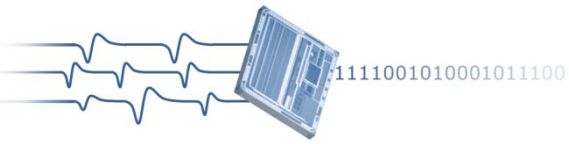


RHD/RHS

STM32 Firmware Framework

Version 1.2

18 June 2025

Features

- Open-source STM32 firmware written in C to stream real-time data from Intan RHD2216, RHD2132, RHD2164, or RHS2116 electrophysiology amplifier chips.
- Optimized SPI communication between STM32 MCU and Intan RHD/RHS chip.
- Interrupt-based code using DMA (direct memory access) allows for sampling rates up to 20 kSamples/s per channel for 32 channels.
- Demonstration of streaming certain acquired data channels, either in real time or offline, via USART
- RHS code includes triggered stimulation waveforms on two channels
- Both HAL and LL libraries included.
- STM32U5 and STM32H7 series supported.

Applications

- Rapid prototyping of Intan Technologies RHD/RHS amplifier-based products.
- Starting point for the development of custom interfaces to RHD2216, RHD2132, RHD2164 or RHS2116 chips.

Description

To facilitate the development of electrophysiology recording systems using the RHD or RHS series of microchips, Intan Technologies provides the following open-source STM32 firmware framework for developers. The framework consists of C code written for the commercially-available **STM32U5** and **STM32H7** microcontroller series produced by STMicroelectronics. The example code streams multi-channel data from Intan RHD or RHS chips at a sample rate of 20 kSamples/s per channel using timers, interrupts, and DMA to maintain high throughput while using only a small fraction of the MCU capacity. RHS chip examples also include stimulation on two independent channels, by default triggered by a GPIO rising-edge on the pin routed to the blue user button on the STM32 NUCLEO boards.

Why is This Code Specific to the STM32U5 and STM32H7?

There are several series of STM32 microcontrollers with a very wide range of specifications suitable for different applications. The example code we provide here is not an indication that only the STM32U5 and STM32H7 series will be the best fit for all projects using Intan chips. However, there are some specific advantages that the U5 and H7 series have over other STM32 series that make them a reasonable starting point for working with Intan chips.

The U5 series was launched in 2021 and is currently the most cutting-edge evolution of the well-established L series. It targets ultra-low power applications while still having a maximum CPU clock rate of 160 MHz and robust peripheral support. The H7 series was launched in 2017 and is designed for high performance, with a maximum CPU clock rate of 550 MHz. High-speed processing and SPI data transmission / reception are the most important features for achieving high sample rate / high channel count communication with Intan chips, and while we have not quite been able to reach the maximum data rate the Intan chips are physically capable of supporting (30 kS/s for 32 amplifier channels + 3 auxiliary commands for RHD, or 16 amplifier channels + 4 auxiliary commands for RHS), we have gotten quite close (20 kS/s) with relative ease, using both the **HAL** (Hardware Abstraction Layer) and **LL** (Low Layer) drivers, while also allowing for streaming of certain channels over USART. The U5 series is sufficient for basic data acquisition from a single RHD2216, RHD2132, RHD2164 or RHS2116 chip and transmission of that data over a USART interface. The H7 series can handle these same tasks while also running at a significantly higher clock rate, and this boosted performance could help with any additional processing tasks that run alongside data acquisition. However, the H7 has significantly less RAM, limiting the amount of memory that can be used for data storage, for example when storing data for offline data transmission. Even those applications requiring a higher sample rate may be able to achieve this by optimizing the interface to omit unnecessary features.

The most important features of the STM32U5 and STM32H7 used in the Intan RHD/RHS firmware framework are **SPI** (Serial Peripheral Interface), **timer-generated interrupts** to achieve a reliable sample rate, and **DMA** (Direct Memory Access) to allow multi-word transactions between memory and peripherals to occur without requiring direct processor intervention. Most applications will also require some way to transmit acquired data somewhere or save it to memory, so peripherals for interacting with USART, Ethernet, wireless systems, or SD cards will probably be useful, and our provided examples demonstrate transmission of certain channels of acquired data over USART.

SPI Communication Requirements

A critical signal in the Intan SPI communication protocol is $\overline{CS}$ (active-low chip select, called NSS in the STM32) rising high, remaining high for at least 154 ns, and then falling low between each 16-bit word (32-bit words for RHS). Unfortunately, the popular **STM32F4** series SPI bus does not appear to have an easy way to achieve this behavior. NSS is indeed driven low during each 16-bit word, but **for these older STM32 chips, NSS is not toggled high between words**, so the Intan chip does not receive the clear NSS/$\overline{CS}$ high signal indicating the end of a 16-bit word (32-bit word for RHS).

This NSS/$\overline{CS}$ pulse between every SPI word is required for the Intan chip to operate correctly, so for the STM32F4 chips we are forced to decouple NSS from the SPI peripheral and instead use a GPIO pin for $\overline{CS}$. Unfortunately, this requires direct processor intervention between every 16 or 32-bit word to write $\overline{CS}$ high, wait, and then write $\overline{CS}$ low again. This does allow the Intan chip to communicate properly, but it wastes CPU clock cycles, and prohibits the use of DMA for bulk data transfers. While this approach may be feasible for relatively low sample rates (5 kS/s or lower), it is inefficient and limits the communication between the MCU and the Intan chip. This manual control of NSS/$\overline{CS}$ is necessary for the F4 series and other STM32 series with similar SPI buses. (In theory it is possible to use a precisely-set timer tied to $\overline{CS}$ that is synchronized with the SPI bus to automate $\overline{CS}$ toggling, but this complicates the SPI communication beyond the scope of entry-level demonstration code.)

The SPI bus implementation details can differ quite significantly between microcontroller series and manufacturer, so we strongly encourage users research their proposed MCU's SPI implementation to ensure shortcomings like this do not hinder or complicate data transfer with the Intan chip. We have verified that the STM32U5 and STM32H7 series have SPI buses that can easily be used with NSS automatically pulsing high between 16 or 32-bit words, and these series also work nicely with DMA for large data transfers.

How Does RHD2164 Firmware Differ from Other RHD Chips?

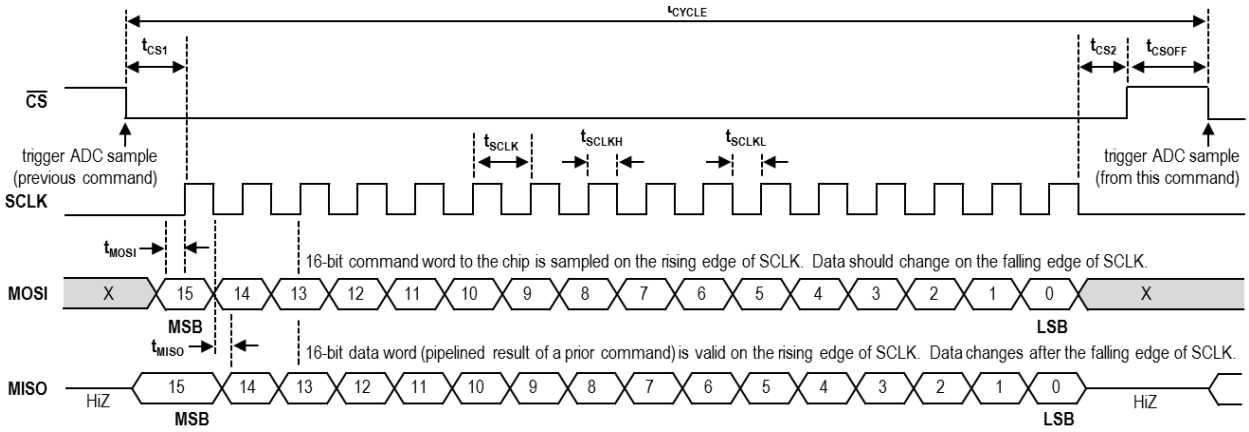
An important difference between the RHD2164 chip and its RHD2216 / RHD2132 cousins is that its SPI bus uses a Double Data Rate (DDR) technique to transfer 32 bits of data, instead of 16 bits, from Intan chip to controller over a single chip select cycle. Rather than the standard SPI implementation, which samples Master In Slave Out (MISO) at the rising edge of each Serial Clock (SCLK) pulse, DDR samples at both the rising and falling edges. Master Out Slave In (MOSI) acts the same between standard SPI and DDR implementations. Refer to the RHD2164 datasheet for further details on how DDR operates.

The RHD2164 chip's use of DDR allows for the ability to stream twice as much data from the Intan chip to controller, but its non-standard mode of SPI communication has historically only been achievable with an FPGA. With this firmware framework, at the cost of increased complexity and use of additional peripherals, we demonstrate techniques that allow an STM32 microcontroller to read and write across a DDR SPI interface

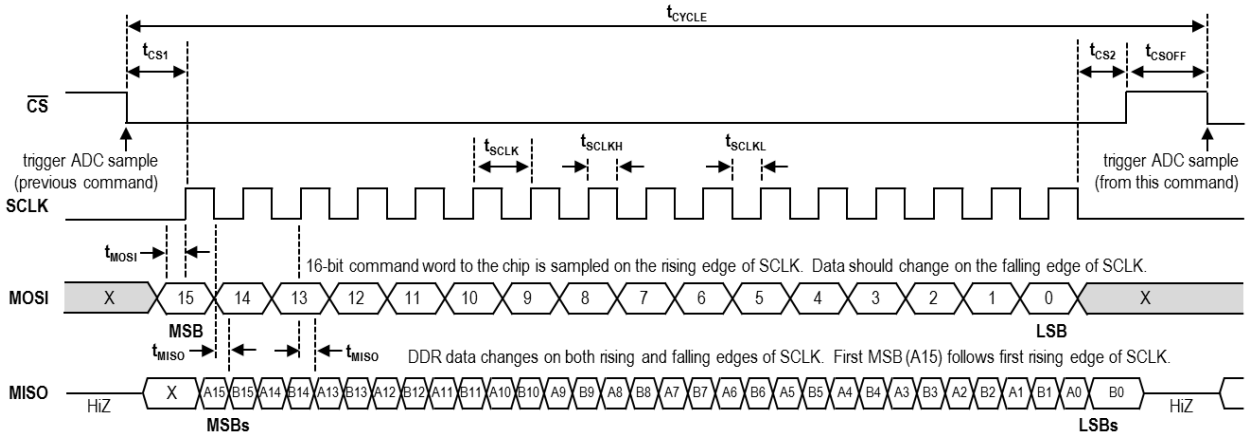
How the DDR SPI Signals Differ from Standard SPI

While the $\overline{CS}$, SCLK, and MOSI signals have no change from standard SPI to DDR SPI, the MISO signal is sampled twice as fast. Additionally, for standard SPI, the first valid MISO bit is sampled at the *rising* edge of the first SCLK pulse, but for DDR SPI, the first valid MISO bit is sampled at the *falling* edge of the first SCLK pulse. Standard SPI yields 16 bits, in order, from most-to-least significant bits, while DDR SPI yields 32 bits, interleaved, from most-to-least significant bits, containing contents from stream A and stream B. These differences are illustrated in the standard SPI timing diagram (top) and the DDR SPI timing diagram (bottom):

Standard SPI timing



DDR SPI timing



While standard SPI communication allows for straightforward use of a single SPI peripheral for both transmission and reception, achieving DDR SPI with a single Intan chip requires use of two SPI peripherals (one for transmission, one for reception), use of one or two timer peripherals (depending on factors like CPU clock rate, SPI peripheral clock speed, and desired SPI Baud rate, it may be necessary for a second timer peripheral to introduce a delay) to generate a "pseudo-SCLK" signal, and extracting two distinct 16-bit data words from the received, interleaved 32-bit data word.

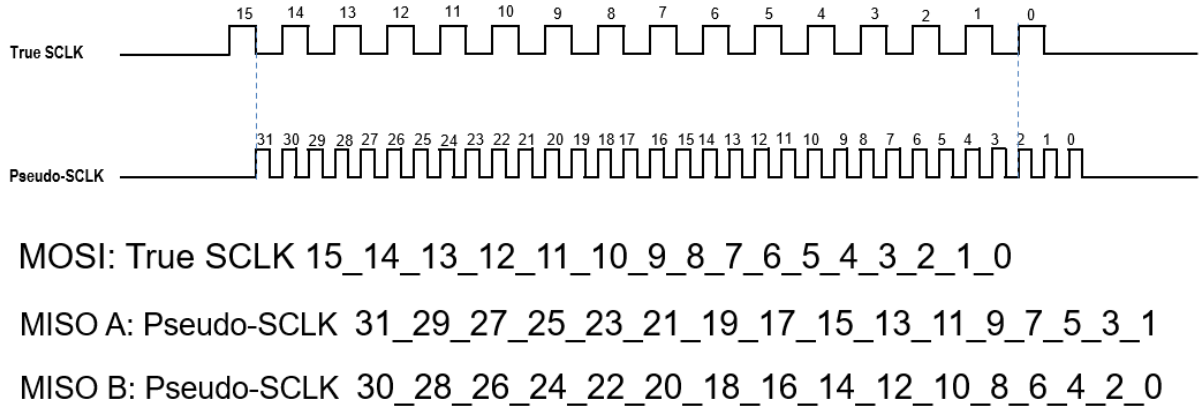
Configuration of Two SPI Peripherals

For DDR, MOSI transmits 16-bit data words while MISO receives 32-bit data words. This is not a mode supported by any U5 or H7 SPI peripherals, so it becomes necessary to dedicate one SPI bus (TRANSMIT_SPI) to transmission of 16-bit words on MOSI, and a second SPI bus (RECEIVE_SPI) to reception of 32-bit words on MISO. Since the other SPI signals, ($\overline{CS}$, SCLK) are no different from standard SPI configuration, TRANSMIT_SPI is responsible for outputting $\overline{CS}$, SCLK, and MOSI, configured as Transmit Only Master, while RECEIVE_SPI is responsible only for receiving MISO configured as Receive Only Slave. (In this mode, note that the MCU is considered the slave, so the data line that the STM32 peripheral RECEIVE_SPI refers to as "MOSI" is actually the same line the Intan chip refers to as "MISO".)

$\overline{CS}$ should be wired directly from TRANSMIT_SPI to RECEIVE_SPI. SCLK cannot be used as-is by RECEIVE_SPI, as the 16-bit SCLK signal that actually reaches the Intan chip cannot be used by RECEIVE_SPI, which expects a 32-bit SCLK signal to sample MISO. This requires the MCU to generate a 32-pulse "Pseudo-SCLK" at twice the real SCLK frequency, and delayed by half an SCLK cycle, which must be connected to RECEIVE_SPI as its SCLK signal. The generation of Pseudo-SCLK is described in the section below.

Use of Timers to Generate a 32-pulse Pseudo-SCLK

As can be seen in the above DDR SPI timing diagram, MISO needs to be sampled at both the rising and falling edge of the true SCLK line, and the 32 samples start at the *falling* edge of SCLK, not the *rising* edge (there must be a delay between the true SCLK's first rising edge and Pseudo-SCLK's first rising edge). So, the timer peripheral (TIM) can be used to generate a Pseudo-SCLK which has rising edges that coincide with these rising/falling edges, and this Pseudo-SCLK is wired to RECEIVE_SPI's SCLK. The necessary synchronization between these SCLK signals can be visualized with this diagram:



The odd MISO samples correspond to data stream A and even MISO samples correspond to data stream B. The dotted blue lines demonstrate how the falling/rising edges of True SCLK synchronize with the rising edges of Pseudo-SCLK.

For both the U5 and H7 MCUs, Pseudo-SCLK is output on the RECEIVE_SCLK_TIM, using PWM Generation and a Repetition Counter of 31 to result in a 32-pulse signal. The provided clock configuration for the U5 just happens to have an inherent delay between the timer trigger event ($\overline{CS}$ low) and the first RECEIVE_SCLK_TIM output that aligns perfectly with the necessary delay between the True SCLK and Pseudo-SCLK rising edges, so no further configuration is necessary. However, the H7 runs significantly faster, so use of a second timer (CS_DELAY_TIM) is necessary to introduce a configurable delay between the trigger event and the first RECEIVE_SCLK_TIM output. In short, for the U5, $\overline{CS}$ triggers RECEIVE_SCLK_TIM, which outputs Pseudo-

SCLK, while for the H7, $\overline{CS}$ triggers CS_DELAY_TIM, which after a short delay triggers RECEIVE_SCLK_TIM, which outputs Pseudo-SCLK.

De-interleaving a Merged 32-bit Word into its 2 Component 16-bit Words

The configuration described above allows for RECEIVE_SPI to read data into MCU memory, but the data it receives contains 32 bits of data, such that every other bit corresponds to MISO stream A (odd bits) and MISO stream B (even bits). In order to extract the original two 16-bit data words that were interleaved on the RHD2164 chip to form each 32-bit data word, it is necessary to de-interleave the data at some point. The provided examples perform this operation in the "extract_ddr_words" function in the "rhdinterface.c" function, which is called immediately after acquisition of each sample. However, it would also be possible to keep the data in these 32-bit words and perform this extraction at some later point, and this may be helpful for applications that need to maximize available processing time during acquisition and do not need immediate access to the extracted data.

The actual extraction in "extract_ddr_words" is explained in the code. A slow, but straightforward, implementation is commented out in the code, and clearly demonstrates shifting every other bit into either word_A or word_B. However, this iterates 16 times through a for loop, and uses many operations (multiplication, addition, bitwise-AND, bitwise-OR, and shifting) that result in very slow execution, ultimately taking too long to finish before the next sample period and causing an ITClip Error at high sample rates. A much faster, but less intuitive, implementation is actually used in the "morton_deinterleave" function. The code and comments demonstrate the principle, inspired by Jeroen Baert's blog post: "Morton encoding/decoding through bit interleaving: Implementations": https://www.forceflow.be/2013/10/07/morton-encodingdecoding-through-bit-interleaving-implementations/, which avoids any for loops, and only uses a few bitwise-AND, bitwise-OR, and shifts to achieve the same results as the straightforward implementation in a fraction of the time.

How Does RHS Firmware Differ from RHD Chips?

While there are significant structural similarities between the RHD and RHS firmware, there are several differences that are important to understand. These are due to the differences between the RHD and RHS chips, and due to the fundamental differences in program goals – typically, RHS chips are used for current stimulation, so a significant portion of the RHS firmware consists of RHS-only code to allow for on-the-fly communication with the Intan chip to control stimulation in real time alongside acquisition.

Communication with the Intan Chip

The most significant difference between RHD and RHS SPI communication is the bit width of the SPI interface. RHD chips use 16-bit words, while RHS chips use 32-bit words. To preserve the ability of a single READ or WRITE command to access a single register in its entirety, that means that the RHS registers are 32 bits in size instead of 16 bits. Also, since RHS chips have DC amplifiers in addition to the RHD-identical AC amplifiers, a single CONVERT command can return the ADC conversion result of both an AC and DC amplifier, as long as the D flag is set in the CONVERT command. The RHS firmware has been altered to comply with these requirements, changing the SPI peripheral configuration and internal processing of data to handle 32 MOSI and MISO bits instead of 16 bits.

Setting Stimulation Current Step Size

Each stimulation sequence (the structure of which is further explained below), specify a series of amounts of current and periods of time for which to hold those currents. However, the units for these values are not fixed. The current step size, governed by RHS register 34, can range from 10 nA to 10 µA. For example, if a stimulation Segment has a current magnitude of -5, and the stim step size was set to 10 nA, then the actual current magnitude will be -50 nA. The file "rhsregisters.h" defines a function "set_stim_step_size" which takes a StimStepSize enum (defined in the same file), allowing the user to easily set what the step size should be in the RHSConfigParameters struct, so that subsequent writes to Register 34 based on this struct will set the actual value on the RHS chip.

By default, this "set_stim_step_size" function is called in two locations in the example project. First, in "rhsregisters.c", in the "set_default_rhs_settings" function, the stim step size is set to 1 µA. Second, in "userfunctions.c", in the "configure_stim_sequences" function, the stim step size is set to 100 nA. Since the second function call is last "set" value that gets written to the chip prior to acquisition/stimulation, this latest value is the actual step size that gets used, so for the example program, the stim step size is 100 nA. This demonstrates not only where this value can be changed, but also how the stim step size can be set and overwritten to take effect in the next stimulation/acquisition session.

Setting Stimulation Time Step Size

Similar to stimulation current step size, the time units for which stimulation Segments are defined are not fixed. However, instead of this being governed by an RHS chip register, the period for a single time step is the equivalent to the period between TIM interrupts (1 / sample rate). So, at the default sample rate of 20 kHz, each time step is 50 µs. For example, if a stimulation Segment has a length of 500 and is run at a sample rate of 20 kHz, this Segment will hold its current for 25 ms. To change this time step size, the user should change the sample rate – see the "Changing sample rate" section for details.

How Stimulation Scheduling Works

In order to allow users to specify stimulation sequences that execute when triggered (triggering is explained further in a later section), it is necessary for the firmware to generate WRITE commands that can be sent to the RHS chip in real time so that changes to parameters like stimulation magnitude and polarity can take effect at the right time, as well as logic to govern how many timesteps should elapse between these commands to achieve desired timing characteristics of stimulation pulses. While users can bypass the provided stimulation scheduler to write code to control all commands directly, it is useful for most general-purpose programs to specify a standard by which stimulation can be scheduled in a uniform manner. By default, after the 16 CONVERT commands in each timestep (sample period), there are four auxiliary command slots, and assuming the preprocessor macro AUTO_STIM_CMD_MODE is defined, these four slots are reserved to issue necessary WRITE commands. The firmware includes some logic to automatically send and, if necessary, queue multiple of these commands in case there are more than four WRITEs

necessary at once (for instance, if multiple channels all require stimulation parameter changes within a short period of time) by using a "command buffer", the details of which can be examined in the "stimscheduler.h" and "stimscheduler.c" files.

The user can specify arbitrary stimulation shapes which will be executed by the stimulation scheduler by instantiating a "StimSequence" object, a C struct that is defined in "stimscheduler.h". The "create_manual_example_seqeunce" function within the "userfunctions.c" file demonstrates how a sequence can be specified, and its use in "configure_stim_sequences", while commented out in favor of demonstrating auto-generate sequences, illustrates where a sequence specified manually in this way is used. The StimSequence structure and its components are described in detail:

`StimSequence`: A structure containing the details of a specific sequence that executes on a single channel, initiated by a single trigger event.

`uint16_t trigger_source`: A number representing the trigger source for this stim sequence. 0 (default) represents inactive or no trigger, 1 represents triggering from rising-edge activity on the GPIO defined as GPIO_EXTI13, routed to the blue button for NUCLEO boards, and any other integers are left unimplemented but can be expanded by the user to include triggers from other events, for example other GPIOs or timers.

`uint8_t channel`: A number representing one of the 0-15 channels this stim sequence occurs on.

`uint32_t num_segments`: A number representing the total number of user-defined "Segments" which are present in this sequence. This must be less than MAX_SEGMENTS_PER_SEQUENCE (by default 594) and communicates to the stimulation scheduler how many Segments are present for processing.

`int32_t loop_repeat`: A number representing how many times a section of user-defined Segments should loop. -1 for infinite, 0 for no loop, and positive number for a number of repeats. Note that the first run-through is not considered a loop – if you wanted a range of Segments to execute 5 times, you would specify a loop_repeat of 4 since the first iteration is not a repeat.

`uint32_t loop_start`: Inclusive index of user-defined Segments where a loop should begin.

`uint32_t loop_end`: Inclusive index of user-defined Segments where a loop should end.

`SequenceStatus status`: Status variable used by software to monitor the state of this sequence (not necessary for the user to alter this variable)

`Segments segments [MAX_SEGMENTS_PER_SEQUENCE]` : Array of user-defined Segments.

`Segment`: A structure containing the description of a single horizontal line period, many of which make up a StimSequence.

`uint32_t length`: Number of timesteps (1 / sample rate) this segment lasts for. For the first Segment of a StimSequence (index 0), this field is not applicable, as this first Segment is the default resting state when stimulation is not actively triggered.

`int32_t magnitude`: When multiplied by stim step size, magnitude of the current sourced during stimulation (can be positive or negative).

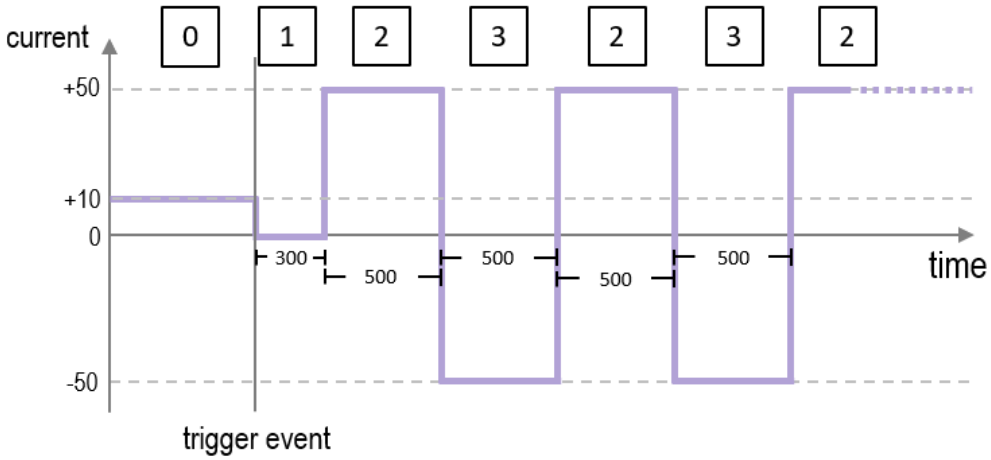
`bool fast_settle`: True if fast_settle is active for this segment, false if inactive.

`bool charge_recovery`: True if charge_recovery is active for this segment, false if inactive.

Here is an example StimSequence with its various Segments described in detail:

Example StimSequence

```
loop_start = 2; loop_end = 3; loop_repeat = -1; num_segments = 4;
```



0	length = 1, magnitude = +10
1	length = 300, magnitude = 0
2	length = 500, magnitude = +50
3	length = 500, magnitude = -50

Current units (magnitude) are stim step size, and time units (length) are timesteps (1 / sample rate). Segment 0 has a length of 1, but this first Segment's length is always N/A – this is the default resting state that the channel is before and after active stimulation, so the length is irrelevant; since this has a magnitude of +10, this channel is held at +10 until the trigger event occurs. Then, Segment 1 has a length of 300 and a magnitude of 0. Then, Segment 2 has a length of 500 and a magnitude of +50. Then, Segment 3 has a length of 500 and a magnitude of -50. Because the StimSequence has a loop_repeat value of -1, that means segments between loop_start and loop_end repeat indefinitely; for non-negative values, those Segments would repeat that many times. So after Segment 3 (loop_end), the sequence goes back to Segment 2 (loop_start) and continues repeating until the acquisition loop ends.

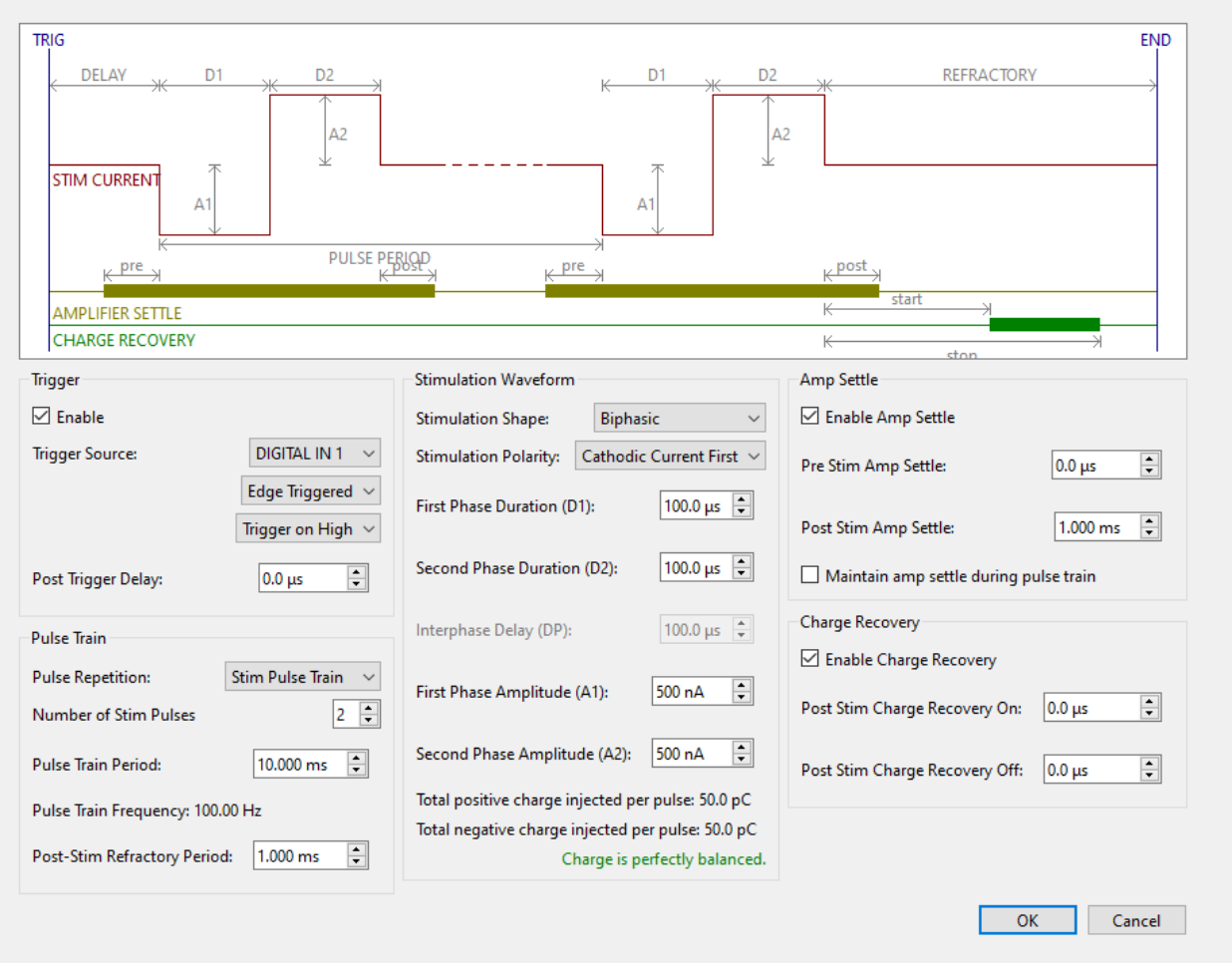
Creating Common Auto-Generated Sequences

While it's possible to manually specify StimSequence structures for complex or infinitely repeating stimulation pulses, commonly used sequences of biphasic and triphasic stimulation waveforms can be easily auto-generated by the user, so that the StimSequence and its component Segments are automatically populated by the program. Two example waveforms, one biphasic and one triphasic, are populated, validated, and created into their respective StimSequences in the *configure_stim_sequences* function in the *userfunctions.c* file. These structures could be modified by the user to fit their desired waveform shapes. If new structures are created, they can be validated and assigned to a position in the 'sequences' array in the same manner the examples are. The "validate" functions check the user-provided parameters for any rule violations (for example, assigning to a non-existent channel or setting the charge recovery "off" signal before the charge recovery "on" signal) and if any are discovered, the program enters an infinite loop, the red LED is illuminated, and the InvalidStimWaveform error is set. Users can run their program in Debug

mode to trace the function call stack to determine which specific rule violations are occurring, and then change their structure initialization accordingly. The BiphasicWaveform and TriphasicWaveform structures and their components are described in detail:

`BiphasicWaveform`: A structure fully describing the characteristics of a biphasic pulse or pulse train.

These characteristics are generally analogous to the user-configurable parameters from the Intan RHX software:



`uint8_t channel`: A number representing one of the 0-15 channels this waveform should occur on.

`uint8_t trigger_source`: A number representing the trigger source for this stim sequence. 0 (default) represents inactive or no trigger, 1 represents triggering from rising-edge activity on the GPIO defined as GPIO_EXTI13, routed to the blue button for NUCLEO boards, and any other integers are left unimplemented but can be expanded by the user to include triggers from other events, for example other GPIOs or timers.

`uint32_t num_pulses`: A number representing how many pulses this waveform should consist of. A single biphasic pulse (amplitude 1, optional interphase delay, amplitude 2) can be specified with a value of 1 – a pulse train repeating a certain number of times can be specified with a larger number. This value is limited to a maximum value of MAX_NUM_PULSES_PER_TRAIN (default 99), which can be increased by the user at the cost of increased memory usage.

`uint32_t pulse_train_period`: If num_pulses is greater than 1, a number representing the number of timesteps (each timestep is 1 / sample rate) between the start of each pulse within the pulse train. If num_pulses is equal to 1, this must be set to 0 as it's not applicable to single pulses.

`int16_t resting_amplitude`: In units of stim step size, the resting current amplitude at which this channel will be held before and after stimulation pulses (and, if interphase delay is not 0, between the first and second phases).

`int16_t first_phase_amplitude`: In units of stim step size, the current amplitude at which this channel will be held during the first stimulation phase, for a period governed by first_phase_duration.

`int16_t second_phase_amplitude`: In units of stim step size, the current amplitude at which this channel will be held during the second stimulation phase, for a period governed by second_phase_duration.

`uint32_t post_trigger_delay`: The number of timesteps (each timestep is 1 / sample rate) after the trigger event specified in trigger_source occurs, but before the first phase of stimulation occurs. Can be set to 0 for instant stimulation as soon as the trigger occurs.

`uint32_t first_phase_duration`: The number of timesteps (each timestep is 1 / sample rate) the specified channel is held at first_phase_amplitude during the first phase of stimulation. Can be set to 0 to skip the first phase.

`uint32_t interphase_delay`: The number of timesteps (each timestep is 1 / sample rate) the specified channel is held at resting_amplitude between the first and second phases of stimulation. Can be set to 0 to skip any interphase delay.

`uint32_t second_phase_duration`: The number of timesteps (each timestep is 1 / sample rate) the specified channel is held at the specified second_phase_amplitude during the second phase of stimulation. Can be set to 0 to skip the second phase.

`uint32_t refractory_period`: The number of timesteps (each timestep is 1 / sample rate) after the second stimulation phase ends for which subsequent triggers are ignored. This must be at least as large as the largest of post_stim_amp_settle, post_stim_charge_recovery_on, and post_stim_charge_recovery_off.

`bool enable_amp_settle`: Whether amp settle should be enabled at any point during this waveform. If true, the amp settle characteristics are further described with pre_stim_amp_settle, post_stim_amp_settle, and maintain_amp_settle. Whether the traditional fast settle or switching lower cutoff frequency method is used is governed by the #define TRADITIONAL_FAST_SETTLE macro in “userconfig.h”.

`uint32_t pre_stim_amp_settle`: The number of timesteps (each timestep is 1 / sample rate) before the first stimulation phase occurs for which amp settle is active. This number should not be larger than post_trigger_delay. If enable_amp_settle is false, this must be set to 0 as it’s not applicable.

`uint32_t post_stim_amp_settle`: The number of timesteps (each timestep is 1 / sample rate) after the second stimulation phase finishes for which amp settle is active. This number should not be larger than post_trigger_delay. If enable_amp_settle is false, this must be set to 0 as it’s not applicable.

`bool maintain_amp_settle`: Whether amp settle should be maintained across all pulses within a pulse train. If true, amp settle will be held high even during the inactive portion of pulse_train_period during which stimulation is not occurring. If false, amp settle will only be held high in the periods before, during, and after each individual pulse within a pulse train. If enable_amp_settle is false, this must be set to 0 as it’s not applicable.

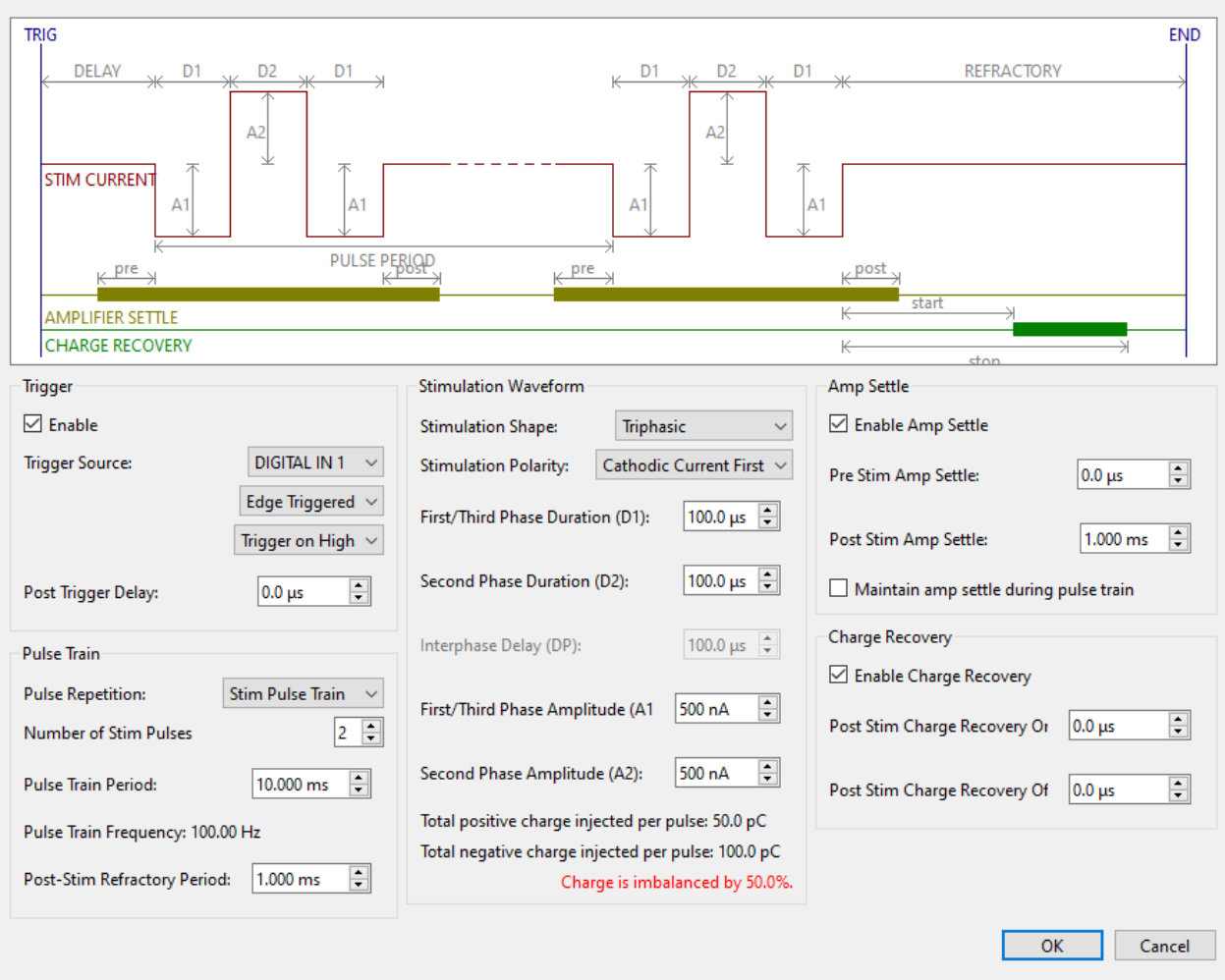
`bool enable_charge_recovery`: Whether post-stimulation charge recovery should be enabled at any point during this waveform. If true, the charge recovery characteristics are further described with post_stim_charge_recovery_on and post_stim_charge_recovery_off. Whether the charge recovery switch or current-limited charge recovery circuit is used is used is governed by the #define CHARGE_RECOVERY_SWITCH macro in “userconfig.h”.

`uint32_t post_stim_charge_recovery_on`: The number of timesteps (each timestep is 1 / sample rate) after the second stimulation phase finishes before the charge recovery function begins. This number should not be larger than post_trigger_delay, or larger than post_stim_charge_recovery_off. If enable_charge_recovery is false, this must be set to 0 as it’s not applicable.

`uint32_t post_stim_charge_recovery_off`: The number of timesteps (each timestep is 1 / sample rate) after the second stimulation phase finishes before the charge recovery function ends. This number should not be larger than post_trigger_delay, or smaller than post_stim_charge_recovery_on. If enable_charge_recovery is false, this must be set to 0 as it’s not applicable.

`TriphasicWaveform`: A structure fully describing the characteristics of a triphasic pulse or pulse train.

These characteristics are generally analogous to the user-configurable parameters from the Intan RHX software:



`uint8_t channel`: A number representing one of the 0-15 channels this waveform should occur on.

`uint8_t trigger_source`: A number representing the trigger source for this stim sequence. 0 (default) represents inactive or no trigger, 1 represents triggering from rising-edge activity on the GPIO defined as GPIO_EXTI13, routed to the blue button for NUCLEO boards, and any other integers are left unimplemented but can be expanded by the user to include triggers from other events, for example other GPIOs or timers.

`uint32_t num_pulses`: A number representing how many pulses this waveform should consist of. A single biphasic pulse (amplitude 1, amplitude 2, amplitude 3) can be specified with a value of 1 – a pulse train repeating a certain number of times can be specified with a larger number. This value is limited to a maximum value of MAX_NUM_PULSES_PER_TRAIN (default 99), which can be increased by the user at the cost of increased memory usage.

`uint32_t pulse_train_period`: If num_pulses is greater than 1, a number representing the number of timesteps (each timestep is 1 / sample rate) between the start of each pulse within the pulse train. If num_pulses is equal to 1, this must be set to 0 as it's not applicable to single pulses.

`int16_t resting_amplitude`: In units of stim step size, the resting current amplitude at which this channel will be held before and after stimulation pulses.

`int16_t first_phase_amplitude`: In units of stim step size, the current amplitude at which this channel will be held during the first stimulation phase, for a period governed by first_phase_duration.

`int16_t second_phase_amplitude`: In units of stim step size, the current amplitude at which this channel will be held during the second stimulation phase, for a period governed by second_phase_duration.

`int16_t third_phase_amplitude`: In units of stim step size, the current amplitude at which this channel will be held during the third stimulation phase, for a period governed by third_phase_duration.

`uint32_t post_trigger_delay`: The number of timesteps (each timestep is 1 / sample rate) after the trigger event specified in trigger_source occurs, but before the first phase of stimulation occurs. Can be set to 0 for instant stimulation as soon as the trigger occurs.

`uint32_t first_phase_duration`: The number of timesteps (each timestep is 1 / sample rate) the specified channel is held at first_phase_amplitude during the first phase of stimulation. Can be set to 0 to skip the first phase.

`uint32_t second_phase_duration`: The number of timesteps (each timestep is 1 / sample rate) the specified channel is held at the specified second_phase_amplitude during the second phase of stimulation. Can be set to 0 to skip the second phase.

`uint32_t third_phase_duration`: The number of timesteps (each timestep is 1 / sample rate) the specified channel is held at the specified third_phase_amplitude during the third phase of stimulation. Can be set to 0 to skip the third phase.

`uint32_t refractory_period`: The number of timesteps (each timestep is 1 / sample rate) after the third stimulation phase ends for which subsequent triggers are ignored. This must be at least as large as the largest of post_stim_amp_settle, post_stim_charge_recovery_on, and post_stim_charge_recovery_off.

`bool enable_amp_settle`: Whether amp settle should be enabled at any point during this waveform. If true, the amp settle characteristics are further described with pre_stim_amp_settle, post_stim_amp_settle, and maintain_amp_settle. Whether the traditional fast settle or switching lower cutoff frequency method is used is governed by the #define TRADITIONAL_FAST_SETTLE macro in “userconfig.h”.

`uint32_t pre_stim_amp_settle`: The number of timesteps (each timestep is 1 / sample rate) before the first stimulation phase occurs for which amp settle is active. This number should not be larger than post_trigger_delay. If enable_amp_settle is false, this must be set to 0 as it’s not applicable.

`uint32_t post_stim_amp_settle`: The number of timesteps (each timestep is 1 / sample rate) after the third stimulation phase finishes for which amp settle is active. This number should not be larger than post_trigger_delay. If enable_amp_settle is false, this must be set to 0 as it’s not applicable.

`bool maintain_amp_settle`: Whether amp settle should be maintained across all pulses within a pulse train. If true, amp settle will be held high even during the inactive portion of pulse_train_period during which stimulation is not occurring. If false, amp settle will only be held high in the periods before, during, and after each individual pulse within a pulse train. If enable_amp_settle is false, this must be set to 0 as it’s not applicable.

`bool enable_charge_recovery`: Whether post-stimulation charge recovery should be enabled at any point during this waveform. If true, the charge recovery characteristics are further described with post_stim_charge_recovery_on and post_stim_charge_recovery_off. Whether the charge recovery switch or current-limited charge recovery circuit is used is used is governed by the #define CHARGE_RECOVERY_SWITCH macro in “userconfig.h”.

`uint32_t post_stim_charge_recovery_on`: The number of timesteps (each timestep is 1 / sample rate) after the third stimulation phase finishes before the charge recovery function begins. This number should not be larger than post_trigger_delay, or larger than post_stim_charge_recovery_off. If enable_charge_recovery is false, this must be set to 0 as it’s not applicable.

`uint32_t post_stim_charge_recovery_off`: The number of timesteps (each timestep is 1 / sample rate) after the third stimulation phase finishes before the charge recovery function ends. This number should not be larger than post_trigger_delay, or smaller than post_stim_charge_recovery_on. If enable_charge_recovery is false, this must be set to 0 as it’s not applicable.

Triggering Stimulation from Interrupts

In addition to properly setting up a stimulation sequence manually via the StimSequence struct or automatically via the BiphasicWaveform or TriphasicWaveform structs, it's also necessary to set up the stimulation triggers to flag when certain events occur. By default, the example stimulation waveforms are triggered off a rising edge on the GPIO pin routed to the blue button on the NUCLEO board, but any interrupt (for example, from a timer if stimulation needs to trigger at a fixed frequency) can be used to flag a trigger.

In the STM32-generated interrupt .c source file ("stm32u5xx_it.c" or "stm32h7xx_it.c"), the "process_trigger(N)" function needs to be called with N as a unique integer representing a specific trigger. See how process_trigger(1) is called by default for the EXTI trigger as an example, using either HAL or LL. This N value should match the trigger_source value for this stimulation sequence or auto-generated biphasic/triphasic waveform. Note that the maximum value for N is limited to NUM_STIM_SEQUENCES, (defined in "userconfig.h") which is by default limited to 2. This number can be increased by the user, but this increases the amount of required memory, so alterations may be necessary to avoid running out memory during execution.

Observing the Compliance Monitor

A useful feature of the RHS chip is a compliance monitor present on each channel to flag when the compliance voltage is reached. If, due to a high electrode impedance, the current specified cannot be provided without reaching the VSTIM+ and VSTIM- voltages (by default, these projects are configured for ±7 V), the RHS chip flags the compliance monitor for this channel so the user can detect this problematic condition. In "rhsinterface.c", the function "process_compliance_data" demonstrates how any READ commands of the compliance monitor can have their results processed and handled by the software. By default, if any compliance monitor is flagged, the software writes the Compliance_Monitor_Pin high and continues execution, but users may want to alter this behavior.

An important note is that when a compliance monitor READ command is issued, the software expects to find a result with the returned data conforming to the result of a READ command – even if a compliance monitor violation occurs, the SPI word returning that data should have a valid READ prefix. If the returned data has some other prefix (usually indicating a problem with the SPI interface, an Intan chip not actually being present, or for some reason the software failing to locate a 2-command later result from a READ command), a specific error InvalidComplianceReading is triggered, and an infinite loop is entered. If the user wishes to change this behavior to handle this condition (for example, be able to run the software without an Intan chip for testing purposes without entering this error loop), they can change the contents of "locate_compliance_result" in "rhsinterface.c", or to bypass this compliance monitor check entirely, comment out the "process_compliance_data()" call in "spi_txrx_cplt_callback" in "rhsinterface.c".

Overview of Program Flow

This Intan STM32 example code was developed using STM32CubeIDE, and uses HAL or LL drivers (this can easily be changed by the user) to configure and control various peripherals. There are six distinct projects:

1. U5 rhd_acquisition – communication using standard SPI to run acquisition with an RHD2216 or RHD2132 and transmit acquired data (either in real-time alongside acquisition or offline, after a set period of time) via USART with an STM32U5 chip.
2. U5 rhd2164_acquisition – communication using DDR SPI to run acquisition with an RHD2164 and transmit acquired data (either in real-time alongside acquisition or offline, after a set period of time) via USART with an STM32U5 chip.
3. U5 rhs_acquisition – communication using RHS SPI to run acquisition with an RHS2116, transmit acquired data (either in real-time alongside acquisition or offline, after a set period of time) via USART with an STM32U5 chip. Additionally, a rising-edge event on a dedicated GPIO pin (by default, blue user button on the NUCLEO board) during acquisition triggers user-configurable stimulation pulses.
4. H7 rhd_acquisition – communication using standard SPI to run acquisition with an RHD2216 or RHD2132 and transmit acquired data (either in real-time alongside acquisition or offline, after a set period of time) via USART with an STM32H7 chip.
5. H7 rhd2164_acquisition – communication using DDR SPI to run acquisition with an RHD2164 and transmit acquired data (either in real-time alongside acquisition or offline, after a set period of time (via USART with an STM32H7 chip.
6. H7 rhs_acquisition – communication using RHS SPI to run acquisition with an RHS2116, transmit acquired data (either in real-time alongside acquisition or offline, after a set period of time) via USART with an STM32H7 chip. Additionally, a rising-edge event on a dedicated GPIO pin (by default, blue user button on the NUCLEO board) during acquisition triggers user-configurable stimulation pulses.

These all follow the same general structure, and only have differences based on the specific implementations for the U5 or H7 chip, whether standard SPI, DDR SPI, or RHS SPI is used, and which registers are present on the specific Intan chip.

The code first configures and initializes the MCU peripherals with various auto-generated functions, which the STM32CubeIDE environment creates based on various settings in the .ioc file. Parameters for configuring the Intan chip are then used to determine values for each of the RHD/RHS registers, and these are written to the chip using WRITE commands over the SPI bus. For RHD, a list of 32 CONVERT commands (one for each channel of an RHD chip) and a list of three auxiliary commands (one of which continually re-writes the above-determined register values to the RHD chip) are created and stored in memory as *command_sequence_MOSI* for later use. For RHS, there are 16 CONVERT commands (one for each channel) and a list of four auxiliary commands. The green LED is illuminated to indicate when data acquisition starts, and a timer interrupt is enabled. The various projects run at different clock cycles, but for each project the INTERRUPT_TIM peripheral is configured with compensating settings to trigger at 20 kHz. We will refer to this timer period as the sample period. The program then enters a data acquisition loop.

This data acquisition loop will only break when the *loop_escape()* function returns 1, which will not occur until the number of acquired samples reaches 10000, meaning 0.5 seconds of data have been acquired. Until that time, this main loop will repeatedly write a dedicated pin *Main_Monitor_Pin* high, which can be used to monitor when this main loop is processing. (Other functions that trigger due to interrupts will keep this pin low for the duration of their execution). Because timer interrupts had just been enabled, this loop will consistently pause every sample period to execute *sample_processing_routine()*.

The *sample_processing_routine()* function executes once per sample period and begins sending a sequence of SPI commands. By default, for RHD: this function sends 35 commands: 32 CONVERT commands + 3 auxiliary commands, where each command is a 16-bit SPI word. For RHS: this function sends 20 commands: 16 CONVERT commands + 3 auxiliary commands, where each command is a 32-bit SPI word. In addition to beginning the SPI transfer, this routine also writes the *Main_Monitor_Pin* low. (The main loop will write it high once it begins executing again.) The function also writes a dedicated *Interrupt_Monitor_Pin* high only for the duration of the function, and does some error checking. The SPI transfer uses DMA to iterate through the full 35 or 20-command list and this routine only begins the transfer, so by the time the routine finishes the command sequence will have only just begun.

An important detail of *sample_processing_routine()* is that it checks to make sure that the previous SPI sequence is complete; if the variable *command_transfer_state* (discussed below) is still *TRANSFER_WAIT* from the previous sequence, then the critical *SampleClip* error has occurred. This error, as well as methods to avoid it, are discussed in more detail later. Briefly, this error indicates that the sample period is shorter than the time required for each SPI sequence, so every sample period the next sequence is being triggered before the previous one finishes. This can be solved by extending the sample period (i.e., using a lower sample

rate) or speeding up each SPI sequence (e.g., sample fewer channels, use fewer AUX commands, or speed up the SPI transfer itself, if possible).

Eventually, one complete SPI transfer sequence will end. The exact way this is detected varies slightly based on whether LL or HAL drivers are used, but in both cases an interrupt triggers the function *spi_rx_cplt_callback()* or *spi_txrx_cplt_callback()*. The *write_data_to_memory()* function is then executed, which is a user-changeable function that by default writes the result of a single CONVERT command (i.e., a sample from a single channel from an RHD2216, RHD2132, or RHS2116, or two A and B samples from an RHD2164) to memory, but this only happens if OFFLINE_TRANSFER is defined. Then, *transmit_data_realtime()* is executed, which by default transmits a few channels of data over USART, using DMA for improved efficiency, but this only happens if OFFLINE_TRANSFER is not defined. The variable *command_transfer_state* is changed; it is set to *TRANSFER_WAIT* once the transfer begins, *TRANSFER_COMPLETE* once the transfer finishes, and *TRANSFER_ERROR* if some error was detected.

The main loop will continue to run, pausing for an interrupt every sample period, until *loop_escape()* returns 1. If OFFLINE_TRANSFER is not defined, *loop_escape()* will always return 0 so the loop will never escape, so the program will simply continue looping with occasional interrupts. This is suitable behavior for long-term data acquisition that is not directly saved to memory, but transmitted elsewhere in real time every sample period. The example code is set to either escape the loop after one second of data acquisition and disable further timer interrupts (OFFLINE_TRANSFER is defined) or stay within the loop, streaming data in real time alongside acquisition (OFFLINE_TRANSFER is not defined).

If the loop is escaped (only occurs if OFFLINE_TRANSFER is defined), the user-changeable function *transmit_data_offline()* is called, which by default sends the 10000 accumulated samples from a chosen channel out via USART in many small DMA transfers. Note that the on-chip memory limitations of the STM32 will not allow for long recordings (exact memory limitations depend on the specific STM32 chip used), especially at high sample rates and channel counts, unless the data is transmitted elsewhere so the used memory can be freed.

For the RHS projects, if AUTO_STIM_CMD_MODE is defined, then a rising-edge signal on a specific GPIO (mapped to the blue user button on all NUCLEO projects so that pressing this button acts as a rising-edge) during acquisition will trigger user-specified stimulation pulses on 2 channels. The channels that are activated, the pulses themselves, and the triggers can be completely configured by the user.

Finally, the green LED is switched off, indicating both data acquisition and transmission have completed, and the program enters a final infinite loop.

I/O Pins

This example code was developed on an STMicroelectronics **NUCLEO-U5A5ZJ-Q** development board (for U5 code), and a **NUCLEO-H723ZG** development board (for H7 code). The I/O pins were chosen to be easily accessible when using these boards. Any changes to I/O pin assignments should be made through the *rhd_acquisition.ioc*, *rhd2164_acquisition.ioc* or *rhs_acquisition*.ioc file.

U5 rhd_acquisition

SPI (SPI)

NSS ($\overline{CS}$): PA4 – should be connected to Intan CS.

SCK (SCLK): PC10 – should be connected to Intan SCLK.

MISO: PC11 – should be connected to Intan MISO.

MOSI: PC12 – should be connected to Intan MOSI.

Timing

Interrupt_Monitor: PD9 - written high each sample period, so its frequency represents amplifier sample rate.

Main_Monitor: PC8 - written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.

User Communication

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

LED_GREEN (Acquisition underway): PC7

LED_RED (Error detected – check bits 3-0): PG2

U5 rhd2164_acquisition

TRANSMIT_SPI (SPI3)

NSS ($\overline{CS}$): PA4 – should be wired to RECEIVE_SPI NSS (PB0) so that RECEIVE_SPI shares the same NSS signal as TRANSMIT_SPI, as well as RECEIVE_SCLK_TIM ETR (PE7) so that Pseudo-SCLK generation is triggered off NSS. Also, should be connected to Intan CS.

SCK (SCLK): PC10 – should be connected to Intan SCLK.

MOSI: PC12 – should be connected to Intan MOSI.

RECEIVE_SPI (SPI1)

NSS ($\overline{CS}$): PB0 – should be wired to TRANSMIT_SPI NSS (PA4) so that RECEIVE_SPI shares the same NSS signal as TRANSMIT_SPI.

SCK (SCLK): PC10 – should be wired to Pseudo-SCLK signal generated by RECEIVE_SCLK_TIM CH1 (PE9).

MOSI: PA7 – RECEIVE_SPI is configured as slave while the Intan naming convention refers to the MCU as master, so should be connected to Intan MISO.

Timing

Interrupt_Monitor: PD9 - written high each sample period, so its frequency represents amplifier sample rate.

Main_Monitor: PC8 - written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.

TIM1_ETR (RECEIVE_SCLK_TIM External Trigger): PE7 – should be wired to TRANSMIT_SPI NSS (PA4) so RECEIVE_SCLK_TIM timer, which generates a 32-pulse Pseudo-SCLK signal, is triggered off NSS.

TIM1_CH1 (RECEIVE_SCLK_TIM Channel 1 Output): PE9 – should be wired to RECEIVE_SPI SCK (PC10) so RECEIVE_SCLK_TIM output, 32-pulse Pseudo-SCLK signal, feeds into RECEIVE_SPI as SCLK.

User Communication

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

LED_GREEN (Acquisition underway): PC7

LED_RED (Error detected – check bits 3-0): PG2

U5 rhs_acquisition

SPI (SPI1)

NSS ($\overline{CS}$): PA4 – should be connected to Intan CS.

SCK (SCLK): PA1 – should be connected to Intan SCLK.

MISO: PA6 – should be connected to Intan MISO.

MOSI: PA7 – should be connected to Intan MOSI.

Timing

Interrupt_Monitor: PD9 - written high each sample period, so its frequency represents amplifier sample rate.

Main_Monitor: PC8 - written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.

User Communication

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

LED_GREEN (Acquisition underway): PC7

LED_RED (Error detected – check bits 3-0): PG2

Stimulation

Compliance Monitor: PG13 – written high when any channel's compliance monitor is set high, indicating that the electrode voltage becomes so high or low that it becomes impossible to deliver the specified current.

GPIO_EXTI13: PC13 – routed to blue button on NUCLEO boards by default, and during acquisition if AUTO_STIM_CMD_MODE is defined, triggers stimulation.

H7 rhd_acquisition

SPI (SPI3)

NSS ($\overline{CS}$): PA4 – should be connected to Intan CS.

SCK (SCLK): PC10 – should be connected to Intan SCLK.

MISO: PC11 – should be connected to Intan MISO.

MOSI: PB2 – should be connected to Intan MOSI.

Timing

Interrupt_Monitor: PD9 - written high each sample period, so its frequency represents amplifier sample rate.

Main_Monitor: PC8 - written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.

User Communication

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

LED_GREEN (Acquisition underway): PC7

LED_RED (Error detected – check bits 3-0): PG2

H7 rhd2164_acquisition

TRANSMIT_SPI (SPI3)

NSS ($\overline{CS}$): PA4 – should be wired to RECEIVE_SPI NSS (PA15) so that RECEIVE_SPI shares the same NSS signal as TRANSMIT_SPI, as well as CS_DELAY_TIM ETR (PA0) so that Pseudo-SCLK generation is triggered off NSS. Also, should be connected to Intan CS.

SCK (SCLK): PC10 – should be connected to Intan SCLK.

MOSI: PB2 – should be connected to Intan MOSI.

RECEIVE_SPI (SPI1)

NSS ($\overline{CS}$): PA15 – should be wired to TRANSMIT_SPI NSS (PA4) so that RECEIVE_SPI shares the same NSS signal as TRANSMIT_SPI.

SCK (SCLK): PA5 – should be wired to Pseudo-SCLK signal generated by RECEIVE_SCLK_TIM CH1 (PE9).

MOSI: PD7 – RECEIVE_SPI is configured as slave while the Intan naming convention refers to the MCU as master, so should be connected to Intan MISO.

Timing

Interrupt_Monitor: PD9 - written high each sample period, so its frequency represents amplifier sample rate.

Main_Monitor: PC8 - written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.

TIM2_ETR (CS_DELAY_TIM External Trigger): PA0 – should be wired to TRANSMIT_SPI NSS (PA4) so CS_DELAY_TIM timer, which ultimately leads to generation of a 32-pulse Pseudo-SCLK signal, is triggered off NSS.

TIM2_CH3 (CS_DELAY_TIM Channel 3 Output): PB10 – not connected to any hardware, but can be probed to see CS_DELAY_TIM output visualize intentional delay added between NSS and 32-pulse Pseudo-SCLK signal. Internally (through .ioc) connected to trigger RECEIVE_SCLK_TIM.

TIM1_CH1 (RECEIVE_SCLK_TIM Channel 1 Output): PE9 – should be wired to RECEIVE_SPI SCK (PA5) so RECEIVE_SCLK_TIM output, 32-pulse Pseudo-SCLK signal, feeds into RECEIVE_SPI as SCLK.

User Communication

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

LED_GREEN (Acquisition underway): PB0

LED_RED (Error detected – check bits 3-0): PB14

H7 rhs_acquisition

SPI (SPI3)

NSS ($\overline{CS}$): PA4 – should be connected to Intan CS.

SCK (SCLK): PC10 – should be connected to Intan SCLK.

MISO: PC11 – should be connected to Intan MISO.

MOSI: PB2 – should be connected to Intan MOSI.

Timing

Interrupt_Monitor: PD9 - written high each sample period, so its frequency represents amplifier sample rate.

Main_Monitor: PC8 - written high as long as main loop is processing, so its duty cycle approximates free CPU clock cycles that could be used for other processing tasks.

User Communication

ErrorCode_Bit_3: PE0

ErrorCode_Bit_2: PG8

ErrorCode_Bit_1: PG5

ErrorCode_Bit_0: PG6

LED_GREEN (Acquisition underway): PB0

LED_RED (Error detected – check bits 3-0): PB14

Stimulation

Compliance Monitor: PG13 – written high when any channel's compliance monitor is set high, indicating that the electrode voltage becomes so high or low that it becomes impossible to deliver the specified current.

GPIO_EXTI13: PC13 – routed to blue button on NUCLEO boards by default, and during acquisition if AUTO_STIM_CMD_MODE is defined, triggers stimulation.

Connecting an Intan Chip

The 'rhd_acquisition' example code was designed to work with an Intan RHD2132 32-channel amplifier chip. The RHD2216 chip can also be used, but unless the code is modified, half of the CONVERT commands per sample period will not correspond to real channels. The 'rhd2164_acquisition' example code was designed to work with an Intan RHD2164 64-channel amplifier chip. The 'rhs_acquisition' example code was designed to work with an Intan RHS2116 amplifier chip.

Intan headstages are set to communicate using LVDS (low voltage differential signaling) signals on the SPI bus, while most microcontrollers use standard non-LVDS SPI signals. To connect an Intan headstage to the STM32, you must either use an Intan **LVDS adapter board** (part #C3490), or tie the **LVDS_en** pin on the Intan chip to ground to disable LVDS signaling. On most Intan headstages the LVDS_en pin is hard-wired to VDD and it is impractical to cut the trace, but in the C3335 headstage (which uses a RHD2132 chip with access to 16 of the 32 amplifiers), there is a zero-ohm resistor labeled R4 that can be removed to set LVDS_en low. Similarly, the 64-channel headstages have a zero-ohm resistor labeled R4 that can be removed, but on the RHD2164 LVDS_en has an internal pull-up resistor, so the LVDS_en pad should also be connected to GND. Finally, the 16-channel RHS headstages have a zero-ohm resistor labeled R5 that can be removed to set LVDS_en low.

Note that if LVDS signaling is disabled, standard CMOS signaling is used instead, which makes reliable transmission of high-frequency data over long wires challenging due to signal reflections. For this reason, if LVDS is disabled, it is critical to keep the SPI wires between the MCU and the RHD chip as short as possible, and/or reduce the SPI data transmission rate.

Additional helpful accessories for development are the RHD SPI cable adapter (part #C3430) and the RHS SPI cable adapter (part #M4430), which plugs into the SPI connector on an Intan headstage and breaks out each signal to a circuit board for easy soldering. Each of the 12/16 signals is assigned its own hole for soldering, from B1 - B6/B8 (bottom row) and T1 - T6/T8 (top row). (The SPI cable adapter is not necessary if the LVDS adapter board is used, because the LVDS adapter board includes an SPI cable connector.)

Assuming LVDS signaling is disabled and SPI signals are kept short, the following connections can be used along with the example program to interface an STM32U5/H7 with an RHD headstage through an SPI cable adapter board. Note that with LVDS disabled, all the (–) polarity signals are unused, and the (+) polarity signals carry the standard CMOS signal.

For RHD:

RHD SPI adapter board pin number	Signal	STM32U5 pin number	STM32H7 pin number
B1	$\overline{\text{CS}}$+	PA4	PA4
B2	SCLK+	PC10	PC10
B3	MOSI+	PC12	PB2
B4	MISO1+	PC11 (rhd_acquisition) PA7 (rhd2164_acquisition)	PC11 (rhd_acquisition) PD7 (rhd2164_acquisition)
B5	MISO2+ (unused)	-	-
B6	VDD	3V3	3V3
T1	$\overline{\text{CS}}$– (unused)	-	-
T2	SCLK– (unused)	-	-
T3	MOSI– (unused)	-	-
T4	MISO1– (unused)	-	-
T5	MISO2– (unused)	-	-
T6	GND	GND	GND

For RHS:

RHS SPI adapter board pin number	Signal	STM32U5 pin number	STM32H7 pin number
B1	VSTIM– (–7V)	-	-
B2	SCLK–	-	-
B3	SCLK+	PA1	PC10
B4	MOSI1–	-	-
B5	MOS1+	PA7	PB2
B6	MISO1–	-	-
B7	MISO+	PA6	PC11
B8	VDD (+3.3V)	3V3	3V3
T1	VSTIM+ (+7V)	-	-
T2	MISO2+	-	-
T3	MISO2–	-	-
T4	MOSI2+	-	-
T5	MOSI2–	-	-
T6	$\overline{\text{CS}}$+	PA1	PA4
T7	$\overline{\text{CS}}$–	-	-
T8	GND	GND	GND

Description of User-Changeable Sections of Example Code

The example code was designed with user modifications in mind. While users are free to modify any and all files, there are three specific files that are intended to be modified to most effectively alter the program's functionality: **userconfig.h**, **userfunctions.h**, and **userfunctions.c**. Each of these files and the changes that may be made to them are discussed in detail below. Note that the .ioc file, which governs pin and peripheral configuration, is not included here and should instead be changed directly through STM32CubeIDE's UI if the user wants to use different I/O pins, different peripherals, or different parameters for those peripherals (e.g., to change SPI baud rate or timer-generated sample rate).

userconfig.h

The user changes in this file are parameters set with `#define` preprocessor directives. These parameters are used throughout various files of the project, but most aspects of the program that users will want to alter can simply be set here. For example, the number of channels converted per sequence, how long to acquire data for, and which channels should have their samples stored. Those that require more involved changes (for example, sample rate) get further explanation in their sections.

`#define USE_HAL`

If this line is left uncommented, the code is compiled for compatibility with HAL (Hardware Abstraction Layer) drivers for all peripherals. It is important that if this is left uncommented, the user navigates to the STM32CubeIDE IOC viewer -> Project Manager -> Advanced Settings, and confirms that all the peripherals that are directly used in user code (GPIO, GPDMA/DMA, USART, TIM, SPI) are set to HAL. In contrast, if this line is commented out, the code is compiled for compatibility with LL (Low Layer) drivers, and these peripherals should all be set to LL instead.

In general, HAL drivers are more user-friendly, simple to work with, and largely compatible even across different STM32 series. LL drivers require more device-specific implementation of basic functions, and are closer to direct manipulation of registers, so the code using LL tends to be more complex but specialized to the chip, typically boosting performance.

`#define OFFLINE_TRANSFER`

If this line is left uncommented, the program will not transmit acquired data via USART during the main acquisition loop, and only write acquired data to memory. Then, the program will escape the main acquisition loop after NUMBER_OF_SECONDS_TO_ACQUIRE (default 0.5), and transmit acquired data from memory to USART after acquisition has completed. Otherwise, if this line is commented out, the program will transmit acquired data via USART immediately instead of writing it to memory, and the program will remain in the main acquisition loop indefinitely.

`#define ERROR_DETECTED_PORT LED_RED_GPIO_Port`

`#define ERROR_DETECTED_PIN LED_RED_Pin`

These two lines specify the port and pin address of the "Error Detection" pin. By default, this is set to the pin that routes to a red LED on the Nucleo board. If another pin is desired to be used instead, that pin's Port and Pin addresses should be changed here.

`#define SAMPLE_DC_AMPS` (RHS only)

If this line is left uncommented, DC amplifiers are sampled alongside AC amplifiers for each CONVERT command. Since MISO results are 32 bits, there both the AC and DC results for a single channel fit into a single SPI word. If commented out, then the bits reserved for DC results are all 0s. A very small amount of power may be saved by not sampling DC amplifiers, but there are no other advantages.

`#define TRADITIONAL_FAST_SETTLE` (RHS only)

If this line is left uncommented, the traditional fast settle grounding switch governed by RHS register 10 is used for any stimulation Segments where fast_settle is true. If commented, then the lower cutoff frequency switch method governed by RHS register 12 is used instead. See the "Amplifier Stimulus Artifact Recovery" section of the RHS chip datasheet for a detailed description of these two methods to recover the amplifiers from stimulus artifacts.

`#define CHARGE_RECOVERY_SWITCH` (RHS only)

If this line is left uncommented, the charge recovery switch governed by RHS register 46 is used for any stimulation Segments where charge_recovery is true. If commented, then the current-limited charge recovery circuit governed by RHS register 48 is used instead. See the "Charge Recovery Switch" and "Current-Limited Charge Recovery Circuit" sections of the RHS chip datasheet for a detailed description of these two methods to balance charge after stimulation pulses.

`#define CONVERT_COMMANDS_PER_SEQUENCE 32/16` (RHD/RHS)

This line specifies that for every sample period, this many CONVERT commands will be sent within a single sequence. The order of these CONVERT commands can be customized in *configure_convert_commands()* in *userfunctions.c*, but if left unchanged, this will be channels 0-31/15 in ascending order. Note that for RHD2164 acquisition, each CONVERT command yields data from two channels, so if the default 32 CONVERT_COMMANDS_PER_SEQUENCE is used, all 64 channels are actually sampled.

`#define AUX_COMMANDS_PER_SEQUENCE 3/4` (RHD/RHS)

This line specifies that for every sample period, after the CONVERT commands, this many auxiliary commands will be sent within a single sequence. The contents of these auxiliary commands can be customized in *configure_aux_commands()* in *userfunctions.c*, but if left unchanged, this will have command 1 cycle through all RHD/RHS registers, re-writing each according to software-configured values, and the remaining commands simply repeat dummy READ commands on ROM registers.

For RHS, if AUTO_STIM_CMD_MODE is defined, then instead these 4 auxiliary command slots are used for on-the-fly WRITE commands to stimulation control registers during acquisition so that pre-defined stimulation sequences can be executed.

The total number of commands sent in a sequence of a single sample period is CONVERT_COMMANDS_PER_SEQUENCE + AUX_COMMANDS_PER_SEQUENCE. By default, for RHD this is 32 + 3 = 35, so at every sample period, there will be a sequence of 35 individual 16-bit SPI command words. For RHS, this is 16 + 4 = 20, so that at every sample period, there will be a sequence of 20 individual 32-bit SPI command words.

`#define AUX_COMMAND_LIST_LENGTH 128`

This line specifies how many auxiliary commands are contained in a single auxiliary command list. Each of the AUX_COMMANDS_PER_SEQUENCE (by default three) has its own slot, and every sequence iteration (which happens once per sample period), all slots advance by one through their lists. Once the end of the list length (dictated by this parameter) is reached, all lists are reset to zero, and the next sequence all slots will begin at the beginning of their list. In the default example, slot 0 reprograms most of the RHD RAM registers. Because all lists are 128 commands long, when a specific write occurs it can be expected that exactly 128 samples later, that write will be repeated during the next cycle of the aux command list.

Note that all auxiliary commands will have this same length, with the exception of Zcheck_DAC command lists, which have variable lengths because different DAC output signal frequencies will require different numbers of commands. Progress through these command lists is tracked separately, and will loop back to the start at some variable rate depending on frequency. All other command lists will loop back to their start every AUX_COMMAND_LIST_LENGTH number of samples.

Also note that this does not apply to RHS when AUTO_STIM_CMD_MODE is defined, as instead of looping through auxiliary command lists, the auxiliary command slots are used for on-the-fly register WRITE commands to execute stimulation sequences.

`#define AUTO_STIM_CMD_MODE` (RHS only)

If this line is left uncommented, the default 4 auxiliary commands per timestep are reserved for sending WRITE commands in real time to the RHS chip. This is necessary to allow for user-specified stimulation sequences to be executed, as the WRITE commands that take place here are what allow the stimulation registers (for instance, stimulation magnitude, polarity, etc.) to change.

If this line is commented out, then these command slots are instead occupied by the auxiliary command lists set up in *configure_aux_commands()*, in *userfuntions.c*, effectively disabling stimulation and treating the auxiliary commands the same as the RHD projects.

```
#define NUM_STIM_SEQUENCES     2       (RHS only)
```

This line specifies how many unique stimulation sequences can be set up. The default of two allows for two unique channels (by default 8 and 9) to be set up for stimulation. While not typical, it's possible for multiple sequences to be assigned to a single channel – however, this leads to potential clashes if multiple sequences are active at the same time for a specific channel, for which the user write some control logic to govern priorities. The default of 2 can be increased arbitrarily, but significant amounts of memory must be dedicated to each stim sequence, so ensure that both at compile time (compiler-level errors) and at runtime (red LED illuminating with OutOfMemoryError occurring), no memory errors occur.

```
#define NUMBER_OF_SECONDS_TO_ACQUIRE      0.5
```

This line specifies how seconds of acquisition should occur before the main acquisition loop is escaped (only valid if OFFLINE_TRANSFER is defined, otherwise this parameter has no effect). By default, after half a second of acquisition has occurred (10000 samples from a single channel at the default sample rate of 20 kHz) the main acquisition loop will exit, the data will be transmitted via USART, and the program will terminate. On-chip RAM is limited, so setting this number excessively high will cause the chip to run out of memory during program execution, resulting in the OutOfMemoryError.

```
#define FIRST_SAMPLED_CHANNEL 8 (or) 5
```

This line specifies which of the 32 RHD or 16 RHS amplifier channels is selected as the starting point of the group of channels that have their samples transmitted via USART. For rhd_acquisition, the default value is 8, for compatibility with the RHD2132 16-channel headstage (which has 32 amplifier channels on the chip, but only 16 of these, 8-23, are routed to the electrode connector on the PCB). For rhd2164_acquisition, the default value is 5, an arbitrary choice. For rhs_acquisition, the default value is 8.

```
#define NUM_SAMPLED_CHANNELS  4
```

This line specifies how many of the 32 RHD or 16 RHS amplifier channels are actually sampled (acquired and transmitted via USART). By default, this value is used to count up starting from the channel with number FIRST_SAMPLED_CHANNEL. For instance, if FIRST_SAMPLED_CHANNEL is 8 and NUM_SAMPLED_CHANNELS is 4, then channels 8, 9, 10, and 11 will be sampled and transmitted via USART. For rhd2164_acquisition, due to DDR each CONVERT command results in samples from 2 channels, so the actual number of acquired channels is NUM_SAMPLED_CHANNELS * 2, with each channel from 0-31 also returning that channel + 32. In the above example, with an RHD2164 chip, channels 8, 9, 10, 11, 40, 41, 42, and 43 will be sampled and transmitted via USART. Note that both USART throughput and RAM memory limit how large this value can be, so caution is advised when increasing this number – make sure to monitor for any runtime errors and verify data integrity if sampling additional channels.

userfunctions.h

This header file contains the declarations of some functions in *userfunctions.c*, but also a few static inline functions that are short and simple enough to go directly in this .h file. Each of these functions is discussed below.

```
static inline void wait_ms(int duration)
```

This function is called very early in the program's life, before any initialization of the RHD chip, to give plenty of time after program upload before any meaningful code is executed. It waits for 'duration' milliseconds, and can be used to intentionally insert a delay. It is recommended to never call this from within an interrupt function.

Users are not likely to change this function, but may find it useful to call in certain situations where intentional delay is desired, as long as any interactions with interrupts are accounted for.

```
static inline void enable_interrupt_timer(bool enable)
```

This function is called directly before the beginning of the main acquisition loop, with enable = true, and directly after the end of the loop, with enable = false. The function either starts or stops the timer and its ability to issue

an interrupt when the timer reaches its target counter value. The exact implementation differs based on HAL and LL drivers, but generally the same behavior is achieved.

Users are not likely to change this function unless drastically changing how timers are used to generate interrupts.

userfunctions.c

This file contains the implementations of various functions that are likely to be changed by the user to affect the behavior of the example program. Each of these functions is discussed below.

`int loop_escape(void)`

This function determines under what conditions the main acquisition loop is exited, and is called both within the main loop and within the interrupt routine to make sure that this condition's fulfilment is detected immediately. When this function returns 1, the main loop exits. If the user wishes to permanently stay in the main loop (for example, extended real time acquisition and transfer), commenting out OFFLINE_TRANSFER will cause this function to always return 0. Otherwise, this function returns 1 once *sample_counter* exceeds the number of samples corresponding to NUMBER_OF_SECONDS_TO_ACQUIRE, indicating that this number of samples has been acquired (0.5 seconds at 20 kHz, or 10000 samples).

Users may want to change this function to check some other condition to exit the main acquisition loop.

`void write_data_to_memory(void)`

This function writes NUM_SAMPLED_CHANNEL's most recent CONVERT command results (starting at FIRST_SAMPLED_CHANNEL) from the command_sequence_MISO array into the *sample_memory* array, incrementing the *sample_counter* variable. For rhd2164_acquisition, this function first extracts two 16-bit samples from each 32-bit MISO result before writing the samples to *sample_memory*. The +2 offset used to index this MISO array is the result of the two-command pipeline delay explained in the Intan chip datasheet: each MOSI command will see its MISO result two commands later, and so this function checks two results further down the pipeline than the original index. Once the main data acquisition loop is escaped, the data in *sample_memory* is transmitted at once via USART. This function only executes if OFFLINE_TRANSFER is defined.

Users may want to change this function for a variety of reasons:

If data from different channels is desired to be saved (some configuration other than NUM_SAMPLED_CHANNELS in order starting from FIRST_SAMPLED_CHANNEL), then the for loop can be replaced with a more specific channel order.

If the auxiliary command slots are used for some more advanced purposes that require reading their results, they can be read here (as demonstrated in the commented-out code in the function). Due to the two-command pipeline, and the fact that these are the last three (for RHS, four) commands in a command sequence, the result of AUX SLOT 1 (and for RHS, AUX SLOT 2) will appear in the current command sequence, whereas AUX SLOT 2 and AUX SLOT 3 (or for RHS, AUX SLOT 3 and AUX SLOT 4) will appear in the next command sequence.

`void transmit_data_realtime(void)`

This function transmits data from the channels specified by NUM_SAMPLED_CHANNELS and FIRST_SAMPLED_CHANNEL, in real time (once per sample period) via USART. This function only executes if OFFLINE_TRANSFER is not defined, indicating that real time acquisition is desired. To achieve greater efficiency, this function begins a non-blocking USART DMA transfer which must be managed and monitored using interrupts – a simpler, blocking, standard USART transfer could also be used, but this will take significantly longer to complete and run the risk of triggering an *InterruptClip* error.

Users may want to change this function if data from different channels is desired to be transmitted (some configuration other than NUM_SAMPLED_CHANNELS in order starting from FIRST_SAMPLED_CHANNEL). In this case, the for loop can be replaced with a more specific channel order. Note that this function executes once per sample period, so if it takes too long, the next sample period will trigger before finishing. It is critical to avoid this important error condition, referred to as *InterruptClip*, so care must be taken to ensure this function does not take very long to complete. For example, make sure data is sent quickly at a high Baud rate. Users

may also want to change this function if they intend to do something else with acquired data other than transmit via USART.

`void transmit_data_offline(void)`

This function transmits acquired data from the channels specified by NUM_SAMPLED_CHANNELS and FIRST_SAMPLED_CHANNEL, once the entire acquisition period has finished and the main loop escaped, via USART. Implementations using both HAL and LL drivers are included in this function, generally accomplishing the same behavior. This function is only ever reached if OFFLINE_TRANSFER is defined.

Users may want to change this function if they have made some change to the way that data is saved, or intend to do something else with acquired data other than transmit via USART.

`void configure_registers(void)`

This function sets reasonable values for the registers in the global *RHDConfigParameters* or *RHSConfigParameters* struct and writes them via SPI. These values are determined programmatically through the functions *write_initial_reg_values* and *set_default_rhd_settings* or *set_default_rhs_settings*, in the *rhdinterface.c* or *rhsinterface.c* and *rhdregisters.c* or *rhsregisters.c* files, before being written via SPI.

Users who want to customize the values of specific registers before acquisition will want to change this function by altering *parameters* after *write_initial_reg_values* is called, and then writing a command specifically for each changed registers. A commented-out example is included in this function, demonstrating how register 2 can be set to allow for an impedance measurement to occur. (Register 3 should be changed sample-to-sample via an aux command list.)

`void create_manual_example_sequence(StimSequence* const sequence)` RHS only

This function creates a stim sequence manually (as opposed to creating an auto-generated waveform sequence with common shapes like biphasic or triphasic), allowing for precise user-control of all current magnitudes, polarities, and durations with more options than the auto-generated waveforms sequences. The default contents of this function demonstrate a four-segment sequence that loops between the final two segments infinitely, and should be modified by the user to implement any stimulation sequence that complies with the StimSequence specifications described above.

`void configure_stim_sequences(void)` RHS only

This function configures any user-defined stim sequences (either manually created with the above function, or with an auto-generated biphasic or triphasic waveform) and initializes the stimulation sequencer in order to begin executing these sequences once they are triggered. The default contents of this function demonstrate an example single-pulse biphasic waveform on channel 8 and an example 5-pulse triphasic waveform train on channel 9, and should be modified by the user to implement whichever stimulation sequence(s) are desired.

`void handle_compliance_result(uint16_t compliance_data)` RHS only

This function handles behavior when compliance data has been detected in the incoming MISO data. By default, the Compliance_Monitor pin is written low if all no compliance monitor activation has occurred, and high if at least one channel has had its compliance monitor activated, but any other custom behavior (for example, halting the program) if a compliance monitor has been activated should be implemented here.

`void configure_convert_commands()`

This function saves the CONVERT_COMMANDS_PER_SEQUENCE (for RHD 32, for RHS 16) CONVERT commands into the *command_sequence_MOSI* array, which is used every sample period to send all 32/16 commands in a single sequence. This implementation should call *create_convert_sequence* to populate each of these 32/16 commands as a (for RHD 16-bit, for RHS 32-bit) size SPI word that the Intan chip will recognize. Passing NULL as the second parameter will automatically order these CONVERT commands from 0-31/15, otherwise an array of `uint8_t` numbers can be passed to specify a specific order for these commands to occur.

If CONVERT_COMMANDS_PER_SEQUENCE has been reduced for performance reasons, this array will specify which channels are sampled at all.

Users who want to alter the order of CONVERT commands or specifically leave out certain channels from conversions will want to change this function by altering the argument to *create_convert_sequence*. A commented-out example is included in this function, demonstrating how to create and pass a *channel_numbers* array so that the sequence populating *command_sequence_MOSI* is ordered from 31/15-0 instead of 0-31/15.

```
void configure_aux_commands(void)
```

This function sets up the AUX_COMMANDS_PER_SEQUENCE (for RHD default 3, for RHS default 4) auxiliary command lists that are loaded into the end of the *command_sequence_MOSI* array, which is used every sample period to send three auxiliary commands at the end of a single sequence. This implementation should call some variation of a *create_command_list* function for each of the auxiliary command slots. A global RHDConfigParameters/RHSConfigParameters struct is used to construct any auxiliary command lists that rely on the configuration parameters.

Users who want to alter the number of auxiliary commands, or the contents of each command list, will want to change this function by changing which *create_command_list* functions are called for each command slot. By default, slot 1 is a register configuration command list, and remaining slots are dummy reads of ROM registers. Since impedance check command lists have variable lengths (all other command lists are created to be AUX_COMMAND_LIST_LENGTH commands long, by default 128), the process for creating an impedance check command list also requires setting the variable *zcheck_DAC_command_slot_position*, and is demonstrated in the commented-out section of this function.

For RHS, note that the aux command lists created here are only used if AUTO_STIM_CMD_MODE is not defined – otherwise, these command slots are reserved for execution of WRITE commands necessary for real time stimulation.

```
void transmit_dma_to_usart(const uint16_t *tx_data, uint16_t num_bytes)
```

This function uses DMA to transmit a certain amount of data from a memory pointer directly to USART. The memory location and size of the data are passed as input arguments. It is non-blocking, so it only begins the DMA transfer, and its progress must be monitored through the use of interrupts and the state of the 'uart_ready' variable.

Users who want to change how data is transmitted to USART may want to alter this function and related USART/DMA interrupt functions, or if they wish to do something else with data instead of transmitting to USART, they can remove this function entirely.

Performance Considerations

This STM32 example code is designed to run comfortably with either HAL or LL drivers, achieving a sample rate of 20 kS/s – for RHD, with 32 channels + 3 auxiliary commands and for RHS, with 16 channels + 4 auxiliary commands per sample period, with most of the time during acquisition spent processing in the main loop: most CPU time is free for other processing tasks. However, some applications might need further performance improvements, for example if 30 kS/s is desired, or if multiple Intan chips are controlled with a single MCU. In these cases, there are some steps that can be taken to optimize performance for specific applications.

HAL vs LL

HAL (Hardware Abstraction Layer) tends to have more simple function calls and is more uniform across all STM32 chip series, sacrificing efficiency for simplicity. LL (Low Layer) allows for more efficient completion of given tasks, but requires a deeper understanding of the individual STM32 registers. We recommend users start with HAL to gain a general understanding of how the example program works, and then if more advanced understanding is required switch to LL to see how the general behavior achieved by HAL can be implemented closer to the register level. LL functions tend to also be executed much faster than HAL functions, so if the user reaches a performance bottleneck simply switching from HAL to LL may speed up execution dramatically.

The performance difference between HAL and LL implementations varies depending on project, and their differences can be summarized with approximations of free clock cycles (what percentage of time the processor is free for other tasks during real time acquisition and USART transmission), which were calculated by measuring the total percentage that Main_Monitor_Pin is high. This is not a perfect representation of clock cycles, but a rough approximation due to Main_Monitor_Pin explicitly being written low at the beginning of several functions in the program. This method is likely to over-estimate the number of free clock cycles using HAL, as there are some HAL interrupt functions that are not easily editable by users and may consume additional clock cycles, despite Main_Monitor_Pin remaining high. Our measurements of these free clock cycles are displayed below:

Project	STM32 MCU	HAL/LL	Approx. Free Clock Cycles
rhd_acquisition	STM32U5	HAL	91%
rhd_acquisition	STM32U5	LL	92%
rhd_acquisition	STM32H7	HAL	85%
rhd_acquisition	STM32H7	LL	92%
rhd2164_acquisition	STM32U5	HAL	75%
rhd2164_acquisition	STM32U5	LL	79%
rhd2164_acquisition	STM32H7	HAL	78%
rhd2164_acquisition	STM32H7	LL	85%

Since RHS processing efficiency is less deterministic (results vary depending on the number, complexity, and trigger frequency of stimulation sequences), they are not included in this table, but can easily be measured by the user using the method described above while running and stimulating with specific sequences.

Changing Sample Rate

The sample rate is governed by the INTERRUPT_TIM (default TIM3) peripheral, specifically the clock input it receives and the counter period, both of which are set in the .ioc file. The different projects in this firmware framework use a variety of system clock speeds, so in turn they set different counter periods, but in all cases the counter periods are selected to result in the timer issuing an interrupt at 20 kHz. **If a different sample rate is desired, it must be changed through the .ioc file by altering the INTERRUPT_TIM counter period and/or clock source**.

Sample rates above 20 kS/s are likely to cause each sample period interrupt to clip into the next (*SampleClip* error), halting program execution and illuminating the red LED. Each action of the sample processing routine will contribute to this, but the most likely culprits are the SPI sequence taking too long to complete, any processing tasks that occur within each sample period (for example, extraction of two 16-bit samples from each 32-bit MISO result from the DDR SPI in rhd2164_acquisition), or if OFFLINE_TRANSFER is commented out, the USART transmission taking too long to complete. The SPI sequence's speed is

physically limited by the minimum time requirements outlined in the RHD datasheet, and the example program is already quite close to these minimum requirements. The details of streamlining the SPI transfer are discussed below.

However, the simplest way to minimize the amount of data the command sequence must transmit, receive, and process, is to reduce the number of commands. Since the total length of the command sequence is CONVERT_COMMANDS_PER_SEQUENCE + AUX_COMMANDS_PER_SEQUENCE, reducing either of these will reduce the amount of time required for the sequence to complete and the amount of received data. If fewer than 32 (or 16 for RHS) channels are required, CONVERT_COMMANDS_PER_SEQUENCE can be reduced to only include the channels that are sampled. (In the default example programs, only four of the 32/16 sampled channels actually have their data saved, or 4 x 2 = 8 sampled channels due to DDR for rhd2164_acquisition.) Similarly, of the three (for RHS, four) auxiliary command slots that are included in the example program, two are dummy command lists that only read ROM registers and act as placeholders for any other auxiliary command lists, so many users will find AUX_COMMANDS_PER_SEQUENCE can be reduced. The first command slot continually reprograms the RHD registers, which allows for quick recovery from any unexpected data corruption during acquisition, and is a good idea for longer acquisition sessions but is not strictly necessary if the speed-up from removing a single SPI command word is critical.

For RHS, if AUTO_STIM_CMD_MODE is defined, all present auxiliary command slots are reserved for the requisite WRITE commands to allow for stimulation, and the number of these commands could be altered, but care must be taken; the file *stimscheduler.c* has a function *load_aux_commands()* which assumes that 4 aux command slots are present, so this must be re-written and the user should ensure that the *rhsinterface.c* file, which includes the function *locate_compliance_result()*, correctly handles any change to how the compliance aux slot is set and how it is correctly identified after the two-command pipeline delay.

Streamlining SPI Communication

In addition to reducing the number of SPI words per sample period, some steps can be taken to make each SPI word faster. Currently, the SPI achieves between a baud rate (the speed at which SCLK switches throughout a 16-bit word) between 20 and 24 Mbit/s by dividing the input clock signal by a prescaler – the specific baud rate varies between projects due to different clock requirements for different projects. The maximum SCLK baud rate the Intan chip accepts is 25 MHz, as seen on the Intan chip datasheets. So, maximum SPI efficiency includes setting the baud rate to as close to, without surpassing, 25 Mbit/s, and this may be achievable for certain configurations by actually reducing the clock speed, and increasing the prescaler value. Obviously, throttling the clock speed will reduce efficiency of all other processing tasks, and due to the example program by default using DMA for SPI transfers (offloading much of the processing from the CPU) there is not necessarily a large benefit to boosting the baud rate much, but the example program does not require very intensive processing generally so this may be worth trying to get the SPI transfers done as quickly as possible within each sample period.

Another important note when discussing changing SPI timing is ensuring the minimum time requirements from the Intan datasheets are met. If SPI speed is changed, the user must make sure these parameters (namely maximum SCLK frequency of 25 MHz and $\overline{CS}$ stays high for 154 ns between words) stay valid. The example program already uses MIDI (Master Inter-Data Idleness), and other SPI control fields like MSSI (Master SS Idleness) can be used to ensure the timing requirements are not violated.

The example code already uses DMA to initiate SPI transfers, monitoring the progress with interrupt flags, which does not physically speed up the SPI transaction, but frees up the processor for other tasks while SPI transactions are underway.

Speeding up USART

When OFFLINE_TRANSFER is defined, the example code transmits data offline, waiting until a certain time period's worth of data (default 0.5) is acquired before transmitting any data via USART, but this behavior is replaced by transmitting via USART in real time instead if OFFLINE_TRANSFER's definition is commented out. For offline data transfer speed is unimportant, but if data needs to be transferred alongside data acquisition, the USART transmission must be as fast as possible to avoid impacting data sampling. While the details of USART communication are beyond the scope of this document, the higher the baud rate the more quickly data can be sent. The example code uses USART baud rates between 6 Mbit/s and 12.5 Mbit/s which is more than sufficient for streaming four channels' data in real time (eight for rhd2164_acquisition), but if additional throughput is needed, the user should consider parallelizing data transfer with additional USARTs or using other peripherals.

The example code already uses DMA to initiate USART transfers, monitoring the progress with interrupt flags, which does not physically speed up the USART transaction, but frees up the processor for other tasks while USART transactions are underway.

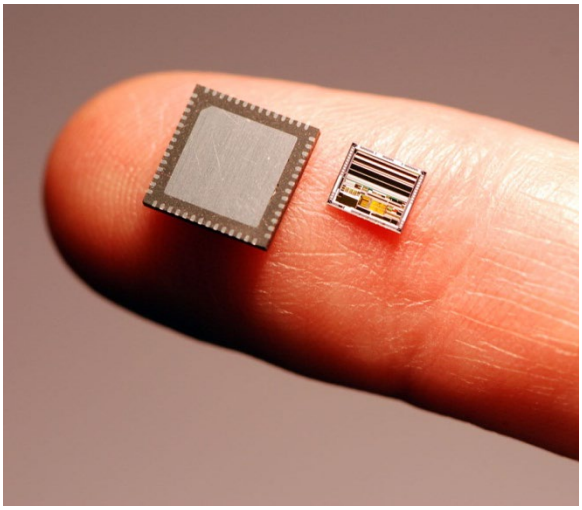
Optimizing Memory Usage

Depending on the limitations of the available hardware and any customizations to the software, it's possible that the program will run out of memory, either causing a build-time or a run-time error (OutOfMemoryError). This is more likely to happen with the H7 chip, with its 564 KB RAM vs. the U5 chip's 2514 KB RAM. The more data that is required to be stored in the program's memory, the more of this RAM is occupied. The most obvious ways to reduce memory requirements are to slow down the sample rate, and lower the number of sampled channels. For offline acquisition, reducing NUMBER_OF_SECONDS_TO_ACQUIRE will reduce the amount of memory dedicated to storing data. In general, offline acquisition demands much more memory than real time acquisition, as real time acquisition does not hold multiple timesteps worth of data at a single time. For stim, the StimSequence structures used to sequence stimulation pulses are allocated significant amounts of memory, much of which may go unused depending on desired stimulation behavior, so reducing the macros defined at the beginning of "stimscheduler.h" can free up a good amount of memory.

Contact Information

This datasheet is meant to acquaint engineers and scientists with the Intan STM32 interface code developed at Intan Technologies. We value feedback from potential end users. We can discuss your specific needs and suggest a solution for your applications.

For more information, contact Intan Technologies at:



intan TECHNOLOGIES, LLC

www.intantech.com
support@intantech.com