SonicLib Programmer's Guide

for SonicLib v4

InvenSense, a TDK Group Company
1745 Technology Drive, San Jose, CA 95110 U.S.A
+1(408) 988–7339
invensense.tdk.com

Document Number: AN-000175
Revision: 3.0
Release Date: 07/15/2024

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

SonicLib is a set of API functions and sensor driver routines designed to easily control InvenSense ultrasonic sensors from an embedded C language application. It allows an application developer to obtain ultrasonic range data from one or more devices, without needing to develop special low-level code to interact with the sensors directly.

The sensors measure distance (range) by emitting an ultrasonic pulse and measuring the time-of-flight (ToF) for that pulse to travel through the air, either reflected back to the transmitting sensor or received by a second sensor.

The SonicLib API functions provide a consistent interface for an application to use the sensors in various situations. This is especially important, because all InvenSense ultrasonic sensors are completely software-defined. Except for a few low-level programming interfaces, the SPI or I^2C "registers" are simply memory locations determined by the specific sensor firmware being used. They are subject to change between firmware revisions and can vary significantly between firmware images with different feature sets, even for the same device hardware. Operation of the ultrasonic sensors and presenting the data is more complicated than in most other types of devices, requiring intimate interaction with the device.

The SonicLib API allows your application to use different sensor firmware images or SonicLib updates without requiring code changes.

This guide explains how to use SonicLib to create your embedded sensing application. Key API functions and sequences are highlighted, with an emphasis on typical use cases. Some options and parameters, and many additional API functions, are not discussed in detail here. For more information on other SonicLib features, refer to the included HTML documentation or the comments in the **soniclib.h** header file, which provides a complete description of all interfaces.

1.1 SonicLib Versions

This document describes SonicLib version 4.x. The version number may be found in **soniclib.h** or by calling the *ch_get_version()* function.

1.2 Types of Applications

InvenSense ultrasonic sensors can be used in a great variety of system designs and operating conditions. SonicLib is designed to support this wide range of applications with a consistent set of standard interfaces. No single application will use the full range of features.

Along with a full set of common interfaces to control ultrasonic sensors, SonicLib includes built-in support for rangefinding applications that need to determine the distance to a target object. Other kinds of applications, such as human presence detection, may require external algorithm packages available from InvenSense. These special algorithms still use SonicLib functions for control of the sensor(s) and process data obtained using the standard interfaces described in this document. Contact InvenSense for more information on special algorithms.

1.3 Supported Sensor Types

SonicLib v4 supports all models of TDK InvenSense ultrasonic sensors. These include the current ICU family (e.g., ICU-10201, ICU20201, ICU-30201) as well as the earlier CH101 and CH201 sensors originally developed by Chirp Microsystems.

Because the sensors themselves, and the internal sensor firmware that runs on them, vary in their features, not all SonicLib functions are available in every sensor and firmware combination. The specific sensor firmware that is programmed into the sensor will determine the exact set of interfaces that are available. This is primarily a factor when using CH101 and CH201 sensors, which are much more resource constrained and require many "either-or" tradeoffs in the feature set, resulting in multiple variants of sensor firmware.

If not otherwise indicated, the feature and interface descriptions in this document apply equally to all sensor models.

ICU Sensors

The current ICU-10201, ICU-20201 and ICU-30201 sensors are the second generation of Chirp/InvenSense ultrasonic sensors. They use the "Shasta" sensor architecture.

ICU sensors have 6kB of programmable program space. The sensor firmware is not stored in the sensor – it must be loaded each time the sensor is powered on. (Firmware programming is handled automatically in SonicLib.) A high-speed SPI bus connection provides host communication and data transfers.

The ICU sensors provide a rich set of controls to customize the ultrasound measurement to suit your application and the operating environment.

Although out of the scope of this document, ICU sensors and SonicLib also support development of custom on-chip algorithms for special processing needs. Contact InvenSense for more information.

CH101 and CH201 Sensors ("CHx01")

The CH101 and CH201 sensors are the first-generation sensors designed by Chirp Microsystems. They use the "Whitney" sensor architecture.

In this document, CH101 and CH201 sensors may collectively be referred to as "**CHx01**" when the comment applies to both sensor types.

Compared to the later ICU sensors, CH101 and CH201 offer comparable acoustic performance at short to medium distances, but they are much less flexible in operation. The program space in CHx01 sensors is only 2 kB, compared to 6 kB in the ICU series. This size restriction forces the CHx01 to use fixed configurations and multiple variants of firmware images, instead of the flexible runtime control available in the ICU models. Like ICU sensors, CHx01 devices require programming after every power on. CHx01 sensors use a separate "PROG" line to select the device during initialization and programming.

CH101 and CH201 sensors use an I2C bus interface for communications, compared to the SPI interface in the ICU sensors. As a result, data transfer times are much longer, which can affect the turnaround time between measurements. Some memory operations involving sample data are also limited, due to 8-bit addressing.

1.4 Sensor Firmware Options

All InvenSense ultrasonic sensors are fully programmable, and the overall operation and feature set can vary between different sensor firmware types.

SonicLib includes the most common, general purpose firmware types as part of the distribution, as listed in Table 1.

Sensor Firmware Type	Sensor Model(s)	Description
icu_gpt	ICU-10201, ICU-20201, ICU-30201	General purpose transceiver firmware, with rangefinding support
icu_init + variants	ICU-10201, ICU-20201, ICU-30201	Special internal-use firmware, loaded temporarily during initialization and/or optimization
ch101_gpr + variants (short range, narrow horn, watchdog-enabled)	CH101	General purpose rangefinding
ch101_gprmt	CH101	General purpose rangefinding with multiple thresholds, no Static Target Rejection (STR)
ch201_gprmt	CH201	General purpose rangefinding with multiple thresholds, no Static Target Rejection (STR)

Table 1 - Included Sensor Firmware Types

If your application is not based on range (distance) data, it may require a different sensor firmware type that implements a different algorithm to process the ultrasound measurements. For example, to support human presence detection **icu_presence** firmware may be used on ICU sensors, or **ch201_presence** firmware on CH201 sensors. These and other specialized firmware types are released and installed separately as SonicLib "Plug-Ins." See Sensor Firmware Plug-ins for more information.

1.5 Source Code Distribution

SonicLib is distributed in source code form to make it easier to build and use on different hardware platforms. The code is designed to be highly portable to different toolchains, etc.

You should not modify the source files contained in the SonicLib distribution. Only the public API is supported by InvenSense, and no support is offered for source modifications. See SonicLib Distribution Files.

2 BASIC CONCEPTS

2.1 Measuring Distance Using Time of Flight

Most of us have had the experience of seeing a lightning bolt and then using the delay between the flash and the arrival of the thunder to estimate how far away the lightning strike was. For many, the initial flash triggers an immediate response of counting the seconds until the thunder is heard. If we happen to know a rough estimate of the speed of sound (e.g., 3 seconds per kilometer, or 5 seconds per mile), we can easily convert the observed time into a useful approximate distance.

The InvenSense ultrasonic sensors use this same approach, measuring the time it takes for sound to arrive after a known event, to determine distances at much closer ranges and with high accuracy. This elapsed time is known as the "Time of Flight" (ToF). An ultrasonic ToF sensor is a transceiver, meaning that it both transmits and receives ultrasound signals. Unlike most types of sensors which passively measure their surrounding conditions, the ultrasonic sensor actively injects a signal into its environment.

To perform a basic distance measurement, the sensor will emit a very brief pulse of ultrasound. It then immediately enters a "listening" state, in which it samples and records the received ultrasound, including any echo of the pulse that has been reflected off an object in the sensor's vicinity. The sensor processes the received data, seeking to identify an ultrasound pulse. If a pulse is identified, the sensor will analyze the timing and then report the ToF of the received pulse. The actual distance travelled by the ultrasound can then be calculated from the ToF based on the speed of sound.

2.2 Anatomy of a Measurement

A single measurement by the ultrasonic sensor is actually a sequence of several distinct phases, as shown in Figure 1.

Figure 1 - Phases in a Measurement

1. In the **Transmit** (Tx) phase, a burst of ultrasound is transmitted by the sensor, through a "horn" or other physical acoustic housing.
2. Next, the sensor enters a **Listening** phase, in which it repeatedly samples the level of received ultrasound at a fixed rate and stores the results. The length of this listening phase determines how distant an object may be detected (i.e., how long the reflected ultrasound takes to arrive). Longer listening times (farther detection distances) will generate more samples during each measurement.
3. After the specified number of samples has been stored, the **Signal Processing** phase begins (shown as "DSP" in Figure 1). The sensor will analyze the received data based on the specific sensing algorithm in the firmware being used. In most cases, this includes a calculating "range" value indicating the distance to the detected object.
4. The sensor will **interrupt** the host processor at the completion of the measurement. In some cases, the sensor may be configured to interrupt only if an object is detected.

5. The host application typically responds to the interrupt by **reading out** the required measurement result, using SonicLib API functions. This may be a simple range (distance) value for a reported "target" object, or the full "raw" data set of all samples in the measurement, or some specific combination of values. However, there is no requirement to read any data from the sensor before the next measurement.

The Listening phase is always a fixed length, based on the maximum range setting, which determines the number of samples (see 2.5, below). The full set of data is collected before the Signal Processing phase begins.

2.3 Time = Distance = Samples

The most important concept to grasp when programming the ICU and CHx01 sensors is the relationship and *interchangeability* of time, distance, and sensor samples. Because the speed of sound is constant (at least enough for this discussion), and the sensor takes samples at a constant rate, each sample within a measurement represents both a point in time as well as a specific distance from the sensor.

Time vs. Distance

In calculations, SonicLib uses 343 meters per second as the speed of sound, which is the standard value for 20°C air.

Samples vs. Distance

During a measurement, the sensor repeatedly samples the received ultrasound, driven by its own internal timing. The exact sampling rate is tied to the resonant frequency of the individual sensor and will therefore vary somewhat from one device to another. The time between samples, and therefore the physical distance the sound travels, will be different. So, the operating frequency of the device must be included in all conversions between internal sample index-based values and real-world physical distance.

Conversion Routines

SonicLib provides two conversion routines to easily translate physical distance into sample counts, and vice versa, based on the measured calibration values and operating frequency for the individual device. The *ch_mm_to_samples()* function takes a physical distance in millimeters and returns the corresponding number of samples. Conversely, the *ch_samples_to_mm()* function takes a sample count and returns the corresponding number of millimeters. Both conversions use only whole millimeters and samples for input and output parameters, so some rounding error should be expected.

2.4 Sensor Operating Modes

During operation, a sensor is always in one of four operating modes that reflect the sensor's state.

CH_MODE_IDLE

In *CH_MODE_IDLE* ("Idle Mode"), the sensor does not transmit ultrasound or perform any measurements. The sensor configuration is ignored.

This is the initial mode for sensors when they are first initialized. The sensor configuration should be applied before the sensor mode is changed.

The *ch_set_mode()* function is used to change the sensor to a new mode. You should be careful that the sensor has been fully configured, including timing, threshold, and measurement parameters, before initially changing the mode from *CH_MODE_IDLE*. When this mode exits, the sensor will immediately begin sensing (or be armed for triggering).

CH_MODE_FREERUN

In *CH_MODE_FREERUN* ("Free-running Mode"), the sensor performs self-timed measurements using its own internal clock. The sensor will periodically wake up, make a measurement, and report the completion (via interrupt). The host may then (optionally) read data from the sensor. The sensor will automatically wakeup at the end of the programmed measurement interval and repeat the pattern.

The sensor's trigger input is not used when the sensor is operated in this mode. This mode is appropriate for independent sensors – use the triggered modes, below, for synchronizing multiple sensors.

The *ch_set_freerun_interval()* function is used to specify the measurement interval.

CH_MODE_TRIGGERED_TX_RX

In *CH_MODE_TRIGGERED_TX_RX* ("Triggered Transmit/Receive Mode"), the sensor initiates measurements based on the trigger input pin. When the trigger signal is asserted, the sensor will begin a measurement, then report completion via interrupt. The measurement interval is determined by how often the sensor is triggered.

Sensors in *CH_MODE_TRIGGERED_TX_RX* mode may also use software triggering (see *ch_set_trigger_type()*), which initiates a measurement by sending a data packet to the sensor over the SPI or I2C bus. (For CHx01, soft triggering is only available in limited firmware types.)

ICU sensors may use either the INT1 or INT2 pin as the trigger pin. CHx01 sensors have only a single pin, which is always used for both triggering and interrupts.

CH_MODE_TRIGGERED_RX_ONLY

In *CH_MODE_TRIGGERED_RX_ONLY* ("Triggered Receive-Only Mode"), the sensor initiates measurements based on the trigger input pin, but no ultrasound is transmitted. The ultrasound must come from another sensor (in *CH_MODE_TRIGGERED_TX_RX* mode) triggered at the same time. When the trigger signal is asserted, the *CH_MODE_TRIGGERED_RX_ONLY* sensor will begin receiving for the required time, then report completion via interrupt.

Software triggering is not recommended for multi-sensor configurations, due to the time skew between triggers sent over the communications bus. So, it is not normally used with *CH_MODE_TRIGGERED_RX_ONLY*.

ICU sensors may use either the INT1 or INT2 pin as the trigger pin.

2.5 Maximum Range Setting

The ultrasonic sensor has a configurable full-scale range (FSR), meaning that the application may set the maximum distance at which the sensor will detect an object. Any value up to the rated maximum range for the device may be specified. The range value determines how many samples will be taken during each measurement (i.e. how long the listening period will be).

The maximum range may be set using the *ch_set_max_range()* function.

The *ch_set_max_range()* function has the following definition:

```
uint8_t ch_set_max_range (ch_dev_t *dev_ptr, uint16_t max_range)
```

The *max_range* value is the one-way distance to a detected object, in millimeters.

Choosing the Maximum Range Value

When you are using ultrasonic sensors for the first time, it may seem natural to always set the maximum range to the farthest distance supported by the device (its specified maximum range). While this is perfectly valid and should work correctly, it may not be the best choice depending on your application needs.

In practice, the maximum range setting is controlling the amount of time that the sensor spends in the listening (receiving) period during a measurement cycle. The extra time is needed for the ultrasound pulse to travel farther through the air. Note that the pulse must travel round-trip to/from an object, so is twice the one-way distance (and therefore requires twice the time).

So, the maximum range setting directly affects the total time required to complete a measurement. Longer full-scale range values will require more time for a measurement to complete. Each additional meter of measurement range requires about 6 milliseconds more time-of-flight (round-trip).

Longer maximum range settings also cause the sensor to capture more samples internally during the extended listening period, generating more sample data. The sensor must then analyze those additional samples at the end of the measurement, potentially increasing the sensor's internal processing time.

The additional amount of sample data generated by long range settings can also be an important consideration if your application needs to transfer the raw I/Q measurement data from the sensor.

For CHx01 sensors, which use I2C for data transfer, the time to read the full set of data (900 bytes for a CH101 or 1800 bytes for a CH201) can be quite significant relative to the overall measurement timing. (See Table 2 – Measurement Time Requirements.) Shorter maximum range settings use fewer samples and therefore reduce the amount of active I/Q data in each measurement and the I/O time required to transfer it.

In comparison, reading the entire I/Q data from ICU-x0201 only takes about 1.5ms, which is typically a small fraction of the time spent receiving the ultrasonic signal.

Because of all these factors, you should choose a maximum range setting that covers the conditions you expect your application to encounter but is not longer than necessary.

2.6 Target Detection Thresholds

Ultrasound is constantly present all around us, so the sensor must be able to distinguish background noise from the signal reflected from an object (the "target"). In most applications, particularly "rangefinding" type applications that report the measured distance to an object, it is important to adjust the sensitivity of the sensor to detect objects at different distances. Farther objects will have a much weaker echo than closer objects.

When the sensor analyzes the results from a measurement, the amplitude of each sample within the measurement is compared with a corresponding threshold value to decide if the signal is strong enough to be a target object. If the measured amplitude for that sample is greater than the threshold, the sensor will process that point in the measurement as a target object, including range calculations. If no sample in the entire measurement is greater than the corresponding threshold value, the sensor will report "no target."

To avoid false positives at close distances but still allow detection of weak target signals at longer distances, the threshold values need to decrease for later (farther) samples in the measurement.

A set of programmable detection thresholds allows the application to configure different required minimum amplitudes for blocks of samples within the measurement. Most applications will need to tune these threshold values for best performance, based on the specific physical environment and target characteristics.

The *ch_set_thresholds()* function is used to set the detection thresholds, which are discussed in detail in Multiple Detection Thresholds.

2.7 Sensing Configurations and Range Calculations

A Chirp ultrasonic sensor may be used alone or in combination with one or more other sensors. This section describes some basic sensing configurations and how these relate to the range (distance) calculations.

Single Sensor (Pulse – Echo)

The most basic configuration is a single device. In this arrangement, the sensor will both transmit and receive ultrasound to perform the measurements. The device will listen for an echo of its own ultrasound signal, calculate the ToF for the received echo, then notify the host system that the measurement has completed. This is often simply called "Pulse-Echo" operation. See Figure 2.

When one sensor is used in Pulse-Echo mode, the most useful range type selection is usually *CH_RANGE_ECHO_ONE_WAY*. This calculates the one-way distance to the target object (based on one-half of the total time-of-flight).

In special cases, the full round-trip distance may be more useful (to reflect the full out-and-back path of the ultrasound), and *CH_RANGE_ECHO_ROUND_TRIP* may be used.

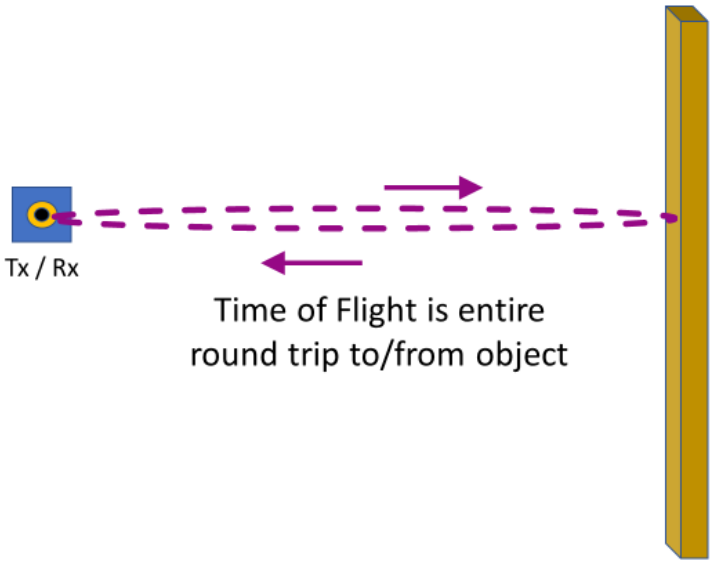

Figure 2 - Pulse-Echo Sensing using a Single Sensor

Multiple Sensors (Pitch – Catch)

In other applications, multiple sensors may be used together, in what is often called "Pitch-Catch" operation. When multiple sensors are used, they must be hardware-triggered for synchronization. They cannot use free-running mode.

In Pitch-Catch operation, one sensor generates an ultrasonic pulse and waits for an echo, as in the single-device configuration (*CH_MODE_TRIGGERED_TX_RX*). One or more other sensors are operated in "receive-only" mode (*CH_MODE_TRIGGERED_RX_ONLY*), so they do not generate ultrasonic pulses – they simply listen for the pulse to arrive from the first device.

All devices (the transmitting sensor and all receive-only sensors) are synchronized so that the receive-only nodes will start their sampling when the first sensor transmits. All devices then process the received signal, calculate the ToF, and report to the host system.

There are two basic approaches to using a pair of sensors together (one transmitting and another receiving). In one configuration, the two sensors are attached to two different objects, and the distance being measured is the direct distance between the two objects (i.e., between the two sensors). This mode of operation gives the best performance in terms of measurement accuracy and stability. See Figure 3.

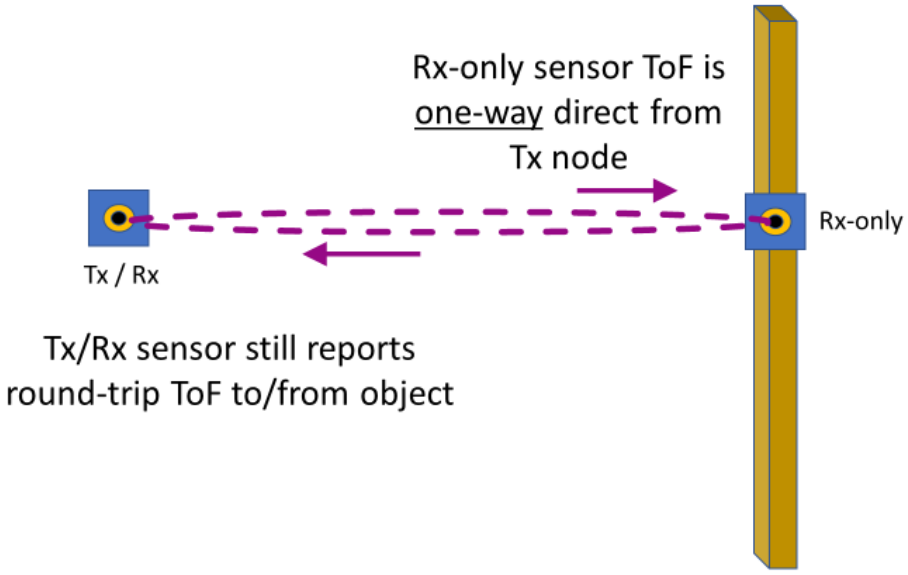

Figure 3 - Direct Pitch-Catch Sensing using Two Sensors

In this situation, the most important data values are the range measurements from the receive-only device. The ToF measured in this case is the one-way, direct path between the transmitting and receiving sensors, so the appropriate *range_type* for the receiving sensor to report is *CH_RANGE_DIRECT*.

Note that the transmitting sensor (*CH_MODE_TRIGGERED_TX_RX*) will still listen for its echo and report the results (round-trip ToF to the object). So, the transmitting sensor operates the same as when it is a single sensor operating independently.

Multiple Sensors (Reflected Pitch – Catch)

The second way two or more sensors may be used in Pitch-Catch operation is for the devices to be mounted to the same object, and the ultrasonic signal is reflected off another object. This is called "Reflected Pitch-Catch" operation. See Figure 4.

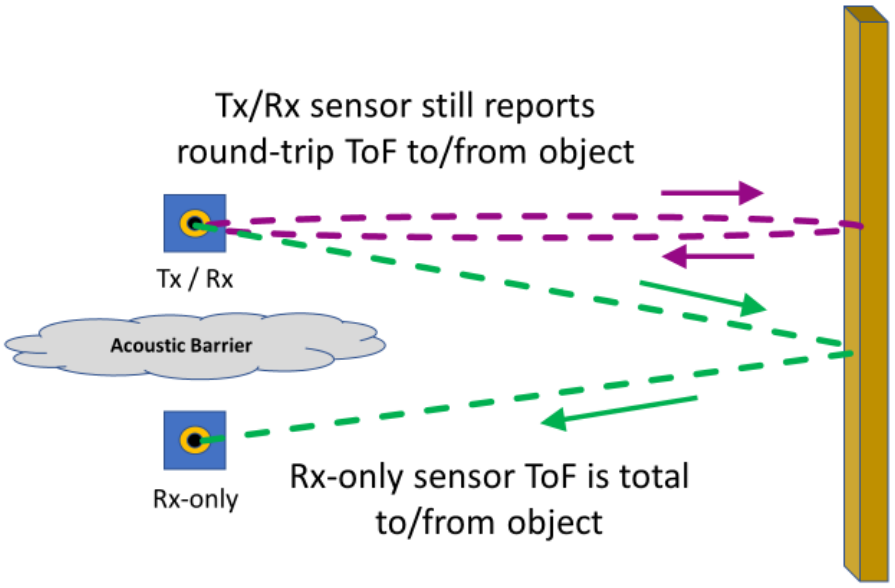

Figure 4 - Reflected Pitch-Catch Sensing using Two Sensors

In Reflected Pitch-Catch operation, the receive-only sensor will measure and report the total ToF for the path from the transmitting sensor, bouncing off the target object, and then back to the receiving sensor. So, a range type of *CH_RANGE_ECHO_ONE_WAY* (or *CH_RANGE_ECHO_ROUND_TRIP*) is appropriate for the receive-only node. Depending on the relative positions of the two sensors and the target object, this distance may differ significantly from a simple single-sensor echo path.

As shown, an acoustic barrier should be used to minimize the direct ultrasound pulse transmission between the sensors.

Operating Frequency & Pitch-Catch Operation

When one sensor is sending and another is receiving, the efficiency of the ultrasound transmission is affected by the operating frequencies of both devices. The most efficient transmission takes place when the sensors operate at the same frequency. The acoustic bandwidth of the sensor is also affected by the physical horn and surrounding surfaces. If the bandwidth is too narrow, there will be a greater penalty for mis-matched frequencies.

In general, two sensors of the same model will be able to operate in pitch-catch mode, because the natural operating frequencies will be reasonably close. (It is not possible to use different model sensors in pitch-catch mode.)

In ICU sensors (and when using certain firmware types for CHx01 sensors) the operation may be optimized by adjusting the frequencies of the devices slightly, to give a more exact match. See Adjusting the Operating Frequency.

2.8 Ringdown

Ringdown is a false signal caused by residual physical and electrical forces in the ultrasonic transducer (PMUT) after sending the short transmit pulse. The ringdown signal appears a high amplitude, consistent signal during the first samples in a measurement, as shown in Figure 5. At their peak, the ringdown amplitudes are generally higher than those of an actual echo. A similar but smaller ringdown artifact occurs even in receive-only sensors, when the PMUT is energized.

To illustrate how ringdown can interfere with the ultrasonic sensor, imagine you are in a bell tower located across from a large, flat building. If you clap your hands, you can distinctly hear the echoes from the building. However, if you ring a large bell, it is much more difficult to identify its echo, because the bell's vibration is still present as the echo arrives.

Compared to a bell, of course, the ultrasonic transducer's ringdown is extremely short, but it does last long enough to affect the first samples that are taken during each measurement (i.e., signals from very short distances).

Multiple techniques are used to minimize the effects of ringdown, which are each described in later sections:

- The sensor always performs some degree of Ringdown Filtering to remove the constant ringdown signal. In ICU sensors, the ringdown filter settings can be controlled by the application.
- ICU sensors are also able to perform Transmit Optimization, which actively dampens the ringdown using transmit controls.
- Lower receiver gain is used for early samples, to avoid saturation and help make real signals during the ringdown phase identifiable. ICU sensors allow Receive Gain Reduction to be specified by the application. CHx01 sensors use settings built into the sensor firmware.
- Sensors which are used in pitch-catch mode can enable Receive Pre-triggering on the receive-only sensors to allow them to fully settle before close-transmitted signals arrive, to improve performance at short distances.

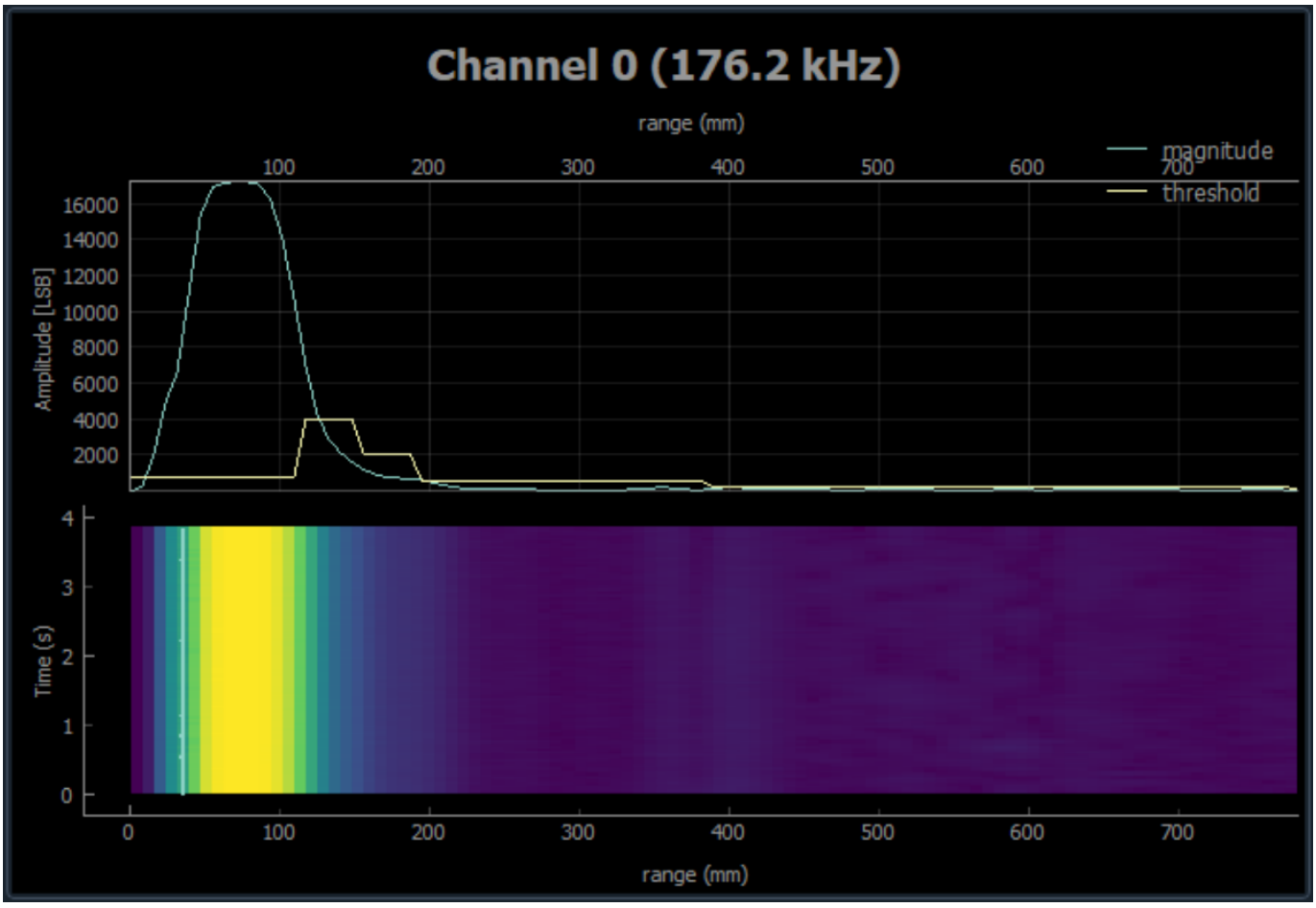

Figure 5 - Ringdown signal on an ICU-10201 (no ringdown filter applied)

3 SONICLIB ORGANIZATION

3.1 SonicLib Distribution Files

A SonicLib installation is organized to separate SonicLib and other distribution files from the board support package and the application itself. This permits flexible project organization and allows an application to run on different hardware platforms or use different InvenSense releases with minimal porting effort.

The standard SonicLib file structure is shown in Figure 6 - SonicLib File Organization. This internal directory structure should be maintained so that include paths and separately-distributed software modules work as expected. You may locate the SonicLib files relative to your application in any way you prefer.

You should never modify any files in the soniclib directory or its sub-directories.

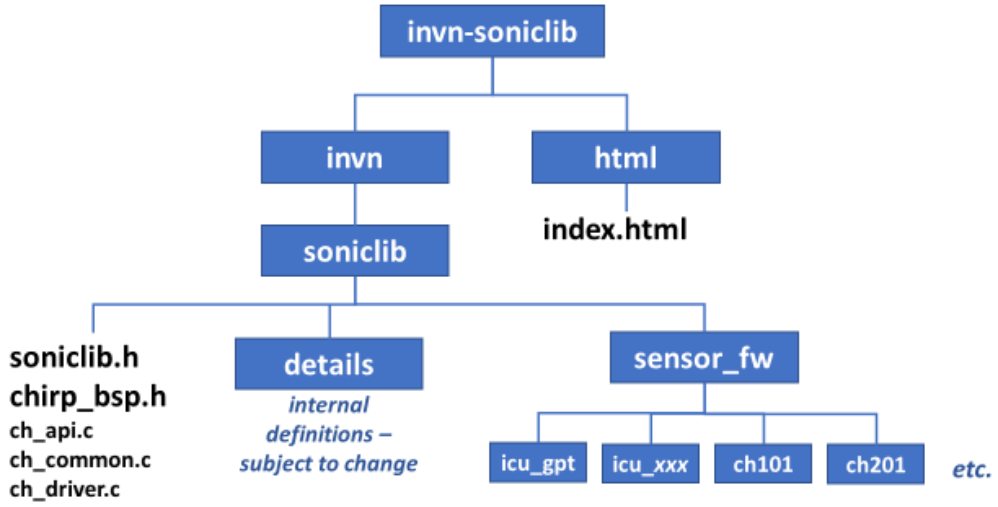

Figure 6 - SonicLib File Organization

index.html

SonicLib includes fully linked HTML documentation, generated from the interface descriptions in **soniclib.h** and other files by the Doxygen tool using the **Doxyfile** definition file. The **index.html** file, located in the top-level **invn-soniclib/html** directory, is the entry point for descriptions of the SonicLib and BSP interfaces. Open this file in any web browser to get started.

soniclib.h

The **soniclib.h** file is the key header file in SonicLib. It contains the complete definition for the SonicLib API including functions, data structures, and symbols. This file must be included by any source modules that will use the sensor interfaces.

chirp_bsp.h

The **chirp_bsp.h** file contains the definitions of the Board Support Package (BSP) interface. See Board Support Package Files.

ch_api.c

The **ch_api.c** file contains the public entry points for all SonicLib API functions. The public API functions all have names beginning with a "***ch_***" prefix. These functions and related structures and symbols are all defined in **soniclib.h**.

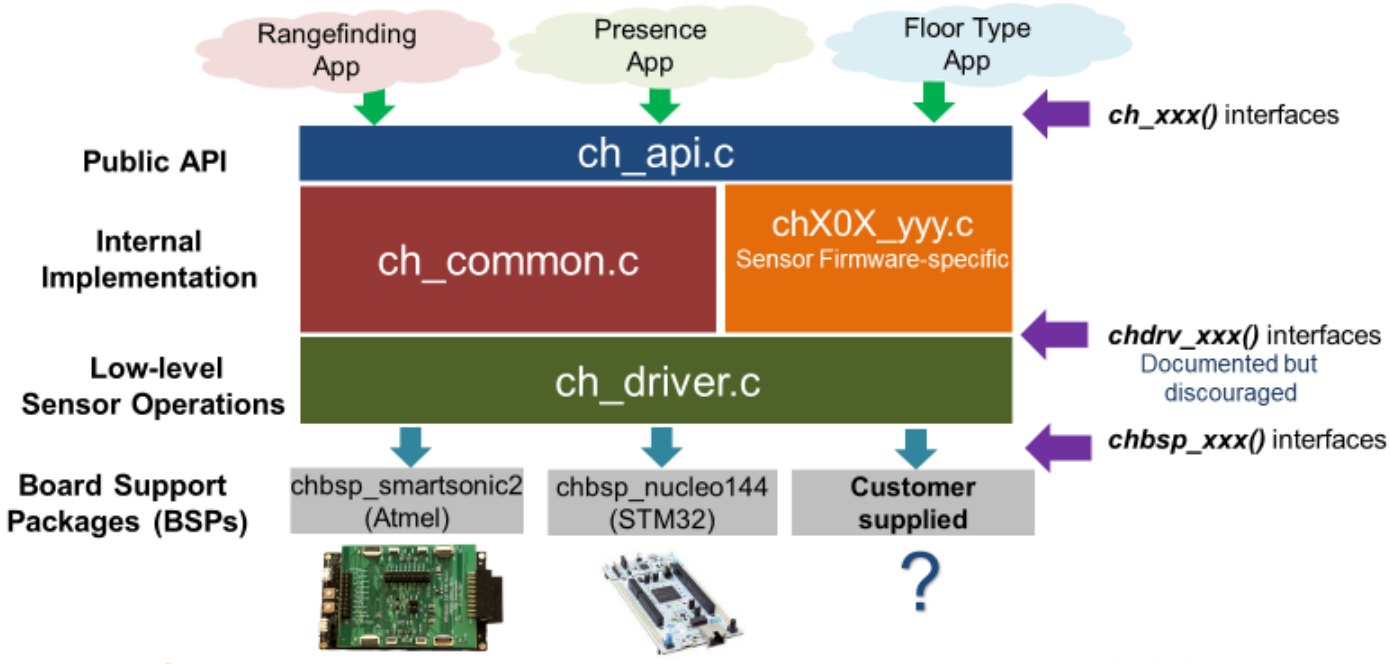

Figure 7. The interfaces provided by ch_api.c and other SonicLib files.

ch_common.c

The **ch_common.c** file contains the internal implementations of many SonicLib functions. These implementations may not be applicable to all sensor firmware and are subject to change or removal without notice. **You should never directly call any functions in ch_common.c** (names beginning "*ch_common_*").

ch_driver.c

The ch_driver.c file contains low-level functions that initialize and access the sensor. The driver functions have names beginning with a "***chdrv_***" prefix.

In general, you should not need to call any driver functions directly. The SonicLib API routines should be used to perform all device operations.

details

The **details** directory contains various files and directories that are internal to SonicLib. Because SonicLib is distributed in source code form, many implementation specifics are present in the definitions used to build it. By necessity, these are subject to change or elimination without notice in future revisions. No header files from the **details** directory sub-tree should ever be included directly from your application, and no symbols or definitions should be referenced.

sensor_fw

TDK InvenSense ultrasonic sensors are fully programmable and use separate sensor firmware images that are downloaded to the sensor and execute there. Different firmware packages include different internal algorithms to process ultrasound data and therefore present specialized sets of interfaces and features. The **sensor_fw** directory contains multiple sub-directories containing different sensor firmware files. For ICU sensors, each firmware type has a separate sub-directory. Due to the large number of firmware types, CHx01 sensor firmware is organized by sensor model, with multiple firmware types in a single directory.

Some sensor firmware types are included in the default SonicLib distribution (see Table 1 - Included Sensor Firmware Types), while others are available separately as SonicLib Plug-ins. See the Sensor Firmware Plug-ins section for more information. When a new sensor firmware package is added, it is stored in a new sub-directory under **sensor_fw** and typically includes a header file and two C source files.

ch_rangefinder.h

This file defines the API specific to range finding firmware, previously defined in soniclib.h. The implementation did not change, only the place they are defined. This API targets CHx01 GPR firmware and is backward compatible with ICUx0201 GPT firmware.

ch_rangefinder_types.h

This file contains definitions specific to the range finding feature but may need to be known by other types of firmware.

3.2 Data Structures

SonicLib uses two primary data structures to manage the sensors. Each sensor device is described by a **ch_dev_t** "device descriptor" structure, while each sensor group is described by a **ch_group_t** "group descriptor." Both of these structures are defined in **soniclib.h**.

Sensors and Sensor Groups

SonicLib manages the sensors both individually and as part of a sensor group. Groups allow multiple sensors to be operated together, for initialization or to coordinate multi-sensor range measurements.

Typically, an application will define only one sensor group, although it may use multiple sensors.

Every sensor belongs to a sensor group, even if there is only one sensor in the system. A group may consist of a single sensor or multiple sensors.

ch_dev_t – Device Descriptor

Each sensor is described by a **ch_dev_t** structure, and a pointer to this structure is used as a handle to identify the device in most SonicLib API calls.

> ***Note:*** By convention, this device descriptor pointer (**ch_dev_t ***) is named "**dev_ptr**". That name may appear in examples in this document without further definition.

The **ch_dev_t** structure contains fields for configuration values, status, function pointers, counters, etc. In general, you should never access the fields directly in your program – the SonicLib API provides dedicated "set" and "get" functions for values of interest to an application. Field names and organization within the **ch_dev_t** structure are subject to change without notice.

The application must allocate a separate **ch_dev_t** structure for each active sensor in the system. These structures are initialized by the *ch_init()* function, which must be called once for each device. The *ch_init()* function also adds each device to a sensor group.

ch_group_t – Group Descriptor

Each sensor group is described by a **ch_group_t** structure, and the address of this structure is used as an identifier in some SonicLib API calls that handle multiple sensors. Numerous BSP functions use the **ch_group_t** structure address as an input parameter for board-level functions.

The **ch_group_t** structure contains fields that reference the member devices, as well as status and configuration values that apply to all sensors in the group.

The application must allocate one **ch_group_t** structure for each sensor group. It is initialized during the *ch_init()* function, along with the device descriptor, when the device and group are associated. (Remember that there must be a sensor group defined, even if there is only one sensor in the system.)

The most important SonicLib API functions that operate on a group are *ch_group_start()*, which is always used during sensor initialization to program and start the sensors, and *ch_group_trigger()*, which triggers all sensors in a group together to start a measurement.

3.3 Board Support Package Files

A board support package (BSP) is a set of standard interfaces implemented for a specific hardware platform, to allow higher-level code to access the hardware resources. SonicLib defines a set of BSP functions to allow such generic access to the peripheral devices and other resources on the board. The **chirp_bsp.h** file contains the definitions of these hardware interfaces. All BSP functions are specified to use names beginning with the ***chbsp_*** prefix.

The BSP implementation is NOT part of SonicLib. On the contrary, the BSP contains control functions that SonicLib needs but cannot implement because they are hardware specific.

For example, SonicLib requires an accurate delay function with millisecond granularity, but such a routine can only be implemented with knowledge of the available hardware timers, etc. So, the SonicLib BSP interface defines the *chbsp_delay_ms()* routine, which SonicLib will call when needed. The BSP must therefore provide a routine called *chbsp_delay_ms()* that is implemented for the particular board and micro-controller being used.

Other defined functions similarly control pin levels, interrupts, etc. on the board. In general, these BSP routines may be implemented by "translating" the required operations into calls to the micro-controller vendor's I/O library functions.

The BSP must be provided separately by the application developer, board vendor, or InvenSense. Contact InvenSense for more information on available BSPs.

chirp_bsp.h

The **chirp_bsp.h** file defines the I/O interfaces that allow the standard SonicLib sensor driver functions to manage one or more sensors on a specific hardware platform. These include functions to initialize and control the various I/O pins connecting the sensor to the host system, the SPI or I^2C communications interface, interrupt handlers, timer functions, etc.

The BSP developer is responsible for implementing these support functions for the desired platform. Typically, this requires writing short routines with the specified *chbsp_* names and parameters that perform the necessary operations by calling lower-level functions in a hardware abstraction library (HAL) provided by the host MCU vendor. The BSP routines are basically a translation layer between the SonicLib callouts and the MCU library functions.

Some BSP functions are optional, depending on the specific runtime requirements (e.g., is non-blocking I/O required?) or development needs (e.g., is debugging support needed?). See the comments in **chirp_bsp.h** or the HTML documentation for more information on implementing the BSP functions and when they are required.

The **chirp_bsp.h** file should not be modified.

chirp_board_config.h

The board support package must supply a header file called **chirp_board_config.h** containing definitions of two symbols used in the SonicLib driver functions.

The following symbols must be defined in **chirp_board_config.h**. They are used to determine the size of arrays within the **ch_dev_t** device descriptor structure.

- **CHIRP_MAX_NUM_SENSORS** = maximum number of InvenSense ultrasonic sensors
- **CHIRP_NUM_BUSES** = number of SPI or I^2C bus interfaces that may have ultrasonic sensors attached
- **CHIRP_SENSOR_INT_PIN** = which sensor interrupt line (0 or 1) is used for interrupts (ICU sensors only)
- **CHIRP_SENSOR_TRIG_PIN** = which sensor interrupt line (0 or 1) is used for triggering (ICU sensors only)

In addition, the following symbols may be defined in the **chirp_board_config.h** file to indicate special operating conditions for CHx01 sensors only:

- **CHIRP_I2C_SPEED_HZ** = I^2C bus speed, if used for substitute RTC calibration
- **USE_STD_I2C_FOR_IQ** = use regular I^2C access for I/Q data instead of debug interface

The **chirp_board_config.h** file must be in the C pre-processor include path when you build your application with SonicLib.

3.4 Including SonicLib Headers from Application Files

As mentioned above, SonicLib does not impose any special requirements or structure on your application.

When the application is built with SonicLib, the **invn-soniclib** directory must be on the include path (see Figure 6 - SonicLib File Organization). SonicLib header file references normally use a partial path relative to this directory.

All application modules that will use the SonicLib interfaces must include **soniclib.h**:

```
#include <invn/soniclib/soniclib.h>
```

Modules that handle initialization or use board resources managed in the board support package also need to include **chirp_bsp.h**:

```
#include <invn/soniclib/chirp_bsp.h>
```

3.5 IDE Project

In order to use SonicLib to create your application, it must be built with an appropriate toolchain for the target platform. Generally, this means using an Integrated Development Environment (IDE) or other build tools to manage the files and build the project.

In addition to the files discussed here, a project for an IDE will usually also contain project definition files, library modules, and other files that are required to build the application. Because all IDEs are different, details on how to organize these additional files in your project are beyond the scope of this document.

Example ultrasonic sensing applications from InvenSense include full project definitions for supported IDEs. Contact TDK / InvenSense or see the **tdk.com** website for more information.

4 SENSING

This section introduces the sensor features and SonicLib functions used for basic ultrasonic sensing. The sensor(s) must have been initialized and configured, which will be described later, in sections 7 and 8.

> ***Note:*** In general, this section discusses rangefinding (distance measuring) type applications using the default sensor firmware types (e.g., **icu_gpt** for ICU sensors, and **ch101_gpr** or **ch201_gprmt** for CHx01 sensors). Most basic interfaces are used in the same way for other types of sensor algorithm firmware and applications (e.g. human presence sensing), although the actual output from the sensor will be different. For convenience, SonicLib includes all API features needed for rangefinding applications. For other sensor firmware algorithms, additional interfaces are provided in external libraries, etc. See the documentation included with the specific algorithm support.

The sensor is an ultrasonic transceiver, meaning that it both transmits and receives ultrasound signals. Unlike most other types of sensors which passively measure their surrounding conditions, the ultrasonic sensor actively injects a signal into its environment and then observes the results.

To perform a basic distance measurement, the sensor will emit a very brief pulse of ultrasound. It then immediately enters a "listening" state, in which it samples the received sound, attempting to identify an echo of the pulse that has been reflected off an object in the sensor's vicinity. If an ultrasound pulse is identified, the sensor will analyze the signal to determine the timing and then report the ToF of the received pulse. The actual distance travelled by the ultrasound during the ToF can then be calculated based on the speed of sound.

The overall cycle of measurements becomes a repeating sequence:

1. Start a new measurement cycle, either by triggering the sensor externally or by the expiration of the sensor's internal timer. The sensor will emit an ultrasound pulse.
2. Wait while the sensor listens for a received ultrasonic pulse (either reflected or from a different sensor).
3. The sensor will indicate data-ready by asserting its INT line.
4. The application's sensor interrupt callback routine (previously registered by *ch_io_int_callback_set()*) is called by SonicLib.
5. The callback routine can call *ch_get_range()* or other SonicLib functions directly to read the measurement results. However, because the callback function runs at interrupt level, it will typically set a flag causing the actual data readout to be done at task level in the regular application loop.

4.1 Starting a Measurement

An independent sensor in self-timed free-running mode (*CH_MODE_FREERUN*) does not require triggering – it will wake-up and begin a measurement based on its internal timer. In all other cases, the sensor must be triggered to begin a measurement cycle.

Triggering refers to the external initiation of a sensor measurement cycle. Usually, this is done by sending a signal on the sensor's INT line. ICU sensors also support "soft" triggering via SPI.

After being triggered, a sensor will begin and complete a measurement cycle, based on the current measurement and configuration settings. When the measurement cycle completes, each sensor will cause a "data-ready" interrupt on the host processor by asserting its configured INT line. (ICU sensors have two lines that may be configured for INT and/or triggering. CHx01 sensors have only a single line used for both functions.) The application should wait for the measurement to complete before reading the range or other data from the device. I/O operations with the sensor between the trigger and the data-ready signal may interfere with the ultrasonic measurement, particularly for CHx01 sensors.

Triggering Sensors

In most cases, all sensors in a group are used together to generate range measurements. Typically, one sensor will operate in transmit/receive mode (*CH_MODE_TRIGGERED_TX_RX*) and generate an ultrasonic pulse, while one or more other sensors

operating in receive-only mode (*CH_MODE_TRIGGERED_RX_ONLY*), in "Pitch-Catch" operation. In this situation, all sensors are triggered together, so that the receive-only nodes know when the pulse was transmitted and can calculate the time-of-flight when the pulse is received.

The *ch_group_trigger()* function is used to trigger all sensors in a group at the same time. It has the following definition:

```
void ch_group_trigger (ch_group_t *grp_ptr)
```

In more complex applications, it may be necessary to trigger individual sensors separately from the other sensors in the group. SonicLib provides a corresponding call, *ch_trigger()*, to trigger a single sensor. It has the following definition:

```
void ch_trigger (ch_dev_t *dev_ptr)
```

If there is only one sensor in the system, there is no practical difference between *ch_trigger()* and *ch_group_trigger()*.

Soft Triggering

Individual ICU sensors may be triggered via the SPI bus instead of the INT line, using the *ch_trigger_soft()* function:

```
void ch_trigger_soft (ch_dev_t *dev_ptr)
```

Soft triggering is not recommended for multi-sensor "pitch-catch" operation, because of timing uncertainty. See Multiple Sensors (Pitch – Catch) for more information.

Using a Periodic Timer

In many applications, the best way to manage measurement cycles is with a hardware-based timer. The SonicLib board support package definition includes a set of functions to access a periodic timer from an application. The periodic timer interfaces include a callback routine registration mechanism so that an application function will be called when the timer expires.

The periodic timer callback function is an excellent way to trigger new measurement cycles. The callback routine simply calls *ch_group_trigger()* or *ch_trigger()* each time it executes. These functions can be called from interrupt level, which is typically the case in the timer callback function.

How Often to Start a New Measurement

Most applications do not make single, isolated measurements. Instead, the application will likely need to continually perform new measurements to update its knowledge of the surrounding environment. This need to regularly obtain fresh measurement data highlights the important question of how often a new measurement can be started. (Note that these timing considerations apply to both externally triggered sensors and those running in free-running mode using an internal measurement interval.)

It is important that you do not start a new measurement if the full handling of the previous measurement has not yet completed. This includes the actual measurement by the sensor, handling of the data-ready interrupt from the sensor, and reading all data from the device via SPI or I^2C. If any of these operations are still active when the new measurement cycle begins, the new measurement may be corrupted.

The amount of time that a measurement cycle will require, including reading all data from the device, is the sum of several factors:

- As discussed above, the maximum range setting for the sensor will directly affect how long it takes to complete the actual time-of-flight measurement, because the sensor must listen while the ultrasound pulse travels the full round-trip distance. Each meter of range requires approximately 6 milliseconds of listening time. Note that the sensor always listens for the full period, recording sample data during this time.
- Between the end of the listening period and the assertion of the INT line to indicate data-ready, the sensor will examine and process the data. The internal processing time by the sensor will vary somewhat, but 5 ms is a useful estimate for CHx01 devices, and 1.5 ms is a useful estimate for ICU-x0201.

- As mentioned above, if static target rejection (STR) is used, it will increase the sensor's internal processing time by approximately 3 ms.
- The time to read basic measurement data (range and amplitude) over I^2C is approximately 1 ms.
- If raw I/Q measurement data is read from the device over SPI or I^2C, significant additional time is required.
    - For CHx01 sensors, the time will vary depending on how much data is read and the I^2C throughput in the BSP. Readout of a maximum CH101 data set (225 samples, or 900 bytes) will generally require 25 to 35 ms. Readout of a maximum CH201 data set (450 samples, or 1800 bytes) will generally require 50 to 60 ms.
    - For ICU-x0201, the maximum raw data size is typically 340 samples, or 1360 bytes of I/Q data. The readout time using 13MHz SPI is approximately 850 microseconds.

These values are summarized below in Table 2.

	Listen for Ultrasound	Sensor Processing	Static Target Rejection	Data transfer (basic)	I/Q data transfer (maximum samples)
CHx01	**6ms * (max range in meters)**	**5ms**	**3 - 6ms**	**1ms**	**25 - 35ms (CH101) 50 - 60ms (CH201)**
ICU-x0201	**6ms * (max range in meters)**	**1.5ms**	**1 - 3ms**	**0.5ms**	**1.5ms**

Table 2 – Measurement Time Requirements

All of these time requirements are only general estimates. To get the best and most reliable performance out of your application, you should analyze the time that will actually be required, based on the above factors, to determine how soon you will be able to start a new measurement. Leaving a margin for error is always a good idea. InvenSense recommends that you observe the actual timing of your running application, using a logic analyzer or similar tool, to confirm that the timing pattern is what you expect.

4.2 Reading the Range Value – *ch_get_range()*

The most basic operation performed by the sensor, beyond the actual generation and reception of the ultrasound pulse, is the calculation of the range (distance to target) based on the observed time-of-flight (ToF). The *ch_get_range()* function is used to obtain data from the sensor and calculate this value.

The *ch_get_range()* function has the following definition:

```
uint32_t ch_get_range (ch_dev_t *dev_ptr, ch_range_t range_type)
```

If the last completed measurement (i.e., the one which generated the INT line assertion) was successful in detecting a target object, the value returned by *ch_get_range()* is the calculated range (in fixed-point format, see Range Value Units below).

If the measurement did not detect a target object, *ch_get_range()* will instead return *CH_NO_TARGET* (0xFFFFFFFF). The amplitude value, as returned by *ch_get_amplitude()*, is not updated if no target was detected.

The *range_type* parameter specifies how the range value should be calculated from the ToF (i.e., one-way, round-trip, or direct). The choice of which range calculation fits your application depends on this physical path that the ultrasound follows when measuring a distance. See Sensing Configurations and Range Calculations.

When to Read the Range

ch_get_range() should only be called after the sensor has indicated that a measurement cycle is complete, by asserting its INT line. Typically, an application will be notified via a callback routine registered using *ch_io_int_callback_set()*.

The range value may be read at any time until a new measurement cycle is initiated (either by external triggering, as described above, or the internal sample interval expiring for a sensor in *CH_MODE_FREERUN* mode).

Range Value Units

The *ch_get_range()* function returns an integer value. To preserve sub-millimeter resolution from the sensor, the range value is reported in fixed-point format, with five fractional bits. Put another way, the range is reported in units of 1/32 millimeter.

To convert the returned range value to whole millimeters, divide by 32 (or shift right 5 bits).

You may want to use floating point arithmetic to preserve maximum resolution and convert the value for convenient display (i.e., divide by `32.0f`).

4.3 Reading the Target Amplitude Value – *ch_get_amplitude()*

Along with the range measurement value, the sensor also reports an amplitude value for the detected object's signal. In general, the amplitude value is much more variable than the range (due to interaction with other sounds, airflow around the sensor and other factors), but it may be useful in some applications.

The amplitude value is an internal unsigned integer value only – it is not calibrated to any standard units. So, it is only useful for relative comparisons. Properly operating sensors will be reasonably consistent in amplitude values, but part-to-part variation is expected.

The amplitude is very easily affected by subtle interactions due to acoustic conditions, both in the environment and the beam pattern produced by the sensor's own "horn" design. Most beam patterns have multiple lobes, and different angles will have different sensitivity. So, the same target at the same distance but in a slightly different position relative to the sensor can produce noticeably different amplitude readings.

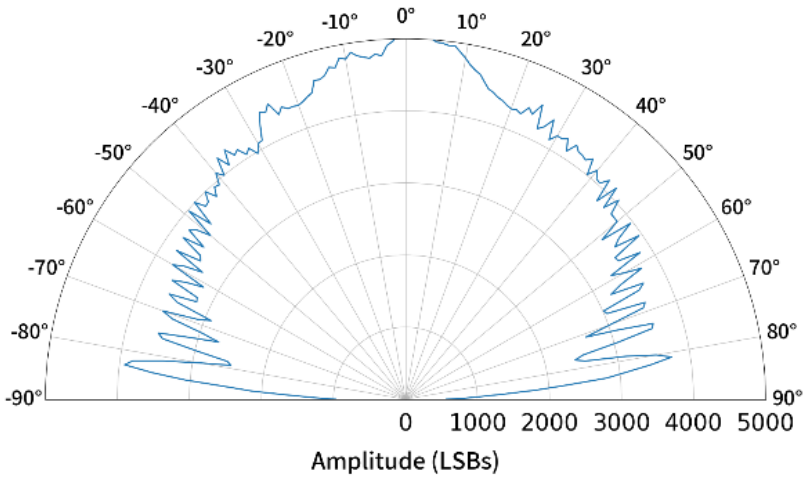

Figure 8 - Omni-Directional Beam Pattern (CH101)

Figure 8 shows the beam pattern for a device with a horn and housing that is well-tuned for omni-directional sensing. Note that differences in the amplitude still occur, depending on the exact angle from the front of the sensor. More directional horns or less optimized designs will have more pronounced lobes in the beam pattern.

The *ch_get_amplitude()* function has the following definition:

```
uint16_t ch_get_amplitude (ch_dev_t *dev_ptr)
```

The target amplitude value is only updated during successful range measurement in which a target is detected (*ch_get_range()* does not return *CH_NO_TARGET*). Otherwise, the amplitude stored during the last successful range measurement will be reported.

It is also possible to obtain or calculate the most recent amplitude values for any or all samples within a measurement, regardless of whether a target was detected. See Detecting Multiple Targets (ICU sensors only), below.

4.4 Detecting Multiple Targets (ICU sensors only)

In most applications, only the range (distance) of the closest detected target object is reported. This is the value returned by *ch_get_range()*.

ICU sensors using the standard **icu_gpt** sensor firmware can report the range to multiple target objects in a single measurement. A maximum of five (5) separate target ranges may be reported, along with their corresponding signal amplitudes.

When multiple targets are detected, they are identified by target number, with the closest target reported as Target 0, the next closest as Target 1, and so on. Note that the targets may be in different directions from the sensor, so consecutively numbered targets may in fact be located apart from each other.

The sensor does not "track" individual target objects from one measurement to the next – this must be done by the application. So, if two target objects are moving relative to each other, the reported target numbers may switch as the second target becomes the closest to the sensor.

The application must also handle dropout conditions, in which a target object is not detected during a particular measurement, although it was previously detected (and may be again in a future measurement). Note that this will affect the target numbers that are reported for any farther targets. So, some filtering or averaging of the results is recommended.

Number of Targets

The maximum number of targets (up to 5) that the sensor should report is specified by the application during measurement initialization. See Initializing a Measurement.

After a measurement completes, the application should determine the number of targets that were detected by calling the *ch_get_num_targets()* function. The returned target count determines how many target results to read from the device. If the count is zero, no targets were detected (equivalent to a return value of *CH_NO_TARGET* from *ch_get_range()*).

To allocate space for results or other needs, the *CH_MAX_NUM_TARGETS* symbol may be used. It equals the maximum number of targets supported by the current sensor firmware (5 for **icu_gpt**). The value returned by *ch_get_num_targets()* will range from zero to *CH_MAX_NUM_TARGETS*.

Reading Multiple Target Results

Each individual target range value may be read using the *ch_get_target_range()* function. This function works like *ch_get_range()* except that it takes the target number as a parameter to specify which range value to report.

Similarly, the amplitude of each individual target may be read using the *ch_get_target_amplitude()* function. This function works like *ch_get_amplitude()* except that it takes the target number as a parameter to specify which amplitude value to report.

The following code example shows how to read the range and amplitude values for all detected targets:

```
uint32_t  target_ranges[CH_MAX_NUM_TARGETS];      /* target range values */
uint16_t  target_amps[CH_MAX_NUM_TARGETS];        /* target amplitude values */
uint8_t   num_targets;                            /* number of targets detected */

num_targets = ch_get_num_targets(dev_ptr);

for (uint8_t target_num = 0; target_num < num_targets; target_num++)  {

    target_ranges[target_num] = ch_get_target_range(dev_ptr, target_num, CH_RANGE_ECHO_ONE_WAY);
    target_amps[target_num]   = ch_get_target_amplitude(dev_ptr, target_num);
}
```

4.5 Reading Sample Data: I/Q and Amplitude

TDK/InvenSense ultrasonic sensors provide access to the full internal sample data gathered during each measurement. This detailed data may be used in sensing applications to support custom algorithms or to analyze the performance of the sensor.

The I/Q data is the captured signal amplitude data from a measurement cycle, as recorded internally in the sensor. Each individual sample in the measurement is reported as a pair of values, I and Q, in a quadrature format. I/Q values retain the signal phase information from the receiver, which is lost once the data is converted to other formats.

As an alternative to the "raw" I/Q data value pairs, the sample data can be output as series of calculated amplitude values. Amplitude values may be more convenient because they are easily understood and may be used without further processing. Amplitude values in the sensor are expressed only in internal ADC counts (least-significant bits or LSBs) and are not calibrated to any standard units.

To convert any given I/Q pair to the amplitude of that sample, square both I and Q, and take the square root of the sum:

$$\text{Amplitude} = \sqrt{I^2 + Q^2}$$

The SonicLib *ch_iq_to_amplitude()* utility function can be used to perform this calculation for an I/Q pair using integer arithmetic (no floating point).

Number of Samples

The number of samples in a measurement determines the amount of the I/Q or amplitude data. For I/Q output, each sample is reported as a signed 16-bit I value and a signed 16-bit Q value. So, the total I/Q size is 4 bytes times the number of samples in the measurement. For amplitude output, each sample amplitude is one 16-bit value, so the total size is 2 bytes times the number of samples.

Different sensor firmware types can support different maximum numbers of samples. So, the maximum possible I/Q or amplitude size will vary based on the sensor firmware being used. Table 3 shows the maximum sample counts and output sizes for various sensor firmware types. These values may be used for allocation of buffers that will receive sample data from the sensor.

In CH101 and CH201 sensors, the output format cannot be changed – it is specified by the sensor firmware being used. In almost all cases, only I/Q output (*CH_OUTPUT_IQ*) is available directly. (The **ch101_floor** firmware has amplitude data output only.)

Sensor Model	Sensor Firmware	Maximum Sample Count	Max I/Q Data (bytes)	Max Amplitude Data (bytes)
ICU-*x*0201	**icu_gpt**	340	1360	680
	icu_medfilt	105	420	210
	icu_presence	300	1200	-
CH101	**ch101_finaltest**	150	600	–
	ch101_floor	150	–	300
	ch101_gppc	350	1400	–
	ch101_gpr_*xxx* (all variants)	225	900	–
	ch101_gprmt	225	900	
CH201	**ch201_finaltest**	450	1800	–
	ch201_gprmt	450	1800	–
	ch201_gprstr	290	1160	–
	ch201_presence	300	1200	–

Table 3 – Maximum Sample Count vs. Sensor Model and Firmware Type

For ICU sensors using **icu_gpt** firmware, the *ICU_MAX_NUM_SAMPLES* symbol may be used for the above maximum sample count value.

The maximum sample count can be obtained programmatically using the *ch_get_max_samples()* function. It will return the maximum number of samples per measurement that is supported by the sensor firmware on the device. Note that this may be larger than the number of samples that are actively being used, depending on the current maximum range setting.

The number of samples used in the measurement is determined by the maximum range setting for the device. If it is less than the maximum possible, not all samples will contain valid data. Use the *ch_get_num_samples()* function to find out the number of samples that are active for the current maximum range setting.

Reading I/Q Data

SonicLib provides the *ch_get_iq_data()* function to easily read the raw I/Q data from a device. The function can read all the I/Q data or any specified subset based on sample number. The data may either be read immediately (blocking mode), or the read operation may be queued for completion via DMA (non-blocking mode, discussed below).

The *ch_get_iq_data()* function has the following definition:

```
uint8_t ch_get_iq_data (ch_dev_t *dev_ptr, ch_iq_sample_t *buf_ptr, uint16_t start_sample,
                        uint16_t num_samples, ch_io_mode_t mode)
```

The buffer specified by *buf_ptr* will receive the I/Q data, whether in blocking or non-blocking *mode*. The buffer is defined as an array of **ch_iq_sample_t** structures, each of which has fields for the 16-bit I and Q values. Note that in each sample, the Q value is actually output before the I value, and the structure fields are ordered accordingly. The buffer must be large enough to hold the requested number of 4-byte sample values.

An application will often need to read the entire raw I/Q data set from a measurement cycle. In this case, the *start_sample* parameter should be zero. The *num_samples* value should be the total number of active samples per measurement, based on the current maximum range setting. This value can be obtained by using *ch_get_num_samples()*. (It is possible to read more than the active number of samples, but the contents of samples beyond the current maximum range are undefined.)

In other cases, an application may be interested in a specific subset of the I/Q sample set. For example, if the application only needs to read the I/Q data for a certain range of physical distance, it may use the *ch_mm_to_samples()* function to determine the appropriate values for *start_sample* and *num_samples*.

> ***Note:*** CH101 and CH201 sensors are restricted in the maximum starting sample offset for I/Q readout, due to the limitations of 8-bit I^2C addressing.

When the I/Q data is read from the sensor, the additional time required to transfer the I/Q data over the bus must be considered when planning how often the sensor will be read (measurement interval). This is particularly a concern with CH101 and CH201 sensors, which use I2C connections and require longer transfer times. See the previous discussion in How Often to Start a New Measurement.

Reading Amplitude Data

As an alternative to the "raw" I/Q data value pairs, the sample data can be output as series of calculated amplitude values. Output of amplitude values may be more convenient because they are easily understood and displayed. In this format, each sample is represented by a single 16-bit unsigned amplitude value. So, the total amplitude data set size is 2 bytes times the number of samples in the measurement.

SonicLib provides the *ch_get_amplitude_data()* function to easily read the sample amplitude data from a device. The function can read the values for all samples, or any specified subset based on sample number. The data may either be read immediately (blocking mode), or the read operation may be queued for completion via DMA (non-blocking mode, discussed below).

The *ch_get_amplitude_data()* function has the following definition:

```
uint8_t ch_get_amplitude_data(ch_dev_t *dev_ptr, uint16_t *buf_ptr, uint16_t start_sample,
                              uint16_t num_samples, ch_io_mode_t mode)
```

The buffer specified by *buf_ptr* will receive the amplitude data, whether in blocking or non-blocking *mode*. The buffer is defined as an array of **uint16_t** values.

To read the entire set of amplitude values from a measurement, the *start_sample* parameter should be zero. The *num_samples* value should be the total number of active samples per measurement, based on the current maximum range setting. This value can be obtained by using *ch_get_num_samples()*. The array specified by *buf_ptr* must be large enough to receive this number of 16-bit values.

ICU sensors must have the sample output format set to *CH_OUTPUT_AMP* to send amplitude values instead of the default I/Q pairs. This is specified by using *ch_set_output_format()* or *ch_meas_set_iq_output()*. See Sample Data Output Format for additional information.

CHx01 sensor have significant limitations due to an 8-bit limit on I2C address offsets combined with lack of a native amplitude output mode in most firmware types. In general, you should use *ch_get_iq_data()* to read the I/Q data, then convert it to amplitude values using *ch_iq_to_amplitude()*.

Non-blocking Read Mode (optional)

To allow more flexibility in your application, the I/Q data from the device may be read in non-blocking mode, by setting *mode* to *CH_IO_MODE_NONBLOCK*. In non-blocking mode, the readout takes place as a background operation by the host processor, typically using DMA transfers for the I/O. The *ch_get_iq_data()* or *ch_get_amplitude_data()* function will return to the caller

immediately and must be followed by a call to *ch_io_start_nb()* to start the non-blocking read. A notification will later be issued when the sensor data has all been read.

To use the non-blocking option, the board support package (BSP) you are using must provide the optional non-blocking read routines (*chbsp_spi_read_nb()* and *chbsp_spi_read_mem_nb()* for ICU sensors, or *chbsp_i2c_read_nb()* and *chbsp_i2c_read_mem_nb()* for CHx01 sensors). To use non-blocking reads of the I/Q data, you must specify a callback routine that will be called when the I/Q read completes. See *ch_io_complete_callback_set()*.

Non-blocking reads are managed together for a group of sensors. To perform a non-blocking read:

1. Register a callback routine using *ch_io_complete_callback_set()*.
2. Define and initialize a handler for the DMA interrupts generated.
3. Begin sensing and wait for the sensor(s) to indicate data-ready.
4. Set up a non-blocking read on each sensor, using *ch_get_iq_data()* or *ch_get_amplitude_data()* with *mode* = *CH_IO_MODE_NONBLOCK*.
5. Start the non-blocking reads on all sensors in the group, using *ch_io_start_nb()*.
6. Your callback routine (set in step #1 above) will be called as each individual sensor's read completes. The callback routine should initiate any further processing of the I/Q or amplitude data, possibly by setting a flag that will be checked from within the application's main execution loop. The callback routine will be called at interrupt level, so the amount of processing within it should be kept to a minimum.

4.6 Sample Data Output Format

The full sample data from the sensor may be read after each measurement, independent of the reported target range and amplitude values. This detailed information can then be used by application-specific algorithms as needed.

ICU sensors can output the data in three different formats, depending on what is most useful for the application. The output format is selected by calling the *ch_meas_set_iq_output()* or *ch_set_data_output()* function. This is usually done during initialization, but the format can be changed dynamically if needed. Each output format has a corresponding *ch_get_*xxx*_data()* API function to read the data, described below.

In CH101 and CH201 sensors the output format cannot be changed – it is specified by the sensor firmware being used. In almost all cases, only I/Q output (*CH_OUTPUT_IQ*) is available directly on CH101 or CH201. (The **ch101_floor** firmware has amplitude data output.)

CH_OUTPUT_IQ

The "raw" signed 16-bit integer I and Q values are reported for each sample as a **ch_iq_sample_t** structure. Each sample requires four bytes total. The Q value is output before I in each pair.

The I/Q data retains signal phase information for the samples, which is lost in other formats. This allows more complex analysis for special applications. The *ch_iq_to_amplitude()* function can be used to calculate amplitude values from sample I/Q pairs, if needed.

Data in this format should be read using *ch_get_iq_data()*. This function allows any specified set of samples to be read.

CH_OUTPUT_AMP

An unsigned 16-bit integer amplitude value is reported for each sample (calculated from the I/Q values). Each sample requires two bytes total.

This output format is often the most convenient for everyday use because the amplitude values are more "human readable" than I/Q pairs. The array of amplitude values from a measurement can easily be transferred to a spreadsheet program to generate an "Amplitude Scan" or A-Scan chart, similar to the graphic display in the ICU-x0201 EVK sensor evaluation kit.

Data in this format should be read using *ch_get_amplitude_data()*. This function allows any specified set of samples to be read.

CH_OUTPUT_AMP_THRESH

Unsigned 16-bit integer values for both the signal amplitude and the detection threshold level are reported for each sample in a measurement. The amplitude value is the same as in *CH_OUTPUT_AMP*. The threshold value is the value that was set by *ch_set_thresholds()* for that particular sample.

This output format is not normally used in regular applications, but it may be useful temporarily during development to assist in tuning the threshold values. The threshold and amplitude values may easily be plotted together to see how the thresholds compare to the actual received signal across the measurement.

Data in this format should be read using *ch_get_amp_thresh_data()*. This function allows any specified set of samples to be read.

4.7 Target Interrupt Filtering

In most cases, the sensor will generate a data-ready interrupt at the completion of each measurement. In rangefinding applications, this means that the sensor will always interrupt whether a target was detected or not.

It is possible to modify the interrupt behavior using Target Interrupt Filtering, which allows the application to control whether the sensor will generate an interrupt after every measurement, or only if a target is detected. By default, a data ready interrupt is generated after every measurement whether a target object was detected or not. (Note that in ICU sensors, the "Done Interrupt" must be enabled for the final receive segment in the measurement. See Receive Segments.)

This feature is often combined with Static Target Rejection (STR) to reduce the number of data ready interrupts, for power conservation.

The *ch_set_target_interrupt()* function enables the filtering of data ready interrupts from the sensor based on target detection and specifies the type of filter behavior:

```
uint8_t ch_set_target_interrupt(ch_dev_t *dev_ptr, ch_tgt_int_filter_t tgt_int_filter)
```

CH_TGT_INT_FILTER_OFF

In normal operation (if target detection interrupt filtering is not enabled), the sensor will assert the INT line at the end of each measurement even if no target is detected. This is the default behavior if *ch_set_target_interrupt()* is not called, or if *tgt_int_filter* is *CH_TGT_INT_FILTER_OFF*.

CH_TGT_INT_FILTER_ANY

When basic target interrupt filtering is enabled, the sensor will assert the INT line at the end of each measurement only if a target object was detected. If no target is detected, the sensor will not interrupt, and there is no indication from the sensor that the measurement has completed.

To use basic target interrupt filtering, set *tgt_int_filter* to *CH_TGT_INT_FILTER_ANY*.

CH_TGT_INT_FILTER_COUNTER

> ***Note:*** Some types of sensor firmware support an additional filtering option, target counter mode. See the description of the specific firmware you are using to determine if target counter mode is supported.

Target counter mode provides another level of filtering, across multiple measurements. When counter filtering is enabled, the sensor maintains a history of target detections and will only interrupt if multiple measurements have detected a target within a certain number of consecutive measurements.

A configurable history of previous measurements is recorded, and the results are compared against a threshold number of target detections. The sensor will generate an interrupt only when the count of target detections within the set of recent measurements meets the threshold value,

To use counter mode filtering, set *tgt_int_filter* to *CH_TGT_INT_FILTER_COUNTER*.

If not otherwise specified, the default measurement history length is five (5) measurements. The current measurement is always combined with the measurements from the history, so a total of six (6) measurements are used. The default interrupt threshold is three (3) target detections. So, at least three of the six most recent measurements must detect a target to generate an interrupt.

The length of the measurement history and the target detection threshold count used in the filter can be set by using the *ch_set_target_int_counter()* function:

```
uint8_t ch_set_target_int_counter(ch_dev_t *dev_ptr, uint8_t meas_hist, uint8_t thresh_count,
                                  uint8_t reset)
```

By default, the counter filter maintains its history and counter values even after an interrupt is generated, so the next measurement may again interrupt, if the detection threshold is again met. The *reset* parameter configures the filter to instead clear its history after generating an interrupt by using *ch_set_target_int_counter()*. In this case, the *thresh_count* threshold must be reached based on new detections, so at least that number of measurements will occur before another interrupt can be generated.

4.8 Static Target Rejection (STR)

You may sometimes find that the sensor's field-of-view allows it to detect unexpected or undesirable objects as targets, such as on a cluttered desktop or when located near a wall. Static Target Rejection (STR) is a runtime option to help manage this situation. It causes the sensor's internal detection algorithm to ignore static (non-moving) objects when determining the range to the nearest moving target.

When STR is enabled, the sensor applies a moving-average high-pass filter over a specified set of samples. In normal operating conditions with a stationary sensor and a non-moving set of target objects, the steady amplitude from the target echoes will allow the filter to reject their signals. Targets whose reflections do not change between measurements will be ignored. However, any moving targets will vary in distance and amplitude, and even movements of millimeters can create a detectable change in the overall measured signal. Note that some conditions, such as high airflow around the sensor, can cause the echo amplitude from a static object to fluctuate enough to register as a target.

When enabled, STR always affects a contiguous block of samples at the start of a measurement. Any number of samples, up to the entire measurement range, may use STR. Any samples in the measurement beyond the STR range will have normal target detection (and may report non-moving objects).

STR may be enabled using the *ch_set_static_range()* function. See Static Target Rejection Settings.

STR is often combined with Target Interrupt Filtering to reduce the number of data ready interrupts, for power conservation.

4.9 Receive Holdoff

The receive holdoff setting specifies a range of samples at the start of each measurement that will be ignored for target detection. Although ultrasound data from these samples will be measured and recorded (and will be reported if amplitude or I/Q output is enabled), no targets will be reported within this range. This may be useful for ignoring objects close to the sensor or handling difficult acoustic situations.

The *ch_set_rx_holdoff()* function is used to set the receive holdoff. It specifies the number of samples to be ignored, beginning at the start of the measurement (closest objects). The *ch_mm_to_samples()* conversion routine may be helpful in determining the number of samples, to ignore within a specific physical distance.

> ***Note:*** For ICU sensors, the receive holdoff setting works by modifying the detection thresholds for the specified range of samples. The specified number of samples will be given a very high (unreachable) threshold level. The detection thresholds should be applied normally before *ch_set_rx_holdoff()* is called.
>
> To "undo" this operation, the original threshold values must be re-applied using *ch_set_thresholds()*. The effects cannot be reversed by simply reducing the Rx Holdoff sample count.

4.10 Ringdown Filtering

The sensor always performs "ringdown cancellation" filtering to remove artifacts from the received signal in the first samples within a measurement. Ringdown is a high amplitude false signal caused by residual vibration in the ultrasonic transducer (PMUT) after sending the transmit pulse. The ringdown signal amplitudes are typically higher than actual echo results. A similar but smaller artifact occurs even in receive-only sensors, when the PMUT is energized at the beginning of a measurement.

Ringdown filtering is always applied to a contiguous block of samples starting at the beginning of a measurement. In ICU sensors, the number of samples to be filtered is configurable. The number of ringdown samples is specified in the *ch_meas_init()* function, in a field within the **ch_meas_config_t** structure. See Initializing a Measurement.

ICU sensors can optionally perform transmit optimization, in which a customized sequence of transducer instructions is constructed to dampen the ringdown effect. Transmit optimization can reduce the duration of the ringdown and thereby reduce the number of samples that must be included in the ringdown filter range. See Transmit Optimization for details.

In CH101 and CH201 sensors, the number of samples subject to the ringdown filter is fixed for each sensor firmware type.

Effects on Target Detection

The ringdown signal profile is generally constant from one measurement to the next, so the filter works by comparing new and previous results and removing the non-changing signal components. Within the ringdown-filtered samples (i.e., very close to the sensor), non-moving objects will generally not be reported, because their constant reflection signals will also be filtered out. However, close objects that are moving will still be detected. This is effectively the same result as enabling Static Target Rejection (STR) filtering within that range of samples.

4.11 Pausing Sensing

It is very easy to temporarily disable and re-enable the ultrasound measurements in a running application by using *ch_set_mode()*. The sensor is placed in idle mode (*CH_MODE_IDLE*) to disable (pause) sensing, and it is later put back into regular free-running or triggered mode to resume measurements. The idle mode of the sensor is the lowest power state.

The following lines will save the current operation mode and place the sensor in idle mode:

```
ch_mode_t saved_mode = ch_get_mode(dev_ptr);
ch_set_mode(dev_ptr, CH_MODE_IDLE);
```

Note: When placing a running sensor into idle mode, avoid changing the mode while a measurement is being made. For sensors in *CH_MODE_FREERUN*, you should change the mode soon after the sensor has issued a data-ready interrupt indicating the end of a measurement. For sensors in triggered modes, you may use the same technique or suspend triggering while the mode is being changed.

The following line will put the sensor back into the original sensing mode:

```
ch_set_mode(dev_ptr, saved_mode);
```

No extra re-configuration is necessary to resume sensing. If no other settings have been changed, the measurements will begin with the same parameters that were active before entering idle mode.

5 MEASUREMENT CONTROLS (ICU SENSORS)

This section describes the features that SonicLib provides to control the ultrasound measurement in ICU sensors. ICU sensors allow the application to define the timing and other details of transmit and receive operations within a measurement. Earlier CHx01 sensors generally cannot change the measurement behavior dynamically and are limited to the settings built into the sensor firmware.

API functions that control the measurement settings have names beginning with *ch_meas*.

5.1 Measurement Configurations

ICU sensors support two independent measurement configurations, so the measurement number is used as an identifier (see Active and Standby Measurements, below). Each measurement configuration consists of a set of operational settings as well as a *measurement queue definition*. Together, these provide full low-level control of the sensor operation.

A measurement queue consists of a series of *segments*, which contain sensor instructions for transmitting or receiving ultrasound. Each segment has a specified duration, and when one segment completes, the next is executed. This mechanism of sequential segments that combine to form the overall measurement is the *measurement queue*.

To define a measurement, the application must first initialize the measurement and set various options using *ch_meas_init()*. Then, a series of segments is added to the measurement using *ch_meas_add_segment()* or equivalent routines. After all segments have been added, the measurement definition is written using *ch_meas_write_config()*. The different types of segments and their options are described below.

Complete defined sensor configurations, including both measurement configurations, may be imported rather than constructing them using individual SonicLib API functions. These external definitions may have been generated by the ICU-x0201 Evaluation Kit or may be manually created. See Importing a Measurement Definition for details.

5.2 Initializing a Measurement

Before segments can be added, a measurement must be initialized using *ch_meas_init()*:

```
uint8_t ch_meas_init(ch_dev_t *dev_ptr, uint8_t meas_num, const ch_meas_config_t *meas_config_ptr,
                     const void *thresh_ptr)
```

This function initializes the measurement specified by *meas_num* with the specified configuration. The argument *meas_config_ptr* is a pointer to a *ch_meas_config_t* structure which must have already been initialized. The fields in this structure specify various parameters for the measurement, including:

- Output data rate (ODR) of the ultrasonic transducer – the sampling rate within each measurement (see below).
- Measurement period – for sensors in CH_MODE_FREERUN, this specifies how often a measurement will begin.
- Mode – if CH_MEAS_MODE_STANDBY, this measurement will initially be put in standby mode (not active), if CH_MEAS_MODE_ACTIVE (zero), this measurement will initially be active and will be performed.
- Note that usage of *thresh_ptr* is deprecated and this argument is not used. You should pass NULL for this argument. To specify the GPT firmware rangefinding-specific configuration, use the function *icu_gpt_algo_configure()* as described below.

Initializing and configuring GPT firmware rangefinding-specific features

The functions *icu_gpt_** below are found in the header icu_gpt.h located within the invn/soniclib/sensor_fw/icu_gpt directory. This is technically a plug-in layer, but it is included with the base SonicLib distribution since it supports the GPT (rangefinding) firmware.

To configure the GPT algorithm do the following steps:

1. **Initialize algorithm configuration variable**

```
uint8_t icu_gpt_algo_init(ch_dev_t *dev_ptr, InvnAlgoRangeFinderConfig *algo_cfg);
```

This function stores the pointer to the variable which will contain the algorithm configuration into devices variables.

2. **Configure algorithm**

```
uint8_t icu_gpt_algo_configure(ch_dev_t *dev_ptr, uint8_t meas_num,
                               const icu_gpt_algo_config_t *algo_config_ptr,
                               const ch_thresholds_t *lib_thresh_ptr)
```

This function initializes the measurement specified by meas_num with the specified configuration and detection thresholds.

- num_ranges - maximum number of separate target range values to report
- ringdown_cancel_samples - number of samples close to sensor to use ringdown cancellation filter
- static_filter_samples - number of samples close to sensor to have static target rejection (STR) filter
- iq_output_format - 0=normal (Q,I) pairs; 1=mag,threshold pairs, 2=mag,phase pairs
- filter_update_interval - how often to update the ringdown and STR filters

Additionally, the argument thresh_ptr is a pointer to a structure containing the definitions of the detection thresholds that will be used to detect a target. (This same type of structure is used by ch_set_thresholds().)

3. **Send algorithm configuration to sensor**

```
uint8_t ch_set_algo_config(ch_dev_t *dev_ptr, const void *algo_cfg_ptr)
```

This function is common to all firmware and will send the algorithm configuration pointed by *algo_cfg_ptr* to the sensor.

4. **Initialize sensor with new algorithm configuration**

```
uint8_t ch_init_algo(ch_dev_t *dev_ptr)
```

This function is common to all firmware and triggers the initialization of algorithm with new configuration on sensor.

Output Data Rate (ODR)

ICU sensors allow the application to select the sample output data rate (ODR) in the sensor, i.e., how often a sample is taken within each measurement. This allows more precise measurements at close distances, by increasing the rate at which internal samples are taken. Conversely, the number of samples in each measurement can be reduced without changing the maximum range (perhaps to conserve sample data storage) by decreasing the ODR.

The ODR is specified relative to the acoustic operating frequency of the ultrasonic transducer (PMUT). By default, the ODR is 1/8 the operating frequency. So, if a sensor operates at 80 kHz, the default sample rate will be 10,000 samples per second.

Possible ODR values are shown in Table 4:

ch_odr_t Name	Sample Rate
CH_ODR_FREQ_DIV_2	Operating freq / 2
CH_ODR_FREQ_DIV_4	Operating freq / 4
CH_ODR_FREQ_DEFAULT or *CH_ODR_FREQ_DIV_8*	Operating freq / 8
CH_ODR_FREQ_DIV_16	Operating freq / 16
CH_ODR_FREQ_DIV_32	Operating freq / 32

Table 4 - Output Data Rate (ODR) Options

Increasing the ODR from *CH_ODR_FREQ_DEFUALT* to *CH_ODR_FREQ_DIV_4* or *CH_ODR_FREQ_DIV_*2 will cause samples to be taken more rapidly. The maximum possible range for the sensor is reduced, because the maximum number of samples will be reached sooner. However, there are advantages to running at increased ODR, particularly for sensing at short ranges.

The other impact of increasing the ODR is that less filtering will be performed on the raw signal recorded by the sensor's ADC. As a result, more noise passes through the DSP filter and appears in the sample values. As a rule of thumb, increasing the **ch_odr_t** value by one step increases the noise in the raw signal by a factor of $\sqrt{2}$. The detection thresholds may have to be increased accordingly, if they were chosen based on the noise floor of the sensor.

5.3 Transmit Segments

In general, ultrasonic sensors generate (transmit) a burst of ultrasound at the start of each measurement. The sensor measurement definition contains one or more **Transmit Segments** that control how the ultrasonic signal is generated by the sensor. Various parameters affecting the transmitted pulse may be defined, to control different performance characteristics.

Receive-only sensors operating in synchronization with another sensor do not include a transmit segment in their measurement definitions. However, they may include a Count Segment to match the transmit timing – see below.

In general, you will only need to define a single transmit segment in a measurement.

> ***Note:*** If transmit optimization is performed, additional transmit segments will be added automatically to actively dampen the transducer. See Transmit Optimization.

Transmit segments may be added to a measurement definition using the *ch_meas_add_segment_tx()* function:

```
uint8_t ch_meas_add_segment_tx(ch_dev_t *dev_ptr, uint8_t meas_num, uint16_t num_cycles,
                               uint8_t pulse_width, uint8_t phase, uint8_t int_enable)
```

Alternatively, a **ch_meas_segment_t** structure can be initialized via the *ch_meas_init_segment_tx()* function or other means, then added using *ch_meas_add_segment()*. The transmit control parameters are the same in all cases.

Figure 9 depicts the beginning of an ultrasound burst and shows the transmit segment values that are used to control the transmitted waveform. Periods when the ultrasound transducer is actively driven are indicated in orange.

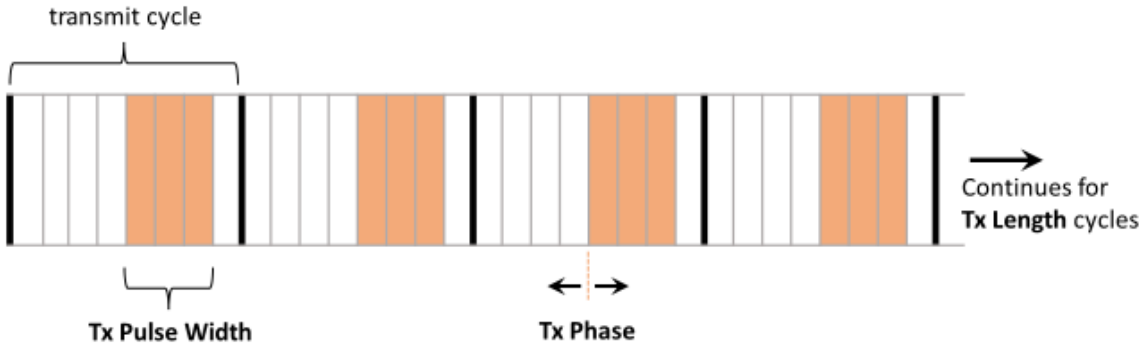

Figure 9 – Transmit Cycles & Controls

Transmit Length

The ultrasound burst sent by the sensor is timed by counts of a clock running at 16 times the acoustic operating frequency. The **transmit length** specifies the total number of transmit cycles and therefore determines the overall duration of the burst transmission. In general, a burst using a longer transmit length will have a higher amplitude when the signal is received. However, this behavior is saturating, so sending an arbitrarily long transmit pulse will not keep increasing the amplitude once a certain threshold has been reached. When the saturation point is reached, the MEMS or DSP filter component is said to be "fully excited." The particular threshold depends on the ODR value (sampling rate in the receiver) – slower ODR settings require longer transmit pulses to saturate, as is shown in Table 5.

For sensing objects close to the sensor, it may be advantageous to reduce the transmit length so it is shorter than the values given in the table below. See application note AN-000400 for more details.

ODR (ch_odr_t)	Clock counts per Rx sample (16 * op freq)	Min Tx length to excite MEMS	Min Tx length to excite DSP filter
CH_ODR_FREQ_DIV_32	512	Approx. 640	1536
CH_ODR_FREQ_DIV_16	256	Approx. 640	768
CH_ODR_DEFAULT ***CH_ODR_FREQ_DIV_8***	128	Approx. 640	384
CH_ODR_FREQ_DIV_4	64	Approx. 640	192
CH_ODR_FREQ_DIV_2	32	Approx. 640	96

Table 5- Minimum Transmit Counts to Excite MEMS and DSP Filter

Transmit Pulse Width

Within each transmit cycle, the transmitter is active for a brief pulse and then recovers. The **transmit pulse width** is the length of the positive pulse of the transmit waveform. Making it shorter will reduce the effective drive strength.

The pulse width units are one eighth (1/8) of a transmit cycle. Valid values are from 1 to 4. Table 6 shows the theoretical relationship between the output sound amplitude and the pulse width setting.

Transmit PW	Duty Cycle	Amplitude (relative to PW=4)
0	0	0
1	0.125	0.383
2	0.25	0.707
3	0.375	0.924
4	0.5	1

Table 6 - Theoretical Transmit Amplitude vs. Transmit Pulse Width

Note: A transmit pulse width of 4 necessarily disables a power-saving feature in the sensor that is otherwise active, so that setting will have an extra impact on power consumption.

Transmit Phase

The **transmit phase** value specifies the timing of the pulse within the transmit cycle, relative to the internal clock generator (start of transmit cycle). Valid values are from 0 to 15. Each unit of phase is $1/16^{th}$ of a cycle.

Done Interrupt

The final transmit segment value specifies whether the sensor should immediately generate an interrupt at the end of the transmission (non-zero = generate interrupt). Note that this occurs in the middle of the measurement, before the listening period.

This option is normally not supported for transmit segments except for special test or diagnostic applications.

5.4 Count Segments

Another type of segment used in a measurement definition is the Count segment. This is a special entry in the measurement that simply introduces a delay before the next segment becomes active.

Count segments are used less often than the regular transmit and receive segments. One typical use is in a sensor that operates in receive-only mode (*CH_MODE_RX_ONLY*). A sensor in this mode will not have any transmit segments in its measurement definition, but it can instead use a count segment before the receive segments, to match the timing of the transmitting sensor. In this case, the count segment cycle count should match the other sensor's transmit cycle count.

Count segments may be added to a measurement definition using the *ch_meas_add_segment_count()* function:

```
uint8_t ch_meas_add_segment_count(ch_dev_t *dev_ptr, uint8_t meas_num, uint16_t num_cycles,
                                  uint8_t int_enable)
```

Alternatively, a **ch_meas_segment_t** structure can be initialized via the *ch_meas_init_segment_count()* function or other means, then added using *ch_meas_add_segment()*.

Count Cycles

This parameter specifies the length of the delay generated by the count segment. The units are sensor cycles (the same as transmit cycles in transmit segments).

Done Interrupt

This value specifies whether the sensor should immediately generate an interrupt at the end of the delay (non-zero = generate interrupt).

This option is normally not supported for count segments.

5.5 Receive Segments

The listening phase of a measurement is defined by one or more receive segments, which control the number of samples that will be taken and the settings for the ultrasound receiver.

Unlike transmit segments, it is common to specify multiple receive segments in a measurement definition. Often, reduced gain settings are used for a small number of samples very close to the sensor (immediately after the transmit pulse is sent). This technique can limit the effect of ringdown artifacts and help avoid false positive detections. In this case, two receive segments are typically used: one for just the first few samples, and another for the rest of the measurement range using normal gain.

Receive segments may be added to a measurement definition using the *ch_meas_add_segment_rx()* function:

```
uint8_t ch_meas_add_segment_rx(ch_dev_t *dev_ptr, uint8_t meas_num, uint16_t num_samples,
                               uint8_t gain_reduce, uint8_t atten, uint8_t int_enable)
```

Alternatively, a **ch_meas_segment_t** structure can be initialized via the *ch_meas_init_segment_rx()* function or other means, then added using *ch_meas_add_segment()*. The receive control parameters are the same in all cases.

Receive Length (Sample Count)

For convenience, the length of a receive segment is specified as a certain number of receive samples that will be taken, using the specified gain and attenuation settings (see below). The sensor will perform this many samples, then the next segment (if any) will execute.

The total length of all receive segments (the total sample count) determines the listening time, and therefore the maximum detection range, for the device.

Receive Gain Reduction

This setting allows the gain of the receiver to be reduced for the samples in this receive segment. The units are dB of gain reduction (from normal). Possible values are 0 (no reduction in gain) to 31 (maximum reduction), but values 28-31 all result in the same reduction of approximately 28dB.

Receive Attenuation

This setting attenuates (reduces) the receiver signal, to avoid saturation. This control is similar to gain reduction but is more dramatic in its effect.

Valid values are 0 (no attenuation) to 3 (maximum attenuation). Each increment (starting from zero) will drop the effective receiver gain by a factor of 8.

Done Interrupt

Normally, the last receive segment in a measurement definition will enable the "Done" interrupt. This value specifies whether the sensor should generate an interrupt after the receive segment is complete.

For the last receive segment, the sensor may complete other processing before issuing the interrupt. During measurements using standard "rangefinding" sensor firmware (e.g., **icu_gpt** for ICU sensors), the range calculation will be completed before the interrupt is generated by the sensor.

The specific conditions for the interrupt to be generated are subject to the Target Interrupt settings described above.

5.6 Applying the Measurement Definition

After the last receive segment has been added to the measurement definition, it must be finalized and written to the sensor, using the *ch_meas_write_config()* function:

```
uint8_t ch_meas_write_config(ch_dev_t *dev_ptr)
```

After the measurement definition has been written, the sensor will use it when sensing begins.

5.7 Active and Standby Measurements

The ICU sensor can simultaneously hold two independent measurement definitions (measurement queues). The two measurements are simply identified by number, Measurement 0 (also known as *CH_MEAS_DEFAULT*) and Measurement 1.

In most applications, only a single measurement definition (*CH_MEAS_DEFAULT*) is used, and the second measurement does not need to be defined. However, it is possible to use the second measurement to pre-configure an alternate set of transmit and/or receive characteristics to be used on demand. This alternate definition can be activated quickly without requiring sensor re-initialization. This may be useful if the sensor must switch sensing modes based on activity or needs to "simultaneously" use different settings (e.g., short plus long-range detection) in alternation.

Any defined measurement is in either "active" or "standby" mode. In active mode, the measurement definition will be used by the sensor to perform readings. In standby mode, the measurement is never performed. At least one of the measurements must be active – they cannot both be in standby mode.

If both measurements are active at the same time, the sensor will perform readings using each definition in alternation. So, measurement 0 will run to completion for one measurement cycle, then measurement 1 will be used during the next cycle, then back to 0 for the next, and so on.

Activating a Measurement

A measurement can be put in active mode by calling *ch_meas_activate()*. The measurement definition will be used when the sensor makes a measurement. The measurement must have previously been defined, either by using *ch_meas_init()* and related calls to specify the measurement segments, or by importing a complete measurement definition. (Note that measurements created using *ch_meas_init()* are active by default but can be specified to start in standby mode.)

If the measurement was previously in standby mode (inactive), it is made active. In all cases, this measurement's definition will be used during the next measurement performed by the sensor, even if the other measurement is also active.

ch_meas_activate() does not affect the active vs. standby status of the other measurement. If both measurements are active after calling this function, they will alternate each time the sensor performs a measurement.

Putting a Measurement in Standby Mode

The *ch_meas_standby()* function places the specified measurement in standby mode. The measurement's definition will not be used by the sensor, and the other measurement definition will be used to perform all readings.

Because there must always be at least one active measurement, this function will activate the other measurement if it was not already active. It is not possible to have both measurements in standby mode simultaneously.

Switching Active and Standby Measurements

The active and standby measurements can be switched on demand by the application, by using the *ch_meas_switch()* function.

The *ch_meas_switch()* function exchanges the mode of the two measurement definitions between active and standby. The currently active measurement is put into standby mode, and the standby measurement is made active.

If this function is called when both measurements are active, the next measurement due to operate (the current measurement) will be placed into standby mode, and the other measurement will remain active. So, this routine can be used to force a specific active measurement to be executed next.

The return value from *ch_meas_switch()* is the number of the next measurement that will be performed by the sensor.

> ***Note:*** Switching the measurement definition only affects the transmit and receive characteristics of the sensor. It does NOT modify the target detection thresholds. You may need to also call *ch_set_thresholds()* to change the detection thresholds when you switch measurements, depending on the differences between your two measurement definitions.

Measurement-Specific Configuration Settings

Some basic SonicLib configuration functions only operate on measurement 0 (*CH_MEAS_DEFAULT*). A separate set of API calls allow the equivalent settings for either measurement specifically. These typically have similar names beginning with "*ch_meas_*"or "icu_gpt_" and use the same parameters as the basic routines, with the addition of the measurement number.

Table 7 lists the regular API functions and their measurement-specific equivalents.

Regular Function	Measurement-Specific Function
ch_get_freerun_interval()	*ch_meas_get_interval()*
ch_set_freerun_interval()	*ch_meas_set_interval()*
ch_get_freerun_interval_ticks()	*ch_meas_get_interval_ticks()*
ch_set_freerun_interval_ticks()	*ch_meas_set_interval_ticks()*
ch_get_freerun_interval_us()	*ch_meas_get_interval_us()*
ch_set_freerun_interval_us()	*ch_meas_set_interval_us()*
ch_get_max_range()	*ch_meas_get_max_range()*
ch_set_max_range()	*ch_meas_set_max_range()*
ch_get_num_samples()	*ch_meas_get_num_samples()*
ch_set_num_samples()	*ch_meas_set_num_samples()*
ch_get_rx_holdoff()	*icu_gpt_get_rx_holdoff()*
ch_set_rx_holdoff()	*icu_gpt_set_rx_holdoff()*
ch_get_static_range()	*icu_gpt_get_static_filter()*
ch_set_static_range()	*Icu_gpt_set_static_filter()*
ch_get_thresholds()	*icu_gpt_get_thresholds()*
ch_set_thresholds()	*Icu_gpt_set_thresholds()*

Table 7 - Measurement-Specific Functions

5.8 Transmit Optimization

ICU sensors can optimize the measurement transmit sequence to improve performance at close distances. Optimization modifies the measurement queue definition to actively dampen the natural ringdown of the ultrasound transducer after a transmit pulse is sent.

A measurement must be fully configured (including adding all transmit and receive segments and writing to the sensor) before transmit optimization can take place. A measurement definition may be optimized whether it was created using SonicLib API functions or was imported (see Importing a Measurement Definition, below). The *ch_meas_optimize()* function is used to perform the optimization in both cases.

During optimization, extra transmit segments will automatically be inserted into the existing measurement queue sequence, after the original user-defined transmit segment(s). These short transmit segments contain specific transducer instructions that counteract the ringdown effect. These instructions are dynamically tuned for each device during *ch_meas_optimize()*. The original receive and count segments from the measurement definition will follow the new, inserted transmit segments.

When optimization occurs, the sensor is briefly re-programmed with special initialization firmware that analyzes the sensor's operation and determines the appropriate optimization. The results are read from the sensor and are combined with the original measurement settings. Finally, the measurement firmware is again programmed into the sensor, and the new settings are applied. This entire sequence will typically take about 250 ms to complete.

Note: A single set of measurement definition values should only be optimized once, because new transmit segments will be inserted each time. If your application needs to perform optimization periodically, it should maintain a copy of the original measurement queue values (before optimization) and pass that in again each time this function is called. A copy of the internal **measurement_queue_t** structure can be obtained using *ch_meas_get_queue()*.

5.9 Importing a Measurement Definition

An application may import a complete measurement definition instead of constructing one using individual API functions, using the *ch_meas_import()* function:

```
uint8_t ch_meas_import(ch_dev_t *dev_ptr, measurement_queue_t *meas_queue_ptr, void *algo_cfg_ptr)
```

This function allows a fully defined measurement queue structure (**measurement_queue_t**) and algorithm-specific configuration to be imported as a unit. Taken together, these two structures fully define the measurements to be performed.

The **measurement_queue_t** structure describes both measurements (0 and 1), so they are always imported together. It includes all segments for both measurements. As always, use of a second measurement is optional. If the second measurement is unused, those fields in the **measurement_queue_t** structure will contain null values.

Different sensor firmware types contain different algorithms for processing data, which are separate from the measurement mechanism. This distinction is supported in *ch_meas_import()* by a second parameter to supply the algorithm configuration values.

Different algorithms will require different configuration controls and will use different configuration structures. Therefore, a **void *** pointer type is used to identify the structure.

If you export a measurement from the ICU-x0201 EVK tool, it will contain separate structure definitions for the measurement queue and the algorithm configuration. Both structures should be specified in the call to *ch_meas_import()*.

Alternatively, it is possible to import either the measurement queue or the algorithm configuration individually, by supplying a NULL pointer for the other structure. In this case, the current settings will continue to be used for the values specified in the other structure. An error is returned if both pointers are NULL.

Optimization During Import

A measurement configuration may be imported, applied, and optimized in one operation by using the *ch_meas_optimize()* function instead of *ch_meas_import()*:

```
uint8_t ch_meas_optimize(ch_dev_t *dev_ptr, measurement_queue_t *meas_queue_ptr, void *algo_cfg_ptr)
```

In this case, the measurement queue specified by *meas_queue_ptr* will be modified during the import – new transmit segments will be added to improve close-range performance. The modified settings will be written to the sensor. The result is the same as if the measurement were imported using *ch_meas_import()* and then separately optimized using *ch_meas_optimize()*. See Transmit Optimization.

The original input measurement queue definition specified by *meas_queue_ptr* will not be modified, so it can be re-used.

The *algo_cfg_ptr* parameter specifies a new measurement algorithm configuration to be applied. If *algo_cfg_ptr* is NULL, the current configuration will be left unchanged.

6 CREATING AN APPLICATION

This section describes how to design an application to use SonicLib for ultrasonic sensing.

6.1 "Hello Chirp" Example

The "Hello Chirp" example applications, available from InvenSense, demonstrate how to use SonicLib to control one or more ultrasonic sensors. The example source code contains extensive comments explaining the steps that the application takes in initializing, configuring, and running the sensors as described here.

You may want to refer to that example project while reading this document, or for additional guidance in creating your own application.

> ***Note:*** ICU and CHx01 sensors use slightly different Hello Chirp example applications, so that ICU-specific features may be demonstrated separately.

6.2 Overall Application Flow

Although every application is different, a program using SonicLib generally has a standard sequence of actions:

1. Initialize hardware: *bsp_init()*
2. Initialize Soniclib structures for sensor's group: *ch_group_init()*
3. Initialize SonicLib structures for each sensor: *ch_init()*
4. Define a specific init firmware (if needed) : *ch_set_init_firmware()*
5. Program and start all sensors: *ch_group_start()*
6. Register callback routine(s) – see below
7. Configure sensors (ultrasound measure and algorithm) : *ch_set_range()*, *ch_set_thresholds()*, etc., followed by *ch_set_mode()*
    a. Start triggering mechanism based on h/w timer, etc. if sensor(s) will not in operate in *CH_MODE_FREERUN*.
8. Enter endless loop to perform sensing:
    a. Sensor interrupt callback function is called by SonicLib/BSP when data ready interrupt occurs.
    b. Get measurement results and data: *ch_get_range()* etc.
    c. Application handles measurement result as needed.

Steps 1 through 4 are discussed below in Initialization. Step 5 is described in Configuration. Step 6 was discussed previously in Sensing.

6.3 Callback Routines

Although SonicLib interfaces are mostly functions that are called by an application, it is sometimes necessary for a routine within the application to be called by SonicLib or the board support package, to notify the application that a certain event has occurred.

Sensor Interrupt Callback

Normally, every application will include a callback function tied to the sensor's interrupt, which is used to signal data-ready at the end of a measurement. The *ch_io_int_callback_set()* function is used to register the callback function to be called by SonicLib when the interrupt occurs.

When the sensor asserts the INT line, an interrupt occurs. The interrupt handler in the BSP notifies SonicLib using the *ch_interrupt()* function. Then the callback function is called at interrupt level. The specific type of sensor interrupt is passed to the callback routine as a parameter.

In most applications, the response to this interrupt is to read data from the sensor, using *ch_get_range()* and other SonicLib functions, often by setting a flag in the callback function to allow the readout to occur at normal task level.

Non-Blocking I/O Complete Callback

Another callback function is only required if your application will read the I/Q or amplitude data from a sensor in non-blocking mode. The function is called to notify the application when a non-blocking read completes. The *ch_io_complete_callback_set()* function is used to register the callback function.

When the non-blocking read completes, an interrupt occurs, and the callback function is called at interrupt level. In most applications, the response to this interrupt is to process the data that was read, either in the callback function itself or by setting a flag to allow processing at normal task level.

Periodic Timer Callback

In addition to the callback interfaces used by SonicLib directly, the board support package (BSP) definition in **chirp_bsp.h** also specifies an optional periodic timer mechanism that uses a callback.

The callback routine is called when a periodic timer (as defined in the BSP) expires. The callback is registered as part of the *chbsp_periodic_timer_init()* function.

A periodic timer callback is often used to trigger a measurement cycle. The periodic timer callback is not required to use the SonicLib API, but it is used in the Hello Chirp application and other example programs from InvenSense.

7 INITIALIZATION

Before an application can perform its main operations, it must initialize the hardware and software in the system. This section describes the steps to initialize a SonicLib application.

7.1 Hardware Initialization – *bsp_init()*

This function executes the required hardware initialization sequence for the board being used. This includes clock, memory, and processor setup as well as any special handling that is needed. This function is called at the beginning of an application, as the first operation.

This function is defined and implemented by the user, it shall initialize the hardware functions of the board (I/O, interrupts, GPIO pins). It does not perform any discovery or initialization of the ultrasonic sensors. Those operations are done during *ch_group_start()*, see below.

7.2 SonicLib Group Software Initialization – *ch_group_init()*

This function initializes several fields within the **ch_group_t** sensor group descriptor, as shown in Table 8

Field	Description
num_ports	Number of possible sensor ports on the board Usually the same as *CHIRP_MAX_DEVICES* in **chirp_board_config.h**. Accessible later using *ch_get_num_ports().*
num_buses	Number of SPI or I^2C buses used by sensors on the board. Usually the same as *CHIRP_NUM_BUSES* in **chirp_board_config.h**.
rtc_cal_pulse_ms	Length (duration) of the pulse sequence sent on the INT line to each sensor during calibration of the real-time clock, in milliseconds. Not required if using alternate RTC calibration – see Real Time Clock (RTC). Accessible later using *ch_get_rtc_cal_pulselength().*

Table 8 - ch_group_t Fields Set by ch_group_init()

7.3 SonicLib Device Software Initialization – *ch_init()*

This function is used to initialize various SonicLib structures before using a sensor. The **ch_dev_t** device descriptor is the primary data structure used to manage a sensor, and its address will subsequently be used as a handle to identify the sensor when calling most API functions.

The *ch_init()* function has the following definition:

```
uint8_t ch_init (ch_dev_t *dev_ptr, ch_group_t *grp_ptr, uint8_t dev_num,
                 ch_fw_init_func_t fw_init_func)
```

The *dev_ptr* parameter is the address of the **ch_dev_t** descriptor structure that will be initialized and then used to identify and manage this sensor. The *grp_ptr* parameter is the address of a **ch_group_t** structure describing the sensor group that will include the new sensor. Both the **ch_dev_t** structure and the **ch_group_t** structure must have already been allocated before this function is called.

dev_num is a simple index value that uniquely identifies a sensor within a group. Each possible sensor (i.e., each physical port on the board that could have an ultrasonic sensor attached) has a number, starting with zero (0). The device number is constant - it remains associated with a specific port even if no sensor is actually attached. Often, the *dev_num* value is used by an application as an index into arrays containing per-sensor information (e.g., data read from the sensors).

Specifying the Sensor Firmware

InvenSense ultrasonic sensors are fully re-programmable, and the specific features and capabilities can be modified by using different sensor firmware images. The firmware to be used is selected during the *ch_init()* call, by specifying a sensor firmware initialization routine.

The *fw_init_func* parameter is the address (name) of the sensor firmware initialization routine that will be used to program the sensor with the appropriate firmware image and prepare it for operation. The selection of this routine name is the only required change in the application when switching from one sensor firmware image to another.

The *ch_init()* function only performs internal initialization of data structures, etc. It does not actually interact with the physical sensor device(s). That process is described in *ch_group_start()*.

7.4 Init Firmware Initialization – *ch_set_init_firmware()*

The sensor firmware images contain code to initialize the sensor's hardware and other features like code to compute algorithm on ultrasound data. Sometimes a specific feature is needed and not present in main firmware; in this case you have to define an init firmware which will be flashed on the sensor to run initialization routines, such as finding the device resonant frequency or computing the optimal reverse drive transmit instructions.

The *ch_init_firmware()* function has the following definition:

```
uint8_t ch_init_firmware (ch_dev_t *dev_ptr, ch_fw_init_func_t fw_init_func)
```

The *fw_init_func* parameter is the address (name) of the sensor firmware initialization routine that will be used to program the sensor with the appropriate firmware image and prepare it for operation.

The *ch_init_firmware()* function only performs internal initialization of data structures, etc. It does not actually interact with the physical sensor device(s). That process is described in the next section.

7.5 Sensor Initialization – *ch_group_start()*

This function performs the actual discovery, programming, and initialization sequence for all sensors within a sensor group. Each sensor must have previously been added to the group by calling *ch_init()*.

The *ch_group_start()* function has the following definition:

```
uint8_t ch_group_start (ch_group_t *grp_ptr)
```

In brief, this function does the following for each sensor:

1. Probe the possible sensor ports using SPI (for ICU sensors) or the I^2C bus combined with each sensor's PROG line (for CH101 and CH201 sensors), to discover if the sensor is connected.
2. Reset each sensor and put it in a known state.
3. Program each sensor with firmware (specific firmware type was specified during *ch_init()*).
    a. For CH101/CH201 sensors, assign a unique I^2C address to sensor (specified by board support package, see *chbsp_i2c_get_info()*).
4. Start sensor execution.
5. Wait for sensor to "lock" (complete initialization, including self-test).
6. Send timed pulse sequence on INT line to calibrate sensor Real-Time Clock (RTC), depending on RTC configuration (see Real Time Clock (RTC)).

After this routine returns successfully, the sensor configuration may be set, as described in the next section.

7.6 Registering Callback Routines

As discussed earlier, the SonicLib API defines mechanisms for an application routine to be called by SonicLib or the board support package to notify the application that an event has occurred. In SonicLib, callback routines can notify the application that data is ready from the device, or that non-blocking I/O has completed. In addition, the board support package interfaces defined in **chirp_bsp.h** include a callback mechanism for periodic timers.

These callback routines are typically called at interrupt level. To avoid timing and latency issues, you should try to minimize the amount of processing done directly in the callback routines. For example, you may want to set flag variables or otherwise arrange for additional handling to be done in the regular application loop at task level.

Sensor I/O Interrupt Callback

In general, all applications use a callback routine tied to the "data-ready" interrupt generated by the sensor asserting its INT line to indicate that a measurement cycle is complete (data-ready). Generally, this callback routine will initiate the reading of measurement data from the sensor, using *ch_get_range()* etc.

The I/O interrupt callback routine must follow the following format:

```
void io_int_callback_name (ch_group_t *grp_ptr, uint8_t io_index, ch_interrupt_type_t int_type)
```

The *io_int_callback_name* is the name of the callback routine in your application. The name may be anything you choose. The *grp_ptr* parameter is a pointer to the sensor group structure for the interrupting device. The *io_index* parameter is the device index number within the sensor group. The int_type parameter specifies what kind of interrupt the sensor generated.

Together, the *grp_ptr* and *io_index* parameters uniquely identify the interrupting device. The address of the corresponding device descriptor (**ch_dev_t** structure) can be determined by passing these values to the *ch_get_dev_ptr()* function.

The I/O interrupt callback is registered by calling the *ch_io_int_callback_set()* function, which is defined as follows:

```
void ch_io_int_callback_set (ch_group_t *grp_ptr, ch_io_int_callback_t callback_func_ptr)
```

The *grp_ptr* parameter is a pointer to the sensor group structure for the interrupting device. The *callback_func_ptr* parameter is the address (name) of your callback routine.

Periodic Timer Callback

Although not part of the SonicLib API, the board support package definition in **chirp_bsp.h** specifies a callback mechanism tied to a periodic timer on the board. If you are using a BSP with periodic timer support, you may use the *chbsp_periodic_timer_init()* function to initialize a timer, which both sets the timer interval and registers the callback routine.

The periodic timer callback routine has the following format:

```
void periodic_timer_callback_name(void)
```

The *periodic_timer_callback_name* is the name of the callback routine in your application. The name may be anything you choose.

The periodic timer callback function is registered at the same time the periodic timer is initialized with its countdown period. The *chbsp_periodic_timer_init ()* function is defined as follows:

```
uint8_t chbsp_periodic_timer_init (uint16_t interval_ms, ch_timer_callback_t callback_func_ptr)
```

The *interval_ms* parameter is the period for the timer – the timer will expire (interrupt) every *interval_ms* milliseconds. The *callback_func_ptr* parameter is the address (name) of your callback routine.

Typically, this callback routine will initiate a new measurement cycle by calling *ch_trigger()* or *ch_group_trigger()*. The periodic timer in the BSP therefore controls the overall measurement timing for the application.

Data I/O Complete Callback

An additional callback routine type is only used in applications that use non-blocking reads of sample data from the sensor (see Non-blocking Read Mode (optional). These applications must have an extra mechanism so they can be notified when the sample data read is complete.

When the non-blocking read completes, it causes an interrupt. The interrupt handler in the BSP will use the *ch_io_notify()* routine to notify SonicLib, which will in turn call the application's I/O complete callback routine (still at interrupt level). The sample I/Q or amplitude data may then be read from the the buffer that was specified during *ch_get_iq_data()* or *ch_get_amplitude_data()*.

The I/O complete callback routine must follow the following format:

```
void io_complete_callback_name (ch_group_t *grp_ptr)
```

The *io_complete_callback_name* is the name of the callback routine in your application. The name may be anything you choose. The *grp_ptr* parameter is a pointer to the sensor group structure for the device.

The I/O complete callback is registered by calling the *ch_io_complete_callback_set()* function, which is defined as follows:

```
void ch_io_complete_callback_set (ch_group_t *grp_ptr, ch_io_complete_callback_t callback_func_ptr)
```

The *grp_ptr* parameter is a pointer to the sensor group structure for the device whose I/Q data is being read. The *callback_func_ptr* parameter is the address (name) of your callback routine.

7.7 Real Time Clock (RTC)

The sensor's internal timing is controlled by an on-chip real time clock (RTC). For accurate time of flight measurements, the frequency of this clock must be known as precisely as possible.

Typically, the RTC frequency is measured during a hardware calibration step during initialization. Performing calibration is recommended, because it gives the best overall sensor accuracy when using the internal clock. However, it may be more convenient and sufficiently accurate to use an alternate method. This section describes normal RTC calibration as well the *ch_set_rtc()* function, which provides different RTC options:

```
uint8_t ch_set_rtc(ch_dev_t *dev_ptr, ch_rtc_src_t rtc_source, uint16_t rtc_freq)
```

When used, *ch_set_rtc()* must be called after *ch_init()* and before *ch_group_start()*.

Available RTC options will vary with sensor model and firmware type.

RTC Clock Calibration

By default, the RTC frequency is determined by calibrating the clock during initialization, by applying an externally timed signal via the sensor INT pin while the clock runs. For ICU sensors, the calibration signal consists of two short pulses (active low) on the INT line, separated by a known delay, typically 100 ms. For CHx01 sensors, the calibration signal is a single long pulse (active high) of a known length, again typically 100 ms.

The pulses are generated automatically by SonicLib during the *ch_group_start()* function, by calling functions supplied in the board support package (BSP) to manipulate the INT line. The length of the calibration delay ("pulse length") is specified in the BSP by setting the **rtc_cal_pulse_ms** field in the group descriptor (**ch_group_t**) during *chbsp_board_init()*.

To perform normal RTC calibration using a timed pulse, no special action is required. *The ch_set_rtc()* function is not needed and should not be called.

Using the Factory RTC Calibration (ICU sensors only)

ICU sensors contain factory test data that includes a measurement of the sensor's internal RTC frequency. This value provides a reasonable substitute for performing live calibration using a timed pulse. Factory RTC calibration values are not available in CH101 or CH201 sensors.

To use the factory RTC calibration data when calling *ch_set_rtc()*, set *rtc_source* to *CH_RTC_SRC_INTERNAL*, and set *rtc_freq* to zero (0).

Normal calibration using a timed pulse is not performed when using the factory calibration data.

Using the I/O Bus Clock to Calibrate RTC Frequency

In some cases, it is possible to use the I/O bus clock as a reference for calibrating the RTC. This feature is only available in select CHx01 sensor f/w variants.

To use the bus clock for calibration, set *rtc_source* to *CH_RTC_SRC_INTERNAL*, and set *rtc_freq* to *CH_RTC_USE_BUS_SPEED*.

For CHx01 sensor types that support this calibration mode, the I^2C bus clock (SCL) speed is assumed to be 400000 Hz by default. If a different, more accurate value for the bus clock speed is available, it can be specified by defining *CHIRP_I2C_BUS_SPEED_HZ* in the **chirp_board_config.h** file in the board support package and setting the value to the correct bus speed. Inaccuracies in the bus clock rate value will affect frequency and range measurements by the sensor.

Normal calibration using a timed pulse is not performed when using the bus speed for calibration.

Supplying an Estimated RTC Frequency

If no other method is available, the RTC frequency can be estimated. If *rtc_freq* does not equal zero or *CH_RTC_USE_BUS_SPEED*, the value will be used in all calculations requiring the RTC frequency. The *rtc_freq* value should be the best available estimate of the RTC frequency for this device.

To use an estimated RTC frequency, set *rtc_source* to *CH_RTC_SRC_INTERNAL*, and set *rtc_freq* to the approximate RTC frequency, in Hz.

If no other estimate is available, *rtc_freq* may be specified as *CH_RTC_FREQ_TYPICAL* (29000 Hz) for a rough approximation of a typical RTC frequency. This option is a last choice, due to a significant impact on measurement accuracy, but it will allow measurements to be completed.

Normal calibration using a timed pulse is not performed when using an estimated RTC frequency.

Using an External Clock Source

The *ch_set_rtc()* function can also set the source of the sensor's real-time clock. By default, the sensor will use its internal oscillator as the RTC clock time base. To use the internal RTC, no special action is required, and this routine does not have to be called unless you are changing the calibration values described above.

For enhanced timing accuracy, an external clock source (e.g., a crystal oscillator) may be connected to the ICU sensor. This technique may be useful to maintain a very exact measurement interval for sensors operating in free-running mode (*CH_MODE_FREERUN*).

To enable an external clock source for the RTC, set *rtc_source* to *CH_RTC_SRC_EXTERNAL*, and set *rtc_freq* to the frequency of the input clock signal in Hz (e.g., 32768 for a 32.768 kHz crystal).

Normal calibration using a timed pulse is not performed when using an external clock source.

7.8 Adjusting the Operating Frequency

Each individual sensor's ultrasonic transducer has a natural resonant frequency. Normally, this natural frequency is used as the acoustic operating frequency for the ultrasound that is sent and received. In single sensor applications, this will give the best performance.

However, when multiple sensors are used with one transmitting and others receiving, see Multiple Sensors (Pitch – Catch), the efficiency of the transfer can be enhanced by setting the sensors to the same acoustic operating frequency. The *ch_group_set_frequency()* function sets all sensors in a sensor group as close as possible to the same frequency:

```
uint8_t ch_group_set_frequency(ch_group_t *grp_ptr, uint32_t request_op_freq_hz)
```

If the *request_op_freq_hz* parameter is *CH_OP_FREQ_USE_AVG* (zero), all sensors will be set to the average of the natural operating frequencies for the group, which generally gives the best performance. After this call returns, you may use *ch_group_get_frequency()* to determine the average frequency that was used.

For special situations, the operating frequency for an individual device may be set, using the *ch_set_frequency()* function:

```
uint8_t ch_set_frequency(ch_dev_t *dev_ptr, uint32_t request_op_freq_hz)
```

The frequency must be within the typical operating range for the sensor model. It is not possible to radically shift the operating frequency.

The final operating frequency of each sensor will vary slightly from the requested value, due to internal timing granularity. The *ch_get_frequency()* function may be used to determine the final frequency for an individual device.

> ***Note:*** If used, the calls to *ch_group_set_frequency()* or *ch_set_frequency()* should generally be placed after the call to *ch_group_start()*, before other configuration is performed.

8 CONFIGURATION

After a sensor has been initialized, it must be configured to operate with the specific settings required by the application. These settings include the overall operating mode for the sensor, the maximum range it will measure, internal sample interval (for devices in free-running mode), static target rejection, and object detection thresholds. (Note not all features are available on all devices or sensor firmware versions.)

The following sections discuss the different configuration settings that are available. In general, these should be set immediately after the sensor has been initialized and started using *ch_init()* and *ch_group_start()*.

The configuration settings should be in place before beginning sensing. Sensing is enabled when the sensor mode changes out of *CH_MODE_IDLE*, so the *ch_set_mode()* call should normally be last in the overall configuration sequence. See Sensor Operating Mode.

8.1 Measurement Configuration (ICU sensors)

As described earlier in Measurement Controls (ICU sensors), ICU sensors have fully controllable measurement characteristics, including both transmit and receive parameters.

Many of the configuration functions described below will affect the measurement configuration. For example, the measurement configuration specifies the number of receive samples, which determines the maximum detection range. However, the high-level *ch_set_max_range()* function can later modify the overall sample count, as discussed below. Other configuration settings can similarly modify transmit or receive segment values.

Therefore, you should define the measurement before other configuration settings are applied, by using the SonicLib API or by importing an externally defined measurement.

8.2 Maximum Range Setting

The ultrasonic sensor has a configurable full-scale range (FSR), meaning that the application may set the maximum distance at which the sensor will detect an object. Any value up to the rated maximum range for the device may be specified.

The maximum range may be set using the *ch_set_max_range()* function. (Alternatively, the maximum range may instead be set using the *ch_set_config()* function, which also sets other configuration values.)

The *ch_set_max_range()* function has the following definition:

```
uint8_t ch_set_max_range (ch_dev_t *dev_ptr, uint16_t max_range)
```

The *max_range* value is the one-way distance to a detected object, in millimeters.

The maximum range setting determines the overall length of the listening period, and therefore the number of samples that will be made during the measurement.

> ***Note:*** In ICU sensors, the *ch_set_max_range()* function will modify the number of samples in the existing measurement definition. If the range is being reduced compared to the original measurement definition, the number of receive samples in the final receive segment(s) will be reduced accordingly, including complete removal of final segments if needed. If the maximum range is being increased, the final receive segment will be modified to include more samples, using the same receive settings. See Receive Segments.

8.3 Internal Sample Interval (Free-running mode only)

When a sensor is used in free-running mode (*CH_MODE_FREERUN*), an internal timer is used to wake up the sensor to perform a measurement. The interval between these measurements may be set using the *ch_set_freerun_interval()* function.

The *ch_set_freerun_interval()* function has the following definition:

```
uint8_t ch_set_freerun_interval (ch_dev_t *dev_ptr, uint16_t interval_ms)
```

The sensor will use its internal clock to wake and perform a measurement every *interval_ms* milliseconds. Similar functions allow the interval to instead be set in units of micro-seconds or RTC timer ticks, if more convenient in your application.

> ***Note:*** It is extremely important that the freerun interval be set before sensing is enabled in *CH_MODE_FREERUN*. If sensing starts before the interval has been set, the behavior is undefined.

The freerun interval setting has no effect when using any of the triggered modes.

8.4 Static Target Rejection Settings

STR is enabled using the *ch_set_static_range()* function. (Alternatively, the static rejection range may instead be set using the *ch_set_config()* convenience function, which specifies additional configuration values.)

The *ch_set_static_range()* function has the following definition:

```
uint8_t ch_set_static_range (ch_dev_t *dev_ptr, uint16_t num_samples)
```

The STR filtering always begins from the first sample in the measurement. The *num_samples* parameter specifies the portion of the measurement trace that will be filtered for static targets. It is expressed in samples (unlike the *ch_set_max_range()* function, which uses millimeters). You may want to use the *ch_mm_to_samples()* function to determine the appropriate number of samples for a specific distance range. A value of zero for *num_samples* will disable STR.

ICU sensors can enable STR filtering as part of the measurement definition. See Initializing a Measurement.

To enable STR filtering for the entire measurement, set *num_samples* to at least the total number of active samples, as returned by *ch_get_num_samples()*. The maximum sample count for the sensor and firmware can be used, using either a symbolic constant from **soniclib.h** (e.g., *ICU_MAX_NUM_SAMPLES* for an ICU sensor with **icu_gpt** firmware) or the return value from *ch_get_max_num_samples()*.

It may be necessary to lower the detection threshold levels in the sample ranges where STR is used to get the desired sensitivity (see Multiple Detection Thresholds, below).

Enabling STR has a negligible impact on power consumption but does require additional processing time by the sensor, resulting in a slight increase in the time to report data-ready, up to 3ms for a sensor at its maximum range setting.

8.5 Multiple Detection Thresholds

The multiple detection threshold controls allow blocks of contiguous samples in the measurement (i.e., different ranges of distance from the sensor) to use different comparison thresholds. This allows both close and far objects to be detected more consistently. Typically, threshold values will steadily decrease across the range of samples (so farther objects are more easily detected).

In some special circumstances, thresholds may be used to selectively tune the sensitivity of the sensor within a specific portion of the detection range. This can define a "sweet spot" range of easier detection (by reducing the threshold for a range of samples) or can be used to ignore targets within a certain distance range (by increasing the threshold).

When two sensors are used in a pitch-catch configuration, the direct path signal strength is usually greater than a reflected signal. So, the receiving sensor can often use higher threshold values and still maintain good detection.

Each threshold is specified with a starting sample number and the amplitude value required for detection. The threshold value will apply to all samples from the starting sample number until the start of the next threshold (if any).

Number of Thresholds

In ICU sensors using the standard **icu_gpt** firmware, up to eight (8) threshold ranges may be defined within each measurement.

CH101 and CH201 sensor support for multiple thresholds varies based on the specific sensor firmware being used. In the **ch101_gprmt** and **ch201_gprmt** firmware, up to six (6) thresholds may be set. Other CH101 and CH201 sensor firmware types do not support standard multiple thresholds but may provide some limited threshold control – see the information for the specific sensor firmware being used.

You do not have to use the entire set of thresholds if your application does not require that many different values. The last valid threshold entry will apply to the remainder of the measurement samples. Unused thresholds entries should set with the starting sample number to zero.

To allocate space for results or other needs, the *CH_NUM_THRESHOLDS* symbol may be used. It equals the maximum number of thresholds supported by the current sensor firmware.

Defining Thresholds

Each threshold is described in a **ch_thresh_t** structure, which has two fields: a starting sample number, and the detection threshold (amplitude value) required for an object to be detected.

The first threshold must always begin with sample zero, and each detection threshold may only extend for a maximum of 255 samples. (You may have two consecutive thresholds with the same level, if the 255 sample limit is too restrictive.)

The individual thresholds are then collected in an array structure, **ch_thresholds_t**. To set the thresholds within the sensor, the address of this array is passed to the *ch_set_thresholds()* function, which has the following definition:

```
uint8_t ch_set_thresholds (ch_dev_t *dev_ptr, ch_thresholds_t *thresh_ptr)
```

8.6 Interrupt & Triggering Configuration (ICU sensors only)

Interrupt & Trigger Pin Selection

ICU sensors have two pins, **INT1** and **INT2**, that can be configured for the sensor to interrupt the host and/or for the host to trigger the sensor. The two functions may be assigned to the same pin, or one pin may be used for triggering and the other pin used for sensor interrupts.

Two pre-processor symbols are used to specify the pin assignments. These are typically defined in the **chirp_board_config.h** header file provided by the board support package (BSP).

- **CHIRP_SENSOR_INT_PIN** – specifies the pin used for interrupts from the sensor to the host processor (0 or 1)
- **CHIRP_SENSOR_TRIG_PIN** – specifies the pin used by the host processor to trigger the sensor (0 or 1)

Latching vs. Pulse Interrupt

The *ch_set_interrupt_mode()* function sets the ICU sensor interrupt to use a pulsed or latching level change. In pulse mode, the sensor will briefly change the INT line level and then restore it to the original state. In latching mode, the sensor will change the INT line level, and it will stay at the active (low) level until reset by the interrupt handler in the board support package. By default, ICU sensors use latching interrupt mode (*CH_INTERRUPT_MODE_LATCH*).

```
uint8_t ch_set_interrupt_mode(ch_dev_t *dev_ptr, ch_interrupt_mode_t mode)
```

To use this function, set *mode* to *CH_INTERRUPT_MODE_PULSE* to enable pulsed interrupt mode. Set *mode* to *CH_INTERRUPT_MODE_LATCH* to disable the pulse interrupt mode and use latching behavior.

This option is only available for ICU sensors. CHx01 sensors always use pulse interrupt mode (active high) for normal sensor interrupts.

Soft Triggering

Instead of normal hardware triggering, ICU sensors may also be "soft triggered" via the SPI bus.

When soft triggering will be used exclusively, the default sensor trigger type may be changed. The *ch_set_trigger_type()* function may be used to set the trigger type to *CH_TRIGGER_TYPE_SW*. After the trigger type is changed, all calls to *ch_trigger()* will use soft triggering instead of the normal hardware triggering.

The *ch_trigger_soft()* function may also be used to initiate a measurement, even if the sensor is otherwise using hardware triggering. This may be useful to perform an extra measurement outside of the normal trigger schedule.

Soft triggering is not recommended for multi-sensor operation, due to timing lag and uncertainty when triggering over the SPI bus.

8.7 Sensor Operating Mode

As described earlier in Sensor Operating Modes, the sensor is initially in idle mode (*CH_MODE_IDLE*) when it is first initialized. No sensing takes place, so the sensor can be configured. The last step in configuration is to change the mode from CH_MODE_IDLE to one of the other operating modes (*CH_MODE_FREERUN*, *CH_MODE_TX_RX*, or *CH_MODE_RX_ONLY*) to begin sensing.

The *ch_set_mode()* function has the following definition:

```
uint8_t ch_set_mode (ch_dev_t *dev_ptr, ch_mode_t mode)
```

CH_MODE_FREERUN – Free-Running (Self-Timed) Transmit/Receive Mode

When the sensor is in free-running mode, it uses a periodic timer based on the sensor's internal real-time clock (RTC) to control the timing of measurements. The timer is set to a specific delay corresponding to the sensing interval. When the timer expires, the sensor will wake up and begin an ultrasonic range measurement. When the measurement is complete, the sensor will notify the remote host device by asserting the INT line.

Free-running mode may only be used by individual sensors operating independently – multi-sensor configurations must use the triggered modes described below.

Once calibrated, the internal RTC used in free-running mode provides good accuracy, but it is not as stable as a crystal-controlled oscillator typically found on a microcontroller board. Therefore, hardware-triggered mode (see next section) should be used for critical timing applications. ICU sensors also allow the RTC clock to be driven by an external crystal.

CH_MODE_TRIGGERED_TX_RX – Hardware-Triggered Transmit/Receive Mode

In many applications, the ultrasonic measurements require more exact timing than the sensor's internal RTC provides in free-running mode, or the sensor operation needs to be coordinated with other application activities. In these cases, the sensor's measurement cycle can be initiated by using a hardware trigger, in which the remote host device asserts and then releases the INT line. When the sensor detects that the INT line has been asserted, it will begin a measurement cycle.

The most typical mode for a single sensor is hardware-triggered transmit/receive (Tx/Rx). In this mode, the sensor will generate an ultrasonic pulse when it is triggered by the INT line from the host. The sensor then listens for a response (echo) for an amount of time based on the maximum range setting of the device. When the measurement cycle is complete, the sensor will notify the host by asserting the INT line. Note that the INT line typically operates in two directions when used in hardware-triggered mode – first as an input to the sensor (output from host) to initiate the measurement, and then as an output from the sensor (input to host) for the measurement-complete notification. ICU sensors have configurable INT and trigger pin assignments, see Interrupt & Triggering Configuration (ICU sensors only).

Generally, the application will repeatedly trigger the sensor based on a periodic timer that can maintain an accurate sensing interval. Conversely, the application may wait until specific conditions are met, then initiate a single isolated measurement.

CH_MODE_TRIGGERED_ RX_ONLY – Hardware-Triggered Receive-Only Mode

When more than one ultrasonic sensor is used, they may be configured so that one device operates in hardware-triggered Tx/Rx mode as described above, and one or more other sensors operate in hardware-triggered receive-only mode (Rx-only). In this case, all sensors are triggered by the remote host together via their INT lines. The single Tx/Rx node generates an ultrasonic pulse and listens for an echo as normal. All Rx-only nodes will simultaneously begin their own listening periods, but without sending an ultrasonic pulse. Instead, the Rx-only sensors simply wait to detect the pulse that was sent from the Tx/Rx sensor (either directly, or as an echo off another object).

When each sensor completes its measurement cycle, it will notify the remote host by asserting its INT line.

Receive Pre-triggering

As described earlier in Ringdown Filtering, a sensor in *CH_MODE_RX_ONLY* will have a small ringdown artifact due to the powering-on of the ultrasound transducer. This false signal can affect the ability of the sensor to receive the ultrasound pulse from the transmitting sensor if the distance is very short, because the pulse arrives while the ringdown is still occurring.

To improve performance at close distances, the receiving sensor can be "pre-triggered," meaning that it is triggered slightly before the transmitting sensor. This allows the *CH_MODE_RX_ONLY* sensor to fully settle before the pulse arrives.

Pre-triggering is enabled by the *ch_set_rx_pretrigger()* function. When enabled, all receive-only devices in the sensor group will use pre-triggering. The receive-only sensor(s) will be triggered 600 microseconds before the transmitting sensor. SonicLib will adjust for the pre-triggering delay when calculating range values.

Enabling pre-triggering will reduce the maximum range of the receive-only sensor(s), relative to the setting specified in *ch_set_max_range()*, by about 200mm. You may want to increase the maximum range setting accordingly.

CH_MODE_IDLE – Idle Mode

If the sensor will not be used by the application for an extended period, it may be placed into a low-power Idle mode. No sensing will be performed when in Idle mode. To resume sensing operation, the sensor must be placed into one of the regular modes.

8.8 Setting Multiple Configuration Values

The preceding sections describe how individual configuration values are set using separate SonicLib API functions. Using the individual functions is recommended, because it allows the application developer full control of all SonicLib and sensor features.

For convenience, several basic settings may instead be changed at one time by using the *ch_set_config()* function. Note that not all features are controlled using this "shortcut" function, so it may not be sufficient for your application, and you may still need to use the individual functions for those additional settings.

The *ch_set_config()* function has the following definition:

```
uint8_t ch_set_config (ch_dev_t *dev_ptr, ch_config_t *config_ptr)
```

The *config_ptr* parameter is the address of a **ch_config_t** structure containing the various configuration values. The fields in this structure directly correspond to the parameters of the individual routines that set single values.

In general, the *ch_set_config()* function and the corresponding individual routines perform the same operations. However, *ch_set_config()* will also call *ch_set_mode()*, so sensing will begin immediately.

Because multiple values are set during *ch_set_config()*, you may need to read the current values for some fields to avoid changing settings. See Getting the Current Configuration Values, below.

8.9 Getting the Current Configuration Values

SonicLib provides routines to get the current configuration values and settings for the sensor(s). In general, for every *ch_**set**_xxx()* API function, there is a corresponding *ch_**get**_xxx()* function to return the corresponding configuration value(s).

In general, these calls only report values that are stored locally. They do not actually query the sensor device, so they do not impose any significant timing overhead.

The SonicLib API functions should always be used to obtain configuration values. They should not be read or interpreted directly from the **ch_dev_t** device descriptor or other structures.

Getting Individual Values

Each configuration setting has a dedicated "get" routine to return its value, corresponding to the "set" routine used to specify the value. So, for example, the maximum range value may be obtained by calling *ch_get_max_range()*, and the operating mode may be obtained by calling *ch_get_mode()*.

Getting Multiple Values

A limited set of common configuration values may be read together using the *ch_get_config()* function, which has the following definition:

```
uint8_t ch_get_config (ch_dev_t *dev_ptr, ch_config_t *config_ptr)
```

The *config_ptr* parameter is the address of a **ch_config_t** structure to be filled in with the current configuration values. This is the same type of structure used to set multiple configuration values using *ch_set_config()*.

9 SENSOR FIRMWARE PLUG-INS

One of the key features of the InvenSense ultrasonic sensor is its programmability. The full internal sensor operation is defined by sensor firmware that is loaded into the device before it starts to operate. SonicLib includes some of the most popular sensor firmware types, for common rangefinding applications. However, other firmware is available from InvenSense for special purposes, such as human presence detection.

The SonicLib "Plug-in" mechanism allows sensor firmware to be installed incrementally into an existing SonicLib installation. The new firmware is then available to all applications that use SonicLib.

In some cases, the sensor firmware is designed to interact with special algorithm libraries, which are separate from SonicLib and are released separately, often combined with a dedicated example program. Such example programs normally are distributed with the sensor firmware plug-in already installed in the included copy of SonicLib.

9.1 Firmware Files

A sensor firmware plug-in typically consists of three files that will be copied into your SonicLib installation. A new subdirectory under **soniclib/sensor_fw** will be created to hold the plug-in files

- Firmware header file – symbol definitions for the new firmware. Includes register set definition (see below).
- Firmware interface source file – initialization routine for new firmware, possibly with other support routines.
- Firmware image source file – sensor firmware executable image, with byte values defined as a C integer array.

As an example, imagine that you are installing a new "ICU NewAlgo" sensor firmware release. (You will have to substitute the actual filenames from the release you are installing.) The release files would be copied to a new **soniclib/sensor_fw/icu_newalgo** directory:

- **icu_newalgo.h** firmware header file
- **icu_newalgo.c** firmware interface source file
- **icu_newalgo_fw.c** firmware image source file
- *other files specific to newalgo firmware*

9.2 Register Set

As mentioned earlier, the SPI or I^2C register locations in the ultrasonic sensor are not hardware locations and are defined solely by the sensor firmware being used. The header file included with a sensor firmware release therefore contains symbolic definitions for the register set for that version. These symbols are used internally by the SonicLib sensor driver.

These definitions are provided for reference but are subject to change without notice. You should not access registers directly in your application to avoid compatibility issues with future releases.

9.3 How to Use a New Firmware Plug-In

Installation

SonicLib plug-ins generally only require copying the new release files from the distribution .zip file to their correct locations in SonicLib. See the instructions included with the specific SonicLib plug-in(s) you are using.

Specify the Sensor Firmware Init Routine

To use the new sensor firmware, specify its initialization routine during the application's call to *ch_init()*. The initialization routine name is based on the sensor firmware filename, with "*_init*" added.

So in our example, the firmware initialization routine would be "*icu_newalgo_init()"*, and the call to *ch_init()* would look something like this:

```
ch_init(dev_ptr, grp_ptr, dev_num, icu_newalgo_init);
```

Add the New Files to the Project Build (if necessary)

Depending on what development environment you are using, you may need to add the new firmware interface and image files to the set of project files that will be built. Some IDEs (e.g., STM32 Cube IDE) will automatically detect the new source files and build them, while others (e.g., Microchip MPLAB X or Atmel Studio) require the new files to be explicitly added.

Refer to your IDE or toolchain documentation for more information.

SonicLib plug-ins frequently support alternate build mechanisms such as CMake. See the individual plug-in documentation for more information.

10 REVISION HISTORY

Revision Date	Revision	Description
07/15/2024	3.0	Updated for SonicLib V4
07/28/2023	2.0	Updated for SonicLib v3 and ICU sensors.
04/10/2020	1.0	Initial public release. Minor changes and reformatting based on review.
01/22/2020	0.2	Assigned doc number, updated copyright.
11/08/2019	0.1	Initial draft version.