Abstract
This is meant to be a guide to the construction and implementation of Arduino control of an AD 9914 DDS evaluation board for generation of linear frequency sweeps. This is version 1.0 and will be incorporated into a larger guide as the full chirped pulse FTMW instrument that utilizes this chirp system is completed. Further information regarding the use of the 9914 can be found in Review of Scientific Instruments (2013) 083104, 84.

Throughout this guide, several pins/jumpers are referenced, these are generally referenced in the AD9914 evaluation board guide/schematic. The board is generally well labeled, but the board schematic should be consulted if there is any confusion.

Important: Several critical steps in using the DDS has been omitted from this guide. We found it necessary to remove a single 0 Ω resistor from the evaluation board in order to enable the necessary external triggering of the board. The DDS should also be externally clocked, so a 4 GHz clock is required. A simple PLL is available to handle this. Please see the supplemental section of the RSI paper above for details and instructions. This was initially omitted as it does not directly involve the Arduino control, and will be added in later versions.

Updated: December 7, 2015

1. Short Version

This is the quick-start version of the material below. It is recommended that anyone setting up Arduino control of an AD9914 read the entire guide.
1. Setup the AD9914 evaluation board including external clock, as outlined in Rev. Sci. Instr., (2013), 083104, 84
2. Wire the SDIO/SCLK/IO_UPDATE/RESET outputs to the Arduino as shown in Figure 5
3. Connect these wires to their counterparts on the AD9914 shown in Figure 4
4. Compile/Load the Arduino code given below onto the Arduino
5. Set pints P204 and P203 to disable, P205 to enable, shown in Figure 2
6. Short the signal of EXTPDCTL and DRHOLD to ground using a jumper, shown in Figure 3
7. Power on the AD9914 and press the reset button on the Arduino
8. Supply a trigger as outlined in the RSI supplement and monitor the chirp output

2. AD9914 DDS

To do this we chose an Arduino Uno micro controller board (http://arduino.cc/en/Main/ArduinoBoardUno). The board is cheap, effective, open source, its software is available on virtually any platform, it has an immense support community, and there are Python libraries for external control.

3. Construction

3.1. Microcontroller Control of the AD9914

Unlike the PLL synthesizers that can be set and then ignored, or changed fairly rarely, the DDS requires constant attention. While developing the system, we noticed that the phase of the chirps coming from the DDS would slowly drift (This drift is a function of the number of chirps generated, not time), causing our time domain signal to average to something very low. The solution to this is to recalibrate the DDS’s digital to analog converter (DAC) approximately every second (Because this is dependent on the number of chirps, not time, this number may need to be lower, 1 - 100ms for very high acquisition rates). This requires sending a DAC calibrate command to the DDS very frequently, something that is not practical with the AD (Analog Devices) DDS control software. The solution is to use a micro controller to send serial commands to the DDS to recalibrate the DAC on a regular basis. A second benefit of this is that the micro controller can be programmed to handle the initial setup of the DDS, which simplifies its use.

The first step is to disable USB control/enable serial control of the DDS. This is done by changing jumpers P203 - 204 to disable. Oddly P205 should remain enabled. This is because the digital ramp status pins, DRSTART and DROVER, the trigger signal to start the chirp and the output that the chirp has completed (which should be used as the digitizer trigger to minimize phase jitter) respectively, are buffered through the IC associated with P205, and disabling it disables these as well. When this is done, the USB control lines are set to high impedance to prevent the USB controller accidentally sending commands to the AD9914. The side effect of this is that several pins that were being controlled are now floating. Specifically the reset, SYNCIO, external power down, and chip select are floating, setting them to enable, which makes the board unusable.

To fix this, the signal line on following pins should shorted to ground with a jumper to disable them: SYNCIO (Figure 4), and EXTPDDNCTL (Figure 3), highlighted in red. Once this is done, the Arduino can be used to control the 9914.

An overview of the Arduino SPI protocol is given (http://arduino.cc/en/Reference/SPI). Briefly, SPI is a simple serial communication protocol supported by the Arduino. When using SPI, several Arduino pins are automatically controlled. Pin 13 is set as the serial clock, and should be linked to the CLK/MIO1 pin on the 9914. Similarly, Pin 12 and Pin 11 are the Master In Slave Out (MISO, essentially data transfer from the 9914 to the Arduino, this feature is not currently used, but could be added to read the state of the 9914 if desired) and Master Out Slave In (MOSI, essentially the pin used to write commands to the 9914) respectively. Pin 12 is not used, but Pin 11 is, and should be connected to the signal line of MP102. Aside from these mandatory pins, two additional pins are needed to send reset and IO_UPDATE commands to the board. For this project, pins 7 and 4 have been arbitrarily chosen, but any digital pin on the Arduino not already in use is acceptable. The first pin, the reset pin, can be connected to either of the RESET pins on the 9914. Pin 4 is used to send IO_UPDATE commands to the 9914, and should be connected to the IO_UPDATE BUF pin on the 9914. A full pinout is given
Figure 2: USB micro controller pins. P204 and P203 should be disabled, P205 should remain enabled to allow external chirp triggering.

Figure 3: EXTPDCTL, RESET, DROVR, DRHOLD, DRCTL control pins. EXTPDCTL must have the signal shorted to ground.

Finally, the ground pin of the reset, MISO/SDIO, SNYC_CLK, and IO_UPDATE on the AD9914 should be connected to ground on the Arduino.
Once the Arduino is connected, the board can be controlled. The complete code is given below, but a brief explanation is also warranted. The AD9914 is controlled by a set of registers that determine its behavior. Serial commands can be sent to control these register values. Serial control of the 9914 is default most significant bit (MSB) first. This is also default for the Arduino. The serial commands are sent in two phases. First a register address is sent, followed by a series of four bytes. A helpful diagram is given in the 9914 chip guide. Once the 9914 is powered the first command should be a reset command. This is not spelled out in the documentation, but the reset command seems to be necessary to ensure good communication between the boards. Once this is done, the initial register commands can be sent and the registers can be set. The DAC calibration is then sent every second until the Arduino is powered off. One final note is the IOMUX UPDATE command. The IOMUX UPDATE command is used to set registers once the command bytes are written. To set registers that control the function of the 9914 the bytes are first written to the 9914, this is the SPI Transfer command. Once the register information is set, it is held in a buffer, but the registers will not be changed until an IOMUX UPDATE command is sent.

![Figure 4: MIO buffers, CS (red box top), SYNC_CLK (Green box top), Serial Clock in (Blue box top) and SYNC_IO (red box bottom), IOMUX UPDATE (Yellow box bottom), and RESET (White box bottom)](image)

The register values can be worked out from the AD9914 chip documentation, but the 9914 eval board software’s debug page is infinitely simpler for accomplishing this. It can also be helpful for testing the board in case of a suspected malfunction.
Figure 5: Wiring diagram for the Arduino Uno. The individual grounding wires (green) for the SYNC_CLK and SDIO are essential to proper function. The IO_UPDATE and RESET pins can use a common ground with no disruption.
3.2. Arduino Code

#include <SPI.h>

/* A program for control of AD9914 DDS using the 9914’s serial interface and an Arduino
Written by Brandon Carroll
Created 1/9/14 */

/*See http://arduino.cc/en/Reference/SPI for an explanation of the SPI protocol used here
The AD9914 chip documentation is also immensely helpful
Code is based on: http://arduino.cc/en/Tutorial/BarometricPressureSensor
Pin labels are given for the AD9914 chip, followed by their EVAL board counterpart, i.e Pin 20 on the AD9914 chip is labeled as D2, and is accessed by connecting to MIO1 on the EVAL board

Assuming an Arduino Uno is in use, in SPI Mode:
− Pin 13 is the SCLK
− Data from the Arduino to the AD9914 is pin 11 (Arduino:MOSI or AD9914:SDIO/Pin 20/D2/MIO1 on the 9914 EVAL)
− Data from the AD9914 to the Arduino (not currently used) is pin 12
  (Arduino:MISO or AD9914:SDO/Pin 19/D3/MIO3 on the 9914EVAL)
− Pin 10 is in use by SPI to select which device is in use, not needed here, but still reserved by the Arduino, MIO0 on the 9914EVAL
− Pin 7 is currently used to send the IO\UPDATE commands to AD9914 Pin 86
− Pin 4 is used to send an initial reset command to the 9914

Of Note:
− The 9914 used most significant bit first (MSB) mode by default. This can be changed, but this code assumes MSB operation */

/***** Serial Communication parameters******/
const int IOUpdatePin = 7;  //The pin used to trigger IO\UPDATE events
const int ResetPin = 4;     //The pin used to send reset commands
const int CSPin = 10;       //The pin used to handle chip select (Not currently used)

/***** Register Addresses******/
//Control function register addresses
const byte CFRAddress[] = {0x00,  //CFR1
                          0x01,  //CFR2
                          0x02,  //CFR3
                          0x03}; //CFR4

//Digital ramp addresses
const byte DigitalRampAddress[] = {0x04,  //Digital Ramp Lower Limit (MHz)
                                   0x05,  //DR Upper Limit (MHz)
0x06, // DR Rising Step (MHz)
0x07, // DR Falling Step (MHz)
0x08}; // DR Step Rate (uS)

// Address for CW frequency tuning word, currently using profile 0 (Test purposes)
// Test code for CW operation
const byte CWToneAddress = 0x0B;
const byte USR0Address = 0x1B;

/∗
======Startup register values======
∗

const byte CFR1Start[] = {0x00, 0x01, 0x60, 0x08};
const byte CFR2Start[] = {0x00, 0x0C, 0x29, 0x00};
const byte CFR3Start[] = {0x00, 0x00, 0x19, 0x1C};
const byte CFR4Start[] = {0x00, 0x05, 0x01, 0x20};
const byte DRGStart[] = {0x00, 0x00, 0x00, 0x00}; // Chirp Start
const byte DRGStop[] = {0x00, 0xA3, 0xD7, 0x0A}; // Chirp stop
const byte DRGStepRise[] = {0x00, 0x00, 0x00, 0x00}; // Chirp Start
const byte DRGStepFall[] = {0x00, 0x72, 0x38, 0x4A};
const byte DRGRate[] = {0x00, 0x01, 0x00, 0x01};
const byte ZeroByte = 0x00;

// Command to enable the DAC Cal,
// should be 0x01XXXXXX, where X is the last 6 digits of CFR4Start
const byte DACCalEnable[] = {0x01, 0x05, 0x01, 0x20};

/∗
======Variables for controlling the 9914======
∗

// Not currently in use, but may be useful if adjustments to the chirp are desired
const int ClockFreq = 4000; // DDS clock frequency in MHz
const float ChirpBandwidth = 2000.0; // In MHz, a 2 GHz chirp
const float ChirpTimeStep = 0.0006; // Time between chirp steps in uS
// this is used for both the rise and fall times
const float ChirpFrequencyStepRise = 9.99999976; // Rising sweep step size in MHz
const float ChirpFrequencyStepFall = 6.971428171; // Falling sweep step in MHz
const float DACCalDelay = 100; // Time in ms between DAC Calibrations

/∗
======Test Tuning Words======
∗

// These can be used to enable a CW tone output for simpler testing
const byte TenMHz[] = {0x00, 0xA3, 0xD7, 0x0A};
const byte FiftyMHz[] = {0x03, 0x33, 0x33, 0x33};
const byte CFR2Test[] = {0x00, 0x84, 0x29, 0x00};

void setup ()
{
  pinMode(IOUpdatePin, OUTPUT); // Set the IO\_UPDATE Pin to output
  pinMode(ResetPin, OUTPUT); // Set the Reset pin to output
  pinMode(CSPin, OUTPUT); // Set the Chip select pin to output (Not currently used)
  digitalWrite(CSPin, HIGH); // Set the chip select pin to high, (Not currently used)
  SPI.begin(); // Start SPI mode on the Arduino
  SPI.setClockDivider(SPI_CLOCK_DIV2); // Set the SPI clock divider to 2,
  // The default is 4, however the DDS/Arduino
  // can handle 2, and it speeds up data transfer
  SPI.setBitOrder(MSBFIRST); // Set the SPI to MSB first, this is the default
  SPI.setDataMode(SPI_MODE0); // Set the SPI to use clock polarity and clock phase 0.
Send_Reset(); //This is suprisingly very imporant. Even after power cycling,
   //a reset command is needed
Initialize_DDR();
Calibrate_DAC(); //Do an initial DAC calibration

/* Test code to check the functionality of the the DDS.
It should set the DDS to output a 10 MHz tone
Write_Register (CWToneAddress, TenMHz[0], TenMHz[1], TenMHz[2], TenMHz[3]);
Write_Register (0x1A, 0xC0, 0x00, 0x00, 0x00); //Set the phase/amp
Send_IO_Update ();
*/

void loop ()
{
  delay (DACCalDelay); //Wait for some amount of time between DAC  
   // cals so we don’t spam the DDS and lock it up
  Calibrate_DAC (); //Send a DAC cal
}

//Register writing function
void Write_Register (byte Register, byte Value1, byte Value2, byte Value3, byte Value4)
{
  digitalWrite (CSPin, LOW); //Set the chip select to low,
     //to tell the DDS we want to send commands
  SPI.transfer (Register); //The 9914 expects a register address first
  SPI.transfer (Value1); //Followed by the register value
  SPI.transfer (Value2); //Registers are 4 bytes, but we can only send 1 at a time
  SPI.transfer (Value3);
  SPI.transfer (Value4);
  digitalWrite (CSPin, HIGH); //Set the CS pin high, so the DDS will now
     //ignore anything on the line
}

//Command for sending the IO_Update needed to set the register
//values that have been written
void Send_IO_Update ()
{
  digitalWrite (IOUpdatePin, HIGH); //The IO\UPDATE is active on logic high
     //so we send the pin high for an update
  digitalWrite (IOUpdatePin, LOW); //Then low to end it.
     //The delay between rise and fall is many clock cycles,
     //so no delay between the high and low commands is needed
}

//Command to send a reset
void Send_Reset ()
{
  digitalWrite (ResetPin, HIGH); //The reset is active on logic high,
     //so we send the pin high for an update
  digitalWrite (ResetPin, LOW); //Then low to end it.
     //The delay between rise and fall is many clock cycles,
     //so no delay between the high and low commands is needed
void Calibrate_DAC()
{
    // The DAC cal is a two step process, first we must set the DAC cal bit to 1 in CFR3, then send the IO\_UPDATE to initiate it
    Write_Register(CFRAddress[3], DACCalEnable[0], DACCalEnable[1], DACCalEnable[2], DACCalEnable[3]);
    Send_IO_Update();
    // Then we set the DAC cal bit back to low, to prevent the board from repeatedly calibrating. Finally an IO\_UPDATE to set the bit
    Write_Register(CFRAddress[3], CFR3Start[0], CFR3Start[1], CFR3Start[2], CFR3Start[3]);
    Send_IO_Update();
}

void Initialize_DDS()
{
    Send_Reset();
    // Write the initial control registers
    Write_Register(CFRAddress[0], CFR1Start[0], CFR1Start[1], CFR1Start[2], CFR1Start[3]);
    Write_Register(CFRAddress[1], CFR2Start[0], CFR2Start[1], CFR2Start[2], CFR2Start[3]);
    Write_Register(CFRAddress[2], CFR3Start[0], CFR3Start[1], CFR3Start[2], CFR3Start[3]);
    Write_Register(CFRAddress[3], CFR4Start[0], CFR4Start[1], CFR4Start[2], CFR4Start[3]);
    // Write the USR register, not sure if this is needed
    Write_Register(USR0Address, 0xE2, 0x00, 0x08, 0x00);
    // Set the start frequency for the DRG
    Write_Register(DigitalRampAddress[0], DRGStart[0], DRGStart[1], DRGStart[2], DRGStart[3]);
DRGStart[1],
DRGStart[2],
DRGStart[3]);

// Set the the stop frequency for the DRG
Write_Register (DigitalRampAddress[1],
DRGStop[0],
DRGStop[1],
DRGStop[2],
DRGStop[3]);

// Set the the DRG rising step size
Write_Register (DigitalRampAddress[2],
DRGStepRise[0],
DRGStepRise[1],
DRGStepRise[2],
DRGStepRise[3]);

// Set the the DRG falling step size
Write_Register (DigitalRampAddress[3],
DRGStepFall[0],
DRGStepFall[1],
DRGStepFall[2],
DRGStepFall[3]);

// Set the the DRG step rate
Write_Register (DigitalRampAddress[4],
DRGRate[0],
DRGRate[1],
DRGRate[2],
DRGRate[3]);

// Send the update to set the control registers
Send_IO_Update ();
}