set upper limit to higher frequency
        sendByte(commandlist[CLrdInd][1]);
        sendByte(commandlist[CLrdInd][2]);
        sendByte(commandlist[CLrdInd][3]);
        sendByte(commandlist[CLrdInd][4]);
        // Set lower limit to lower frequency
        sendByte(commandlist[CLrdInd][5]);
        sendByte(commandlist[CLrdInd][6]);
        sendByte(commandlist[CLrdInd][7]);
        sendByte(commandlist[CLrdInd][8]);

        toggleUpdateClock();

        // Set DRCTL to start high
        BitWrPortI(PEDR, &PEDRShadow, 1, 7);

        // Set DRCTL down to begin sweep from high to low
        BitWrPortI(PEDR, &PEDRShadow, 0, 7);
    }
    else
    {
        // Handle sweep from low to high

        // Send step size to DDS

```

```

        // Will set increment size
        // as designated by index 9 through 12
// Decrement size will be
        // as large as possible
        sendByte(DR_STEP_ADDRESS); // Address for step size
        // Set decrement step size
        sendByte(0xFF);
        sendByte(0xFF);
        sendByte(0xFF);
        sendByte(0xFF);
        // Set increment step size

        sendByte(commandlist[CLrdInd][9]);
        sendByte(commandlist[CLrdInd][10]);
        sendByte(commandlist[CLrdInd][11]);
        sendByte(commandlist[CLrdInd][12]);

        /*
        // Set both ramp limits to starting frequency
        sendByte(DR_LIMIT_ADDRESS); // Address for ramp limits
        // Set upper limit to lower frequency
        sendByte(commandlist[CLrdInd][5]);
        sendByte(commandlist[CLrdInd][6]);
        sendByte(commandlist[CLrdInd][7]);
        sendByte(commandlist[CLrdInd][8]);
// Set lower limit to lower frequency
        sendByte(commandlist[CLrdInd][5]);
        sendByte(commandlist[CLrdInd][6]);
        sendByte(commandlist[CLrdInd][7]);
        sendByte(commandlist[CLrdInd][8]);
        */

        //toggleUpdateClock();

        // Set both ramp limits to appropriate limit frequencies
        sendByte(DR_LIMIT_ADDRESS); // Address for ramp limits
        // Set upper limit to higher frequency
        sendByte(commandlist[CLrdInd][1]);
        sendByte(commandlist[CLrdInd][2]);
        sendByte(commandlist[CLrdInd][3]);
        sendByte(commandlist[CLrdInd][4]);
// Set lower limit to lower frequency
        sendByte(commandlist[CLrdInd][5]);
        sendByte(commandlist[CLrdInd][6]);
        sendByte(commandlist[CLrdInd][7]);
        sendByte(commandlist[CLrdInd][8]);

        toggleUpdateClock();

// Set DRCTL to start low
BitWrPortI(PEDR, &PEDRShadow, 0, 7);

// Set DRCTL high to begin sweep from low to high
BitWrPortI(PEDR, &PEDRShadow, 1, 7);
    }

    // Have command list read index indicator circle through command list
    if(CLrdInd == CL_COLUMN_LEN - 1)
    {
        CLrdInd = 0;
    }
    else
    {
        CLrdInd++;
    }

    // The row of command list that has just been read is not emptied
    // but is free to be written over. Value of CLused prevents

```

```

        // reusing the row until it has been written over.
        CLused--;

        // When the all queued sweeps have been programmed,
        // switch back to listening for an Ethernet connection
        if(CLused == 0)
        {
            printf("Command List is Empty\n");
            listen = 1;
        }

    }

    // In case the Rabbit was switched to listening for a trigger
    // when the command list was empty, switch back to listen
    // for an Ethernet connection.
    else
    {
        printf("Command List is Empty\n");
        listen = 1;
    }

}

}

}
}

```

## Appendix C: Rabbit Sweep Generator Code Written For AD9854

```

/*****
* AD9854_Sweep_Waystation.c
* By Justin Winkler
*
* This program stores bytes necessary for programming the AD9854
* to perform frequency sweeps.
*
* Bytes are received from an Ethernet connection and placed in
* an array. When triggered, the stored bytes are sent from the
* Rabbit to the AD9854, programming the DDS to perform
* sweeps between two frequencies with a desired delta frequency.
*
* This program will usually simply listen for a TCP connection.
* Once a connection is established, the Rabbit can be sent bytes which indicate
* particular operations for it to perform. As of this version, there are 4
* operations that can be performed.
* The first is to receive data from the Ethernet connection
* that will be used to program a sweep in the DDS, storing that
* data in an array. The second operation is to clear out the array of
* data for programming the DDS (this would be used if the end user had made
* a typo and sent incorrect data). The third operation is to master reset the
* DDS and program in some default settings to the DDS. The fourth operation
* stops the Rabbit from listening for TCP connections, and instead the Rabbit
* starts listening for a trigger. When triggered, the Rabbit programs the DDS
* to perform the stored sweeps, in the order that the Rabbit was sent. While
* waiting for a trigger, the Rabbit cannot establish a TCP connection, and
* this will continue until the Rabbit runs out of stored sweeps to perform.
*
* Once the appropriate indicator byte is sent to this program to
* store bytes for a sweep, this program expects a set number of
* bytes sent to it from an Ethernet connection, and it expects to receive
* the information in a particular order.
* The first byte of data is just a number indicating the direction
* of the sweep: 1 (or non-zero, really) to indicate a sweep from
* a higher frequency to a lower frequency, or 0 to indicate a
* sweep from low to high. The next 6 bytes should be the first
* frequency tuning word (FTW 1), which should be the lower of the
* two frequencies. The 6 bytes after that should be the second
* frequency tuning word (FTW 2), which is the higher frequency.
* The final 6 bytes should be the delta frequency word.
* Naturally, this read in setup leaves the burden of calculating
* tuning words to the program sending bytes to the Rabbit.
*
* As of this version, the DDS can be programmed to perform
* sweeps in any direction. The ramp rate that the DDS
* is set to is kept static, but sweeps can be performed over
* varying amounts of time by adjusting the delta frequency
* sent to this program. Tweaks have been made from version
* to version in the attempt to decrease the amount of time
* it takes the Rabbit to setup a sweep after being triggered.
*****/

/*****
* Pin Configuration
* List of connections between the Rabbit and the DDS
*
* Parallel Mode (for possible future use)
* D0 - D7 ----- PA0 - PA7
* A0 - A5 ----- PF0 - PF5
* MRESET ----- PE0
* OSK ----- PE3
* PMODE ----- PE4
* RDB ----- PE5
* WRB ----- PE6
* I/O UD ----- PE7
*
* Serial Mode
* SDIO (A0) ----- PF0

```

```

* SDO (A1) ----- PF1 (not used)
* IO RESET (A2) ----- PF2
* MRESET ----- PE0
* OSK ----- PE3
* PMODE ----- PE4
* CS (RDB) ----- PE5
* SCLK (WRB) ----- PE6
* I/O UD ----- PE7
*
* Additional Connections
* FSK/BPSK Input ----- PE1
* Trigger ----- PG0 (used to be PG1)
* REFCLK Connection Control ----- PG7
*****/

#class auto

/*****
* TCP Connection Macros and Constants
*****/

#define TCPCONFIG 1 // Macro used to setup the Rabbit for TCP communication

// More TCP macros
#define MY_IP_ADDRESS "10.10.6.101" // IP Address used by Rabbit
#define MY_NETMASK "255.255.255.0" // Netmask used by Rabbit
#define MY_GATEWAY "10.10.6.19" // Gateway used by Rabbit

// TCP relevant constants
#define PORT 1111 // Port to be used by the Rabbit when listening for a connection

/*****
* AD9854 Register Layout Constants
*
* List of the serial addresses of each
* register and the number of bytes to be
* sent to each register.
*****/

// Not all of these are used, but I've
// gone ahead and defined all of them
// in case they prove useful later

#define PHASE_ADJUST_1_SERIAL_ADDRESS 0x00
#define PHASE_ADJUST_1_BYTES_EXPECTED 2

#define PHASE_ADJUST_2_SERIAL_ADDRESS 0x01
#define PHASE_ADJUST_2_BYTES_EXPECTED 2

#define FTW_1_SERIAL_ADDRESS 0x02
#define FTW_1_BYTES_EXPECTED 6

#define FTW_2_SERIAL_ADDRESS 0x03
#define FTW_2_BYTES_EXPECTED 6

#define DFW_SERIAL_ADDRESS 0x04
#define DFW_BYTES_EXPECTED 6

#define UPDATE_CLOCK_RATE_SERIAL_ADDRESS 0x05
#define UPDATE_CLOCK_RATE_BYTES_EXPECTED 4

#define RAMP_RATE_CLOCK_SERIAL_ADDRESS 0x06
#define RAMP_RATE_CLOCK_BYTES_EXPECTED 3

#define CONTROL_REGISTER_SERIAL_ADDRESS 0x07
#define CONTROL_REGISTER_BYTES_EXPECTED 4

#define OSK_I_MULT_SERIAL_ADDRESS 0x08

```



```

#define OSK_I_MULT_BYTES_EXPECTED 2

#define OSK_Q_MULT_SERIAL_ADDRESS 0x09
#define OSK_I_MULT_BYTES_EXPECTED 2

#define OSK_RAMP_RATE_SERIAL_ADDRESS 0x0A
#define OSK_RAMP_RATE_BYTES_EXPECTED 1

#define QDAC_SERIAL_ADDRESS 0x0B
#define QDAC_BYTES_EXPECTED 2

/*****
 * A few useful settings for the control register bits
 *****/

// Meant for bits 24 through 31 of control register
#define CONTROL_1_DEFAULT 0x10 // AD9854 default settings

// Meant for bits 16 through 23 of control register
#define CONTROL_2_DEFAULT 0x64 // AD9854 default settings

// Meant for bits 8 through 15 of control register
#define CONTROL_3_DEFAULT 0x01 // AD9854 default settings
#define CONTROL_3_RAMPED_FSK_UDCLK_EXT 0x04 // Set mode to ramped FSK and update clock to external
#define CONTROL_3_SINGLETONE_UDCLK_EXT 0x00 // Set mode to single tone and update clock to external
#define CONTROL_3_CLR_ACC_1_RAMPED_FSK_UDCLK_EXT 0x84 // Set mode to ramped FSK and
// update clock to external
// and clear the frequency accumulator
#define CONTROL_3_CLR_ACC_2_RAMPED_FSK_UDCLK_EXT 0x44 // Set mode to ramped FSK and update
// clock to external and clear both phase
// and frequency accumulators
#define CONTROL_3_CLR_ACC_1_SINGLETONE_UDCLK_EXT 0x80 // Set mode to single tone and update
// clock to external and clear the
// frequency accumulator

// Meant for bits 0 through 7 of control register
#define CONTROL_4_DEFAULT 0x20 // AD9854 default settings
#define CONTROL_4_OSK_BYPASS 0x00 // Bypasses OSK
#define CONTROL_4_OSK_INT 0x30 // OSK on and internally generated

/*****
 * Miscellaneous macros and definitions
 *****/

#memmap xmem
#use dcrtcp.lib

// Row and column length of the 2-D command list.
// I've simply opted to use a static array for storing
// data in the Rabbit, putting a limit on how many sweeps
// can be queued in the Rabbit.
#define CL_COLUMN_LEN 20 // Number of events that can be triggered
#define CL_ROW_LEN 19 // Amount of characters stored per event

// Some command list formatting numbers
// These are meant for navigating the command list with a for loop.

// This version doesn't use these much. They aren't, strictly speaking, necessary,
// though do make for good style when using them to navigate the
// command list with a for loop. To widdle down the time needed for
// programming a sweep, the for loops were removed in favor of
// hardcoding the array indices accessed, a technique which is slightly faster.
#define DIR_INDEX 0 // Indicates index of command list where the direction should be stored
#define FTW_1_START 1 // Indicates index of command list where the 6 bytes of FTW 1 start
#define FTW_2_START 7 // Indicates index of command list where the 6 bytes of FTW 2 start
#define DFW_START 13 // Indicates index of command list where the 6 bytes of delta frequency word start

// When receiving data from the Ethernet connection, if these bytes are received

```

```

// they indicate an operation to be performed by the Rabbit.
#define COMMAND_RECEIVE 0x00 // Indicates that the next CL_ROW_LEN worth of bytes from the connection
// are to be stored for programming frequency sweeps
#define COMMAND_CLEAR 0x01 // Empties the command list of any stored data.
#define COMMAND_SETUP_DDS 0x02 // Master restarts the DDS and programs some initial settings for the DDS
#define COMMAND_FINALIZE 0xff // Switch from listening for a connection to listening for a trigger.

char commandlist[CL_COLUMN_LEN][CL_ROW_LEN]; // Where all bytes used to program a sweep will be stored
int CLused; // Value of how many sweeps have been stored
int CLwrInd; // When writing to the command list, indicates which column to write to
int CLrdInd; // When reading from the command list, indicates which column to read from

tcp_Socket commandsock; // Socket used for connections

/*****
 * sendByte
 *
 * Sends a single byte to the DDS.
 *
 * Return value: None
 *
 * Parameters:
 * oneByte - A character that has the value
 * of the byte to be sent to the DDS.
 *
 * Function Details:
 * The unsigned character oneByte is read using
 * bitwise operations and used to set the SDIO
 * pin to each bit value of oneByte (the pin
 * is set high or low). The rising edge of
 * the SCLK pin causes the DDS to register the
 * bit value currently set for SDIO, and doing
 * this 8 times sends a byte.
 *
 * Default operation of the AD9854 reads bytes
 * MSB-first(big endian) and this function is
 * written assuming the AD9854 is setup to
 * read MSB-first.
 *****/
void sendByte(unsigned char oneByte)
{
    int c;

    // I've written this routine partially using assembler, for the sake of speed.
    // WARNING: This routine does not update the shadow registers for port E or F.

    // For whatever reason, the following assembler routine did not work after mounting
    // the DDS and the Rabbit to the box. The C version of this function worked fine,
    // so that will be used for now.

    // Regardless, I'm not very familiar with assembler, so the assembler version needs to be
    // improved anyways.

    /*
    for(c = 7; c >= 0; c--) // Read through all 8 bits of oneByte, starting at the most significant byte
    {
        if((oneByte >> c) & 01) // If the c-th bit of oneByte is 1, set SDIO to 1...
        {
            #asm
                set 0,a ; set bit 0 only (SDIO)
            ioi ld (PFDR),a ; write data to port f
            #endasm
        }
        else // ...otherwise set SDIO to 0
        {
            #asm
                res 0,a ; clear bit 0 only (SDIO)
            */

```

```

ioi ld (PFDR),a ; write data to port f
    #endasm
}
// Toggle SCLK so the DDS registers the value of SDIO
#asm
    set 6,a ; set bit 6 only (SCLK)
ioi ld (PEDR),a ; write data to port e

    res 6,a ; clear bit 6 only (SCLK)
ioi ld (PEDR),a ; write data to port e
    #endasm
}
*/

// The below code performs the same operation, but uses pure
// C (slower), rather than assembler (faster)

// Note that the print commands should only ever be used for debugging,
// as they slow down the following routine even more than usual

//Print Hex Value of Sent Byte
//printf("Sending Byte, Hex Value: %x\n", oneByte);

for(c = 7; c >= 0; c--)
{
    BitWrPortI(PFDR, &PFDRShadow, (oneByte >> c) & 01, 0); // Input to SDIO

    // Print Binary Value of Sent Byte
    //printf("%d", (oneByte >> c) & 01);

    BitWrPortI(PEDR, &PEDRShadow, 1, 6); // Toggle SCLK
    BitWrPortI(PEDR, &PEDRShadow, 0, 6);
}

//printf("\n");
}

/*****
* boardSetup
*
* Sets up the value and operation of the Rabbit's pin outs.
*
* Return value: None
*
* Parameters: None
*
* Function Details: This is a function that is called once,
* at the beginning of the program. It's simply used
* for initial pin settings.
*
* As of this version, only Ports E, F, and G are
* used. All pins except pins 1 and 0 of port G are set to
* output, and all pins are set low initially (not that this
* effects the input pins PGO and PG1).
*
* Pin 1 of port G was to be used to read the trigger,
* but that pin was found to be broken, so pin 0 of port G
* is instead used for reading the trigger, which is why
* those two pins are set to output.
*****/
void boardSetup()
{
    // Configure Port A
    // This port would be used if the DDS is set to parallel mode,

```

```

// although whether this particular port can be used conventionally
// still needs to be tested.
WrPortI(PADR, &PADRShadow, 0x00);      //set all bits low

// Configure Port E
WrPortI(PECR, &PECRShadow, 0x00);      // clear all bits to pclk/2
WrPortI(PEFR, &PEFRShadow, 0x00);      // clear all bits to normal function
WrPortI(PEDDR, &PEDDRShadow, 0xFF);    // set all bits to output
WrPortI(PEDR, &PEDRShadow, 0x00);      // set all bits output low

// Configure Port F
WrPortI(PFCR, &PFCRShadow, 0x00);      //clear all bits for pclk/2
WrPortI(PFFR, &PFFRShadow, 0x00);      //clear all bits for normal function
WrPortI(PFDCR, &PFDCRShadow, 0x00);    //set all bits to totem-pole output
WrPortI(PFDDR, &PFDDRShadow, 0xFF);    //set all bits to output
WrPortI(PFDR, &PFDRShadow, 0x00);      //set all bits low

// Configure Port G
WrPortI(PGCR, &PGCRShadow, 0x00);      //clear all bits for pclk/2
WrPortI(PGFR, &PGFRShadow, 0x00);      //clear all bits for normal function
WrPortI(PGDCR, &PGDCRShadow, 0x00);    //set all bits to totem-pole output
WrPortI(PGDDR, &PGDDRShadow, 0xFC);    //set all bits but 1 and 0 to output
//set bits 1 and 0 to input
WrPortI(PGDR, &PGDRShadow, 0x00);      //set all bits low

// printf("Board Setup\n");
}

/*****
 * ioReset
 *
 * Toggles the IO RESET pin on the DDS.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: A simple helper function. This
 * toggles PF2 high then low. PF2 is connected to the
 * IO RESET (A2) pin of the DDS. Calling this function
 * thus restarts the input cycle.
 *****/
void ioReset()
{
    BitWrPortI(PFDR, &PFDRShadow, 1, 2);
    BitWrPortI(PFDR, &PFDRShadow, 0, 2);

// printf("IO Reset Toggled\n");
}

/*****
 * masterReset
 *
 * Toggles the MRESET pin on the DDS.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: A simple helper function. This
 * toggles PEO high then low. PEO is connected to the
 * MRESET pin of the DDS. Calling this function
 * thus restarts the DDS, resetting all registers to their
 * default values.
 *****/
void masterReset()
{

```

```

    BitWrPortI(PEDR, &PEDRShadow, 1, 0);
    BitWrPortI(PEDR, &PEDRShadow, 0, 0);

//    printf("Master Reset Toggled\n");
}

/*****
 * toggleUpdateClock
 *
 * Toggles the I/O UD pin on the DDS.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: A simple helper function. This
 * toggles PE7 high then low. PE7 is connected to the
 * I/O UD pin of the DDS. This function thus controls
 * the update clock when the update clock is set to
 * be externally generate (which is the case for this
 * program). A rising edge of the update clock transfers
 * all buffered binary input to the registers of the DDS
 *****/

void toggleUpdateClock()
{
    BitWrPortI(PEDR, &PEDRShadow, 1, 7);
    BitWrPortI(PEDR, &PEDRShadow, 0, 7);

//    printf("Update Clock Toggled\n");
}

/*****
 * DDSInitialSetup
 *
 * Restarts the DDS and sets the control register and
 * various other registers to desired starting values.
 *
 * Return value: None
 *
 * Parameters: None
 *
 * Function Details: This function is called whenever the user
 * wishes to reset the DDS. The function resets the DDS and
 * then programs the control register, FTW 1 register, and
 * ramp rate to useful initial settings.
 *
 * This function turns off the AND gate providing the
 * reference clock to the DDS. This is done because
 * when the DDS is reset, it starts generating the
 * update clock internally. Thus, the reference clock
 * needs to be disconnected to set up the Rabbit to
 * provide update clock as an external source.
 * When the reference clock is disconnected, the update clock
 * doesn't tick, preventing data transfers from being
 * prematurely sent to the registers, which would cause
 * data to be only partially transferred.
 * When the bytes to set the control register are sent to the DDS,
 * the bytes are all buffered while the DDS is unlocked.
 * Only once the reference clock is reconnected are these
 * buffered bytes then transferred to the register,
 * which then puts the control of the update clock
 * in the hands of the Rabbit.
 *****/

void DDSInitialSetup()
{
    int r;

```

```

BitWrPortI(PGDR, &PGDRShadow, 0, 7); // Disconnect REFCLK while setting up DDS

    masterReset();
    ioReset();

    //printf("Setting intial control register...\n");

    // Set control register
    // First Byte: 0x10, AD9854 default setting
    // Second Byte: 0x64, AD9854 default setting
    // Third Byte: 0x00, sets mode to single tone and update clock to external
    // Fourth Byte: 0x00, bypasses OSK
    sendByte(CONTROL_REGISTER_SERIAL_ADDRESS); // Serial Address of Control Register
    sendByte(CONTROL_1_DEFAULT); // First Byte Sent
    sendByte(CONTROL_2_DEFAULT); // Second Byte Sent
    sendByte(CONTROL_3_SINGLETONE_UDCLK_EXT); // Third Byte Sent
    sendByte(CONTROL_4_OS_K_BYPASS); // Fourth Byte Sent

    BitWrPortI(PGDR, &PGDRShadow, 1, 7); //Connect REFCLK

    //printf("Setting hardcoded ramp rate...\n");
    // Hardcoded ramp rate (Ramp rate is set to 1)
    sendByte(RAMP_RATE_CLOCK_SERIAL_ADDRESS); // Address for ramp rate
    // The ramp rate is 20 bits, but 4 bits need to be sent as dummy bits,
    // so 3 bytes are sent.
    sendByte(0x00);
    sendByte(0x00);
    sendByte(0x01);

    // Set FTW 1 to zero
    // DDS is to idle at 0 Hz until a sweep starts
    sendByte(FTW_1_SERIAL_ADDRESS); // FTW 1 serial address
    for(r = FTW_1_START; r < FTW_1_START + FTW_1_BYTES_EXPECTED; r++)
    {
        sendByte(0x00); // Set all 6 bytes of the register to zero
    }

    toggleUpdateClock(); // Transfer all of these commands to the registers of
    // the DDS
}

main() {
    char bytein; // Used for reading bytes from the Ethernet connection
    int status; // Integer used for TCP functions
    int fskon; // Indicates direction of sweep
    int listen; // Used to indicate whether the Rabbit is listening for an Ethernet connection or not
    int c, r; // Used for for loops

    listen = 1; // Start off listening for an Ethernet connection

    boardSetup();

    // Clear the command list to all zeros
    for(c = 0; c < CL_COLUMN_LEN; c++)
    {
        for(r = 0; r < CL_ROW_LEN; r++)
        {
            commandlist[c][r] = 0;
        }
    }

    CLused = 0; // The command list is considered empty
    CLwrInd = 0; // Start writing at index zero
    CLrdInd = 0; // Start reading from index zero

    DDSInitialSetup();

```

```

sock_init();

while(1)
{
    while(listen)
    {
        // Functions used for listening for an Ethernet connection

        //printf("Listening for a connection\n");
        tcp_listen(&commandsock,PORT,0,0,NULL,0);
        sock_wait_established(&commandsock,0,NULL,&status);

        //printf("Receiving incoming connection\n");
        sock_mode(&commandsock,TCP_MODE_ASCII);

        while(tcp_tick(&commandsock))
        {
            sock_wait_input(&commandsock,0,NULL,&status); // Wait for input
            bytein = sock_getc(&commandsock); // Get a character from the socket

            //printf("Received byte: %x\n", bytein);

            // Process bytein
            switch(bytein)
            {
                case COMMAND_RECEIVE: // Fills current row with bytes sent through Ethernet
                    if(CLused < CL_COLUMN_LEN) // Only read in bytes if the command list is not full
                    {
                        //printf("Filling Command List Row\n");

                        // Get CL_ROW_LEN characters from the socket
                        for(r = 0; r < CL_ROW_LEN; r++)
                        {
                            sock_wait_input(&commandsock,0,NULL,&status); // Wait for input
                            commandlist[CLwrInd][r] = sock_getc(&commandsock); // Get a character from the socket
                            //printf("Received byte: %x\n", commandlist[CLwrInd][r]);
                        }

                        CLused++; // Indicate that another row of the socket has been filled

                        // Have command list write index indicator circle through command list
                        if(CLwrInd == CL_COLUMN_LEN - 1)
                        {
                            CLwrInd = 0;
                        }
                        else
                        {
                            CLwrInd++;
                        }
                    }
                else
                {
                    //printf("Command List is Full\n");

                    // Get the next CL_ROW_LEN characters from the socket,
                    // but do nothing with them.
                    // This prevents the switch statement from processing
                    // bytes meant to be stored in the command list.
                    for(r = 0; r < CL_ROW_LEN; r++)
                    {
                        sock_wait_input(&commandsock, 0, NULL, &status);
                        sock_getc(&commandsock);
                    }
                }
            }
            break;

```

```

        case COMMAND_CLEAR: // Clear command list
            //printf("Clearing Command List\n");

            // Clears the command list and all command list
            // related variables to zero
            for(c = 0; c < CL_COLUMN_LEN; c++)
            {
                for(r = 0; r < CL_ROW_LEN; r++)
                {
                    commandlist[c][r] = 0;
                }
            }

            CLused = 0;
            CLwrInd = 0;
            CLrdInd = 0;
            break;
            case COMMAND_SETUP_DDS: // Effectively restart the DDS and program a few initial settings
                DDSinitialSetup();
                break;
            case COMMAND_FINALIZE: // Switch the DDS over to listening for a digital trigger,
                // rather than an Ethernet connection
                listen = 0;
            default: // Do nothing for unspecified cases
                break;
        }
    }

    sock_err:
    switch(status) // Handle errors
    {
        case 1: /* foreign host closed */
            //printf("User closed session\n");
            break;
        case -1: /* time-out */
            //printf("\nConnection timed out\n");
            break;
    }
    while(!listen)
    {
        costate
        {
            //printf("Current logic value of Port G, pin 0: %d\n", BitRdPortI(PGDR, 0));
            //if (!BitRdPortI(PGDR, 0)) // Wait for trigger (read PG0)
            //abort;
            //waitfor(DelayMs(50)); // Waits 50 milliseconds to see if the trigger is released.
            // This sort of wait will be removed when triggered by the ADWIN,
            // but I'm currently triggering this manually for development.

            if (BitRdPortI(PGDR, 0))
            {
                //printf("Trigger received...\n");
                if(CLused >= 1)
                {
                    /*
                    // Print out command list row to be sent, for debugging purposes
                    for(c = 0; c < 19; c++)
                    {
                        if(c == 0)
                        {
                            if(commandlist[CLrdInd][c])
                            {
                                printf("Sweep from high to low");
                            }
                        }
                        else
                        {
                            printf("Sweep from low to high");
                        }
                    }
                */
            }
        }
    }

```



```

}
}
else
{
if(c == FTW_1_START)
{
printf("\nFTW 1\n");
}
if(c == FTW_2_START)
{
printf("\nFTW 2\n");
}
if(c == DFW_START)
{
printf("\nDFW\n");
}
printf("%x ", commandlist[CLrdInd][c]);
}
}
printf("\n\n");
*/

fskon = commandlist[CLrdInd][0]; // Determines direction of sweep

if(fskon) // Handle case of sweep from high to low
{
// For whatever reason, the DDS is not great at programming
// sweeps from high to low. The way I've found that works
// for such sweeps is to first hold the DDS in single tone mode,
// then program the DFW and FTW 2 registers and set the FSK pin high,
// then switch to ramped FSK and program the FTW 1 register, and
// finally set the FSK pin low. This ensures that the down sweep can
// be performed properly from any previous setting.

// Toggle CLR ACC1 bit high to clear frequency accumulators
sendByte(CONTROL_REGISTER_SERIAL_ADDRESS);
sendByte(CONTROL_1_DEFAULT);
sendByte(CONTROL_2_DEFAULT);
sendByte(CONTROL_3_CLR_ACC_1_RAMPED_FSK_UDCLK_EXT);
sendByte(CONTROL_4_OSK_BYPASS);

// Set FTW 1 to higher frequency since that's the frequency
// we're starting at. This is a temporary setting for when
// the DDS is switched briefly to single tone mode
sendByte(FTW_1_SERIAL_ADDRESS);
sendByte(commandlist[CLrdInd][7]);
sendByte(commandlist[CLrdInd][8]);
sendByte(commandlist[CLrdInd][9]);
sendByte(commandlist[CLrdInd][10]);
sendByte(commandlist[CLrdInd][11]);
sendByte(commandlist[CLrdInd][12]);

// Toggle CLR ACC1 bit low
// Also change mode temporarily to single tone mode
sendByte(CONTROL_REGISTER_SERIAL_ADDRESS);
sendByte(CONTROL_1_DEFAULT);
sendByte(CONTROL_2_DEFAULT);
sendByte(CONTROL_3_SINGLETONE_UDCLK_EXT);
sendByte(CONTROL_4_OSK_BYPASS);

toggleUpdateClock();

// Set FSK pin to start high
BitWrPortI(PEDR, &PEDRShadow, 1, 1);

// Set FTW 2

```

```

        sendByte(FTW_2_SERIAL_ADDRESS);
        sendByte(commandlist[CLrdInd][7]);
        sendByte(commandlist[CLrdInd][8]);
        sendByte(commandlist[CLrdInd][9]);
        sendByte(commandlist[CLrdInd][10]);
        sendByte(commandlist[CLrdInd][11]);
        sendByte(commandlist[CLrdInd][12]);

        // Set DFW
        sendByte(DFW_SERIAL_ADDRESS);
        sendByte(commandlist[CLrdInd][13]);
        sendByte(commandlist[CLrdInd][14]);
        sendByte(commandlist[CLrdInd][15]);
        sendByte(commandlist[CLrdInd][16]);
        sendByte(commandlist[CLrdInd][17]);
        sendByte(commandlist[CLrdInd][18]);

        // Switch mode to ramped FSK
        sendByte(CONTROL_REGISTER_SERIAL_ADDRESS);
        sendByte(CONTROL_1_DEFAULT);
        sendByte(CONTROL_2_DEFAULT);
        sendByte(CONTROL_3_RAMPED_FSK_UDCLK_EXT);
        sendByte(CONTROL_4_OSK_BYPASS);

        // Set FTW 1 to lower frequency
        sendByte(FTW_1_SERIAL_ADDRESS);
        sendByte(commandlist[CLrdInd][1]);
        sendByte(commandlist[CLrdInd][2]);
        sendByte(commandlist[CLrdInd][3]);
        sendByte(commandlist[CLrdInd][4]);
        sendByte(commandlist[CLrdInd][5]);
        sendByte(commandlist[CLrdInd][6]);

        toggleUpdateClock();

        // Toggle FSK pin low to begin sweep
        BitWrPortI(PEDR, &PEDRShadow, 0, 1);

    }
    else // Handle case of sweep from low to high
    {
        // Toggle CLR ACC1 bit high to clear frequency accumulators
        sendByte(CONTROL_REGISTER_SERIAL_ADDRESS);
        sendByte(CONTROL_1_DEFAULT);
        sendByte(CONTROL_2_DEFAULT);
        sendByte(CONTROL_3_CLR_ACC_1_RAMPED_FSK_UDCLK_EXT);
        sendByte(CONTROL_4_OSK_BYPASS);

        // Set FTW 1 to lower frequency since that's the frequency we're starting at
        sendByte(FTW_1_SERIAL_ADDRESS);
        sendByte(commandlist[CLrdInd][1]);
        sendByte(commandlist[CLrdInd][2]);
        sendByte(commandlist[CLrdInd][3]);
        sendByte(commandlist[CLrdInd][4]);
        sendByte(commandlist[CLrdInd][5]);
        sendByte(commandlist[CLrdInd][6]);

        // Toggle CLR ACC1 bit low
        // Also change mode temporarily to single tone mode
        sendByte(CONTROL_REGISTER_SERIAL_ADDRESS);
        sendByte(CONTROL_1_DEFAULT);
        sendByte(CONTROL_2_DEFAULT);
        sendByte(CONTROL_3_SINGLETONE_UDCLK_EXT);
        sendByte(CONTROL_4_OSK_BYPASS);

        toggleUpdateClock();

        // Set FSK pin to start low

```

```

BitWrPortI(PEDR, &PEDRShadow, 0, 1);

    // Set FTW 2
    sendByte(FTW_2_SERIAL_ADDRESS);
    sendByte(commandlist[CLrdInd][7]);
    sendByte(commandlist[CLrdInd][8]);
    sendByte(commandlist[CLrdInd][9]);
    sendByte(commandlist[CLrdInd][10]);
    sendByte(commandlist[CLrdInd][11]);
    sendByte(commandlist[CLrdInd][12]);

    // Set DFW
    sendByte(DFW_SERIAL_ADDRESS);
    sendByte(commandlist[CLrdInd][13]);
    sendByte(commandlist[CLrdInd][14]);
    sendByte(commandlist[CLrdInd][15]);
    sendByte(commandlist[CLrdInd][16]);
    sendByte(commandlist[CLrdInd][17]);
    sendByte(commandlist[CLrdInd][18]);

    // Switch mode to ramped FSK
    sendByte(CONTROL_REGISTER_SERIAL_ADDRESS);
    sendByte(CONTROL_1_DEFAULT);
    sendByte(CONTROL_2_DEFAULT);
    sendByte(CONTROL_3_RAMPED_FSK_UDCLK_EXT);
    sendByte(CONTROL_4_OSK_BYPASS);

    toggleUpdateClock();

    // Toggle FSK pin high to begin sweep
    BitWrPortI(PEDR, &PEDRShadow, 1, 1);
}

// Have command list read index indicator circle through command list
if(CLrdInd == CL_COLUMN_LEN - 1)
{
    CLrdInd = 0;
}
else
{
    CLrdInd++;
}

// The row of command list that has just been read is not emptied
// but is free to be written over. Value of CLused prevents
// reusing the row until it has been written over.
CLused--;

// When the all queued sweeps have been programmed,
// switch back to listening for an Ethernet connection
if(CLused == 0)
{
    //printf("Command List is Empty\n");
    listen = 1;
}

}

// In case the Rabbit was switched to listening for a trigger
// when the command list was empty, switch back to listen
// for an Ethernet connection.
else
{
    //printf("Command List is Empty\n");
    listen = 1;
}

}
}

```

}  
}  
}