CTC PRO

WA100 Series

WIRELESS APPLICATION

PROTECTION & RELIABILITY INSTRUMENTS

CONTENTS

CONTENTS

Please refer to the following documentation based upon the needs of your setup.

CTC Wireless Application Users - If you plan to use CTC's software (including use of the WA100 tablet) as the primary way of taking and viewing readings from CTC wireless sensors. CTC software contains an easy way to view time-waveforms and FFTs, as well as organize transmitters, sensors and receivers. Please refer to the CTC Wireless Application Guide, pages 3-23.

Standalone Reader Users – If you plan on using our command-line software only to create datafiles that will be consumed by your own systems. Our software stores these datafiles in a json format, and it is assumed that customers will have a procedure to read/process these files as readings into their own systems. Please refer to the CTC Wireless Application Guide pages 24-END.

Raw BLE Data Users – If you plan to interface your software and platforms directly with WA100 transmitters on a raw Bluetooth level.Please refer the Developers Guide, and the Excel File of Raw BLE Service and Characteristic data.

CTC Wireless Application

Requirements

- Windows 10 minimum build version: 1703
- Administrator Privileges
- Bluetooth® 4.0 Built-in or Dongle
- Microsoft SQL Server 2012 or above (See Below)
- Approximately 1Gb of free storage (5Gb recommended) to create application files, log files, and database records

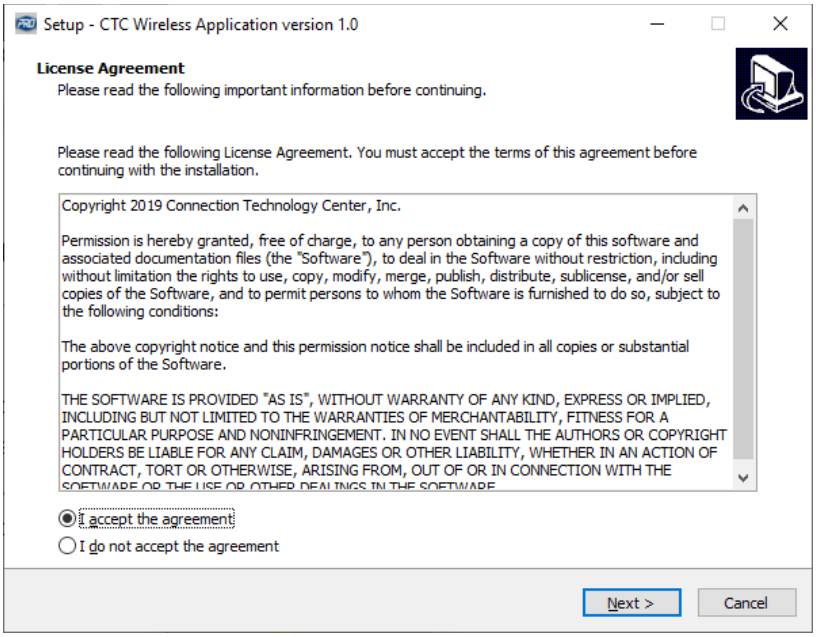
Embedded Server Mode Requires SQL Server

This software has several different configurations. The default configuration is to run an embedded API server, which requires Microsoft SQL Server. If you need a free version of Microsoft SQL Server, copy and paste this link into your browser (https://www.microsoft.com/en-us/sql-server/sql-server-downloads) and download/install the free/specialized version "Microsoft SQL Server Express." If you already have a Microsoft SQL Server, after you install the software, please make corrections to the "Connection String" in the settings to connect to your desired server.

MIT LICENSE

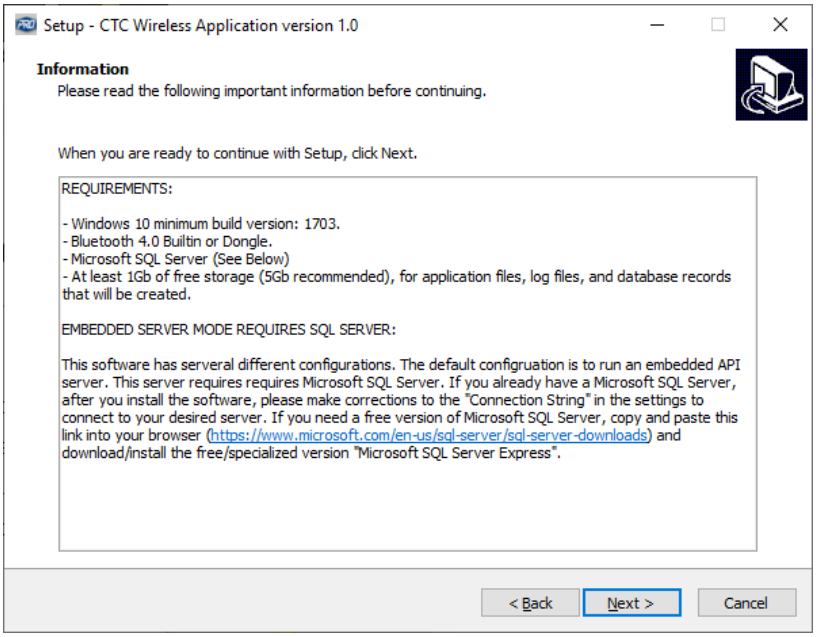
Installation

1. Run the program labeled “CTC Wireless Installer”
   a. Please note that this installer must be run under an administrator account



   b. Read and Accept the license -> Click Next

2. Review the requirements

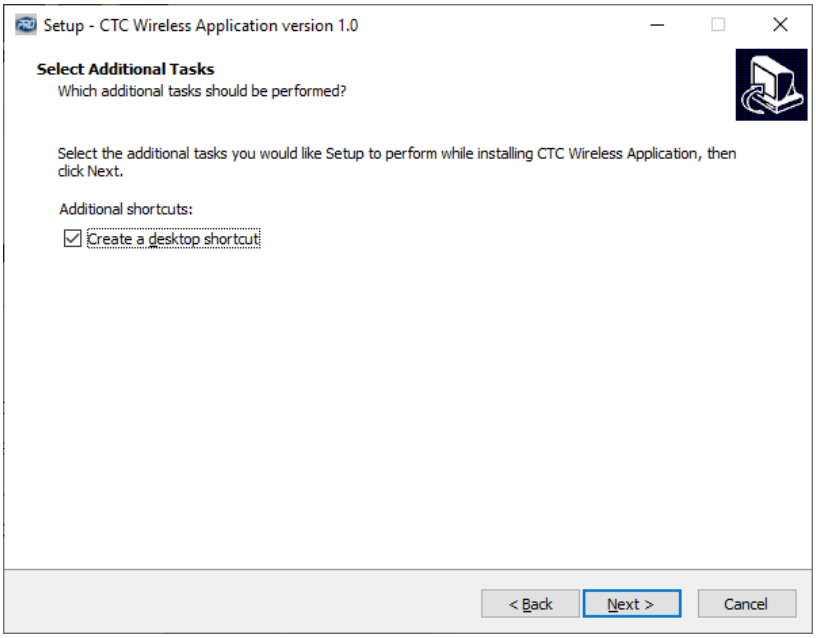


   a. A Microsoft MSSQL Server 2012 or later is required to run the software; make sure there is reserved space on an existing MSSQL database within

your organization, or install the free latest MSSQL Express from the Microsoft website *https://www.microsoft.com/en-us/sql-server/sql-server-editions-express*

b. Click Next

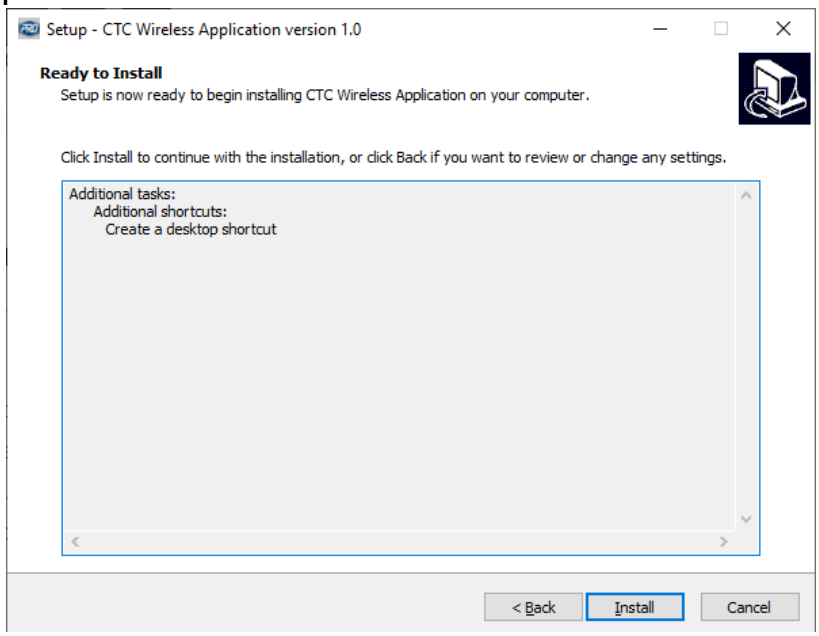
3. Make sure the "Create a desktop shortcut" option is selected



a. This will allow the user to easily enable admin privileges later

b. Click Next

4. Install the application

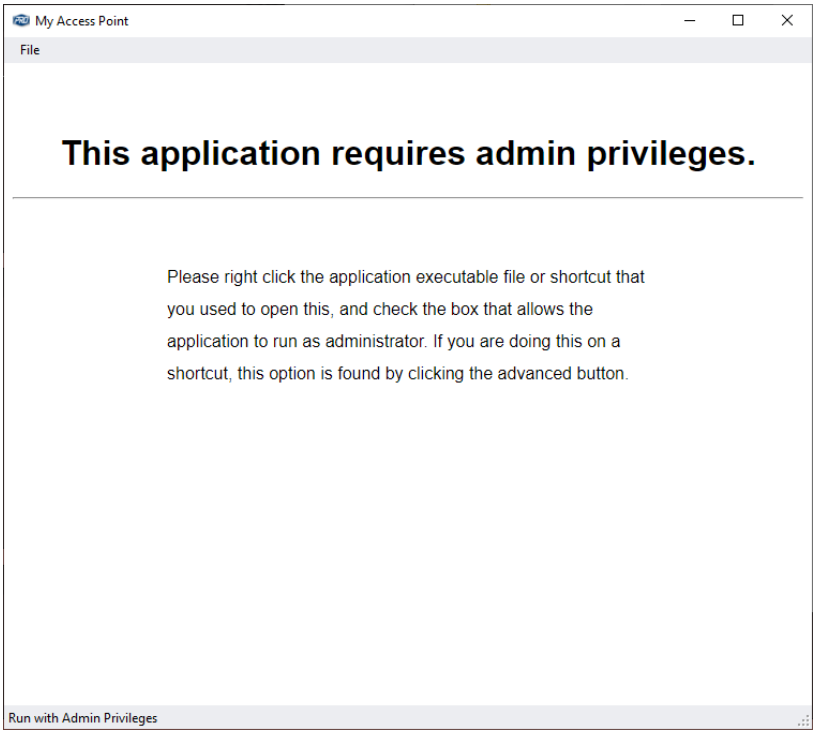


a. Click Install

Starting the Application for the First Time

The first time the application is started the Windows Firewall notification may pop up, asking the user to grant the application access to the public and private networks. It is best to allow the application access to both the private and public networks. If running the embedded server, the application may only need access to the private network, but if there is a need to change the configuration later, it would be best to give it access to both networks.

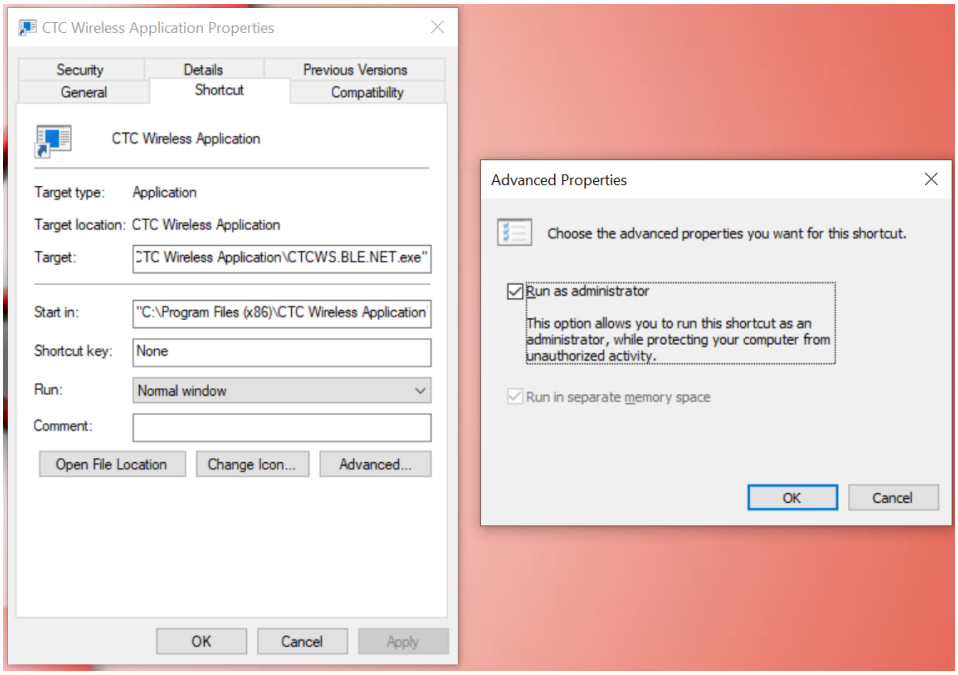
Additionally, the user may encounter a window stating that the CTC Wireless App will need to be run with administrative privileges.



The application requires administrative privileges in order to write the configuration files for the embedded server, as well as write log files for sensor readings and cached settings files for offline use.

To assign the application administrative privileges, find the desktop shortcut that the installation created, right-click it, and select properties.

In the window that pops up, click the advanced button. When the “Advanced” window pops up, check the box that says run as administrator, and click OK. Finally, click “Apply” in the original “Properties” window.



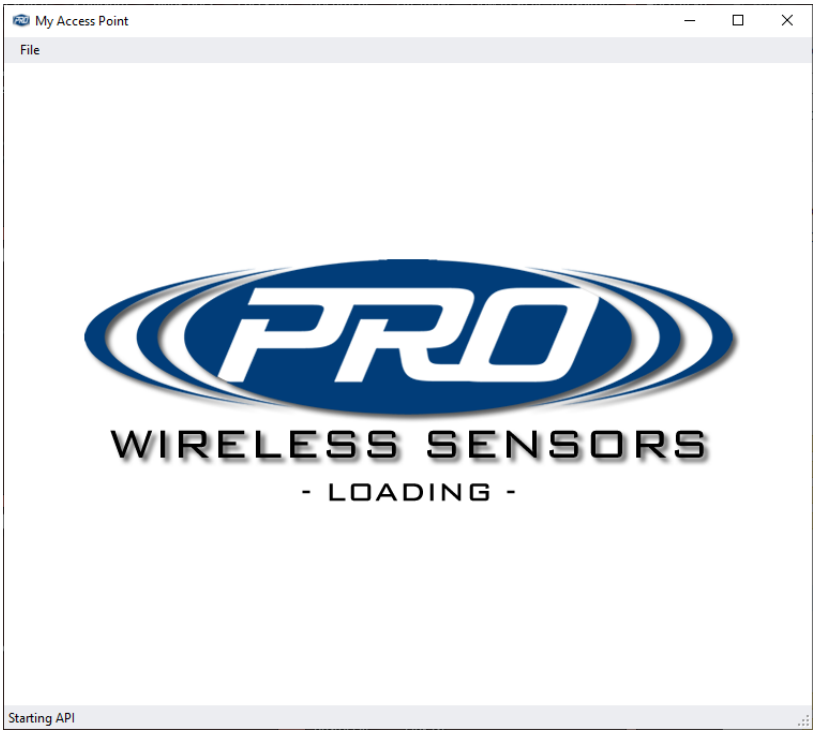
A confirmation dialog may appear that requires the user to provide administrative credentials. Click continue, and if the Windows Smart Screen pops up, type the computer's administrative credentials into the appropriate fields.

Please note: some organizations do not allow non-administrators to install or run unsigned programs. In this event, the application will need to be installed and run under the Administrator account. Please see your system administrator or IT department if you are unable to get the program to run correctly.

The application is now ready to take readings.

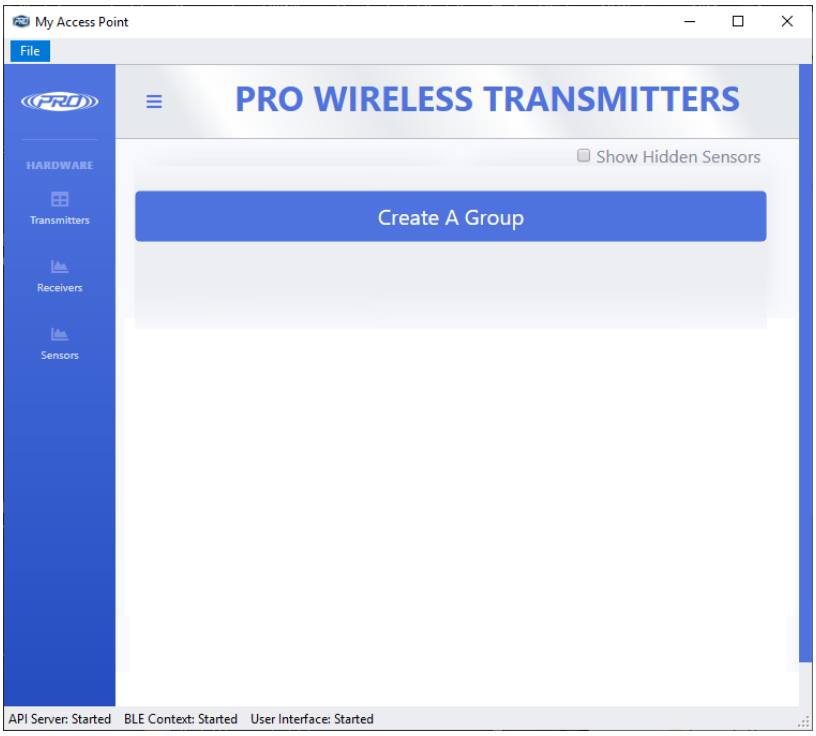
Running the application

When you run the CTC Wireless Application, the first thing that will appear is a loading screen. This screen may take up to 30 seconds on the first run, as it creates necessary directories, configures the server, starts the embedded server, and synchronizes its own configuration.

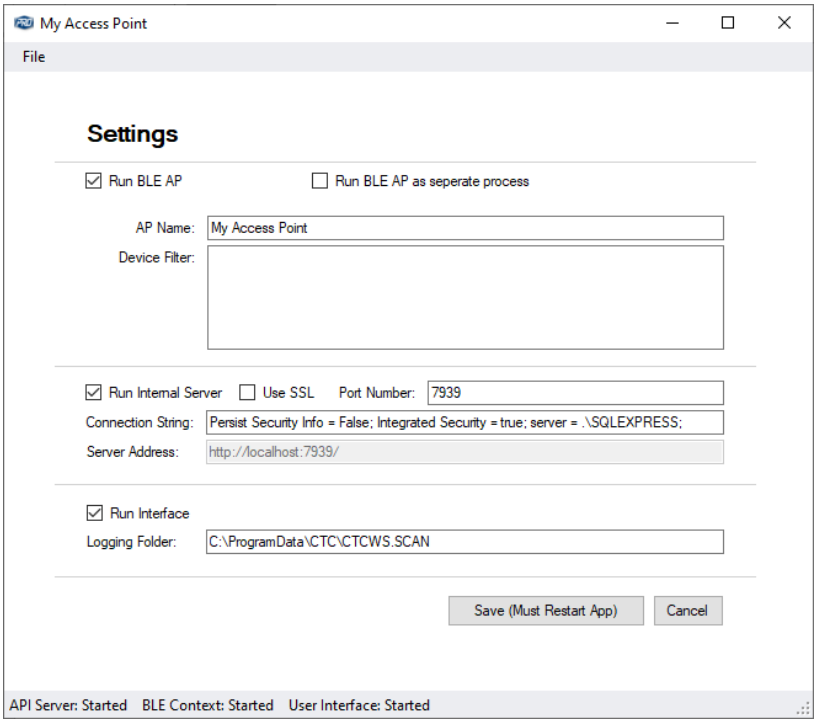


Settings

Upon Successful Install and Setup, the following window will be shown.



For the moment ignore this window and jump into the software settings screen. Choose "File" and then "Settings," to display the screen below.



To the left, there is a checklist with three main options: Run BLE AP, Run Internal Server, and Run Interface. To the right, there is a checkbox labeled "RUN BLE AP as separate process". Checking this box allows the Access Point Software to run as a separate executable on the Windows operating system. If running on a slower computer, or with many BLE devices, or even if you are having difficulty connecting to one device, running the BLE AP as a separate process can help improve performance of the application. This option may help complete readings that may otherwise get bottlenecked by other processes or slow execution on Windows.

These are the three main components of this application. They are described below in more detail.

Run BLE AP

This document will define all of the little black boxes with MIL connectors as CTC Wireless Transmitters. They have all been created using the latest Bluetooth® technology called Bluetooth® Low Energy, which appeared in the Bluetooth® specification around Bluetooth® 4.0. These transmitters all require communication with one or many central locations. We refer to these central locations as Receivers or Access Points (AP).

The first feature of the CTC Wireless Application is the Access Point. This central point has the code and workflow built into it in order to allow it to communicate with the transmitters properly. Checking the "Run BLE AP" box allows you to run this software automatically when the application starts.

When running the embedded receiver feature, the user can also assign an AP name to this software, as it will appear in the application interface with that name.

An alternative to running this feature when the software starts is to run it as a separate process. To do this, uncheck this box and run the receiver as a batch file. Please see the use cases later in this document for more details on how to do this.

Lastly, the application can be run in a separate process, but not even on the same machine, or on several other dedicated computers scattered throughout your facilities; this is also possible through said functionality.

Run Internal Server

The internal server is the second piece of the software that runs by default. To stop this, uncheck the second checkbox. All of the receiver/AP software that CTC builds has integrated functionality that understands how to communicate with our server. The server acts as a WEB API built on Microsoft's ASP.NET Core Stack.

Options are provided when running the internal software, which allow the user to turn SSL on and off. It is possible to change the port number the server runs through, or the connection string the software uses to communicate with an "MSSQL 2012+" server. Essentially, this allows the SQL server to be installed in a different location from this server or application. **Warning:** this feature uses a development certificate – it is much more secure to buy and install your own certificate on your server.

If opting not to run the embedded server, it is possible to change the "Server Address" that the rest of the software uses for sending and receiving data.

Run Interface

The last feature built into the CTC Wireless Application is the GUI or User Interface. This interface is run off the server through a single page web application, using AngularJS, Bootstrap, and other standard web front end frameworks.

The alternative to running the GUI through this application is to browse to the server address through a modern web browser like Chrome, Edge, Safari, or Firefox.

Logging Folder

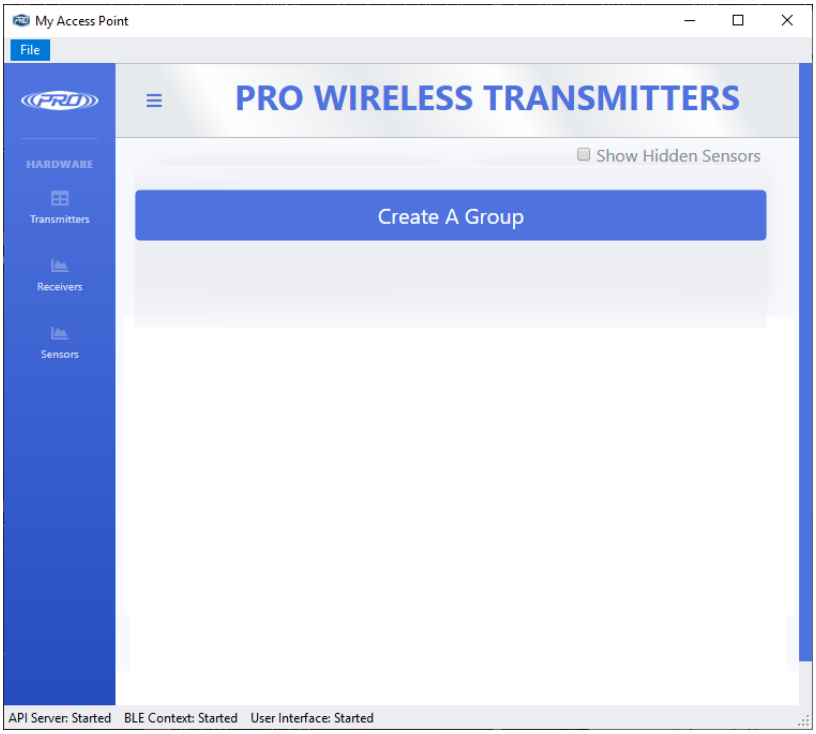
In addition to the location where the application saves its settings, there is another system folder that is used to read and write configurations and pending files. This folder is called the logging folder.

Summary

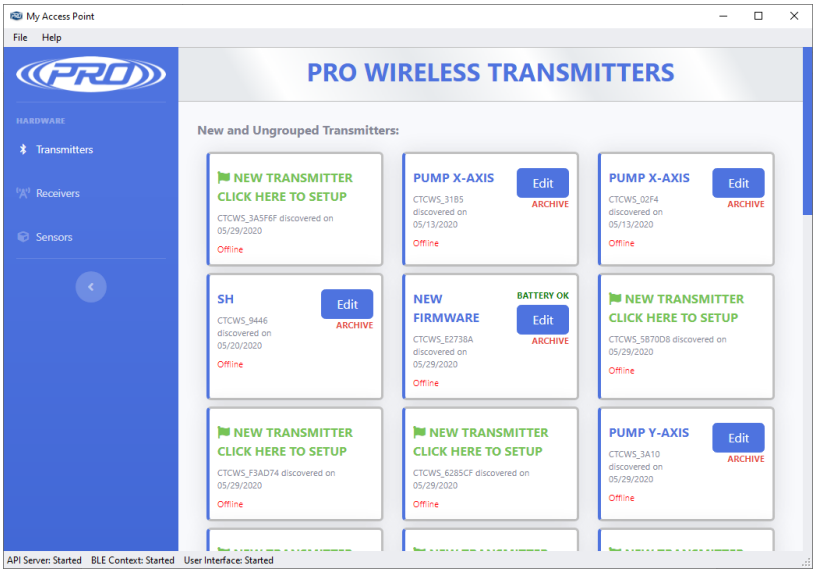
There are several ways to run this software, and it is possible to break all of the individual pieces apart and run them separately over a distributed network, in the cloud, or however is most convenient.

Transmitters

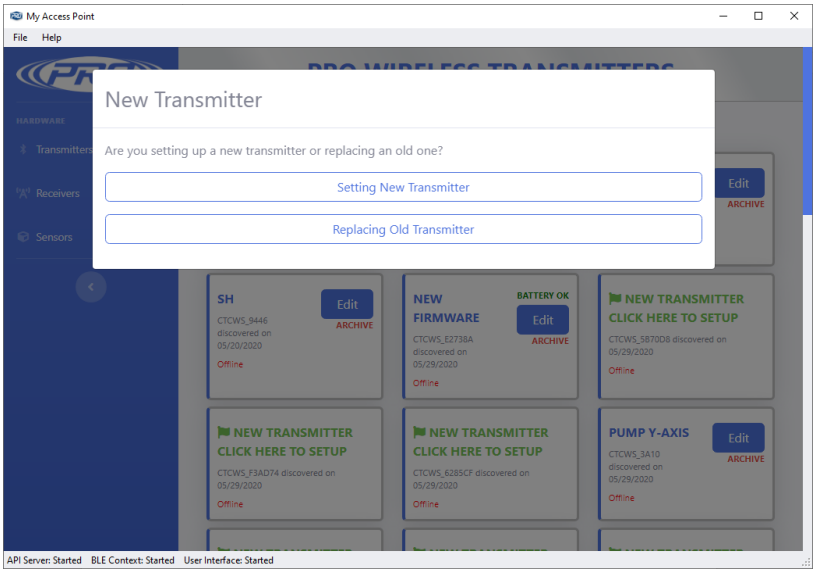
Once the program loading completes, the "Transmitters" window will be displayed. **Please note:** when discussing Transmitters, the program refers to the individual wireless connector components with which it communicates.



As the application begins to sense one or more CTC Wireless Transmitters, they will automatically begin to appear, as illustrated in the example window below.



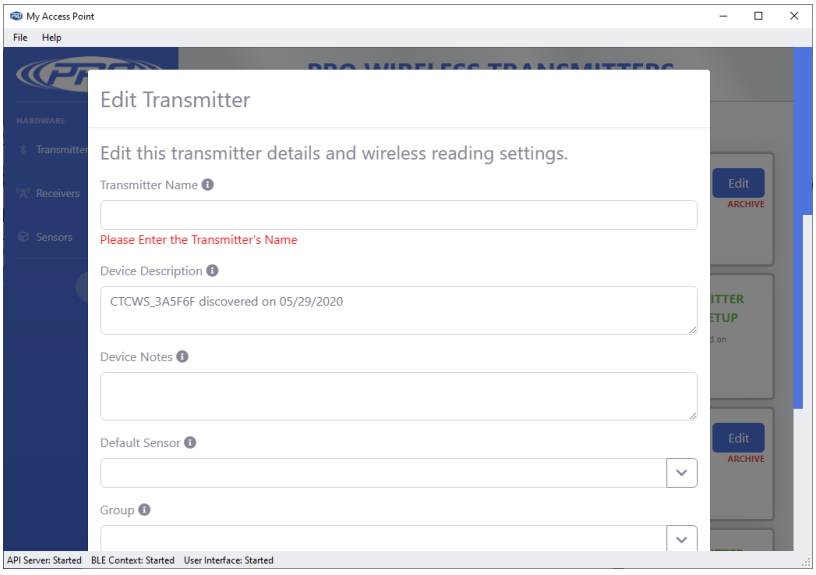
When sensors begin to appear in the application, they will appear with the text “New Transmitter Click here to setup.” You will not be able to use your new transmitters with this software until you have completed a short setup process.



Clicking one of the new transmitter cards will create a prompt to either set up a new transmitter or replace an old one.

Setting up a new transmitter

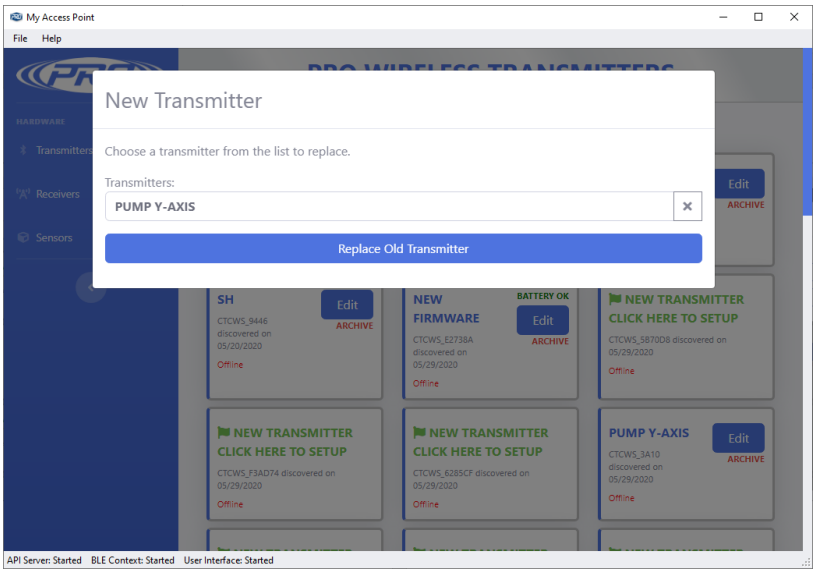
When setting up a new transmitter, the only requirement is to fill in the Transmitter Name field. Doing this and clicking the save button will allow usage with the new transmitter.



Replacing a Previous Transmitter

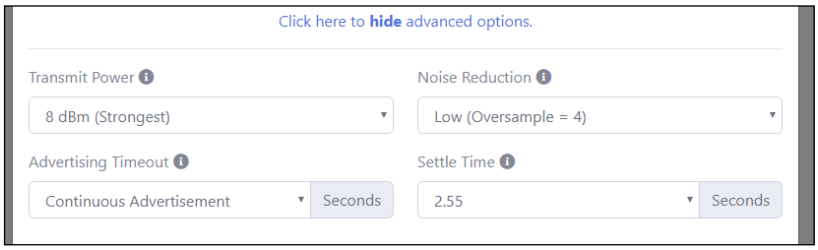
When replacing an old transmitter, the user will be asked to provide the name of the transmitter to replace. Select the old transmitter's name from the list, and click the button labeled Replace Old Transmitter.

After a brief pause the interface will refresh, and the new transmitter will assume the role of the previous one. It will inherit all of the properties of the old transmitter, including the name. All previous readings (viewable in the readings pane) will have been automatically transferred over to the new transmitter.



Advanced Transmitter Options

Lastly, on the transmitter settings screen, there is a link labeled "Click here to show advanced options." Click this to see some of the more advanced options for taking readings.



1. Transmit Power – the strength of the wireless signal sent from the transmitter. By default, the power is set to its highest setting of 8 dBm, but can be adjusted down to lower values as the situation permits. Adjusting to a lower value is useful to conserve battery life if the receiving device is located in very close proximity to the wireless transmitter. However, weaker signal strength reduces

overall range, and you may experience some signal loss.

2. Noise Reduction – the averaging of signal readings to produce smooth signal output. By default, this is set to a low value of 4, meaning each data point is an average of 4 readings on the sensor. The other options available are "None," "Medium," and "High." None allows the highest sampling rate with no smoothing (1 reading for every point), while medium and high offer smoother and the smoothest signals, respectively. The drawback to higher levels of noise reduction is that the maximum sample rate frequency will be reduced.
3. Transmit Rate – the data rate which the receiver will talk to the transmitter. Most BLE compatible computers will only support the 1 Mbps option, however if the users' computer uses a newer BLE chip it may be able to use the 2 Mbps option. If 2 Mbps is chosen and the computer does not support it, the system will fall back to the 1 Mbps option. Please note that using 2 Mbps requires closer distances between the transmitter and tablet/receiver. After the first reading, additional information about the sensor will be accessable.
4. Settle Time – the settle time of a transmitter should be set minimally to the settle time of the sensor. Settle time can be configured to wait for up to 2.55 seconds after applying power to a sensor to take a reading, which is necessary because all sensors take a brief amount of time before the signal is flat after being turned-on/powered.
5. Model Number – The model number of the transmitter.
   Hardware Revision: The hardware revision of the transmitter board.
   Firmware Revision: The firmware revision on the transmitter board.
   Software Revision: The software revision on the transmitter board. (Not the reader software)

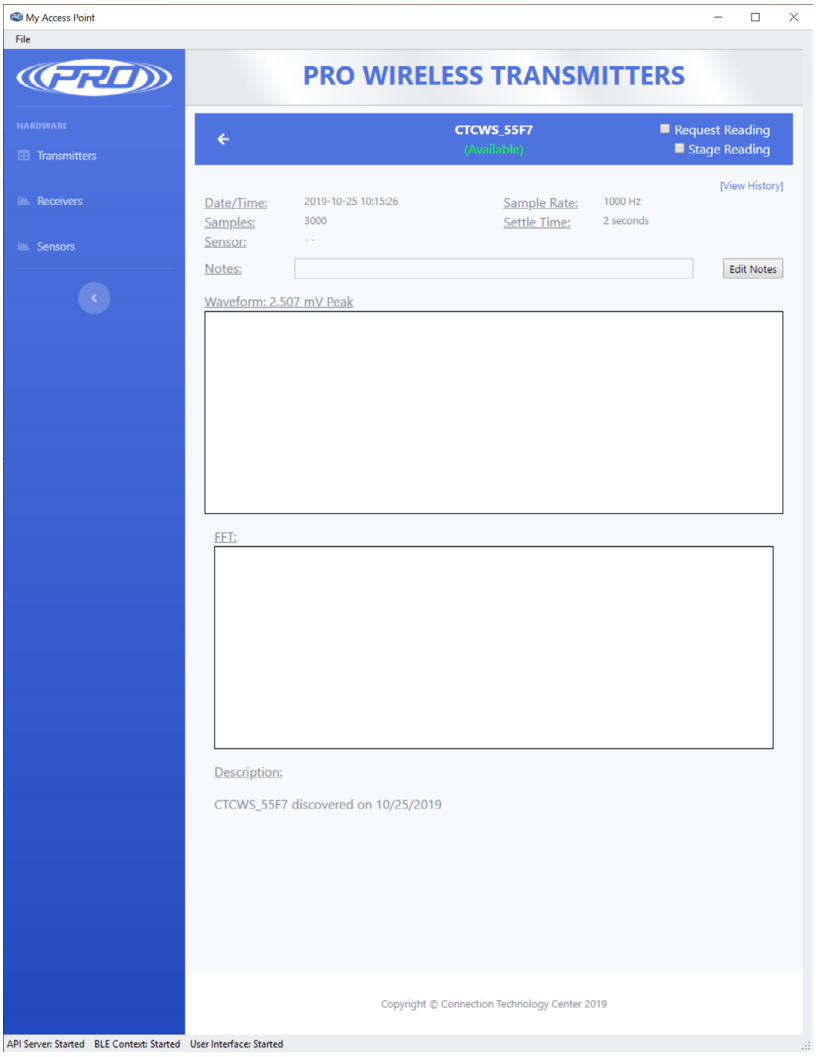
Back on the Transmitters window, we will look more in-depth at a sensor indicator. Please see some additional options, as illustrated below.

1. The device name - this can be changed in the device settings. When a sensor is first discovered, it sets to the ID of the transmitter.

2. The battery indicator - displays either Battery OK when a sensor is running with good power, or Battery Low when a sensor needs to be replaced.
3. The sensor's description - this is also configurable through the device settings. When a sensor is first discovered, this field sets to the ID of the transmitter. For identification purposes on this screen, it's a good idea to keep either Name or Description as the transmitter ID. It is recommended leave the name as the ID and use the description as the user identifier.
4. The "Edit" button - brings up the Edit Device screen, as seen in the previous example.
5. The sensor's status indicator - if you are in the vicinity of the sensor and are not taking a reading, this will display Available. It can also display a series of other statuses, like Connecting, Connected, Ready, Requesting, Receiving, or Done, based on the current operation. Lastly, if the sensor is not in the vicinity, dead, or not available, the status will display Offline.
6. Delete and Hide options - Delete erases the sensor, and all of its readings, from the database; while Hide removes the sensor from view.
    a. If you choose to delete a sensor, and it is not truly gone (dead or out of range), it will reappear once it is discovered again.
    b. Deleting a sensor erases all of its associated readings, so please practice caution when deciding to delete a sensor.
    c. If you simply don't want to see a sensor, sometimes is it better to hide it, as this removes it from view.
    d. Instead of hiding, another option for removing a sensor from view is to put it in a group. Best practice would be to logically group all sensors based on some shared detail like location or use.
7. In the window containing all of the sensors, there is an option that says "Show Hidden Sensors." Checking this box shows all sensors that were previously hidden, allowing the user to unhide, or perform any other set of actions on said sensors.

Clicking anywhere else on the sensor indicator box other than the Edit, Hide or Delete links, opens the sensors reading pane.

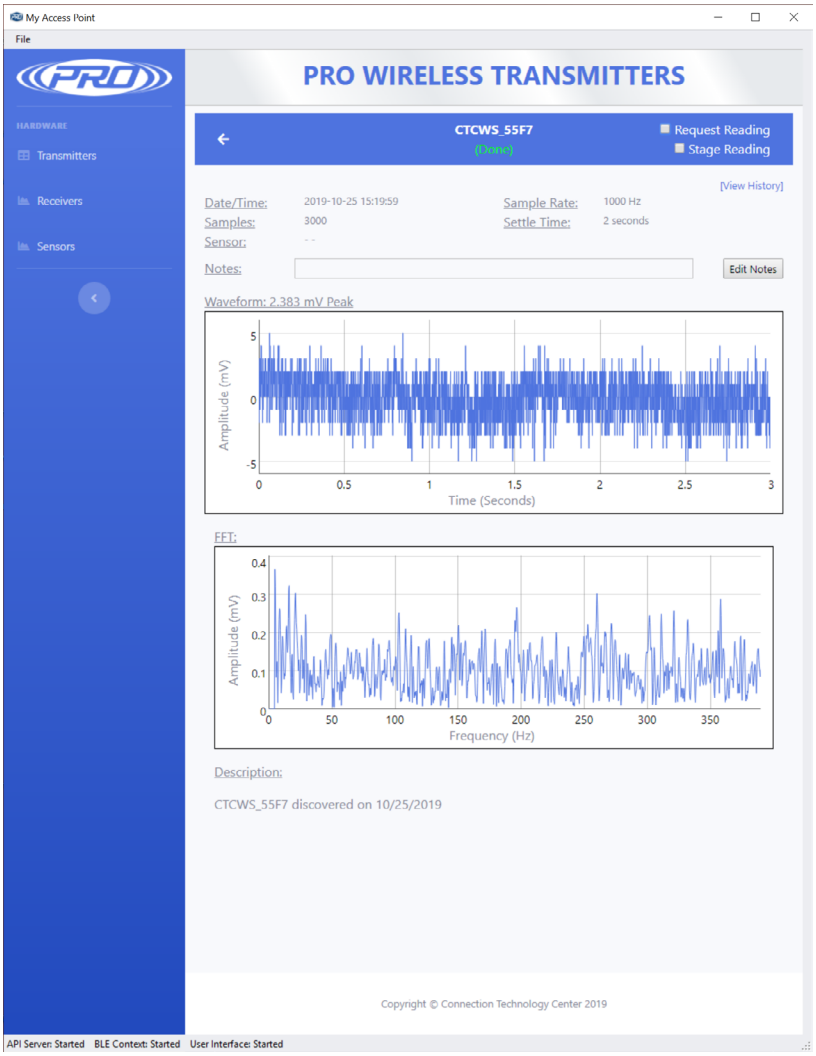
At first, this pane shows no data, as illustrated below.



Check the “Request Reading” checkbox, to initiate a reading.

After checking this box, the sensor will cycle through the other statuses: Connecting, Connected, Ready, Requesting, Receiving, or Done, until a successful reading is taken. If there a weak signal or latency, the sensor may connect and disconnect several times before completing a successful reading.

Once a reading has been taken, the output is displayed in the Waveform and FFT graphs.



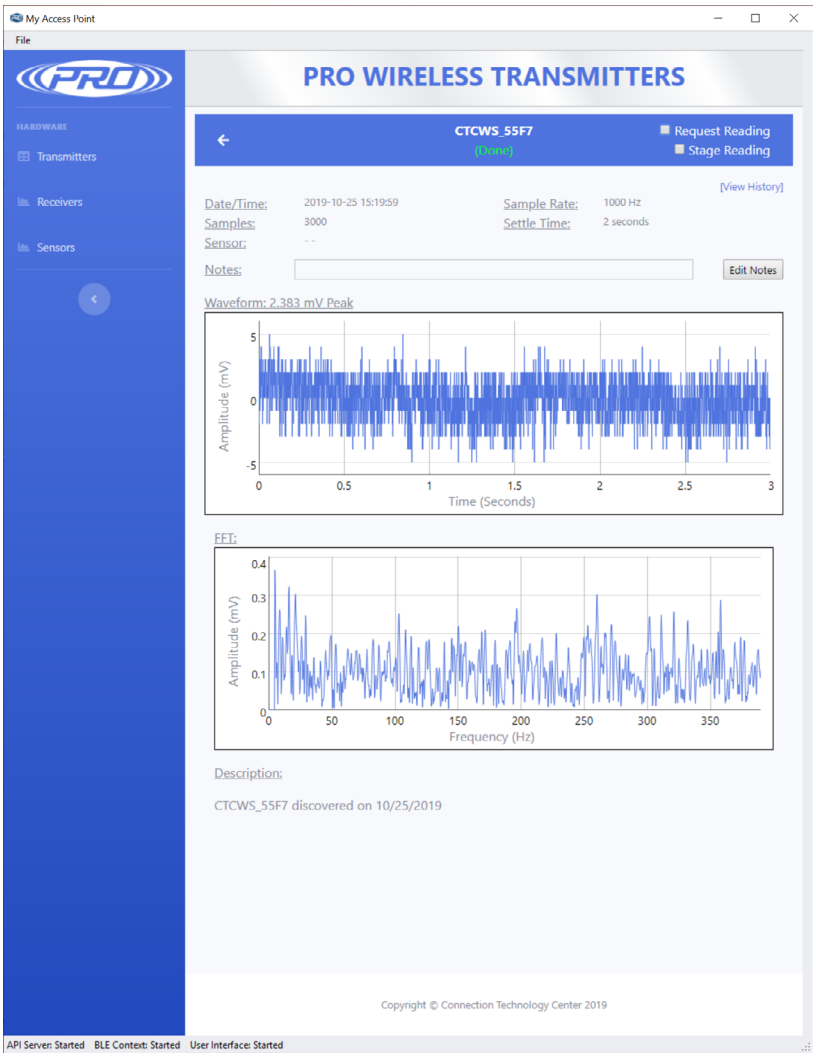
Click the edit notes button to keep any relevant notes about the readings for later review. The View History link displays all previous readings from that sensor.

The other option on this screen is the Stage Reading option. Stage reading is similar to the request reading option; however, it does not automatically initiate a reading. As the label implies, it simply “stages” or puts the sensor in a state for a reading to be immediately requested. Staging is useful for scenarios where the user has to initiate a reading without delay. For example, the startup of a piece of equipment or any other timing-sensitive event, where waiting for the sensor to initialize will cause said event to be missed.

It is important to note that staging a sensor is the most intensive operation in terms of battery usage. It is recommended to use this feature sparingly, and utilize Request

Reading whenever possible.

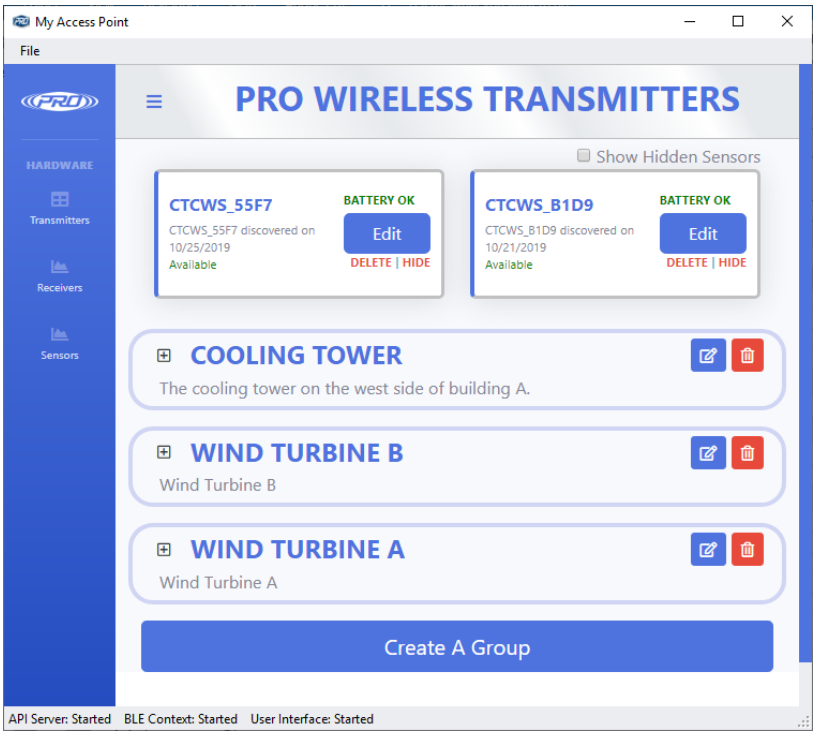
Please note: in the previous example, a sensor has not been selected. As a result of no sensor being selected, the output is displayed in 'mV's and not 'g's. Creating a sensor is described in the next section.



Lastly, clicking the back button brings opens the primary sensor screen.

On the left-hand side of the screen, please notice two other sections.

1. Receivers
2. Sensors



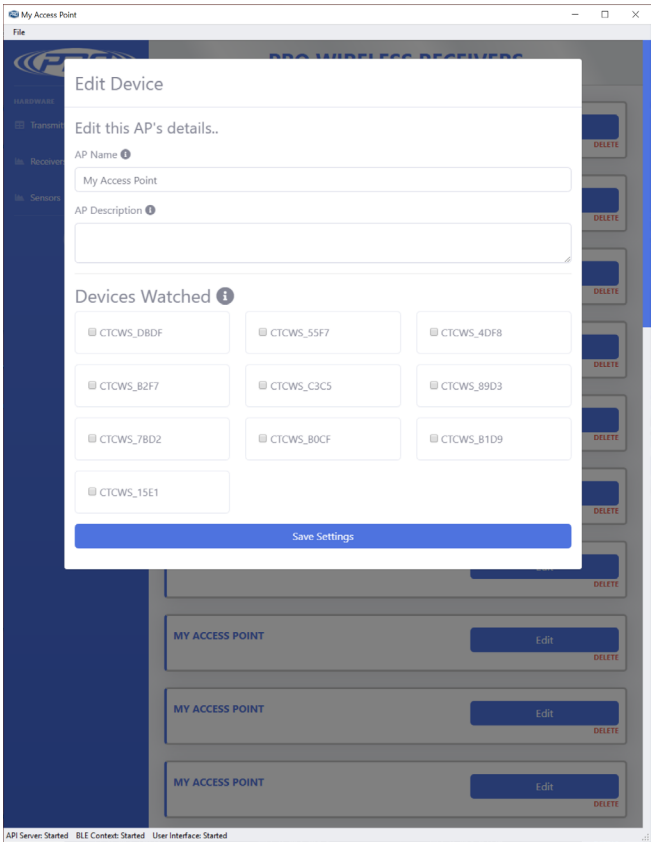
Receivers

This section holds a distinct record for each receiving device that has registered/synchronized with the server.

A receiver can be a couple of things.

1. First, a receiver may be a program just like the one you are using. This program can act as a receiver, a server, a user interface, or any combination of those devices. By default the program operates as all three options at once.
2. Second, a receiver can be command line based. It can be a scheduled task that opens a command line version of the receiver program. This utility and its configuration options are described later in this document.
3. Third, a receiver can be a dedicated computer that sits within the vicinity of a series of sensors/transmitters for the purpose of taking on-demand or scheduled readings. For example, a small Linux powered computer that takes readings and forwards them on to a server.

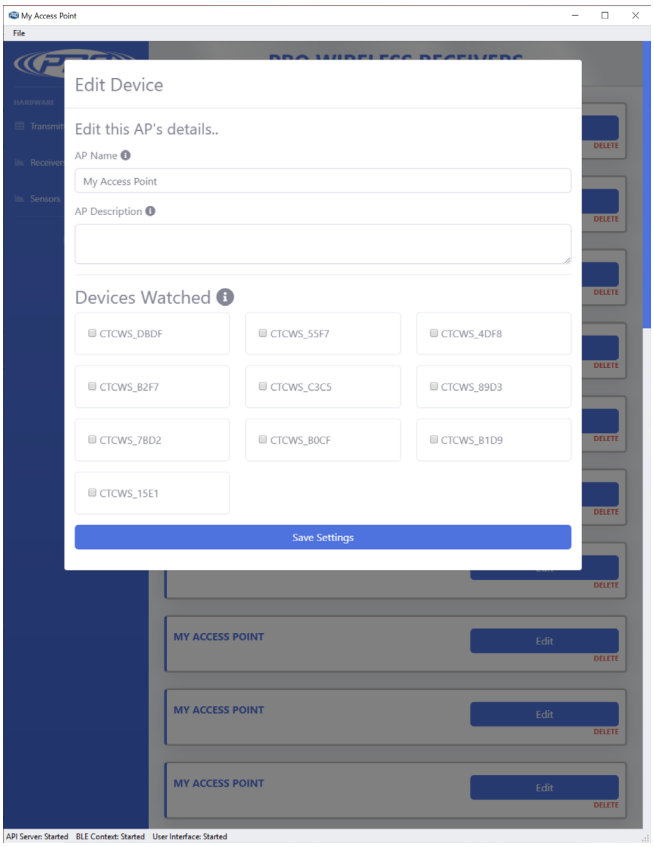
In the example below, the receiver is the application itself, and has automatically assumed the name “My Access Point.”



Just like the sensors section, the receivers allow the user to fill-in and save information to make them distinct, and provide some specific operational data. Please note that receiver names are stored on the receivers themselves, so overwriting the name in the form shown below has no effect. The receiver name needs to be maintained on each device or implementation.

There is a section in the example below that shows all of the "Devices Watched." On the screen, several transmitters are displayed that have been discovered by the receiver, each with an empty checkbox to the left of the transmitter name. When a receiver is configured to watch a particular transmitter (or transmitters), no other receiver will be able to connect to that transmitter(s), unless that other receiver is also configured to watch that transmitter(s) as well. Alternatively, if no receivers are watching a particular transmitter, then any receiver within range will be allowed to connect and take a reading.

This feature is useful when partitioning specific receivers to watch specific transmitters. It may be done to reduce latency, or logically keep specific groups of equipment separate from others within the sensor network.

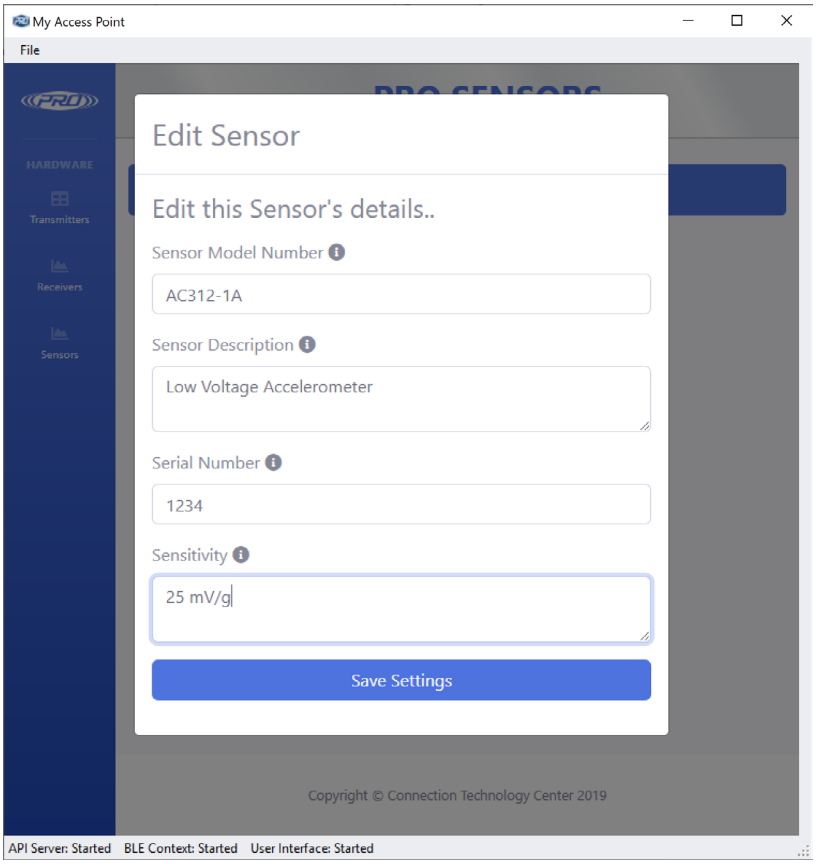


Sensors

This section is useful for setting up and providing essential details about the actual transistors purchased from CTC. The important details that you may record about a sensor include model number, description, serial number, and sensitivity.

The first three fields are self-explanatory: Model, the sensor model number; Description, to record any other relevant information; Serial, the serial number of the sensor being used.

Lastly the fourth field, Sensitivity. This field is used for converting readings, from 'mV' to 'g' or 'IPS' values, back on the transmitter pane. The sensitivity field is also dynamic such that you can always come back and make small adjustments to the sensitivity, which will instantaneously and retroactively affect any readings that were done with the connected sensor. This is useful for custom scaling of the output, so the user can pseudo calibrate a sensor with a known vibration level.



Under Default Sensor, enter the sensor that you want to use to take readings. Enter the sensitivity that came with the calibration certificate of the sensor. The above example uses an AC312-1A sensor, which will have a 25mV/g sensitivity.

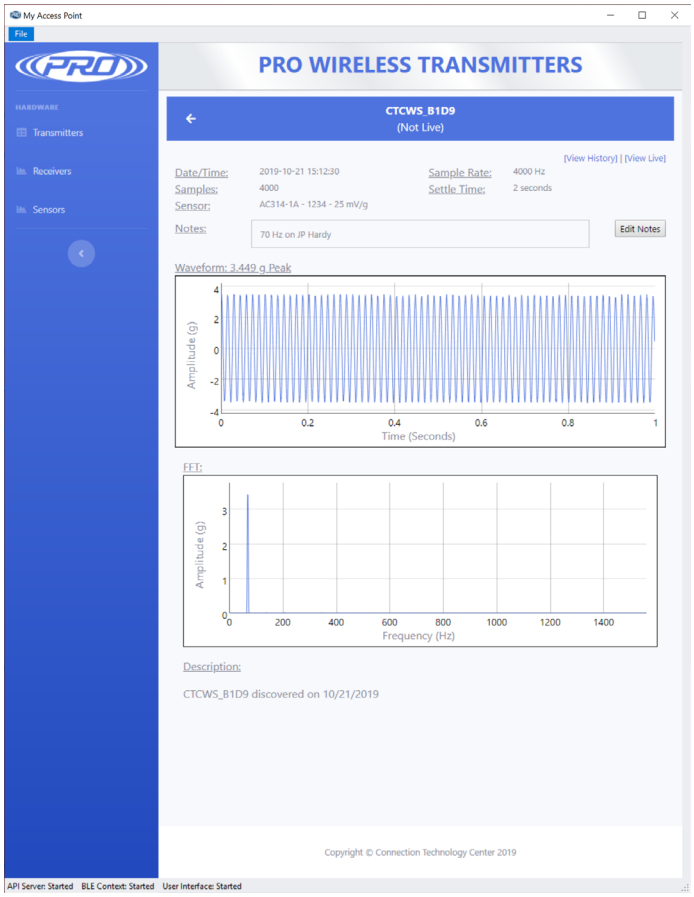
When entering the sensitivity, be sure to use the following format:

"{Numreic Value} mV/{Unit Of Sensor Output}" (i.e. "25 mV/g").

The blank space between the number and the measurement unit is essential. It ensures that the program can parse the data correctly in the readings window.

Click Save.

In the transmitters section, find the transmitter that connects to the created sensor, click the Edit button, choose the sensor, and click Save.



Just as before, click in the sensor indicator to go to the reading pane, and check the "Request Reading" box to take another reading.

This time, once the reading is complete, an output will display that reads 'g's or 'IPS's based on the sensor configuration entered.

Please note that if a sample is rate too low, the amplitude may display as artificially low or high, as low sample rates diminish amplitude accuracy. Although the Nyquist formula recommends a minimum sample of 2.56 times the expected maximum frequency, five times is recommended for a more accurate amplitude calculation.

It is useful to have an existing knowledge of the vibration level an application produces, so that once a good amplitude calculation is obtained slight adjustments can be made to sensitivity in the sensors window so that output is an exact match.

The Standalone Reader

The standalone reader is a command-line based application that comes bundled with the CTC Wireless Application. The reader is included for the purpose of taking on-demand and scheduled readings.

The most straightforward script is "CTCWS.BLE.READER.exe -watchall 1 -target CTCWS_58E5" which will automatically engage as watching all transmitters and target the transmitter named CTCWS_58E5. This results in the reader attempting 1 successful reading (with all default settings) from CTCWS_58E5, and sending the data to the default server (http://localhost:7939).

Parameters

Here is a list of all the parameters that are available for the reader program:

watchall

Using the "-watchall 1" argument is recommended for any automatically scheduled reading. This setting overrides any watch limitations imposed by the server. In many cases, the reader is not able to connect to a server and will save the output locally until a server is within range. It is not recommended to use this option in continuous mode.

target

The "-target CTCWS_58E5" argument tells the program to automatically take a reading from a device it finds called "CTCWS_58E5". There is no default target, and the use case for not providing this value is when running the reader in "continuous" mode, where the reader is always running and reacting to requests from the server.

server

The "-server https://localhost:1234" argument can be used to change the address of the server this application communicates with. The default configuration uses a value of http://localhost:7939, which is the default server if this option is not provided.

folder

The "-folder C:/Scan" argument can be used to change the folder in which the program loads its configuration and saves its temporary outputs. C:\ProgramData\CTC\CTCWS.SCAN is the default if this value is not provided.

apid
The "-apid cc8f1b91-60f4-4e84-b69f-082ebb075ae8" argument can be applied to allow distinct and separate records for each of the readers. All readers assume a default value of "0DEFA170-DEFA-DEFA-DEFA-DEFA17DEFA17" if this value is not provided.

savelocal
"-savelocal 1" This argument allows the user to save local copies of the readings for other purposes like consumption by a different program/software. The default is 0 and will only save pending files, which will then be deleted when the server is available to collect them.

justpending
"-justpending 1" The reader can be run with this argument to only upload any pending readings, and not attempt a new reading. The application does this every time, and the default of 0 will run the program normally where it tries to upload any pending files first, then runs the data collection.

c9502
"-c9502 010000000A0D0202FFC80000" allows the user to enter a custom configuration. Please see the next section on how to calculate a 9502-configuration hex value. The default is either the last configuration fetched from the server for this device, or a software default if the server configuration is not available and has never been fetched.

c9501
"-c9501 00000BB8" allows the user to enter a custom number of samples to be taken. Please see the next section on how to calculate a 9501-configuration hex value. The default is either the last configuration fetched from the server for this device, or a software default if the server configuration is not available and was never fetched.

continuous
"-continuous 1" places the reader into "continuous" mode. This setting prevents a timeout of the application and it will run forever. This is useful when setting a device as a permanent reader. This program can run in the background, or minimized, and react to any server requests for readings. When the reader is not in "continuous" mode, the default timeout is 4 minutes. If there is no successful readings within 4 minutes, the application will automatically exit.

Use Cases

Navigate to the application install directory to find some examples which illustrate two use case scenarios.

"C:\Program Files (x86)\CTC Wireless Application\cmd"

Use Case 1

"usecase1.bat" - The scheduled reading – This example is intended to be set up in the windows scheduler. Running the batch file once attempts a reading, as described at the beginning of this section. It will be helpful to have previously run the GUI at least once and connected to the server you are using, as all settings will otherwise be default. Using the GUI interface allows you to adjust the setting of the reading, which is saved to a local file. If the server is unavailable when the scheduled task executes a reading, the local file will serve as a reference to the desired settings, and the reading will perform as intended.

For an introduction on how to use Windows task scheduler, look at this article from Experts Exchange: *https://www.experts-exchange.com/videos/1598/How-to-use-the-Windows-Task-Scheduler-An-Introduction.html*

Use Case 2

"usecase2.bat" – The standalone reader – This example uses all defaults as in the previous example but remains open continuously. It is best practice to also schedule this batch file to exist on a schedule that runs every minute and bypasses the next run if the program is already running. This way, if something happens and the program closes or crashes, it can always wakeup again with the next scheduled run. In this example, you would run a server and configure this program to connect to it. The use case uses the default server of http://localhost:7939/.

An important feature prevents the program from staying open while in continuous mode. This feature is to close the program if the timeout has passed, and there is no server connection available. If the scheduled task has been set according to the previous paragraph, the program will exit after the 4-minute timeout has expired and launch another instance at the next scheduled run. Each new instance attempts a fresh connection to the server. This feature prevents an inactive reader in the event a server connection was lost.

Use Case 3: An uploader

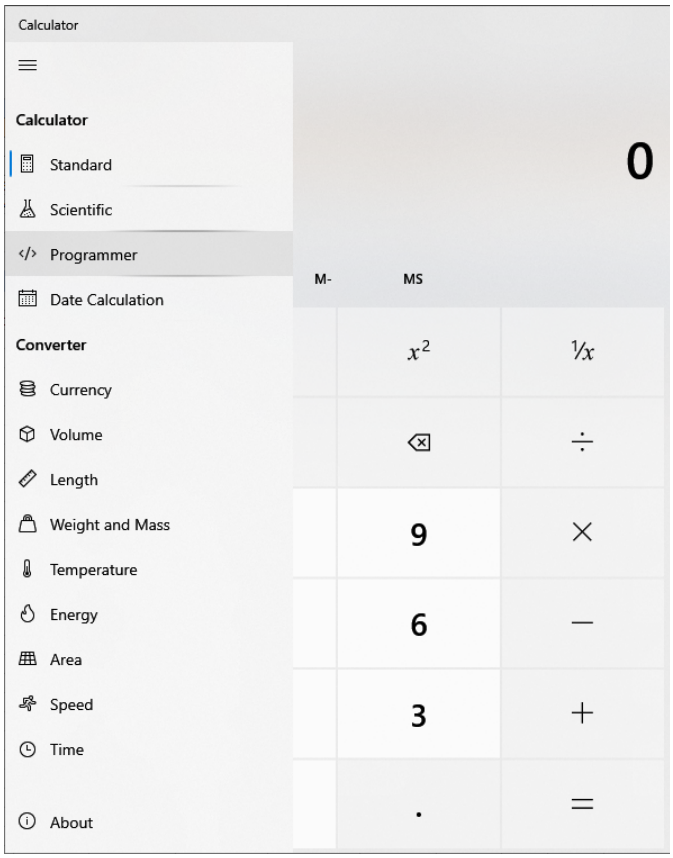
"usecase3.bat" – A simple program that you could run every couple of minutes to upload any pending readings.

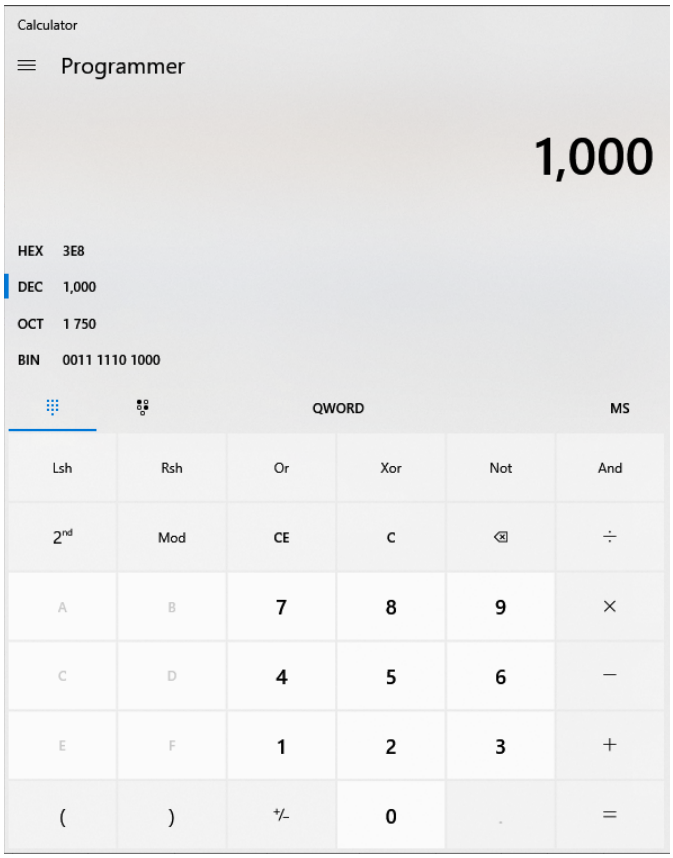
How to calculate a 9502 or 9501 configuration hex value.

The format of the 9501 or 9502 arguments can be calculated using the grid below:

SERVICE - CONFIG- 0x9500											
CHAR NAME	UUID	BYTE1	BYTE 2	BYTE 3	BYTE 4	BYTE 5	BYTE 6	BYTE 7	BYTE 8	BYTE 9	BYTE 10
SAMPLES	0X9501	SAMPLES 31...24	SAMPLES 23...16	SAMPLES 15...8	SAMPLES 7...0						
CONFIG	0X9502	FREQ	SLEEP 31...24	SLEEP 23...16	SLEEP 15...8	SLEEP 7...0	TX PWR	TX RATE	NOISE_ REDUCTION	ADV_ TIMEOUT	SENSOR_ SETTLE

Please see the chart above for the configuration of each of the bytes strings that represent 9501 and 9502. Open the computer calculator app, and navigate to the programmer section.





Once there, type the value to convert (i.e. 1000). The text next to the label for "HEX" will display something like "3EB." Take this value and pad '0' on to the left until the length of the text is 8 characters (i.e. 000003EB). That will become the value for the c9501 argument (i.e. "-c9501 000003EB").

9502

9502 is a bit trickier in that there are several variables bundled into this byte string. See the table from the beginning of this section for reference.

Byte 1 is the sample rate: use the calculator to calculate the hex string for a value between 1 and 40. A value of 1 (01 in hex) represents 1 kHz, and a value of 42 (2A in hex) represents 40 kHz. If the output is less than 2 characters, pad the left with '0' until the length is 2 characters.

Byte 2 – 5 is the device sleep time; this is the amount of time the device will "turn off" in-between advertising cycles. The default in our software is 10 seconds, but this value can be set anywhere from seconds to years. **Warning:** Misuse of this field will result in a sensor that virtually never returns from sleep.

Byte 6 is the transmit power. Use this table to determine what your value should be:

TX PWR VAL	TX PWR	TX PWR VAL	TX PWR
0	Don't Change	7	2 dBm
1	-40 dBm	8	3 dBm
2	-20 dBm	9	4 dBm
3	-16 dBm	10	5 dBm
4	-8 dBm	11	6 dBm
5	-4 dBm	12	7 dBm
6	0 dBm	13	8 dBm *default*

Use the calculator to calculate the hex string for a value between 1 and 13. A value of 1 (01 in hex) represents -40 dBm, and a value of 13 (2A in hex) represents 8 dBm. If the output is less than 2 characters, pad the left with '0' until the length is 2 characters. Additionally, 00 can be used to use the last value entered.

Byte 7 is deprecated, use the value 00 for this section.

Byte 8 is noise reduction, use this table to determine your value.

NOISE_REDUCTION VAL	LEVEL
0	Don't Change
1	Off
2	Low (Oversample = 4) *default*
3	Medium (Oversample = 8)
4	High (Oversample = 16)

Note: Medium requires sample rate less than 24 kHz

Note: High requires sample rate less than 12 kHz

Use the calculator to calculate the hex string for a value between 1 and 4. A value of 1 (01 in hex) represents no noise reduction, and a value of 4 (04 in hex) represents high noise reduction. Make sure if the output is less than 2 characters, pad the left with '0' until the length is 2 characters. Also, 00 can be used to use the last value entered.

Byte 9 is advertising timeout, use this table to determine your value.

Use the calculator to calculate the hex string for a value between 1 and 4. A value of 1 (01 in hex) represents an Advertising Timeout of 10 seconds, and a value of 255 (FF in hex) represents no timeout. If the output is less than 2 characters, pad the left with '0' until the length is 2 characters. Additionally, 00 can be used to use the last value entered.

Byte 10 is the sensor settle time. For this byte, choose a value between 01 and 255. A value of 1 (01 in hex) meaning .01 second and 255 (FF in hex), meaning 2.55 seconds.

Bytes 11 and 12 are in place for future enhancements. By default this value should be four zeroes (0000).

Finally, take all the values calculated for each byte and string them together like this:

0F + 0000000A + 0D + 00 + 02 + FF + FF = 0F0000000A0D0002FFFF0000

Double check that the length of the string is 24 characters long; if not, go back and recalculate the values.

Warning: Entering invalid data could put the sensor into an irretrievable state, so be sure to triple-check any calculations.

Reader Output

The basic structure of the reader's output is as follows:

{time} - {name}{channel} - [connection status] - [request status] -[status note]

Interpreting a single line of output:

{time} – The time of the current diagnostic

{name} – The assigned name of the sensor, maximum ten characters

{channel} – The diagnostic output channel

[Connection status] - {Available}{Connecting}{Connected}{Ready}

Output	Definition
1000	The reader has located the sensor
1100	The reader is attempting a connection
1110	The reader successfully connected to the sensor
1011	The sensor is available to take a reading

Note: 1011 will display for each stage of a reading. Refer to the status note to determine the individual stage of the reading.

[request status] – {Watched}{Staged}{Requested}

Output	Definition
100	Watched
110	Staged
101	Requested

Below are some sample status outputs that will display when the reader is running:

14:39:47 - CTCWS_58E51 - 1000 - 100 – Available

14:39:47 - CTCWS_58E51 - 1100 - 101 – Connecting

14:39:52 - CTCWS_58E51 - 1110 - 101 – Connected

14:39:56 - CTCWS_58E51 - 1011 - 101 – Ready

14:39:57 - CTCWS_58E51 - 1011 - 101 – Requesting

14:40:00 - CTCWS_58E51 - 1011 - 101 – Receiving

14:40:01 - CTCWS_58E51 - 1011 - 101 – Complete

CTC Wireless Project

The Wireless Application - Quick Start

The Downloadable Application - The current build can be run on any Windows 10 computer that is capable of Bluetooth® 4.0 or higher. If you are not interested in any of the development discussion, and just want to run the current software.

- *Quick Start Installer and User Guide*

The Wireless Application - For Developers

The application is made up of 3 pieces of software, which are discussed further below.

- *Server* - Simple RESTFUL API with underlying database
- *Client* - HTML Interface implemented in HTML, CSS, and Javascript
- *Receiver* - Windows 10 / BLE 4.0+ Capable Device that supports UWP

The Wireless Sensor - For Developers

- *Bluetooth® and Configuration Information* - An Excel Document outlining the Bluetooth® Services and Characteristics and acceptable values for each Characteristic.

Server

The server acts as a central repository for all readings, sensor data, and access point data. The client, as well as the reader, connects to the server to retrieve and update sensor, access point, and reading information.

Technology Used / Requirements

- .NET Core 2.2 - Compatible with Windows, Mac and Most Linux Platforms
  - Web API - The application uses Web API as the communication mechanism between the server, receiver, and client.
  - REST - Most of the data objects are exposed through a RESTFUL format. However, access to the core RESTAPI has been disabled for security purposes; the functions critical to the operation of this application are available through a supplemental implementation.
  - SingnalR - A socket-like communication framework that servers can use to communicate with HTML/Javascript front ends.
  - Dapper - Dapper has been used as an alternative to the Entity Framework. Dapper provides excellent raw execution performance with minimal overheads and allows the user to retain control over the SQL.
  - JSON - JSON is the default technology for Web API, and has been selected for this project, being more lightweight and less strict than XML.
- Microsoft SQL Server 2012 or Above - While this application should work with any MSSQL2012+, SQL Express was used for the development of the current CTC Wireless Application. SQL Express is a free, world-class database provided as an entry-level offering to their paid enterprise SQL server platform. It is an excellent choice for a small database like this, which gives the implementer room to grow should they someday (or already) want to use the enterprise version. SQL Express can run on Windows, Linux, and Docker.
- Other Databases - While we have focused on our implementation to run on MSSQL since the server application is the only entry point into the database interaction, it would be a relatively easy modification to communicate with other database platforms such as MySQL, SQL Lite or Oracle.

Data Structures

Primary Data Structures

```
/// <summary>
/// A BLE device
/// </summary>

public class Device

{
    public Guid? id { get; set; }
    public string name { get; set; }
    public string description { get; set; }
    public string notes { get; set; }
    public string addr { get; set; }
    public int? samples { get; set; }
    public int? frequency { get; set; }
    public int? sleeptime { get; set; }
    public int? txpower { get; set; }
    public int? txrate { get; set; }
    public int? noisered { get; set; }
    public int? advtimeout { get; set; }
    public int? centralwait { get; set; }
    public string master { get; set; }
    public string sleeptimeuom { get; set; }
    public string advtimeoutuom { get; set; }
    public string centralwaituom { get; set; }
    public int? settle { get; set; }
    public string stat { get; set; }
    public string dt { get; set; }
    public bool? stage { get; set; }
    public int? battery { get; set; }
    public bool? request { get; set; }
    public bool? hidden { get; set; }
    public Guid? sensor { get; set; }
    public Guid? devicegrp { get; set; }
}

/// <summary>
/// A BLE device reading
/// </summary>
```

```
public class Reading

{
    public Guid? id { get; set; }
    public string dt { get; set; }
    public string addr { get; set; }
    public int? samples { get; set; }
    public int? frequency { get; set; }
    public int? noisered { get; set; }
    public string points { get; set; }
    public int? settle { get; set; }
    public string notes { get; set; }
    public Guid? device { get; set; }
    public Guid? sensor { get; set; }
}

/// <summary>
/// A BLE Access Point
/// </summary>

public class Accesspoint

{
    public Guid? id { get; set; }
    public string name { get; set; }
    public string description { get; set; }
    public string dt { get; set; }
    public string addr { get; set; }
    public string devices { get; set; }
}

/// <summary>
/// A Physical Sensor Readings are taken from (I.E. AC314-1A)
/// </summary>

public class Sensor

{
    public Guid? id { get; set; }
    public string name { get; set; }
    public string description { get; set; }
    public string serial { get; set; }
    public string sensitivity { get; set; }
  }
```

```
/// <summary>
/// A Grouping of Devices
/// </summary>

public class Devgroup

{
    public Guid? id { get; set; }
    public string name { get; set; }
    public string description { get; set; }
}
```

Shared Data Structures

```
 /// <summary>
/// Copy and paste of the classes that are in CTCWS
/// </summary>

public class SlimBluetoothLEDevice

{
    public ulong BluetoothAddressAsUlong { get; set; }
    public string SamplesPerSecond { get; set; } = "1";
    public string SamplesToTake { get; set; } = "1000";
    public string TimeToSleep { get; set; } = "60";
    public string TransmitPower { get; set; } = "13";
    public string TransmitRate { get; set; } = "2";
    public string NoiseReduction { get; set; } = "2";

    public string AdvertisementTimeout { get; set; } = "30";
    public string TimeToWait { get; set; } = "120";

    public string DeviceName { get; set; } = "No Name";
    public string DeviceDescription { get; set; } = "";
    public string DeviceNotes { get; set; } = "";
    public bool DeviceMaster { get; set; } = false;

    public string TimeToSleepUOM { get; set; } = "S";
    public string TimeToWaitUOM { get; set; } = "S";
    public string AdvertisementTimeoutUOM { get; set; } = "S";
    public string Settle { get; set; } = "200";
    public bool Stage { get; set; } = false;
    public bool Request { get; set; } = false;
}
```

```
/// <summary>
///  The data contained within a reading
/// </summary>

public class WSReading

{
    public DateTime dt { get; set; }
    public SlimBluetoothLEDevice dev { get; set; }
    public Dictionary<int, WSPacket> Packets { get; set; } = new Dictionary<int, WSPacket>();
    public string Notes { get; set; } = "";
}

/// <summary>
/// A single packet of data within a reading
/// </summary>

public class WSPacket

{
    public int Num { get; set; }
    public bool Waiting { get; set; }
    public List<int> Points { get; set; } = new List<int>();
}
```

Interface/Client Data Structures

```
/// <summary>
/// A collection of the reading and fft
/// </summary>

public class WaveData

{
    public bool setStageFalse = false;
    public Conn.Reading reading { get; set; }
    public string fft { get; set; }
}

/// <summary>
/// A simple version of the reading for the search results
/// </summary>

public class SimpleReading
```

```
{
    public Guid id { get; set; }
    public string notes { get; set; }
    public string dt { get; set; }
}

/// <summary>
/// an opject to do a partial update on the status of an object
/// </summary>

public class StatUpdate

{
    public ulong BluetoothAddressAsUlong { get; set; }
    public string Status { get; set; }
    public int Battery { get; set; }
}
```

Tertiary Data Structures

```
/// <summary>
/// A Generic SearchResult
/// </summary>
/// <typeparam name="T"></typeparam>

public class SearchResult<T>

{
    public int ct { get; set; }
    public T obj { get; set; }
}

/// <summary>
/// A Count Object
/// </summary>

public class Count

{
    public int ct { get; set; }
}
```

Application Flow

There is no real flow to the server component of this application, as each endpoint represents a stateless action to the interface. Listed below are the endpoints the GUI and Receiver portions of the application use.

Device Methods

- url: api/Device, method: GET
  - This endpoint returns all sensors that have been added to the database.
- url: api/DeviceImpl2, method: GET
  - This endpoint returns all sensors that have been added to the database in the SlimBluetoothLEDevice format used by the reader application.
- url: api/DeviceImpl2, method: POST
  - This endpoint updates the staged status of a sensor.
  - It expects a Device object to be passed as JSON in the body of the request.
- url: api/DeviceImpl2, method: POST
  - This endpoint receives a reading, appends it to the database, and creates the device if it doesn't already exist. It also creates FFT and WAV files.
  - It expects a WSReading object to be passed as JSON in the body of the request.
- url: api/Device/{Device ID}, method: PUT
  - This endpoint updates the reading settings of a sensor.
  - {Device ID} should be the valid GUID of a sensor.
  - It expects a Device object to be passed as JSON in the body of the request.
- method: DELETE, url: api/DeviceImpl2/{Device ID}
  - This endpoint deletes a sensor and all of its readings from the database.
  - {Device ID} should be the valid GUID of a sensor.

Access Point Methods

- method: GET, url: api/Accesspoint
  - This endpoint will return all the access points that have been added to the database.
- method: GET, url: /api/AccesspointImpl2?id={Access Point ID}
  - Gets the information for a single access point.
  - {Access Point ID} should be the valid GUID of an access point.

- method: PUT, url: api/Accesspoint/{Access Point ID}
  - This endpoint updates the settings of a specific access point.
  - {Access Point ID} should be the valid GUID of an access point.
  - It expects an Access Point object to be passed as JSON in the body of the request.
- method: DELETE, url: api/AccesspointImpl2/{Access Point ID}
  - This endpoint deletes a specific access point from the database.
  - {Access Point ID} should be the valid GUID of an access point.

Reading Methods

- method: PUT, url: api/ReadingImpl2/{offset}/{limit}
  - This endpoint allows a partial set of data to be returned. This is useful for paging data.
  - It expects a Reading Guide to be passed as JSON in the body of the request.
  - {offset} is where to set the current lookup, while {limit} limits the number of results returned.
  - This method is a PUT because its base implementation allows the PUT of a full reading object for the purpose of searching.
- method: DELETE, url: api/ReadingImpl2/{Reading ID}
  - This endpoint allows for the deletion of a specific reading.
  - {Reading ID} should be the valid GUID of a sensor.
- method: GET, url: api/ReadingImpl2?xid={Device ID}
  - This endpoint gets the very last reading for a specific sensor.
  - {Device ID} should be the valid GUID of a sensor.
- method: GET, url: api/ReadingImpl2/{Reading ID}
  - This endpoint gets a specific reading.
  - {Device ID} should be the valid GUID of a sensor.
- method: PUT, url: api/ReadingImpl2
  - This endpoint allows the notes of a reading to be updated.
  - It expects an Access Point object to be passed as JSON in the body of the request. Only the notes will be allowed to pass through for updates.
- method: POST, url: api/ReadingImpl2
  - This endpoint allows the status of a reading to be updated.
  - It expects an Access Point object to be passed as JSON in the body of the request. Only the status will be allowed to pass through for updates.

Group Methods

- method: "GET", url: api/devgroup"
  - This endpoint will return all the groups that have been added to the database.
- method: = "POST", url: api/devgroup/";
  - This endpoint creates a new group.
  - It expects a Group Object to be passed as JSON in the body of the request.
- method: = "PUT", url: api/devgroup/{Group ID}
  - This endpoint updates the settings of a specific group.
  - {Group ID} should be the valid GUID of a group.
  - It expects a Group Object to be passed as JSON in the body of the request.
- method: "DELETE", url: api/devgroup/{Group ID}
  - This endpoint deletes a specific group from the database.
  - {Group ID} should be the valid GUID of a group.

Sensor Methods

- method: "GET", url: api/Sensor"
  - This endpoint will return all the sensors that have been added to the database.
- method: = "POST", url: api/Sensor/";
  - This endpoint creates a new sensor.
  - It expects a Sensor object to be passed as JSON in the body of the request.
- method: = "PUT", url: api/Sensor/{Sensor ID}
  - This endpoint updates the settings of a specific group.
  - {Sensor ID} should be the valid GUID of a sensor.
  - It expects a Sensor Object to be passed as JSON in the body of the request.
- method: "DELETE", url: api/Sensor/{Sensor ID}
  - This endpoint deletes a specific sensor from the database.
  - {Group ID} should be the valid GUID of a sensor.

Common Methods

- method: "PUT", url: api/{Type}/name/{Offset}/{Results Per Page}/1"
  - This method retrieves a paged result of object of a particular type sorted alphanumerically by its name.
  - {Type} - The type of object being queried (sensor, devgroup, reading, device, accesspoint).

- {Offset} - The paged offset - start at the defined value.
- {Results Per Page} - the number of results to return.
- It expects an object of { name: {searchtext}} to be passed as JSON in the body of the request.

Client

The client to the software is a simple single-page web app. It appears as the default page shown by the CTC Wireless Application, or can also be browsed to by pulling up the address of a CTC Wireless Server in a browser window.

Technology Used / Requirements

- CefBrowser - CefBrowser is a Chromium web pane that can be drawn onto any .NET Framework Web Form or Windows XAML Window.
- Angular JS and Bootstrap 4 are common web development frameworks that were used building the client application.
- FontAwesome - is used for icons and symbols.
- SignalR - is used for socket-like communication with the server.

Data Structures

Except for the shared data structures, the client application uses the same data structures as the server. Please review the server section to understand which data objects are being passed back and forth between the server and the client.

Application Flow

There are 4 entry points into the application, which are represented by 4 URL formats used by the angulars routing engine.

```
 $routeProvider

// The default view - devices and groupings

.when("/", {
    templateUrl: "views/main.html"
})

// The device view - defaults to current reading data

.when("/sen/:devid", {
    templateUrl: "views/red.html",
})

// The Reciever/AP view - see all recievers and APs

.when("/aps/", {
    templateUrl: "views/apmain.html",
})
```

```
// looking at one ap - not implemented

.when("/aps/:apid", {
    templateUrl: "views/apred.html",
})

// The Sensor view - see all sensors

.when("/sensor/", {
    templateUrl: "views/sensormain.html",
})

// looking at one sensor - not implemented

.when("/sensor/:sensorid", {
    templateUrl: "views/sensorred.html",
});
```

Initialization

The application behaves like any other Angular JS application and is kicked off by the ng-init directive.

```
<div id="content" ng-app="ctcws.app" ng-controller="AppCtrl" ng-init="initialize()">
```

The initialize function does the following:

- Starts up a SignalR Connection with the server.
- Sets an interval to refresh the data on the page.
- Initializes all the fields and their defaults.
- Refreshes the data - As long as there is nothing in process, the refresh fetches the following data.
  - All Devices
  - All Access Points
  - All Sensors
  - All Groups

Main Application

At this point, the application is live and the rest of the program is based on events that come from the server, the automatic refreshes, or user actions.

The user can choose to change:

- Devices: (Edit, Delete, Update, Hide)
  - A reader discovering a new device handles creation.
  - The view feature brings the user to a window where they will see the latest reading.
  - Reading: (Edit Notes, Delete)
    The reading pane also has a path to get to the history of readings stored by the database.
- Sensors: (Create, Edit, Delete, Update)
  - Viewing Sensors is not implemented.
- Device Groups: (Create, Edit, Delete, Update)
  - Viewing Device Groups is not implemented.
- Access Points: (Edit, Delete, Update, Hide)
  - Creation is handled by a server being notified by a new reader.
  - Viewing Access Points is not implemented.

All of the classes of objects in the system utilize roughly the same set of functions for CRUD operations as well as lookup and searching.

SelectableList Component

The 'selectableList' component is created for the management of objects within the application. It significantly reduces some of the mundane coding required to set up the interfaces for managing individual objects and has been used extensively throughout this application.

```
<selectable-list array="appdata.devices" val="appdata.seldeviceid"
    edit="true" sl-class="col-6" folder="devices"
        filter="{devicegrp: '!'}"
    selectfcn="selectDevice" pre-validationfcn="saveDevice"
    saveselect="false" cancelfcn="canDevice"
    loadfcn="loadDevice" delete="true" delfcn="delDevice"
    other="{timeuom:appdata.timeuom,freqs:appdata.freqs,
    noises:appdata.noises,txpowers:appdata.txpowers,
    settles:appdata.settles,showhidden:appdata.showhidden,
    advtimeouts:appdata.advtimeouts,adtimeoutchg:adtimeoutchg}"
    othfcn1="hideDevice"></selectable-list>
```

The 'selectableList' allows the user to show a collection of an object type and provides access to edit, delete, add, and update functionality by exposing these calls to functions in the appropriate scope. Every 'selectableList' is defined with a folder that points to its list and modal templates; if no folder is assigned, the primary folder is chosen by default. For creating unique templates, its best to make a copy of the primary folder, as much about the way these lists work can be derived from the functionality presented by the basic templates. These template folders can be found in the lists folder.

Within a 'selectableList,' a collection is shown by rendering a set of bootstrap divs; this is managed by a list template that defines how each list will appear and which shared buttons are available. When a user presses the Add or Edit buttons, they are presented with a modal screen, which is driven by a modal template of the 'selectableList' folder. The modal is a form that should have fields that represent the type of object being edited. The modal contains some functionality for validation, and any other utilities common of Angular JS templates.

Typical functions include: Save, Delete, Cancel, Edit/Load, Set, and Select. Pressing the save button invokes a save operation on the main scope. Pressing the delete button on the list invokes a delete operation on the main scope. When a modal opens, a load function is fired that allows for added functionality to be performed before a modal fully loads. A Select function on the list allows for custom functionality on the main scope, which redirects to a new URL governing the object the user has selected. A cancel function exits the modal and does not save changes.

Here is a basic outline of the functions that are paired with each class of objects needed by the selectable list:

1. $scope.select{Type} = function (x, y, z)
   a. Happens when the user selects an object. It redirects to the URL that governs that object. The redirection causes the routing change to fire, and the interface will transport the user to the applicable pane on the interface. This is implemented primarily by the device object.
2. $scope.save{Type} = function (x, cb)
   a. Gives the user a new set of data that can be saved to the database.
3. $scope.can{Type} = function (x, cb)
   a. Primarily cancels the modal popup.
4. $scope.load{Type} = function (ob, tx, oth)
   a. Allows the user to edit the data of an object as it is loaded into the modal.

5. $scope.del{Type} = function (x)
   a. Allows the user to handle the delete action of a selectableList object.
6. $scope.setSel{Type} = function (x, y)
   a. The reaction to the change to the route, resulting in an object being selected, and additional functionality being performed.
7. function set{Type}(x)
   a. The actual code that selects the particular object.

The use of 'selectableList' controls much of the front end tasked with managing object properties for this application, and the navigation needed for seeing different views for a given object.

Routing

This application uses the angularjs $routeProvider to determine which screen the user should be seeing based on the URL they are visiting.

Most importantly, when a user selects a device, the 'selectableList' component invokes the 'selectDevice' function, which redirects our screen to the correct URL that governs that device. The redirection invokes angularjs's $routeChangeStart event which makes variable adjustments in the program, so the user is transported to the correct window on the interface. As is standard with Angular JS routing, the whole sequence of routing events will also occur if a user navigates to the same URL for a specific device. By the presence of a specific template URL determined by Angular JS routing or by the presence of a specific object not being null, it will send the user to the appropriate screen.

SignalR

The initialization sequence starts up a connection to a SignalR server. The SignalR server is used to transmit new readings and status information to users logged into and watching the live updates on a specific device.

Readings

When the server receives a reading, it is broken apart into its wav and fft components. Sensitivity information is gathered from the linked sensor, and calculations are performed on the reading and fft to convert its value to g or IPS or another UOM provided by the sensor. If no sensor is linked, the program defaults to display mV. The

end arrays of readings and fft data are then sent to the 'drawGraph,' which prints them to the screen using the 'Dygraph' javascript graphing library.

The same functionality applies if a user browses to a historical reading, or when the user first connects to a device which brings up the very last reading. The only difference is that these historical readings are read through WebAPI calls.

Status Updates

As the sensors connect and prepare to take readings, a series of status messages is sent to the SignalR server, which is then logged and also passed back to any users signed into the device, which is broadcasting its status.

Requesting, Staging or Executing a Reading

The final purpose SignalR serves is to send messages when a user checks the Request/Stage check-boxes, or click the Execute Reading button on a staged device. These events send messages to the server, which will be passed onto the reader to signal that something is supposed to happen. The reader reacts by taking said action(s) which result in the initialization of each process that is supposed to occur on the readers.

Receiver/Access Point (Central)

Next to the actual functioning of the sensor, the Receiver/Access point software is the most essential piece of functionality for using the CTC Wireless Sensors. Without a receiver, all the user has are RAW Bluetooth® transactions, which would prove difficult, if not impossible, to complete sensor readings.

Technology Used / Requirements

- Bluetooth® 4.0+ BLE Communication
- Windows 10
- UWP - Universal Windows Platform

CTCWS Project

- CTCWS.BLE.LIB (UWP) - The core library that controls and manages the Bluetooth® connections and all of the data that goes with them.
  - Connection level functionality and logic were borrowed from Microsoft's open-source Bluetooth® LE Explorer project.
  - Minimal edits to the original files and inclusion of partial files afford added functionality required by the CTC Wireless Sensor App.
  - All Deviations are labeled mod or modx.
  - The CTC Sensor file represents the core logic specific to the current iteration of the CTC wireless sensor as of Q4-2019. Said functionality would find one of many sensors in the vicinity, connect to them, set any/all reading parameters, and record a reading.
- CTCWS.BLE.NET.LIB - This is a wrapper class for the UWP functionality for ease of use in .NET Framework.
- CTCWS.BLE.READER - This is a bare-bones command line application that uses the .NET wrapper library to take readings.
- CTCWS.BLE.NET - The GUI part of the project, mainly this serves as a shell to house and start the reader functionality, run or connect to the server, and display web front-end/GUI.

CTCWS.BLE.LIB (UWP), CTCWS.BLE.NET.LIB, CTCWS.BLE.READER

The CTCWS.BLE.LIB project derives from the Microsoft Open Source Project Bluetooth® LE Explorer. Bluetooth® LE Explorer is a generic BLE reader program, that can be used to connect to any BLE sensor, browse its information, and execute raw Bluetooth® functionality. This core functionality of connecting to sensors, enumerating properties,

and setting/getting characteristic data has primarily been reused from Microsoft's Bluetooth® LE Explorer project. There was enough new functionality specific to the CTC Wireless Transmitter that it derives from the project rather than forks development. As a result, the CTC Wireless App has become a new software development on its own.

The Bluetooth® LE Explorer was MIT licensed by Microsoft, allowing CTC to use the code and make modifications to achieve the desired result. We have used this benefit to create the CTC Wireless App, a program that communicates with a CTC Wireless Transmitter to change configuration settings and ultimately take wireless vibration readings. The files that were reused from the Bluetooth® LE Explorer have been kept as unchanged as possible, keeping the Microsoft copyright intact, and also allowing for the merger of any future development by Microsoft into said files. For more information on Bluetooth® LE Explorer, visit the GitHub page for *Bluetooth® LE Explorer*. Additionally, the Bluetooth® LE Explorer user application is freely available through the Microsoft Store and can serve as a useful side program to the CTC Wireless App. It was utilized during development for performing diagnostics and debugging CTC Wireless Transmitters outside of the CTC Wireless App.

Examining the project

Upon opening the CTCWS.BLE.LIB project, the user will notice three main folders (GattHelper, Models, and Services). Everything in the 'GattHelper' and 'Services' folders were directly borrowed from the Bluetooth® LE Explorer project and contain utility classes that serve functions useful for data conversion and BLE protocol definitions. These are important to the underlying Bluetooth® communication but do not serve the CTC Wireless Reader with any direct functionality. The Models folder houses the important functionality for the CTC Wireless Reader. Take note of the file naming convention. Any files that do not end with modx.cs or mod.cs (except CTCSensors.cs) are the core files that are packaged with Bluetooth® LE Explorer. These are the same files that were in the Bluetooth® LE Explorer project, but have had small changes made to get them to work with the rest of the CTC code. Any files labeled modx.cs or mod.cs are modifications to the original program, and finally, the single file named CTCSensors.cs is the core logic for working with a CTC Wireless Sensor. The first part of the name of sets of files are the same. This is because any file with the same first part of its file name belong to the same C# class. Using C# partial files allowed for this functionality.

Application flow

CTCWS.BLE.READER and CTCWS.BLE.NET.LIB act as a segway into describing the application flow of the CTCWS.BLE.LIB project. CTCWS.BLE.READER is a straightforward project which collects some command line arguments and passes them onto a series

of properties in the CTCWS.BLE.NET.LIB class called 'NETGattSampleContext.' These properties control such behavior as which server to connect to, what folder stores output, how to communicate a timeout, and what id/names to use for identification. 'NETGattSampleContext' also exposes a 'load()' function that starts the reader routine, which serves as the primary means of creating and executing an instance of the underlying GattSampleContext class. The instance of 'NETGattSampleContext' represents the receiver as an object to all of the implementing .NET Framework programs. It also handles all of the communication with the server and file system. Communications include transmitting status updates and readings or storing configurations and pending reading files to the file system. 'NETGattSampleContext' also is a wrapper class to the CTCWS.BLE.LIB(UWP) class 'GattSampleContext,' which holds all the functionality required to communicate with a CTC Wireless Transmitter. The whole reader process is started from the load() function through a call on the GattSampleContext of 'StartEnumeration().'

Flow from within the GattSampleContext

When the 'StartEnumeration()' function begins to run, it uses .NET UWP Bluetooth® functionality to collect advertisements from nearby Bluetooth® devices. It looks closely at each advertisement looking for a device that starts with a prefix of 'CTCWS_.' When it finds these devices, it encapsulates a reference to them in a new instance of its device class called 'ObservableBluetoothLEDevice.' When an ObservableBluetoothLEDevice is created, it also creates and holds onto a reference to an overarching "Work Flow" class that implements the IBLEDevice interface. The new instance is then queued in an array of objects that are available to establish a connection.

A side note on the "class implementing the IBLEDevice interface:" it functions similarly to a "Work Flow" class, which is designed to do all the high-level and implementation-specific communication with connected a device. In the case of this program, it is represented by the CTCSensor class. As mentioned above, this class handles all of the functionality specific to the CTC Wireless Sensor.

Additionally, the 'IBLEDevice interface' has been designed to be independent of the communication protocol (despite its name). The program is built for expansion, allowing the addition of new future implementations of the IBLEDevice interface. This is for connecting to sensors or protocols that may be implemented in the future.

When an end user with the CTC Wireless App places a Request or Stage command on a sensor, a field in the database is set, indicating what the user wants to do. On the reader, each time a device advertisement is received, the application requests updated

data from the server. When the new data is received, if the application sees that the user has placed a command, it initiates a connection to that device.

The connection is initiated through the '_Connect()' function of the ObservableBluetoothLEDevice class. Generally, what the _Connect() function does is make sure everything is in the right state for a connection, and also reset any other state-specific variables and allow the user to connect with a fresh start. Next, the actual connection is run by the native 'Connect()' command (no underscore). The Connect command runs a 'GetGattServicesAsync' on the native BluetoothLEDevice object, which collects all BLE services, and creates 'ObservableGattDeviceService' out of them. This process of enumerating, collecting, and encapsulating continues down to the descriptor level in the BLE structure until the full object is described, and everything is quantified.

Meanwhile, on the 'ObservableBluetoothLEDevice' class, a previously set up CharDiscovered function gets notified for every characteristic that each service finds. This function is waiting for there to be a match to a target characteristic name so that it can be confident that the sensor it is talking to legitimately is a CTC Wireless Sensor. Once this happens, the IsReady flag is set to true. Setting the IsReady to true invokes the 'SensorInit()' function of the CTCSensor(IBLEDevice), which initializes the higher-level logic of the CTCSensor implementation.

Flow from within CTCSensor

In the CTCSensor implementation, the SensorInit function call first gets the reading of the four critical characteristics.

```
<code csharp>
    string w = await o.Read(p9501, DisplayTypes.Hex); // The number of samples to read
    string x = await o.Read(p9502, DisplayTypes.Hex); // The configuration of the sensor
    string y = await o.Read(p9503, DisplayTypes.Hex); // THe max memory size of the sensor
    read only
    string z = await o.Read(p9504, DisplayTypes.Hex); // The sensor status
<code>
```

By now, the CTCSensor object also has a copy of the configuration parameters that the user requested (gathered from the server data). It compiles a hex string of new configuration data for the sensor and compares it to the configuration that was read from the server. If there is a difference, it writes the configuration to the server.

The call also engages notification on the p9601 (Data Received) and the p9504 (Status Changed) characteristics. This creates a listener. A Bluetooth® characteristic is analogous to a property or a field. Engaging a notification registers change notifications to said characteristics.

After resetting a few additional variables, the program is ready to collect some data. If the user made an ordinary data request, all that happens is a value is written to the 9501 (number of samples to read) characteristic. If the user made a Staged data request, then the sensor must be staged by writing a '05'+(number of samples hex) to the 9504(sensor status) characteristic.

```
<code>
    string samples = o.hexFromDec(SamplesToTake, 8);
                //"000007D0"; // 2000 Samples -
                // Watchout the max samples and min sample
                // rate will produce a sensor that samples for 48+ days
                // I dont think the memory will hold this though.
                //await    logData("Requesting Data");
                //string c9501 = samples;
                //var _w = await Write(p9501, c9501);
        // 05 is the command to tell the sensor to go into staging mode
    string mode = "05";
    // Compile the command
    string c9504 = mode + samples;
    // Write the command
    var _z = await o.Write(p9504, c9504);
<code>
```

Doing this should result in receiving a 101 code on the 'StatusChanged()' function. When this is received, the status is updated, which returns to the user, showing an "Execute Reading" button. When the user clicks the button, SignalR sends a command "STAGEGO," which will initiate a reading on any readers connected to a staged sensor.

On the reader, whether the request was initiated directly or through receiving 'STAGEGO,' starting a reading is handled by writing the number of "Samples To Take" to the p9501 characteristic. Once this happens, the user begins to receive data packets on the p9601 (Data Received) characteristic. Data is parsed and converted to int values by the DataPacketRcv() function. All of these values are saved to a List in order of reception until no more packets are received. At the same time, the program looks for a 100 command which indicates that the sensor is finished sending data. At this time a flag is set indicating that the data is ready for validation.

Once the reader has stopped receiving new data for over .3 seconds and the 'ready to validate' flag is set, validation of the data begins. If some packets are missing, the reader starts a loop to request specific packets and attempt further validation. If more than 49 readings were missed, the reading is declared as failed the reader disconnects from the sensor.

If collection of all of the readings is successful, the Success command is executed, and the reader returns the data to the original NETGattSampleContext through an Action object. The NETGattSampleContext receives the data and posts it to the server, or saves it to a file.

Finally, the Reader disconnects from the sensor. That concludes the reading.

CTCWS.BLE.NET

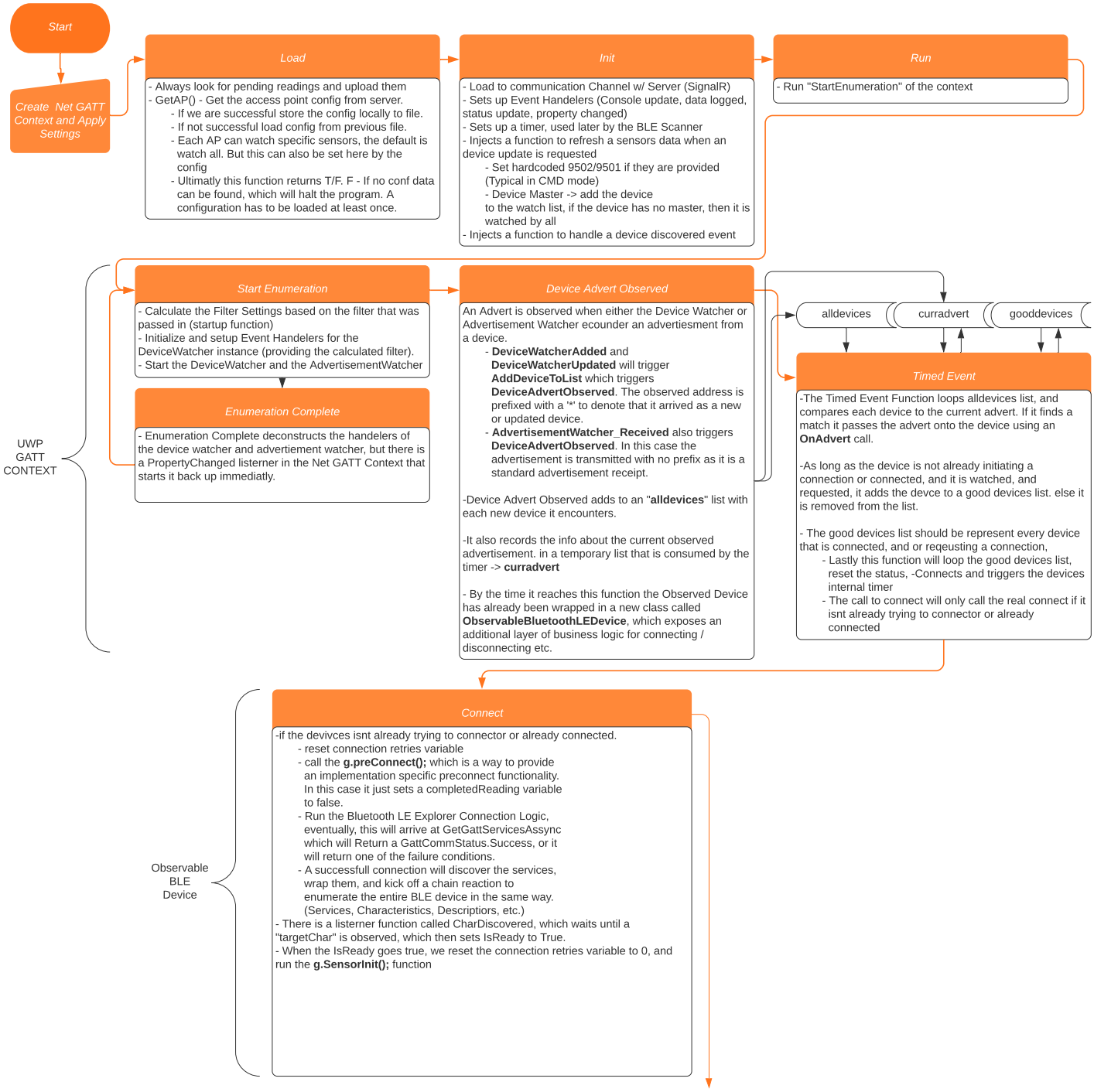
This is the installable portion of the whole project, which has several different configurations.

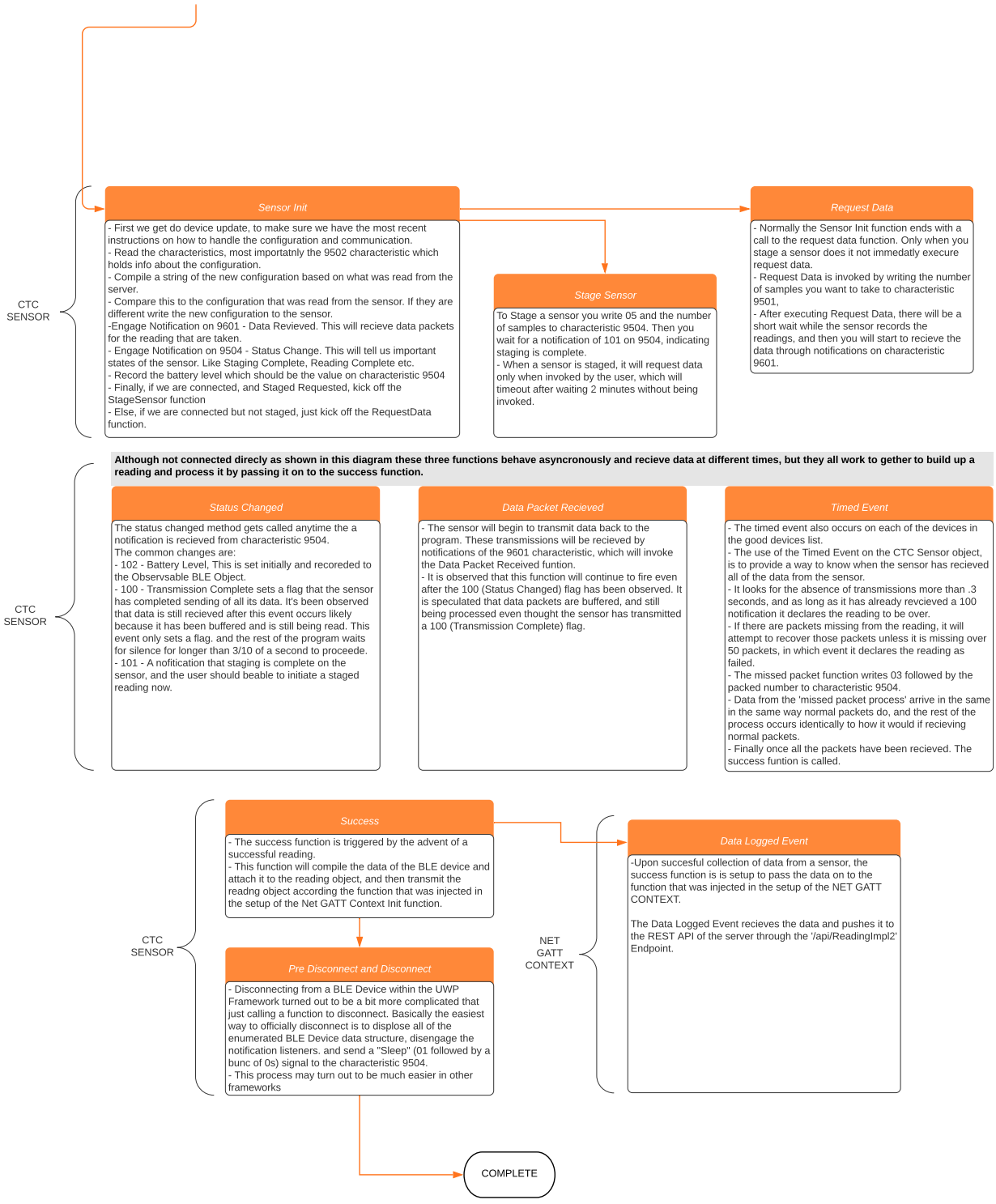
The default configuration is to run an embedded API server, which is described by the server section of this guide. The other parts of default configuration are to run an embedded copy of the reader app, and the interface which is described in the interface section of this guide. Refer to these sections for more information about the different parts of this program.

For more information on the application, please refer to the user guide.

Wireless Access Point Process Flow

Shaun McLarney | March 16, 2020




CTC SENSOR
Sensor Init
- First we get do device update, to make sure we have the most recent instructions on how to handle the configuration and communication.
- Read the characteristics, most importatnly the 9502 characteristic which holds info about the configuration.
- Compile a string of the new configuration based on what was read from the server.
- Compare this to the configuration that was read from the sensor. If they are different write the new configuration to the sensor.
-Engage Notification on 9601 - Data Revieved. This will recieve data packets for the reading that are taken.
- Engage Notification on 9504 - Status Change. This will tell us important states of the sensor. Like Staging Complete, Reading Complete etc.
- Record the battery level which should be the value on characteristic 9504
- Finally, if we are connected, and Staged Requested, kick off the StageSensor function
- Else, if we are connected but not staged, just kick off the RequestData function.
Stage Sensor
To Stage a sensor you write 05 and the number of samples to characteristic 9504. Then you wait for a notification of 101 on 9504, indicating staging is complete.
- When a sensor is staged, it will request data only when invoked by the user, which will timeout after waiting 2 minutes without being invoked.
Request Data
- Normally the Sensor Init function ends with a call to the request data function. Only when you stage a sensor does it not immedatly execure request data.
- Request Data is invoked by writing the number of samples you want to take to characteristic 9501,
- After executing Request Data, there will be a short wait while the sensor records the readings, and then you will start to recieve the data through notifications on characteristic 9601.
CTC SENSOR
Although not connected direcly as shown in this diagram these three functions behave asyncronously and recieve data at different times, but they all work to gether to build up a reading and process it by passing it on to the success function.
Status Changed
The status changed method gets called anytime the a notification is recieved from characteristic 9504.
The common changes are:
- 102 - Battery Level, This is set initially and recoreded to the Observsable BLE Object.
- 100 - Transmission Complete sets a flag that the sensor has completed sending of all its data. It's been observed that data is still recieved after this event occurs likely because it has been buffered and is still being read. This event only sets a flag. and the rest of the program waits for silence for longer than 3/10 of a second to proceede.
- 101 - A nofitication that staging is complete on the sensor, and the user should beable to initiate a staged reading now.
Data Packet Recieved
- The sensor will begin to transmit data back to the program. These transmissions will be recieved by notifications of the 9601 characteristic, which will invoke the Data Packet Received funtion.
- It is observed that this function will continue to fire even after the 100 (Status Changed) flag has been observed. It is speculated that data packets are buffered, and still being processed even thought the sensor has transmitted a 100 (Transmission Complete) flag.
Timed Event
- The timed event also occurs on each of the devices in the good devices list.
- The use of the Timed Event on the CTC Sensor object, is to provide a way to know when the sensor has recieved all of the data from the sensor.
- It looks for the absence of transmissions more than .3 seconds, and as long as it has already revcieved a 100 notification it declares the reading to be over.
- If there are packets missing from the reading, it will attempt to recover those packets unless it is missing over 50 packets, in which event it declares the reading as failed.
- The missed packet function writes 03 followed by the packed number to characteristic 9504.
- Data from the 'missed packet process' arrive in the same in the same way normal packets do, and the rest of the process occurs identically to how it would if recieving normal packets.
- Finally once all the packets have been recieved. The success funtion is called.
CTC SENSOR
Success
- The success function is triggered by the advent of a successful reading.
- This function will compile the data of the BLE device and attach it to the reading object, and then transmit the readng object according the function that was injected in the setup of the Net GATT Context Init function.
Pre Disconnect and Disconnect
- Disconnecting from a BLE Device within the UWP Framework turned out to be a bit more complicated that just calling a function to disconnect. Basically the easiest way to officially disconnect is to displose all of the enumerated BLE Device data structure, disengage the notification listeners. and send a "Sleep" (01 followed by a bunc of 0s) signal to the characteristic 9504.
- This process may turn out to be much easier in other frameworks
NET GATT CONTEXT
Data Logged Event
-Upon succesful collection of data from a sensor, the success function is is setup to pass the data on to the function that was injected in the setup of the NET GATT CONTEXT.
The Data Logged Event recieves the data and pushes it to the REST API of the server through the '/api/ReadingImpl2' Endpoint.
COMPLETE

How close does my tablet need to be to the desired transmitter to receive data?
The tablet must be within 150 ft. of the desired wireless transmitter in order to receive data.

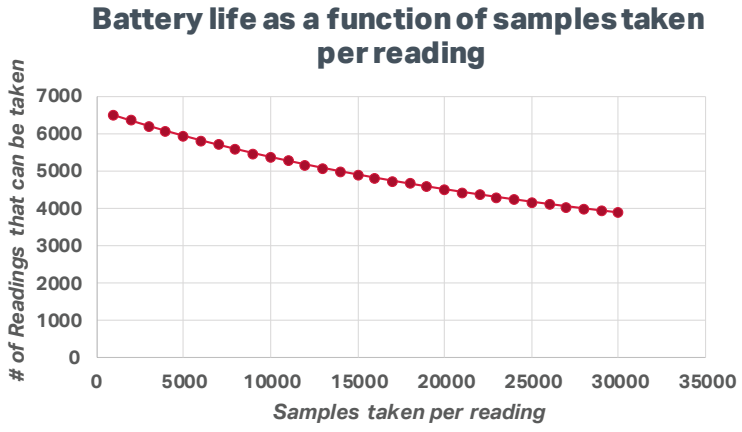
Will my tablet still receive data if there are obstructions between the tablet and the transmitter?
BLE Bluetooth signals will allow for some level of obstructions. The material composition of obstruction will determine the level of signal loss. We have seen moderate signal loss with materials like drywall, and complete signal loss with other materials like reinforced concrete. Any obstructions will produce some level of signal loss and reduce the maximum distance you are able to receive data within. For best performance, Line-of-Sight is recommended.

If my tablet is not within 150 ft. of the transmitter, can I daisy chain transmitters to collect data?
No, the tablet must be within 150 ft. of the desired wireless transmitter.

Does the Transmitter store data?
The Access-Point / Receiver is the primary storage location of data. The transmitter only stores 1 reading at a time, once this reading is transmitted to the receiver and overwritten with the next reading.

How many readings do CTC wireless transmitters take per day?
The number of readings is a user defined setting.

What is the battery life of WA100 Series Transmitters?
Battery life varies depending upon the number & type of readings, and the environment the transmitter is in. The more readings taken per day, the shorter the battery life will be.



Is battery life covered under warranty?
Battery life is not covered under CTC warranty.

Is the battery on my WA100 Series transmitter replaceable or rechargeable?
No, WA100 Series transmitters are disposable units. The molded design seals the transmitter from the environment, and therefore does not allow for replaceable or rechargeable batteries.

Will my WA100 Tablet talk to my data analyzer?
Various analyzers support data import. The CTC Wireless Software that runs on our tablet and access point will provide the data in a variety of common formats that can be used for data import/export.

Is the data processing done in the sensor or the software?
The data processing is done in the software. The Wireless transmitter collects the data, which is transferred to the tablet and processed using CTC wireless software.

Do I own my data?
Yes, CTC wireless transmitter users own 100% of data collected.

Can I "nickname" sensors?
Yes, you can nickname sensors. CTC suggests renaming transmitters to note their location and position on the machine they are monitoring.

Can I create groups of transmitters?
Yes, you can create transmitter groups. Transmitter groups are typically used to logically group all the transmitters associated with a single machine.

What format is my data stored in?
The raw data is fed through Bluetooth Characteristics, the software converts the raw Bluetooth data to a JSON file format which is fed into a Microsoft SQL Database.

How many readings will my tablet store?
Out of the box, the tablet will store over 100,000 readings. The number of readings is only limited by the storage space of the tablet. Storage is expandable through the Micro SD slot.

How can I access historical readings?
Historical readings are accessible through each transmitter reading view in the CTC Software, or directly through the SQL database.

Is there a lag time between when the reading is taken and when I can view my data?
Yes, the lag time is related to how long it takes the transmitter to collect and transmit the signal. For instance, a reading of 10000 data points at 1000 Hz will take 10 seconds to complete, the time to transmit will take a couple of seconds. With more readings, the transmission will take proportionally longer. Signal strength also plays a factor in transmission speed.

What is the IP rating of CTC wireless transmitters?
WA102 carries an IP67 rating, which means it will remain protected and fully operational in almost all industrial applications, including those where the transmitter is exposed to water spray, precipitation, dust, and other airborne fine particles.