Verified Telemetry integration with Azure IoT Solutions

# Overview

IoT devices today are deployed from agricultural farms, factory floors, to smart buildings to sense and monitor the environment. Data quality plays a vital role in the increasing adoption of IoT devices, where organizations rely on the dependability of the data for all the decision support and management functions.

Dependability of an IoT device is directly linked to the trustworthiness and reliability of the data received from the IoT device.

*Reliability* of the data depends on whether the device collecting the data is non-faulty, calibrated properly and whether the reported data is acceptable.

This document provides an overview of integrating Verified Telemetry with Azure RTOS. IoT developers can now leverage reliability features supported by Verified Telemetry seamlessly with sophisticated Azure IoT services. The outline of the document is as follow:

## [**Working of Verified Telemetry**](#_Working_of_Dependable)

## [**Getting started with Azure RTOS**](#_Getting_Started_with)

## [**Preparing the device**](#_Preparing_the_device)

## [**Demo**](#_Demo)

## [**Integrating with IoT plug and Play**](#_Integrating_DIoT_with)

## [**Device certification with Dependability**](#_Device_certification_with)

**7.** [**Appendix usage with Legacy MQTT**](#_Appendix-_Usage_with)

# Working of Verified Telemetry

Verified Telemetry is a state-of-the-art solution to determine the working of the sensor, i.e., working or faulty, consequently used to determine the quality of the sensed data. *This is achieved by devising an intelligent fingerprint for a sensor to determine the status of the sensor.* The sensor fingerprint captures electrical properties of the sensor such as voltage and current using seamless software code running on the IoT device. Thus for every sensed data point we also collect and verify the sensor fingerprint to determine its quality.

The working of Verified Telemetry is as follows (shown in Figure 1):

1. A working fingerprint of the sensor is collected and stored in the device.
2. Every time the data is collected in the field, sensor’s fingerprint is also collected. The frequency of fingerprint collection depends on the application requirement.
3. The collected fingerprint is compared with the stored fingerprint to determine the status of the sensor (working or faulty).

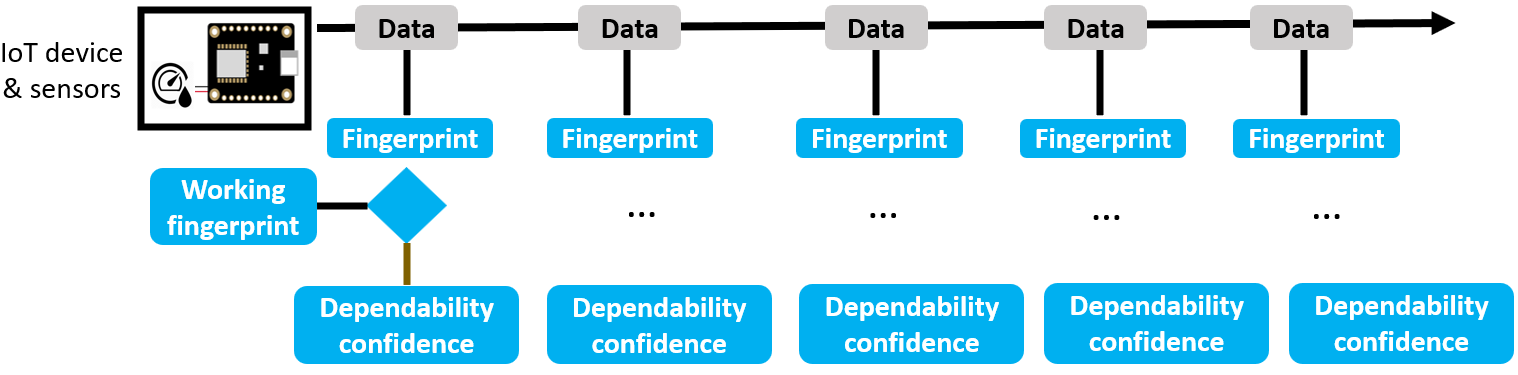


Figure 1: Working of Verified Telemetry

In order to support the Verified Telemetry features, below are the minimal set of requirements:

1. Ability to add/modify device code: This is required to add the library and associated code to support Verified Telemetry.
2. Ability to turn On/Off the sensors attached to the device: This is required to collect sensor fingerprint and the turn off state can be just a few milliseconds.

# Getting Started with Azure RTOS and Azure IoT Middleware

We will now walk you through the essential changes to enable dependability with your Azure RTOS device.

**Pre-requisite:**

We ask the IoT developer to refer to the [getting started with RTOS tutorials](https://go.microsoft.com/fwlink/p/?linkid=2129824) that shows how to connect an IoT device to Azure IoT using Azure RTOS. In this document we use MXChip AZ3166 IoT devkit and ask the developer to build their device by following the [getting started](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166) tutorial and validate that they are successfully able to view device telemetry, properties on Azure IoT Hub.

**Steps to include dependability features into your existing Azure RTOS device code:**

We will now enumerate the steps involved with Azure IoT Middleware:

Step 1: Include Verified Telemetry library to your device code

Step 2: ***Initialize PnP components and Dependability DB***

Step 3: Include implementations for properties and commands

## Step 1: Include Verified Telemetry library to your device code.

* Download the library from the <link>
* Extract and place it in “[*getting-started*](https://github.com/azure-rtos/getting-started)*/[MXChip](https://github.com/azure-rtos/getting-started/tree/master/MXChip)/*[*AZ3166*](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166)*/lib/* “ folder.
* Add new subdirectory in “[*getting-started*](https://github.com/azure-rtos/getting-started)*/[MXChip](https://github.com/azure-rtos/getting-started/tree/master/MXChip)/*[*AZ3166*](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166)*/lib/*CMakeLists.txt”.
* Link new library in “[*getting-started*](https://github.com/azure-rtos/getting-started)*/[MXChip](https://github.com/azure-rtos/getting-started/tree/master/MXChip)/*[*AZ3166*](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166)*/app/*CMakeLists.txt”.
* Make sure that the ***VT\_DEVICE*** option in “[*getting-started*](https://github.com/azure-rtos/getting-started)*/*[*MXChip*](https://github.com/azure-rtos/getting-started/tree/master/MXChip)*/*[*AZ3166*](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166)*/lib/verified\_telemetry/*CMakeLists.txt” is configured correctly.

|  |  |  |
| --- | --- | --- |
| **Option** | **Description** | **Available Values** |
| ***VT\_DEVICE*** | Links the HAL implementation to VT library. | ***MXCHIP\_AZ3166***  ***STM\_BL475EIOT01A***  ***CUSTOM\_DEVICE*** |

In this case, ***VT\_DEVICE*** should be configured to ***MXCHIP\_AZ3166.***

***Note****: If****CUSTOM\_DEVICE*** *option is selected, user must provide implementations for the functions present [vt\_dsc\_custom.c](https://dev.azure.com/DependableIoT/AzureRTOS-DIoT/_git/AzureRTOS-DIoT?path=%2FMXChip%2FAZ3166%2Flib%2Fverified_telemetry%2Fsrc%2Fplatform%2Fvt_dsc_custom.c&version=GBSTM32_AzureRTOS) file.*

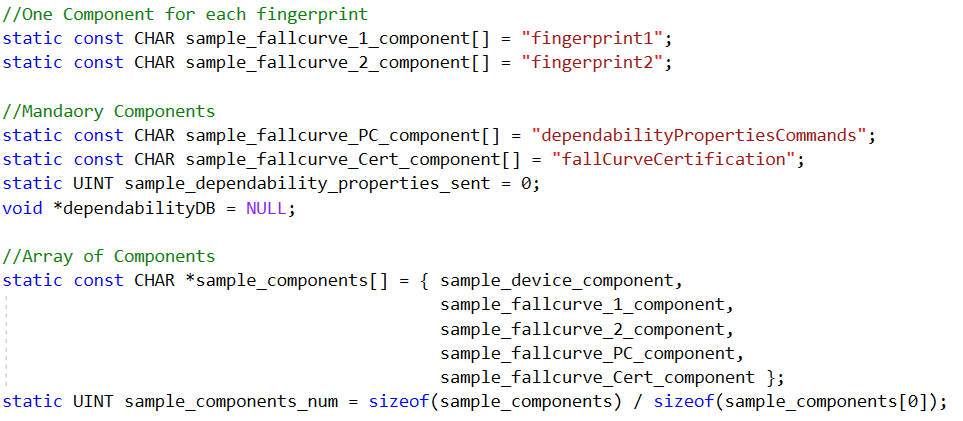
## Step 2: Initialize PnP components and Dependability DB

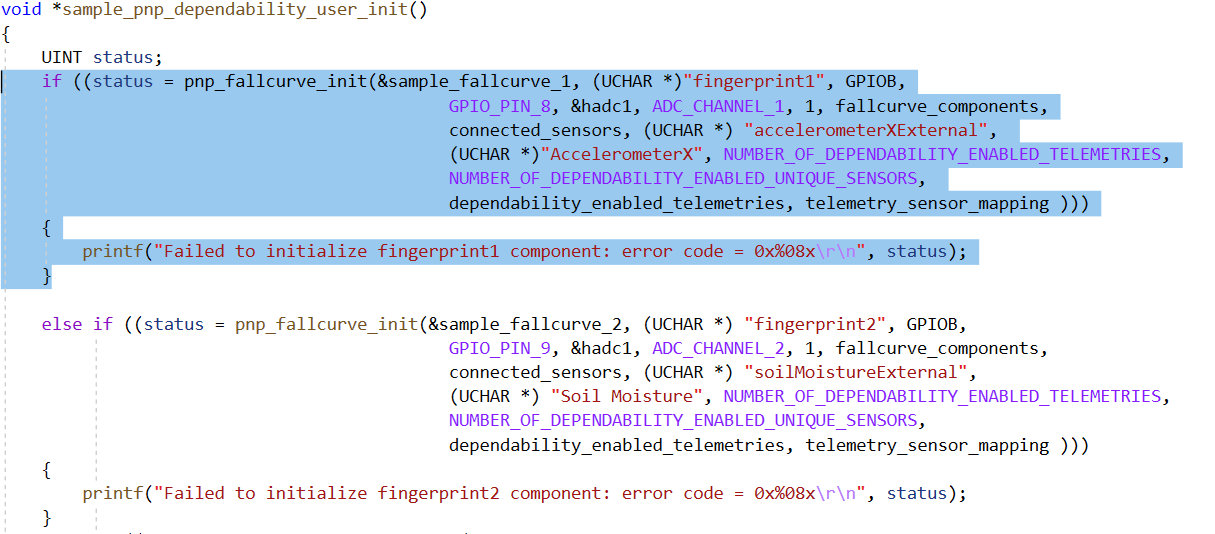
* **Declare and initialize variables for including PnP components in Dependability layer**

Dependability consists of two types of components:

* + Fingerprint Component: One for each Telemetry.
  + Mandatory Components: These are dependability Properties, Commands, and certification components.

We define and add them to an existing array of components as shown below:



The components are then initialized by using ***pnp\_fallcurve\_init*** functions, as shown below:

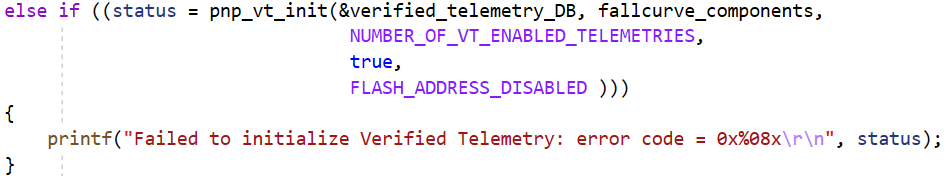
1. Component Pointer: Pointer of type ***PNP\_FALLCURVE\_COMPONENT***
2. Fingerprint Name: Same as the fingerprint names added to the existing component array
3. GPIO (Port and Pin): Connected to Sensor power supply
4. ADC (Port and Pin): Connected to Sensor analog output
5. Timer: Provide initialized timer for better performance, if not available pass NULL.
6. Sensor Connected: Sensor Identifier
7. Number of Dependability enabled unique sensors: Total Number of unique sensors
8. Dependability enabled telemetries (Array Pointer and length): Pointer to an array allocated in global memory.
9. Telemetry Sensor Mapping: Pointer to an array allocated in global memory

* **Declare and initialize verified\_telemetry\_DB**

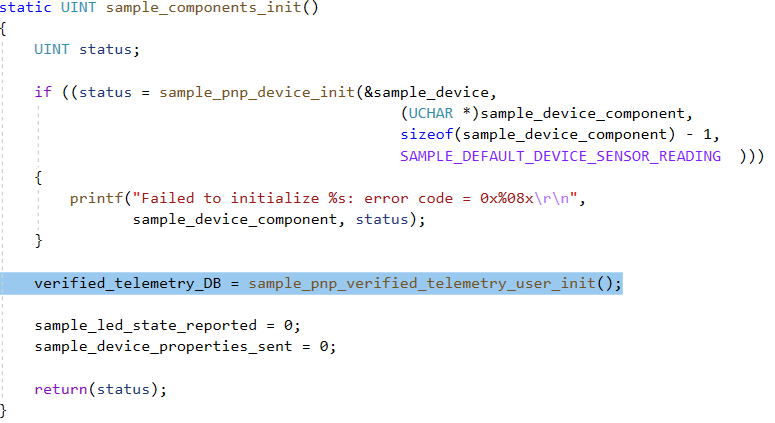
Verified Telemetry library maintains a database of pointers to all the pnp components, so that they don’t have to be referenced individually for any implementation. Pointer to this database is used for adding all the required functionalities to the application code.

The database is represented by ***VERIFIED\_TELEMETRY\_DB*** structure, we declare it such that it is available to all the running threads.

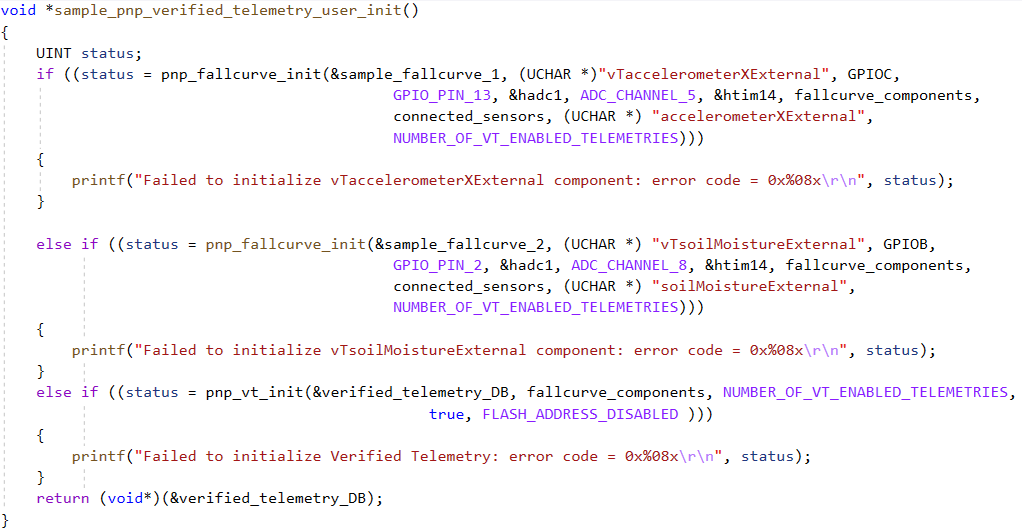
This structure is then initialized using ***pnp\_dependability\_init*** function as shown below:



Application Developer can either call the required initialization functions individually or call a wrapper function ***sample\_pnp\_verified\_telemetry\_user\_init***, along with other system initializers, as shown in the image below:



***sample\_pnp\_verified\_telemetry\_user\_init*** internally calls all the initializers required to enable the dependability layer and return the initialized ***verified\_telemetry\_DB*** pointer.

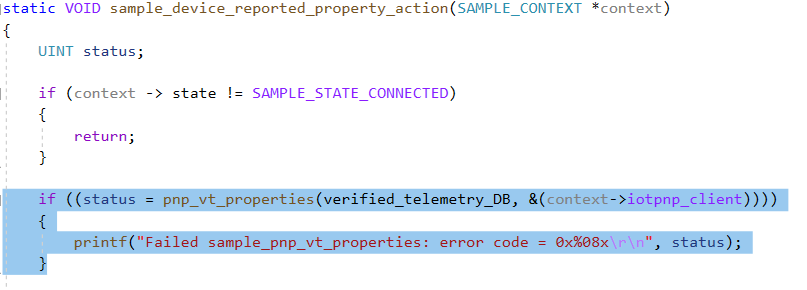


## Step 3: Include implementations for properties and commands.

**Add implementation for Verified Telemetry properties:**

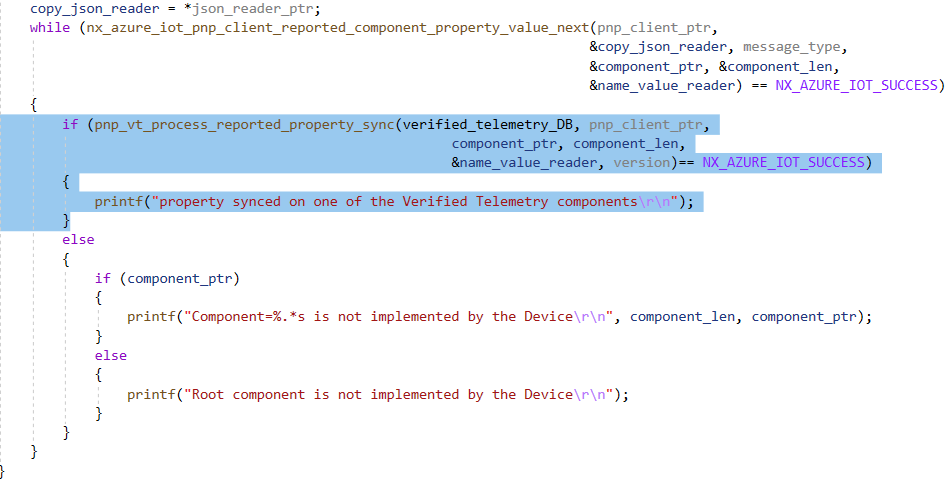
Modify the callback functions of Reported and desired properties for adding Verified Telemetry properties.

* **Reported Properties**

****

* **Desired Properties**

Add ***pnp\_vt\_process\_property\_update*** to processes all supported desired property updates and ***pnp\_vt\_process\_reported\_property\_sync*** to syncronizes Verified Telemetry Settings stored in digital Twin as reported properties on startup.

****

**Add implementation for dependability commands:**

Modify the command action callback function for implementing dependability layer commands.



# Preparing the device

We now describe the hardware and software initializations to enable Verified Telemetry.

## Hardware details:

* Connect the intended analog sensor (ADXL335 for this demo) to the MXChip devkit on the following pins

|  |  |  |
| --- | --- | --- |
| Sensor  (ADXL335) | MxChip AZ3166 Devkit | Kitronik / Sparkfun Breakout |
| VCC | PC\_13 | 8 |
| X-Out | PA\_5 | 4 |
| GND | GND | GND |

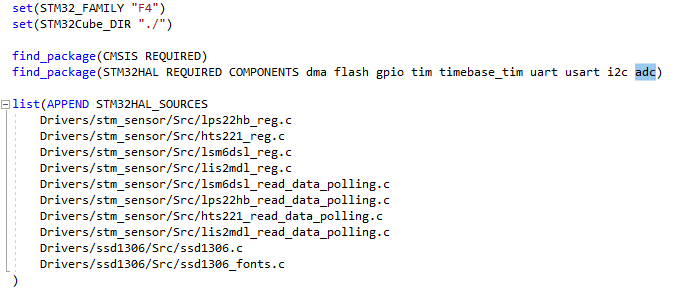
These connections are in line with the arguments used for port registration in Step 2 of Getting started section.

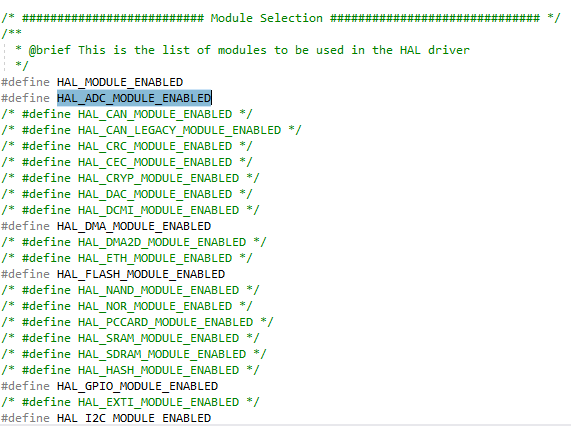
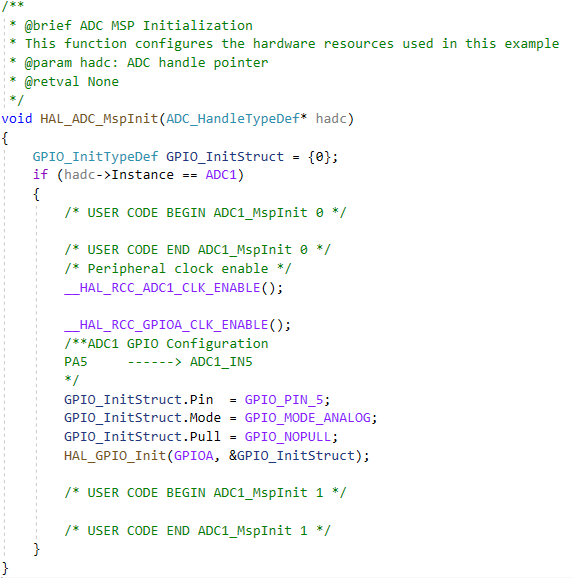
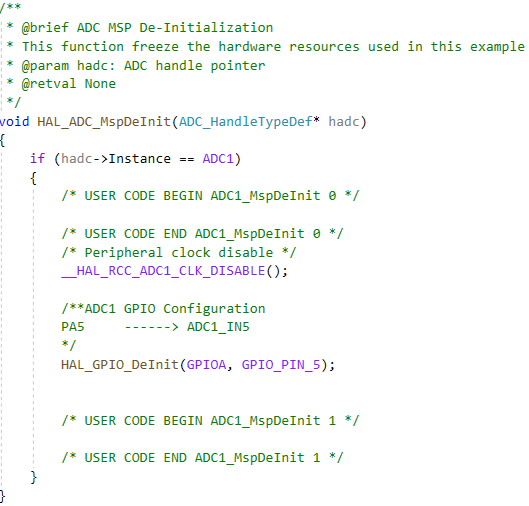
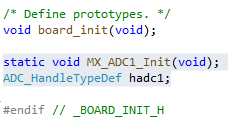
## Software details:

The pins used for connecting the sensor must be properly initialized for its functioning.

In Getting Started Sample**, *PC\_13*** is already initialized as GPIO Pin, but ***PA\_5*** is uninitialized. We will now describe the steps to Enable ADC module and initialize ADC controller connected to ***PA\_5*** to read sensor data.

* Enable ADC module: Add ***adc*** module in ***getting-started/MXChip/AZ3166/lib/stm32cubef4/* *CMakeLists.txt***

****

* Add/Uncomment **HAL\_ADC\_MODULE\_ENABLED** header definition in ***getting-started/ MXChip/ AZ3166/ lib/stm32cubef4/config/stm32f4xx\_hal\_con.h ***
* Add ***HAL\_ADC\_MspInit*** and ***HAL\_ADC\_MspDeInit*** functions in ***stm32f4xx\_hal\_msp.c***
* Define prototypes for ***MX\_ADC1\_Init*** and ADC Controller in [***getting-started***](https://github.com/azure-rtos/getting-started)***/[MXChip](https://github.com/azure-rtos/getting-started/tree/master/MXChip)/***[***AZ3166***](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166)***/***[***app***](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166/app)***/board\_init.h,*** as shown below

Note: This declared controller (***hadc1***), is later passed as argument to ***dx\_port\_register*** function.

* Define ***MX\_ADC1\_Init*** function in [***getting-started***](https://github.com/azure-rtos/getting-started)***/[MXChip](https://github.com/azure-rtos/getting-started/tree/master/MXChip)/***[***AZ3166***](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166)***/***[***app***](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166/app)***/board\_init.c*** , and call it in ***board\_init*** function. 

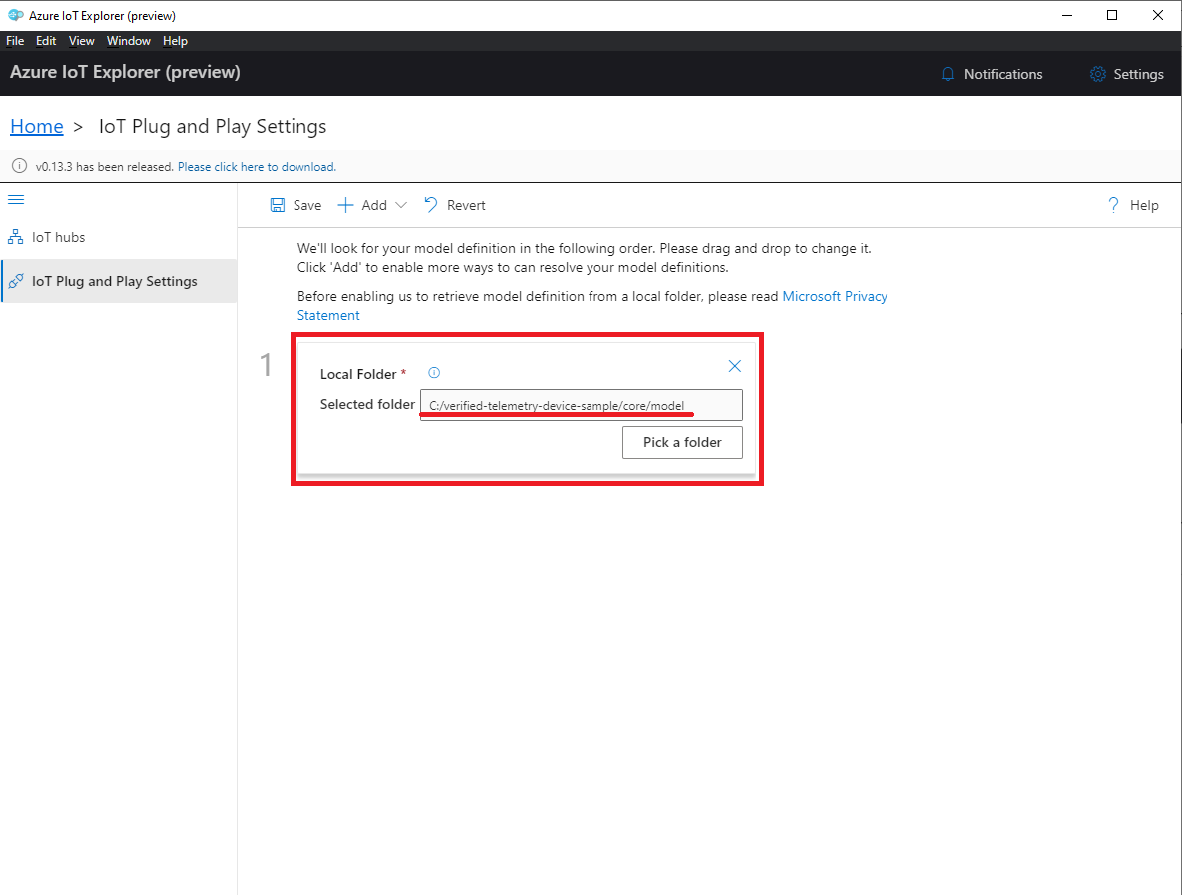
Compile and flash the edited code to the device. Follow the steps in [Getting Started](https://github.com/azure-rtos/getting-started/tree/master/MXChip/AZ3166) tutorial to view device output on serial monitor and Azure IoT explorer.

# Demo

We will now show the interactions between IoT hub and the device to showcase dependability features.

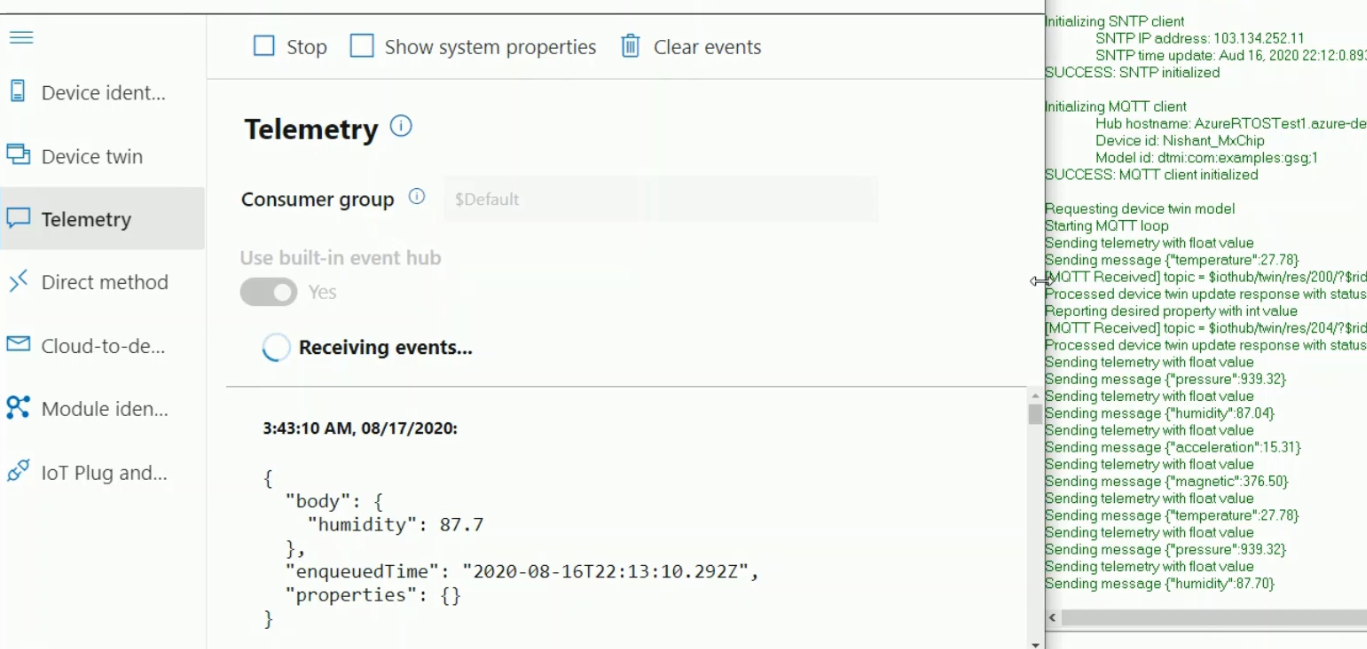
* *Add PnP Model Files to IoT Explorer:*

Navigate to IoT Plug and Play Settings and add model files from **“*getting-started* /core/model/”**



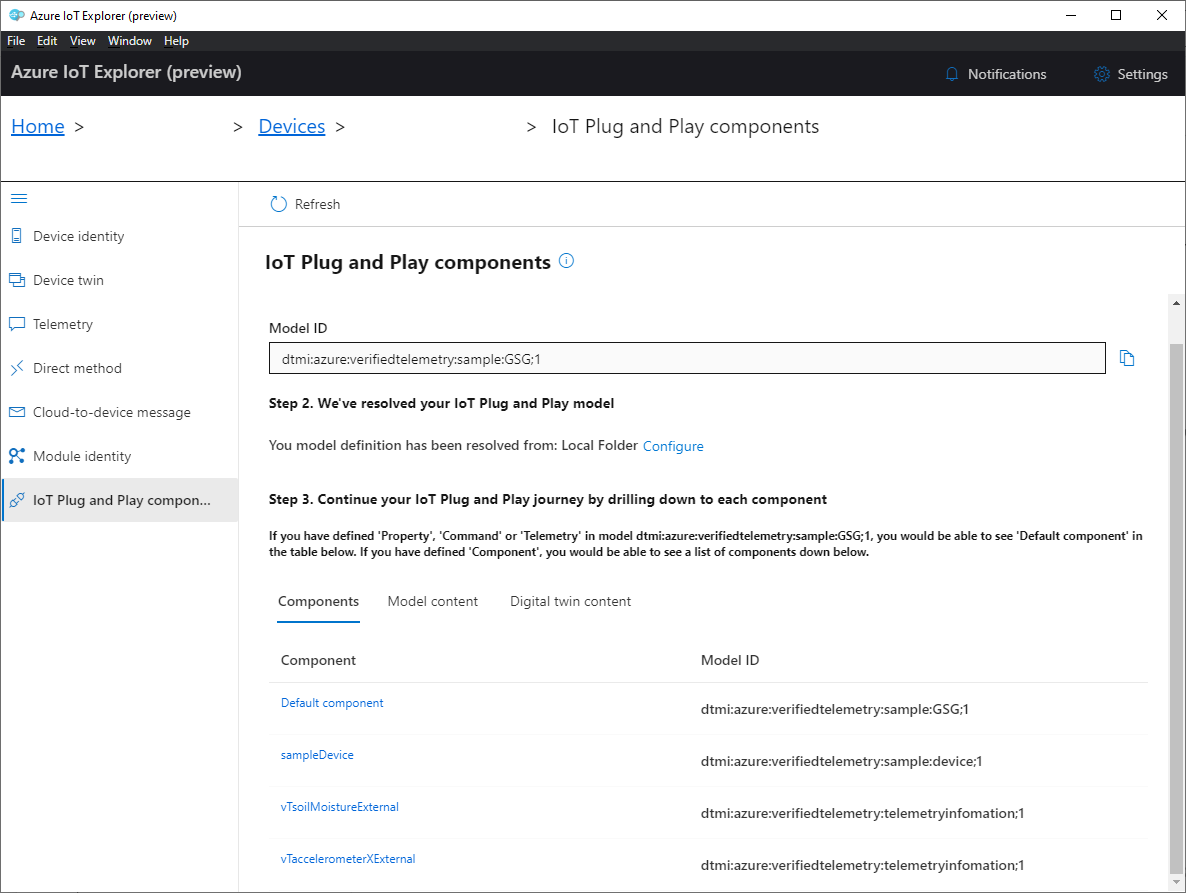
* Default mode:

By default, dependability is disabled and hence we just see the sensor telemetry sent to the Hub



* Access Dependability IoT Plug and Play components:

We will use the Dependability IoT Plug and Play components to interact with the device. To view the dependability components, navigate to the connected ***device*** and then ***IoT Plug and Play components***. Here, you will see a list of components extracted from the model files that we had uploaded earlier.

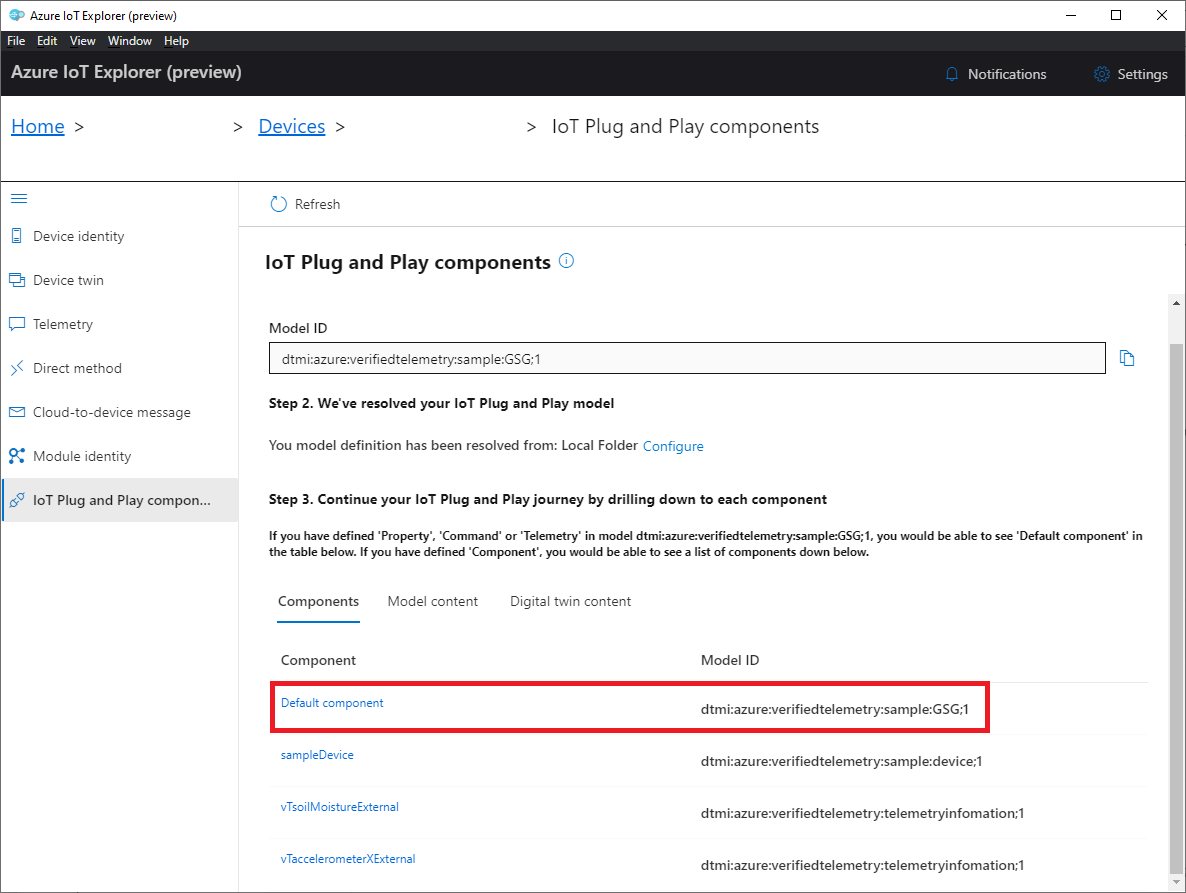


* Enable Verified Telemetry:

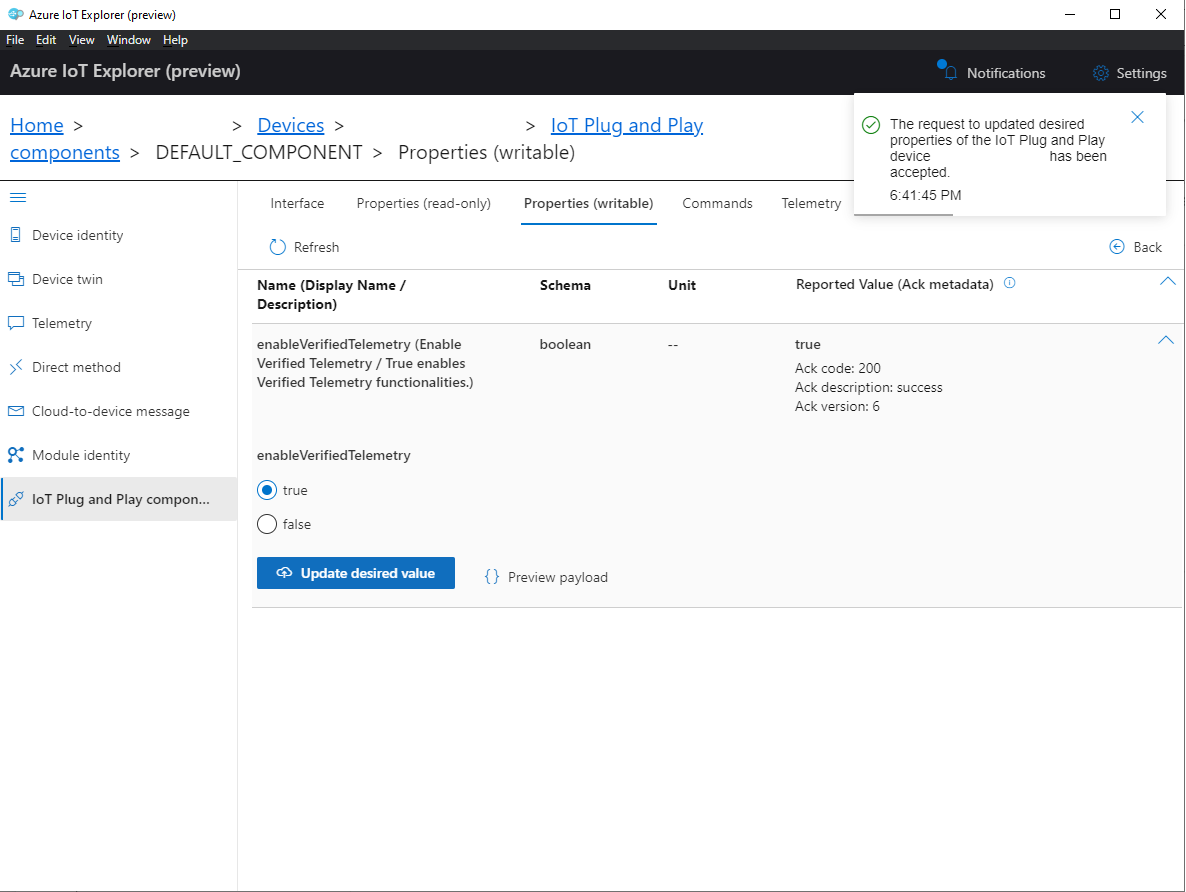
To enable verified telemetry, we set the writable property,

enableVerifiedTelemetry = TRUE

To do this, select ***Default\_Components*** component in IoT Plug and Play components tab.

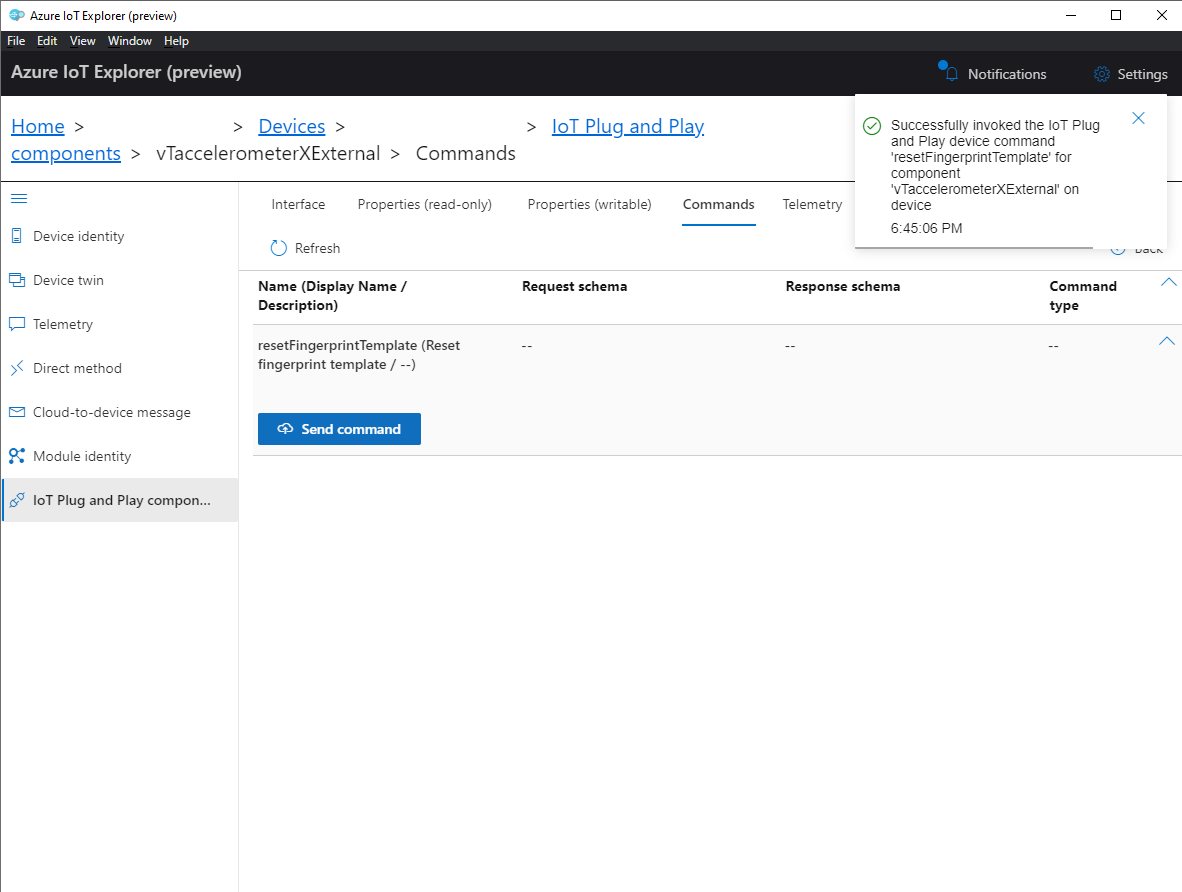


Navigate to ***Properties (Writable)*** tab, select ***true*** and click on ***Update desired value.***



* Update Fingerprint Template:

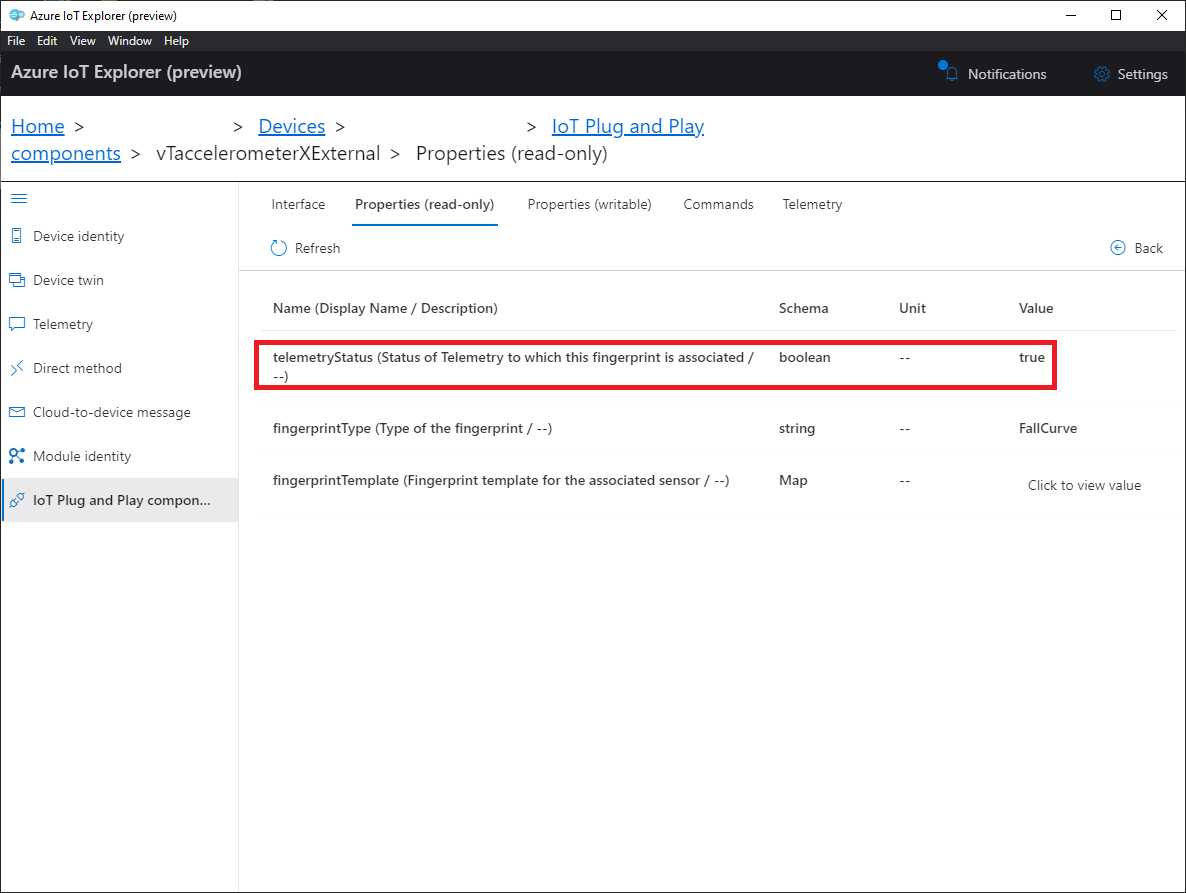
We will update the existing fingerprint template with the latest fingerprint computed for Accelerometer sensor, this template will be further used for sensor status verification. To do so, Navigate to ***Commands*** tab in ***vTaccelerometerXExternal*** component, and send a ***resetFingerprintTemplate*** command.



* Sensor Status:

The device is now capable of evaluating the sensor status and determine whether a sensor is working or faulty, it will do this by comparing the stored fingerprint template with the live fingerprint to determine the status of the sensor.

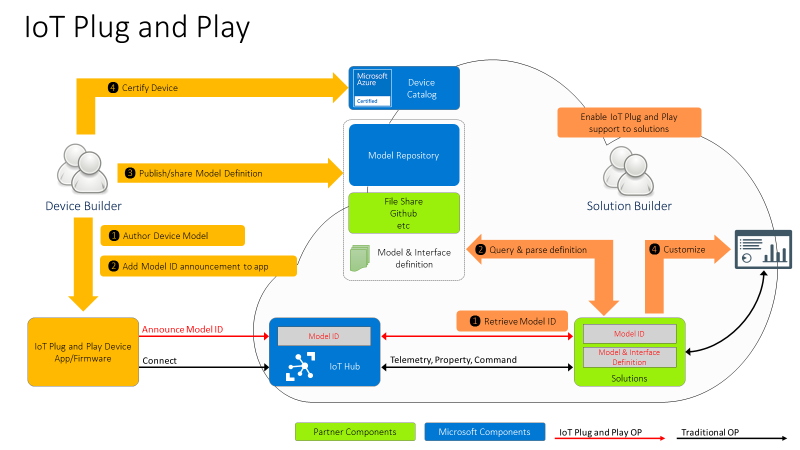
The property ***telemetryStatus*** in the ***vTaccelerometerXExternal*** component indicates whether the telemetry ***accelerometerXExternal*** is correct or not.



# Integrating Verified Telemetry with Azure IoT Plug and Play

IoT Plug and Play (PnP) enables solution developers to integrate their IoT devices seamlessly with Azure IoT services. At the core of PnP, is a device models that a device uses to advertise its capabilities to an IoT PnP enabled application such as IoT hub and central.

Device model is described in terms of telemetry, properties, and commands using the Digital Twins Definition Language (DTDL) version 2. Figure below showcases the development activities for device and solution developer.



Steps performed by device developer to enable PnP:

1. Author a device model using DTDLv2
2. Use the Model ID in the connection string
3. Share the Model ID with solution developers through private/public repo

|  |
| --- |
| VT Properties:  enableVerifiedTelemetry  VT Commands:  resetFingerprintTemplate |

In order to integrate DIoT with IoT PnP, all you need to do is to include the dependable IoT interface into your device model.

Adding Verified Telemetry Interface files to PnP device model

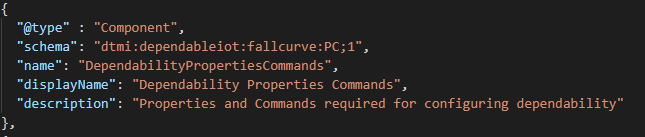
The table shows the default telemetry, properties and commands supported by the Verified Telemetry library. One can add the above details using a single component or multi-component implementations.

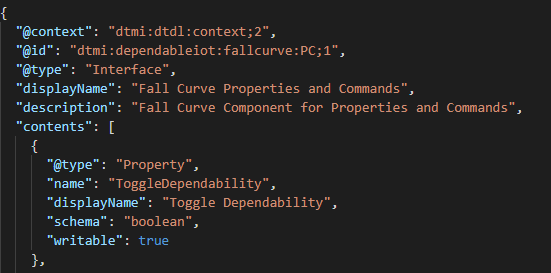
Multiple component interfaces allow us to define Verified Telemetry templates for properties/commands enabling reuse across devices. We now describe, how easily Verified Telemetry multi-components can be defined.

Today, we define two components for DIoT that are easy to use and can be reused across devices.

1. Dependability Properties and Commands:

This component includes all the supported properties and commands required for Verified Telemetry.





In order to support reuse, we plan to publish the two DIoT components in the model repository, which can be easily copied in the device model appropriately.

DIoT Telemetry Component: "dtmi:dependableiot:fallcurve:telemetry;1"

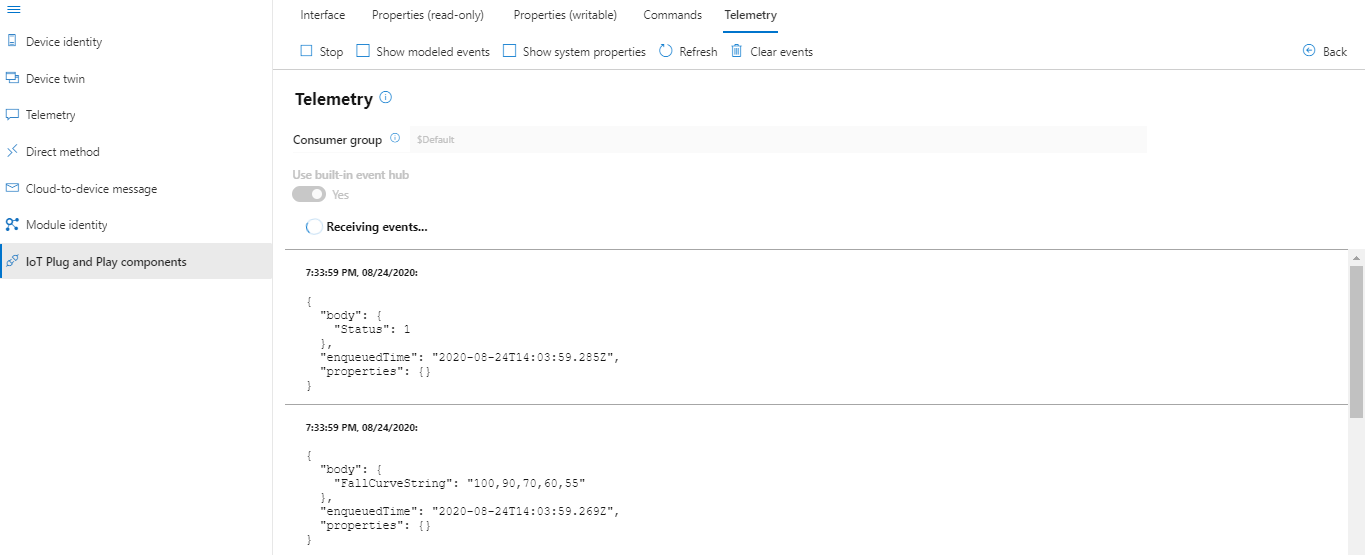
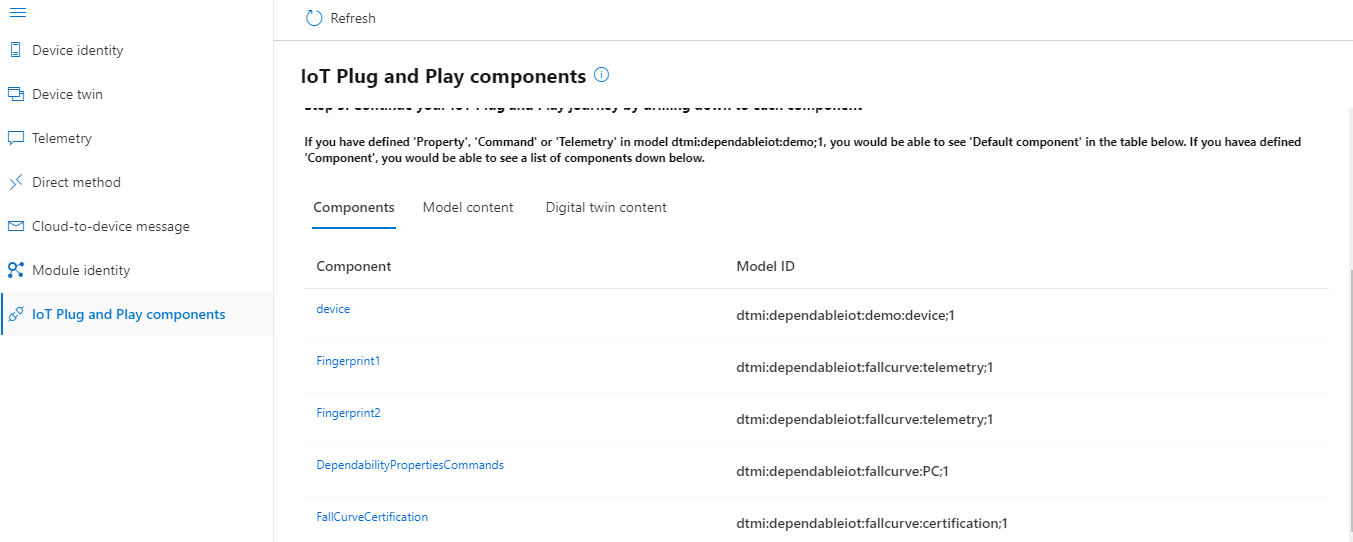
DIoT Properties/commands Component: "dtmi:dependableiot:fallcurve:PC;1"

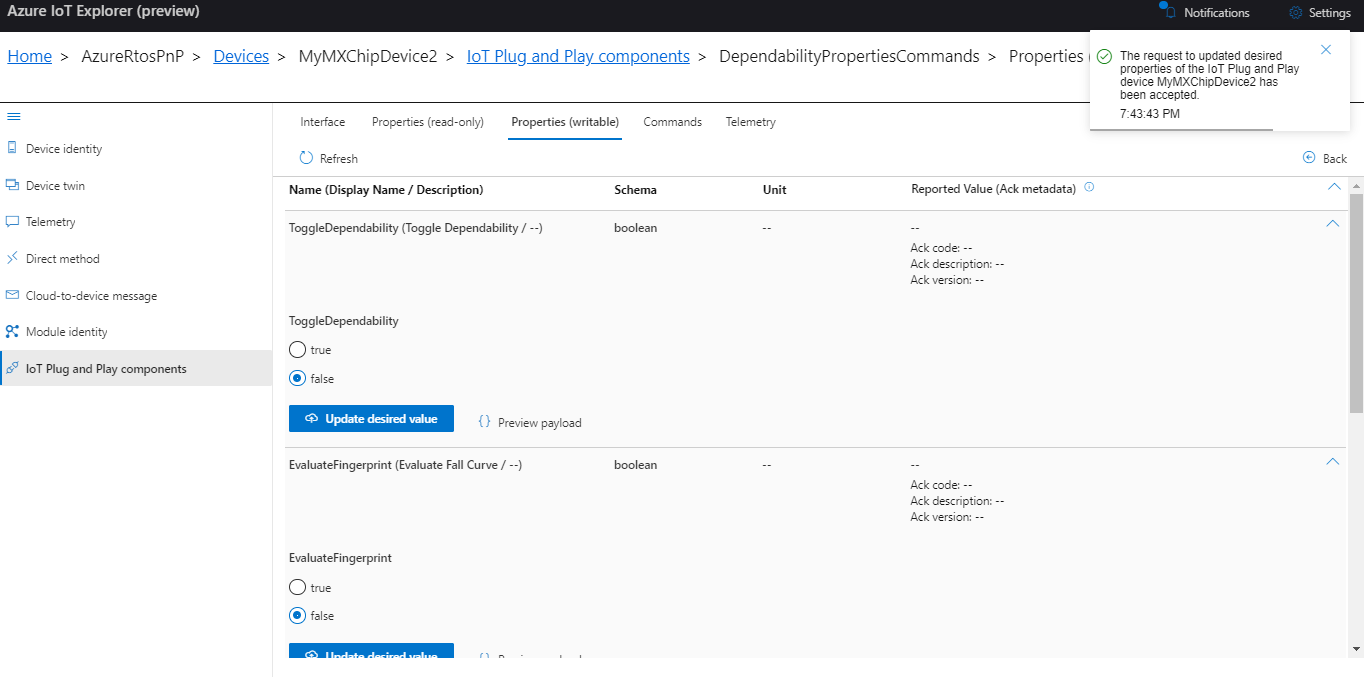
A sample Device Model with DIoT components



Azure IoT explorer to validate device implementation

We now validate the device implementation against the Device Model defined previously. We first load the device models with all the components to Azure IoT explorer, which will populate the IoT plug and play view with a default UI built from the model.



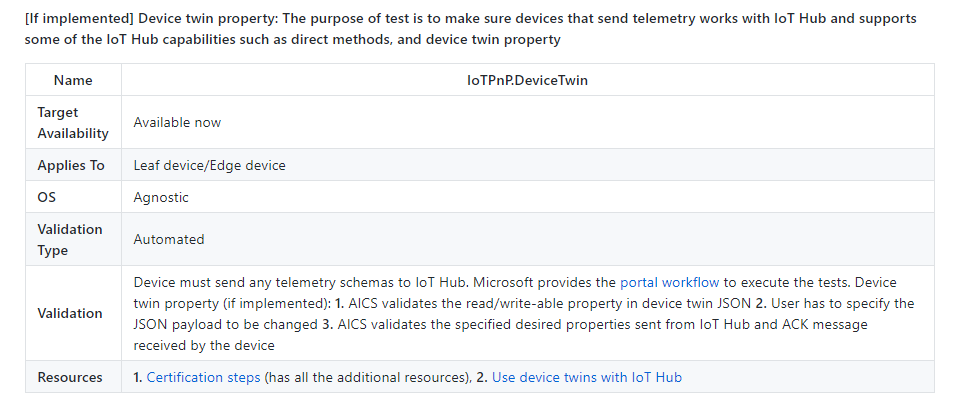


# Device certification with Dependability

Device builders can certify their devices for solution builders by going through Azure IoT certification service (AICS). This allows solution builders easy integration with Azure IoT services such as IoT Hub and Central. Current IoT Plug and play certification validates:

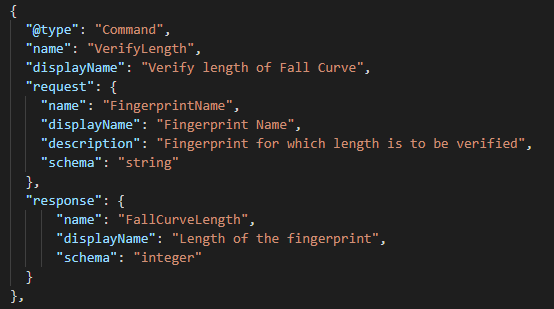
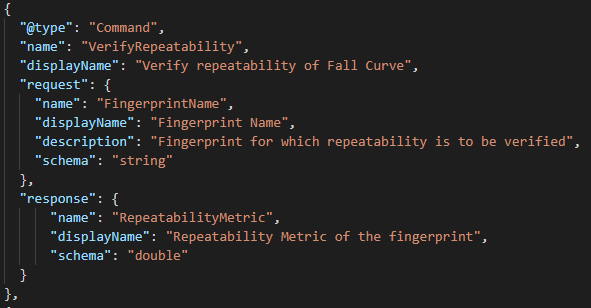
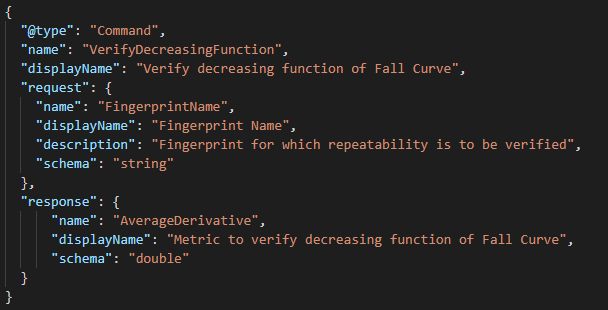
1. Model compliance with DTDLv2
2. Secure provisioning

Here is an example of Device Twin property tests:



Adding additional validation tests for Dependable IoT:

We now describe the additional tests to validate proper implementation of DIoT components and also verify the fingerprint collected. These tests can be implemented at the device or by AICS.

1. **DIoT component check:** AICS validates if the DIoT components are present in the device model, which triggers additional validation checks
2. **VerifyLength:** Verifies the length of the fingerprint collected by the device
   1. Invokes collect fingerprint command and analyses the response received
3. **VerifyRepeatability:** Verifies the consistency of the fingerprints received from the device
   1. Triggers a command to collect 5 to 10 fingerprints and analyzes the variance between the fingerprints. If variance < X, then the fingerprint collected is validated
4. **VerifyDecreasingFunction:** Verifies if the fingerprint collected follows a decreasing function
   1. Invokes collect fingerprint and determines if it adheres to decreasing function.

If the device passes these additional tests, it will now be certified as **Dependable device**. The solution developers can now leverage the Sensor Status information to rely on only high quality data for modeling and taking decisions.