# 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 GoDice from Particula, a company based in Flushing, New York. Ahead of time I did a little research; there’s already a simple code sample from the Particula company on GitHub at [ParticulaCode/GoDiceJavaScriptAPI: GoDice JavaScript API](https://github.com/ParticulaCode/GoDiceJavaScriptAPI).

## 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 GoDice device in the list of devices to see the Bluetooth Device Information details.

Graphical user interface, application

Description automatically generated

In this case, the device shows up with a telephone icon and the legend “BLE UART COM port” because the device uses the same services and characteristics of the Nordic UART service. This, alas, is all to common in the Bluetooth world: so many creators don’t properly set up their services and characteristics even though it’s pretty easy to set up services and characteristics correctly ([link](https://sunriseprogrammer.blogspot.com/2017/05/your-bluetooth-is-bad-and-you-should.html)).

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 it’s 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. The company is **Particula** and the device is **GoDice**.

Click on “Copy Language” and select “JSON”. This will copy a JSON description of the device onto the clipboard. Run a text editor (I like Notepad++) and paste the JSON in. Save the file with a name with the pattern “<Maker>\_<Device>.json” in the repo directory BluetoothDeviceController\BluetoothDeviceController\Assets\ChacteristicsData.

Yes, the word Characteristics was misspelled in the folder name. Sorry about that. I saved the file as “Particula\_GoDice.json” and it looks like this:

Text

Description automatically generated

Now we’re going to start customizing the 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.

**Name** is already filled in, but it should be trimmed to not include the MAC address of the device. I trimmed the name to GoDice\_.

**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 Particula\_GoDice.

**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. The TI SensorTag 1350 (and others) do this if you want to see examples of this used well. In this case, I knew that I’d want to specialize the protocol class, so I set this to “partial”.

**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. For example, the Triones LED protocol is also used by lights with the name “LEDBlue”.

**Links** should be added. It’s a list of strings of URLS that point to useful information about the device. For the GoDice, I added the ordering page, the Kickstarter page, and the official GitHub repo.

**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 GoDice, most of the names were already OK; I changed the “Transmit” service to “DiceTransmit” so it wouldn’t be the same as the “Transmit” characteristic.

**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 GoDice, curse their eyes, used a bizarre and inexplicable difficult scheme for their dice. There are three commands to send to the dice which are not ASCII. And there are 7 events in return where the first byte is always ASCII, but sometimes the command is 3 ASCII chars (Bat and Col) and the rest are 1 byte (R=Roll started S=Stable) or 2 bytes (FS, TS, MS for fake, tilt and move stable).

I’ve simply declared that all commands are BYTES and HEX and the resulting events are just BYTES and HEX. It’ll be up to a more custom specialization to convert these into useful events and values. I also named the transmit to “DiceCommand” and the Receive to “DiceEvent”.

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\ChacteristicsData folder (note the spelling).

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 tot 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.

In Bluetooth Protocols project, in the Bluetooth Protocol directory, make a new class file. Right-click the Bluetooth Protocols directory and Add / Class with a class name that matches the ClassName that’s in the app (e.g., Particula\_GoDice.cs)

In the app, click Copy Language and pick “C# Protocol”. This will copy code into the clipboard. Go back to Visual Studio, and replace the entire class file you made in the previous step with the contents of the clipboard (^A to select all and ^V to paste)

Do the same thing for the specialty XAML and CS files. In the BluetoothDeviceController project in the SpecialtyPages project, right-click and Add New Item and from the template pick User Control. Use the same pattern for the name, but add “Page” (e.g., Particula\_GoDicePage.xaml). This makes both the XAML and C# files. Then replace the contents of those files with the generated code from the app via the Copy Language and Specialty Page XAML and Specialty Page C#.

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.
* Add a “Description”:”…” for the dice 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.

The final results are below. The Transmit 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.

## 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.

Graphical user interface, application

Description automatically generated