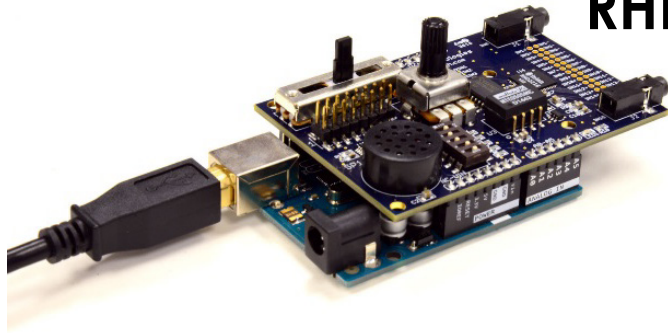# intan
TECHNOLOGIES, LLC

1111001010001011100

RHD2216-SHIELD

RHD2216 Arduino Shield

3 February 2017

Features

♦ Simple plug-in connection to Arduino Uno or Due microcontrollers (not included).

♦ Open-source Arduino C++ software and hardware expansion ports allow users to develop custom applications.

♦ Easy access to two amplifier channels through 3.5 mm phone jacks. Electrode leads with snap connectors included.

♦ CMOS-level SPI (serial peripheral interface) bus communication with Intan RHD2216 amplifier chip.

♦ Solder holes offer access to an additional 14 amplifier channels and other auxiliary I/O.

♦ Software-reconfigurable amplifier bandwidth settings.

♦ Included software implements a 1 kHz per channel sampling rate, suitable for EMG and ECG signals.

♦ On-board two-channel digital-to-analog converter (DAC) can reconstruct analog waveforms from two amplifier channels in real time with sub-millisecond latency.

♦ On-board configuration switches select 50/60 Hz notch filter operation, audio output, and total gain.

♦ Activity detection digital output pins signal when the input signal level is above a user-selectable threshold.

♦ Output signals and unused Arduino pins routed to headers for expansion.

♦ Built-in speaker and volume knob optionally play channel-specific sounds when large signals are detected (e.g., EMG muscle contractions).

Description

The **RHD2216 Arduino Shield** allows users to monitor biopotential signals (typically EMG or ECG) from two low-noise amplifier channels using the RHD2216 digital electrophysiology chip from Intan Technologies. An **Arduino Uno** or **Due** microcontroller (not included) is connected to the shield via a standard header. Included electrode leads with standard snap connectors can be plugged into the two on-board 3.5 mm phone jacks. Free **open-source Arduino C++ software** from Intan allows users to simultaneously monitor two EMG or ECG signals. Large signal amplitudes on these two channels can trigger both sound cues and digital activity detection outputs.

The shield features an on-board **Intan RHD2216 amplifier chip**, and provides an interface for the Arduino Uno or Due to control the chip. Users can reconfigure the settings of the interface through **on-board switches**, **slide and rotary potentiometers,** and **C++ software.** When using the included software, the switches control notch filter settings (which can be used to remove 50 or 60 Hz power line noise), audio enabling, output modes (the system can output individual samples from each channel or the average energy per 20 ms period), and an option for scaling the gain to accommodate large signals.

The primary function of the shield is to stream digitized electrophysiology signals from the RHD2216 chip to the host computer. Data can be sent through the Arduino's **USB Serial Port** and can also be sent as analog voltage signals through the on-board two-channel DAC.

Additionally, the shield can perform basic **activity detection**, in which a digital output signal is toggled when a channel increases or decreases in activity. The detection threshold is controlled with an on-board slide potentiometer, allowing the user to adjust the sensitivity of the activity detection in real-time. As a demonstration of activity detection, the on-board speaker is used in the open-source software to play a different sound when each of the two channels surpasses the activity threshold.
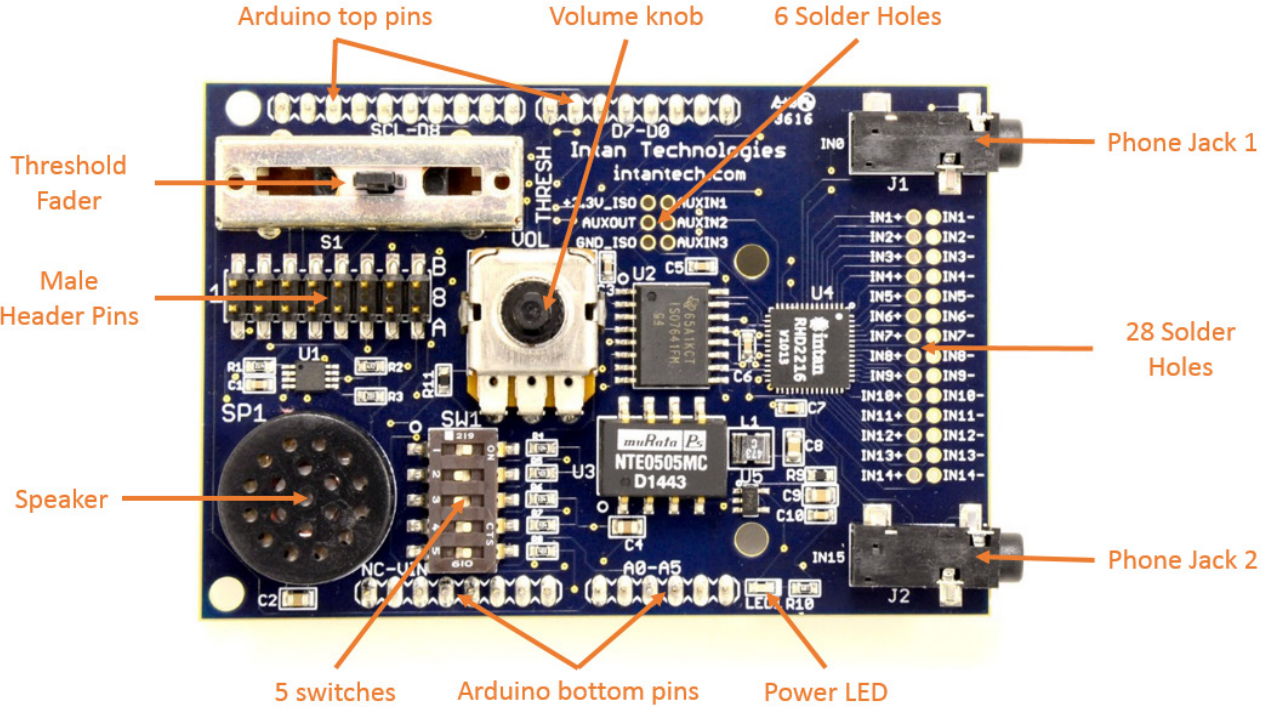
RHD2216 Arduino Shield Board

Figure 1. RHD2216 Arduino Shield Board with user interaction parts labeled

Arduino Shield Header Pins

The Arduino microcontroller has two distinct rows of female header pins. To connect the RHD2216 Arduino Shield Board, ensure that the **top row of male pins** (18 pins) is positioned above the top row of the Arduino's female headers, and that the **bottom row of male pins** (14 pins) is positioned above the bottom row of the Arduino's female headers as shown below. Taking care to avoid bending the male pins, align them and slide them into their corresponding female pins, mechanically securing the shield to the Arduino board. This connection supplies power to the shield and routes digital and analog signals to and from the Arduino. If the shield is properly connected to a powered Arduino, the green **Power LED** will light up.
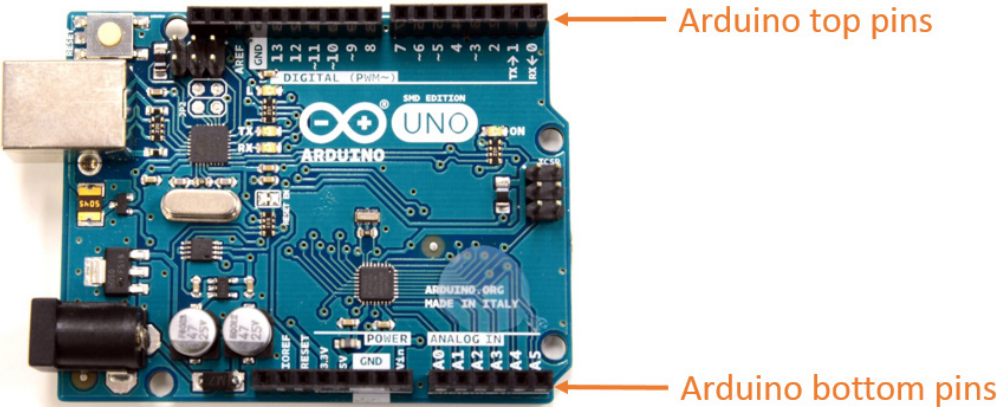

Figure 2. Arduino Uno with top and bottom pins labeled

Output Signal Pins

Besides the pins that connect to the Arduino, the shield also features an array of 8x2 **male header pins** with a pitch (pin separation) of 0.1 inches. These pins can be connected to in a variety of ways, including female jumper wires, soldering of male jumper wires, or direct clipping to an oscilloscope probe.
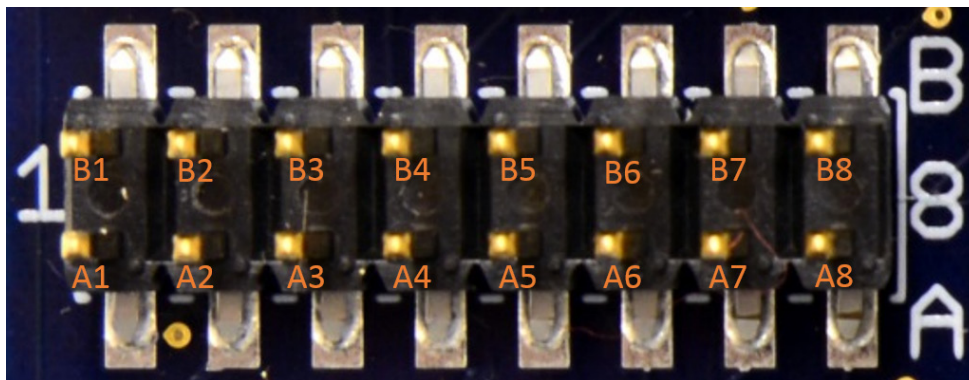

Figure 3. Male header pins

Signals from both the Intan shield and the Arduino are routed to these headers, and important output signals are assigned the following pins:

A1: <u>DAC0</u>: *Output* from channel 0 of the Digital to Analog Converter (DAC). By default, this corresponds to the signal on Phone Jack 1 measured by the RHD2216.

B1: <u>DAC1</u>: *Output* from channel 1 of the Digital to Analog Converter (DAC). By default, this corresponds to the signal on Phone Jack 2 measured by the RHD2216.

A2: <u>GND</u>: *Ground* with reference to the Arduino's power supply.

A4: <u>D4</u>: Digital 4, an *Activity Detection* pin. High when the electrode connected to Phone Jack 1 senses activity above the threshold.

B4: <u>D5</u>: Digital 5, an *Activity Detection* pin. High when the electrode connected to Phone Jack 2 senses activity above the threshold.

B8: <u>GND</u>: *Ground* with reference to the Arduino's power supply. (Connected to pin A2.)

The other pins have uses for developers interested in expanding the shield. See *On-Board Male Header Pins* under "Expanding the Arduino Shield" for more details.

Phone Jacks

Electrodes should be connected to **Phone Jacks 1 and 2** (see Figure 1). These jacks mate with 3.5 mm (1/8") male connectors, and are also commonly known as audio jacks. Three electrodes mate to every connector, corresponding to in+, in-, and ground. The RHD2216 measures the potential difference between in+ and in- over a frequency range set by the software. The ground electrode established a common DC potential between the board and the tissue. Figure 19 at the end of this document shows the Intan Arduino Shield with the included electrode leads attached.

Phone Jack 1 is connected to channel 0 on the RHD2216 chip, and Phone Jack 2 is connected to channel 15. The software addresses this remapping by referencing the #define FIRSTCHANNEL as input from Phone Jack 1, and SECONDCHANNEL as input from Phone Jack 2 in the library header file IntanShield.h.

On-Board Switches

Five switches on the shield allow the user to configure various settings on the fly without having to reset the Arduino. When used with the included software, they have the following functions and will take effect within 1 ms of flipping the switch:

1: **Audio_en:** When in the on position, enables audio to play through the speaker. Note that even when enabled, the sound may be hard to hear if the volume knob is turned too low. When off, disables audio completely.

2. **Low_gain_mode**: When on, scales the output signals down by a factor of four. Reducing the gain allows larger input signals to be observed without clipping on the DAC, and can be helpful if large signals are present on the electrodes. (The gain of the RHD2216 in not changed, but the signals sent to the DACs are scaled down.) Depending on the quality of the electrode connections and the spacing of the electrodes, this setting may be helpful in viewing strong signals like ECG (see *Viewing Output* for details). When off, leaves the output signals at their default scaling. If the output waveforms do not appear clipped, there is no need to enable this setting. Total gain between input (electrode) voltage and output (DAC) voltage when enabled is 750 V/V (57.5 dB). Total gain when disabled is 3000 V/V (69.5 dB). *Note*: If the user desires finer control over the gain, this can be changed through software; see *Output Scaling* for more details.

3. **Average_energy_mode**: When on, sends the average channel energy for every 20 ms period through the DAC and Serial port. When off, sends the electrode waveform (sampled at 1 kHz) through the DAC and Serial port. The activity detection algorithm is based on the average channel energy calculated in 20 ms periods; enabling this setting is useful for monitoring the input to the activity detector. If the user simply wishes to monitor the channel data directly, this setting should be disabled to improve temporal resolution. See *Activity Detection and Thresholding* for more details

4. **Notch_en**: When on, enables the software notch filter. This filter attenuates signals at a specific frequency (either 50 or 60 Hz by default, selected with switch 5), and is intended to minimize interference due to coupling from power wires. If the user needs to reduce CPU load, this setting should be disabled (see *Managing CPU Time* for more details), otherwise it is recommended to leave this enabled to reduce AC mains interference.

5. **Notch_6050**: When on, the software notch filter attenuates 60 Hz noise. When off, the software notch filter attenuates 50 Hz noise. When the previous switch Notch_en is disabled, this setting is ignored entirely. Power line frequency varies by country, and this setting should be selected to match the local power frequency. Generally, countries in the Americas use a power line frequency of 60 Hz, while in most of Asia, Europe, and Africa, power line frequency is 50 Hz.

Threshold Fader

The **threshold fader** allows the user to set the sensitivity of the activity detector. If the threshold is too low, all signals will be considered activity (including noise), and the digital output signal will always be high. If the threshold is too high, nothing will be considered activity (the signal will always be low). The threshold should be set to an intermediate value so that electrophysiological events of interest like muscle contractions will be detected. The threshold fader allows the user to control this parameter in real-time. See *Activity Detection and Thresholding* for more details. The user can view this parameter in real-time through the Serial port when the Average_energy_mode switch is in the "on" position. See *Viewing Output* for more details.

Audio Hardware

The board also includes a **speaker** and **volume knob**, which produce audio cues when activity surpassing the threshold is detected. If audio is enabled, activity detection on each channel will trigger a channel-specific sound to play through the speaker. By default, activity through Phone Jack 1 will trigger a short A note (440 Hz), and activity through Phone Jack 2 will trigger a short E note (330 Hz). To change the sounds, see *Sound Customization* for more details. The volume knob controls how loud the audio sounds when it plays. Rotating the knob clockwise increases volume, and rotating counterclockwise decreases volume.

Plated Solder Holes

Additionally, the board includes **34 plated solder holes** for further development options. These holes are unnecessary for the basic operation of the shield, but offer connections to the RHD2216 isolated power supply as well as an auxiliary output, three auxiliary inputs, and an additional 14 bipolar amplifier inputs. See *Optional Plated Solder Holes* under "Expanding the Arduino Shield" for more details.

A complete schematic of the shield board is available from Intan Technologies.

Getting Started

Note: This evaluation board is intended for use for engineering development, demonstration, or evaluation purposes only and is not considered by Intan to be a finished end product fit for general consumer use. Persons handling the product must have electronics training and observe good engineering practice standards. As such, the goods being provided are not intended to be complete in terms of required design-, marketing-, and/or manufacturing-related protective considerations, including product safety and environmental measures typically found in end products that incorporate such semiconductor components or circuit boards.

While the board allows developers to customize its operation more fully (See "Expanding the Arduino Shield" for more details), users can quickly get their board up and running with the default configuration by taking the following steps:

Setting up Software

♦ Download and install the free Arduino IDE from www.arduino.cc. When the installation dialog says "Completed", you can close it. Download the free RHD2216 Arduino shield software from the Intan web site, www.intantech.com. Open the operation sketch, IntanShieldOperation.ino in the Arduino IDE.

♦ By default, the only amplifier channel enabled is the channel connected to **Phone Jack 1**; all other channels are powered down. If you wish to amplify the two channels from **Phone Jack 1** and **Phone Jack 2**, locate the line `bool SecondChannelPwr = false` towards the beginning of the file. Comment out this line (i.e., begin the line with two slashes `//` in order to keep the Arduino from executing this command), and uncomment the previous line: `bool SecondChannelPwr = true` (i.e., delete the two slashes at the beginning of the line to tell the Arduino to execute this command). See *Setting Amplifier Power* for more details.

♦ The default amplifier bandwidth is 10 Hz to 500 Hz, which works well with EMG signals. For ECG signals, a 0.1 Hz to 500 Hz bandwidth will perform better. If the user is measuring ECG, locate the `band_setting` variable assignment in the setup() function of the operation sketch. Comment out the assignment `band_setting = LowCutoff10Hz`, and uncomment the assignment `band_setting = LowCutoff100mHz`. Further bandwidth customization options are available; see *Configuring Amplifier Bandwidth* for more details.

Plugging In

♦ Connect the RHD2216 Shield Board to the Arduino, lining up the two rows of header pins and plugging them in. Locate the five switches on the board, and set them to your preferred configuration. The recommended starting settings are:

Switch Number	Position
1	ON
2	OFF
3	OFF
4	ON
5	If in a country with 60 Hz power line frequency – ON If in a country with 50 Hz power line frequency - OFF

Figure 4. Recommended switch settings for beginning operation

♦ Connect the Arduino to your computer through a USB cable.

♦ If you wish to view two channels, plug two 3.5 mm (1/8") terminated electrode connectors into Phone Jack 1 and Phone Jack 2. If you only wish to view one channel, plug one electrode connector into Phone Jack 1.

♦ Each electrode connector should have 3 electrodes. Attach the GND electrode (likely colored black) to a region with little EMG activity, for example, the skin over the bony part of the elbow. Attach the remaining two electrodes an inch or more apart over the region of interest.

Monitoring Signals

♦ Click Tools → Port to display serial ports. One of these will display the name of your Arduino board. Make sure this port is selected before uploading code to your Arduino.

♦ Click "Upload" on the Arduino IDE, and wait for the "Done Uploading" message to appear.

♦ Click Tools → Serial Plotter to see real-time data from the electrodes connected to Phone Jack 1. You may need to select the baud rate of 250000 through a drop-down menu to read the data correctly. If the plotter hangs up and an "exception" error message occurs, simply close the plotter window and reopen it. This is the result of a bug within the Arduino IDE, and it only occurs the first time data is displayed through the plotter. Examples of the Serial Plotter working properly are shown below:
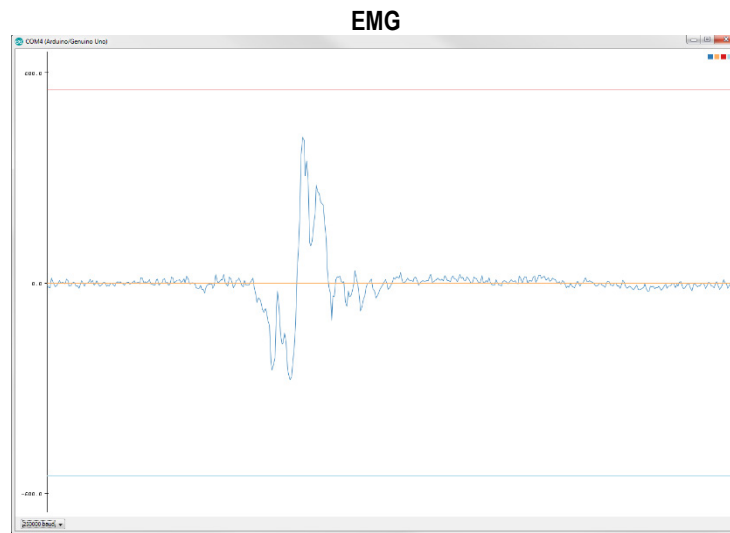
EMG

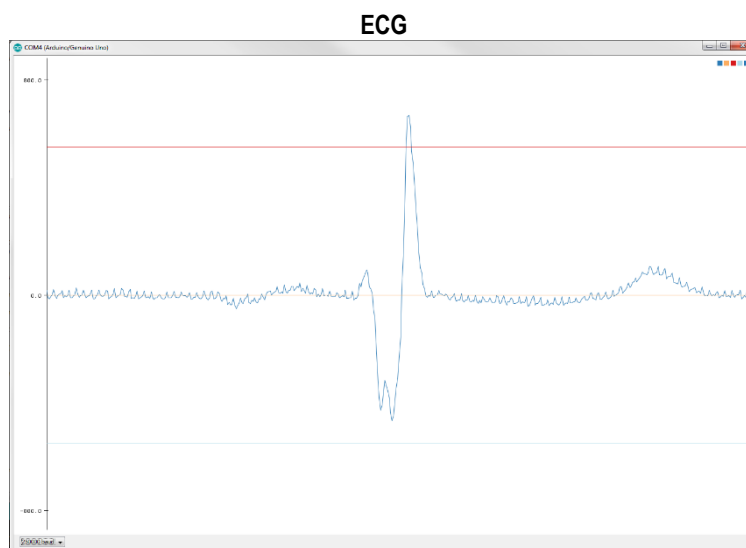**Figure 5**. Example EMG signal viewed with the Serial Plotter

ECG

Figure 6. Example ECG signal viewed with the Serial Plotter

www.intantech.com ● info@intantech.com

♦ Monitor the output signal from DAC0, located on the 8x2 male header pins on the board. This is pin A1 in Figure 3. Connect an oscilloscope probe to A1, clipping the probe's ground to either GND pin (A2 or B8 in Figure 3). The oscilloscope should display the same waveform as the Serial Plotter (though likely with different scaling). Users monitoring two channels can connect a second oscilloscope probe to DAC1 (pin B1 in Figure 3) and view the second channel simultaneously.

♦ Monitor the activity detection signal from Digital Pin 4 (pin A4 in Figure 3). Connect another oscilloscope probe to this pin, clipping the probe's ground to GND (pin A2 or B8 in Figure 3). When the threshold is properly set, a muscle flex (for EMG) or heartbeat (for ECG) should trigger a high signal for the length of the muscle contraction, and otherwise be low. Users monitoring two channels can connect a second oscilloscope probe to Digital Pin 5 (pin B4 in Figure 3).

♦ Locate the threshold fader on the board and adjust the threshold to a suitable value. If the activity detection signal is always low, try lowering the threshold by sliding the fader to the left. If the Activity Detection signal is always high, try raising the threshold by sliding the fader to the right. Examples of the output signal and activity detection signal are shown below:
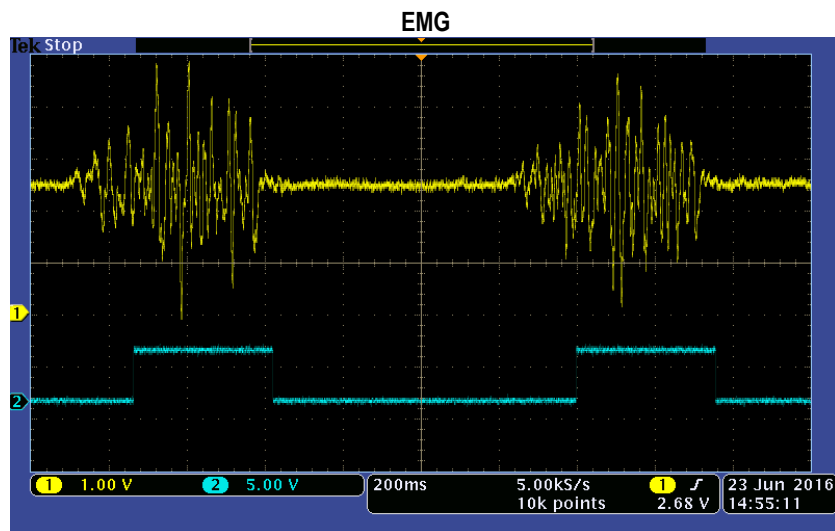

Figure 7. Example EMG signal (during muscle contractions) and activity detection viewed with oscilloscope
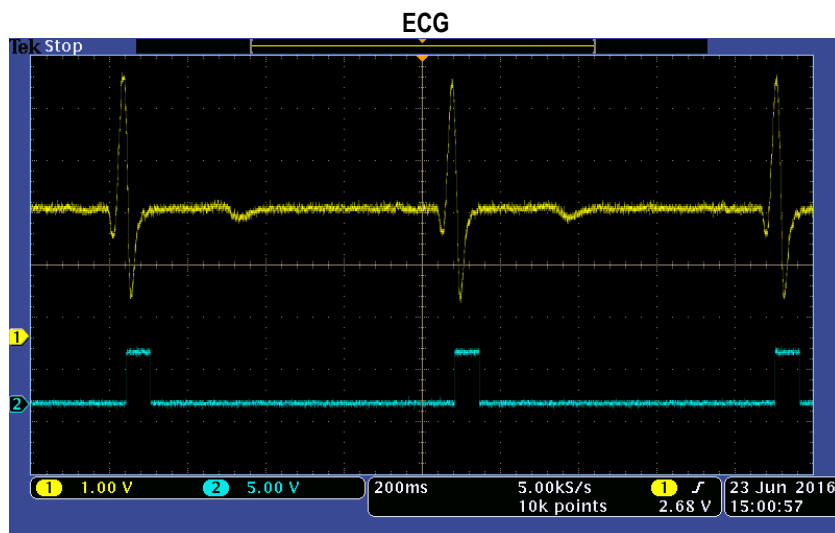

Figure 8. Example ECG signal and activity detection viewed with oscilloscope

♦ Locate the volume knob (next to the speaker) and adjust the volume to a comfortable setting.

♦ Have fun triggering the audio with biopotential signals. If you haven't already, try connecting a second electrode connector to Phone Jack 2 so that you can control two sounds at the same time.

RHD2216 Arduino Shield Software: Detailed Description

The RHD2216 Arduino Shield interface is controlled through an Arduino operation sketch. Some simple changes that users may wish to make can be done by directly editing the main IntanShieldOperation.ino file through the Arduino IDE. However, some changes will require editing of the included library files, which can be done with any text editor. In either case, it is helpful to understand the general program flow before altering the code.

Operation Sketch Overview

Like most Arduino sketches, the operation sketch begins with the **setup()** function after some pre-processor directives and variable declarations. The setup() function initializes the SPI interface (see *Serial Peripheral Interface* section for more details) to allow communication with the RHD2216. After initializing variables, configuring pins as inputs or outputs, and setting up the analog-to-digital converter (ADC) to read analog signals quickly, the Arduino sends various Write commands to the RHD2216. These commands configure settings for the RHD2216 by writing 8-bit values to different registers, and are explained individually with comments in the code. Commands are then used to calibrate and improve performance before the first ADC Convert commands (which actually sample from the amplifiers) are issued. These also recur later in the program, but are first used to initialize the data. Due to pipelining, their order is important, and developers should consult the *Serial Peripheral Interface* section before adding or removing commands. Setup() then initializes the I2C interface (also known as TWI) using the Wire library, so that the Arduino can communicate with the digital-to-analog converter (DAC) to provide output. Setup() ends by configuring Timer 1 to send an interrupt every 0.5 ms (see *Interrupts* section for more details) so that sampling occurs at defined intervals.

Before continuing with the loop() function, it is important to examine the **ISR (Interrupt Service Routine)** because loop() and ISR() will alternate in their execution (see *Interrupts* section for more details on why we use an ISR). The ISR executes every 0.5 ms, and can be thought of as a loop that repeats at well-defined intervals. The ISR begins by reading the values of the digital pins connected to the on-board switches to determine the values of user-selected settings (see *On-Board Switches* for more details). Then the ISR handles audio playback in case a sound started playing during a previous loop. Next, data from a single channel is read from the RHD2216 (the channel being sampled alternates with every loop so that two channels are sampled at 1 kHz each), and notch filter functions will format the data into 8-bit integers with 50 Hz, 60 Hz, or no frequency removed, depending on the switch settings. The channel data is entered into an accumulator which estimates the energy of 20 samples over a 20 ms period. Once 20 samples have been added to each accumulator, Analog Pin A0 is read to determine the position of the threshold slider, and the threshold is calculated. The value of each accumulator is compared to the threshold, and if an accumulator value surpasses the threshold, its activity detection signal is set high and a sound plays if audio is enabled. If an accumulator value is lower than 50% of the threshold, its activity detection signal is set low. ISR() ends by setting variables to appropriate values for the next iteration.

The **loop()** function is simpler than the ISR(), though it is important to remember that the processor stops executing loop() at 0.5 ms intervals to execute the ISR. Similar to the ISR function, loop() begins by reading the values of a digital pin connected to the on-board switch to determine whether the boolean variable average_energy_mode should be true or false. If true, the Arduino's TWI (I2C) registers are configured to send two channels' accumulator values (used in the ISR for activity detection) to the DAC every 20 ms. If false, the Arduino's TWI registers are configured to send two channels' individual sample values to the DAC every 1 ms, corresponding to the 1 kHz sampling rate. Once this setting is initialized, loop() sends the two channels of data to the on-board DAC using Arduino's Wire library, and sends the data to the host computer using the Serial port.

Operation Sketch Tradeoffs

While Arduino microcontrollers are great for general-purpose functionality, ease-of-use, and price, they have some serious limitations that need to be considered. The main drawback of the Arduino for this application is the relatively low clock speed of 16 MHz. This is a problem when we want to sample high frequency signals, because for each sample there are computations that need to be executed before the next sample period starts. This window of time is divided by N when we want to sample N channels,

so if we try to sample many channels we quickly reach the speed limit of Arduino. For this shield, chose a compromise between sampling rate, number of channels, and complexity of computation.

Our software's default sampling rate is 1 kS/s per channel; we sample from two channels and allow some time for necessary computations: communication between Arduino and RHD2216, formatting data to 8-bit format to send to DAC, and communication between Arduino and DAC. Some helpful but non-essential computations are also included: notch filtering to remove noise, ADC sampling to read the user-controlled threshold fader, and serial communication to a host computer to monitor a signal in real-time through the Arduino IDE. This approach offers the advantage of working well for a variety of low-frequency signals like EMG and ECG with several configurable settings, but with the disadvantage of neglecting signals above 500 Hz (the Nyquist frequency with our sampling rate), and only offering two channels for use.

The developer who wants to increase the sampling rate or the number of channels has some options, but likely must sacrifice some of the default functionality of the shield interface, and absolutely must understand the structure of the operation sketch, namely its use of interrupts. For those unfamiliar with interrupts, the following is a brief discussion on interrupts and their application in the shield's operation sketch.

Interrupts

The Arduino has the ability to set **timer interrupts**. If the Arduino's registers are set to start a timer, then it is possible to enable an interrupt every time the timer reaches a certain value. When an interrupt occurs, the processor stops whatever it was working on and jumps to a block of code called an **interrupt service routine (ISR)**. This code is executed, and when it is finished the processor returns to its previous task and continues where it left off. Timer interrupts have the advantage of triggering at reliable intervals more accurately than software functions such as delay(), millis(), or micros() that are more common in Arduino programming. We use this accuracy in order to ensure our 1 kS/s sampling rate really does occur every 1 ms (two channels, each at 0.5 ms).

Another advantage of using interrupts is the ability to run separate blocks of code roughly in parallel. For our sketch we need to sample using the RHD2216 through the Serial Peripheral Interface (SPI) and process all channels of the electrode data every 1 ms, but we also need to send all channels of the data over an I2C interface to a DAC and over the Serial port to a computer every 1 ms. Simultaneously, we need to be able to play a sample of digital audio at exact periods of 0.5 ms. If we did all of this in our main loop() function, we would have no way of reliably making this happen exactly every 1 ms. If we did it all in the ISR() the SPI communication, processing, and I2C communication (which hogs the processor for the full length of period of time the communication lasts) would together take longer than 1 ms. Our solution is to move the I2C and Serial communication to the loop() function and set the Arduino's TWI registers to send each channel of data every 0.5 ms. (Since we have two channels of data, we end up with a total period of 1 ms.) Thus, we use interrupts to both sample at known, reliable rates, and also to keep the processor from being hogged by the fairly slow communication commands.

The main disadvantage of using interrupts in this way is code complexity. The Arduino's timer registers need to be set to the proper values, and important changes to the sketch such as number of channels sampled and sampling rate have to be properly implemented in both loop() and ISR(). While not a disadvantage unique to the interrupt method (the core problem of processing speed would still exist without interrupts), another pitfall to be wary of is the danger of making the ISR() function too processor-intensive. ISRs are intended to be simple blocks of code executed quickly, because if they take too long, the Arduino runs the risk of reaching the next interrupt period before the previous ISR has finished executing. Allowing the timer to attempt to trigger an interrupt before the ISR is finished causes unreliable execution of the ISR as well as an unreliable timer period, compromising the integrity of the RHD2216 data. This is the main concern a developer must address before increasing the sampling frequency or the number of channels read. See *Changing Number of Channels* and *Changing Sampling Frequency* for more details.

In order to monitor how much time the ISR takes compared to the interrupt period, it is helpful to use Digital Pin 2 (B3 on Figure 3). At the start of each ISR, pin 2 is set high, and at the end of each ISR, pin 2 is set low. Thus, by measuring the duty cycle of this signal, the developer gets an estimate of processing time vs. allotted time. For example, when the notch filter is disabled (switch 4 is off), the ISR takes 68 microseconds to execute (and executes every 500 microseconds). Hence, the duty cycle on pin 2 is about 14%. When the notch filter is enabled, the ISR takes about 182 microseconds, so the duty cycle increases to 36%. Due to potential fluctuations in the logic paths and processing efficiency of different ISRs, it is recommended to keep this duty cycle below 80%. If the user changes the processing load of loop() within the .ino file, it may be helpful to use pin 2 to monitor loop() instead of the ISR. Simply comment or delete the digitalWrite(2, HIGH/LOW) statements in the ISR, and place them in loop() surrounding the processor-intensive code of interest to determine how much time the processor is spending on it.

To reinforce the idea of loop() and ISR() executing separately, we have included ISR() separately in the library file IntanShield.cpp despite being as essential as loop(). The developer who intends to change the operation of ISR() must edit the library file and save it before compiling the code through the Arduino IDE.

Activity Detection and Thresholding

While much of the operation sketch focuses on sampling from a channel and performing some basic operations on the data before outputting it, an additional function the shield can serve is to provide **activity detection**. The Arduino takes the sampled data from each channel, accumulates it (similar to averaging, but with some important differences explained shortly) over the course of 20 samples, and compares that accumulator to a threshold. The general idea is to set each channel's activity detection pin high or low if the accumulator is above or below the threshold. For example, if a user measures EMG signals from a muscle, then activity detection would be set high when the muscle contracts, and low when the muscle relaxes.

In order to ensure reliable activity detection under a wide variety of conditions, the actual implementation has to be a bit more complex. The threshold is user-controlled. Since signal strength depends on many variables (e.g., electrode connection quality, electrode spacing, and signal type), different trials can give different magnitudes of signal strength. By setting the threshold from a reading of an analog-to-digital converter (ADC) that measures a voltage from a user-controlled potentiometer, the user can change the threshold over a wide range of values simply by moving the threshold fader left or right. The threshold can be viewed through the Serial Plotter when the board is in average_energy_mode.

Another feature implemented to improve performance is the "low" threshold, set to 50% of the "high" threshold in software. While the overall average magnitude of biopotential signals increases under greater activity, spikes may occur with both positive and negative polarity, and can sometimes cancel out to give an average near zero. This is why the accumulator measures the maximum and minimum samples of an accumulator period rather than average them altogether. Also, by measuring the difference between the maximum and minimum sample of an accumulator period rather than just the maximum, the accumulator calculates the strength of a biopotential event without biasing towards signals with a positive DC offset.

Additionally, in order to keep the system from classifying a long biopotential event as many short back-to-back biopotential events, the accumulator uses an average of maximum difference between maximum and minimum over 20 samples. The activity detection pin cannot be set low until at least one accumulator period has measured an average energy of less than 50% of the threshold. If audio is enabled, this prevents one biopotential event from triggering a sound over and over until the event is over. Instead, one biopotential event triggers one sound.

A final feature used to improve the accumulator's performance is that once the maximum difference between samples in an accumulator period is found, it is compared to the maximum absolute amplitude (referenced to zero) of that accumulator period and assigned the higher of the two values. For high frequency signals, the maximum difference should always surpass the maximum amplitude, so they are unaffected. However, if this was not implemented then low-frequency signals could have very large amplitudes but would only be classified by how much they change over a 20 ms period. Signals below 50 Hz can take multiple accumulator periods to complete a cycle, so if the maximum sample is larger than the maximum difference, the maximum sample is entered into the accumulator instead, allowing low-frequency signals to also trigger activity detection (assuming they are not attenuated by the chip's bandpass filter - see *Configuring Amplifier Bandwidth* for more details).

Viewing Output

The shield supports two ways of viewing output. The first, **DAC output**, sends two output signals (one for each channel), out through the two pins DAC0 and DAC1. These can be viewed through an oscilloscope, and can either show channel data 1 ms sample by 1 ms sample, or as an average of accumulated energy over 20 ms. See "average_energy_mode" in *On-Board Switches* for more details. The DAC output will saturate at maximum or minimum if the input signal is larger than the highest non-clipping input value (in low_gain_mode, 4.4 mV peak-to-peak. Otherwise, 1.1 mV peak-to-peak).

The second method of viewing output, **Serial output**, makes use of Arduino's built-in serial communication library to send data over the USB connection to a host computer which can then be viewed with the Arduino IDE's **Serial Monitor**. Make sure the Serial Monitor is set to receive data at 250000 baud, the highest data transfer rate supported by the Arduino IDE. Serial communication is fairly slow compared to the capabilities of the RHD2216, so the user should avoid adding more print statements than the default calls in loop().

The data read from the RHD2216 chip is multiplied by 0.195 before being sent through the Serial port, so that the output displayed on the Serial Plotter is in microvolts. This scale factor comes from the smallest step size on the RHD2216 chip, which is 0.195 µV. Users can change this scaling if they wish to see signals at a higher or lower level. See *Output Scaling* for details.

The Arduino IDE's Serial Plotter features auto-scaling and zooms in or out in order to show the full waveform being displayed. This may be desired in some applications, but often the user would rather view the signal at a fixed zoom value. The sketch accomplishes this by sending comma-delimited constants along with the channel data in the `Serial.println()` function call. These constants are chosen to correspond to the signal levels which would result in saturation of the DAC. The exact values vary depending on the mode the board is in (low_gain_mode and average_energy_mode). As long as the signal is within these thresholds, the Serial Plotter will display the signal without zooming in.

When the Average_energy_mode switch is on, the user can also use Serial communication to view the **accumulated energy** from two channels over 20 ms. As explained above, constants are chosen to correspond to signal levels that would result in saturation of the DAC. An additional value is displayed in Average_energy_mode, which shows the current value of the threshold. When a signal's accumulated value surpasses that threshold, its corresponding digital signal is high (and if audio is enabled, sound plays).

If the Serial Plotter is used to monitor output, the user should note that the Serial Plotter will continue to retain any data that is not overwritten. This means that switching between modes (using the DIP switches) can confuse the Serial Plotter. In case any unwanted data from the previous mode is retained, simply close and reopen the Serial Plotter. If the problem persists, try uploading the code again to force the Serial communication to reset.

Low_gain_mode is available when input signals are larger than 1.1 mV peak-to-peak, causing the DAC output to saturate and the Serial Plotter to likely auto-scale the output. Enabling low_gain_mode extends the input range to 4.4 mV peak-to-peak by lowering the DAC output by a factor of four. In the case of the Serial Plotter, this mode keeps the units in microvolts but extends the upper and lower constants that are sent through Serial by a factor of four. This means that the outer constant lines, which are intended to keep the Serial Plotter from auto-scaling all signals that lie within them, are four times farther out. In other words, the Serial Plotter zooms out 4x. An example is shown in Figures 9-12 below:
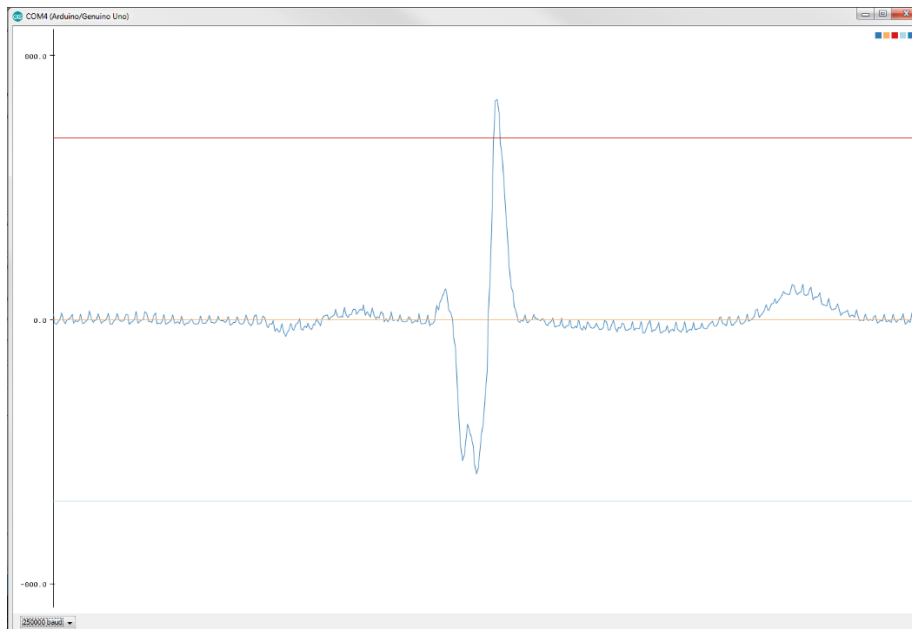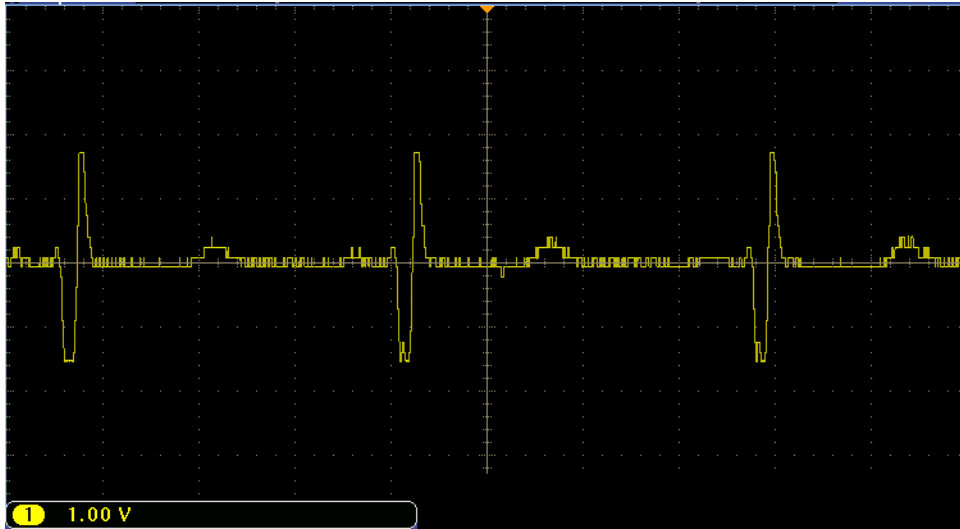

Figure 9. An ECG signal surpasses the boundaries as viewed through the Serial Plotter (low_gain_mode disabled).

Figure 10. Similar ECG signals surpass the DAC output range, resulting in clipping (low_gain_mode disabled).

When the user activates low_gain_mode, both the Serial Plotter and DAC output range can cover signals four times as large. Once this is activated, the resulting waveforms will appear as shown:
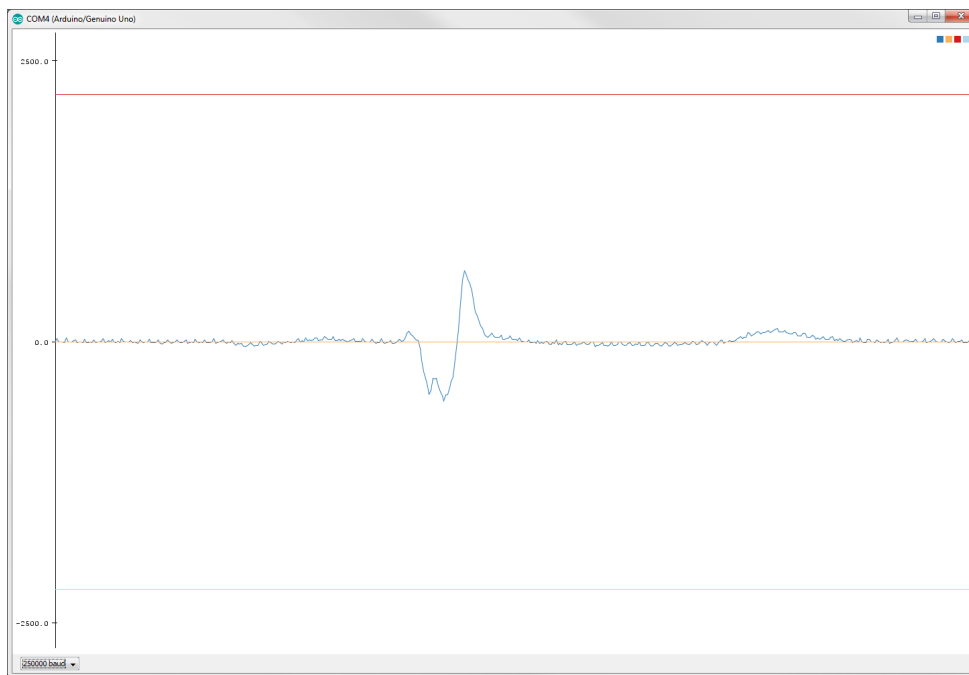
Figure 11. An ECG signal lies fully within the boundary constants when low_gain_mode is enabled.
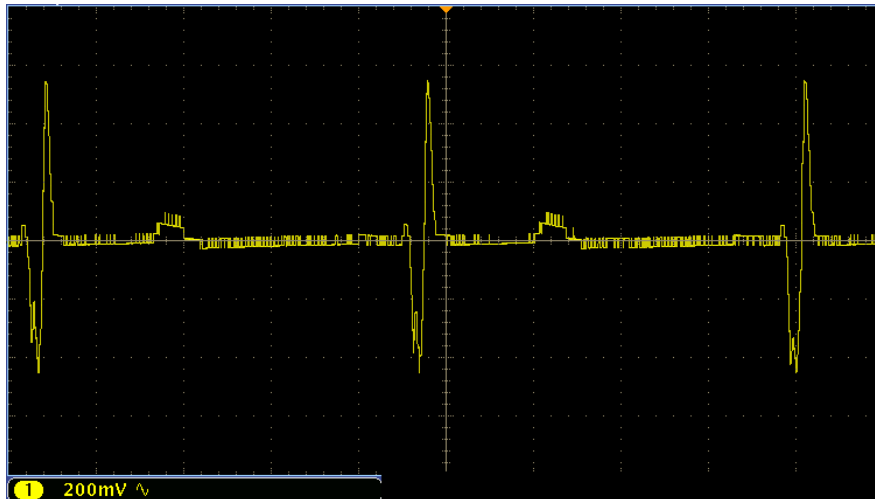
Figure 12. An ECG signal no longer clips when low_gain_mode is enabled.

As illustrated in this example, the constant boundaries of the Serial Plotter indicate the signal values that would result in clipping of the DAC output.

Serial Peripheral Interface (SPI)

The RHD2216 communicates with the Arduino microcontroller through a data exchange protocol known as **SPI**. (The RHD2000 series chip datasheet, available from the Intan website, describes the command structure of this interface in detail.) The included Intan library contains functions such as SendReadCommand and SendConvertCommand that take care of the lower-level formatting and SPI transfers that exchange data between the RHD2216 and Arduino. The RHD2216 uses pipelined SPI commands, so that after command #1 is issued, the result to command #1 is processed during command #2, and only appears as command #3 is being transferred. Three commands must be transferred before the first result is returned.
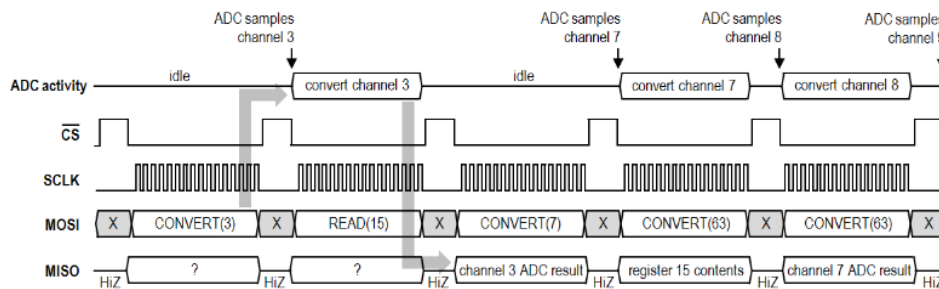

Figure 13. Pipelined SPI communication diagram. MOSI is the command signal, and MISO is the response signal.

The Intan library functions send various commands through SPI, and the developer who wishes to add or remove these functions must consider that their **return value is the result of the function two commands ago**. This may be obscured by the 2-channel operation sketch loop, where the assignment loop alternates between:

A' = SendConvertCommand(A), B' = SendConvertCommand(B)   correct

This happens to always give the expected result, which may obscure how the communication pipeline works. One might erroneously expect that to extend this to three channels:

A' = SendConvertCommand(A), B' = SendConvertCommand(B), C' = SendConvertCommand(C) ⟩ incorrect

However, since the result of a command is always pushed to two commands later, the proper assignment loop would actually be:

B' = SendConvertCommand(A), C' = SendConvertCommand(B), A' = SendConvertCommand(C) ⟩ correct

This is the purpose of sending two Convert commands in the setup() function, beginning the pipeline.

See the RHD2000 series chip datasheet for further information on SPI commands and pipelining.

I2C Interface (TWI)

The digital-to-analog converter (DAC) communicates with the Arduino's processor, the ATmega328P, through a data exchange interface known as **I2C**, or referred to by the ATmega328P datasheet as Two-Wire Interface, or **TWI**. The speed of the I2C communication is determined by Two-Wire registers in the Arduino processor, TWSR and TWBR (see the ATmega328P datasheet for more information). In the operation sketch, the values to be written to TWSR and TWBR are determined by the switch settings (see *On-Board Switches* for more information). Depending on the switch setting, either individual samples or accumulated signals are sent to the DAC for output (see *Activity Detection and Thresholding*). In either case, the communication begins with Wire.beginTransmission(address). With this hardware configuration of the board, the DAC chip's address is 56. In the next byte, we need to describe which DAC (the chip has two built-in DACs, one for each channel) is being sent data, and start sending the data. Eight bits of data must be split up between the four Most Significant Bits (MSBs) and the four Least Significant Bits (LSBs) to be sent through two separate Wire.write() functions per I2C transmission. The on-board DAC, a Maxim MAX5820, accepts input in the following form:

Address Byte: Wire.beginTransmission(56)	0	0	1	1	1	0	0	0
	Address of DAC chip – hardwired to 56							
1st Data Byte Wire.write(data >> 4) OR Wire.write(0b1000 & data >> 4)	DAC	0	0	0	D7	D6	D5	D4
	1 or 0 — Command bits – set to 0				4 MSB's of data to be sent			
2nd Data Byte Wire.write(data<<4)	D3	D2	D1	D0	0	0	0	0
	4 LSB's of data to be sent				Sub bits – set to 0			

Figure 14. Recommended I2C DAC communication diagram.

It is recommended to format all DAC data this way, and only alter the **DAC** bit or **D7-D0**. The "DAC" bit will be 0 if this data will output on pin DAC0, and it will be 1 if this data will output on pin DAC1. D7-D0 are the 8 bits of data describing the voltage level to output through the DAC.

Expanding the Arduino Shield

Some users may wish to alter the operation of the shield more radically than changing the switch settings can accomplish. These developers should gain an understanding of the hardware layout and the software structure before expanding the shield. Once the mechanisms of the operation sketch are grasped, the developer should note their objectives and read the following sections to determine how to proceed.

On-Board Male Header Pins

The developer may already be familiar with some of the pins shown in Figure 3, namely the DAC outputs, activity detection signals, and GND. However, there are an additional 12 signals routed to these pins intended for developers to expand the shield. The full list of signals is included below along with their position with reference to Figure 3:

A1: DAC0: *Output* from channel 0 of the Digital to Analog Converter (DAC). By default, this corresponds to the signal on Phone Jack 1 measured by the RHD2216.

B1: DAC1: *Output* from channel 1 of the Digital to Analog Convert (DAC). By default, this corresponds to the signal on Phone Jack 2 measured by the RHD2216.

A2: GND: *Ground* with reference to the Arduino's power supply.

B2: D0: Digital 0, an *unused* Arduino digital pin that can be accessed normally by the user. *Warning:* aliased as RX, and used in the Arduino for the USB Serial connection. Maintains this specialized functionality in the shield.

A3: D1: Digital 1, an *unused* Arduino digital pin that can be accessed normally by the user. *Warning*: aliased as TX, and used in the Arduino for the USB Serial connection. Maintains this specialized functionality in the shield.

B3: D2: Digital 2, an ISR duty cycle *monitoring* pin that can be used by the software to measure CPU usage, or can be removed from the software and left as an unused pin (see *Interrupts* section for more details).

A4: D4: Digital 4, an *Activity Detection* pin. High when the electrode connected to Phone Jack 1 senses activity above the threshold.

B4: D5: Digital 5, an *Activity Detection* pin. High when the electrode connected to Phone Jack 2 senses activity above the threshold.

A5: D8: Digital 8, an *unused* Arduino digital pin that can be accessed normally by the user.

B5: D9: Digital 9, an *unused* Arduino digital pin that can be accessed normally by the user.

A6: VIN: Input Voltage, an *unused* Arduino pin that can be used to power the Arduino from an external source.

B6: +5V: Voltage Supply, a *regulated* 5V power supply from the Arduino.

A7: +3.3V: Voltage Supply, a *regulated* 3.3V power supply from the Arduino.

B7: IOREF: Logic Reference, a *regulated* 5 (Uno) or 3.3V (Due) power supply from the Arduino. Matches logic level voltage.

A8: NC: Not Connected, an *unconnected* pin that currently has no functionality on the Arduino.

B8: GND: *Ground* with reference to the Arduino's power supply.

Optional Plated Solder Holes

The 34 plated solder holes on the board provide functionality useful for developers interested in expanding the board. The group of **six solder holes** are labeled on the board, and are separated with 2 mm pitch (as opposed to the 0.1-inch pitch of the header pins) and are arranged as follows:
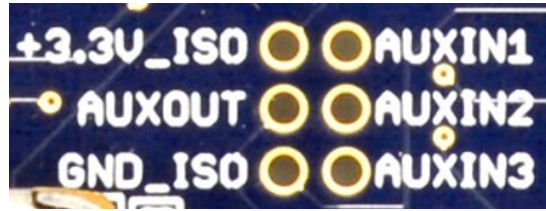

Figure 15. Labeled group of six solder holes.

These holes provide connections that are not essential to the basic operation of the shield, but allow options for developers to use more of the functionality of the RHD2216 chip. They can be accessed like any other plated hole, commonly through male jumper wires, soldered through-hole components, 2-mm-pitch through-hole male header pins, or voltmeter probes. The holes have the following connections:

+3.3V_ISO and **GND_ISO**: Isolated 3.3V supply voltage for the RHD2216. The shield includes a Murata NTE0505MC DC-DC converter and TPS79333 voltage regulator to provide a regulated 3.3V that is electrically isolated from Arduino power and ground. Generally, the right half of the board is the isolated side, providing an isolated ground for the electrode inputs. *Note:* GND and GND_ISO are similarly separated, so +3.3V_ISO should only be measured with reference to GND_ISO.

AUXOUT: Single user-programmable digital output pin which may be used to control an external device via SPI commands. For example, this pin could control the gate of an external MOSFET that enables an LED. If not used, this pin should be left unconnected, and by default the shield does not use this pin. Take care never to tie this pin to ground or power, as the operation of this pin is undefined at power-up. *Note*: For most applications in which a developer wants to control another device and the Arduino and chip are spatially close to each other, it is simplest to use one of the currently unused Arduino digital pins instead. If use of this pin is desired, consult the RHD2000 series chip datasheet for details. This pin should be referenced to GND_ISO.

AUXIN1-3: Auxiliary analog inputs to the on-chip ADC (0.10V-2.45V range). If they are desired as ADC inputs, **auxin1, auxin2,** and **auxin3** may be sampled on ADC channels 32, 33, and 34, respectively. It is recommended to consult the RHD2000 series chip datasheet. These pins should be referenced to GND_ISO.

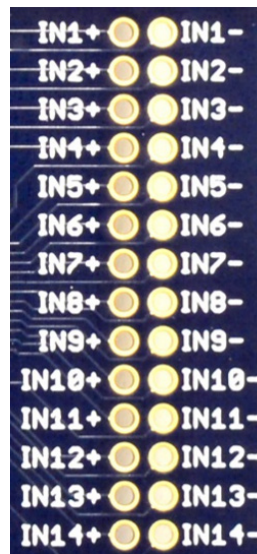A further **28 solder holes** are present on the board, also separated with 2 mm pitch, and are arranged as follows:

Figure 16. Labeled group of 28 solder holes.

These holes allow access to the remaining 14 RHD2216 amplifier channels, beyond the two channels connected by default (Phone Jack 1 connected to IN0+ and IN0-, and Phone Jack 2 connected to IN15+ and IN15-). Sampling from these additional channels is possible, but has a severe impact on sampling rate due to the limited speed of the Arduino. If a developer desires to sample from these channels as well, they must be aware of the tradeoffs that come with extra channels. See *Changing Number of Channels* for more details.

Configuring Amplifier Bandwidth

Setting the bandwidth of the amplifiers can improve signal quality, depending on the types of signals being monitored. The operation sketch comes with three selectable bandwidths: **10 Hz – 500 Hz**, **1 Hz – 500 Hz**, and **0.1 Hz – 500 Hz**. The upper cutoff frequency for all of these bandwidths is set to 500 Hz because the sampling frequency is 1 kS/s. For this sampling rate, 500 Hz is the Nyquist frequency, meaning that signals above 500 Hz cannot be identified uniquely. Therefore, if a higher cutoff frequency is desired, a faster sampling rate is needed (See *Changing Sampling Frequency* for more details). These three bandwidths have different lower cutoff frequencies, and depending on the signals being measured, one may match the signal frequency range better than the others.

By default, the 10 Hz – 500 Hz bandwidth is used, but it can be replaced by another easily. Registers 12 and 13 control the lower cutoff of the on-chip bandpass filter, and are set in the register initialization section of the setup() function. Locate the variable assignment `band_setting = LowCutoff10Hz` and comment or delete this statement. Uncomment either `band_setting = LowCutoff1Hz` if the 1 Hz – 500 Hz amplifier bandwidth is desired, or uncomment `band_setting = LowCutoff100mHz` if the 0.1 Hz – 500 Hz bandwidth is desired. As long as `band_setting` is only assigned once, and it is assigned the value corresponding to your chosen bandwidth, the amplifier bandwidth will be configured to your specifications.

The 10 Hz – 500 Hz bandwidth is most suitable for EMG recording since this attenuates low frequency movement artifacts. The lower bandwidths may be used for ECG recording.

It is also possible to customize the bandwidth beyond these three selectable choices. In order to configure a custom bandwidth, determine the desired upper and lower cutoff frequencies and use the charts on page 25 of the RHD2000 series chip datasheet to determine the values for RH1 DAC1, RH1 DAC2, RH2 DAC1, RH2 DAC2, RL DAC1, RL DAC2, and RL DAC3. Once these values are determined, write these values to registers 8-13 on the RHD2216. The function SendWriteCommand can be used.

Setting Amplifier Power

Inactive amplifiers can be powered down to reduce noise and save power. By default, the operation sketch powers only the channel connected to Phone Jack 1 (FIRSTCHANNEL, which corresponds to channel 0 on the RHD2216). Any combination of powering on or off Phone Jack 1 and/or Phone Jack 2 is possible, and the user only needs to uncomment the proper variable assignments of FirstChannelPwr and SecondChannelPwr towards the beginning of the .ino file.

When the chip is first initialized through SPI commands, part of the process involves selecting which amplifiers should receive power. First, all amplifiers are set to power down. Then, the amplifiers chosen by the user receive a second command to power on through the function SetAmpPwr(). FirstChannelPwr and SecondChannelPwr are used by this function to determine whether or not to send the SPI command to power that channel back on.

Another important use of these variables is to set the data from inactive amplifiers to zero. Turning off an amplifier doesn't ensure that its output is zero, so the data should be explicitly set. If either variable is false (i.e. the channel is inactive), then that channel's data is overwritten by zero after every reading within the ISR (see *Interrupts* section for more details), as well as in the main loop().

Output Scaling

The RHD2216 amplifier system provides linear gain from input (differential electrode) to output (DAC voltage) when average_energy_mode is disabled. With low_gain_mode disabled, this gives a gain of approximately 3000 V/V (69.5 dB). With low_gain_mode enabled, this becomes 750 V/V (57.5 dB). Average_energy_mode does not have a constant linear gain, because the algorithm that generates the output introduces non-linearity by calculating the average amplitude (which changes depending not only on the amplitude of the signal, but also on the proportion of time it spends at each value) and including large offsets in order to recognize the presence of signals at frequencies significantly lower than the accumulator frequency. The user should instead focus on keeping the input signal within saturation levels while in average_energy_mode, which are 1.1 mV peak-to-peak when low_gain_mode is disabled and 4.4 mV peak-to-peak when low_gain_mode is enabled.

The developer may desire finer control over the gain of the system than simply dividing it by a factor of four. This is possible by disabling low_gain_mode and changing the pre-processor macro DAC_SCALE_COEFFICIENT, which is defined in the IntanShield.h file. The default value is 12, because it allows all electrode signals below roughly 1.1 mV peak-to-peak to be output without clipping. In order to scale to a different maximum voltage range, calculate DAC_SCALE_COEFFICIENT (abbreviated DSC) using the following equation:

$$DSC = \frac{12.8 \text{ mV}}{inputV_{pp}}$$

where inputVpp is the maximum peak-to-peak input voltage you wish to see clearly before clipping. This 12.8 mV value is derived from the RHD2216 on-chip ADC's Least Significant Bit voltage step size as shown:

$$12.8 \text{ mV} = 2^{ADCbits} * LSBstepsize$$

where ADCbits is the number of bits used on the chip's analog-to-digital converter (for the RHD2216, 16 bits) and LSBstepsize is the ADC's Least Significant Bit voltage step size (for the RHD2216, 0.195 µV). This means 12.8 mV peak-to-peak is the maximum input the RHD2216 can accurately convert. If we need a 1.1 mV peak-to-peak to be considered the maximum input, then it must be multiplied by 12 to approximate that range.

Sound Customization

By default, two sounds are included in the Intan library to play when signals from Phone Jack 1 and Phone Jack 2 trigger Activity Detection. These sounds are sine waves at an A pitch (440 Hz), and an E pitch (330 Hz). Included in the library is a MATLAB file generatesounddata.m that demonstrates how the A and E data were created. Frequency values can be altered to give different pitches, and once the MATLAB script has run, the user only needs to copy the generated MATLAB variables' data into the arrays in sound.h.

Sounds other than sine waves can also be copied into the arrays in sound.h. Included in the library is a MATLAB file wavtoarduino.m. If a .wav format sound file is present in the working directory of MATLAB, the function can be called using the file name to generate a MATLAB variable that can then be copied into sound.h. It is recommended that any user-generated sounds are kept short (around 200 ms) so that when a second activity is detected, the first activity's sound has already ended.

Professional quality audio is usually sampled at a bit depth of 16 or 24 bits and played at 44.1 kHz. Due to processing limitations our operation sketch has to manage with a bit depth of 8 bits and Pulse Width Modulation (PWM) at 2 kHz. Sounds sampled and played back under these restrictions would not sound very good through the best of speakers, much less a small PCB-mounted speaker. If high quality sound is desired, the developer might consider attaching a peripheral device that instead triggers from the digital Activity Detection signals and does the heavy lifting of audio playback itself.

Managing CPU Time

The general-purpose operation sketch is a compromise between supporting software options (e.g., notch filtering), sampling rate, and number of channels. The developer who wishes to increase the sampling rate, the number of channels, or the processing tasks done in software must be aware of these tradeoffs. All of these tasks share the common need to properly manage CPU time. In order for channel data to be sampled at reliable rates and avoid compromising the execution of the Interrupt Service Routine (ISR; see *Interrupts* section for more details), the ISR should complete execution within 80% of the allotted ISR period. By default, the ISR period is 0.5 ms.

This can be monitored by viewing the signal from Digital Pin 2 (B3 on Figure 3), which is high while the ISR is executing and low otherwise. If this signal's duty cycle ever surpasses 80%, or if the rising edge of the signal no longer has the period set in Timer 1, the ISR's processing is taking too long to perform reliable sampling. If this occurs, instructions must either be removed from the ISR (removing supporting software options), or the ISR period must be increased (decreasing the sampling frequency or the number of channels sampled).

The amount of time taken per ISR is directly related to the number of instructions completed. For most samples, this amount of time should be roughly the same. However, every 20N samples (for N channels), the accumulator code is executed. By default, the calculations and ADC sampling that occur in the accumulator code take approximately 75 µs. This is about 15% of the default ISR time, and is the reason for the rule of thumb that the signal's duty cycle should never surpass 80%. As the developer removes unneeded functionality from the sketch, the ISR execution time may shrink such that the accumulator code takes up a significant amount of time compared to the other ISR's much higher than 15%. The developer may choose to fit the allotted ISR period to

the shorter non-accumulator ISR's, but with the drawback that the accumulator ISR will take longer to execute than the allotted time. In this case, the developer may consider removing the accumulator code (if sampling is all that is desired, as opposed to Activity Detection), so that every 20th sample is measured at the same period as the others.

Reducing CPU time is important for the developer who wishes to increase the sampling rate or number of channels, and can be accomplished primarily by eliminating the number of mathematical operations the processor performs. The most processor-intensive operation is the 60/50 Hz notch filter which performs an IIR filter algorithm every ISR, for every incoming sample. Floating-point mathematical operations accomplish this, and the execution time suffers as a result. The notch filter algorithm takes about 0.1 ms to complete, or 20% of the default allotted ISR time. As a result, disabling the notch filter through the on-board switch will free up about 0.1 ms from the ISR's execution.

Performing all of the above recommendations will result in an ISR execution period of around 80 μs. If Timer 1's period is set to 100 μs (so that ~80% of the period is spent executing the ISR), the ISR rate will be increased by a factor of five, to 10000 ISRs per second. In theory this could give a maximum sampling rate of 10 kHz for one channel. However, the developer should be aware that I2C communication still needs to occur between the Arduino and the DAC. While the DAC is rated for a maximum SCL rate of 400 kHz, the developer would need to reconfigure the Arduino TWI registers to match their desired communication rate, and Intan makes no guarantee that the I2C interface will function properly beyond the default setting.

Changing Number of Channels

While the operation sketch samples a default of two channels, the board includes solder holes to allow sampling up to an additional 14 channels (see *Optional Plated Solder Holes*). The general-purpose operation sketch is a compromise between supporting software operations (i.e. notch filtering, serial print statements, etc.), sampling rate, and number of channels. The developer who wishes to sample from additional channels has the option to remove supporting software operations (see *Managing CPU Time*) and/or decrease the sampling rate (see *Changing Sampling Frequency*), increasing the number of channels that can be sampled. The developer who wishes to add more supporting software operations and/or increase the sampling rate has the option to sample from only one channel.

Whether the number of channels is increased or decreased, the developer should ensure that the only amplifiers receiving power are the amplifiers actually being used. See *Setting Amplifier Power* for more details. If the developer increases the number of channels, it may be helpful to additionally define ThirdChannelPwr, etc. to mirror FirstChannelPwr and SecondChannelPwr. If the developer decreases the number of channels, it may be helpful to delete all instances of SecondChannelPwr. In either case, the developer should ensure that at each use of these variables, the number of variables match the number of active channels. The important occurrences that need to be trimmed/added include:

a) Variable definitions, towards the beginning of the .ino file.
b) Function call of `SetAmpPwr(bool Ch1, bool Ch2)`, which occurs once in the setup() function. The function itself is explained further below.
c) Function definition of `SetAmpPwr(bool Ch1, bool Ch2)`, a function which accepts the variables from the .ino file and reassigns their values to the global variables `firstchannelpower` and `secondchannelpower` to allow use of these values within the ISR, towards the end of the .cpp file. This function also sends SPI commands to set registers 14 through 17 depending on which amplifiers should receive power (see *Serial Peripheral Interface (SPI)* for more details). The parameters of this function and its contents should reflect the number of amplifiers to use.
d) Within the ISR, after the reading of each channel through SPI commands, immediately after assigning the read data. If the channel is inactive, the data should be overwritten by zero. Occurs in the middle of the .cpp file, within the ISR() function.

Developers who wish to change the number of channels must first address the problem of the Interrupt Service Routine (ISR) execution period (see *Interrupts* section for more details), and either change the CPU time or change the sampling rate. Once this is done, the developer may use that new time to sample a different number of channels than the default two per sampling period. The new ISR period should be the sampling frequency divided by the number of channels. The ISR period is set by Timer 1, which by default is set within the setup() function in the sketch. Timer 1 is set from the macro INTERRUPT_RATE, defined at the beginning of the IntanShieldOperation.ino sketch. Consult the ATmega328P datasheet for more information on proper register values for setting Timer 1 to the desired period.

Since audio playback is performed by playing a sample for each interrupt, changing the interrupt rate effectively changes the playback rate of the audio. In order to avoid the audio sounding artificially faster or slower, it should be sampled at the same rate

as playback (INTERRUPT_RATE). Intan provides MATLAB files for generating the default A and E notes, or formatting any .wav sound to be played by the Arduino. See *Sound Customization* for more details. In either case, the file assumes the default audio sample rate of 2000 Hz, so the MATLAB code should be modified, replacing all instances of 2000 Hz with the new interrupt rate before generating the sound data.

An important software change to implement for changing number of channels is allowing the ISR loop to cycle through each channel in turn. By default, this is implemented with the variable i, which alternates between 0 (for the channel connected to Phone Jack 1) and 1 (for the channel connected to Phone Jack 2). The variable i should reach a maximum value of N – 1, where N is the number of channels. The variable x is used similarly to i, and should share the same maximum value. When the ISR loop ends, i should increment by 1 until it reaches its maximum value, at which point it should be reset to 0. By default, various conditional statements in the ISR check if the value of i is 1. This is to determine if all of the channels have been sampled, so these occurrences of 1 should be replaced with N-1.

Additionally, before the ISR many variables are defined including arrays of length two. These arrays are generally used to store information about each channel, and every occurrence of an array definition (of size 2 x 1 or 2 x 3) should be replaced (with arrays of size N x 1 or N x 3). Similar arrays also appear in the NotchFilter functions, and should be replaced the same way. Functions ReadChannelData and ReadAccumulatorData similarly assume a channel number of two, and thus should be altered.

Furthermore, the actual sampling occurs as a result of the SendConvertCommand(CHANNEL) functions. By default, macros FIRSTCHANNEL and SECONDCHANNEL are defined so that FIRSTCHANNEL = Phone Jack 1 = RHD2216 channel 3, and that SECONDCHANNEL = Phone Jack 2 = RHD2216 channel 12. The developer who wishes to sample from more channels must add additional SendConvertCommand(CHANNEL) functions, and it is recommended to define THIRDCHANNEL, FOURTHCHANNEL, and so on for consistency. The four extra RHD2216 channels the developer has access to are 6, 7, 8, and 9. The SendConvertCommand(CHANNEL) functions are sent through a SPI 3-command pipeline, which can give unintended consequences if the data assignments are not properly aligned with the commands themselves. See the *Serial Peripheral Interface* section for more details. The developer who wishes to sample from one channel only needs to replace the SendConvertCommand(SECONDCHANNEL) function in setup() with an additional SendConvertCommand(FIRSTCHANNEL) function, and remove the SendConvertCommand(SECONDCHANNEL) function from ISR().

A final problem that must be addressed for a different number of channels is the viewing of the output. The on-board DAC chip only has two DACs available. To clearly monitor the output of more than two channels simultaneously, the developer needs to add additional DAC(s) to the system, which will increase the complexity of the sketch and introduce communication concerns. To ensure satisfactory communication speed, SPI or I2C interfaces are recommended for any peripheral devices. For developers seeking to sample from only one channel, the loop() function can be modified to only send data from FIRSTCHANNEL to the DAC. In this case, I2C communication should be slowed down by a factor of two so that the DAC updates at the same rate as the sample data. I2C communication speed is configured through registers TWSR and TWBR. Consult the ATmega328P datasheet for more information on proper register values for setting the TWI (I2C) communication to a different speed. While the DAC is rated for a maximum SCL rate of 400 kHz, the developer would need to reconfigure the Arduino TWI registers to match their desired communication rate, and Intan makes no guarantee that the I2C interface will function properly beyond the default setting. However, slowing down the I2C interface should be safe.

Changing Sampling Frequency

While the operation sketch samples at a default rate of 1 kS/s per channel, the shield can support higher frequencies with some tradeoffs. The general-purpose operation sketch is a compromise between supporting software operations (i.e. notch filtering, serial print statements, etc.), sampling rate, and number of channels. The developer who wishes to increase the sampling frequency has the option to remove supporting software operations (see *Managing CPU Time*) and/or decrease the number of channels sampled (see *Changing Number of Channels*), reducing the sampling period. The developer who wishes to add more supporting software operations and/or increase the number of channels has the option to reduce the sampling frequency.

Developers who wish to change the sampling frequency must first address the problem of the Interrupt Service Routine (ISR) execution period (see *Interrupts* section for more details), and either change the CPU time or change the number of channels sampled. Once this is done, the developer may use that new time to sample at a different rate than the default 1 kS/s. The new ISR period should be the sampling frequency divided by the number of channels. The ISR period is set by Timer 1, which by default is set within the setup() function in the sketch. Timer 1 is set from the macro INTERRUPT_RATE, defined at the beginning of the IntanShieldOperation.ino sketch. Consult the ATmega328P datasheet for more information on proper register values for setting Timer 1 to the desired period.

Since audio playback is performed by playing a sample for each interrupt, changing the interrupt rate effectively changes the playback rate of the audio. In order to avoid the audio sounding artificially faster or slower, it should be sampled at the same rate as playback (INTERRUPT_RATE). See *Sound Customization* for more details.

Depending on the desired sampling frequency, it may be necessary to customize the amplifier bandwidth. The default bandwidths have an upper cutoff frequency of 500 Hz, which is the Nyquist frequency for the sampling rate of 1 kHz. The upper cutoff should be no more than half the sampling frequency, so decreasing the sample frequency mandates reconfiguring the amplifier bandwidth to be narrower. Increasing the sample frequency gives the developer the option to reconfigure the amplifier bandwidth to be wider, to monitor frequencies above 500 Hz, but otherwise it is not required. See *Configuring Amplifier Bandwidth* for more details.

Similar to when the developer changes the number of channels, it may also be necessary to change the speed of I2C communication so that the DAC updates at the same rate as the sample data. I2C communication speed is configured through registers TWSR and TWBR. Consult the Atmega328P datasheet for more information on proper register values for setting the TWI (I2C) communication to a different speed. While the DAC is rated for a maximum SCL rate of 400 kHz, the developer would need to reconfigure the Arduino TWI registers to match their desired communication rate, and Intan makes no guarantee that the I2C interface will function properly beyond the default setting.

A final problem that must be addressed for a changed sampling frequency is the use of the notch filter. The notch filter coefficients have been calculated assuming a sampling rate of 1 kS/s, so developers who plan to use the notch filter at a different sample rate must recalculate the coefficients used to filter out 50 or 60 Hz noise. The steps to calculate each coefficient are outlined in comments above the NotchFilter60() function definition, and once calculated, the new coefficients should replace the default coefficients in the notch filter algorithm.

Arduino Shield Hardware Interface

While this device was designed to interface with an Arduino, some users may wish to use a different microcontroller to control the board. Many other types of microcontrollers or FPGAs can be used to interface with the male header pins typically used for the Arduino connection. The functions of these header pins are described below.
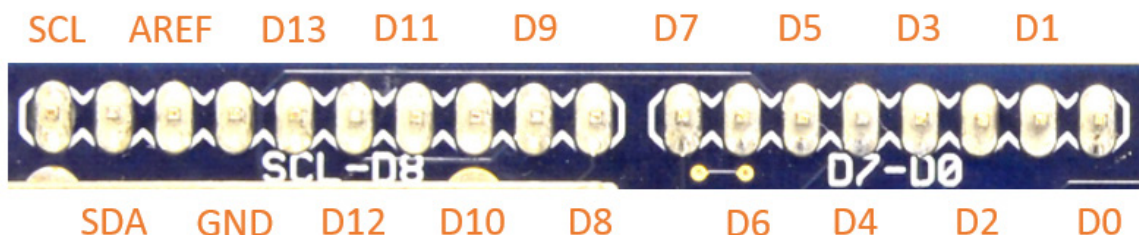
Top Row Header Pins

Figure 17. Top row of male header interface pins.

SCL: A pin used along with SDA for TWI (I2C) communication with the on-board 8-bit DAC. If the user wishes to operate the DAC with a different controller, this I2C interface must be preserved. See *I2C (TWI) Interface* for more details.

SDA: A pin used along with SCL for TWI (I2C) communication with the on-board 8-bit DAC. If the user wishes to operate the DAC with a different controller, this I2C interface must be preserved. See *I2C (TWI) Interface* for more details.

AREF: A pin connected to the analog reference voltage used to determine the upper limit of the ADC. In order to preserve proper reading of the threshold fader, this pin should be supplied with +3.3V.

GND: One of three header pins connected to the board's ground. Since they are all connected on the board, the user can choose to use them all or connect only one of these pins to the controller.

D13: A pin connected to SCLK (serial clock) in SPI communication with RHD2216 chip. Along with MISO, MOSI, and CS, this interface is vital to the control of the amplifier chip. See *Serial Peripheral Interface (SPI)* for more details.

D12: A pin connected to MISO (master in, slave out) in SPI communication with RHD2216 chip. Along with SCLK, MOSI, and CS, this interface is vital to the control of the amplifier chip. See *Serial Peripheral Interface (SPI)* for more details.

D11: A pin connected to MOSI (master out, slave in) in SPI communication with RHD2216 chip. Along with SCLK, MISO, and CS, this interface is vital to the control of the amplifier chip. See *Serial Peripheral Interface (SPI)* for more details.

D10: A pin connected to CS (chip select, also often referred to as SS) in SPI communication with RHD2216 chip. Along with SCLK, MISO, and MOSI, this interface is vital to the control of the amplifier chip. See *Serial Peripheral Interface (SPI)* for more details.

D9: An unused pin that is connected only to the 8x2 male header pin B5 to allow users access to additional Arduino I/O. Can similarly allow users on-board access to I/O from other controllers, if desired.

D8: An unused pin that is connected only to the 8x2 male header pin A5 to allow users access to additional Arduino I/O. Can similarly allow users on-board access to I/O from other controllers, if desired.

D7: A pin connected to LOW_GAIN_MODE, a digital signal that is HIGH (the value of IOREF) when the on-board switch is in the ON position, and is LOW (0 V) when the switch is in the OFF position.

D6: A pin connected to AUDIO_EN, a digital signal that is HIGH (the value of IOREF) when the on-board switch is in the ON position, and is LOW (0 V) when the switch is in the OFF position.

D5: Activity Detection pin, which is also connected to the 8x2 male header pin B4. By default, the Arduino will set this digital signal HIGH when the electrode connected to Phone Jack 2 picks up a signal higher than the threshold set by the threshold fader.

D4: Activity Detection pin, which is also connected to the 8x2 male header pin A4. By default, the Arduino will set this digital signal HIGH when the electrode connected to Phone Jack 1 picks up a signal higher than the threshold set by the threshold fader.

D3: Digital output pin, which uses PWM (pulse width modulation) to drive the on-board speaker when audio is enabled. In order to preserve speaker functionality, the different controller should output an audio signal onto this pin. For more details on PWM, see the functions "ISR", "StartSound", and "StopSound" in *IntanShield.cpp*.

D2: Digital output pin, which is also connected to the 8x2 male header pin B3, and allows optional ISR duty cycle monitoring. See *Managing CPU Time* for more details.

D1: An unused Arduino digital pin that is only connected to the 8x2 male header pin A3 to allow users access to additional Arduino I/O. Can similarly allow users on-board access to I/O from other controllers, if desired. While the Arduino aliases this pin as TX and uses it for the USB Serial connection (a potential pit-fall for Arduino users planning on using USB while controlling this pin), this should be of no concern if a different controller is used.

D0: An unused Arduino digital pin that is only connected to the 8x2 male header pin B2 to allow users access to additional Arduino I/O. Can similarly allow users on-board access to I/O from other controllers, if desired. While the Arduino aliases this pin as RX and uses it for the USB Serial connection (a potential pit-fall for Arduino users planning on using USB while controlling this pin), this should be of no concern if a different controller is used.
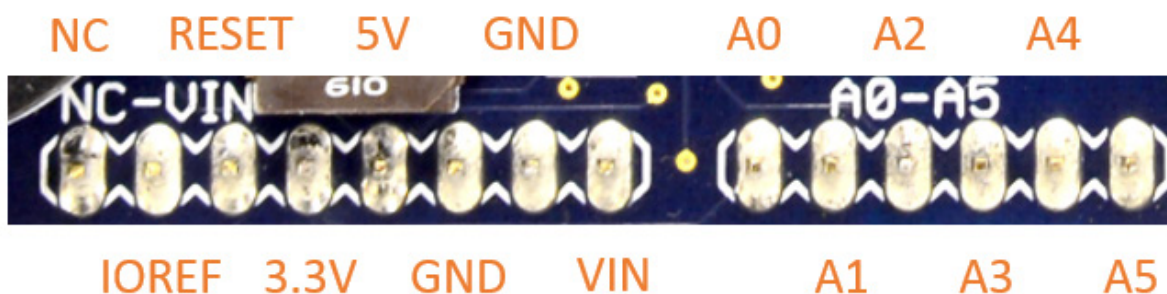
Bottom Row Header Pins

Figure 18. Bottom row of male header interface pins.

NC: An unused pin that is only connected to the 8x2 male header pin A8 to allow users access to additional Arduino I/O. Can similarly allow users access to I/O from other controllers, if desired. At the time of the writing of this document, the Arduino itself leaves this pin unconnected. However, this should be of no concern if a different controller is used.

IOREF: A power pin that should be supplied by the controller, and is connected to the 8x2 male header pin B7 as well as the DAC and digital isolator for SPI communication. This voltage corresponds to the digital voltage level used by the controller (5 V for the Arduino Uno, 3.3 V for the Arduino Due). The controller must supply the voltage level at which it operates to this pin.

RESET: An unused pin that is left unconnected on this board, since users can just use the push-button on the Arduino board itself when a reset is desired. A different controller has no need to interface with this pin.

3.3V: A power pin that should be supplied by regulated 3.3 V. This pin is connected to the 8x2 male header pin A7, and also supplies the reference used to measure the value of the threshold signal, given by the position of the threshold fader.

5V: A power pin that should be supplied by regulated 5 V. This pin is connected to the 8x2 male header pin B6, and supplies the on-board DC-DC converter that provides an isolated 3.3 V power source, which is used to power the RHD2216 chip.

GND (2X): One of three header pins connected to the board's ground. Since they are all connected on the board, the user can choose to use them all or connect only one of these pins to the controller.

VIN: An unused pin that is only connected to the 8x2 male header pin A6. The Arduino includes this pin as an alternative power supply, but it should be of no concern if a different controller is used.

A0: A pin connected to the output of the user-controlled threshold fader. This pin will carry an analog signal between 0V and 3.3V, corresponding to the position of the on-board threshold fader.

A1: A pin connected to DAC_ACC_MODE, a digital signal that is HIGH (the value of IOREF) when the on-board switch is in the ON position, and is LOW (0 V) when the switch is in the OFF position.

A2: A pin connected to NOTCH_EN, a digital signal that is HIGH (the value of IOREF) when the on-board switch is in the ON position, and is LOW (0 V) when the switch is in the OFF position.

A3: A pin connected to NOTCH_5060, a digital signal that is HIGH (the value of IOREF) when the on-board switch is in the ON position, and is LOW (0 V) when the switch is in the OFF position.

A4: An unused pin that is left unconnected on this board. A different controller has no need to interface with this pin.

A5: An unused pin that is left unconnected on this board. A different controller has no need to interface with this pin.
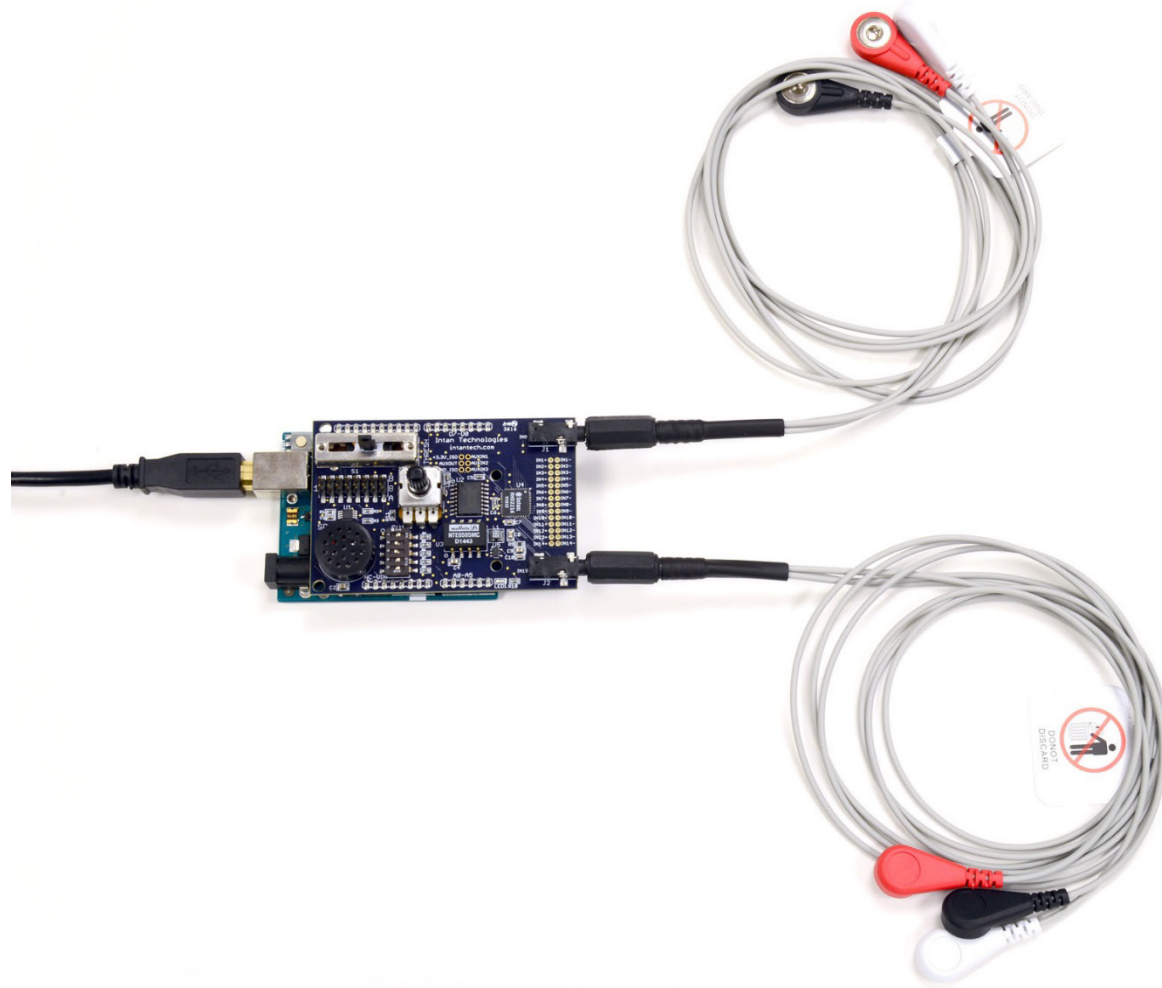
Figure 19. RHD2216 Arduino Shield Board with included snap electrode leads.