

Construction of an Agile, Controllable Radio Frequency Source

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor
of Science degree in Physics from the College of William and Mary

by

Harrison Cantor-Cooke

Advisor: Dr. Seth Aubin

Senior Research Coordinator: Henry Krakauer

Date: April 28, 2014

Abstract

Precise and smooth frequency generation is a crucial part of manipulating ultra-cold atoms. I developed a frequency source using an off-the-shelf Direct Digital Synthesizer chip and an Arduino Due microcontroller. The source can be programmed over the web to perform chained frequency sweeps triggered by an external controller.

I. Introduction

What do you need to manipulate something as small as an atom? What do you need to cool something into the micro-Kelvin range? As it turns out, a major component of this manipulation and cooling of ultra-cold atoms is a reliable source of frequencies in the Microwave to Radio Frequency (RF) range.

Hyperfine state manipulation requires very accurate frequencies in order to induce transitions between states. For an idea, three transition points for Potassium, used in Dr. Aubin's lab, are ^{39}K , ^{40}K , and ^{41}K which are 462MHz, 1285MHz, and 254MHz respectively. While this would seem a large range, the most important part of hyperfine state manipulation is that the frequencies be constant and smooth. We can use single-sideband mixing methods to step the accurate frequencies up (i.e. 462MHz from an accurate 62MHz + standard 400MHz) so the important part is that the 62MHz and 400MHz are accurate.

On the cooling front, RF evaporation comes after laser cooling, and is the methods used to get the atoms to their lowest temperature they will attain in the lab: the nano-Kelvin range. RF cooling an atom requires cutting off higher energy atoms by performing a downward frequency sweep from 400MHz to 0. This sweep must be performed smoothly and ideally in an exponential manner. Because the atoms are less sensitive to the exponential part, we can create a rough approximation with a series of smooth linear sweeps. Figure 1 shows the kind of pattern desired for evaporation. The pulses at the end are also needed for the evaporation along with the slope.

Precise timing is needed over the entire sweep sequence in order for the cooling to work optimally.

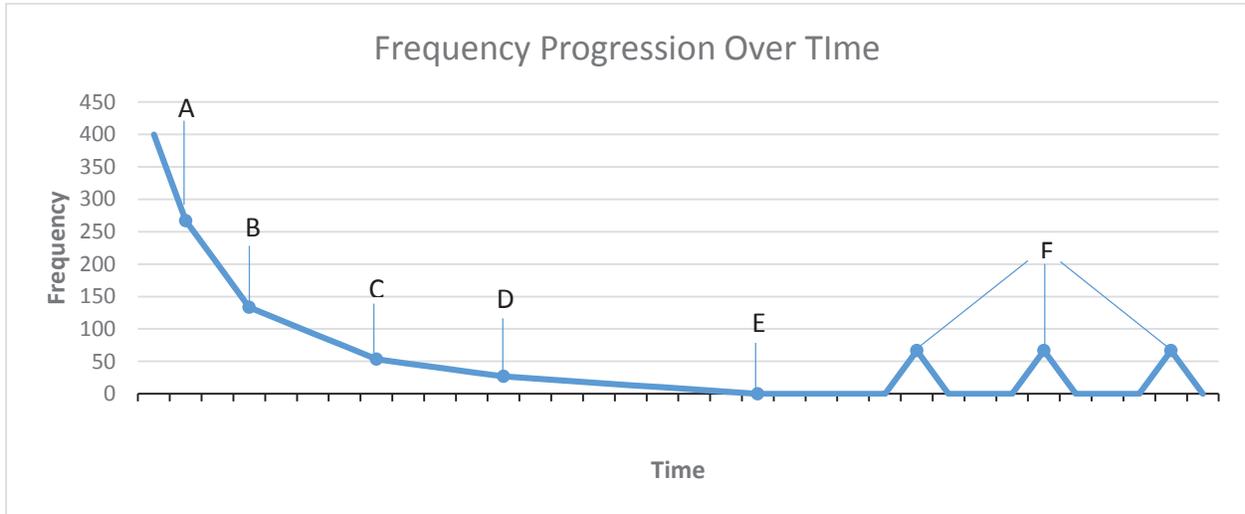


Figure 1: Progression of wave frequency over time. A,B,C,D,E,F refer to trigger points

The objective of my project is to build an RF generator that meets these requirements for manipulation and cooling. The generator must be programmable for different sweeps and different slopes. In addition, the reprogramming should not take longer than $70\mu\text{s}$ and should be determinate, that is, the reprogramming time should never take longer than a set amount – It must be ready to start a new sweep by then. This is because each sweep will be triggered by a centralized timing system. This timing and reliability is crucial and is what caused previous attempts at this project to fail.

The basic design for my generator is as shown below in figure 2.

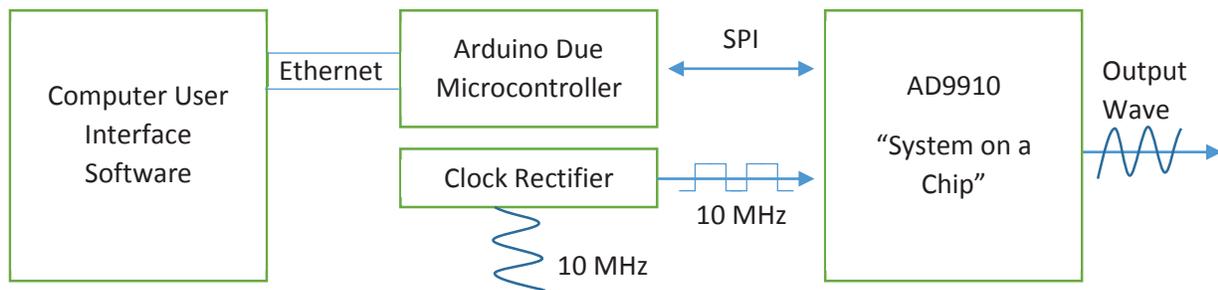


Figure 2: Block diagram of basic hardware and important communication interconnects. SPI is a basic serial communication protocol used by many chips. The AD9910 is a programmable microchip that outputs a frequency that it has been programmed to output. The Arduino Due is an open-source ARM-based microcontroller that is programmed by compiled C/C++ code.

A microcontroller, the Arduino Due, will program and operate the AD9910 chip. This will involve sending commands and information over SPI to program values into the registers on the AD9910. The AD9910 chip actually outputs the wave and its sample clock is driven by a 10 MHz square wave that has been rectified from an analog 10 MHz source. The Arduino will interact with users through a web-based user interface run on a computer which will communicate with the Arduino over an Ethernet network using the TCP/IP protocol. The software will be the front-end with which the user can program the source with the wanted output sweeps.

In summation, the frequency generator, when completed, will be able to set up sweeps quickly and accurately in preparation to be triggered by the centralized timing system; the source need not worry about timing the sweeps, just programming them in time for the trigger. The source will be programmable over an Ethernet network from a host computer. This system is very similar to the first attempt at this project, but seeks to address all the timing and reliability issues that plagued the first. Due to the Arduino's hardware interrupts, hardware SPI (Serial Peripheral Interface), and hardware Ethernet, no time will be wasted polling and programming times will be

fast. In addition, the DDS will be programmed during a sweep, with the central timing system triggering the DDS directly. This will eliminate much if not all of the time delay between sweeps.

II. Hardware

Ordinarily, creating a time-varying, hyper-accurate, Megahertz-generating wave would be quite difficult. This is made much easier with the AD9910 Direct Digital Synthesizer (DDS) system-on-a-chip. In addition to supporting a sample rate of 1GHz for megahertz wave generation, the AD9910 is programmable such that it can perform frequency (as well as phase and amplitude) sweeps by itself based on values programmed into its registers. This feature is called the **Digital Ramp Generator (DRG)**.

The DRG is a crucial part of the chip's usefulness in this task. The AD9910, at its core, takes a **Frequency Tuning Word (FTW)**, which is an unsigned 32-bit value, converts it to a frequency, and makes a smooth sinusoidal wave at this frequency. When fed a different word, it smoothly transitions to the next frequency without any breaks in phase. As the chip can accept many of these values in rapid succession, Analog Devices has included a number of features on the chip that aid in outputting these FTWs, amongst them the DRG. The DRG takes a high value, low value, word step value, and time step value, and will send FTWs following these parameters to the AD9910 output when directed to by external pins. All of the aforementioned values are stored in several different registers within the AD9910. Thus all control of the DRG and the DDS is performed through writing values to onboard registers.

The registers onboard the AD9910 are programmed via the Serial Peripheral Interface (SPI) standard, an extremely basic method of data transmission. SPI only really requires a minimum of two transmission lines to operate if there are only two devices on the bus. One line carries the highs and lows corresponding to the transmitted bits, the other a device-generated clock signal to

go along with the bits on the other line. Based on configuration, the bit line is sampled on either the rising or falling edge of the clock line. This form of communication is simple, easy to troubleshoot, and decently fast (The DDS supports bit rates of up to 70 Mb/s).

The DDS was programmed in previous iterations of this project by a Rabbit Microcontroller. I have elected to use an Arduino Due for the task. The Arduino Due is a 32-bit, ARM-based (the same low-power architecture used by modern cell phones) microcontroller, and the newest addition to the Arduino family. Arduinos have been extremely popular with hobbyist crowds and scientists alike due to their price, open-source libraries, ease of use, reliability, plethora of configurable pins, and ability to easily expand into regions that previously needed specialized microcontrollers. While the Arduino Due is capable of many things, I primarily use it for controlling/monitoring the input/output (I/O) pins connected to the DDS, as well as programming it with commands received over a local network. I use the Due's open-source SPI and Ethernet libraries respectively for these tasks. The Due also operates an LCD display and control pad for displaying and inputting generation information respectively. These different

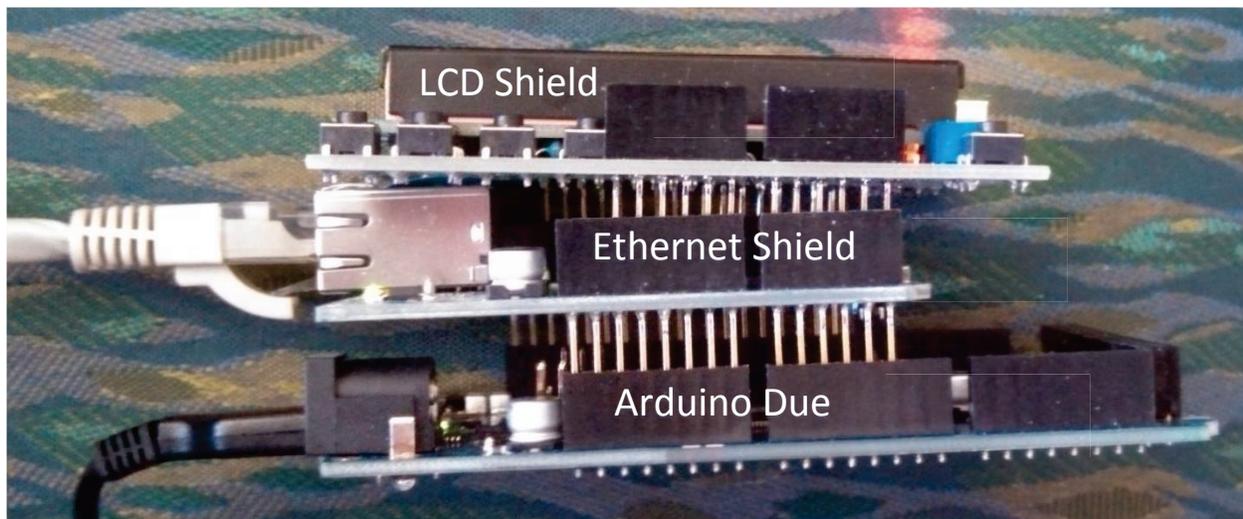


Figure 3: The Arduino with the Ethernet shield stacked on top and the LCD shield on top of it.

capabilities are on what are known as Arduino “shields” and simply stack on top of each other

and the Arduino. As the Arduino project is open source, there is no “one” manufacturer. A downside to this is that sometimes things don’t stack as neatly as possible. Also, the LCD display was not compatible with the SD card reader on the Ethernet shield. Both wanted to use pin 4, so I simply cut off pin 4 from the LCD shield and rerouted it to pin 3.

Due to the speed of the Arduino, there is no needed buffer, shift registers, or any other complications in communication hardware. In addition to the DDS and the Arduino, there is a simple circuit to rectify an external clock signal that serve as the driving system clock of the DDS. All connections between boards/chips will be made as noiseless as possible to ensure high accuracy of the output waveform. Additionally, all communications to the outside world will be opto-coupled in order to prevent ground loops from affecting the output. As Ethernet is the only connection of the device to the outside world, the Ethernet cable will be opto-coupled out of the device.

III. Firmware Theory

Although it is somewhat difficult to separate the terms firmware from software in this project, my definition of firmware is the code that runs on the Arduino whereas “software” refers to code running on the computer communicating with the Arduino. Excellent firmware is vital to the success of this project. Firmware is far harder to debug than software and cannot have room for error as it is the joiner of software to hardware and doing something wrong could conceivably destroy the hardware. It is with this in mind that I spent much of my time ensuring correct coding of the firmware for the Arduino. I elected to take extra time to create and use classes/objects in C/C++ in order to make a very clear and useful **Application Programming Interface (API)**. With

a good API, I, and anyone else who may work on reprogramming/repurposing the software, can interact with the hardware of the DDS without having to deal with problem-creating serial communications.

An example of how the API can greatly help development: I want to flip a bit that turns on the “No-Dwell-High” feature of the Digital Ramp Generator (DRG) circuit in the DDS. This feature makes the DRG begin ramping down once it has finished ramping up. In order to do this simple operation I have to:

1. Lookup the hardware address of the register corresponding to the DRG
2. Read the values of all the bytes in the register
3. Flip the correct bit in the sequence of bytes that corresponds to No-Dwell-High
4. Rewrite the bytes to the correct register

This can encapsulate on the order of 100 lines of basic I/O operations. With my API however, the DRG class is initialized at the beginning with all the necessary information so that instead of the 100 lines in a row, a simple call of `DRG.setNoDwellHigh(true)` is all that's necessary. My DRG class is actually a higher level abstraction utilizing my `Register` class.

```
class DRG{
    public:
        DRG(Register &CFR1, Register &CFR2,
            Register &DRL, Register &DRS,
            Register &DRR, int DROVER,
            int DRCTL, int DRHOLD);
        void setDigitalRampDestination(byte value);
        void setDigitalRampEnable(boolean value);
        void commitCFR2();
        void setUpperLimit(unsigned int tuningWord);
        void setLowerLimit(unsigned int tuningWord);
        void commitLimits();
        void setNoDwellHigh(boolean value);
        void setNoDwellLow(boolean value);
        void setNegativeSlope(short value);
        void setPositiveSlope(short value);
        void commitSlopes();
        void setPositiveStepRate(unsigned int step);
        void setNegativeStepRate(unsigned int step);
        void commitStepRates();
        void commit();
};
```

Figure 4: A snippet of the public functions of the DRG API

While creating a thorough API takes more of an initial time commitment over simply brute-forcing the lines needed to do a specific task, it begins to build upon itself and allows much more flexibility in the functionality of the device.

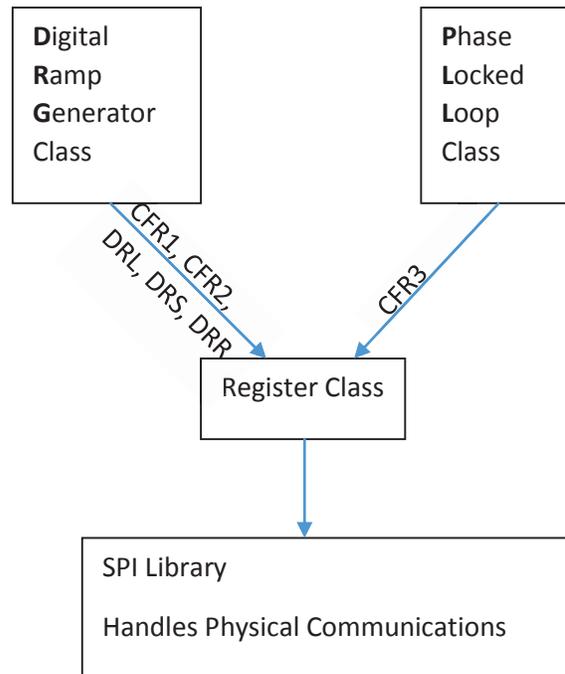


Figure 5 A simple diagram of how a higher level class uses a lower level class in my implementation of the device firmware. CFR1, CFR2, CFR3, DRL, DRS, and DRR are all separate registers on the device. Thus in the code they are different instances of the same Register class

My classes allow for a frequency sweep to be set up in a few lines of code and then executed.

Figure 5 shows a relatively simple representation of how my different classes interact with each other and with the DDS (via SPI communication). Higher level classes can always be created to utilize these lower ones. These classes are crucial to getting advanced operations done easily and efficiently.

The current revision of the firmware has several classes:

- Register – Controls I/O operations with registers on the DDS. The basis of other classes
- PLL – Controls the PLL operations of the AD9910. Sets multiplier, clock speed, etc.
- DRG – Controls the Digital Ramp Generator circuit on the AD9910. Can set and unset bits and values needed to make sweeps
- Profile – Controls single frequency operations of the DDS. Can output a constant frequency sine wave if the RAM stall is not in effect.
- Ramp – Contains all the values needed to perform a single frequency sweep.
- Ramper – Takes a Ramp or an array of Ramps and sweeps through the frequencies.
- RAM – Class that handles writing to, and enabling the RAM.

IV. Device Development

IV.I Clock Rectifier

Regardless of how/if it has been programmed, the AD9910 cannot produce a frequency-accurate waveform without a clock. The AD9910's maximum sample rate is 1 Billion Samples per second (1GSps). As the DDS needs to be able to output frequencies in the hundreds of MHz, it is optimal to use the highest sample rate we can. Operating at this sample rate however, would require a clean accurate clock signal of 1 Gigahertz. 1 Gigahertz is a rather high clock rate to get out of an oscillator and then pipe to another chip. This method would introduce many other design issues as well. Analog Devices solve this problem by including a **Phase Locked Loop** (PLL) system on the AD9910. The PLL can take a clean and accurate external clock signal and multiply it up by a factor programmed in a register of the AD9910. So it can take a very accurate 10 MHz clock, a factor of 100, and output a 1 GHz clock. It does this by syncing its internal high-frequency, high precision (but not accurate frequency-wise) clock to the 10 MHz one

passed through a feedback loop. This stabilizes the internal clock and results in a highly precise and accurate 1 GHz clock. This PLL circuit had already been set up by Justin Winkler [1], so all that was needed was the clock.

The AD9910 needs a square wave input to use in the PLL. As any oscillator outputs a signal as an analog sine wave, it is necessary to rectify it into a square wave. The rectifier was built using opto-couplers in order to also isolate the external clock from the rest of the device. The opto-

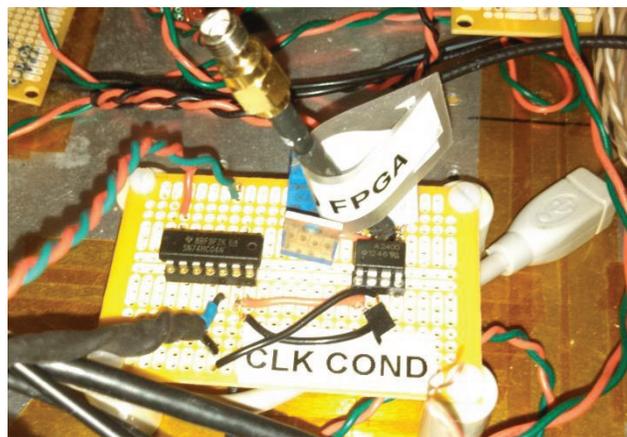


Figure 6: The clock rectifier. The "to FPGA" label is on the output of the rectifier (although the FPGA is no longer in use)

couplers output a “High” or “Low” depending on what the voltage is on the input as compared to a reference pin. So in addition to some capacitors, this works quite nicely as a rectifier system given that the reference voltage is selected correctly. High-speed opto-couplers were used to ensure that the clock signal would not get muddled.

IV.II Arduino Firmware

As the main point of this project is to perform frequency sweeps, I created several classes that deal with components related to sweeping. First and foremost is the PLL class. The PLL class sets up the phase-locked loop values on the AD9910 so that the chip can have a 1GHz sample

clock. In addition, all values for frequency tuning words will depend on the device's sample clock, so the PLL class has a static member exposing the current clock of the DDS.

After the PLL circuit has been set up and a clock established, the DRG class can come into play. As explained earlier, the DRG class controls the Digital Ramp Generator circuit of the AD9910. The DRG will perform a frequency sweep that it has been programmed with. The DRG class handles programming the two frequency limits that the DRG will sweep between as well as the time step and word step. In addition, it also controls setting and unsetting the other various bits and pins that can control the DRG. The DRG class does not perform sweeps by itself, but controls all the functions needed to perform a sweep. As each sweep will have different parameters that need to be programmed into the DRG, I created the Ramp class.

The Ramp class is a simple class that contains all the data necessary to perform a sweep. Start point, end point, step rate, direction, etc. are all contained in one Ramp instance. An instance of the Ramper class can take a Ramp and act on an instance of the DRG class to perform it. Why do we need the Ramper if we have a Ramp and a DRG class already? While the DRG class contains all the material needed to control sweep generation, there needs to be an object that operates it in much the same way that a car cannot go anywhere without a driver. So the Ramper (our driver) accepts a Ramp or an array of Ramps (our destination(s)), and then sweeps our frequency through them using the DRG (drives us there). The Ramper also makes use of a RAM class created in order to help solve "the glitch."

IV.III The Glitch

“The glitch” is a problem which Justin Winkler originally ran into when he was first designing his DDS controller. It has to do with the Digital Ramp Generator (DRG) on the actual AD9910 chip. More specifically, the problem has to do with handling the DRG at the end of one sweep before beginning the next. When sweeping **UP** through frequency, the DRG has no problems. When it hits the top value, it stops and awaits reprogramming. By simply programming a higher level into the register, the DRG will continue sweeping up until it hits the limit again. This is exactly the kind of behavior we were looking for except we wanted to sweep **DOWN**. Although the documentation [2] suggests that downward sweeps will work the same as upward sweeps, the actual chip does not. It will not continue to sweep downward again after a sweep if the lower

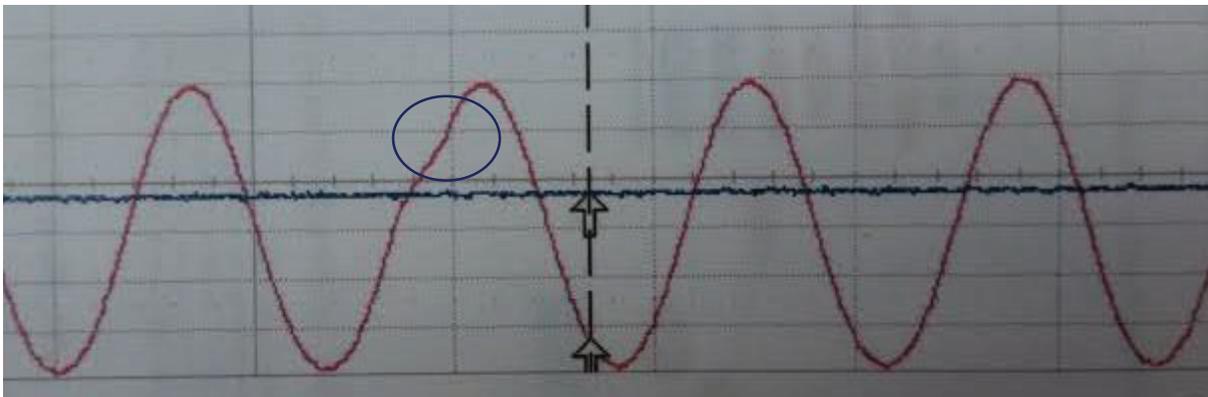


Figure 7: The glitch in one of John Hagee's wave captures

limit is reprogrammed. It has no problem sweeping up again; in fact it has no problem sweeping down again **AFTER** it has swept up. This creates the problem of a small increase in frequency before it can decrease again. After finding this issue, John Hagee tried many things including calling Analog Devices but in the end was only able to make it barely noticeable but not gone. Why do we care so much about a small glitch? While it's a pretty minute glitch, the atoms, which will be manipulated with the frequencies, tend to be picky when it comes to these sort of things.

I came up with a way to do away with the glitch by using a different feature of the AD9910. The main functioning part of the AD9910 is that it turns Frequency Tuning Words (FTWs) into an actual wave. There are multiple ways to feed the AD9910 these words both from off the chip and on the chip. For example, the DRG sends out FTWs at the right time with the right gap between them to produce a frequency sweep. One could also have the parallel port on the chip enabled and also be feeding in a FTW at the same time as the DRG. This would ordinarily create a race condition of some sort except that the AD9910 has a built in priority list: if two sources are enabled, the generation will follow whichever is highest in the list.

Table 5. Data Source Priority

Priority	DDS Signal Control Parameters					
	Frequency		Phase		Amplitude	
	Data Source	Conditions	Data Source	Conditions	Data Source	Conditions
Highest Priority	RAM	RAM enabled and data destination is frequency	RAM	RAM enabled and data destination is phase or polar	OSK generator	OSK enabled (auto mode)
	DRG	DRG enabled and data destination is frequency	DRG	DRG enabled and data destination is phase	ASF register	OSK enabled (manual mode)
	Parallel data port and FTW register	Parallel data port enabled and data destination is frequency	Parallel data port	Parallel data port enabled and data destination is phase	RAM	RAM enabled and data destination is amplitude or polar
	FTW register	RAM enabled and data destination is phase, amplitude, or polar	Parallel data port concatenated with the POW register LSBs	Parallel data port enabled and data destination is polar	DRG	DRG enabled and data destination is amplitude
	FTW in active single tone profile register	DRG enabled and data destination is phase or amplitude	POW register	RAM enabled and destination is frequency or amplitude	Parallel data port	Parallel data port enabled and data destination is amplitude
	FTW in active single tone profile register	Parallel data port enabled and data destination is phase, amplitude, or polar	POW in active single tone profile register	DRG enabled and data destination is frequency or amplitude	Parallel data port concatenated with the ASF register LSBs	Parallel data port enabled and data destination is polar
	FTW in active single tone profile register	None	POW in active single tone profile register	Parallel data port enabled and data destination is frequency or amplitude	ASF in active single tone profile register	Enable amplitude scale from single tone profiles bit (CFR2[24]) set
Lowest Priority			POW in active single tone profile register	None	No amplitude scaling	None

Figure 8: The table of tuning word priority for the DDS. Of note is that the RAM is higher than the DRG in the frequency column

The DRG is the next to highest on the priority list and the parallel port is behind it, so in the example above, the AD9910 would act as though it was not even receiving the FTWs that were being sent in on the parallel line.

Luckily, there is one method of FTW feeding that has a higher priority than the DRG which is the onboard RAM. The AD9910 is equipped with RAM that can hold 1024 FTWs for playback. If the RAM is enabled, the FTW(s) played back from the RAM will take precedence over the FTWs being put out by the DRG. Thus, by turning on the RAM while I induce the glitch in the DRG and then turning it off again, I can make it look like the DRG just continued on sweeping rather than sweeping up and then down. It appears to have worked. While finding the glitch was never easy as it was quite small. I have sent out triggers to look precisely at where the issue should be and see no traces of the glitch.

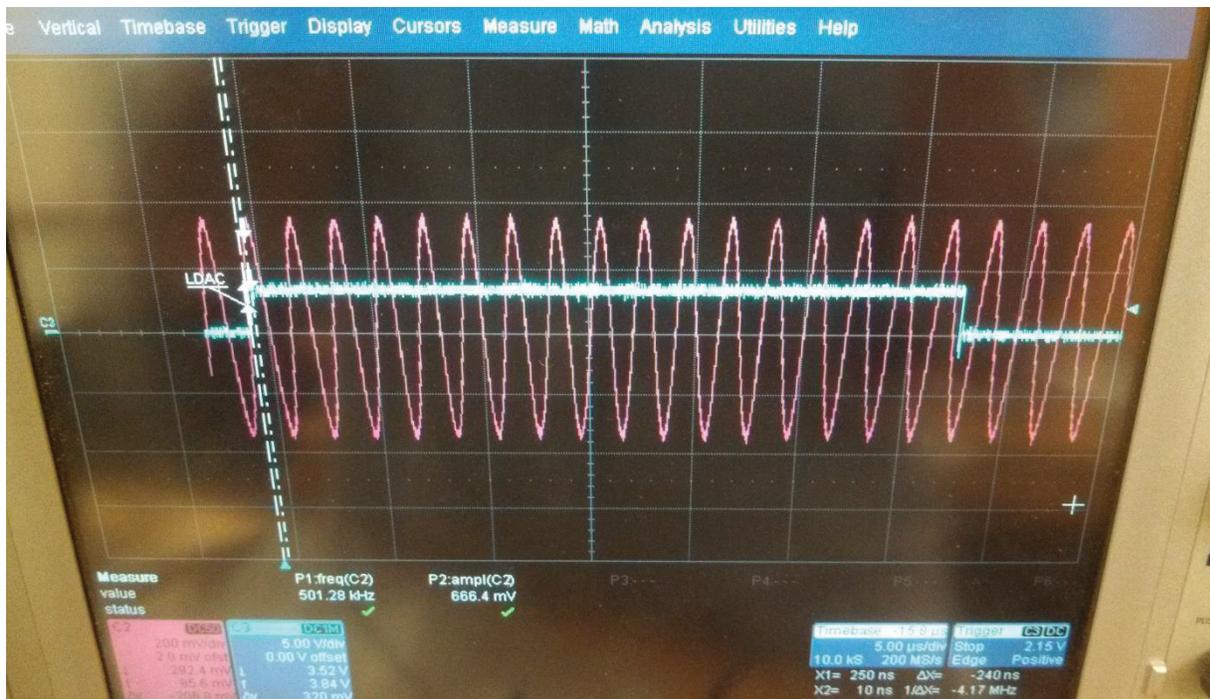


Figure 9: A scope trace of a frequency sweep. The blue line level changes represent when the first sweep has ended and when the second has begun. There is no noticeable glitch in the waveform

Device Boot-Up Process

The first thing the Arduino does upon booting is to open its serial port (it does this regardless of whether a serial device is present on the line). The serial port is used to log information that is occurring internally. This is done so that, should there ever be recurring issues during setup, a serial device can be connected to the port and set to 9600 baud to listen in. After the port has been started, the Arduino resets the AD9910 to default values via the RESET pin and then attempts to establish communication via SPI with the AD9910. As the Arduino prefers 4-wire SPI to the 3-wire SPI the AD9910 defaults to using, the first command sent to the AD9910 has to be to change to 4-wire mode. After this command is sent and the I/O_UPDATE pin has been triggered, the Arduino establishes that 4-wire SPI has been enabled by attempting to read the values from a register. If these values match what are expected to be the default values, then the setup continues, if it fails, it will retry resetting and establishing communications two more times before failing and halting setup. The setup continues on to setup the Phase Locked Loop (PLL). The PLL class is initialized and reads in all register values. Next, the PLL is setup using the clock parameters located in the file on the microSD card inserted in the Arduino. If the card is not found, the value 10MHz ->1000MHz hardcoded into the firmware is used. The PLL_LOCK pin is checked and setup stops if things did not work out. If all is well, an interrupt is attached to the PLL_LOCK pin that will halt all Arduino programming if the lock is lost and print a message to serial (this would occur if the clock feeding the DDS somehow went wrong). After the PLL lock is established, the firmware creates the DRG object, then the RAM object, and finally a Ramper object. The main setup is then complete.

After the main setup is complete, the Arduino moves on to set up network connections. It connects to the Local Area Network (LAN) and attempts to lease an IP address via DHCP (if there is no static IP specified in “CONFIG.TXT” on the microSD card). Once it has its network address, the Arduino displays “Waiting at <IP ADDRESS>” on its LCD and begins listening on TCP port 80 for commands. It is now fully booted and ready to use.

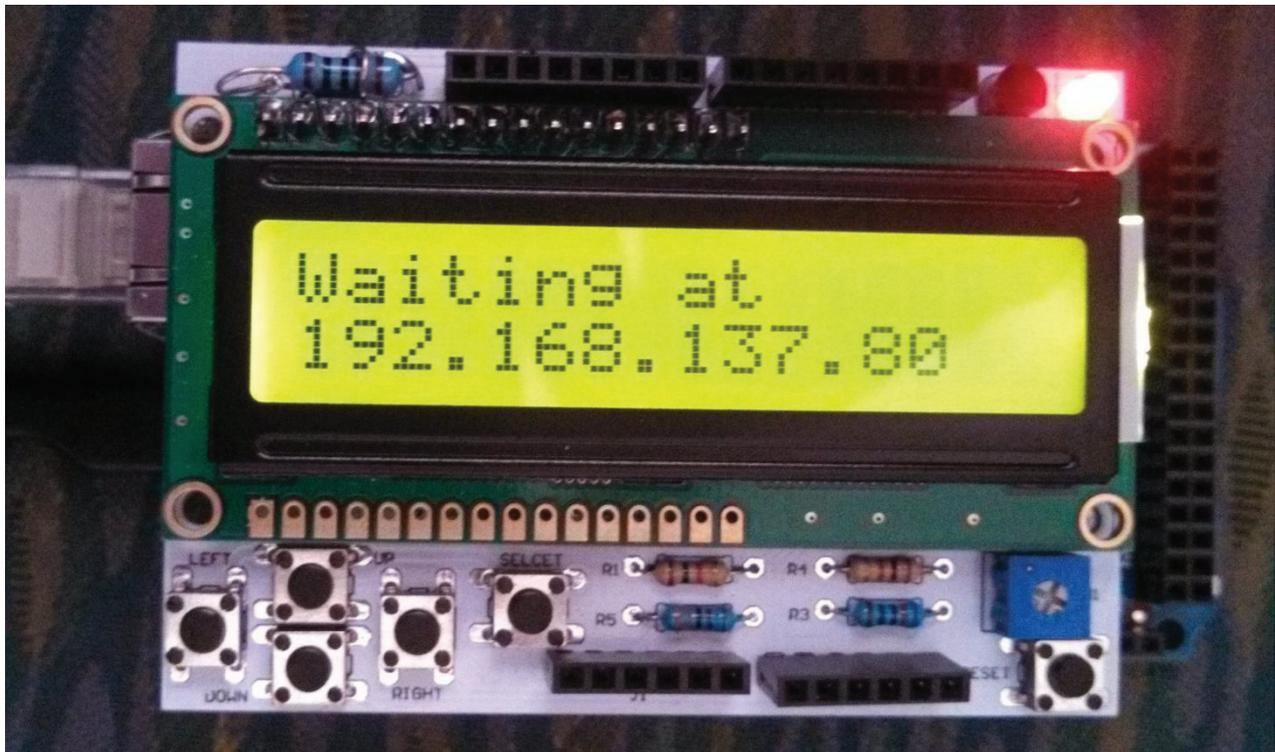


Figure 10: The DDS after it has finished booting and is ready to receive commands

V. Communication with the Arduino

V.I Choosing a Connection Type

Once the Arduino has been programmed such that it can program the DDS to perform sweeps, it becomes necessary to enable it to communicate with other devices so that it can be told what to do. The most obvious choice for communicating with the Arduino is its native serial port

(accessed through the Arduino's USB connection). The serial port has the advantage of speed and simplicity. The main disadvantage of using a serial port, is that you need a device connected to the Arduino via USB to use it. This means that there would need to be a computer controlling the Arduino controlling the DDS. This of course now ties all of the Arduino's reliability and availability to the reliability and availability of the computer it is connected to. While still not necessarily a bad idea, the core design pillar of reliability made me shy away from this concept.

In addition to the reliability issues using another computer would introduce, this scheme would tie the DDS system a certain operating system (Windows, Linux, Mac). The computer that talked to the Arduino would have most likely been running Windows XP, which at the time of writing, is 13 years old and has just lost official support from Microsoft. Not building something for an outdated platform is always a good idea when you have the ability to prevent it. While using a higher-level language that is platform-agnostic (Python, Java, etc.) would solve this issue, I still wanted the device to be as independent as possible, which flies directly in the face of serial control.

Instead of serial, I decided to go with Ethernet as the primary physical layer of communication with the Arduino. Ethernet allows any computer connected to the local network to communicate with the Arduino; similar to a virtual USB cable to every computer in the lab. This solves the issue of having a dedicated computer. As for the software layer, I decided on a protocol that would be universally supported by all major operating systems: **HyperText Transfer Protocol** (HTTP). HTTP (and its secure version: HTTPS) are the universal language of the web. Using HTTP, the control program for the Arduino is a website loaded in a browser and the Arduino is a webserver. This allows any device that has a web-browser to communicate with the Arduino.

V.II Arduino Web Server

The Arduino is programmed as a very basic web server. As soon as it has booted and acquired an IP address via DHCP, the Arduino will display its address on the LCD screen and begin listening for connections on port 80 (the standard port for the HTTP protocol). When the DDS receives a connection it will read in the HTTP request sent along. If it cannot locate the “Command” variable in the request, it will proceed to serve the webpage to the device. If the Arduino does find the “Command=” variable, it will read the command, and perform the action assigned to it. Should any data be needed from the sender to process the command, the Arduino will look for a “PL=” somewhere in the body of the request and interpret what comes directly afterward as the data (PayLoad). Below is an example of an HTTP header sent to save a ramp to the DDS:

```
POST http://192.168.137.45/?Command=sr1: HTTP/1.1
Host: 192.168.137.45
Proxy-Connection: keep-alive
Content-Length: 81
Accept: application/json, text/javascript, */*; q=0.01
Origin: null
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/33.0.1750.154 Safari/537.36
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8

PL=[{"n":1,"h":2147483,"l":858993,"t":58,"w":1,"H":0,"L":0,"D":0,"d":0,"term":0}]
```

The parts that the DDS cares about have been bolded. First the DDS looks for “Command=”, it then interprets what follows as a command (“sr1” tells it to load 1 sweep). As this command requires data, the Arduino then scans the rest of the request to find the “PL=”; what follows is the data needed to describe each sweep. When the DDS has finished processing the command, it sends out a response that varies depending on the command. In the case of our command above, it returns:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: null
Connection: close
Content-Type: application/json
{ "transmission": "Okay", "info" : { "clock": 1000000000, "status": "Idle", "mode" : "sf" } }
```

The response is encoded in **JavaScript Object Notation (JSON)**. JSON is widely used by web browsers and servers to communicate not because it is the most efficient but because it is easy for both people and machines to read; thus, JSON is not hard to understand:

- A JSON object is everything that is between two braces {JSON object}
- A string is surrounded by quotes, “string”, a number is not, 1
- A JSON object consists of key:value pairs separated by commas {“a” : 1 , ”b”: “spam”}
- A list/array is designated by brackets [1 , 2 , “item3”]

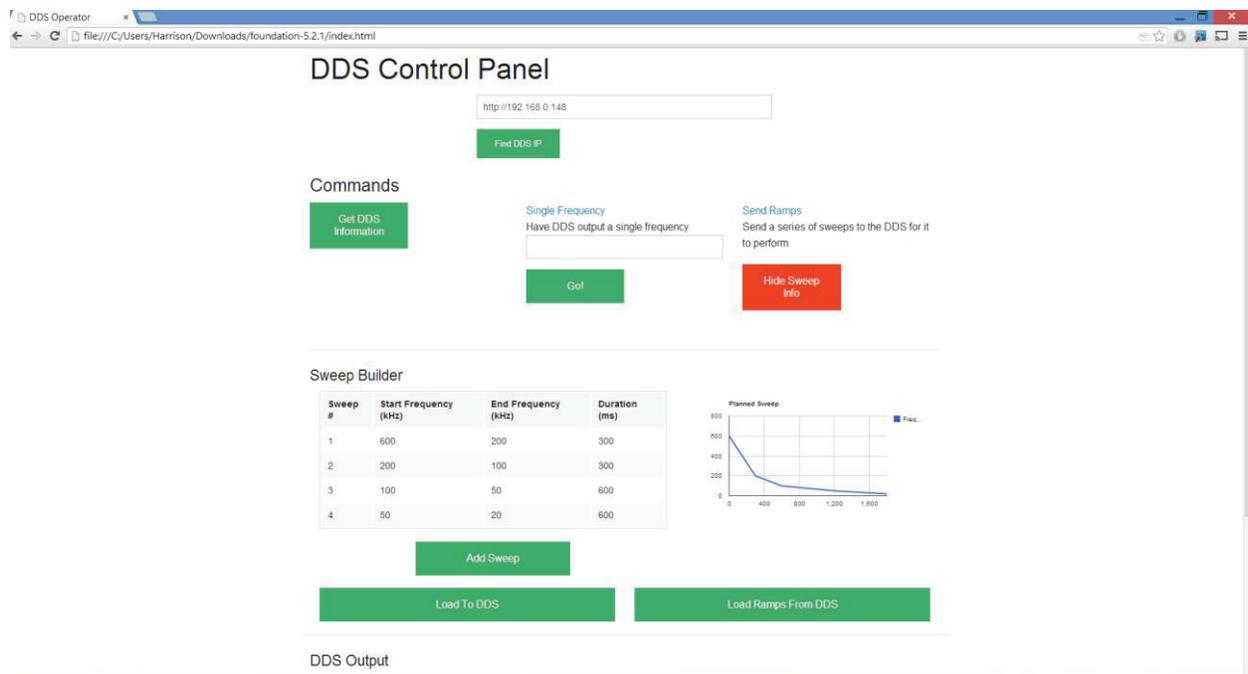
So from the returned data, we can see that the key **"info"** has another JSON object in it which contains the clock, status, and mode of the DDS.

By making the DDS controllable over Ethernet in this fashion, I have in effect, created a Web API for operating the DDS remotely. Anyone who wishes, can write their own website front-end to communicate with it. In fact, as the Arduino just searches for “Command=” and then optionally for a payload, it does not technically need to receive a message in the form of the HTTP protocol. Any software that opens a TCP connection on port 80 can then send a command (“Command=” [command]) and optionally a payload (“PL=”[payload]). The DDS will then process the command and return the result formatted in JSON (with HTTP headers). This means that it is not very difficult to integrate the DDS with other software such as LabWindows or LabView. In a nutshell then, the DDS is network controlled via a TCP socket on port 80 and is HTTP compatible.

[The final paper will have an appendix with the web API (all commands/syntax and return data)]

V.III Front-End Software

Due to the fact that the DDS can communicate using the HTTP protocol, the front-end software I elected to make is actually a website. Like any other modern website it uses HTML for the structure and JavaScript for code operations. The webpage can be loaded either from a flash-drive or folder, or by navigating to the Arduino's network address. Loading the webpage from a local drive will be faster as the Arduino does not have to serve the files (total file size is currently 257 kB which takes roughly 17 seconds). If the client web browser is decompression-enabled (all modern browsers), it can serve the compressed 48.7 kB compressed version in approximately 2.5 seconds.



The webpage provides a sleek and lightweight GUI for programming the DDS. All the setup required is for the user to either input the DDS's IP address or find it using the "Find DDS IP" button. After the DDS's IP address has been set correctly, by clicking on the various buttons, the

user can send commands to the Arduino to operate the DDS. The webpage's features currently include:

- Finding the IP address of the DDS (if unknown)
- Getting DDS state and information
- Loading sweeps from the DDS
- Creation of a series sweep
- A graph to indicate what the currently tabulated sweep will look like
- Loading sweeps into the DDS
- Single frequency output
- A “console” to more directly view DDS responses

The webpage communicates with the DDS via **Asynchronous JavaScript And XML (AJAX)** - requests. An AJAX request is performed as though the web browser loaded another webpage in the background, read it, and returned the information. Every time a website updates information without you having to navigate to a new URL, an AJAX request has occurred. So when the webpage sends a command to the Arduino, it will fire off an AJAX GET request with the command in it, say “GET http://192.168.0.148/?Command=i” (“i” is the command for retrieving DDS state and information). The Arduino will process the command and send the JSON-encoded response. Upon receiving the response, the webpage will pass the data along to a function that will print it to the console and decide what to do with it, if anything.

VI. Future

There are many more features of the AD9910 that I did not get a chance to expose. The AD9910 can perform sweeps through amplitude and phase in addition to frequency. There are also several different programmable RAM modes, a parallel port, multiple switchable RAM and/or single-

tone profiles, and more. With not too much work, my current firmware could expand into these regions using the existing Register class if there were any need for them.

As mentioned in the section above, my front-end performs the basics but can be replaced by any different front end utilizing the web API. If features beyond those offered in the web API are needed, the Arduino firmware will actually have to be edited. Using the web API alone, one could easily code control of the DDS into LabWindows or any other controlling software.

VII. Conclusion

I set out to design a controller for the AD9910 that would allow the programming and output of single frequency signals as well as rapidly chained linear sweeps through frequency. In addition, it was to be remotely operable, remotely triggered, and reliable. The DDS has been completed and can do all of these things. While reliability can only really be decided by time, the parts of the firmware driven by the Ethernet communications (the core reads and writes to the DDS) have been working for a while, so I predict good reliability.

I did not get a chance to test the triggering mechanisms with the Adwin system that will eventually control them. During my testing, I simply wired up the device's trigger pin to another pin on the Arduino so that it could trigger itself when the sweep was complete, simulating the Adwin system. In addition, the other Adwin-controlled trigger, the reset from sweep pin, have not been tested.

Other Thoughts

On the AD9910: The chip is a very good at the wave production part. That being said, the documentation, while not by any means poor, is incorrect sometimes, and unclear about performance at its limits. This coupled with the complete lack of feedback from the chip (the only thing it will ever tell you is what register values are when you send a read command) make it difficult to develop firmware for. The chip could also benefit from some non-volatile memory and/or extra volatile memory (RAM profiles and Single-Tone profiles occupy the same register. You can only have one or the other). Seeing as they have not updated the chip in 5 years, it's unlikely to ever see these changes.

References

- [1] J. Winkler, "Agile Radio Frequency Source for Hyperfine Manipulation of Ultra-Cold Atoms," The College of William & Mary, Williamsburg, 2009.
- [2] Analog Devices, "AD9910: 1 GSPS 14-bit, 3.3 V CMOS Direct Digital Synthesizer Data Sheet (Rev. D)," Analog Devices, Norwood, 2012.

APPENDIX A: Web API

Command: i

Definition: Gets clock and state information from DDS

Arguments: None

Response: `MainResponse.info = {"clock" = clockValue , "status" = statusValue}`

Sample Command String: `GET /?Command=i HTTP/1.1`

Sample Response:

```
{ "transmission":"Okay", "info" :  
  {"clock":1000000000,"status":"Idle", "mode" : "sf"}  
}
```

Command: sr#: (Where # is the number of ramps in string form)

Definition: Load a compound sweep into the DDS

Arguments: Payload is an array of JSON objects, each containing a string with the variables (IN THIS ORDER):

n: Sweep# [1-indexed]

h: High Word

l: Low Word

t: Time Step

w: Word Step

H: No Dwell High [Enabled = 1 , Disabled = 0]

L: No Dwell Low [Enabled = 1 , Disabled = 0]

D: Word Destination [Frequency = 0]

d: Direction [Up=1, Down=0]

f: 1 – Filler Value to help make reading values less complicated on the DDS end

Response: See “Command=sl”.

Sample Command String: `GET /?Command=sr2: HTTP/1.1`

```
PL=[{"n": 1, "h": 4000, "l": 2000, "t": 16, "w": 54, "H": 0, "L": 0,  
"D": 0, "d": 0, "f": 1}, {"n":2, "h": 2000, "l": 1000, "t": 16, "w":54,  
"H":0, "L":0, "D":0, "d":0, "f":1}]
```

Sample Response String: see “Command=sl”

Command: sl:

Definition: Respond with sweep(s) that are currently loaded into the DDS. Is called directly after a load (sr) operation.

Arguments: None

Response: MainResponse.ramps = [{Sweep 1}, {Sweep 2}, {Sweep 3}, ...]

The following samples assume the load (sr) operation of the load example has been executed prior

Sample Command String: GET /?Command=sl HTTP/1.1

Sample Response String :

```
{“transmission”: “Okay”, “ramps”:  
  [{"l": 2000 , “h”: 4000, “dir”: 0 }, {“l”: 1000 , “h”: 2000, “dir”: 0 }]  
}
```

Command: cw

Definition: Produce a sine wave at the desired frequency tuning word (FTW)

Arguments: Payload is a JSON object with the following values:

w: FTW

f: 1 – Filer value to help DDS read JSON objects quickly

Response: MainResponse.singleTone = FTW

Sample Command String: GET /?Command=cw HTTP/1.1

PL={“w”: 54000 , “f”: 1}

Sample Response String:

```
{“transmission”: “Okay”, “singleTone”:54000}
```

Command: cf

Definition: Produce a sine wave at the desired frequency

Arguments: Payload is a JSON object with the following values:

f: Frequency in Hertz

d: 1 – Filler value to help DDS read JSON objects quickly

Response: `MainResponse.singleTone = FTW`

Sample Command String `GET /?Command=cf HTTP/1.1`

Sample Response String:

```
{“transmission”: “Okay”, “singleTone”:54000}
```

Command `r`:

Definition: Reset the Arduino. Equivalent to sending a rising edge to the reset pin

Arguments: None

Response: Standard Response

Sample Command String `GET /?Command=r HTTP/1.1`

Sample Response String:

```
{“transmission”: “Okay”}
```

APPENDIX B: Source Code

On next page (in order of appearance):

DDS_Operator_MKVI (Main Script)

PinDefinitions.h

DRG

Definitions

Ethernet_Handler

LCD_Handler

MemoryFree

PLL

Profile

RAM

Ramp

Ramper

Register

SD_Handler

Setup

```
// MK VI
//ChangeLog
//Bump function added
//Comments added
//Several performance modifications
//-----
//

//Needed for Network Communications
#include <LiquidCrystal.h>
#include <Ethernet.h>
#include "LCD_Handler.h"
#include <SPI.h>
#include "Ramp.h"
#include <SD.h>
#include "SD_Handler.h"
#include "Ethernet_Handler.h"

//Need for DDS
#include "PinDefinitions.h"
#include "Definitions.h"
#include <SPI.h>
#include "Register.h"
#include "PLL.h"
#include "DRG.h"
#include "Profile.h"
//#include "Ramp.h"
#include "Ramper.h"
#include "RAM.h"

#include "Setup.h"

//For Checking Memory Use
#include "MemoryFree.h"

//Globals for DDS
DRG myDRG;
Profile profile0;
RAM myRAM;
Ramper sweeper;

//Ramp Array for storing a multi-part sweep
Ramp rampArray[255];
byte rampArrayLen=0;

//Function that performs a software reset
void Software_Reset() {
  Serial.println("Going down for reset!");
  const int RSTC_KEY = 0xA5;
  RSTC->RSTC_CR = RSTC_CR_KEY(RSTC_KEY) | RSTC_CR_PROCRST | RSTC_CR_PERRST;
  while (true);
}
```

```

void myTest () {

    Ramp myRamp = Ramp(50000, 500000, 10000, 0, 0);

    Ramp r1 = Ramp(5000000, 50000000, 10000, 0, 0);
    Ramp r2 = Ramp(2000000, 5000000, 8000, 0, 0);
    Ramp r3 = Ramp(1000000, 2000000, 8000, 0, 0);

    Ramp myArray[] = {r1,r2,r3};

    profile0.produceFreq(10*1000*1000);

    Serial.println(F("Beginning Chained Sweep Test"));
    Serial.println(F("Begin"));
    //sweeper.chainedSweeps(myArray, 3);
    Serial.println(F("Finished"));

}

//In Arduino, "setup" function runs first, followed by continous running of "loop" function

//Runs once, sets up the Arduino for operation
void setup () {
    //Start the serial port at 9600 baud
    Serial.begin(9600);
    Serial.println(F("ARDUINO BOOTED"));

    Serial.print("Amount of free RAM pre-setup: ");Serial.println(freeMemory());

    //From LCD_Handler; sets up LCD display
    setupLCD();

    //From SD_Handler; sets up SD card
    setupSD();

    //From Ethernet_Handler; sets up Ethernet Host
    //Passed DHCP_ENABLE which is True/False
    setup_Ethernet(DHCP_ENABLE);

    //From Setup; sets up DDS communications and settings
    fullSetup();

    Serial.print("Amount of free RAM post-setup: ");Serial.println(freeMemory());
}

boolean handleSweeps () {
    boolean funcStatus = false;
    switch(client.read()){
        case 'r':
            rampArrayLen = loadRamps( rampArray);

```

```
    case 'l':
        funcStatus = sendOutRamps(rampArrayLen, rampArray);
        break;
    case 'f':
        break;

    break;
}
return funcStatus;
}

boolean handleConstFreq(){
    char c = client.read();
    if(c=='f'){
        memset(smallBuffer,0,10);
        client.readBytesUntil(':',smallBuffer,10);
        int freq = atoi((char*)smallBuffer);
        Serial.print("Producing Single-Mode Frequency: ");Serial.println(freq);
        profile0.produceFreq(freq);
        return true;
    }
    // else if(c=='w'){
    // }
    return false;
}
```

```
char mainCharHandle(char readValue){
    int returnVal = 1;
    boolean commandStatus=false;
    switch(readValue){
        case 'c':
            commandStatus = handleConstFreq();
            break;
        case 's':
            commandStatus = handleSweeps();
            break;
        case 'G':
            sweeper.chainedSweeps(rampArray,rampArrayLen);
            commandStatus=true;
            break;
        case 'i':
            commandStatus=handleInformation();
            break;
        case '!':
            client.println("Stop function");
            break;
        case 'r':
            returnVal = 'r';
            break;
        case 'p':
            commandStatus = sendAddressAcross();
            break;
    }
```

```

    case 't':
        myTest();
        commandStatus=true;
        break;
    default:
        readValue='%';
        //client.println("Command not recognized; Displaying help");
        //showHelp();
}
sendCommand(readValue);
sendCommandStatus(commandStatus);
return returnVal;
}

//Loop function runs continuously
void loop() {
    Serial.print("Amount of free RAM: ");Serial.println(freeMemory());

    //Check if a client is available
    client = server.available();

    //Print IP address to Serial and LCD
    Serial.print(F("Ethernet server established at: ")); Serial.println(Ethernet.localIP());
    myLCDprint("Waiting at",Ethernet.localIP());

    //Block until a client is connected
    while(!client){
        client = server.available();
    }

    //myLCDprint("Client Connected!", "");

    char result;

    if(client.connected()){
        client.setTimeout(10); //Set timeout on client operations to 10 ms

        if(!navigateTo("Command=", 8 , 200)){ //nagateTo function from Ethernet_Handler
            //If there is no command in the incoming request, serve the webpage
            Serial.println(F("No Command found"));

            //Check that that the SD card was initialized correctly(SD card is in)
            if(SDisBegun){
                //Open the compressed webpage: indexc.txt
                Serial.print("Opening File: INDEXC.TXT");
                myFile= SD.open("INDEXC.TXT");

                //If file is opened correctly (it exists)
                if(myFile){
                    //Headers for the webpage
                    client.println("HTTP/1.1 200 OK");
                    client.println("Content-Type:text/html");
                    client.println("Content-Encoding: gzip");
                }
            }
        }
    }
}

```

```
client.println();

//Copy data from file into buffer, and then to write to the client
while(myFile.available() && client.connected()){
  memset(Buffer,0,500);
  myFile.read(Buffer,sizeof(Buffer));
  client.write(Buffer, sizeof(Buffer));
}

//Cleanup after serving page
Serial.println(F("Finished Sending webpage"));
myFile.close();
}
else{
  //If file could not be opened
  sendFailHeader();
  Serial.println(F("Couldn't find file"));
}
}
else{
  //If SD card did was not initialized correctly
  sendFailHeader();
  Serial.println(F("SD card was not initialized correctly"));
}
}
else{
  //If a command was found

  //Read (first) command byte into char variable
  char command = client.read();

  //Send out successful header indicating JSON response
  sendSuccessfulHeader();

  //Dispatch first command byte to function, store result
  result = mainCharHandle(command);

  //End the Response
  endClientTransmission();

}
//Close connection to client
client.stop();

//If reset condition has been made, perform software reset
if(result=='r'){
  Software_Reset();
}
}
client.stop();
}
```

```
//Define Pinout Constants
//LCD Pins
#define LCD1 8
#define LCD2 13
#define LCD3 9
#define LCD4 3
#define LCD5 5
#define LCD7 6
#define LCD8 7

//CS for Ethernet, SD Card
#define ETH_CS 10
#define SD_CS 4

//CS for DDS
#define DDS_CS 52

//Used
#define MASTER_RESET 18 //Goes to DDS pin 14
#define IO_RESET 37 //Goes to DDS pin 71
#define IO_UPDATE 50 //Goes to DDS pin 59
#define DROVER 23 //Goes to DDS pin 61
#define DRCTL 25 //Goes to DDS pin 62
#define DRHOLD 27 //Goes to DDS pin 63
#define PLL_LOCK 17
#define SWEEP_RESET_TRIGGER 15
#define SWEEP_TRIGGER 14
#define FAUX_TRIGGER 16
#define P0 53 //Goes to DDS pin 52
#define P1 51 //Goes to DDS pin 53
#define P2 49 //Goes to DDS pin 54

//Unused
#define EXT_PWR_DWN 2 //Goes to DDS pin 18
#define F0 41 //Goes to DDS pin 49
#define F1 39 //Goes to DDS pin 50
#define TxENABLE 45 //Goes to DDS pin 41
#define OSK 24 //Goes to DDS pin 60
```

```
#ifndef DRG_h
#define DRG_h

#include "Arduino.h"
#include "Register.h"

/*DRG class:
This class contains methods and data for controlling the DRG
part of the AD9910 chip
*/
class DRG{
public:
    DRG();
    DRG(int blah);
    void hold();
    void release();
    void slopeControlLow();
    void slopeControlHigh();
    boolean isAtLimit();
    void setDigitalRampDestination(byte value);
    void setDigitalRampEnable(boolean value);
    void commitCFR2();
    void setUpperLimit(unsigned int tuningWord);
    void setLowerLimit(unsigned int tuningWord);
    void commitLimits();
    void setNoDwellHigh(boolean value);
    void setNoDwellLow(boolean value);
    void setNegativeSlope(unsigned short value);
    void setPositiveSlope(unsigned short value);
    void commitSlopes();
    void setPositiveStepRate(unsigned int step);
    void setNegativeStepRate(unsigned int step);
    void commitStepRates();
    void commit();

private:
    union data4{
        byte b[4];
        unsigned int value;
    };
    union data2{
        byte b[2];
        short value;
    };
    union data4 _upperLimit;
    union data4 _lowerLimit;
    union data4 _negStep;
    union data4 _posStep;
    union data2 _posSlope;
    union data2 _negSlope;
    byte _CFR1[4];
    byte _CFR2[4];
    byte _DRL[8];
};
```

```
byte _DRS[8];
```

```
byte _DRR[4];
```

```
};
```

```
#endif
```

```
#include "DRG.h"
#include "PinDefinitions.h"
#include "Definitions.h"

//Blank Constructor so memory can be set aside before DRG class is actually initialized
DRG::DRG() {}

//Different constructor called when it is actually time to set up DRG
//Blah parameter is to differentiate the constructor; never used
DRG::DRG(int blah) {
    //Read registers needed for DRG
    CFR1.read();
    CFR2.read();
    DIGITAL_RAMP_RATE.read();
    DIGITAL_RAMP_STEP_SIZE.read();
    DIGITAL_RAMP_RATE.read();

    //Read Register data values into DRG private register data reserves
    for(int i=0; i<4;i++){
        _CFR1[i]=CFR1._data[i];
        _CFR2[i]=CFR2._data[i];
        _DRR[i]=DIGITAL_RAMP_RATE._data[i];
    }

    for(int i=0; i<8;i++){
        _DRL[i]=DIGITAL_RAMP_LIMIT._data[i];
        _DRS[i]=DIGITAL_RAMP_STEP_SIZE._data[i];
    }
}

//Send Digital Ramp Direction Control pin Low (Downward sweep)
void DRG::slopeControlLow() {
    digitalWrite(DRCTL, LOW);
}

//Send Digital Ramp Direction Control pin High (Upward sweep)
void DRG::slopeControlHigh() {
    digitalWrite(DRCTL, HIGH);
}

//Pause current DRG sweep
void DRG::hold() {
    digitalWrite(DRHOLD, HIGH);
}

//Continue current DRG sweep
void DRG::release() {
    digitalWrite(DRHOLD, LOW);
}

//Return true if DRG has reached bottom or upper limit
boolean DRG::isAtLimit() {
    return digitalReadDirect(DROVER);
}
```

```
}

//Set Digital Ramp Destination Bit to value
void DRG::setDigitalRampDestination(byte value){
    _CFR2[1]=(value << 4) | ((B11001111)&(_CFR2[1]));
}

//Set or unset Digital Ramp Enable bit
void DRG::setDigitalRampEnable(boolean value){
    _CFR2[1]=(value << 3) | ((B11110111)&(_CFR2[1]));
}

//Commit changes to CFR2 Register (values stored in buffer, not active until an Update)
void DRG::commitCFR2(){
    CFR2.writeReg(_CFR2);
}

//Set Upper Limit of the sweep. Takes a tuning word
void DRG::setUpperLimit(unsigned int tuningWord){
    _upperLimit.value=tuningWord;
    //Serial.print("Upper Limit set to: ");
    //Serial.println(_upperLimit.value);
}

//Set Lower Limit of the sweep. Takes a tuning word
void DRG::setLowerLimit(unsigned int tuningWord){
    _lowerLimit.value=tuningWord;
    //Serial.print("Lower Limit set to: ");
    //Serial.println(_lowerLimit.value);
}

//Commit values of the Upper and Lower sweep limits to the buffer (not active until an Update)
void DRG::commitLimits(){
    DIGITAL_RAMP_LIMIT.sendWriteAddr();
    DIGITAL_RAMP_LIMIT.writeBytesREndian(_upperLimit.b,4,3);
    DIGITAL_RAMP_LIMIT.writeBytesREndian(_lowerLimit.b,4,3);
}

//Set or unset No-Dwell High Bit
void DRG::setNoDwellHigh(boolean value){
    _CFR2[2]=(value << 2) | ((B11111011)&(_CFR2[2]));
}

//Set or unset No-Dwell Low Bit
void DRG::setNoDwellLow(boolean value){
    _CFR2[2]=(value << 1) | ((B11111101)&(_CFR2[2]));
}

//Set negative slope to value
void DRG::setNegativeSlope(unsigned short value){
    _negSlope.value = value;
}

```

```
//Set positive slope to value
void DRG::setPositiveSlope(unsigned short value) {
    _posSlope.value = value;
}

//Commit positive and negative slope valeus to buffer (Don't take effect until an Update)
void DRG::commitSlopes() {
    DIGITAL_RAMP_RATE.sendWriteAddr();
    DIGITAL_RAMP_RATE.writeBytesREndian(_negSlope.b,2,1);
    DIGITAL_RAMP_RATE.writeBytesREndian(_posSlope.b,2,1);
}

//Set the positive step rate to the given value
void DRG::setPositiveStepRate(unsigned int step) {
    _posStep.value=step;
}

//Set the negative step rate to the given value
void DRG::setNegativeStepRate(unsigned int step) {
    _negStep.value=step;
}

//Commit positive and negative step rates to buffer (Don't take effect until an Update)
void DRG::commitStepRates() {
    DIGITAL_RAMP_STEP_SIZE.sendWriteAddr();
    DIGITAL_RAMP_STEP_SIZE.writeBytesREndian(_negStep.b,4,3);
    DIGITAL_RAMP_STEP_SIZE.writeBytesREndian(_posStep.b,4,3);
}

//Commit all values related to DRG to buffer (Don't take effect until an Update)
void DRG::commit() {
    commitCFR2();
    commitLimits();
    commitSlopes();
    commitStepRates();
}
```

```
#ifndef Definitions_h
#define Definitions_h

#include "Register.h"
#include "DRG.h"
#include "RAM.h"
#include "Ramper.h"

/*Definitions File
This defines functions and objects that will want to be shared across libraries
*/

//Compiler Macro that defines a direct port write. Is much faster than digitalWrite (for time
sensitive uses)
static __inline__ void digitalWriteDirect(int pin, boolean val){
    if(val) g_APinDescription[pin].pPort -> PIO_SODR = g_APinDescription[pin].ulPin;
    else    g_APinDescription[pin].pPort -> PIO_CODR = g_APinDescription[pin].ulPin;
}

//Compiler Macro that defines a direct port read. Is much faster than digitalRead (for time
sensitive uses)
static inline int digitalReadDirect(int pin){
    return !!(g_APinDescription[pin].pPort -> PIO_PDSR & g_APinDescription[pin].ulPin);
}

//Variables that define the state of the system
extern boolean fastUpdate;
extern boolean DHCP_ENABLE;
extern int MHZinCLK;
extern int MHZoutCLK;

//Pre-defined multi-purpose buffers
extern byte Buffer[500];
extern byte smallBuffer[10];

//Register Definitions
extern Register CFR1;
extern Register CFR2;
extern Register CFR3;
extern Register AUX_DAC_CONTROL;
extern Register IO_UPDATE_RATE;
extern Register FTW;
extern Register POW;
extern Register ASF;
extern Register MULTICHIP_SYNC;
extern Register DIGITAL_RAMP_LIMIT;
extern Register DIGITAL_RAMP_STEP_SIZE;
extern Register DIGITAL_RAMP_RATE;
extern Register SINGLE_TONE_0;
extern Register RAM_PROFILE_0;
/*extern Register SINGLE_TONE_1(0x0F,8);
extern Register RAM_PROFILE_1(0x0F,8);
extern Register SINGLE_TONE_2(0x10,8);
```

```
extern Register RAM_PROFILE_2(0x10,8);
extern Register SINGLE_TONE_3(0x11,8);
extern Register RAM_PROFILE_3(0x11,8);
extern Register SINGLE_TONE_4(0x12,8);
extern Register RAM_PROFILE_4(0x12,8);
extern Register SINGLE_TONE_5(0x13,8);
extern Register RAM_PROFILE_5(0x13,8);
extern Register SINGLE_TONE_6(0x14,8);
extern Register RAM_PROFILE_6(0x14,8);
extern Register SINGLE_TONE_7(0x15,8);
extern Register RAM_PROFILE_7(0x15,8);*/
extern Register RAM_WRITE;
```

```
//Update Function used to write values in buffer of AD9910 to active registers
```

```
void update();
```

```
#endif
```

```
#include "Definitions.h"
#include "PinDefinitions.h"

boolean fastUpdate = false;
boolean DHCP_ENABLE = false;

int MHZinCLK = 10;
int MHZoutCLK=1000;

//Pre-defined multi-purpose buffers
byte Buffer[500];
byte smallBuffer[10];

Register CFR1(0x00,4);
Register CFR2(0x01,4);
Register CFR3(0x02,4);
Register AUX_DAC_CONTROL(0x03,4);
Register IO_UPDATE_RATE(0x04,4);
Register FTW(0x07,4);
Register POW(0x08,2);
Register ASF(0x09,4);
Register MULTICHIP_SYNC(0x0A,4);
Register DIGITAL_RAMP_LIMIT(0x0B,8);
Register DIGITAL_RAMP_STEP_SIZE(0x0C,8);
Register DIGITAL_RAMP_RATE(0x0D,4);
Register SINGLE_TONE_0(0x0E,8);
Register RAM_PROFILE_0(0x0E,8);
/*Register SINGLE_TONE_1(0x0F,8);
Register RAM_PROFILE_1(0x0F,8);
Register SINGLE_TONE_2(0x10,8);
Register RAM_PROFILE_2(0x10,8);
Register SINGLE_TONE_3(0x11,8);
Register RAM_PROFILE_3(0x11,8);
Register SINGLE_TONE_4(0x12,8);
Register RAM_PROFILE_4(0x12,8);
Register SINGLE_TONE_5(0x13,8);
Register RAM_PROFILE_5(0x13,8);
Register SINGLE_TONE_6(0x14,8);
Register RAM_PROFILE_6(0x14,8);
Register SINGLE_TONE_7(0x15,8);
Register RAM_PROFILE_7(0x15,8);*/
Register RAM_WRITE(0x16,4);

void update(){
    //Special update used to maximize speed of update
    digitalWriteDirect(IO_UPDATE, HIGH);
    if(!fastUpdate){
        delayMicroseconds(1);
    }
    digitalWriteDirect(IO_UPDATE, LOW);
}
```



```
#ifndef Ethernet_Handler_h
#define Ethernet_Handler_h

#include <Ethernet.h>
#include "Ramp.h"

//Variables that need to be accessed out of the context of Ethernet_Handler
extern EthernetClient client;
extern EthernetServer server;

//Functions pertaining to general Ethernet port use
void setup_Ethernet(boolean DHCP_ENABLE);
byte loadRamps(Ramp *bufferArray);
boolean sendOutRamps(int numOfRamps, Ramp* rampArray);
boolean sendAddressAcross();
void sendCommand(char c);
void sendCommandStatus(boolean v);
void sendSuccessfulHeader();
void endClientTransmission();
void sendErrorMessage(char* errorMessage);
void sendFailHeader();
unsigned int getJSONint(char token);
boolean navigateTo(char* token, int N, int count = 500);
char* readUpTo(char token);
boolean handleInformation();

#endif
```

```

#include "Ethernet_Handler.h"
#include "SD_Handler.h"
#include "LCD_Handler.h"
#include "Definitions.h"
#include "PLL.h"

//Variables used to define Ethernet constants
EthernetClient client; // Client instance for connected client
EthernetServer server(80); //Server instance on port 80
byte mac[] = {0x90,0xa2,0xda,0x0f,0x19,0x4c}; //MAC address
IPAddress ip(192,168,0,178); // Static IP address

//small Char buffer
char dataBuffer[10];

//Sets up Ethernet after Arduino boot
void setup_Ethernet(boolea DHCP_ENABLE){
  //Use DHCP if DHCP_ENABLE is true (set by SD card: false by default)
  if(DHCP_ENABLE){
    Serial.print(F("Leasing IP from DHCP..."));
    if(Ethernet.begin(mac)){
      Serial.println(F("Success"));
    }
    else{
      Serial.println(F("Failed"));
    }
  }
  //Otherwise ues the hardcoded static IP
  else{
    Serial.print(F("Setting up at static ip: "));
    Ethernet.begin(mac,ip);
    Serial.println(Ethernet.localIP());
  }
  //Start the server (Listens for connections)
  server.begin();
}

//Write the currently saved ramps into the JSON response
boolean sendOutRamps(int numofRamps, Ramp* rampArray){
  client.print(",\"ramps\":[");
  for(int i =0; i<numofRamps;i++){
    client.print("{");
    client.print("\"l\":"); client.print(rampArray[i].lowWord); client.print(" , ");
    client.print("\"h\":"); client.print(rampArray[i].highWord); client.print(" , ");
    client.print("\"dir\":"); client.print(rampArray[i].direction); client.print("}");
    if(i!=numofRamps-1){
      client.print(",");
    }
  }
  client.print("]");
  return true;
}

```

```
//Load ramps passed in the payload "PL="
byte loadRamps( Ramp *bufferArray) {
    //Get the number of ramps to load (last part of the command)
    char temp[5];
    client.readBytesUntil(':',temp,5);
    byte numOfRamps = atoi(temp);

    //Inform if there are too many ramps for a sweep
    if(numOfRamps>255){
        Serial.print(numOfRamps);Serial.println(F(" is too many ramps for one sweep! (Max=255)"));
        sendErrorMessage("Too many Ramps; Only 255 Ramps supported to keep resources free");
        return false;
    }

    //Create temporary Ramp to write values into
    Ramp tempRamp = Ramp();

    //Navigate to payload
    navigateTo("PL=", 3);
    int tic = millis();//For performance monitoring

    for( int j=0; j<numOfRamps; j++){

        //If indices don't match, cancel load
        if(j!=getJSONint('n')-1){
            Serial.println(F("Mismatch in ramp indexes: Load cancelled."));
            return false;
        }
        myLCDprint("Loading Ramp", j);

        //Find values in JSON string and set them as the values for the temporary Ramp
        tempRamp.highWord= getJSONint('h');
        tempRamp.lowWord= getJSONint('l');
        tempRamp.timeStep = getJSONint('t');
        tempRamp.wordStep = getJSONint('w');
        tempRamp.noDwellHigh = getJSONint('H');
        tempRamp.noDwellLow = getJSONint('L');
        tempRamp.wordDest = getJSONint('D');
        tempRamp.direction = getJSONint('d');

        //Copy the temporary ramp into the buffer array
        bufferArray[j] = tempRamp;
    }
    int toc =millis(); //For performance monitoring

    Serial.print("Time taken to load ramps to RAM: ");
    Serial.print(toc-tic);
    Serial.println(" milliseconds");

    //Return the number of ramps loaded
    return numOfRamps;
}
```

```
//Writes the device's IP address into the JSON response
boolean sendAddressAcross() {
    client.print(",\"address\": \"");
    client.print(Ethernet.localIP());
    client.print("\");
    return true;
}

//Writes the given character into the JSON response
void sendCommand(char c){
    client.print(",\"C\": \"");
    client.print(c);
    client.print("\");
}

//Writes the given command status result into the JSON response
void sendCommandStatus(boolean v){
    client.print(",\"CS\":");
    client.print(v);
}

//Write headers for a successful communication with the DDS
void sendSuccessfulHeader(){
    //HTTP OK Response
    client.println("HTTP/1.1 200 OK");

    //Needed for cross domian JSON requests
    client.println("Access-Control-Allow-Origin: null");

    //Informs the browser that the DDS will close the connection upon finishing writing
    information
    client.println("Connection: close");

    //The data sent in the response will be JSON
    client.println("Content-Type: application/json");

    //Blank line seperating JSON from data
    client.println();

    //Begin the JSON response with a "transmission" : "Okay" field
    client.print("{\"transmission\": \"Okay\"");
}

//End the JSON object begun with sendSuccesfulHeader
void endClientTransmission(){
    client.println("}");
}

//Write an error message into the JSON response
void sendErrorMessage(char* errorMessage){
    client.print("{\"error\": \"");
```

```
client.print(errorMessage);
client.println("\n");
}

//Write a 404 -- Page Not Found error to the client
//Used when it can not find the webpage on the SD card
void sendFailHeader(){
    client.println("HTTP/1.0 404 Not Found");
}

//Function that finds a JSON key character in the Client stream
unsigned int getJSONint(char token){
    //create acutal key sequence string -- "X":
    char key[4] = {"",token,"",':'};

    //Pass key to navigateTo
    navigateTo(key, 4);

    //read up to next key and convert value to integer
    unsigned int val = atoi(readUpTo(', '));

    //return value
    return val ;
}

//Write DDS information into JSON response
boolean handleInformation(){
    client.print(F("{\"info\":{\"clock\":\"}"));

    //System clock given by PLL static variable
    client.print(PLL::fSysClock);

    client.print(F("{\"status\":\"Idle\"}"));
    return true;
}

//Function to read up to a given token in the client stream
//Returns the Data up to and including the token
char* readUpTo(char token){
    memset(dataBuffer, ' ', 10);
    if(client.readBytesUntil(token,dataBuffer,10)==0){
        Serial.println("Couldn't Find Token");
    }
    return dataBuffer;
}

//Experimental function to take a file name
//and write the file to the client
boolean writeFileToClient(char *file){
    if(!SD.exists(file)){
        return false;
    }
}
```

```
}
myFile = SD.open(file);
if(!myFile){
    return false;
}
while(client.connected() && myFile.available()){
    memset(Buffer, 0, sizeof(Buffer));
    myFile.read(Buffer, sizeof(Buffer));
    client.write(Buffer, sizeof(Buffer));
}
return true;
}

//Experimental function to take data from the client
//and write it to a given file name on the SD card
boolean writeClientStreamToFile(char* file){
    if(SD.exists(file)){
        return false;
    }
    myFile=SD.open(file, FILE_WRITE);
    if(!myFile){
        return false;
    }
    while(client.available()){
        memset(Buffer, 0, sizeof(Buffer));
        client.readBytes(Buffer, sizeof(Buffer));
        myFile.write(Buffer, sizeof(Buffer));
    }
}

//Function to navigate up to and including a certain token
//Everything up to and the token is tossed
//Count defaults to 500 if not given and is the number of characters to look through before
giving up
//returns true if found, false if not
boolean navigateTo(char* token, int N, int count){
    int foundChars = N;
    while(foundChars > 0 && count > 0){
        foundChars=N;
        for(int i =0; i<N;i++){
            char c = client.read();
            count--;
            if(c==token[i]){
                foundChars--;
            }
            else if(c==token[0]){
                foundChars=N-1;
                i=0;
            }
            else{
                break;
            }
        }
    }
}
```

```
    }  
  }  
  if(count==0){  
    return false;  
  }  
  return true;  
}
```

```
#ifndef LCD_Handler_h
#define LCD_Handler_h

#include <LiquidCrystal.h>
#include <Ethernet.h>

//External LCD definition so other functions can access it if need be
extern LiquidCrystal lcd;

//Functions that print various data types to the
//2-line LCD
void myLCDprint(char* line1, int line2);
void myLCDprint(char* line1, IPAddress line2);
void myLCDprint(char* line1, char line2);
void myLCDprint(char* line1, unsigned int line2);
void myLCDprint(char* line1, char* line2);

//Function that sets up the LCD screen initially
void setupLCD();

#endif
```

```
#include "LCD_Handler.h"

//Object that controls the LCD screen
//Needs these pins in this order
LiquidCrystal lcd(8, 13, 9, 3, 5, 6, 7);

//Initializes the LCD and prints message
void setupLCD() {
    lcd.begin(16,2);
    lcd.clear();
    lcd.print("LCD Screen");
    lcd.setCursor(0,1);
    lcd.print("Initialized");
}

//All LCD print functions do the same thing
//Print Argument 1 to the first line and Argument 2 to the second
void myLCDprint(char* line1, int line2){
    myLCDprint(line1, (unsigned int) line2);
}
void myLCDprint(char* line1, IPAddress line2){
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print(line1);
    lcd.setCursor(0,1);
    lcd.print(line2);
}
void myLCDprint(char* line1, char line2){
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print(line1);
    lcd.setCursor(0,1);
    lcd.print(line2);
}
void myLCDprint(char* line1, unsigned int line2){
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print(line1);
    lcd.setCursor(0,1);
    lcd.print(line2);
}
void myLCDprint(char* line1, char* line2){
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print(line1);
    lcd.setCursor(0,1);
    lcd.print(line2);
}
```



```
// memoryFree header
// From http://www.arduino.cc/cgi-bin/yabb2/YaBB.pl?num=1213583720/15
// ...written by user "mem".

//External library for memory management

#ifndef MEMORY_FREE_H
#define MEMORY_FREE_H

int freeMemory();

#endif
```

```
#ifdef __arm__
// should use unistd.h to define sbrk but Due causes a conflict
extern "C" char* sbrk(int incr);
#else // __ARM__
extern char *__brkval;
#endif // __arm__

int freeMemory() {
    char top;
#ifdef __arm__
    return &top - reinterpret_cast<char*>(sbrk(0));
#elif defined(CORE_TEENSY) || (ARDUINO > 103 && ARDUINO != 151)
    return &top - __brkval;
#else // __arm__
    return __brkval ? &top - __brkval : &top - __malloc_heap_start;
#endif // __arm__
}
```

```
//PLL class for handling the Phase-Locked-Loop functions of the AD9910
```

```
#ifndef PLL_h
```

```
#define PLL_h
```

```
#include "Arduino.h"
```

```
#include "Register.h"
```

```
class PLL{
```

```
public:
```

```
    PLL();
```

```
    void setDRV0(byte option);
```

```
    void setVCO(int value);
```

```
    void setN(byte multiplier);
```

```
    void setPLLEnable(boolean value);
```

```
    void setInputDividerBypass(boolean value);
```

```
    void setInputDividerResetB(boolean value);
```

```
    void setDividerBypass(boolean value);
```

```
    void setDividerResetB(boolean value);
```

```
    void setPFDReset(boolean value);
```

```
    static unsigned int freqToFTW(int Freq);
```

```
    void setOutputCurrent(byte option);
```

```
    boolean isLocked();
```

```
    void commit();
```

```
    void quickSetup(int inMHz, int outMHz);
```

```
    static unsigned int fSysClock;
```

```
private:
```

```
    //Register *CFR3;
```

```
    byte _CFR3[4];
```

```
    union{
```

```
        struct {
```

```
            unsigned int :2;
```

```
            unsigned int DRV0 : 2;
```

```
            unsigned int :1;
```

```
            unsigned int VCO :3;
```

```
            unsigned int :2;
```

```
            unsigned int Icp :3;
```

```
            unsigned int :3;
```

```
            unsigned int InDivBypass: 1;
```

```
            unsigned int InDivBypassB: 1;
```

```
            unsigned int :3;
```

```
            unsigned int PFD_Reset: 1;
```

```
            unsigned int :1;
```

```
            unsigned int PLL_Enable: 1;
```

```
            unsigned int N :7;
```

```
            unsigned int :1;
```

```
        };
```

```
        byte bytes[4];
```

```
    }_data;
```

```
};
```

```
#endif
```

```
#include "PLL.h"
#include "PinDefinitions.h"
#include "Definitions.h"

//Static variable that is set when the PLL is. Contains
//the set clock; used for calculating frequency words
unsigned int PLL::fSysClock;

//Constructor for PLL class object.
//Reads CFR3 data into private data
PLL::PLL() {
    CFR3.read();
    for(int i=0; i<4;i++){
        _CFR3[i]=CFR3._data[i];
        _data.bytes[i] = CFR3._data[i];
    }
}

//Function that sets the DRV0 part of the CFR3 register to the given option
void PLL::setDRV0(byte option) {
    _data.DRV0=option;
    _CFR3[0]=(option << 4) | ((B11001111)&(_CFR3[0]));
}

//Function that sets the VCO part of the CFR3 register to the given value
void PLL::setVCO(int value) {
    _data.VCO=value;
    _CFR3[0]=(value) | ((B11111000)&(_CFR3[0]));
}

//Sets the PLL multiplier section of CFR3 to the given multiplier
void PLL::setN(byte multiplier) {
    _data.N=multiplier;
    _CFR3[3]=(multiplier << 1) | ((B00000001)&(_CFR3[3]));
}

//Sets or unsets the PLL enable bit in CFR3
void PLL::setPLLEnable(boolean value) {
    _data.PLL_Enable = value;
    _CFR3[2]=(value) | ((B11111110)&(_CFR3[2]));
}

//Sets or unsets the Divider Bypass bit in CFR3
void PLL::setDividerBypass(boolean value) {
    _data.InDivBypass = value;
    _CFR3[2]=(value << 7) | ((B01111111)&(_CFR3[2]));
}

//Sets or unsets the Divder Bypass Reset bit in CFR3
void PLL::setDividerResetB(boolean value) {
    _data.InDivBypassB = value;
    _CFR3[2]=(value << 6) | ((B10111111)&(_CFR3[2]));
}
```

```
}

//Sets or unsets the Set PFD Reset bit in CFR3
void PLL::setPFDReset(boolean value){
    _data.PFD_Reset = value;
    _CFR3[2]=(value << 2) | ((B11111011)&(_CFR3[2]));
}

//Sets the output current section of CFR3 to the given value
void PLL::setOutputCurrent(byte value){
    _data.Icp = value;
    _CFR3[1]=(value << 3) | ((B11000111)&(_CFR3[3]));
}

//Returns true if PLL lock is established, false otherwise
boolean PLL::isLocked(){
    return digitalRead(PLL_LOCK);
}

//Static function that converts a frequency (in Hz) to a tuning word based on the current clock
unsigned int PLL::freqToFTW(int Freq){
    //Serial.print("Converting ");
    //Serial.println(Freq);
    return (unsigned int)(
        ((long long)(0xFFFFFFFF)*(long long)Freq)
        /((long long)PLL::fSysClock));
}

//Function that sets all needed PLL parameters based on an input clock and desired output
void PLL::quickSetup(int inMHz, int outMHz){
    PLL::fSysClock = outMHz*1000000;
    if(outMHz > 200){
        fastUpdate=true;
    }
    Serial.print("Setting PLL ");
    Serial.print(inMHz);
    Serial.print(" MHz -> ");
    Serial.print(outMHz);
    Serial.println(" MHz");

    setN(outMHz/inMHz);
    if (outMHz>=880){
        setVCO(5);
    }
    else if(outMHz>=770){
        setVCO(4);
    }
    else if(outMHz>=650){
        setVCO(3);
    }
    else if(outMHz >= 550){
        setVCO(2);
    }
}
```

```
else if(outMHz >= 450){
    setVCO(1);
}
else{
    setVCO(0);
}
setPLLEnable(true);
commit();
}

//Save all changes to CFR3 to buffer. Changes do not take effect until in IO Update
void PLL::commit(){
    CFR3.writeReg(_CFR3);
}
```

```
//Library that defines getting a single-tone frequency output from the AD9910
```

```
#ifndef Profile_h
#define Profile_h

#include "Arduino.h"
#include "Register.h"

class Profile{
public:
    Profile();
    Profile(Register &ProfReg);
    void setFTW(unsigned int FTW, boolean com= false);
    static unsigned int freqToFTW(int Freq);
    void setFreq(int freq, boolean com= true);
    void setPOW(int POW, boolean com = false);
    void setASF(int ASF, boolean com = false);
        void produceFreq(int Freq);
        void produceFTW(unsigned int FTW);
    void commit();
private:
    void commit(boolean com);
    union i_ftw{
        unsigned int word;
        byte bytes[4];
    }indexedFTW;
    Register *profReg;
    byte _data[8];
    union{
        struct{
            unsigned int :2;
            unsigned short ASF :14;
            unsigned short POW :16;
            unsigned int FTW ;
        };
        byte bytes[8];
    }_reg;
};

#endif
```

```
#include "PLL.h"
#include "Profile.h"
#include "Definitions.h"

//Empty Profile Constructor
//Used to set aside memory for a profile instance before it has been [can be] initialized
Profile::Profile() {}

//Main constructor
//Accepts a Profile Register
Profile::Profile(Register &ProfReg) {
    profReg = &ProfReg;
    profReg->read();
    for(int i=0; i<8;i++){
        //_data[i]=profReg->_data[i];
        _reg.bytes[i]=profReg->_data[i];
    }
}

//Set the Phase Word in the Profile to the value. Com indicates whether to commit afterwards
void Profile::setPOW(int POW, boolean com) {
    _reg.POW = POW;
    commit(com);
}

//Set the Amplitude Word in the Profile to the value. Com indicates whether to commit afterwards
void Profile::setASF(int ASF, boolean com) {
    _reg.ASF=ASF;
    commit(com);
}

//Set the Frequency Word in the Profile to the value. Com indicates whether to commit afterwards
void Profile::setFTW(unsigned int FTW, boolean com) {
    indexedFTW.word = FTW;
    _reg.FTW = FTW;
    //Serial.print("Writing Tuning Word: ");
    //Serial.println(FTW);

    /* for( int i =4; i<8;i++){
        //Serial.println(indexedFTW.bytes[7-i]);
        _data[i]=indexedFTW.bytes[7-i];
    } */
    if (com){
        Serial.println("Committing...");
        commit();
    }
}

//Converts a frequency in Hertz into a tuning word
unsigned int Profile::freqToFTW(int Freq) {
    //Serial.print("Converting ");
    //Serial.println(Freq);
    return (unsigned int) (
```

```
    ((long long) (0xFFFFFFFF) * (long long) Freq)
    / ((long long) PLL::fSysClock));
}

//Sets a frequency in Hertz into the Profile. Com indicates whether to commit
void Profile::setFreq(int Freq, boolean com) {
    Serial.print("Attempting to set to: ");
    Serial.println(Freq);
    unsigned int FTW = freqToFTW(Freq);
    Serial.print("Result of Calculation: " );
    Serial.println(FTW);
    setFTW(FTW, com);
}

//Produces the given tuning word at the output of the DDS. Disables all other modes of input
words
void Profile::produceFTW(unsigned int FTW) {
    setFTW(FTW, true);
    //Set DRG Disabled, RAM Disabled, Profile Enabled
    //RAM
    CFR1._data[0] = (CFR1._data[0] & (0b01111111));
    CFR1.writeReg(CFR1._data);
    //DRG
    CFR2._data[1] = (B11110111) & (CFR2._data[1]);
    CFR2.writeReg(CFR2._data);

    update();
}

//Produces the given frequency at the output of the DDS. Disables all other modes of input words
void Profile::produceFreq(int Freq) {
    produceFTW(freqToFTW(Freq));
}

//Commits changes to AD9910 buffer. changes don't take effect until an IO update
void Profile::commit(boolean com) {
    if (com) {
        commit();
    }
}

//Commits changes to AD9910 buffer. changes don't take effect until an IO update
void Profile::commit() {
    profReg->sendWriteAddr();
    profReg->writeBytes(_reg.bytes, 4);
    profReg->writeBytesREndian(_reg.bytes, 4, 7);
}
```

```
//Class that controls writing to DDS RAM

#ifndef RAM_h
#define RAM_h

#include "Arduino.h"
#include "Register.h"

#define FREQUENCY 0
#define PHASE 32
#define AMPLITUDE 64
#define POLAR 96

class RAM{
public:
    RAM();
    RAM(Register &cf1, Register &prof0, Register &ramWrite);
    void stallFreq(unsigned int FTW);
    void stopStall();
    void setupStall(unsigned int FTW);
    void setSingleTone();
private:
    void setMode(byte mode);
    void writeWord(unsigned int word);
    void setRAMEnabled();
    void setRAMDisabled();
    void commitCFR1();
    void totalCommit();
    union{
        struct{
            unsigned int WORD ;
        };
        byte bytes[8];
    }_RAMreg;
    union{
        struct{
            int :8;
            int addStepRate :16;
            int end_address :10;
            int :6;
            int start_address:10;
            int :8;
            int noDwell :1;
            int :1;
            int zeroCross :1;
            int ramMode :3;
        };
        byte bytes[8];
    }_profReg;
    byte currentMode;
    Register *CFR1;
    Register *Prof0;
    Register *RAMWrite;
```

```
};  
#endif
```

```

#include "RAM.h"
#include "PinDefinitions.h"
#include "Definitions.h"

//Blank constructor. Sets aside memory for object without initializing it
RAM::RAM() {}

//Main Constructor for a Ram object. Intializes to SingeTone mode
RAM::RAM(Register &cfR1, Register &prof0, Register &ramWrite) {
    CFR1 = &cfR1;
    Prof0 = &prof0;

    RAMWrite= &ramWrite;
    currentMode = 0; //Current mode is 0 (Frequency)

    //Read bytes into private storage
    for (int i =0; i<8; i++){
        _profReg.bytes[i]=0;
    }

    //Set RAM mode to single tone mode
    setSingleTone();
}

//Sets the Ram to single Tone mode
void RAM::setSingleTone() {
    _profReg.ramMode=0;
    _profReg.noDwell=0;
    _profReg.zeroCross=0;
    _profReg.start_address=0;
    //_profReg.end_address=0;
    _profReg.bytes[4]=0b01000000;
    Prof0->writeReg(_profReg.bytes);
    update();
}

//Sets the word destination mode to the given value
void RAM::setMode(byte mode) {
    currentMode = mode;
    //Serial.print("Setting mode:\nBefore:");
    //Serial.println(CFR1->_data[0]);
    CFR1->_data[0]=((CFR1->_data[0]&(0b10011111)) |mode);
    //Serial.print("After:");
    //Serial.println(CFR1->_data[0]);
}

//Enables the RAM in the CFR1 Register. When RAM is enabled, it overrides all other word sources
void RAM::setRAMEnabled() {
    //Serial.print("Setting Ram Enabled:\nBefore:");
    //Serial.println(CFR1->_data[0]);
    CFR1->_data[0]=((CFR1->_data[0]&(0b01111111)) | (0b1000000));
    //Serial.print("After:");

```

```
//Serial.println(CFR1->_data[0]);
}

//Set the RAM enable bit to disabled in the CFR1 register. The RAM will cease to be a word source
void RAM::setRAMDisabled() {
    CFR1->_data[0]=(CFR1->_data[0]&(0b01111111));
}

//Writes a FTW into the RAM.
void RAM::writeWord(unsigned int word) {
    _RAMreg.WORD = word;
    //Serial.print("Writing ");
    //Serial.print(_RAMreg.WORD);
    //Serial.println(" to RAM");
    RAMWrite->sendWriteAddr();
    RAMWrite->writeBytesREndian(_RAMreg.bytes, 4, 3);
    RAMWrite->writeBytesREndian(_RAMreg.bytes,4,3);
}

//Commit changes to CFR1 to the buffer. Changes do not take effect until an update
void RAM::commitCFR1() {
    CFR1->writeReg(CFR1->_data);
}

//Commits CFR1 and then performs an update
void RAM::totalCommit() {
    commitCFR1();
    update();
    //CFR1->readout();
}

//Sets up a RAM stall at the given FTW
void RAM::setupStall(unsigned int FTW) {
    if(currentMode != FREQUENCY) {
        //Serial.println("Mode not in Frequency; Setting...");
        setMode(FREQUENCY);
    }
    writeWord(FTW);
    setRAMEnabled();
    commitCFR1();
}

//Performs a stall at the given FTW
void RAM::stallFreq(unsigned int FTW) {
    setupStall(FTW);
    update();
}

//Stops a stall in effect
void RAM::stopStall() {
    setRAMDisabled();
    //commitCFR1();
    totalCommit();
}
```

}

```
//Ramp object. Contains data needed to make a Ramp
//Should take up 17 bytes

#ifndef Ramp_h
#define Ramp_h

#include "Arduino.h"

class Ramp{
public:
    Ramp(unsigned int lowWord, unsigned int highWord, unsigned int timeStep, unsigned int
wordStep, boolean noDwellHigh, boolean noDwellLow, byte wordDest, boolean dir);
    Ramp(int lowFreq, int highFreq, long long milliseconds, byte wordDest, boolean dir = 1);
    Ramp();
    struct {
        unsigned int lowWord;
        unsigned int highWord;
        unsigned int timeStep;
        unsigned int wordStep;
        unsigned int noDwellHigh :1;
        unsigned int noDwellLow : 1;
        unsigned int wordDest :3;
        unsigned int direction: 1;
        unsigned int :1;
    };

private:
};
#endif
```

```
#include "PLL.h"
#include "Ramp.h"

//Returns empty(initialized to 0) ramp object
Ramp::Ramp() {}

//Places data into fields
Ramp::Ramp(unsigned int lowWord, unsigned int highWord, unsigned int timeStep,
           unsigned int wordStep, boolean noDwellHigh,
           boolean noDwellLow, byte wordDest, boolean dir){
    lowWord= lowWord;
    highWord = highWord;
    timeStep = timeStep;
    wordStep = wordStep;
    noDwellHigh = noDwellHigh;
    noDwellLow = noDwellLow;
    wordDest = wordDest;
    direction = dir;
}

//Converts readable data and places into fields (Slowest)
Ramp::Ramp(int lowFreq, int highFreq, long long milliseconds, byte wordDest, boolean dir){
    lowWord = PLL::freqToFTW(lowFreq);
    highWord= PLL::freqToFTW(highFreq);
    direction = dir;
    int count = 0;
    timeStep=0;
    while (timeStep==0){
        count++;
        timeStep = ((count*milliseconds*((long long) (PLL::fSysClock)))/(highWord-lowWord))/4000;
    }
    wordDest = wordDest;
    noDwellHigh=0;
    noDwellLow=0;
    if(wordDest==0){
        //Frequency word
        wordStep=count;
        //wordStep = ((unsigned long long)timeStep * PLL::fSysClock)/(0x100000000);
    }
}
```

```
//Class that controls the DRG and RAM using instances
//of the Ramp class in order to create sweeps and chained sweeps
#ifndef Ramper_h
#define Ramper_h

#include "Arduino.h"
#include "DRG.h"
#include "Ramp.h"
#include "RAM.h"

class Ramper{
public:
    Ramper();
    Ramper(DRG &drg , RAM &ramObject);
    void loadSingleSweep(Ramp ramp);
    void doSweep(Ramp ramp);
    void chainedSweeps(Ramp* rampArray, int length);
    void beginSweep(boolean dir);
//        static void beginSweepUp();
//        static void beginSweepDown();
    static void trigger();
    static void resetTrigger();
    void Bump();

private:
    void turnOnDRG();
    RAM *myRAM;
    DRG *drg;
};
#endif
```

```
#include "Arduino.h"
#include "Ramper.h"
#include "PinDefinitions.h"
#include "Definitions.h"

//Blank constructor. Sets aside memory for object before it is initialized
Ramper::Ramper() {}

//Normal Constructor. Ramper only needs a reference to a DRG object instance and a RAM object
instance
Ramper::Ramper(DRG &myDrg , RAM &ramObject){
    drg = &myDrg;
    myRAM = &ramObject;
}

//Turns on the DRG as a frequency word source
void Ramper::turnOnDRG() {
    drg->setDigitalRampEnable(true);
    drg->commitCFR2();
    update();
}

//Performs a single sweep. Takes one sweep to perform
void Ramper::doSweep(Ramp ramp) {
    loadSingleSweep(ramp);
    turnOnDRG();
    beginSweep(ramp.direction);
}

//class variable informing whether a sweep has been loaded
boolean loaded = false;

//class variable informing if the device has been told to come out of sweep mode
boolean RESET_FLAG=false;

boolean bumped= false;
//Chains sweeps in rampArray together to perform one constant sweep
//Length is the number of sweeps to go through
void Ramper::chainedSweeps(Ramp *rampArray, int length){
    RESET_FLAG = false; //set reset flag to false

    //attach sweepReset interrupt
    attachInterrupt(SWEEP_RESET_TRIGGER, Ramper::resetTrigger, RISING);

    loaded=false; //Set loaded to false as a ramp hasn't been loaded yet

    //attach sweep trigger interrupt
    attachInterrupt(SWEEP_TRIGGER, Ramper::trigger, RISING);

    turnOnDRG(); //Ensure that the DRG is in control of word generation for the AD9910
    myRAM->setSingleTone(); //Ensure that the Profile is in RAM mode in case single tone
    was used before
```

```

    //For each ramp in the array:
    for(int i =0; i<length; i++){

        //Load sweep
        if(!loaded){
            loadSingleSweep(rampArray[i]);
            loaded=true;
        }

        //Standalone Trigger mechanism
        //Waits until sweep is completed, triggers sweep pin
        while(!drg->isAtLimit()){

            digitalWrite(FAUX_TRIGGER, HIGH);

            while(loaded){
                if(RESET_FLAG){
                    detachInterrupt(SWEEP_RESET_TRIGGER);
                    detachInterrupt(SWEEP_TRIGGER);
                    return;
                }
            };

            digitalWrite(FAUX_TRIGGER, LOW);
            int tic=micros();
            beginSweep(rampArray[i].direction);
            int toc=micros();
            Serial.print("Time taken to begin sweep:");Serial.println(toc-tic);
        }
    }

```

```

//Sweep stored after the other sweeps is the "Bump" sweep

```

```

loadSingleSweep(rampArray[length]);
int tic = millis();
bumped = false;
int timeout = 60*1000; // 1 minute
attachInterrupt(SWEEP_TRIGGER,Bump, RISING);
//Stay in bump mode until reset is triggered or timeout occurs
while(!RESET_FLAG && millis()-tic < timeout){
    if(bumped){
        tic = millis();
        bumped = false;
    }
}
detachInterrupt(SWEEP_TRIGGER);
detachInterrupt(SWEEP_RESET_TRIGGER);

```

```

}

//Loads a single sweep into the AD9910
void Ramper::loadSingleSweep(Ramp ramp){

```

```

int tic = micros(); //For perfomance monitoring

    //Set DRG parameters to those given by the ramp
drg->setNoDwellHigh(ramp.noDwellHigh);
drg->setNoDwellLow(ramp.noDwellLow);
drg->setDigitalRampDestination(ramp.wordDest);
drg->setUpperLimit(ramp.highWord);
drg->setLowerLimit(ramp.lowWord);

    //enable DRG ramp
drg->setDigitalRampEnable(true);

    //If a positive sweep:
if(ramp.direction) {
    drg->setPositiveSlope(ramp.timeStep);

    drg->setNegativeSlope(1); //Fastest Step

    drg->setPositiveStepRate(ramp.wordStep);

    drg->setNegativeStepRate(0xFFFF); //Largest Step

    drg->slopeControlLow();
}

    //If a negative sweep
else{
    drg->setPositiveSlope(1); //Fastest Step

    drg->setNegativeSlope(ramp.timeStep);

    drg->setPositiveStepRate(0xFFFF); //Largest Step

    drg->setNegativeStepRate(ramp.wordStep);

    myRAM->setupStall(ramp.highWord);
}

    //Commit values to BUFFER,
    //new ramp values will not take effect until an update
drg->commit();

int toc = micros(); //for performance monitoring

Serial.print("Time taken to load ramp: ");
Serial.println(toc-tic);
}

//Trigger function -- indicates new ramp needs to be loaded
void Ramper::trigger() {
    loaded=false;
}

```

```
void Ramper::resetTrigger () {
    RESET_FLAG=true;
}

void setDrgLow () {
    detachInterrupt (DROVER);
    digitalWrite (DRCTL, LOW);
    attachInterrupt (SWEEP_TRIGGER, Bump, RISING);
}

Ramper::Bump () {
    detachInterrupt (SWEEP_TRIGGER);
    bumped=true;
    digitalWrite (DRCTL, HIGH);
    attachInterrupt (DROVER, setDrgLow, RISING);
}

//Function to begin the currently loaded (in buffer) sweep
void Ramper::beginSweep (boolean dir) {
    if (dir) {
        update ();
        digitalWrite (DRCTL, HIGH);
        //drg->slopeControlHigh ();
    }
    else {
        //Then we want to stall at ramp.highWord
        update ();

        //Then we want to sweep up ramp until high
        digitalWrite (DRCTL, HIGH);
        //drg->slopeControlHigh ();

        while (!digitalReadDirect (DROVER)) {}
        //Remove RAM control
        myRAM->stopStall ();

        //Begin sweep down
        digitalWrite (DRCTL, LOW);
        //drg->slopeControlLow ();
    }
}
```

```
//Register class -- Backbone of AD9910 communication and programming
```

```
#ifndef Register_h
```

```
#define Register_h
```

```
#include "Arduino.h"
```

```
#include "SPI.h"
```

```
class Register{
```

```
public:
```

```
    Register(byte address, int dataLength);
```

```
    void readout();
```

```
    void writeReg(byte* data);
```

```
    void read();
```

```
    void sendReadByte();
```

```
    void sendWriteByte();
```

```
    void sendReadAddr();
```

```
    void sendWriteAddr();
```

```
    void writeBytesREndian(byte* data, int length, int start);
```

```
    void writeBytes(byte* data, int length, int start=0);
```

```
    byte _data[8];
```

```
    byte _addr;
```

```
    byte _regLength;
```

```
private:
```

```
};
```

```
#endif
```

```
#include "Arduino.h"
#include "SPI.h"
#include "Register.h"
#include "PinDefinitions.h"

//Constructor that takes the register's address and length
Register::Register(byte address, int dataLength){
    _regLength=dataLength;
    _addr=address;
}

//Sends the write code for the register (it's address)
void Register::sendWriteAddr(){
    //Serial.print("Sending Write Byte: ");
    //Serial.println(_addr);
    SPI.transfer(DDS_CS,_addr,SPI_CONTINUE);
}

//Sends the read code for the regegister (0x80| it's address)
void Register::sendReadAddr(){
    //Serial.print("Sending read byte: ");
    //Serial.println(0x80|_addr);
    SPI.transfer(DDS_CS,0x80|_addr,SPI_CONTINUE);
}

//Write an array of bytes to the register(for use in sending custom length data)
void Register::writeBytes(byte* data, int length, int start){
    for(int i=start;i<length-1;i++){
        //Serial.print("Sending byte: ");
        //Serial.println(data[i]);
        SPI.transfer(DDS_CS, data[i],SPI_CONTINUE);
    }
    SPI.transfer(DDS_CS, data[length-1]);
}

//Write an array of bytes backward to the register (for use in sending custom length data)
//Used because the Arduino stores in opposite Endianess of the AD9910 so
//indexed structs are "backwards"
void Register::writeBytesREndian(byte* data, int length, int start){
    for(int i =start; i>start-(length-1); i--){
        //Serial.print("Sending byte: ");
        //Serial.println(data[i]);
        SPI.transfer(DDS_CS,data[i],SPI_CONTINUE);
    }
    SPI.transfer(DDS_CS,data[start-(length-1)]);
}

//Write the register's length of bytes to the Register from the given array
void Register::writeReg(byte* data){
    sendWriteAddr();
    writeBytes(data, _regLength);
}
```

```
//Print to serial the data contained in the register
void Register::readout () {
    read ();
    for (int i=0;i<_regLength;i++) {
        Serial.println(_data[i]);
    }
}

//Read into private data storage, the register's values
void Register::read () {
    sendReadAddr ();
    for (int i=0;i<_regLength-1;i++) {
        _data[i]=SPI.transfer (DDS_CS,0,SPI_CONTINUE);
    }
    _data[_regLength-1]=SPI.transfer (DDS_CS,0);
}
```

```
//Library for handling SD functions
#ifndef SD_Handler_h
#define SD_Handler_h

#include <SD.h>

extern boolean SDisBegun;
extern File myFile;

void setupSD();
boolean navigateToSD(char* token, int N);

#endif
```

```
#include "SD_Handler.h"
#include "Definitions.h"

//Boolean to check if an SD card has been inserted
boolean SDisBegun = false;

//File instance put aside to be used for all File I/O operations
File myFile;

//general use buffer for SD functions
char charBuffer[15];

//Handles initialization of SD card [Run during setup]
void setupSD(){
    Serial.print("SD Card...");

    //If SD card cannot be begun (things are not wired correctly, SD card not inserted)
    if(!SD.begin(4)){
        Serial.println("Not Inserted: Using default setup values");
    }
    //Otherwise
    else{
        SDisBegun = true;
        Serial.println(F("Found"));

        //Open config file
        myFile = SD.open("Config.txt");

        //If file exists (opens)
        if(myFile){
            Serial.println(F("Using setup values in Config.txt:"));
            //Navigate to CLK_IN parameter and use it
            if(navigateToSD("CLK_IN(MHZ):", 12)){
                memset(charBuffer, ' ', 15);
                myFile.read(charBuffer, sizeof(charBuffer));
                int val = atoi(charBuffer);
                Serial.print("  Clock In: ");Serial.println(val);
                MHZinCLK=val;
            }
            //Navigate to CLK_OUT parameter and use it
            if(navigateToSD("CLK_OUT(MHZ):", 13)){
                memset(charBuffer, ' ', 15);
                myFile.read(charBuffer, sizeof(charBuffer));
                int val = atoi(charBuffer);
                Serial.print("  Clock Out: ");Serial.println(val);
                MHZoutCLK=val;
            }
            //Navigate to DHCP parameter and use it
            if(navigateToSD("DHCP:", 5)){
                if(myFile.read()=='E'){
                    Serial.print("  DHCP: ");Serial.println("Enabled");
                    DHCP_ENABLE = true;
                }
            }
        }
    }
}
```

```
    }
    else{
        Serial.print("  DHCP: ");Serial.println("Disabled");
        DHCP_ENABLE = false;
    }
}
myFile.close();
Serial.println(F("Setup Values Input"));
}
//If file can't be opened, print error
else{
    Serial.println(F("Couldn't open file for reading"));
}

}
}

//Naviagte to function for SD card
boolean navigateToSD(char* token, int N){
    myFile.seek(0);
    int foundChars = N;
    while(foundChars > 0 && myFile.available()){
        foundChars=N;
        for(int i =0; i<N;i++){
            char c = myFile.read();
            if(c==token[i]){
                foundChars--;
            }
            else if(c==token[0]){
                foundChars=N-1;
                i=0;
            }
            else{
                break;
            }
        }
    }
    if(!myFile.available()){
        return false;
    }
    return true;
}
```

```
//Library defining functions needed to set up the DDS
```

```
#ifndef Setup_h  
#define Setup_h
```

```
#include "PLL.h"  
#include "Profile.h"  
#include "DRG.h"  
#include "RAM.h"  
#include "Ramper.h"
```

```
//Globals
```

```
extern DRG myDRG;  
extern RAM myRAM;  
extern Ramper sweeper;  
extern Profile profile0;
```

```
boolean setupPLL(PLL& myPll, int inMHz, int outMHz);  
boolean setupCommunications();  
void set4Wire();  
void resetSPI();  
void DDSdefaultReset();  
void PLL_Lost_Lock();  
void setPinModes();  
void fullSetup();
```

```
#endif
```

```
#include "Setup.h"
#include "PinDefinitions.h"
#include "Definitions.h"
#include "LCD_handler.h"

//Function that sets up the PLL clock settings using a reference
// a pll object, the input clock (MHz) and the desired output clock (MHz)
boolean setupPLL(PLL& myPll, int inMHz, int outMHz){

    //reset SPI communications to ensure accurate communication
    resetSPI();

    //Call quickSetup function of PLL class that peforms the real action
    myPll.quickSetup(inMHz, outMHz);

    //Update the values so they take effect
    update();
    Serial.print(F("PLL..."));

    //Clear communication again (purely precautionary)
    resetSPI();

    //Check if PLL has locked
    if(myPll.isLocked()){
        //If Locked, attach interrupt pausing all action if lock is lost
        Serial.println(F("Locked!"));
        delay(1);
        attachInterrupt(PLL_LOCK, PLL_Lost_Lock, FALLING);

        //Confirm successful setup of PLL
        return true;
    }
    else{
        //If not locked, return false
        Serial.println(F("Failed - Check Clock/Clock Settings"));
        return false;
    }
}

//Function to be performed if PLL lock is lost after setup
//Blocks until lock is reaquired
void PLL_Lost_Lock(){
    Serial.print("PLL Lock Lost...");
    while(!digitalRead(PLL_LOCK)){
        myLCDprint("PLL LOCK", "LOST!");
    }
    Serial.println("Reaquired");
}

//Function that sets up communications between the
//Arduino and the AD9910
boolean setupCommunications(){
    //Start SPI and set it to desired protocol
```

```
Serial.println(F("Starting SPI"));
SPI.begin(DDS_CS); //Start SPI on chip select pin for AD9910
SPI.setDataMode(DDS_CS,SPI_MODE0); //Set to Mode0 (look it up on wikipeda)
SPI.setBitOrder(DDS_CS,MSBFIRST); //Set communications to MSB first
SPI.setClockDivider(DDS_CS,8); //Set clock divider up at 8 [84Mhz/8 ~ 10MHz ~ 10Megabits/s]

//Reset the DDS to default settings so it is in a known state
Serial.print(F("Resetting DDS to default settings..."));
DDSdefaultReset();
Serial.println(F("Done"));

//Establish Communications with AD9910
Serial.print(F("Communications... "));
set4Wire(); //Function to put AD9910 in wire SPI mode
resetSPI(); //Reset physical communications (precautionary)

//Check to ensure AD9910 has been set correctly
CFR1.read(); //Read values from CFR1
if(CFR1._data[0]==0&& CFR1._data[1]==0&&
    CFR1._data[2]==0&& CFR1._data[3]==0x02){
    //If values are correct, return true
    Serial.println(F("Success!"));
    return true;
}
else{
    //Otherwise return false
    Serial.println(F("Failed!"));
    SPI.end();
    return false;
}
}

//Function to set AD9910 into 4-wire mode after defaults have been reset (default is 3-wire)
void set4Wire(){
    //CFR1 values needed for 4 wire mode
    byte set4Wire[4] = {0x00,0x00,0x00,0x02};

    resetSPI(); //Reset physical communications (precautionary)
    CFR1.writeReg(set4Wire); //Write values to buffer
    update(); //Update buffer values into register
}

//Physical Communications reset function
//clears buffers of AD9910 SPI reciever [NOT pre-register buffer values]
void resetSPI(){
    digitalWrite(IO_RESET,HIGH);
    delay(1);
    digitalWrite(IO_RESET,LOW);
    delay(1);
}
```

```
//Reset Function
//Resets DDS to DDS default settings
void DDSdefaultReset () {
    digitalWrite(MASTER_RESET,HIGH);
    delay(1);
    digitalWrite(MASTER_RESET,LOW);
    delay(1);
}

//Set pin modes to correct values
//PINS ON ARDUINO DO NOT WORK IF THEY HAVE NOT BEEN SET
void setPinModes () {
    Serial.print(F("Setting pin modes... "));

    //Lock Profile into Profile 0
    pinMode(P0,OUTPUT);
    digitalWrite(P0,LOW);

    pinMode(P1,OUTPUT);
    digitalWrite(P1,LOW);

    pinMode(P2,OUTPUT);
    digitalWrite(P2,LOW);
    //-----

    pinMode(SWEEP_RESET_TRIGGER,INPUT);
    pinMode(SWEEP_TRIGGER,INPUT);

    pinMode(FAUX_TRIGGER,OUTPUT);
    digitalWrite(FAUX_TRIGGER,LOW);

    pinMode(MASTER_RESET,OUTPUT);
    digitalWrite(MASTER_RESET,LOW);

    pinMode(26,OUTPUT);
    digitalWrite(26,LOW);

    pinMode(LCD4, OUTPUT);

    pinMode(IO_UPDATE,OUTPUT);
    digitalWrite(IO_UPDATE,LOW);

    pinMode(DROVER,INPUT);

    pinMode(DRCTL,OUTPUT);
    digitalWrite(DRCTL,LOW);

    pinMode(DRHOLD,OUTPUT);
    digitalWrite(DRHOLD,LOW);

    pinMode(PLL_LOCK,INPUT);

    pinMode(IO_RESET,OUTPUT);
```

```
digitalWrite (IO_RESET, LOW);

delay (1);
Serial.println ("Done");
delay (1);
}

//fullSetup function runs all setup functions in the correct order to set up AD9910
void fullSetup () {

    Serial.println (F ("Setting up device for DDS control"));
    setPinModes ();

    int count = 0;
    while ( count < 3 && !setupCommunications ()) {
        count++;
    }

    if (count==3) {
        Serial.println (F ("Connections Failed"));
        while (true) {}
    }

    resetSPI ();
    PLL pll = PLL ();
    if (!setupPLL (pll, MHZinCLK, MHZoutCLK)) {
        Serial.println (F ("PLL Lock Failed!"));
        while (true) {}
    }
    Serial.print (F ("Creating Profile Controller..."));
    profile0 = Profile (SINGLE_TONE_0);
    Serial.println (F ("Done"));

    Serial.print (F ("Creating DRG..."));
    myDRG = DRG (5);
    Serial.println (F ("Done"));

    Serial.print (F ("Creating RAM controller..."));
    myRAM = RAM (CFR1, RAM_PROFILE_0, RAM_WRITE);
    Serial.println ("Done");

    Serial.print (F ("Creating Sweeper..."));
    sweeper = Ramper (myDRG, myRAM);
    Serial.println (F ("Done"));

    Serial.println (F ("Setup Complete"));
}
```