ICU-20201 EVK PLATFORM USER GUIDE

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-000398
Revision: 1.0
Date: 02/17/2023

Table of Contents

1 INTRODUCTION

ICU-20201 is an ultra-low power, miniature system-in-package, long-range ultrasonic Time of Flight (ToF) sensor, capable of providing millimeter-accurate range measurements of multiple objects in its Field of View (FoV) at distance of up to 5m, independent of the target`s structure, color, optical transparency, or the environment around it. The sensor is an easy to integrate solution that can be programmed to work with both long- and short-range modes, the FoV can be customized up to 180^{o} depending on the intended sensing application and the echo information can also be processed with several available algorithms for the various use cases.

This purpose of this document is to walk through the hardware setup, the execution of the GUI and the evaluation of ICU-20201 sensors. It explains all the available tools and support options that can be used to configure the device to achieve the best optimal performance.

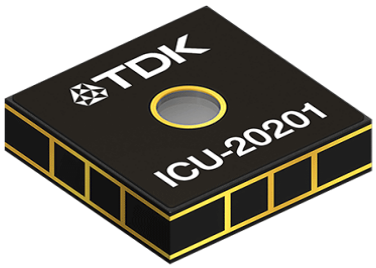


Figure 1: ICU-20201 Sensor

2 HARDWARE & SOFTWARE SETUP

This section elaborates the hardware and software required to set up and run the development kit for the ICU-20201.

2.1 INSTALLING THE PACKAGE

Download *ICU-x0201_EVK-sierra_evk-x.x.x-installer.exe*, here *x.x.x* reflects the current version number and varies/changes with each release.

Then, double click on the downloaded file package.

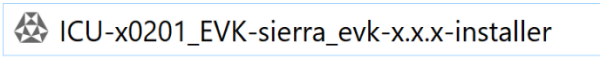


Figure 2: Download the package

User Account Control window will pop-up, click Yes.

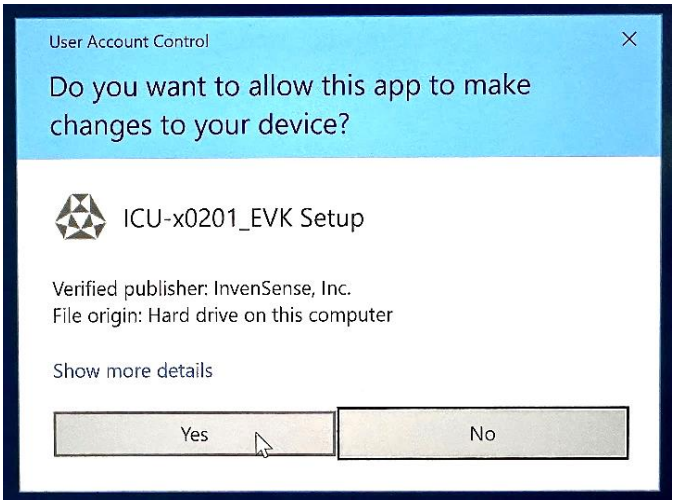


Figure 3: Download setup

Select I accept the agreement and click on Next.

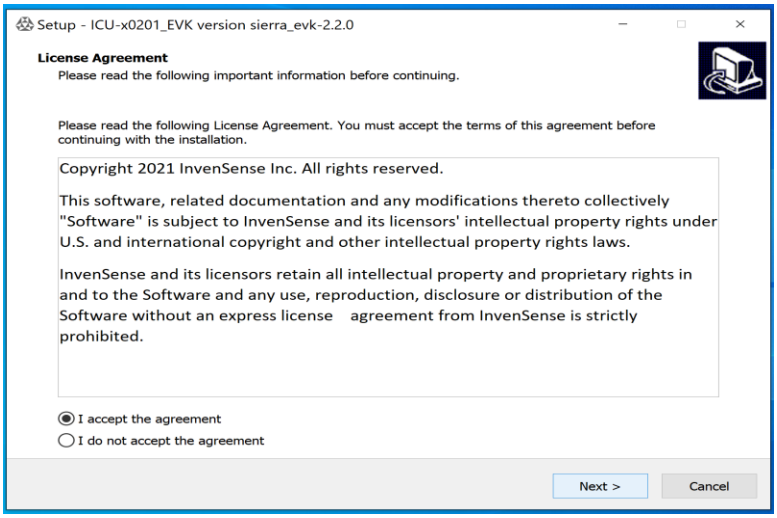


Figure 4: Install the setup

Choose where you want to install it and then click Next, and then Next again.

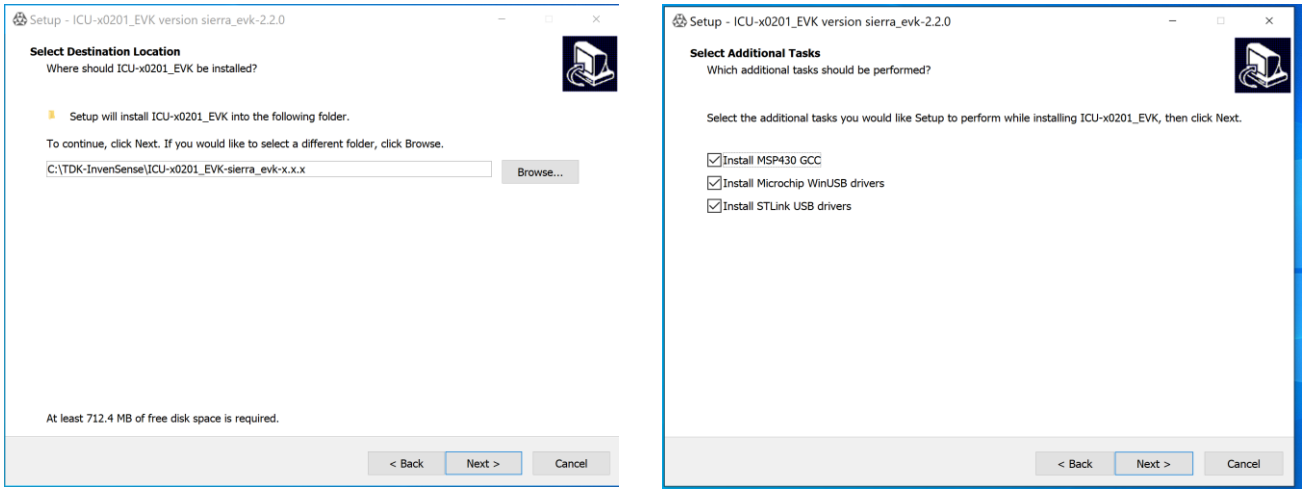


Figure 5: Installation steps

Click on Install.

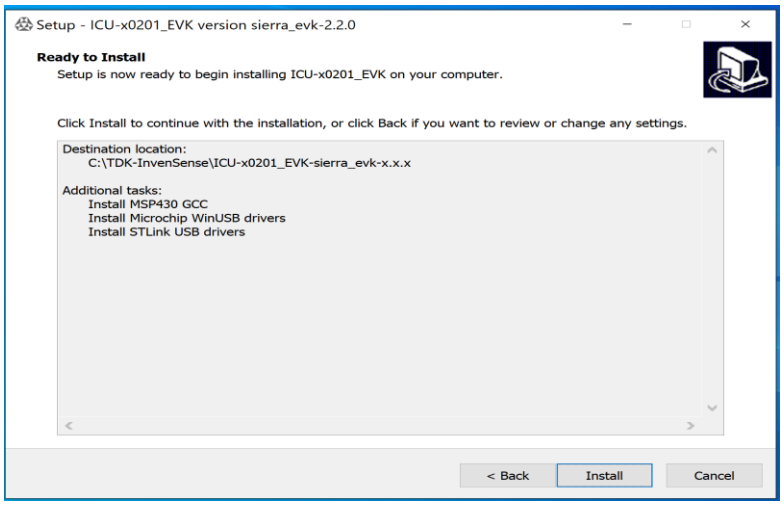


Figure 6: Ready to install

The install will begin.

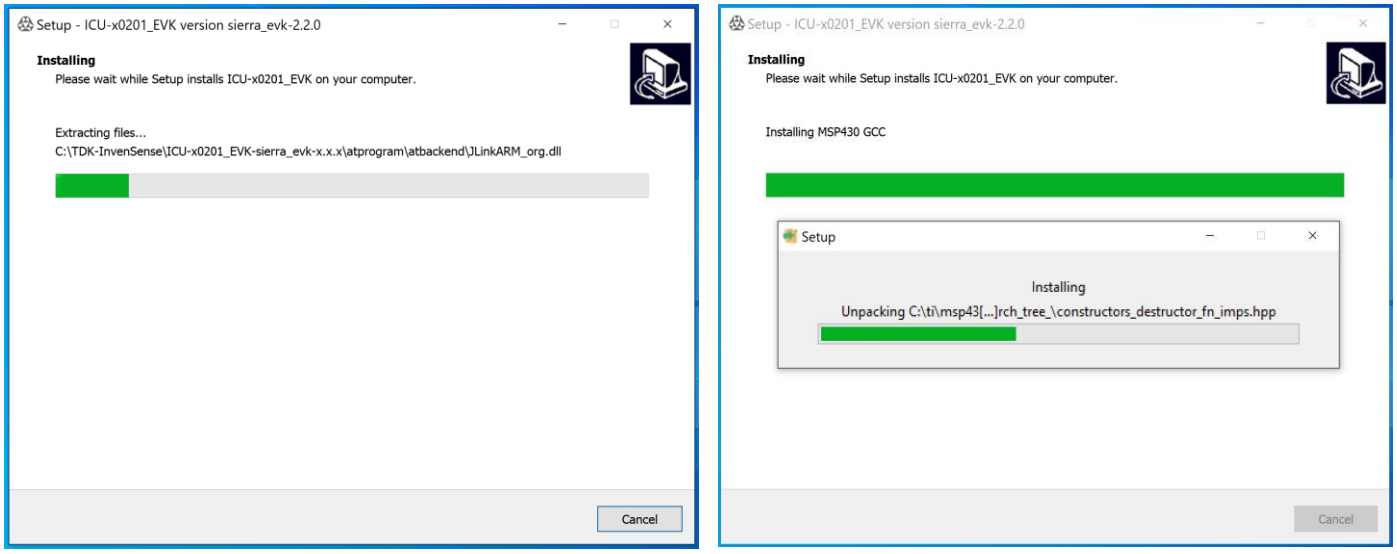


Figure 7: Installation in progress

Click on finish once the install is complete.

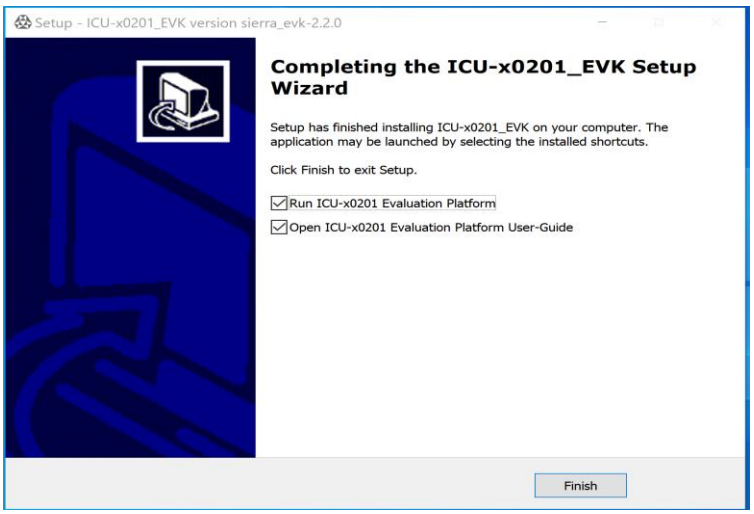


Figure 8: Installation complete

For the first time, the GUI will launch automatically. For later use, go to the location the package was installed at.

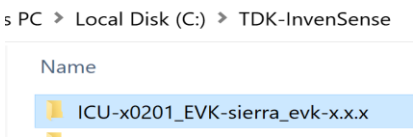


Figure 9: Opening the GUI

Scroll down to ICU_x0201_EVK to launch the GUI.

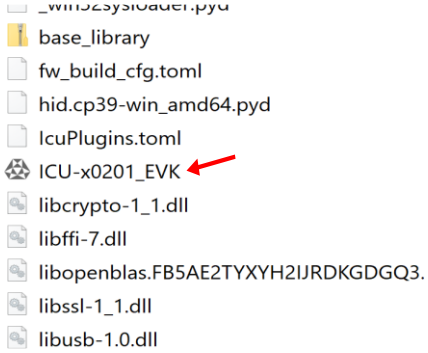


Figure 10: Finding the app

This will open the evaluation platform.

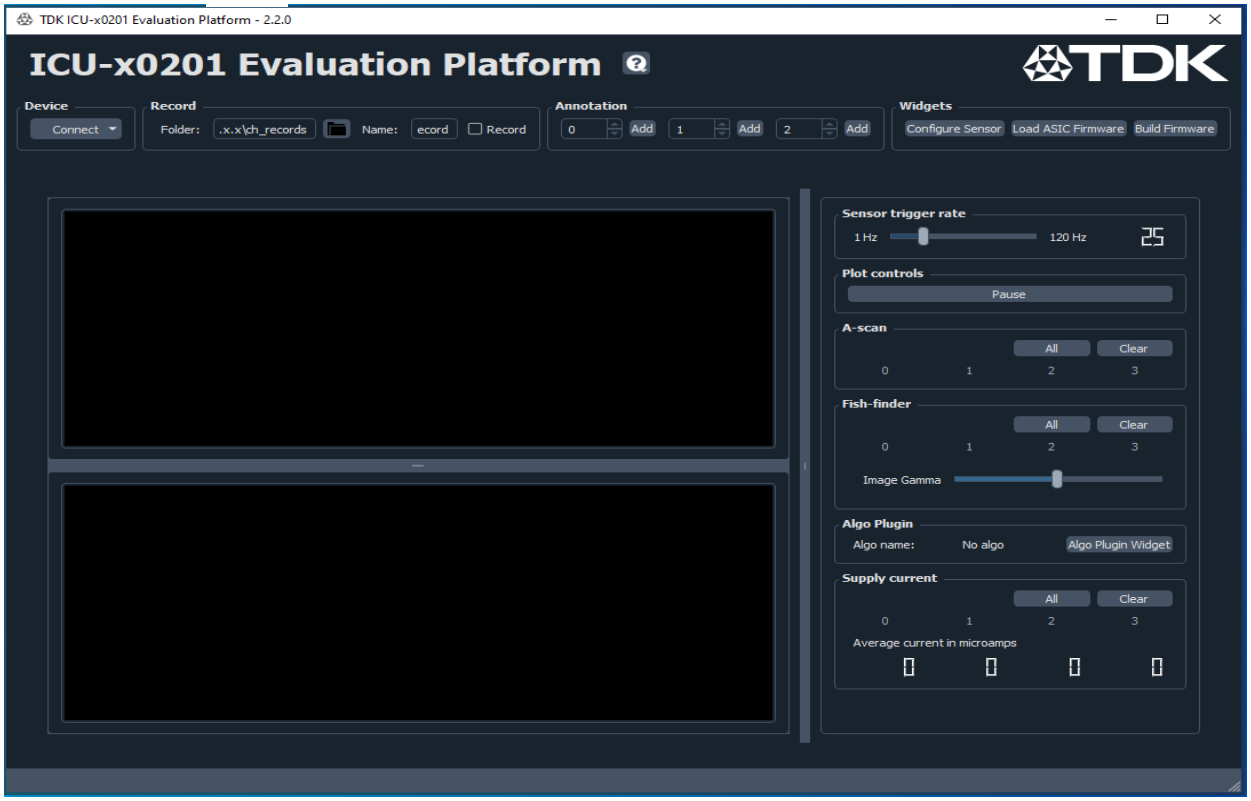


Figure 11: EVK platform

You can also go to the Device Manager and verify that the USB connections have been established, by checking the listed USB serial devices under Ports (COM & LPT) drop down. You should be able to see the connected COM ports.

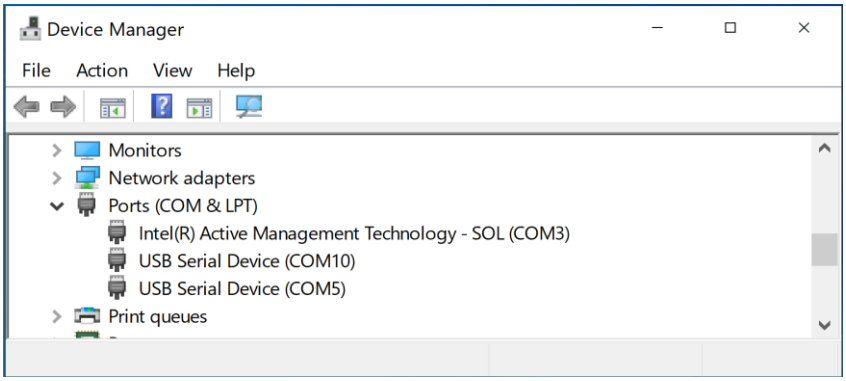


Figure 12: USB Serial Devices

2.2 HARDWARE REQUIREMENTS

The following hardware is needed to the establish connection and set the platform in order to evaluate the sensor.

i) The development/host board - DK-x0201

ii) The daughter/evaluation board - PN100-06351

iii) ICU-20201 module(s) - EV_MOD_ICU-20201-00-0x

iv) Flat Flex Cable(s)

v) USB Cable

Note: The sample kit comes with 2 Flat Flex Cables (FFC); if needed additional cables can be order from the distributors using the manufacturer Multicomp Pro`s part number MP-FFCA05123052A. Recommended distributor: **Newark, Part No.** 83AH2524

2.3 CONNECTIONS

The image below shows the physical connections to run the ICU-20201 EVK. Up to 4 sensors can be attached to the connector on the daughterboard using the flat flex cable (FFC). The image below shows the physical connections to run the ICU-20201 software.

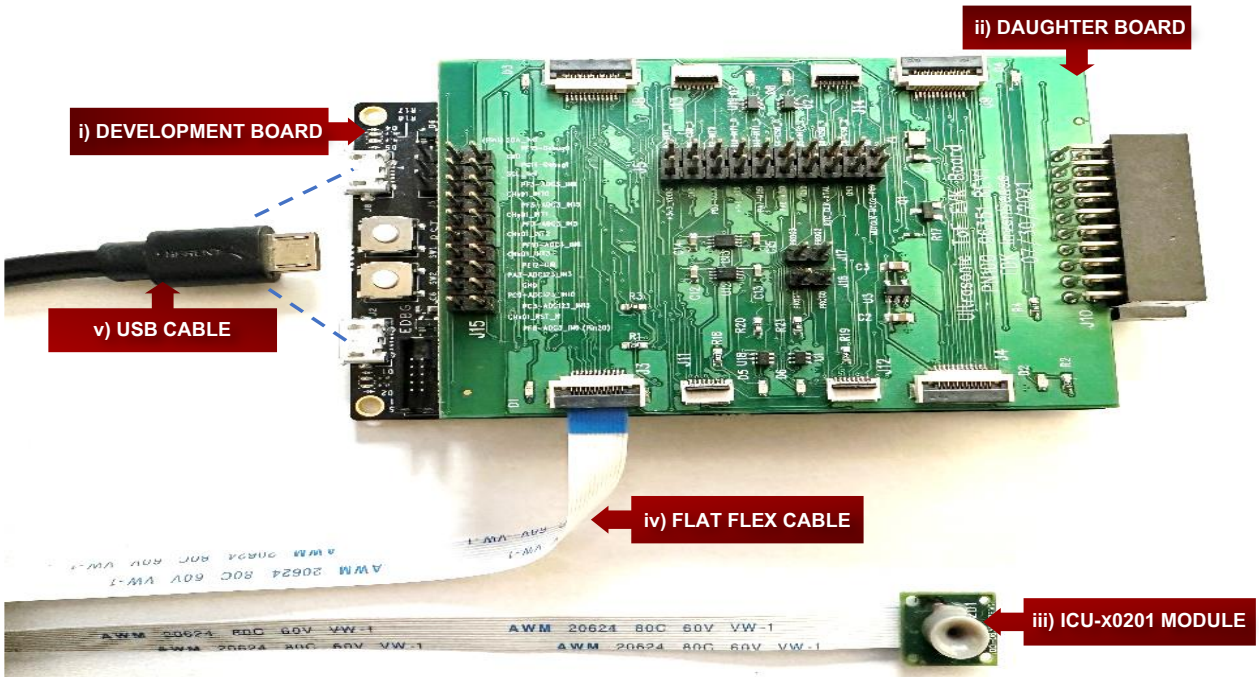


Figure 13: Hardware setup

2.4 UPDATING THE FIRMWARE

To update the firmware, connect the board`s EDBG connector via the USB cable to the PC.

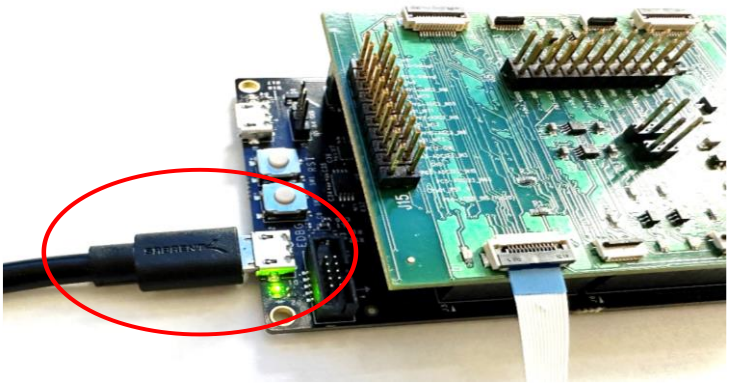


Figure 14: Connection to update the firmware

The firmware can be flashed directly from the GUI. Click the small arrow on the right side of the connect button and select ‘flash default firmware’.

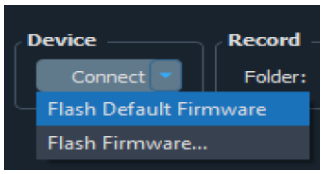


Figure 15: Flashing firmware

The following confirmation message will appear.

Figure 16: Confirmation message

Press the “Yes” button and the Update device firmware procedure will start.

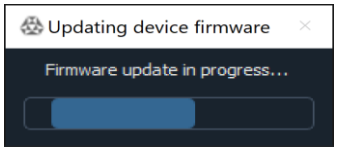


Figure 17: Update in progress

At the end of the FW update procedure, a success message will appear.

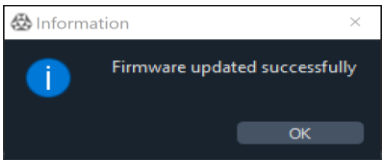


Figure 18: Firmware updated

Press “OK” to complete the procedure.

3 GUI OVERVIEW

To run the ICU-20201 evaluation kit and the platform, connect the board`s FTDI connector via the USB cable to the PC.

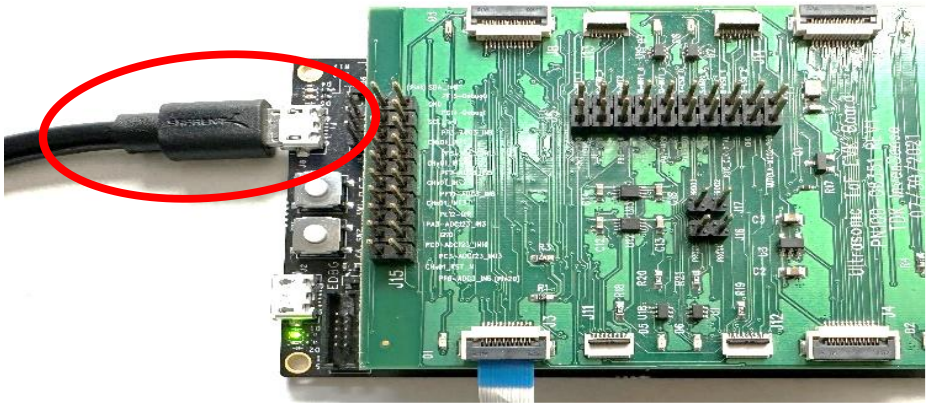


Figure 19: USB connection to run the EVK

Once the connections are made, click on the Connect 1 button to start the data acquisition. To stop the data acquisition or change the number of sensors connected to the evaluation board, press Disconnect 1

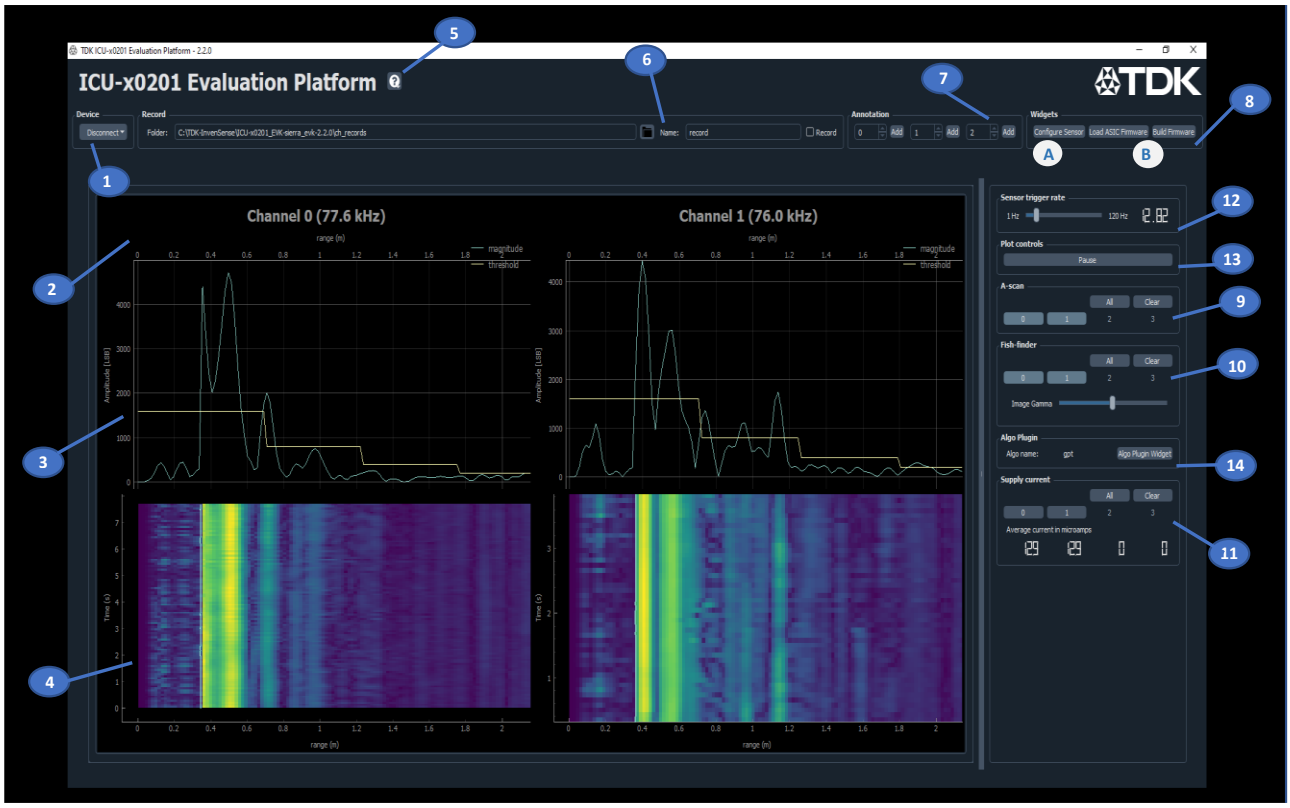


Figure 20: GUI main window

3.1 UNDERSTANDING THE PLOTS

The data plots 2 can be previewed in the A-scan and Fish-finder section of the interface. The Pause button 13 will freeze the plots but does not stop the sampling. Press the Resume button to start to plot the data again.

A-Scan

A-scan 3 captures and displays the amplitude of the current echoes measured by the sensor in term of least significant bits (LSBs) versus the range in meters. The amplitude is proportional to the received sound pressure. You can enable plots for connected sensors 0-3 to visualize the data of the concerned connected sensor. Each sensor data is displayed on a separate plot, stacked vertically, with horizontal axes for the range locked together.

→ Display control

The display control section 9 has buttons that control which plots are shown for which sensors. For each type of plot, there are buttons for adding the different connected sensors (up to 4).

Fish-finder

The Fish-finder 4 tracks and displays history of the A-scan. The x-axis is the range in meters and shares the same scale as the A-scan plot. Each new A-scan capture is added at the top of the image, and therefore the y-axis is the time in seconds, and the time axis scales according to the sample rate. The color of the image indicates the level of intensity or amplitude of the ultrasound pulses. The range data is overlayed as vertical lines to help visualize where the sensor is finding targets. Again, you can choose from 0-3 to visualize the data of the concerned connected sensor.

→ Image gamma control

This 10 adjusts the gamma of the fish-finder image. Lower values of gamma effectively compress the image contrast, making it easier to see low level signals. Higher values of gamma instead highlight the strong signals and tend to crush all the weak signals into the background noise. Note that his is purely a display affect and has no effect on the raw sensor data.

Algo Plugin

The algo plugin **14** is designed to show the results of the embedded algorithm running on the ICU-x0201 ASIC. The default algo is General Purpose Transceiver (GPT), which measures the range to up to 5 targets. For the gpt plugin, the A pop-up graph shows the range and amplitude output over time for selected sensors. Each sensor is displayed on a separate plot. Each line on a plot is a different target.

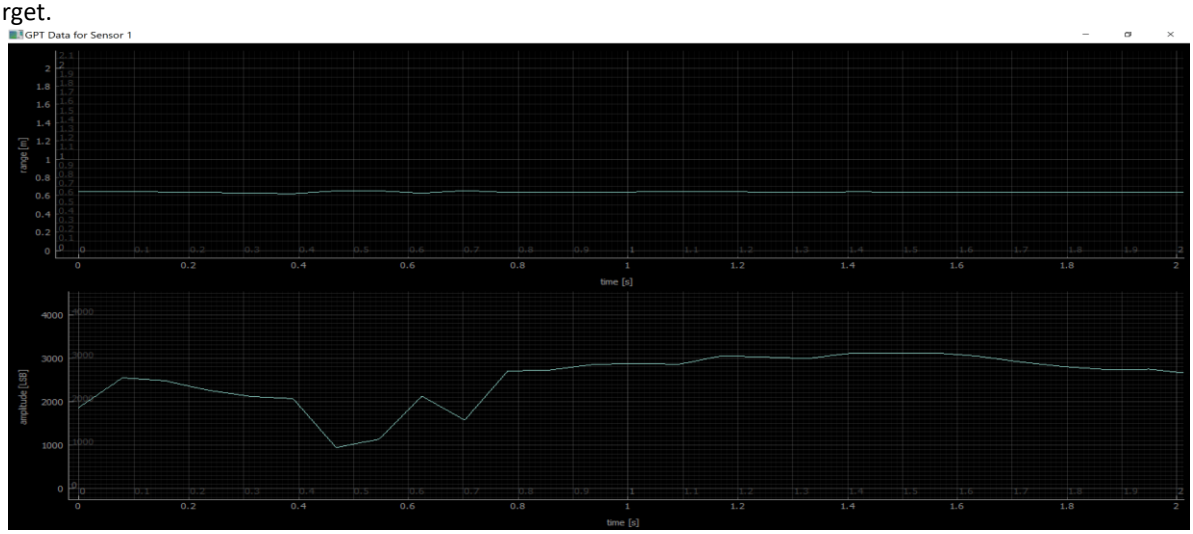


Figure 21: Range and Amplitude for Sensor 1

Note: Range can be defined/fixed by adjusting the RX sample accordingly, see section **4.1** below.

Supply Current

The supply current **11** graph shows the data of the supply current consumption for each connected sensor over time.
Note: Since the EVK is not calibrated, this feature is used for evaluation purposes only; in particular, the supply current measurement may include some offset error which makes it difficult to measure small currents (<20µA) accurately.

3.2 TRIGGER SETTINGS

Sensor trigger rate

The sensor trigger rate **12** control changes the rate at which measurements are triggered, effectively setting the output data rate. This control has no effect in the *self-timed* mode. The sensor trigger rate control will enforce an upper limit that depends on the current maximum range setting. Reducing the sensor trigger rate reduces the average current consumption while increasing the latency for detection.

3.3 DATA RECORDING AND ANNOTATION

Record

The Record **6** section is used to record incoming sensor data to CSV files:

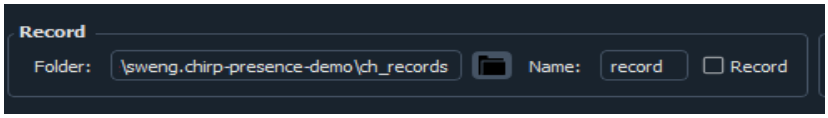
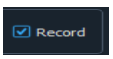


Figure 22: Recording the output

To create a new record and start to save data, set the *Record* checkbox: 

Record files are saved in the specified folder, prefixed with the configured record name and suffix with the date/time of when the record starts. Use the Browse button to select the folder where you want to save the data.

Annotation

The Annotation 7 buttons are used to annotate sensor data in the CSV files. When record is active, you can add annotation into the record, to remember of a particular event, (e.g.: approaching object, removing object, ...), using the *Annotation* section:

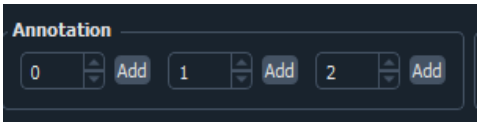


Figure 23: Annotation

When clicking 'Add', the specified value will be added to the record CSV file, under the 'annotation' column.

3.4 THE WIDGETS

The Widgets 8 section is used to launch special purpose widgets with more complex configuration options.

3.4.1 Configure Sensor

The Configure Sensor A (see Figure 20) in the widget section launches a dialog box, on that you can set and specify different sensor parameters.

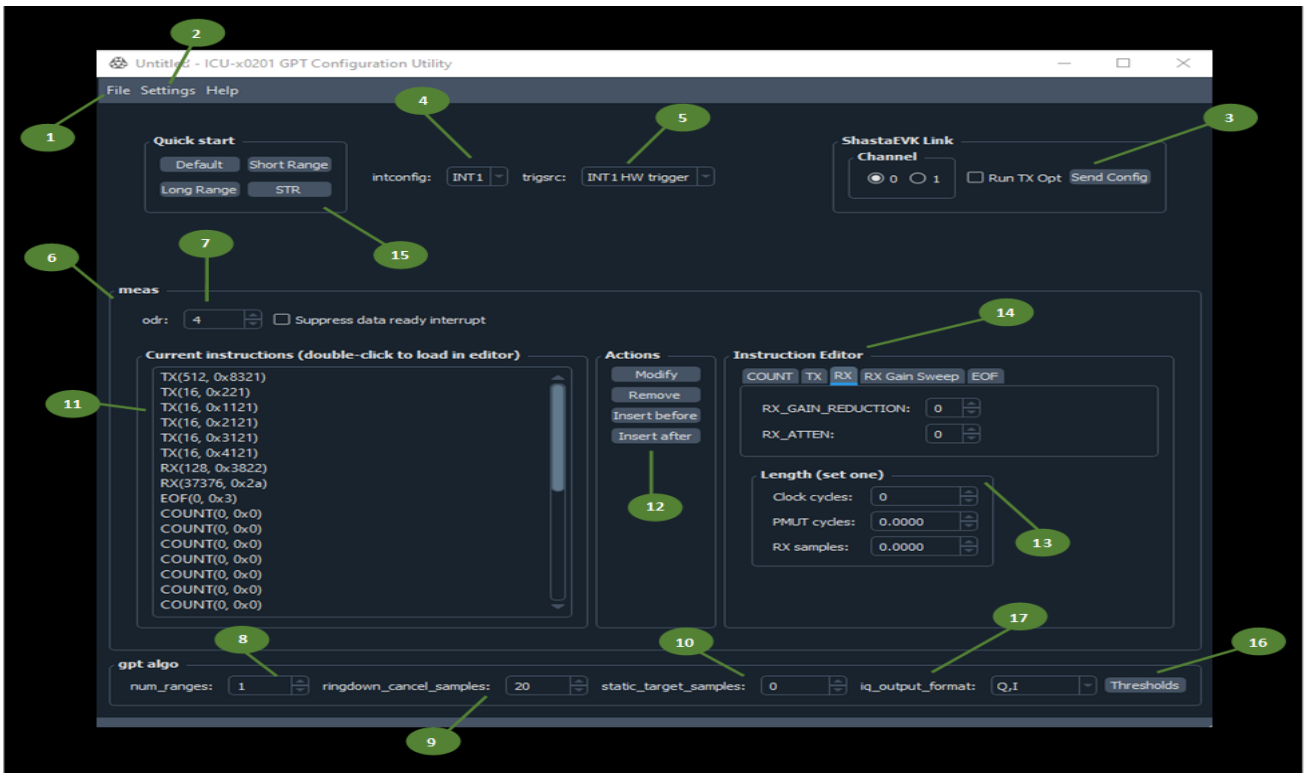


Figure 24: Configuring the sensor

File

1 Clicking the file menu opens options to open or save the sensor configuration JSON files (refer to QUICK START). The configuration can also be exported to a C file, which is consumed by the ICU-x0201 embedded driver (SonicLib). The currently open configuration is listed in the title bar.

Settings

2 Settings contain an option for enabling *Developer mode as* well as selecting *algo plugin* mode.

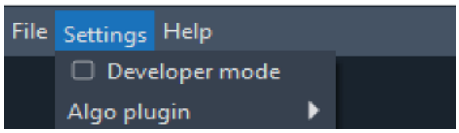


Figure 25: Settings

→ **Algo plugin:** The algo plugin setting allows you to edit configurations for different algo plugins. This will automatically be set to the correct setting upon loading new ASIC firmware. You can change it manually, but you will be unable to send the configuration to the sensor until you load the correct ASIC firmware.

You must load the correct ASIC firmware before switching to a different algo plugin in the configuration widget. If you do not load the ASIC firmware, you will be unable to send the configuration to the sensor. You will still be able to edit and save the configuration. Additional supported plugin firmware is located in invn/pysonic/plugins. The provided algorithm example also includes plugin support.

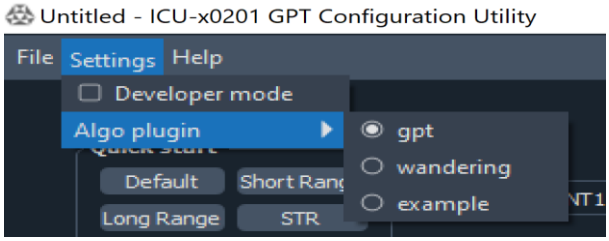


Figure 26: Algo plugin

→ **Developer mode:** The developer mode exposes settings that should not normally be changed. We suggest not using this option unless at the instruction of TDK Invensense.

Send Config

3 Clicking the Send Config button sends the sensor configuration to the sensor. If the *Run TX Opt* box is checked, the transmit optimization will be run in order to improve the short-range performance. This should only be run once after a new configuration is sent to the sensor. To run again, reload a fresh configuration, or delete the added transmit instructions. You will see these as a sequence of several short TX instructions after the main transmit.

intconfig

4 Through interrupt configuration setting you get to choose which interrupt pin ICU-x0201 will use to indicate data ready back to the host.

Trgsrc

5 Select the trigger source through this setting. The ICU-x0201 sensor requires a measurement trigger, the trigger is used to start an ultrasonic measurement. The measurement is defined by the measurement configuration, e.g., produced by this utility. A measurement consists of executing all *State machine instructions* (15) and then generating a data ready interrupt according to the *Interrupt configuration* (4). There are several possibilities for the trigger source:

- ↘ **SW trigger:** Measurements are triggered by writing a command via SPI.
- ↘ **INT1 HW trigger:** Measurements are triggered by pulsing the INT1 pin low.
- ↘ **INT2 HW trigger:** Measurements are triggered by pulsing the INT2 pin low.
- ↘ **Self-timed:** Measurements are triggered by the internal RTC (approximately 30 kHz). See the *Measurement period setting* info below.

Note that when configuring the trigger source through the evaluation software, the software will automatically reconfigure how it triggers the sensor to match. You can verify this using an oscilloscope or logic analyzer.

meas_period

6 Measurement period setting applies only when the trigger source is self-timed. It is the number of RTC cycles between triggers. The RTC is approximately 30kHz. This setting is only visible when the trigger source setting is self-timed.

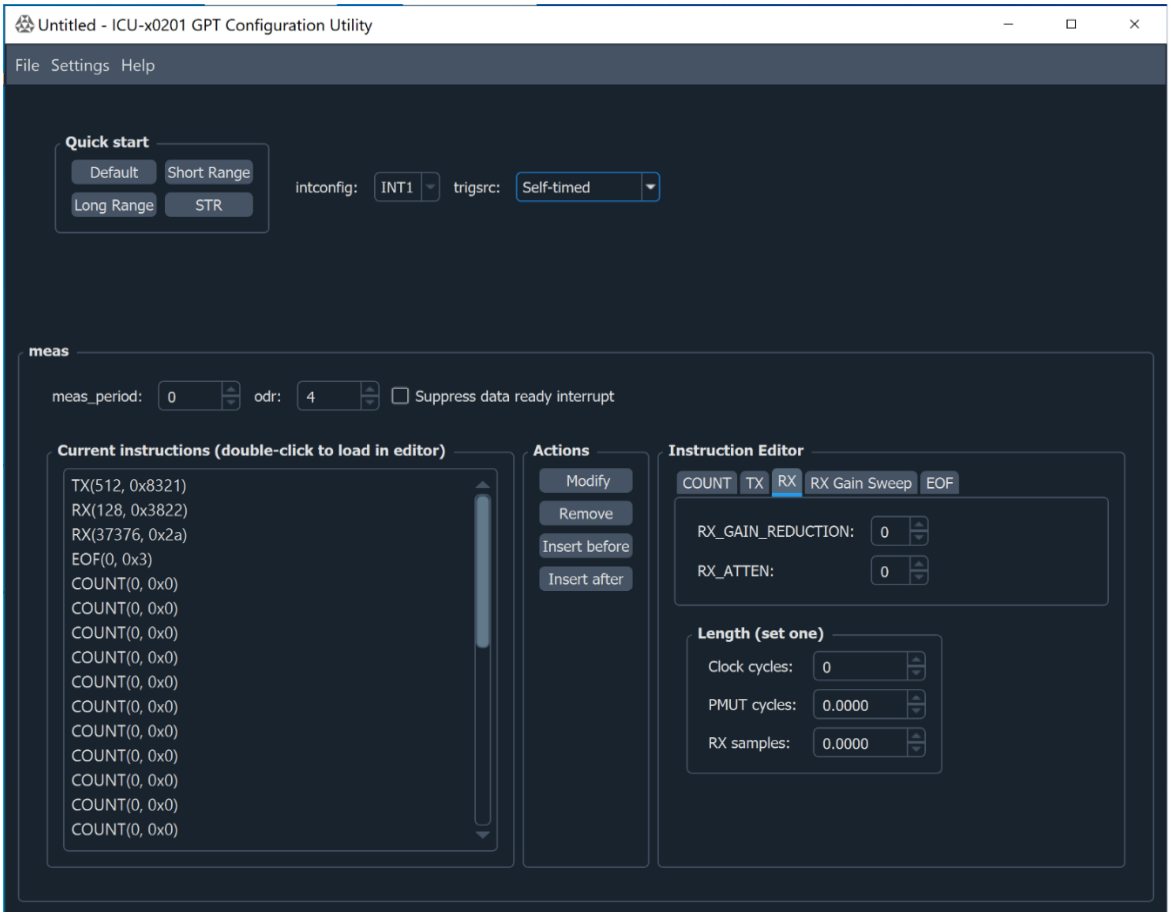


Figure 27: Settings when trgsrc is Self-timed

ODR

7 ODR is the Output Data Rate, this setting controls the data rate of the range-axis. It can be reported in either Hz or 1/m, which are related by the speed of sound. This should not be confused with the range data rate, or number of range samples per second. Each range requires a capture of the A-scan. An A-scan capture can be thought of as a frame. Then the range data rate is the inter-frame data rate, or the number of frames per second. In contrast, the ODR is the intra-frame data rate, or the number of ultrasonic amplitude samples per second (or samples per meter).

The rate is set relative to the PMUT operating frequency, typically in the range 70-95 kHz for ICU-20201. To convert between the sample rate in Hz and the sample rate in 1/m, divide by the speed of sound (343 m/s at room temp). See the table below for the sample rates.

CIC_ODR	RATE
2	fop/32
3	fop/16
4	fop/8
5	fop/4
6	fop/2

Current instructions

11 State machine instructions; these are the instructions associated with the measurement configuration. These instructions are executed by a special purpose state machine in the ICU-x0201 ASIC. The ASIC has an ultrasonic transducer specific state machine that executes instructions from on-chip memory. There are four available instructions: COUNT, TX, RX, and EOF. The sensor will execute the instructions sequentially, then produce a data ready interrupt as specified by the *interrupt configuration*. Note that not any arbitrary sequence of instructions is sensible. Generally, instructions should conform to the following pattern:

- Any number of COUNT and TX instructions intermixed
- At least one RX instruction
- Any number of COUNT and RX instructions intermixed
- EOF

Other sequences are possible but may require special support from the ICU-x0201 firmware. Additionally, extreme caution should be exercised in the *Developer mode*, as it exposes settings that can create invalid measurement sequences that may cause the sensor to hang.

Actions

12 These are instruction editor action buttons, used to modify the state machine instructions. Typical usage of these buttons is as follows:

- Double click an instruction to load it into the editor.
- Modify the instruction. You can change the type of the instruction by selecting a different tab (18).
- Click *Modify* to replace the selected instruction with the current configuration in the editor. You can also click either *Insert before* or *Insert after* if you wish to insert the instruction rather than replace an existing one.
- Click *Send Config* to send the new state machine instructions to the sensor.

Instruction Editor

14 Instruction type setting, these tabs are used to select the type of state machine instruction.

→ **COUNT:** The count instruction does not transmit nor receive. It is useful for creating delays.

→ **TX:** The *TX* instruction is used to configure an ultrasonic transmit. The transmit waveform is a square wave with programmable positive pulse width and phase. Each TX instruction can specify a pulse width and phase.

↘ **PHASE:** Controls the TX phase. This is the phase of the transmit waveform relative to the internal clock generator. Valid values are from 0-15. Each unit of phase is 1/16th of a cycle at the operating frequency, or 22.5°.

↘ **PULSE_WIDTH:** Controls the TX pulse width. This is the length of the positive pulse of the TX waveform. Making it shorter will reduce the effective drive strength. Valid values are from 0-4. Each unit of pulse_width is 1/8th of a cycle. Setting the pulse width to 0 will result in zero output for the TX instruction. The table below shows the theoretical duty cycle and amplitude vs the pulse width (PW) setting.

PW	Duty Cycle	Amplitude rel. PW=4
0	0	0
1	0.125	0.383
2	0.25	0.707
3	0.375	0.924
4	0.5	1

- **RX:** The *RX* instruction is used for ultrasonic reception. The gain is configurable through the RX_GAIN_REDUCTION and RX_ATTEN settings.
  - **RX_GAIN_REDUCTION**: The amount to reduce the receiver gain in dB to prevent saturation. Valid values are from 0-31 dB, but values 28-31 all result in the same reduction of approximately 28dB.
  - **RX_ATTEN**: Controls the receiver attenuator and is used to prevent saturation. Prefer to use rx_gain_reduction before resorting to rx_atten. Each increment of rx_atten (starting from zero) will drop the effective receiver gain by 1/8th. See the table below.

rx_atten	atten factor
0	1
1	1/8
2	1/64
3	1/512

- **RX Gain Sweep:** The RX Gain Sweep allows to linearly ramp the RX gain as time since transmission increases. Both Start gain reduction and stop gain reduction are the amounts to reduce the receiver gain in dB to prevent saturation; valid values are from 0-31 dB, with values between 28-31 dB all result in the same reduction of approximately 28 dB. Note that the value of Start gain reduction should be greater than or equal to the value of Stop gain reduction.
- **EOF:** The EOF instruction is used to indicate the end of a measurement sequence. This should always be the last instruction.

Length (set one)

13 Instruction length setting, this is used to set the length of the selected instruction. Each instruction except for EOF has a programmable length. Use the *Length* section (17) to configure this. The length can be set in three different units: *Clock cycles*, *PMUT cycles*, and *RX samples*.

- **Clock cycles**: Set the length in terms of state machine clock cycles (SMCLK cycles). There are 16 SMCLK cycles per one transducer clock cycle.
- **PMUT cycles**: Set the length in terms of transducer clock cycles. The transducer clock frequency is normally equivalent to the transducer resonant frequency.
- **RX samples**: Set the length in terms of the number of RX samples. The relationship between RX samples and the former two parameters depends on the CIC ODR. The total allowable RX samples depend on the ICU-x0201 FW used. The default GPT FW has a maximum allowable RX samples of 340.

  You are only required to set one of the above. The other fields will automatically update.

num_ranges

8 Number of ranges is the maximum number of targets that ICU-x0201 will report. Must be set to at least 1 to get range measurements. The maximum value of num_ranges is 5.

Ringdown_cancel_samples

9 This is the number of A-scan samples to include in the ringdown filter. The ringdown filter is used to remove the signal due to transducer recovery from transmit. If set to 0, there will be no ringdown cancellation. The maximum value of this variable is 60.

Static_target_samples

10 This is the number of samples to include in the static target rejection filter. Frequently, it is useful to ignore objects in the sensor's field of view which are static, or stationary. This can be accomplished by enabling the static target rejection filter. This utility allows

for configuring the number of A-scan samples to include in the static target filter. A value of 0 disables the filter entirely. Values greater than zero enable the filter, including as many samples as specified.

iq_output_format

17 IQ output format, this changes the format of the output data. Default is IQ, which gives the raw data as in-phase (I) and quadrature (Q) components. If changed to magnitude, the sensor will output magnitude data (i.e., square root of sum of squares of I and Q values). However, the GUI will not support this mode and it should only be used for data recording purposes.

Thresholds

16 Click the *Threshold* tab near the *state machine instructions* list to open the threshold configuration utility. The utility allows for configuring a threshold that changes with range (sample index). See below for usage.

Figure 28: Setting the Threshold

→ A **stop_index:** One after the last index to which the threshold applies. For example, if stop index is 20, the last index the threshold applies to is 19.

→ B **threshold:** The threshold to set up to the corresponding *stop index*.

→ C **Plot:** A plot that visualizes the threshold.

Note that the threshold is also displayed as an overlay on the A-scan plot in the EVK software. The overlay is plotted vs range instead of sample number. The threshold will update on the A-scan plot after clicking Send Config

○ Wandering - Algo Plugin

This application-specific algorithm is designed to interrupt a host processor when at least **event_window_threshold** events have occurred in the last **event_window_len** measurements. Typically used in conjunction with static target rejection mode, this algorithm can be used: a) when an interrupt should only be generated after an end user has been "wandering" in front of the sensor for several seconds, or b) when false negatives are unacceptable and the sensor should have a very high confidence in the presence of a target before waking the host processor.

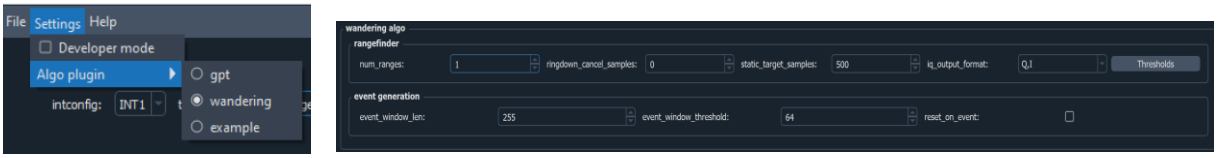


Figure 29: Wandering algo

- **event_window_len:** number of samples of history to accumulate detection events over. By dividing **event_window_len** by the sample rate, the window length in seconds can be calculated. This is the amount of time over which detection events will accumulate.
- **event_window_threshold:** number of detections needed in the last *event_window_len* samples to trigger a wandering event. By dividing **event_window_threshold** by the sample rate, the minimum latency before an interrupt is generated (assuming each measurement has a detectable target) can be calculated.
- **reset_on_event:** defines the action to take when a wandering event is generated. By default, the event count continues accumulating when it crosses the event threshold. If *reset_on_event* is set to 1, then the event count will reset to 0 on each wandering event

○ Example – Algo Plugin

You can define your own algo plug in here if you develop your own algorithm.

- **baz, qux:** Indicates your customized variable that you want to add to the GUI Plugin.

Figure 30: Example plugin

Quick Start

These buttons 15 are used to load pre-defined settings optimized for several typical use-cases. After loading settings, they can be customized and saved to a JSON file. There are four templates provided with the configuration utility:

- **Default:** The default settings for general purpose rangefinding.
- **Short range:** Settings optimized for short range measurements.
- **Long range:** Settings optimized for long range measurements.
- **STR:** Settings optimized for static target rejection.

Click a button to load a template, then edit it to fit your application needs.

Note: You can also save the customized settings and import them at a later instance. An example of it is presented below.

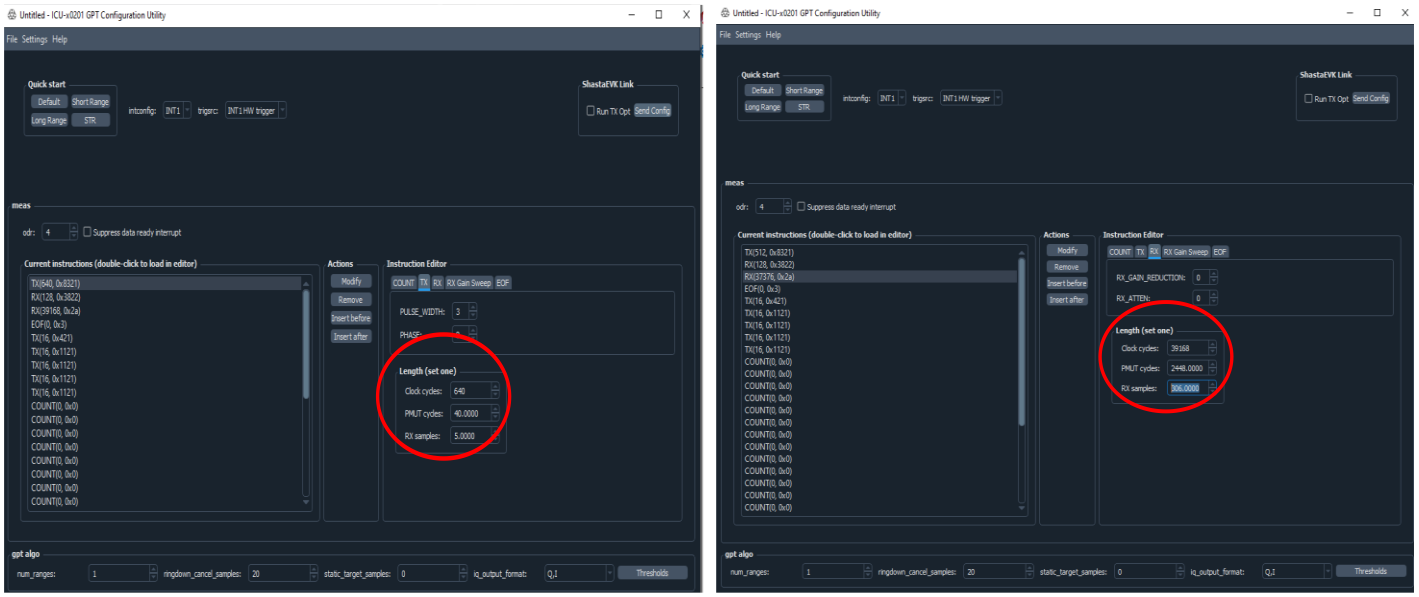


Figure 31: Changing the RX, TX samples (example)

Click on Modify after the customization for each field, that will implement the change.

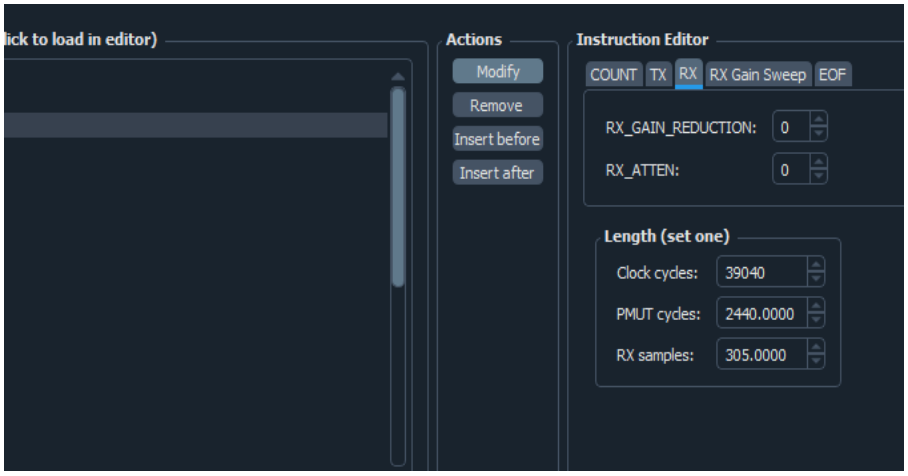


Figure 32: Modifying

Then save the file in the directory of your choice.

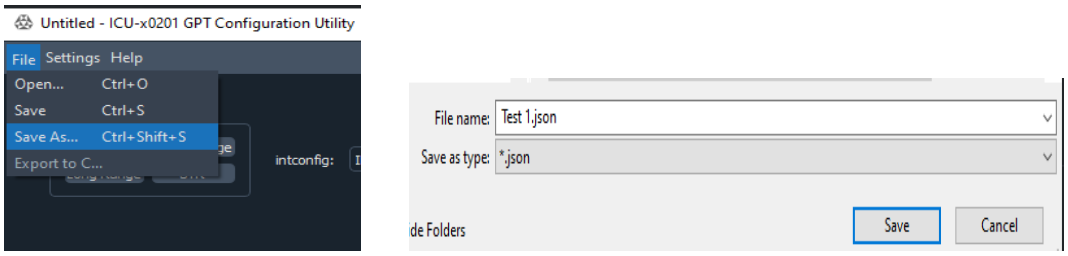


Figure 33: Saving the settings

At any time, the file can be retrieved by opening it back from the location it was saved at.

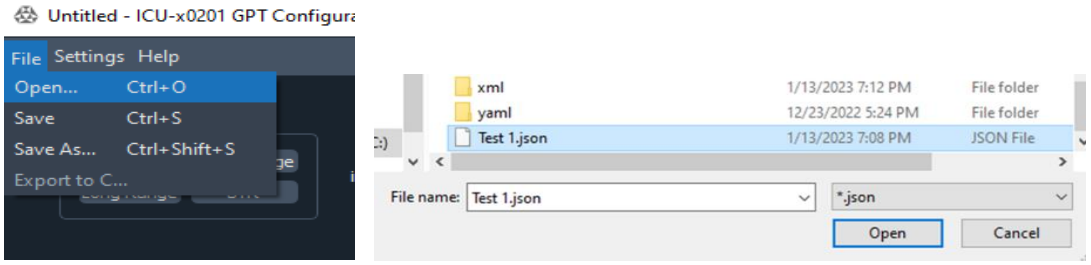


Figure 34: Reopening the file

Then, click the Send Config button to upload the custom template.

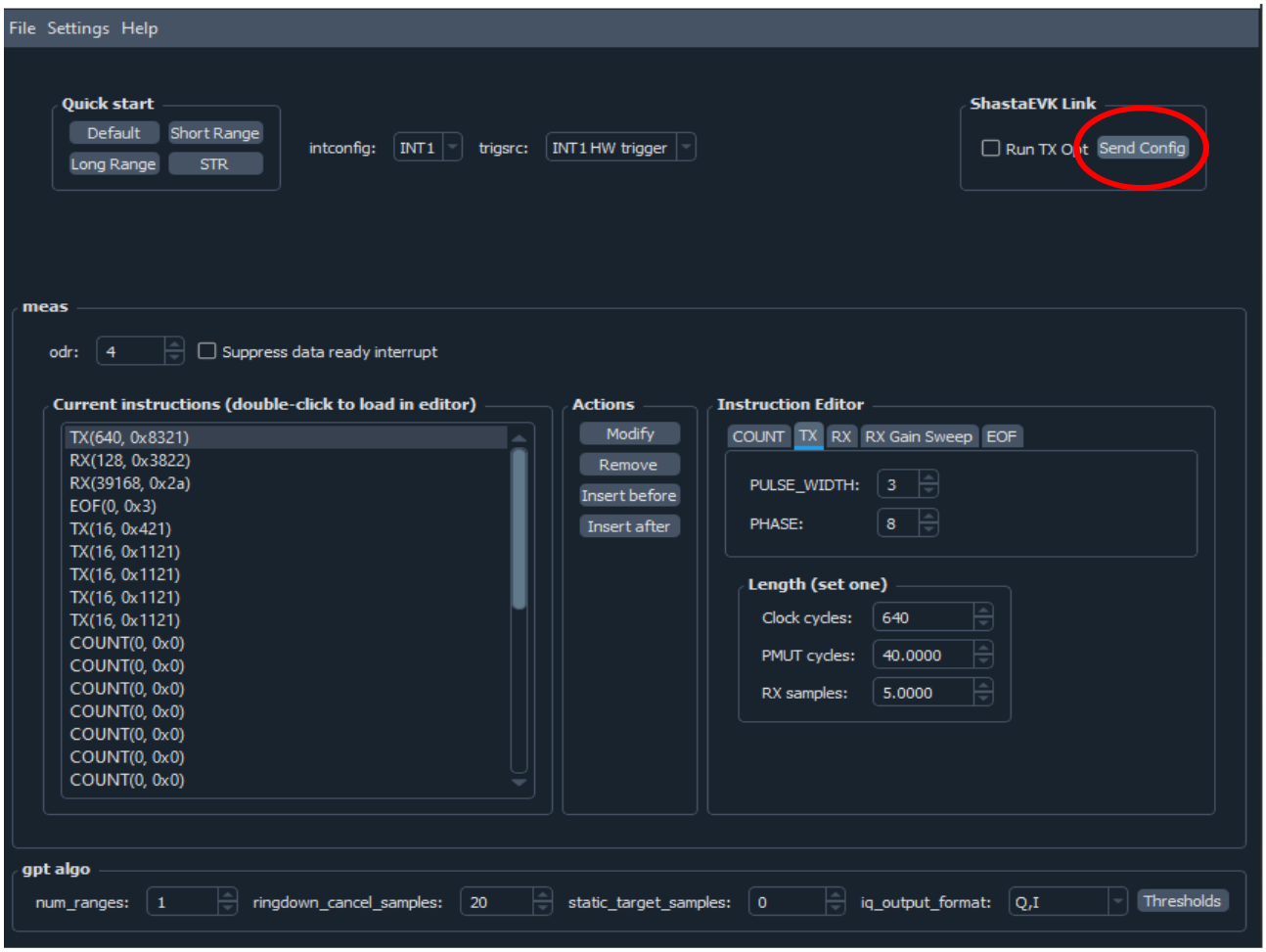


Figure 35: Settings reloaded

3.4.2 Build and Load ASIC Firmware

With Build Firmware **B** (see Figure 20), you can create custom plugins that work with the ICU EVK software to display data from different ASIC application firmware. The procedure is described below:

i. Develop your custom ASIC algorithm using the open-source algorithm support in this EVK.

→ **Building the example**
You can program your own custom algorithms using the ICU-x0201 EVK. To get started, launch the EVK and click the "Build Firmware" button, pictured below.

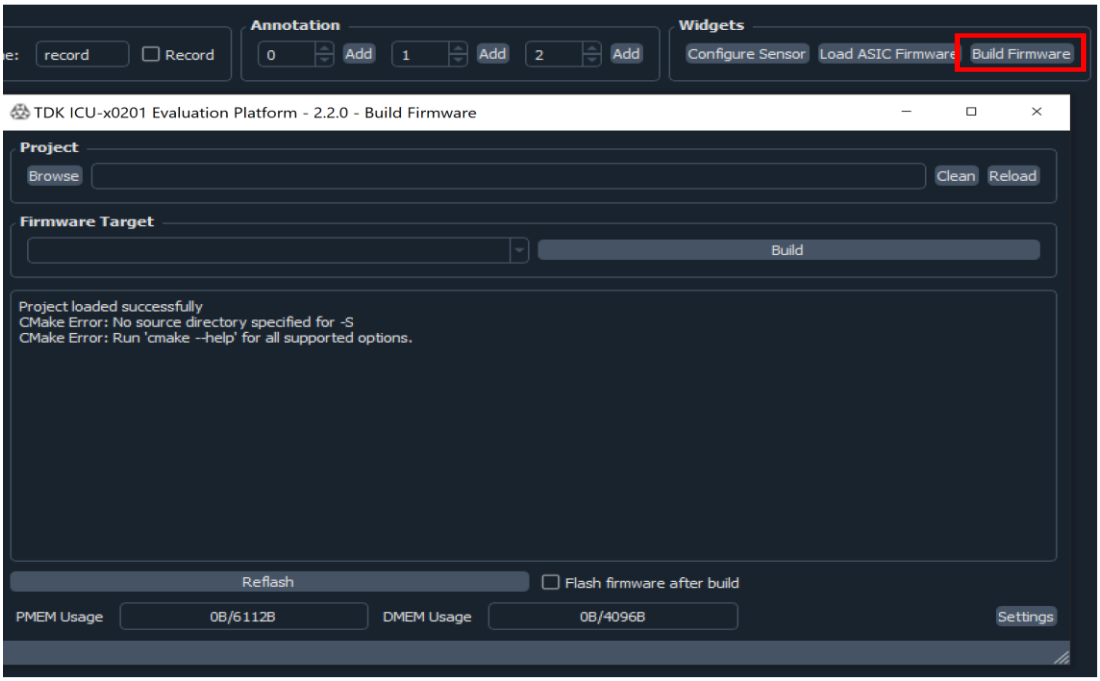


Figure 36: Building Firmware

Click the "Browse" button in the window that opens and navigate to the included workspace. It is in the ICU-x0201 EVK install directory and is called icu_open_algo. Navigate to this folder, then click "Select Folder" to set this as the active workspace.
Next, click the "Build" button. This should print some text to the console. You should see the build complete successfully.
Next, click "Reflash". This will flash the example on to the ASIC.
This illustrates the process of building and flashing. Next we'll cover how to modify the example to support your algorithm.

→ **Modifying the example**

You can copy the icu_open_algo directory out of the EVK directory and place it elsewhere. Or, you can choose to work out of the directory in place. This will serve as the workspace for all new algo projects. The example code itself is well documented. You should read through the algo/example/src/main.c, algo/example/src/example_algo.h, and include/app_core.h files to get an idea of what you need to do. A high level summary is as follows:

- Create a copy of the algo/example directory and rename it to your liking, placing it next to example in the algo directory. We will refer to our custom project as foo.
- Modify the user defined algo data structures in foo/example_algo.h. You can also rename this file, e.g., foo_algo.h.
- Modify foo/main.c to include the algo header from the previous step. If you changed the name of the algo structs, update the typedefs near the top of main.c.
- Implement your algorithm in the provided template functions algo_update, algo_init, algo_event_get, and algo_event_set.

→ **Building and loading your firmware**

Follow the same steps as for building the example, but make sure to navigate to the new project directory if you moved it. Also, select the appropriate firmware target from the drop-down box. Note that before flashing your firmware, you will need to create the pysonic plugins file.

→ **Creating the default algo configuration**

You will need to create a JSON settings file that contains the default algo configuration.
The easiest way to do this is to copy icu_open_algo/algo/example/example_defaults.json to your project directory, e.g., algo/foo, rename it appropriately (foo_defaults.json), and then make some modifications to it.
After making a copy of the settings file, open it up in a text editor. Find the entry algo_name and change it to your algo name, e.g., foo.

Next, modify the contents of algo_cfg to provide default values to the members of your algo configuration struct. You can see in the example JSON file, we are setting the baz and qux members to zero. These are the members defined in example_algo.h.

ii. **Create a new entry in a PysonicPlugins.toml file** for your algorithm.

Support for new ASIC firmware can be added via a plugin interface that is loaded at run-time. The plugin configuration is read from a file named `PysonicPlugins.toml`. We search the following locations for this file:

- The current working directory.
- The pysonic* package directory.
- The directory defined by `sys.executable`. This will be the directory containing the main executable for packaged applications (`ICU-x0201_EVK.exe`).

* Pysonic (invn.pysonic) is a python package developed by TDK Invensense that supports communication with ICU-series sensor hardware. The file specifies the location of the headers that define the algorithm config and output structs. It also specifies a search pattern used to match ASIC firmware to a particular plugin implementation. See the example below, which was created to support the example algorithm.

```
# Pysonic plugin configuration

[plugins.example]
# A list of strings to match against the firmware name. You can use the glob
# patterns with the character *. If the firmware matches the pattern, then
# this plugin is used with said firmware.
use_with_fw = ["*example*.hex"]
# The path to the header containing the algorithm struct definitions. This may
# either be an absolute path or a relative path. If a relative path, the
# directories searched are: the directory containing this file,
# the current working directory, the pysonic directory, sys.executable
# (path to the main exe).
header_path = """
icu_open_algo/algo/example/src/example_algo.h"""
# The default sensor configuration JSON file to use with this plugin. This file
# MUST contain an entry "algo_name" that matches that supplied here. The
# directory search strategy is the same as header_path.
json_defaults = "icu_open_algo/algo/example/example_defaults.json"
# The name of the algo config struct defined in the header file. This is the
# name in the struct namespace (i.e., not the typedef name).
config_struct_name = "user_algo_config"
# The name of the algo output struct defined in the header file. This is the
# name in the struct namespace (i.e., not the typedef name).
output_struct_name = "user_algo_output"

[plugins.foo]  # another plugin
...
```

Pysonic will read the plugin files when the package is imported into an application. After loading, the full path names of the header and JSON files are available in the global invn.pysonic.plugins_metadata dictionary. The following are defined for each plugin_name:

- plugins_metadata[plugin_name]["plugin_config_path"]: The path to the plugin file loaded.
- plugins_metadata[plugin_name]["header_path"]: The full path to the header file associated with the plugin.
- plugins_metadata[plugin_name]["json_path"]: The full path to the JSON defaults file associated with the plugin.

External dependencies can use this to read in pysonic plugin configuration data directly if needed.

For information on the toml file format, see the toml website.

iii. **Create your EVK plugin** by developing a new class derived from IcuPlugin.

This step assumes the reader is familiar with the Python programming language, which is used to extend the EVK functionality.
In this step, you will create the actual display widgets that will be loaded by the EVK software to visualize your plugin's output data. To do this, you will implement a custom class that derives from IcuPlugin. The IcuPlugin interface is given below. Note that your Python file will be loaded into the EVK's runtime environment, which will make the base IcuPlugin class available as long as your import statements match the example which follows.
Here is the IcuPlugin interface:

```
class IcuPlugin(ChDataSubscriber):
    """Base class for ICU EVK plugin creation."""
    def enable_event(self):
        """Called to enable the plugin.

        Child classes should override this method with whatever startup
        code they require."""
        pass

    def disable_event(self):
        """Called to disable the plugin.

        Child classes should override this method with whatever shutdown
        code they require."""
        pass

    def data_ready_event(self, timestamp, algo_data, tx_sensor_id,
                         rx_sensor_id, m_per_range_lsb, pmut_frequency,
                         **kwargs):
        """Called when data is ready.

        Child classes should override this method to catch new data.

        Arguments:
        timestamp -- The timestamp in microseconds.
        algo_data -- The algo output data as a dict. Keys correspond to names
                     of fields in the algo out struct.
        tx_sensor_id -- The ID of the transmitting sensor.
        rx_sensor_id -- The ID of the receiving sensor.
        m_per_range_lsb -- The distance represented by 1 LSB of range in
                           meters.
        pmut_frequency -- The PMUT operating frequency in Hz.
        """
        pass

    def get_connected_sensors(self):
        """Return a list of connected sensors.

        For example, if sensor sites 0 and 2 are populated on the EVK board,
        the returned list will be [0, 2].
        """
        return self._connected_sensors
```

Your class should inherit from `IcuPlugin`, overriding the methods as needed. An implementation supporting the algo example is included with the EVK. See the icu_plugins/exampleplugin.py script.
As additional reference, the following is the implementation of the GPT plugin, which shows range and amplitude data. All modules imported by the example are available for your custom plugin to use as well. Pyqtgraph is used to create the plots for data display. This is a general purpose, open-source graphing library with extensive documentation available here.

```
"""ICU EVK plugin for display of range and amplitude data from the GPT
firmware."""

__copyright__ = "Copyright (c) TDK Invensense, 2022"

from icuplugin import IcuPlugin
from plotcolors import get_color_dark
import logging
import pyqtgraph as pg
```

```
import numpy as np

logger = logging.getLogger(__name__)

BUF_SIZE = 200  # buffer this many range and amplitude readings
NUM_RANGES = 8  # maximum number of targets supported
NUM_SENSORS = 4  # maximum number of sensors supported

class GptSensorView(pg.GraphicsLayoutWidget):
    """A GraphicsLayoutWidget implementation for displaying GPT data for
    one sensor."""
    def __init__(self, sensor_id, *args, **kwargs):
        """Create all the plots and necessary buffers."""
        logger.info("begin GptPlugin.__init__")
        self.sensor_id = sensor_id
        super().__init__(*args, **kwargs)
        self.range_plot_item = self.addPlot(row=0, col=0)
        self.range_plot_item.addItem(pg.GridItem())
        self.amplitude_plot_item = self.addPlot(row=1, col=0)
        self.amplitude_plot_item.addItem(pg.GridItem())
        self.amplitude_plot_item.setXLink(self.range_plot_item)
        # one range, one ampitude data item for each potential target
        self.range_data_items = [
            pg.PlotDataItem(pen={'color': get_color_dark(i)},
                            skipFiniteCheck=True)
            for i in range(NUM_RANGES)
        ]
        self.amplitude_data_items = [
            pg.PlotDataItem(pen={'color': get_color_dark(i)},
                            skipFiniteCheck=True)
            for i in range(NUM_RANGES)
        ]
        for data_item in self.range_data_items:
            self.range_plot_item.addItem(data_item)
        for data_item in self.amplitude_data_items:
            self.amplitude_plot_item.addItem(data_item)
        self.range_plot_item.setLabel('left', 'range [m]')
        self.range_plot_item.setLabel('bottom', 'time [s]')
        self.amplitude_plot_item.setLabel('left', 'amplitude [LSB]')
        self.amplitude_plot_item.setLabel('bottom', 'time [s]')
        self.range_buf = [deque([0.] * BUF_SIZE, maxlen=BUF_SIZE)
                          for i in range(NUM_RANGES)]
        self.amplitude_buf = [deque([0] * BUF_SIZE, maxlen=BUF_SIZE)
                              for i in range(NUM_RANGES)]
        self.abs_time_buf = deque(maxlen=BUF_SIZE)

    def data_ready(self, timestamp, algo_data,
                   tx_sensor_id,
                   m_per_range_lsb, pmut_frequency, **kwargs):
        """Update the buffers and display."""
        # logger.debug(f"gpt {rx_sensor_id} {timestamp} {algo_data}")
        if len(self.abs_time_buf) == 0:
            self.abs_time_buf.extend([timestamp / 1e6] * BUF_SIZE)
        self.abs_time_buf.append(timestamp / 1e6)
        time_buf = [t - self.abs_time_buf[0] for t in self.abs_time_buf]
        target_list = algo_data['tL']
        targets = algo_data['tL']['targets']
        num_targets = target_list['num_valid_targets']
        for i in range(NUM_RANGES):
            if i < num_targets:
                self.range_buf[i].append(targets[i]['range'] * m_per_range_lsb)
                self.amplitude_buf[i].append(targets[i]['amplitude'])
            else:
                self.range_buf[i].append(0)
                self.amplitude_buf[i].append(0)
            if self.amplitude_buf[i].count(0) < BUF_SIZE:
                self.range_data_items[i].setData(time_buf,
                                                 self.range_buf[i])
                self.amplitude_data_items[i].setData(time_buf,
                                                     self.amplitude_buf[i])


class GptPlugin(IcuPlugin):
```

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.sensor_views = [GptSensorView(i, title=f"GPT Data for Sensor {i}")
                             for i in range(NUM_SENSORS)]

    def data_ready_event(self, timestamp, algo_data, tx_sensor_id,
                         rx_sensor_id, m_per_range_lsb, pmut_frequency,
                         **kwargs):
        """Call the data ready method for the appropriate sensor using the
        rx_sensor_id"""
        self.sensor_views[rx_sensor_id].data_ready(timestamp, algo_data,
                                                   tx_sensor_id,
                                                   m_per_range_lsb,
                                                   pmut_frequency,
                                                   **kwargs)

    def enable_event(self):
        """Called by the main application after connecting to device."""
        logger.info("gptplugin enable_event")
        for i in self.get_connected_sensors():
            self.sensor_views[i].show()

    def disable_event(self):
        """Called by the main application after disabling device."""
        logger.info("gptplugin disable_event")
        for sensor_view in self.sensor_views:
            sensor_view.hide()
```

Save your implementation as a python file. The configuration file created in the next step will allow you to provide the path to this file.

iv. **Create a new entry in an IcuPlugins.toml file** describing where to find your plugin from step 3.

In this step you will add a new entry to an IcuPlugins.toml file describing how to load your custom plugin. This file can be located in the same directory as ICU-x0201_EVK.exe. An example is below.

```
[plugins.example]
# Specify the name of the python class.
name = "ExamplePlugin"
# Specify the location of the python file. The current working directory and
# directory containing the main exe are searched.
location = "icu_plugins/exampleplugin.py"
# Specify which firmware to associate with this plugin. If the firmware name
# contains the provided string, then we associate this plugin with the
# firmware.
use_with_firmware = "example"
```

This example has already been added to the included IcuPlugins.toml. You can add to this file to support additional algorithm widgets.

v. **Create a sensor config widget plugin** (optional) to enable easy control of your algo configuration parameters.

This step is optional. Completion of this step will give you a set of custom controls in the Sensor Configuration Widget for setting your algorithm parameters. Working knowledge of Python and Qt is assumed.

To create the custom controls, you will need to implement a class derived from invn.shconfig.ShastaConfigPlugin. The interface of this class is below.

```
class ShastaConfigPlugin(ShastaConfigBase):
    def __init__(self, widget, mq_writer):
        """Create the UI by modifying widget.

        Here, you should create all of your UI elements. You should create
        a QLayout for widget, then add your UI elements to the layout.

        You should also connect the UI element signals to slots defined
```

```
        in this class. There is a default signal handler provided in the base
        class. The default handler will propagate the signal value to an
        algo configuration parameter matching `key`. For example,

        self.connect_to_dict_update(self.ui.num_ranges_spin_box.valueChanged,
                                    'num_ranges')

        will connect the value associated with the spin box valueChanged
        signal to the `num_ranges` key of the algo configuration dict.

        The algo configuration dict is available as `self.get_algo_cfg().mq`.
        """
        super().__init__(widget, mq_writer)

    def switch_to_device(self, dev_idx):
        """Make the device with index dev_idx the active configuration device.

        You may override this method if you need to intercept this call. You
        should do this when your plugin widget contains its own subwidgets,
        which also inherit from ShastaConfigBase. In this case, call
        switch_to_device on all subwidgets, and don't forget to call the
        base class method with super()."""
        super().switch_to_device(dev_idx)

    def display_error(self, error):
        """Display the error message error.

        Errors are generated when attempting to set illegal values into
        parameters. Override this method to change the error display behavior.
        """
        logger.warning(error)

    @abc.abstractmethod
    def update(self):
        """This method should update the UI from the backend data structure.

        Implementation will be dependent on your particular UI elements. For
        example, if you have a spin box, you will want to set the displayed
        value of the spin box to match what is in the algo configuration
        dictionary.
        """
        pass

    def set_mq_writers(self, mq_writers):
        """This method should update the config writers (previously called
        measurement queue writers) Normally, all that is required is calling
        `super().set_mq_writers(mq_writers)`.

        You may override this method if you need to intercept this call. You
        should do this when your plugin widget contains its own subwidgets,
        which also inherit from ShastaConfigBase. In this case, call
        set_mq_writers on all subwidgets, and don't forget to call the
        base class method with super()."""
        super().set_mq_writers(mq_writers)

    def set_meas_idx(self, idx):
        """This method switches the measurement index.

        Some algo configurations support multiple configurations. You should
        override this method when you need to propagate the set_meas_idx call
        down to subwidgets."""
        super().set_meas_idx(idx)

    def get_meas_idx(self):
        """Get the active measurement index.

        If your algo config doesn't use more than one index, you can simply
        return 0 from this method."""
        return super().get_meas_idx()

    @classmethod
    def create_c_writer(cls, filename=None):
```

```
        """This method needs to return an object for writing the C file from the JSON file filename.

        To implement this in a derived class, you can use the _create_c_writer method of this class. For example,      the
following will create the create_c_writer method for the FooConfig class.

        import functools

        class FooConfig(ShastaConfigPlugin):
            create_c_writer = functools.partial(ShastaConfigPlugin._create_c_writer, "foo")
            ...

        The algo name ("foo" in this case) should match the "algo_name" field of the JSON configuration file.
        This default implementation returns None. The export to C option will not be available."""

        return None
```

An implementation supporting the example algo is provided in icu_plugins/exampleconfig.py.
Additionally, the GPT implementation of ShastaConfigPlugin is given below. The implementation of the threshold widget is omitted. The purpose is to show when you might need to override some of the optional methods, such as set_mq_writers, set_meas_idx, etc. Likely your implementation will not require sub-widgets; in this case, you do not need to provide implementations of these methods.

```
class Ui_gptAlgoWidget:
    def setupUi(self, widget):
        self.layout = QtWidgets.QHBoxLayout(widget)

        self.num_ranges_label = QtWidgets.QLabel(widget)
        self.num_ranges_label.setText("num_ranges:")
        self.layout.addWidget(self.num_ranges_label)

        self.num_ranges_spin_box = QtWidgets.QSpinBox(widget)
        self.num_ranges_spin_box.setMinimum(0)
        self.num_ranges_spin_box.setMaximum(65535)
        self.num_ranges_spin_box.setValue(1)
        self.layout.addWidget(self.num_ranges_spin_box)

        self.ringdown_cancel_label = QtWidgets.QLabel(widget)
        self.ringdown_cancel_label.setText("ringdown_cancel_samples:")
        self.layout.addWidget(self.ringdown_cancel_label)

        self.ringdown_cancel_spin_box = QtWidgets.QSpinBox(widget)
        self.ringdown_cancel_spin_box.setMaximum(65535)
        self.ringdown_cancel_spin_box.setValue(20)
        self.layout.addWidget(self.ringdown_cancel_spin_box)

        self.static_target_label = QtWidgets.QLabel(widget)
        self.static_target_label.setText("static_target_samples:")
        self.layout.addWidget(self.static_target_label)

        self.static_target_spin_box = QtWidgets.QSpinBox(widget)
        self.static_target_spin_box.setMaximum(100000)
        self.layout.addWidget(self.static_target_spin_box)

        self.iq_output_format_label = QtWidgets.QLabel(widget)
        self.iq_output_format_label.setText("iq_output_format:")
        self.layout.addWidget(self.iq_output_format_label)

        self.iq_output_combo_box = QtWidgets.QComboBox(widget)
        self.iq_output_combo_box.addItem('Q,I', 0)
        self.iq_output_combo_box.addItem('Mag,Thresh', 1)
        self.iq_output_combo_box.addItem('Mag,Phase', 2)
        self.layout.addWidget(self.iq_output_combo_box)

        self.thresholdsButton = QtWidgets.QPushButton(widget)
        self.thresholdsButton.setText("Thresholds")
        self.layout.addWidget(self.thresholdsButton)

        self.num_ranges_spin_box.setToolTip(getdoc(RangeFinder.num_ranges))
        self.ringdown_cancel_spin_box.setToolTip(
            getdoc(RangeFinder.ringdown_cancel_samples))
```

```
        self.static_target_spin_box.setToolTip(
            getdoc(RangeFinder.static_target_samples))
        self.iq_output_combo_box.setToolTip(
            getdoc(RangeFinder.iq_output_format))


class ShastaGptAlgoConfig(ShastaConfigPlugin):
    c_writer = None
    create_c_writer = functools.partial(ShastaConfigPlugin._create_c_writer, "gpt")

    def __init__(self, widget, mq_writer):
        self.ui = Ui_gptAlgoWidget()
        self.ui.setupUi(widget)

        self.threshold_win = QtWidgets.QWidget(widget, QtCore.Qt.Window)
        self.threshold_widget = ThresholdWidget(self.threshold_win,
                                                mq_writer)

        super().__init__(widget, mq_writer)
        self.connect_to_dict_update(self.ui.num_ranges_spin_box.valueChanged,
                                    'num_ranges')

        self.connect_to_dict_update(
            self.ui.ringdown_cancel_spin_box.valueChanged,
            'ringdown_cancel_samples')

        self.connect_to_dict_update(
            self.ui.static_target_spin_box.valueChanged,
            'static_target_samples')

        self.connect_to_dict_update(
            self.ui.iq_output_combo_box.currentIndexChanged,
            'iq_output_format')

        self.ui.thresholdsButton.clicked.connect(
            self.on_thresholds_button_clicked)

    def set_mq_writers(self, mq_writers):
        super().set_mq_writers(mq_writers)
        self.threshold_widget.set_mq_writers(mq_writers)

    def switch_to_device(self, dev_idx):
        super().switch_to_device(dev_idx)
        self.threshold_widget.switch_to_device(dev_idx)

    def set_meas_idx(self, meas_idx):
        super().set_meas_idx(meas_idx)
        self.threshold_widget.set_meas_idx(meas_idx)

    def display_error(self, error):
        # TODO: Connect to UI
        super().display_error(error)

    def update(self):
        ac = self.get_algo_cfg().mq
        self.ui.num_ranges_spin_box.setValue(ac['num_ranges'])
        self.ui.ringdown_cancel_spin_box.setValue(ac['ringdown_cancel_samples'])
        self.ui.static_target_spin_box.setValue(ac['static_target_samples'])
        self.ui.iq_output_combo_box.setCurrentIndex(ac['iq_output_format'])
        self.threshold_widget.update()

    def on_thresholds_button_clicked(self):
        self.threshold_widget.show()
```

In the final step, we will create another configuration file to tell the EVK application where to load the configuration plugin.

vi. **Create a new entry in ShastaConfigPlugins.toml** (only if you do 5) to point the application to your config widget implementation.

The last step is creating a configuration file to tell the EVK software where to find your custom configuration widget implementation. This file should be called ShastaConfigPlugins.toml and located in the same directory as ICU-x0201_EVK.exe or the current working directory. An example configuration file is given below. A default implementation is provided with the EVK as well.

```
# The config tool will launch with the "default" algo unless it is initialized
# with an existing configuration writer.
default = "gpt"

[algos]

# Each algo is defined as a member of [algos]. The algo name must match the
# algo_name field in the corresponding algo configuration JSON file.
[algos.gpt]
# The file path to the module where the plugin class is defined. The following
# locations are searched: the current working directory, the shconfig package
# directory, the directory containing the packaged executable.
module = "gptplugin.py"
# The class name of the plugin.
plugin = "ShastaGptAlgoConfig"
# If the JSON defaults are to be taken from an installed python package, then
# provide that package name here. Otherwise, set it to an empty string.
json_defaults_pkg = "invn.pysonic.shastagpt"
# If you provided defaults_pkg, then this should be the relative path from the
# package to the deafults file. Here, we are taking defaults from the pysonic
# package. If not providing a defaults package, this should be an absolute path
# or a relative path from the current working directory.
json_defaults = "cfg_json/defaults.json"

[algos.wandering]
module = "wanderingplugin.py"
plugin = "WanderingConfig"
json_defaults_pkg = "invn.pysonic"
json_defaults = "plugins/wandering_defaults.json"

[algos.example]
module = "icu_plugins/exampleconfig.py"
plugin = "ExampleConfig"
json_defaults = """
icu_open_algo/algo/example/example_defaults.json"""
```

4 ADDITIONAL INFORMATION

4.1 MODIFYING THE RANGE

To modify the range of sensing, it is necessary to act on the RX number of samples in the configuration sensor widget.

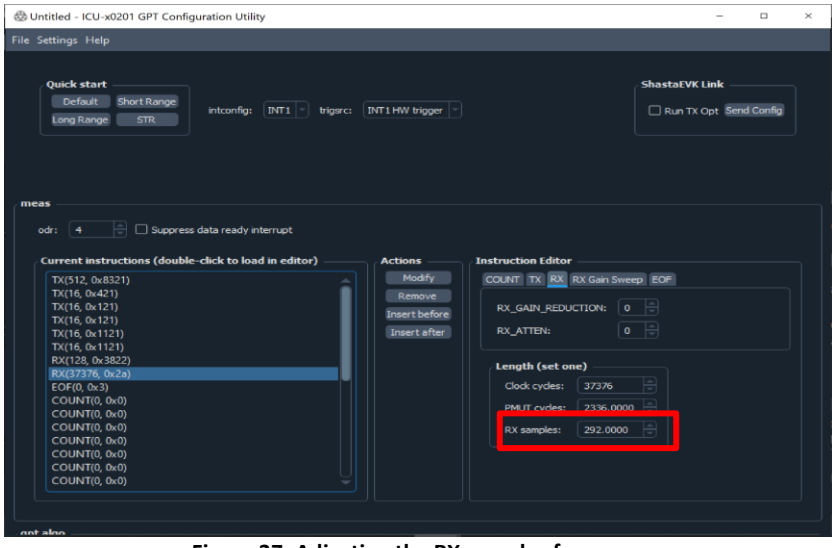


Figure 37: Adjusting the RX samples for range

The relationship between the number of samples and the range in meter is done by the following formula:

Range in m = no. of samples * distance measured by a single sample

= no. of samples * ODR / fs * Speed of Sound/ 2

Figure 38: Formula for defining the range

By default, the number of samples is set to 292, when the cic_ODR is set to 4. That means that the correspondent ODR is 8 and the range measured by the device is:

Range = 292 * 8 / 80000 [Hz] * 343 [m/sec] / 2 = 5 m

To set the number of samples based on the desired range it is required to use the inverse formula:

no. of samples = Range in m / distance measured by a single sample

= Range in m * fs / ODR * 2 / Speed of Sound

For example, in order to set 3m range with cic_odr = 4, it is necessary to set RX sample at the following value:

Rx sample = 3 [m] * 80000 [Hz] / 8 * 2 / 343 [m/sec] = 175

In the configuration sensor widget it is possible to modify the latest RX instruction before the EOF, changing the RX samples field to 175 [1], push the Modify Action button [2], and finally press the Send Config Button [3].

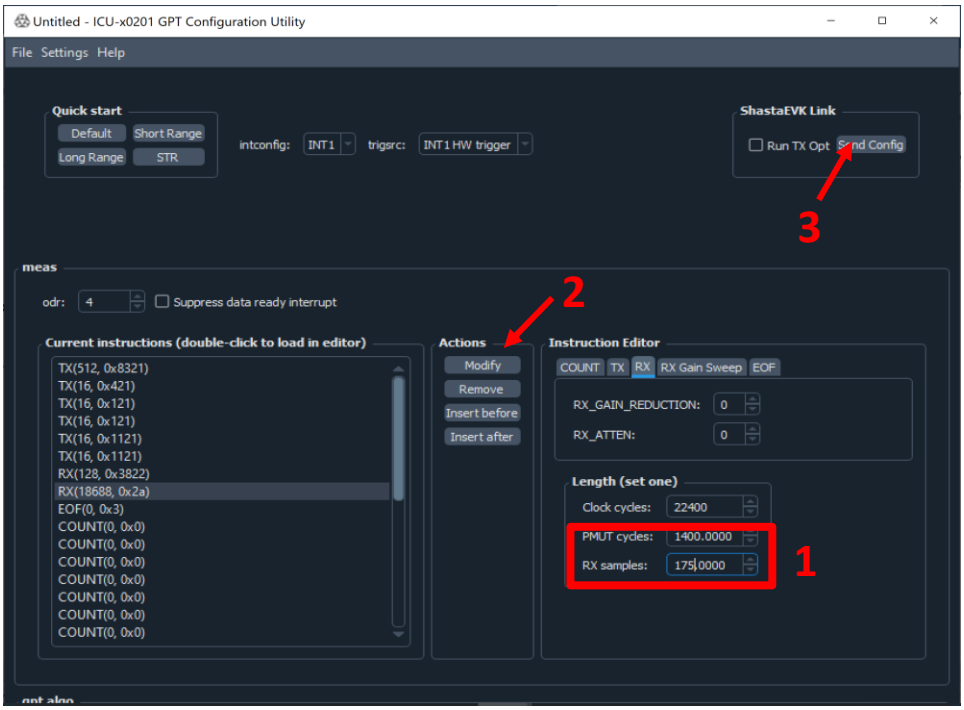


Figure 39: Modifying and implementing the new range settings

After that close the configuration sensor widget, the GUI will show the sensor data according to the new range set.

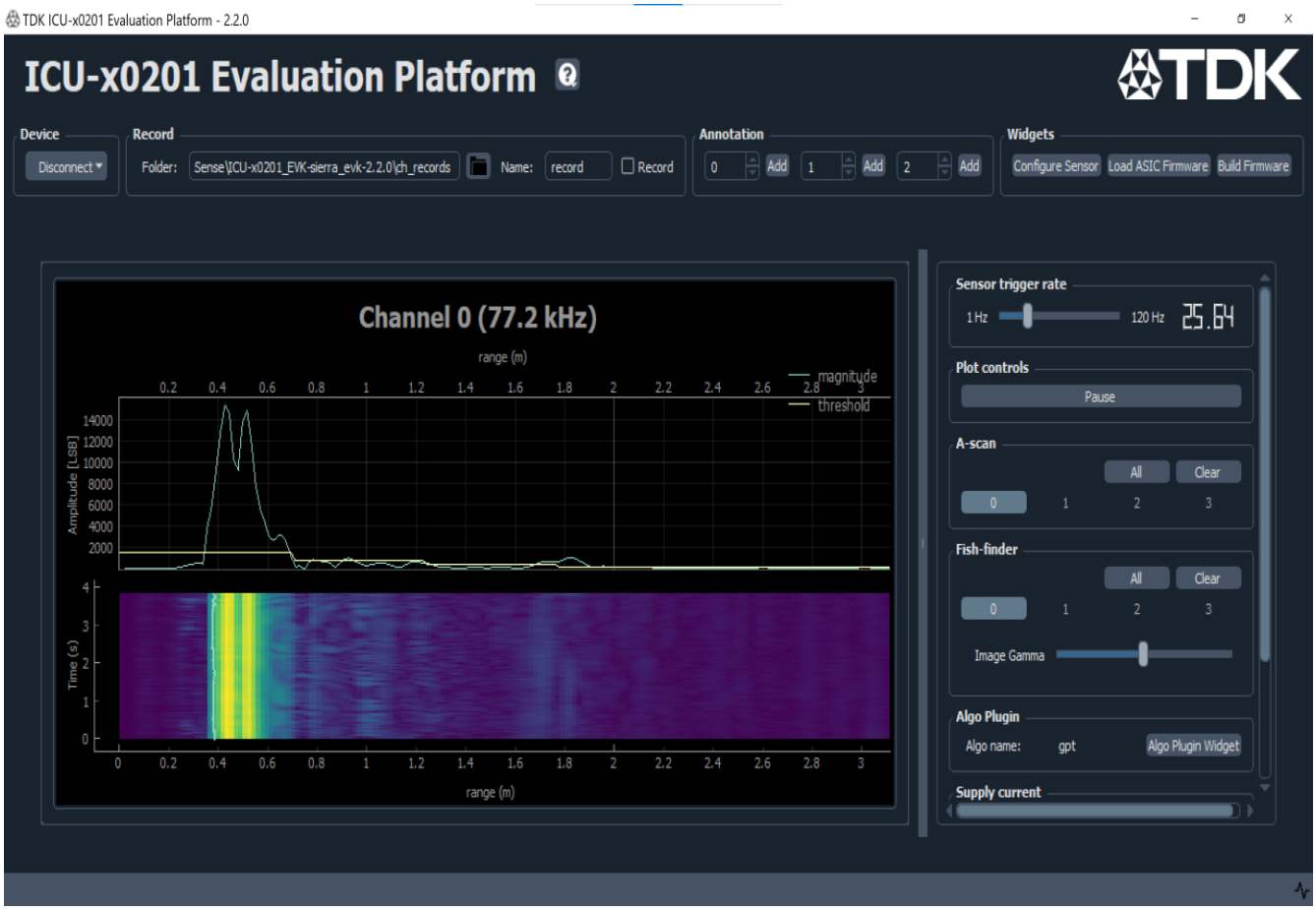


Figure 40: GUI overview for 3m ranges

4.2 MAXIMUM SAMPLE RATE

The ICU-x0201 measures the time-of-flight of a burst of ultrasound in order to determine the range to a target. The maximum range sampling rate is limited by the time-of-flight to the furthest object of interest.

For example, if you are interested in objects up to 3 meters away, the maximum time of flight is

3m * 2 / (343 m/s) = 17.4 ms,

where the factor of 2 is to account for the round-trip delay. If this were the only constraint, the maximum sample rate would be 57 Hz. In practice, there is also time required to process the data, so the maximum rate is slightly less than this. This evaluation software will automatically limit the maximum rate.

4.3 TRANSMIT OPTIMIZATION

As described in the description of the configuration window, the transmit optimization is run when the measurement configuration is sent with the *Run TX Opt* box checked. This should only be run once after a new configuration is sent to the sensor. To run again, reload a fresh configuration, or delete the added transmit instructions. You will see these as a sequence of several short TX instructions after the main transmit.

The purpose of the transmit optimization is to improve the sensor short-range performance. It does this by reducing the amplitude of the "ringdown" signal, which is the strong residual signal from the transducer ringing after the transmit pulse ends.

To observe the ringdown, you can set the "ringdown cancel samples" to zero in the measurement configuration window. Additionally, you can set the RX instruction to have a gain of 18 and an atten of 1. This will allow observation of the raw ringdown without risk of saturating the ADC.

To quantify the reduction of ringdown due to the transmit optimization, the following steps can be followed:

i. Load a measurement queue with no pre-existing reverse-drive instructions.
ii. Set the "ringdown cancel samples" to 0.
iii. Configure a single RX instruction with gain 18 and atten 1.
iv. Send the measurement queue.
v. Note the amplitude of the ringdown signal.
vi. Check the *Run TX Opt* box.
vii. Send the measurement queue.
viii. Note the new amplitude of the ringdown signal.
ix. Find the ratio of the ringdown signals from steps 5 and 8.

5 APPENDIX

5.1 SUPPORTING DOCUMENTS

The following AppNotes are also available for reference:

EV_MOD_ICU-20201-01 EV Module User Guide	AN-000357
EV_ICU-20201-00 EV Board User Guide	AN-000396

6 REVISION HISTORY

REVISION DATE	REVISION	DESCRIPTION
02/17/2023	1.0	Initial Release