Chirp Microsystems
SonicLib Programmer's Guide

Chirp Microsystems
2560 Ninth Street, Ste 220, Berkeley, CA 94710
U.S.A
+1(510) 640–8155
www.chirpmicro.com

Document Number: AN-000175
Revision: 1.0
Release Date: 07/31/2020

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Chirp SonicLib is a set of API functions and sensor driver routines designed to easily control Chirp 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.

Chirp 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 Chirp sensors in various situations. This is especially important, because all Chirp sensors are completely software-defined. Except for certain low-level programming interfaces, the I^2C “registers” are simply memory locations determined by the specific sensor firmware being used. They are subject to change between firmware revisions, and often vary significantly between firmware images with different feature sets.

The SonicLib API allows your application to use different Chirp sensor firmware images without requiring code changes. Only a single initialization parameter must be modified, and one #include line added to a header file, to use a new sensor firmware version.

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. For more information on SonicLib, refer to its included HTML documentation, which provides a complete description of all interfaces.

2 FILE ORGANIZATION

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

By convention, these file groups are organized into three separate sub-directories in the installation. Figure 1 shows a simple directory layout and the location of key files discussed in later sections.

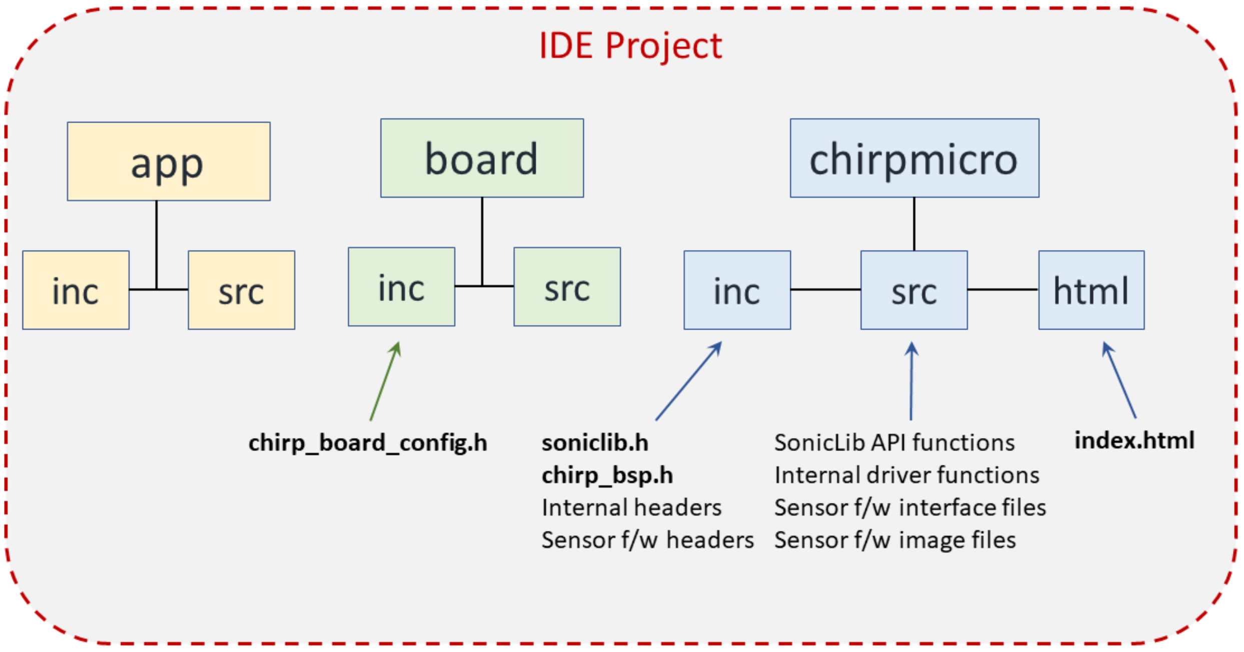

Figure 1. SonicLib Organization

The top-level sub-directories shown above are:

- **app** – Application-specific files. You may organize your application files in any way that makes sense for your development.
- **board** – Board support package (BSP) files. These files are specific to the hardware platform being used. The organization of these files is open to the BSP developer, but the **chirp_board_config.h** file must be provided.
- **chirpmicro –** Chirp distribution files. These directories hold the SonicLib API modules, the internal Chirp sensor driver, sensor firmware files, and HTML documentation. The **sonicLib.h** file contains the definitions for all SonicLib interfaces and is the key header file for all applications. The **chirp_bsp.h** file contains the definitions for all interfaces that must be implemented in a board support package. The **index.html** file in the **chirpmicro/html** directory is the entry point for the HTML documentation.

2.1 SonicLib and Chirp Distribution Files

The **chirpmicro** directory shown in Figure 1 holds the SonicLib, internal sensor driver, and sensor firmware files as well as documentation for the SonicLib interfaces.

Note: If you are using an example application from Chirp, the SonicLib distribution files are located in **source/drivers/chirpmicro**.

soniclib.h

The **soniclib.h** file, located in **chirpmicro/inc**, contains the complete definitions for the SonicLib API. This file must be included by any source modules that will use the sensor interfaces.

index.html

SonicLib includes fully linked HTML documentation, generated from the interface descriptions in **soniclib.h** and other files. The **index.html** file, located in **chirpmicro/html**, is the entry point for descriptions of the SonicLib and BSP interfaces. Open this file in a web browser to get started.

Sensor Firmware Files

The Chirp ultrasonic sensors are fully programmable and use separate sensor firmware images that are part of Chirp distributions. Each firmware package includes a header file (in the **chirpmicro/inc** directory) and two C source files (in **chirpmicro/src**). Some sensor firmware releases may be included in the default SonicLib distribution, while others may be available separately. See the "Sensor Firmware Releases" section for more information.

2.2 Board Support Package Files

The **board** directory shown in Figure 1 holds the board support package (BSP) files.

A board support package 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 (in **chirpmicro/inc**) contains the definitions of these hardware interfaces.

The BSP implementation is NOT part of SonicLib, and must be provided by the developer, board vendor, or Chirp. Contact Chirp for more information on available BSPs**.**

Note: If you are using an example application from Chirp, the BSP files are located under **source/board**. The file organization will vary.

chirp_bsp.h

This file defines the I/O interfaces that allow the standard Chirp 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 I^{2}C communications interface, timer functions, etc.

The BSP developer should not need to modify this file. However, that developer is responsible for implementing these support functions for the desired platform.

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.

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 two symbols must be defined in **chirp_board_config.h**:

- **CHIRP_MAX_NUM_SENSORS** = maximum number of Chirp sensors
- **CHIRP_NUM_I2C_BUSES** = number of I^{2}C bus interfaces

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

2.3 Application Files

The **app** directory shown in Figure 1 holds the application files.

As mentioned above, SonicLib does not impose any requirements or structure on your application. Example projects from Chirp use the **inc** and **src** directory names, but you are free to name or organize your application in any way.

All application modules that will use the SonicLib interfaces must include **soniclib.h**. Modules that handle initialization or use board resources managed in the board support package may also need to include **chirp_bsp.h**.

Note: If you are using an example application from Chirp, the application files are located under **source/application/<name-of-application>**.

2.4 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) to manage the files and build the project. Figure 1 indicates this with the "IDE Project" box surrounding the application, BSP, and Chirp distribution files.

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.

3 CREATING AN APPLICATION

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

3.1 "Hello Chirp" Example

The "Hello Chirp" example application, available from Chirp, demonstrates 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.

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

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 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.

Each sensor belongs to a sensor group, even if there is only one sensor in the system. (A group may have only one sensor or multiple sensors.)

ch_dev_t – Device Descriptor

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

The **ch_dev_t** structure contains fields for configuration values, status, function pointers, counters, etc. In general, you should not access the fields directly in your program – the SonicLib API provides dedicated "set" and "get" functions for most values of interest to an application.

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 a handle to identify the device in some SonicLib API calls that handle multiple sensors. Numerous BSP functions also use the **ch_group_t** structure address as an input parameter.

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 the **ch_group_t** structure for each sensor group.

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

3.3 Overall Application Flow

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

1. Initialize hardware (*chbsp_board_init()*).
2. Initialize SonicLib structures for each sensor (*ch_init()*).
3. Register callback routine(s) – see below.
4. Program and start all sensors (*ch_group_start()*).
5. Configure sensors (*ch_set_config()* etc.).
6. Enter endless loop to perform sensing:
    a. Callback functions are called by SonicLib/BSP to notify the application of events.
    b. Use *ch_get_range()* etc. to get measurement data.

Steps 1 through 4 are discussed below in the "Initialization" section of this document. Step 5 is covered in "Configuration" and Step 6 is discussed in "Sensing."

Callback Routines

Although most of the SonicLib interfaces are functions that are called by an application, in some cases it is 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.

SonicLib provides the ability to register two callback routines for specific events.

The most commonly used callback function is tied to the sensor's data-ready interrupt. When the sensor asserts the INT line, it causes an interrupt to occur. In most applications, the response to this interrupt is to read data from the sensor, using *ch_range_get()* and other SonicLib functions. The *ch_io_int_callback_set()* function is used to register a callback function to be called by the interrupt handler.

The other callback function is tied to the completion of a non-blocking read of data from the sensor. This is used when reading the raw I/Q amplitude data from a sensor in non-blocking mode. When the non-blocking read completes, it causes an interrupt to occur. In most applications, the response to this interrupt is to process the data that was read. The *ch_io_complete_callback_set()* function is used to register the callback function.

In addition, the Chirp board support package definition in **chirp_bsp.h** specifies a periodic timer mechanism that uses a callback. The callback routine is called when the periodic timer expires and is often used to trigger a measurement cycle. The callback is registered as part of the *chbsp_periodic_timer_init()* function.

4 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.

4.1 Hardware Initialization – *chbsp_board_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.

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

```
void chbsp_board_init (ch_group_t *grp_ptr)
```

Along with the actual hardware initialization, this BSP function also initializes the following fields within the **ch_group_t** sensor group descriptor:

ch_group_t 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_i2c_buses	Number of I^{2}C buses used by sensors on the board. Usually the same as *CHIRP_NUM_I2C_BUSES* in **chirp_board_config.h**.
rtc_cal_pulse_ms	Length (duration) of the pulse sent on the INT line to each sensor during calibration of the real-time clock, in milliseconds. Accessible later using *ch_get_rtc_cal_pulselength()*.

4.2 SonicLib Software Initialization – *ch_init()*

This function is used to initialize various Chirp 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 a Chirp 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

The Chirp sensor is fully re-programmable, and the specific features and capabilities can be modified by using different sensor firmware images. The *fw_init_func* parameter is the address (name) of the sensor firmware initialization routine that should be used to program the sensor 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.

Note: *ch_init()* only performs internal initialization of data structures, etc. It does not actually interact with the physical sensor device(s). That process is described below in “Sensor Initialization.”

4.3 Registering Callback Routines

As discussed earlier, the SonicLib API defines callback mechanisms for SonicLib or for the board support package to call an application routine to notify the application that an event has occurred. In SonicLib itself, callback routines can notify the application that data is ready from the device, or that non-blocking I/O has completed. In addition, the Chirp 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 flags or otherwise arrange for additional handling to be done in the regular application loop at task level.

I/O Interrupt Callback

Most applications will 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)
```

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.

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.

I/O Complete Callback

Another callback routine type is only used if the application performs non-blocking reads of raw I/Q data from the sensor. When the non-blocking read completes, it causes an interrupt to occur. In most applications, the response to this interrupt is to process the I/Q data that was read and is now stored in the buffer that was specified during *ch_get_iq_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 can 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.

Periodic Timer Callback

Although not part of the SonicLib API, the Chirp 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_function_name (void)
```

The *periodic_timer_callback_function_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.

4.4 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 I^{2}C bus and each sensor's PROG line, to discover if sensor is connected.
2. Reset sensor and put in known state.
3. Program sensor with firmware (version specified during *ch_init()*).
4. Assign unique I^{2}C address to sensor (specified by board support package, see *chbsp_i2c_get_info()*).
5. Start sensor execution.
6. Wait for sensor to lock (complete initialization, including self-test).
7. Send timed pulse on INT line to calibrate sensor Real-Time Clock (RTC).

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

5 CONFIGURATION

After the 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. These descriptions identify the individual SonicLib API calls that are used to set the values. For convenience, multiple settings may be changed at one time, see “Setting Multiple Configuration Values” below.

5.1 Sensor Operating Modes

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 overall pattern of operation. 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.

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.

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 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.

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 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.

By default, the single Tx/Rx node and all Rx-only nodes are triggered simultaneously. For improved performance at close distances, receive sensor pre-triggering may be enabled using the *ch_set_rx_pretrigger()* function. When pre-triggering is enabled, the Rx-only sensors will be triggered slightly before the Tx/Rx sensor.

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 (see above).

5.2 Maximum Range

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.

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 almost 6 milliseconds more time-of-flight (round-trip).

Longer maximum range settings also require the sensor to capture more data internally during the extended listening period. Typically, this does not significantly affect the sensor’s internal processing time, but it can be an important consideration if your application needs to read the raw I/Q measurement data. The time to read the full set of data (600 bytes for a CH101 sensor and up to 1800 bytes for a CH201) over I^{2}C can be significant in the context of the overall measurement timing. Shorter maximum range settings reduce the amount of valid I/Q data in each measurement, and therefore reduce the I/O time required to transfer the data.

Because of 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.

Samples vs. Distance

The sensor generates samples driven by its own internal clock whose frequency will vary somewhat from one device to another. As a result, the time between samples (and therefore the physical distance each sample index represents) will be different. So, the operating frequency of the device must be included in any conversions between internal sample index-based values and real-world physical distance.

SonicLib provides two conversion routines to easily translate physical distance into sample counts, and vice versa, based on the measured calibration values and device frequency. 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.

5.3 Internal Sample Interval (Free-running mode only)

For sensors in free-running mode (CH_MODE_FREERUN), the sample interval may be set using the *ch_set_sample_interval()* function. (Alternatively, the sample interval may instead be set using the *ch_set_config()* function, which also sets other configuration values.)

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

```
uint8_t ch_set_sample_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.

The sample interval setting has no effect on sensors using one of the triggered modes.

5.4 Static Target Rejection

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. Static Target Rejection (STR) is special sensing feature 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 the entire measurement sample set. In normal operating conditions with a stationary sensor and a static target, the steady amplitude from the target's echo will allow the filter to reject this signal. However, moving targets will vary in distance and amplitude, and even movements of millimeters can create a detectable signal. Note that some conditions, such as high airflow around the sensor, can cause the echo amplitude of a static target to vary enough to register as a target.

Due to restrictions in the sensor, STR is implemented slightly differently for CH101 vs. CH201.

STR for CH101

For CH101, STR is a runtime option available in General Purpose Rangefinding (GPR) sensor firmware. STR may be enabled using the *ch_set_static_range()* function. (Alternatively, the static rejection range may instead be set using the *ch_set_config()* function, which also sets other 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 <u>samples</u> (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. If you want to enable STR for the entire range of the sensor, you can use the *ch_get_max_samples()* function to determine the maximum possible number of samples for the sensor firmware being used, and set *num_samples* to the obtained value. Using a value of zero for *num_samples* will disable STR.

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 3 ms for a sensor with a maximum range setting of 1m.

STR for CH201

For CH201, a separate sensor firmware type is used to provide the STR feature. The CH201 GPR STR firmware always operates with STR enabled for the full range of samples in a measurement. The *ch_set_static_range()* function is not used.

5.5 Multiple Detection Thresholds (CH201 only)

The long-range CH201 sensor supports multiple detection thresholds, to tune the ability of the sensor to detect objects at various distances. When the sensor analyzes the results from a measurement, the amplitude of each sample in the trace is compared with a threshold value to decide if the signal is strong enough to be a target object. Multiple detection thresholds allow blocks of contiguous samples in the measurement set (i.e. different distance ranges from the sensor) to use different comparison thresholds. This allows farther objects, with weaker return signal amplitudes, to be detected more consistently.

Up to six thresholds may be set when using the standard CH201 GPRMT (General Purpose Rangefinding / Multi-Threshold) firmware. Each threshold is described in a **ch_thresh_t** structure, which consists of a starting sample number and the detection threshold (amplitude value) required for an object to be detected. The first threshold must 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 set of six individual thresholds are 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)
```

5.6 Setting Multiple Configuration Values – *ch_set _config()*

The preceding sections describe how individual configuration values are set using separate SonicLib API functions. For convenience, multiple settings may be changed at one time by using the *ch_set _config()* function.

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.

The *ch_set_config()* function and the individual setting routines perform exactly the same operations and may be freely mixed. Use whichever calls are most convenient in your application.

Because multiple values are set during *ch_set_config()*, you must be careful to initialize all fields in the **ch_config_t** structure before it is called. You may need to read the current values for some fields to avoid changing settings. See “Getting the Current Configuration Values” below.

5.7 Getting the Current Configuration Values

SonicLib provides routines to get the current configuration values for the sensor(s). Similar to the functions which set the configuration values, configuration values may be read one-at-a-time using dedicated functions, or multiple configuration values may be read at once. 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.

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()*.

The basic set of configuration values may be read 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()*.

You may combine these functions to allow partial updating of the configuration settings, by calling *ch_get_config()* to get the current settings, modifying some of the fields, then writing back the full configuration set by calling *ch_set_config()*.

6 SENSING

This section describes the various SonicLib functions used for basic ultrasonic sensing. The sensor(s) must have been initialized and configured as described in the preceding sections.

The Chirp sensor is an ultrasonic transceiver, meaning that it both transmits and receives ultrasound signals. Unlike various type of passive sensors which simply measure their surrounding conditions, the Chirp device 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 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 (trigger) a new measurement cycle. The sensor will emit an ultrasound pulse.
2. Wait while the sensor listens for a receive ultrasonic pulse (either reflected or from a different sensor).
3. The sensor will indicate data-ready by asserting its INT line.
4. The I/O Interrupt Callback routine (registered by *ch_io_int_callback_set()*) is called by SonicLib.
5. The callback routine calls *ch_get_range()* or other functions to read sensor data. (The callback function may simply set a flag causing the data read to be done in the regular application loop.)

6.1 Triggering a Measurement

Triggering refers to the external initiation of a sensor measurement cycle by setting the sensor’s INT line high.

A single sensor in free-running mode (*CH_MODE_FREERUN*) does not require external triggering. In all other cases, the sensor must be triggered by a pulse on its INT line to begin a measurement cycle.

SonicLib provides two functions to trigger sensors, depending on the needs of the application.

In most cases, all sensors in a group are used together to generate range measurements. Specifically, one sensor may 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 must be triggered together, so that the receive-only nodes know when the pulse was transmitted and can calculate to time-of-flight when the pulse is received.

The *ch_group_trigger()* function is used to trigger all sensors in a group together. 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, it does not matter whether you use *ch_trigger()* or *ch_group_trigger()*.

After being triggered, a sensor will begin and complete a measurement cycle, based on the current configuration settings. When the measurement cycle completes, each sensor will cause a “data-ready” interrupt on the host processor by asserting its INT line. The application should wait for all sensors to complete before reading the range or other data from the device. Any I/O operations with the sensor between the trigger and the data-ready signal may interfere with the ultrasonic measurement.

Using a Periodic Timer

In many applications, the best way to trigger 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.

Receive Sensor Pre-triggering

Receive sensor pre-triggering is an option for multi-sensor configurations in which one sensor is in Tx/Rx mode (*CH_MODE_TRIGGERED_TX_RX*) and one or more other sensors are in Rx-only mode (*CH_MODE_TRIGGERED_RX_ONLY*). Pre-triggering significantly reduces the minimum distance at which pairs of sensors can operate and generally improves performance at close distances.

By default, the single Tx/Rx node and all Rx-only nodes are triggered simultaneously. Receive sensor pre-triggering may be enabled using the *ch_set_rx_pretrigger()* function. When pre-triggering is enabled, the Rx-only sensors will be triggered slightly before the Tx/Rx sensor. Later, the *ch_get_range()* function will automatically adjust for the timing difference when calculating range values.

When pre-triggering is enabled, the maximum range for Rx-only sensors is reduced relative to the value set by *ch_set_max_range()*. The effective maximum range of the Rx-only sensor(s) will be reduced by approximately 200 mm. (The range of the Tx/Rx sensor is not affected). You may want to increase the maximum range setting in your application for Rx-only sensors to compensate for this reduction. However, you cannot increase the range beyond the maximum for the sensor (which is ultimately reduced by 200 mm for Rx-only sensors when pre-triggering is used).

When used, receive sensor pre-triggering is enabled for all Rx-only sensors in a sensor group.

6.2 Reading the Range Value – *ch_get_range()*

The most fundamental 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 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 preceding measurement cycle (i.e. the one which generated the INT line assertion) was successful in detecting a target object, the value returned by this function contains the calculated range in fixed-point format (see “Units” below).

If the measurement did not detect a target object, *ch_get_range()* will 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 the type of range to 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 “Selecting the Range Type to Report” below.

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 to generate a data-ready interrupt. (Typically, an application will be notified via a callback routine registered using *ch_io_int_callback_set()*.)

The range value (and other measurement results) 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). Once a new measurement has started, do not use *ch_get_range()* or other functions that interact with the sensor, or the measurement may be corrupted.

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`).

Selecting the Range Type to Report

A Chirp ultrasonic sensor may be used alone or in combination with one or more other sensors.

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.

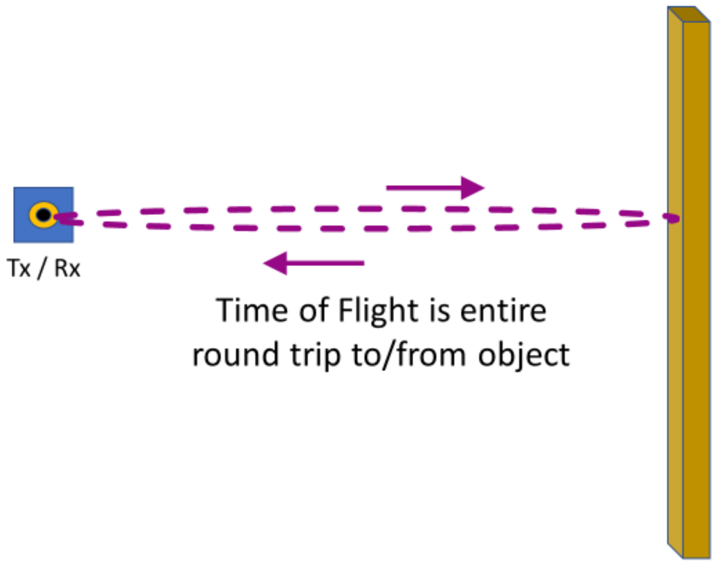

Figure 2. Pulse-Echo Sensing using a Single Sensor

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. This mode of operation gives the best performance in terms of measurement accuracy and stability. See Figure 3.

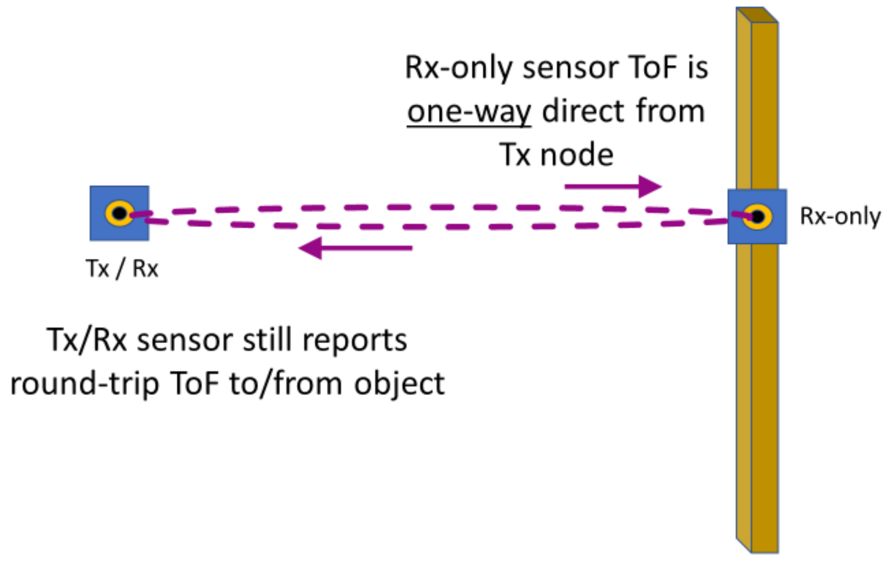

Figure 3. Direct Pitch-Catch Sensing using Two Sensors

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*.

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.

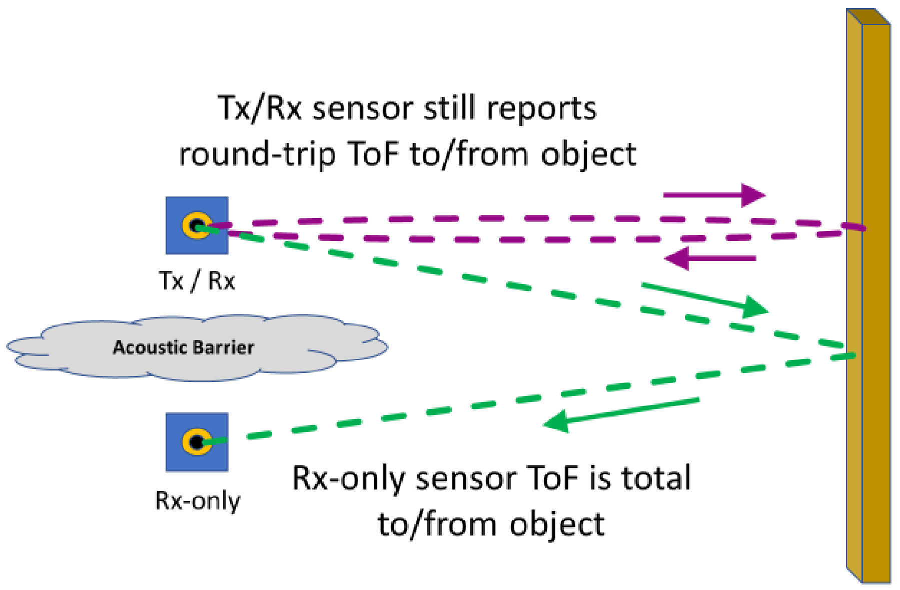
Tx/Rx sensor still reports
round-trip ToF to/from object
Tx / Rx
Acoustic Barrier
Rx-only
Rx-only sensor ToF is total
to/from object

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_ROUND_TRIP is appropriate for the receive-only node, similar to the single-sensor configuration. 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.

Note the use of an acoustic barrier to minimize the ultrasound pulse transmission directly between the sensors.

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

Along with the range measurement value, the sensor also reports an amplitude value for the detected target object's signal. In general, the amplitude value is much more variable than the range (due to 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.

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 a successful range measurement, i.e. one in which *ch_get_range()* does not return *CH_NO_TARGET*. Otherwise, the amplitude of the last successful range measurement will be reported.

6.4 Reading Amplitude Data – *ch_get_amplitude_data()*

In some cases, it is useful to get more detailed data from the sensor than the simple range and target amplitude. The *ch_get_amplitude_data()* function allows an application to obtain individual amplitudes for the internal samples that make up a measurement.

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, which is defined as an array of unsigned 16-bit integers. The buffer must be large enough to hold the requested number of amplitude values.

To read the entire set of amplitude values from a measurement cycle, the *start_sample* parameter should be zero. The *num_samples* value should be the total number of active samples in each measurement, based on the current maximum range setting. This value can be obtained using *ch_get_num_samples()*.

However, an application may only be interested in a specific subset of the amplitude values. For example, if the application only needs to obtain the amplitude 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*.

When using some types of Chirp sensor firmware, the amplitude values are calculated directly on the sensor. More typically, the *ch_get_amplitude_data()* function works by reading raw I/Q data (see below) from the sensor into a temporary buffer and calculating the amplitude for each I/Q pair.

6.5 Raw I/Q Data

What is I/Q data?

The I/Q data from the sensor is the "raw" amplitude data from a measurement cycle. Each individual sample in the measurement is reported as a pair of values, I and Q, in a quadrature format. (For a CH101 device, this is up to 150 sample pairs).

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:

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

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. The *ch_iq_to_amplitude()* function may be used to perform the above calculation for a single I/Q pair.

The number of samples used in the I/Q trace is determined by the maximum range setting for the device. If it is set to 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.

Each sample I/Q pair consists of two signed 16-bit integers. So, a complete CH101 measurement will contain up to 600 bytes of data (150 samples x 4 bytes per sample). When the I/Q data is read from the sensor, the additional time required to transfer the I/Q data over the I^2C bus must be taken into account when planning how often the sensor can be read (sample interval). It is important that any data I/O to the sensor, including reading the I/Q data, completes before a new measurement cycle is triggered.

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. It 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. The buffer must be large enough to hold the requested number of sample values. In the CH101 sensor, up to 150 samples are taken during each measurement cycle. So, a complete CH101 I/Q trace will contain up to 600 bytes of data (150 samples x 4 bytes per sample). A complete CH201 I/Q trace may contain up to 1800 bytes of data (450 samples).

An application will typically 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 is the total number of active samples per measurement, based on the current maximum range setting. This value can be obtained using *ch_get_num_samples()*.

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*.

Non-blocking Read Mode (optional)

To allow more flexibility in your application, the I/Q data readout from the device may be done in a 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. The *ch_get_iq_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 I/O. A notification will later be issued when the I/Q has been read.

To use the non-blocking option, the board support package (BSP) you are using must provide the optional *chbsp_i2c_read_nb()* and *chbsp_i2c_read_mem_nb()* functions. 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. Synchronize with all sensors whose I/Q data should be read, by waiting for all to assert their INT lines to indicate data ready.
4. Set up a non-blocking read on each sensor, using *ch_get_iq_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 data, possibly by setting a flag that will be checked from within the application's main execution loop. The callback routine will likely be called at interrupt level, so the amount of processing within it should be kept to a minimum.

6.6 How Often to Start a New Measurement

Most applications do not make single, isolated range 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 the same considerations apply to both externally triggered sensors and those running in free-running mode using an internal sample interval).

It is very 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 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.
- The standard internal processing time by the sensor will vary somewhat, but 5 ms is a useful estimate. (This is the time between the end of the sensor's listening period and its assertion of the INT line to indicate data-ready).
- 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 I^2C, significant additional time is required. The amount will vary depending on how much data is read and the I^2C throughput in the BSP. Readout of a full CH101 data set (150 samples, or 600 bytes) will generally require 30 to 40 ms. Readout of a full CH201 data set (450 samples, or 1800 bytes) will generally require 90 to 120 ms.

These values are summarized below.

LISTEN FOR ULTRASOUND	SENSOR PROCESSING	STATIC TARGET REJECTION	I^2C TRANSFER (BASIC)	I^2C TRANSFER (I/Q DATA)
6 ms per meter of range	5 ms	3 ms (if used)	1 ms	up to 120 ms (if used)

Table 1. Measurement Time Requirements

All of these time requirements are only general estimates. To get the most 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. Chirp 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.

7 SENSOR FIRMWARE RELEASES

One of the key features of the Chirp 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.

From time to time, Chirp releases new firmware images for supported sensors. These new versions may add (or remove) features or improve performance. SonicLib is organized to make it easy for you to add these new versions and use them with your application.

7.1 Files

A sensor firmware release typically consists of three files that must be copied into your SonicLib installation.

- Firmware header file – symbol definitions for the new firmware. Includes register set definition (see Section 7.2).
- Firmware interface source file – initialization routine for new firmware, possibly with other support routines.
- Firmware image source file – sensor firmware image, with byte values defined as a C integer array.

7.2 Register Set

The I^{2}C register locations in the Chirp sensor are not hardware locations. They are defined solely by the specific sensor firmware being used and are subject to change. The header file included with a sensor firmware release contains the symbolic definitions for the register set for that version. These symbols are used internally by the SonicLib sensor driver.

You may use these definitions for reference, but Chirp strongly discourages you from accessing registers directly in your application to avoid compatibility issues with future releases.

7.3 How to Use a New Firmware Version

Copy Files

The three files that typically make up a sensor firmware release must be copied to the same directory tree containing your SonicLib files. The header file is copied to the **inc** directory, and the two .c source files are copied to the **src** directory.

As an example, assume that your SonicLib distribution directory is called **chirpmicro** and you are installing the new “CH101 NewStuff” sensor firmware release. (You will have to substitute the actual filenames from the release you are installing). The release files would be handled like the following:

• **ch101_newstuff.h**	firmware header file – copy to **chirpmicro/inc**
• **ch101_newstuff.c**	firmware interface source file – copy to **chirpmicro/src**
• **ch101_newstuff_fw.c**	firmware image source file – copy to **chirpmicro/src**

Add Header to soniclib.h

The **soniclib.h** header file includes all sensor firmware-specific headers, so the header file for the new firmware must be added to its list of headers.

Using the same example names as in the previous section, do the following to add the header for the new release:

- Open **chirpmicro/inc/soniclib.h** in a text editor.
- Near the start of the file, locate the section marked, “Chirp header files for installed sensor firmware packages”
- Add the following line at the end of the list of Chirp firmware header files:

```
#include “ch101_newstuff.h”
```

- Save and close the **soniclib.h** file.

Specify the Sensor Firmware Init Routine

The final step to use the new sensor firmware is to 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 "*ch101_newstuff_init()*", and the call to *ch_init()* would look something like this:

```
ch_init(dev_ptr, grp_ptr, dev_num, ch101_newstuff_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. OpenSTM32 System Workbench) will automatically detect the new files and build them, while others (e.g. Atmel Studio) require the new files to be explicitly added.

Refer to your IDE or toolchain documentation for more information.

8 REVISION HISTORY

Revision Date	Revision	Description
07/31/2020	1.0	Initial release. Updated for SonicLib v2.1.