# How to add BT devices to the Bluetooth Device Explorer

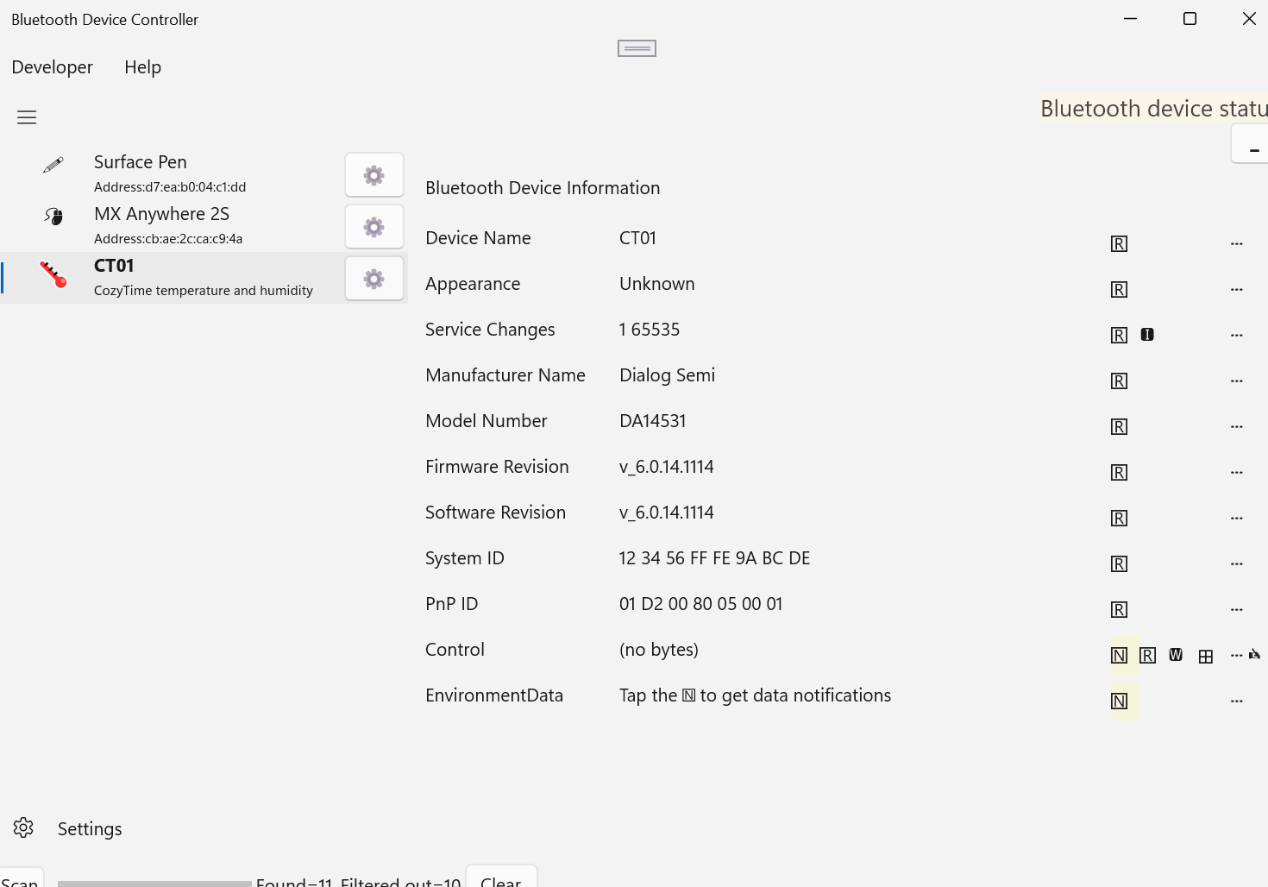
The Bluetooth Device Explorer can be customized to allow a user to find, view, and control specialized Bluetooth devices. A specialized device is one where the device is known to the Bluetooth Device Explorer app and which, when viewed, will show a potentially customized display.

This guide will show you how to add specializations of devices to the Bluetooth Device Explorer.

The sample device is the CozyTime CT01 Thermometer / Hygrometer from <https://www.hypersynes.com/about-1?pgid=m3wtei7g-ed3b0674-2c92-4902-a7fc-c385678ee9c6>. They have a variety of other thermometers that I haven’t tested.

## 10 Get a JSON description of the device

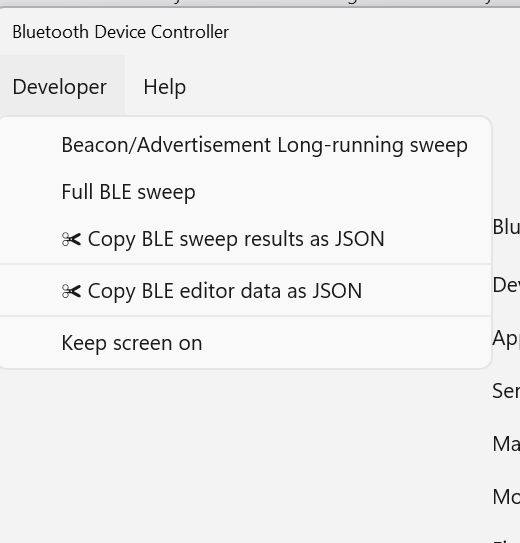
Turn on the device, and with the Settings set to “Show BLE devices with names”, and make sure that “Automatically Read from Device” is checked. Exit the settings and press Search to find the device. Then click on the CT01 device in the list of devices to see the Bluetooth Device Information details.



We will use an automatically JSON description of the device as a starting point and will add additional information so that we can generate a useable Bluetooth protocol C# file and XAML UI file.

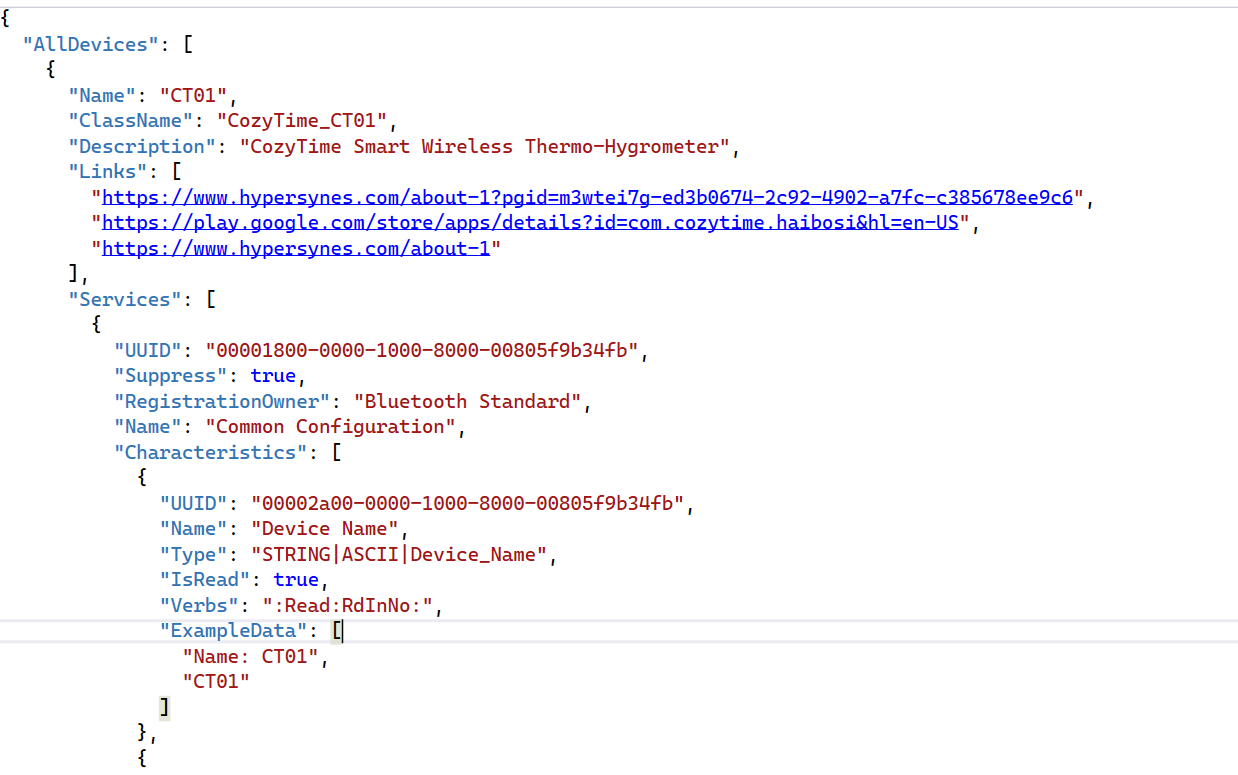
What you need:

* You need to see the device on the screen with a list of its attributes
* You need to know the company that makes the device and the name of the device. These will be used to make names for the files and classes we’re going to create. In this case, we’re going to name this after the Zenggee company because AFAICT this is the company whose protocol the Magic Light uses.

**Copy the JSON description**. Click on the Developer menu and click “Copy BLE editor data as JSON”. This will copy a JSON description of the device onto the clipboard. Run a text editor (I like the Windows 11 Notepad with multiple tabs) and paste the JSON in.   


**Save the JSON file**. Save the file with a name with the pattern “<Maker>\_<Device>.json” in the repo source code directory. Sometimes more generic descriptions get a name with the pattern “<ProtocolName>\_<Type>.json” (like Elk\_LedLight.json). Save it to BluetoothDeviceController\BluetoothDeviceController\Assets\CharacteristicsData.

I saved the file as “CozyTime\_CT01.json” and it looks like this:



## 15 Customize the JSON file

Now we’re going to start customizing the JSON file. The “type” of device is a “NameDevice” in the NameDevice.cs file in the side folder “BluetoothProtocols\Names”; that file has hints on what to add here. **See also the Json\_About.md** **Json\_Graphs.md** **Json\_StringOrientedCommands.md** and **Json\_Tweaks.md** files in the Assets\Help directory; those contain more details information about the JSON format.

Now start updating the JSON to be more specific to the device.

**Name** is already filled in, but it should be trimmed to not include the MAC address of the device. In this case, the name (CT01) is OK..

**ClassName** needs to be added. It is the name of the C# classes we will generate. The standard pattern is [maker]\_[device] using a user-friendly name of both maker and device and where the result is a valid C# class name. I used CozyTime\_CT01.

**ClassModifiers** can be added but is rarely used. It will be inserted into the class definition directly. The most common is the word “partial”; this enables the various classes that are generated to be added to from separate .CS files. A good example of this is the TI SensorTag 1350 (and others) do this if you want to see examples of this used well.

**Description** must be added. It should be a nice description of the device. It will be visible to the end-user.

**DefaultPin** can be added but is often not included (or blank) for modern BT devices that don’t require pairing to work. For devices that do require a PIN for pairing, this is a handy place to stick a reminder of what the PIN is. Is not currently used by the code.

**Aliases** can be added but is often not included (or blank). It’s a list of other devices that share the same protocol. In this case, I don’t have any other devices that use this same protocol.

For an example of when to use aliases, the Triones LED protocol is also used by lights with the name “LEDBlue”. For the Magic Light bulbs, the 7 Watt bulb says it’s “IOTWF8FF” but the 9 Watt bulb says it’s “LEDnetWF”.

**Links** should be added. It’s a list of strings of URLS that point to useful information about the device. For the CT01, I included links to the Google play app, and some to the maker. There weren’t any pages on Github (or anywhere else) that describe the protocol.

**Services and Characteristics Name values**: the available services and characteristics will be filled in for you. But often they don’t have useful names. Change the names into useful values. For the CozyTime, I set the sensor service to “Sensor\_Service” with two characteristics “Control” and “Sensor\_Data”.

**Characteristics Types**: this is a much longer discussion. See a full discussion at [Modern IOT number formats | shipwrecksoftware (wordpress.com)](https://shipwrecksoftware.wordpress.com/2019/10/13/modern-iot-number-formats/). The short, though is simple: each readable characteristic has a specific, known format. There’s no way to automatically detect that format , and sometimes the formats are very complex.

Example: “**I8|DEC|Temp|c U8|DEC|Pressure U8|DEC|Humidity**” for a data sensor means that there are 3 bytes, each a different value. The first is signed (I8) and the next two are unsigned (U8). They should be presented as decimal (not hex) (DEC), and have names Temp, Pressure and Humidity. The Temperature one is degrees c.

(Note on automatically detecting the format: in theory a device can include hints about the data formats it uses. I’ve just never seen a device that actually used them).

The CozyTime company uses a complex and inexplicably difficult scheme for their sensor.

As a hint, it’s helpful to validate your JSON – it helps you find issues with quotes and commas. I just use online JSON validators that I find with a web search (no need to download an app).

## 20 Install the JSON into the app

To use the JSON in the app

Make sure the JSON is in the correct Assets\CharacteristicsData folder.

In Visual Studio, “Add / Existing Item” the file and “Add” it (not “Add as Link”)

Click the file in Visual Studio and set the properties to Build Action=Content and Copy to Output Directory=Copy if newer.

Rebuild and run the app and examine the same device. Be sure to run the app in Release mode (CTRL-F5); the next step will be modifying the project and you can’t do that in Debug mode. The snippet of JSON at the bottom of the screen should reflect your changes:

Graphical user interface, text, application

Description automatically generated

## 30 Create and install the Protocol C# and the UI XAML and C# files

In these steps, you’ll create new files in the Visual Studio projects from data in the running app. This is done with the BluetoothCodeGeneratordotNetCore program.

Go to the correct directory where that program is located; there should also be a Templates directory.

.\BluetoothCodeGeneratordotNetCore.exe -inputJsonFile CozyTime\_CT01.json

This generates a bunch of files in the **output** directory in three sub-directories. Copy the files based on the table below:

|  |  |
| --- | --- |
| **Directory** | **Directory to copy to** |
| BluetoothProtocols | BluetoothDeviceController\BluetoothProtocolsDevice |
| Help | BluetoothDeviceController\BluetoothDeviceController\Assets\HelpFiles |
| SpecialtyPages | BluetoothDeviceController\BluetoothDeviceController\SpecialtyPages |

You will also need to add the files into the BluetoothDeviceController project. There’s a simple Copy\_Output\_Files.bat which will copy them for you, but you have to add them to the project yourself.

Recompile the app. It should compile fine, although Visual Studio now complains (for no good reason) that the AppxBundlePlatform isn’t set. Any errors are likely caused by the JSON file being incorrect.

Now you have to tell the app to use your new code. In the MainPage.cs file, find the List<Specialization> Specializations line. This is a list of all speciality devices that are known. Dice quality as a “fun” specialization, so I added it as a “fun” specialization. Keep the list in roughly alphabetical order.

The specialization list includes

* The page to use (which will be the one you just created)
* A list of strings to match for the Bluetooth device
* The icon to use
* A short string that will be displayed when the device is found, and a longer string which is the hover-over alt-text for the device.

Recompile and re-run the app; the device will show up with the specialization.

Graphical user interface, application

Description automatically generated

## 40 Improving the specialization

The JSON (and the resulting Protocol and UX files) can be improved by setting some of the optional values for the Service.

* Add a “Suppress”:true for services that the user doesn’t need to see. This includes the Common Configuration and the Generic Service (or you can just delete them)
* Add a “Description”:”…” for the transmit service
* Add examples to the transmit and receive characteristics to show the set of valid commands. These are just for programmers; they aren’t used by the program at all.
* Set the priority of the services. Larger priorities are more important, and priorities >= 10 will start off expanded. Services will be listed in priority order.
* Set "AutoNotify": true for the Sensor\_Data. That way data will start to flow in right away.

The final results are below. The Sensor\_Data section is now first, and starts off expanded (because the Priority >= 10). When the Notify button is pressed, the notify event will be enabled and you’ll start to see data come in for the dice.

## 45 Adding Graphs

TODO: how to add charts for data like from the CT01!

## 50 Proper functionality: Customizing the Protocol file

Add a \_Custom protocol C# file to add additional features to the device protocols. The Protocol C# file provides a clean interface into the messy requirements of handling Bluetooth. It can only support features that the JSON includes. By making the class a partial class (as was done here), the automatically-generated C# protocol file can be left unchanged (and potentially regenerated as better templates are developed); all of the customization is done in a separate C# file.

## 60 Add to the XAML and move to the Custom directory

Update the XAML file to make it exactly how the device should work, including getting data that will be needed. When you do this, be sure to move the XAML and the backing CS file to the SpecialtyPagesCustom directory.

The SpecialtyPages and the Protocols directories are designed so that the files can be quasi-automatically replaced based on the JSON protocol files in the Assets\CharacteristicsData directory.

Graphical user interface, application

Description automatically generated

## 70 Add images to Assets\DevicePictures and Assets\ScreenShots

You’ll need some images of the device with the correct name (it’s how they are found). In the Assets\DevicePictures directory add two images with the name <Manufacturer>\_<Device>-175.png and <Manufacturer>\_<Device>-350.png. The -175 must be 175x175 pixels and the -350 is 350x350 pixels.

You will also need screenshots (most likely) for your help file (which is discussed below). Because every device is different, there isn’t a fixed set of screenshots you need to make. They should all be named <Manufacturer>\_<Device>\_<More-info>.png

The images must be correctly added to Visual Studio

1. Add them to the correct Asset directory
2. Make sure each image is type “content”
3. Make sure each image is set to “Copy when newer”

## 80 Create Assets\HelpFiles\Device\_<Manufacturer>\_<Device>.md and link it in

Create a Markdown (.MD) help file. Use other files as a template.

Link to the markdown file from both the Welcome.md file (which will include a picture) and the Help.md file (which doesn’t). Both of these add in devices by category; be sure to pick a good category.