UM2688

User manual

Getting started with the STM32Cube High Speed Datalog function pack for STWIN evaluation kits

Introduction

The FP-SNS-DATALOG1 function pack implements High Speed Datalog application for STEVAL-STWINKT1 and STEVAL-STWINKT1B. It provides a comprehensive solution to save data from any combination of sensors and microphones configured up to the maximum sampling rate.

The application also allows configuring ISM330DHCX Machine Learning Core unit and reading its output.

Sensor data can be stored onto a micro SD card (Secure Digital High Capacity - SDHC) formatted with the FAT32 file system, or streamed to a PC via USB (WinUSB class) using the companion host software (cli_example) provided for Windows and Linux.

The FP-SNS-DATALOG1 allows configuring the board via JSON file as well as starting and controlling data acquisition. Commands can be sent from a host via command line interface.

The application can be controlled via Bluetooth using the STBLESensor app (available for Android and under development for iOS) which lets you manage the board and sensor configurations, start/stop data acquisition on SD card, control data labelling and display the output of the Machine Learning Core.

To read sensor data acquired using FP-SNS-DATALOG1, easy-to-use scripts in Python and Matlab are provided within the software package. The scripts have been successfully tested with MATLAB v2019a and Python 3.7.

RELATED LINKS

Visit the STM32Cube ecosystem web page on www.st.com for further information

1 FP-SNS-DATALOG1 software expansion for STM32Cube

1.1 Overview

FP-SNS-DATALOG1 is an STM32 ODE function pack and expands STM32Cube functionality.

The software package provides a comprehensive solution to save data from any combination of sensors and microphones configured up to the maximum sampling rate.

The key package features are:

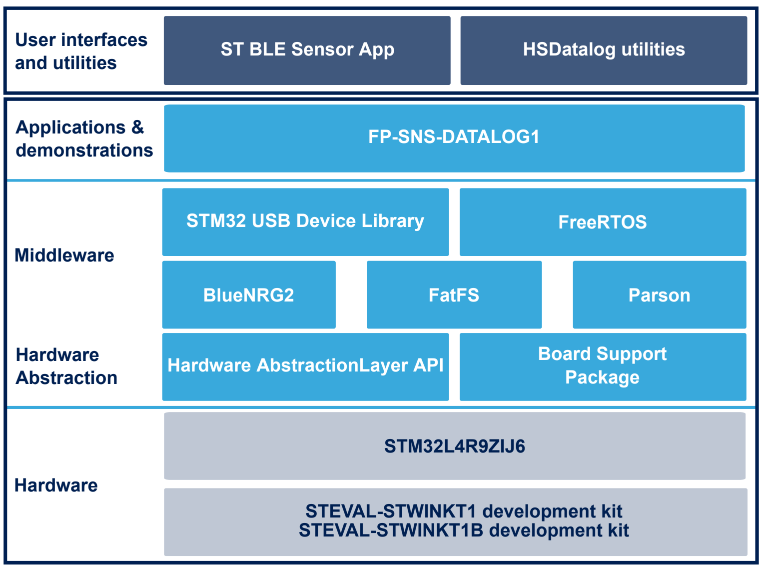
- High-rate (up to 6 Mbit/s) data capture software suite:
  - BLE app for system setup and real-time control
  - Python and C++ real-time control applications
  - Dedicated Python SDK for sensor data analysis
  - Host developer's API enables integration into any data science design flow
  - Compatible with Unico-GUI which enables configuration of ISM330DHCX Machine Learning Core unit
  - Timestamping for sensor data synchronization
- Embedded software, middleware and drivers:
  - FatFS third-party FAT file system module for small embedded systems
  - FreeRTOS third-party RTOS kernel for embedded devices
  - STWIN low-level BSP drivers
- Based on STM32Cube software development environment for STM32 microcontrollers

1.2 Architecture

The application software accesses the STWIN evaluation kits through the following software layers:

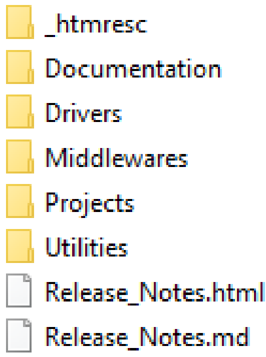
- the **STM32Cube HAL layer**, which provides a simple, generic, multi-instance set of application programming interfaces (APIs) to interact with the upper application, library and stack layers. It has generic and extension APIs and is directly built around a generic architecture, allowing successive layers, like the middleware layer, to implement functions without requiring specific hardware configurations for a given microcontroller unit (MCU). This structure improves library code reusability and guarantees an easy portability on other devices
- the **board support package** (BSP) layer, which supports all the peripherals on the STM32 Nucleo except the MCU. This limited set of APIs provides a programming interface for certain board-specific peripherals like the LED, the user button, etc. This interface also helps in identifying the specific board version.

Figure 1. FP-SNS-DATALOG1 software architecture



1.3 Folder structure

Figure 2. **FP-SNS-DATALOG1 package folder structure**



The following folders are included in the software package:

- **Documentation**: contains a compiled HTML file generated from the source code detailing the software components and APIs (one for each project).
- **Drivers**: contains the HAL drivers and the board-specific drivers for each supported board or hardware platform, including those for the on-board components, and the CMSIS vendor-independent hardware abstraction layer for the ARM Cortex-M processor series.
- **Middlewares**: libraries and protocols featuring BlueNRG-2, STM32 USB Device Library, FreeRTOS, FatFs, parson.
- **Projects**: contains a sample application implementing the High Speed Datalog. This application is provided for the STEVAL-STWINKT1 and STEVAL-STWINKT01B platforms with three development environments: IAR Embedded Workbench for ARM, RealView Microcontroller Development Kit (MDK-ARM-STR) and STM32CubeIDE.
- **Utilities**: contains some complementary project files (i.e., Python and Matlab scripts, cli_example, UCF and JSON configuration examples).

1.4 APIs

Detailed technical information with full user API function and parameter description are in a compiled HTML file in the “Documentation” folder.

2 Getting started

As HSDatalog application included in the FP-SNS-DATALOG1 function pack is not the default firmware on the STEVAL-STWINKT1 and STEVAL-STWINKT1B, you have to download it on the board, using the pre-compiled binary provided in the Projects/HSDatalog/Binary folder.

To update the firmware, follow the procedure below.

Step 1. Connect the STWIN core system board to the STLINK-V3MINI programmer.

Step 2. Connect both boards to a PC using micro USB cables.

Step 3. Open STM32CubeProgrammer, select the proper binary file and download the firmware.

Step 4. Reset the board once the proper firmware is flashed.

RELATED LINKS

For further details, refer to UM2622, Section 3

2.1 USB mode - command line example

Once you plug the STWIN to a PC via micro-USB cable with the HSDatalog firmware, Windows should recognize the board as a new USB device and automatically install the required drivers.

To verify it, check whether you can see a new device called STWIN Multi-Sensor Streaming in the Device Manager Windows settings.

Figure 3. Device Manager Window

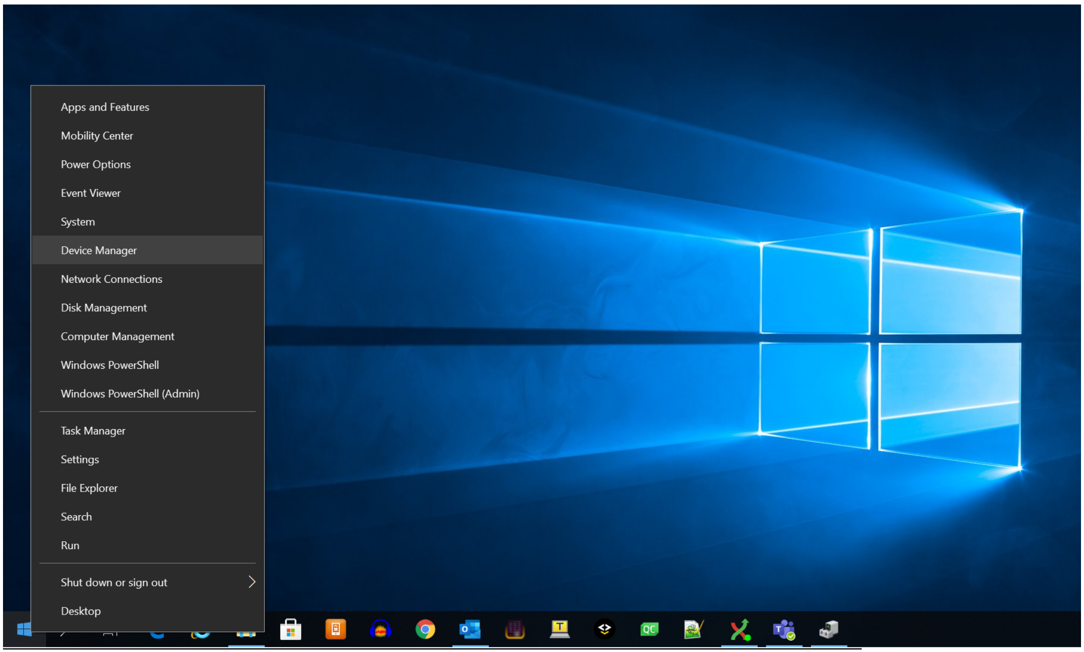
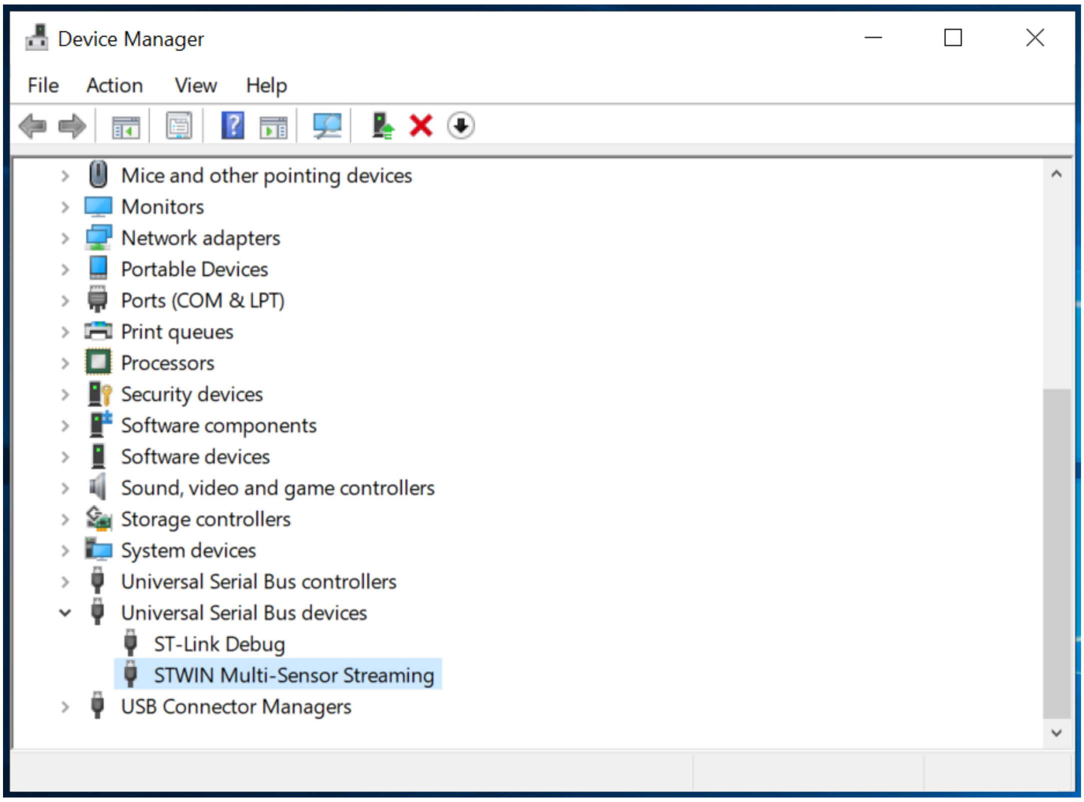


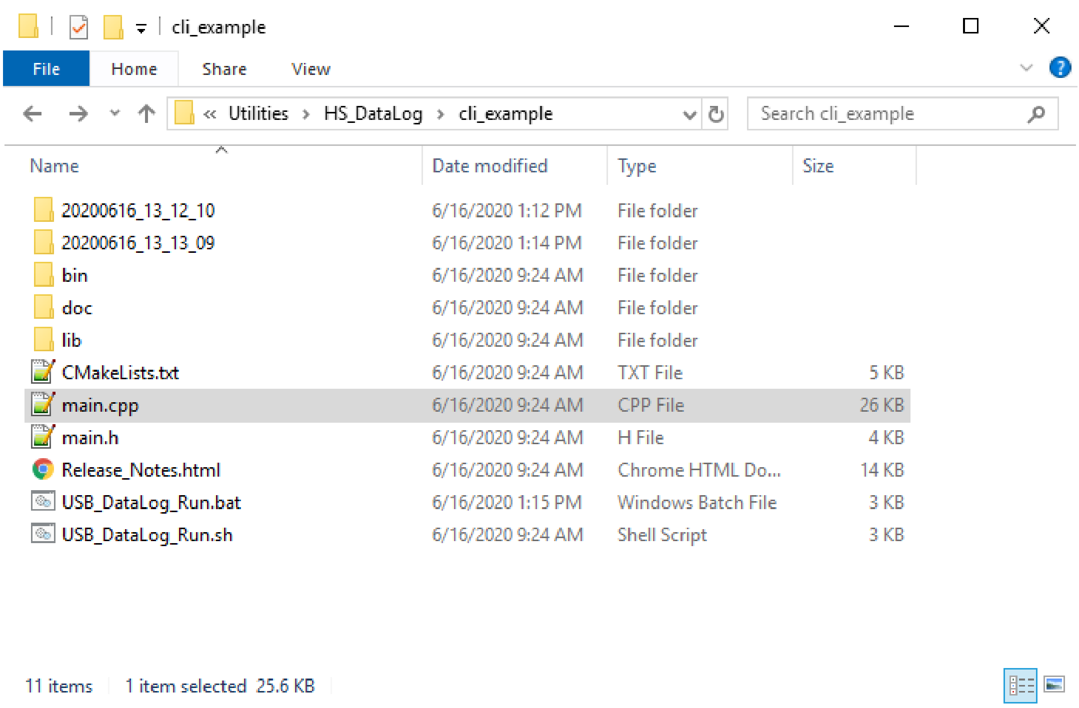
Figure 4. **STWIN Multi-Sensor Streaming**



A command line example is located in the Utilities folder.

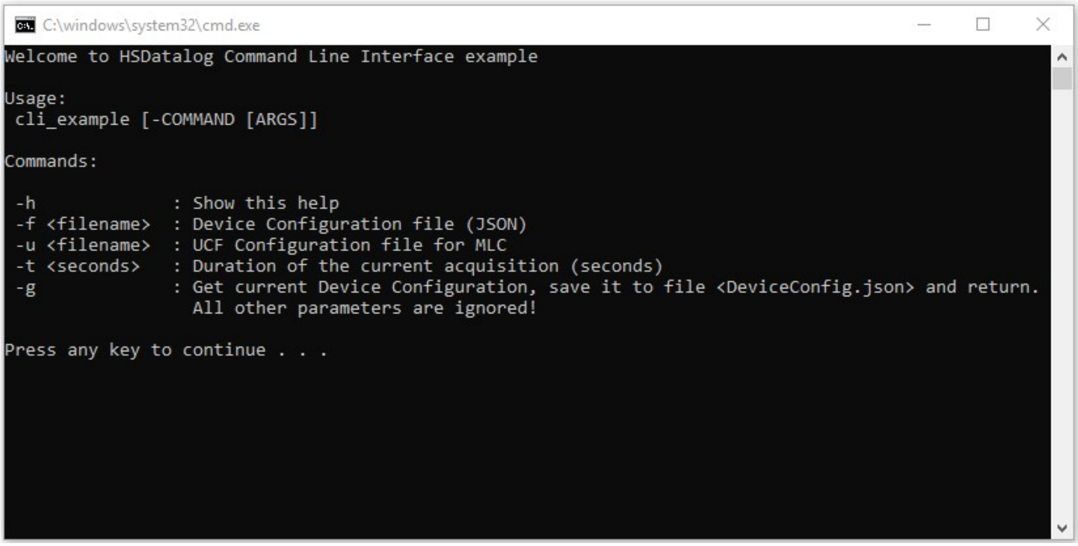
The bin folder contains a pre-compiled version of the program for Linux and Windows. A CMake project is also provided to make recompiling the application easy.

Figure 5. **HSDatalog application - cli_example**



If needed, the application can receive a configuration file for the STWIN in .json format, a configuration file for the ISM330DHCX Machine Learning Core unit in .ucf format and a timeout as parameters.

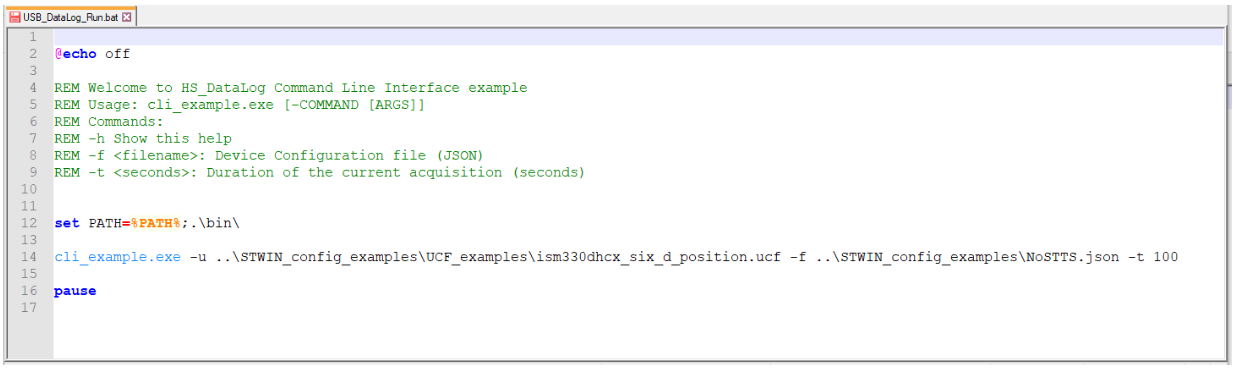
Figure 6. HSDatalog application - help



```
C:\windows\system32\cmd.exe
Welcome to HSDatalog Command Line Interface example

Usage:
 cli_example [-COMMAND [ARGS]]

Commands:

 -h              : Show this help
 -f <filename>   : Device Configuration file (JSON)
 -u <filename>   : UCF Configuration file for MLC
 -t <seconds>    : Duration of the current acquisition (seconds)
 -g              : Get current Device Configuration, save it to file <DeviceConfig.json> and return.
                   All other parameters are ignored!

Press any key to continue . . .
```

`USB_DataLog_Run.bat` for Windows and `USB_DataLog_Run.sh` for Linux scripts provide a ready-to-use example. You are free to customize the scripts to run the desired configurations.

Figure 7. HSDatalog application - Datalog_Run script



```
USB_DataLog_Run.bat

@echo off

REM Welcome to HS_DataLog Command Line Interface example
REM Usage: cli_example.exe [-COMMAND [ARGS]]
REM Commands:
REM -h Show this help
REM -f <filename>: Device Configuration file (JSON)
REM -t <seconds>: Duration of the current acquisition (seconds)


set PATH=%PATH%;.\bin\

cli_example.exe -u ..\STWIN_config_examples\UCF_examples\ism330dhcx_six_d_position.ucf -f ..\STWIN_config_examples\NoSTTS.json -t 100

pause
```

The Utilities/HSDatalog/STWIN_config_examples folder also contains some JSON configuration examples that can be freely modified to save only necessary data and UCF_examples folder which contains UCF configuration files to enable the Machine Learning Core feature available on the ISM330DHCX sensor.

Other UCF examples are freely available on github: https://github.com/STMicroelectronics/STMems_Machine_Learning_Core.

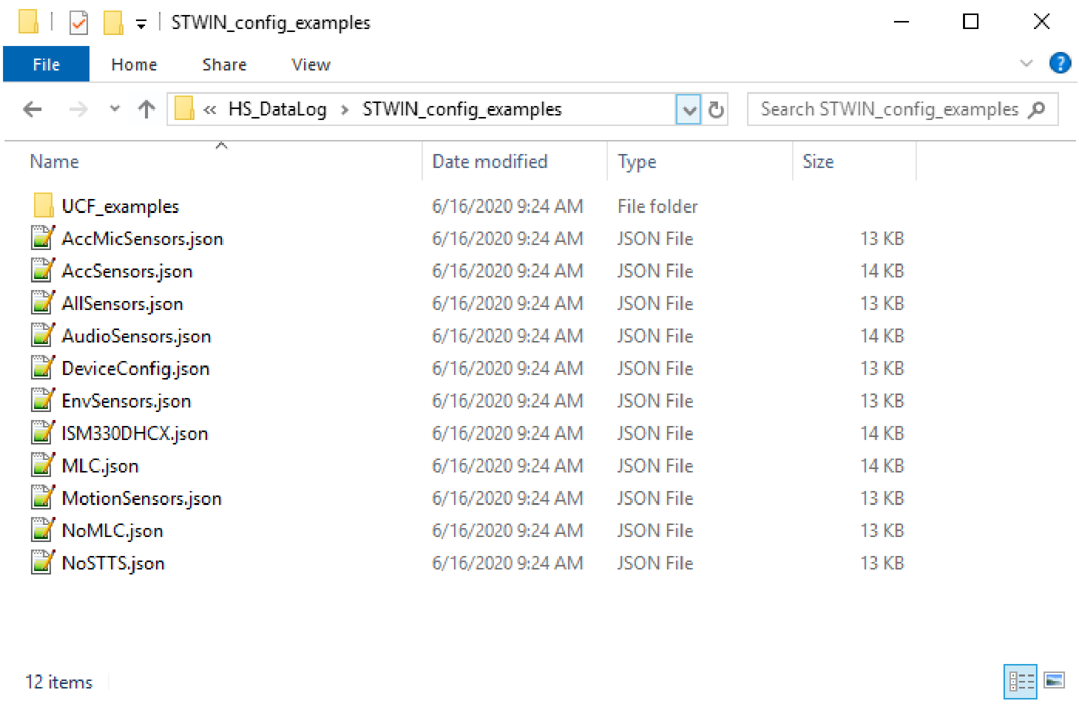


Figure 8. HSDatalog application - JSON configuration examples

By double clicking on the `USB_DataLog_Run` batch script, the application starts and the following command line appears, showing information about the connected board.

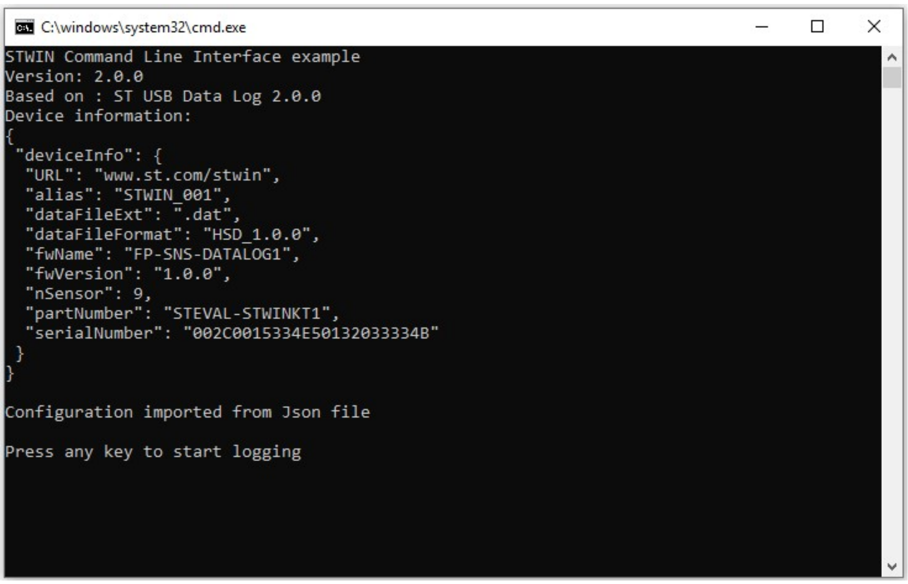
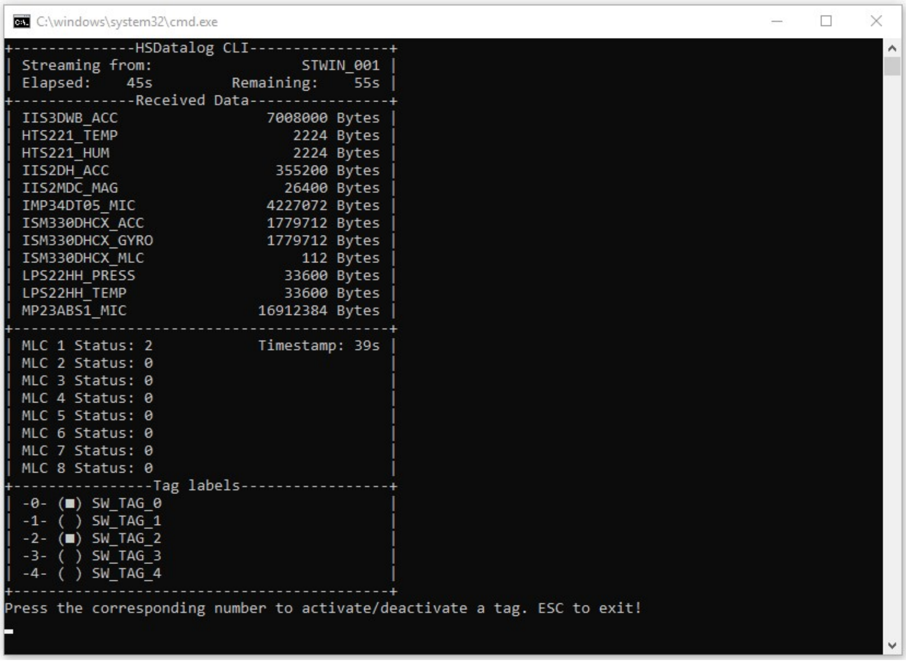


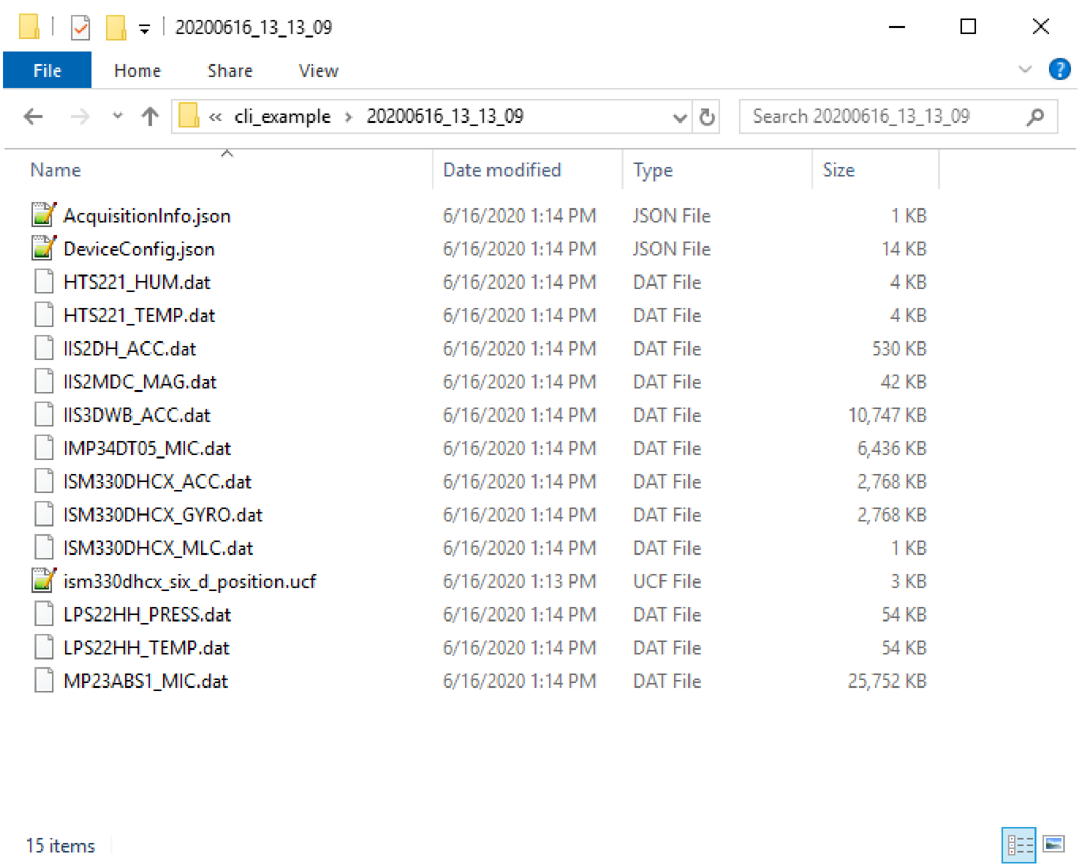
Figure 9. HSDatalog application - command line

Figure 10. **HSDatalog application - command line received data**



The application creates a YYYYMMDD_HH_MM_SS (i.e., 20200128_16_33_00) folder containing the raw data, the JSON configuration file and the UCF configuration file, if loaded.

Figure 11. **HSDatalog application - folder creation**



RELATED LINKS

2.5.1 DeviceConfig.json on page 17

2.2 SD card

To acquire sensor data and store them onto an SD card, follow the sequence of operations below.

Step 1. Insert an appropriate SD card into the STWIN board (see Section 2.2.2 SD card considerations).

Step 2. Reset the board.

The orange LED blinks once per second. If a JSON configuration file (DeviceConfig.json) is present in the root folder of the SD card, the custom sensor configuration is loaded from the file itself.

If a UCF configuration file is present in the root folder of the SD card, the MLC configuration is loaded onto the ISM330DHCX component.

If the AutoMode configuration file is present in the root folder of the SD card (execution_config.json), Automode is enabled (see Section 2.2.1 Automode).

Step 3. Press the [**USR**] button to start data acquisition on the SD card

The orange LED turns off and the greed LED starts blinking to signal sensor data is being written into the SD card.

Step 4. Press the [**USR**] button again to stop data acquisition.

Important:
Do not unplug the SD card or turn the board off before stopping the acquisition or the data on the SD card will be corrupted.

Step 5. Remove the SD card and insert it into an appropriate SD card slot on your PC.

The log files are stored in STWIN_### folders, where ### is a sequential number determined by the application to ensure log file names are unique.

Each folder contains a file for each active sub-sensor called `SensorName_subSensorName.dat` containing raw sensor data coupled with timestamps, a `DeviceConfig.json` with specific information about the device configuration, necessary for correct data interpretation, an `AcquisitionInfo.json` with information about the acquisition and the data labelling and a copy of the .ucf file used to configure the MLC, if available.

When using the SD card, it is possible to select between Continuous or Intermittent mode by changing the `HSD_SD_LOGGING_MODE` define in the `main.h` file.

Continuous Mode (default)

```
#define HSD_SD_LOGGING_MODE HSD_SD_LOGGING_MODE_CONTINUOUS
```

The acquisition can be started or stopped by pressing the USR button.

Data are stored in a single folder, one single file for each sub-sensor for the complete duration of the acquisition.

Intermittent Mode

```
#define HSD_SD_LOGGING_MODE HSD_SD_LOGGING_MODE_INTERMITTENT
```

The acquisition can be started or stopped by pressing the USR button.

In this case, the acquisition is not continuous and for each cycle:

- a. A new folder is created (STWIN_###)
- b. In this folder, data are stored in separate files for each sub-sensor over the acquisition time defined by the user with the `HSD_LOGGING_TIME_SECONDS_ACTIVE` command
- c. The acquisition is then paused for the number of seconds defined by the user with the `HSD_LOGGING_TIME_SECONDS_IDLE` command
- d. Back to step a

The duty cycle parameters can be changed in the `sdcard_manager.h` file:

```
#if (HSD_SD_LOGGING_MODE == HSD_SD_LOGGING_MODE_INTERMITTENT)
/* Define the duty cycle of the data logging */
#define HSD_LOGGING_TIME_SECONDS_IDLE 5
#define HSD_LOGGING_TIME_SECONDS_ACTIVE (15*60 - HSD_LOGGING_TIME_SECONDS_IDLE)
#endif
```

RELATED LINKS

2.5 Acquisition folders on page 15

2.2.1 Automode

HSDatalog also features the Automode, which can be initiated automatically at device power-up or reset. To enable it, a file called execution_config.json (see Section 2.5.3 execution_config.json) must be placed in the root folder of the SD card before switching on the STWIN core system board.

This mode can be used to start the datalog operations or to pause all the executions for a specific period of time by putting the sensor node in "idle" phase.

execution_config.json contains the information about the execution phases when the sensor node is working in autonomous mode (for example, phases, timer, which is the time to run an execution phase, etc.).

To customize properly the execution_config.json file, see Section 2.5.3 for further details

2.2.2 SD card considerations

Using large buffers is far more efficient than small ones when writing data to the SD card.

As the data logging application may involve large volumes of sensor data, the micro SD card must be capable of handling the data rates without issues. SD card performance varies significantly depending on the size, speed class, and even on the manufacturer.

Our sample high speed data logging application was tested with the following cards, formatted FAT32 with 32 KB allocation table:

- SanDisk 32 GB Ultra HC C10 U1 A1 (p/n SDSQUAR-032G-GN6MA)
- Verbatim 16 GB Class 10 U1 (p/n 44082)
- Transcend Premium 16 GB U1 C10 (TS16GUSDCU1)
- Kingston 8 GB HC C4 (SDC4/8 GB)

Note: *Smaller allocation tables may impact performance.*

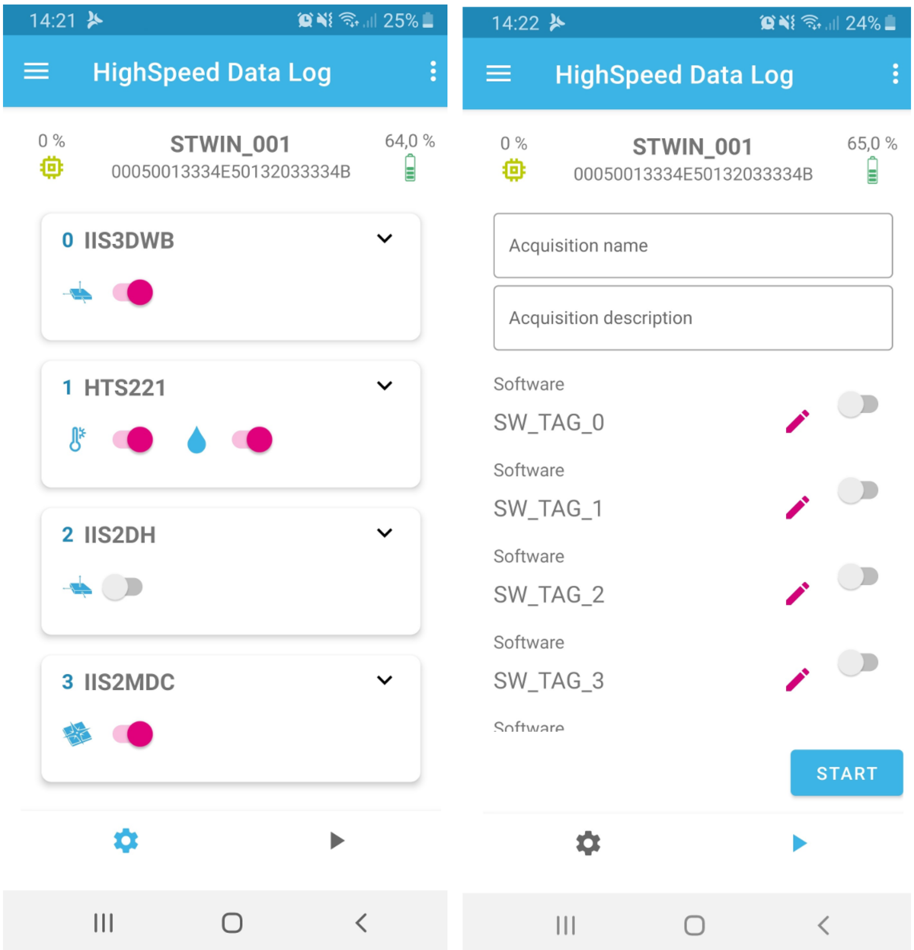
2.3 BLE control

STWIN programmed with HSDatalog can be controlled via BLE using the ST BLE Sensor Android app (version 4.7.0 and above) which lets you change the device configuration and a few sensor parameters, such as sensitivity and ODR. It also allows controlling an acquisition and managing data labelling, by activating or deactivating tags.

Through the ST BLE Sensor app you can also configure the ISM330DHCX Machine Learning Core unit and visualize its outputs.

The HSDatalog demo page contains two tabs (Configuration and Run), accessible through the bottom navigation bar.

Figure 12. HSDatalog demo page - Configuration tab (on the left) and run tab (on the right)



Under the first tab (after clicking on), you can:

- configure the device by:
  – enabling/disabling a specific sensor
  – changing sensor parameters
  – updating the device Alias
  – sending a UCF configuration file to setup the ISM330DHCX sensor Machine Learning Core. The UCF file could be retrieved either from the smartphone memory or from a cloud storage (e.g. Google Drive, Microsoft OneDrive, etc.)
- save the current device configuration on the smartphone (JSON file)
- overwrite the default device configuration so that the new one is loaded automatically at power-on (an SD card is needed to use this feature)
- load a specific device configuration (JSON file) from the smartphone

The second tab is dedicated to acquisitions settings and control. After clicking on , you can:

- start and stop an acquisition (to an SD card)
- choose which tag classes will be used for the next acquisition (both HW and SW tags)
- handle hardware and software data tagging and labelling of an ongoing acquisition
- set up the acquisition name and description

The battery status and CPU usage are always shown at the top of the two tabs.

Note: *When the acquisition starts, data are saved on the SD card inserted in the STWIN board. If the SD card is not available, data cannot be saved and the START button will be disabled.*

2.4 Data labelling

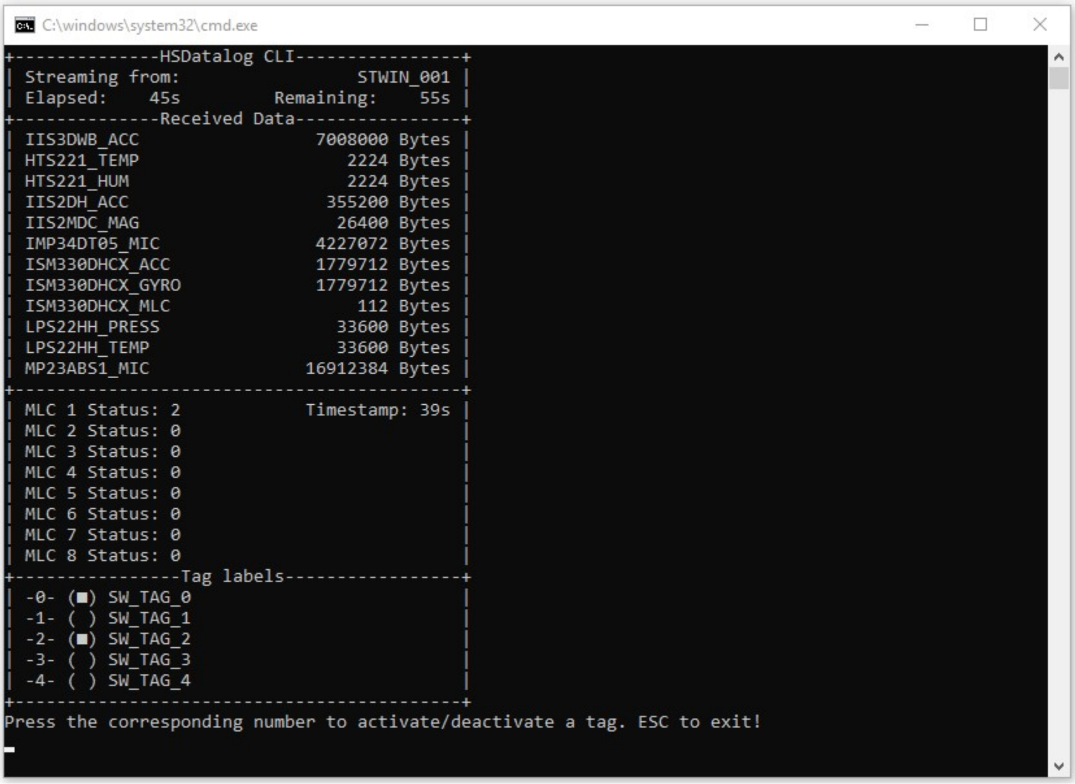
Labelled data is a group of samples that have been tagged with one or more labels. Labelled data are specifically useful in certain types of data driven algorithms such as supervised machine learning.

HSDatalog allows setting up labels to tag data during an acquisition.

Two types of tags can be used in HSDatalog: software tags and hardware tags, saved in a separate file called `AcquisitionInfo.json`, available in the acquisition folder.

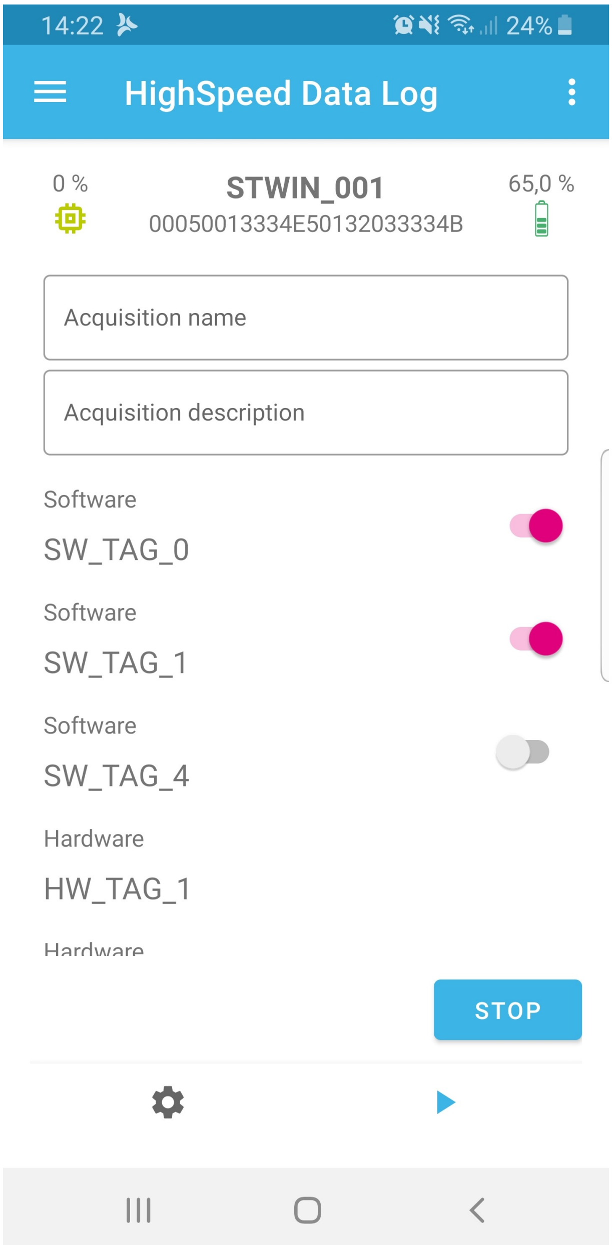
Software tags are enabled/disabled manually through the ST BLE Sensor app or the cli_example application on the PC.

Figure 13. CLI example interface - activating/deactivating software tags



```
C:\windows\system32\cmd.exe
+-------------HSDatalog CLI---------------+
| Streaming from:                STWIN_001 |
| Elapsed:    45s         Remaining:    55s |
+-------------Received Data---------------+
| IIS3DWB_ACC                7008000 Bytes |
| HTS221_TEMP                   2224 Bytes |
| HTS221_HUM                    2224 Bytes |
| IIS2DH_ACC                  355200 Bytes |
| IIS2MDC_MAG                  26400 Bytes |
| IMP34DT05_MIC              4227072 Bytes |
| ISM330DHCX_ACC             1779712 Bytes |
| ISM330DHCX_GYRO            1779712 Bytes |
| ISM330DHCX_MLC                 112 Bytes |
| LPS22HH_PRESS                33600 Bytes |
| LPS22HH_TEMP                 33600 Bytes |
| MP23ABS1_MIC              16912384 Bytes |
+-----------------------------------------+
| MLC 1 Status: 2            Timestamp: 39s |
| MLC 2 Status: 0                          |
| MLC 3 Status: 0                          |
| MLC 4 Status: 0                          |
| MLC 5 Status: 0                          |
| MLC 6 Status: 0                          |
| MLC 7 Status: 0                          |
| MLC 8 Status: 0                          |
+---------------Tag labels----------------+
| -0- (■) SW_TAG_0                         |
| -1- ( ) SW_TAG_1                         |
| -2- (■) SW_TAG_2                         |
| -3- ( ) SW_TAG_3                         |
| -4- ( ) SW_TAG_4                         |
+-----------------------------------------+
Press the corresponding number to activate/deactivate a tag. ESC to exit!
```

Figure 14. STBLESensor app - activating/deactivating software tags

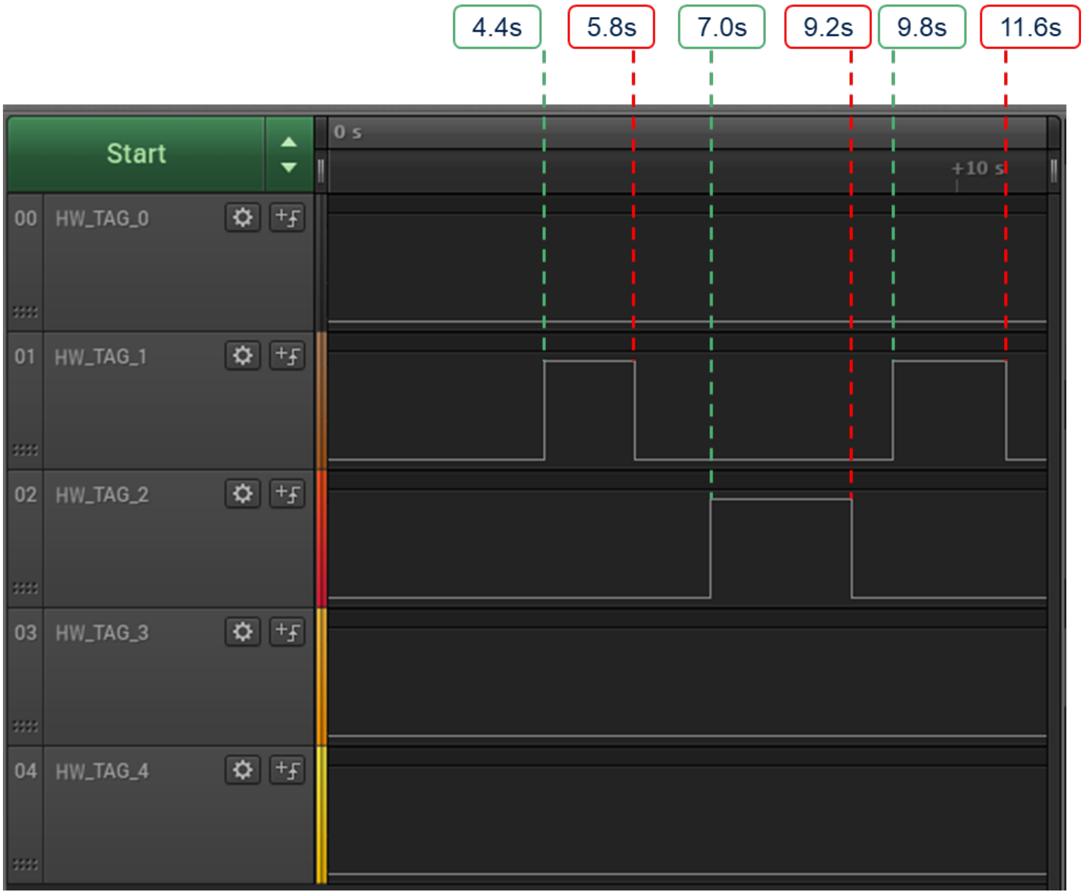


Hardware tags allow automatically enabling/disabling a tag according to the logical state of a pin on the STWIN STMOD+ connector.

This can be extremely useful when the monitored equipment already provides some electrical signals that reflect the machine status; connecting these signals to the hardware tag pins allows retrieving this information during data acquisition.

By default, five STMOD+ pins can be used as hardware tags (pins 7, 8, 9, 10 and 11). The pins are set in Pull-up configuration so that they can be used with an open-drain output pin.

Figure 15. **Hardware tag signals - example**



The AcquisitionInfo.json shown in the following picture contains the resulting tag list for the above example.

Figure 16. **Hardware tag signals - resulting tag list**

AcquisitionInfo.json


```json
{
    "Description": "descriptionTest",
    "Name": "testName",
    "Tags": [
        {
            "Enable": true,
            "Label": "HW_TAG_1",
            "t": 4.4000491666666655
        },
        {
            "Enable": false,
            "Label": "HW_TAG_1",
            "t": 5.800049166666667
        },
        {
            "Enable": true,
            "Label": "HW_TAG_2",
            "t": 7.00004945
        },
        {
            "Enable": false,
            "Label": "HW_TAG_2",
            "t": 9.200049450000002
        },
        {
            "Enable": true,
            "Label": "HW_TAG_1",
            "t": 9.800049166666666
        },
        {
            "Enable": false,
            "Label": "HW_TAG_1",
            "t": 11.600049166666668
        },
    ],
    "UUIDAcquisition": "21e890f0-1e89-4775-8c09-9fcf39163c29"
}
```


The tag labels (by default, `SW_TAG_#` and `HW_TAG_#`) can be changed by editing the DeviceConfig.json file or directly using the ST BLE Sensor app.

2.5 Acquisition folders

When an acquisition is performed, both in SD and USB modes, HSDatalog generates a folder in which you can find different files:

- DeviceConfig.json
- AcquisitionInfo.json
- raw data, saved into .dat files, whose name is based on the sensor name and type (i.e., HTTS221_HUM.dat or ISM330DHCX_GYRO.dat)
- a .ucf configuration file, if the Machine Learning Core feature of the ISM330DHCX component is enabled

Figure 17. **SD card output folder**

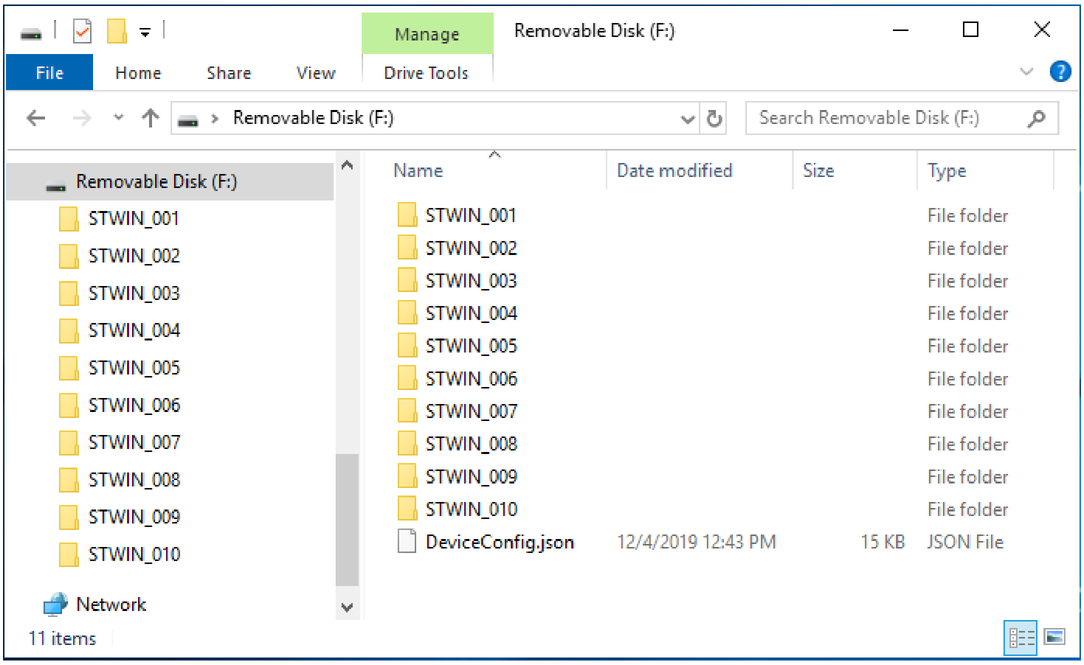
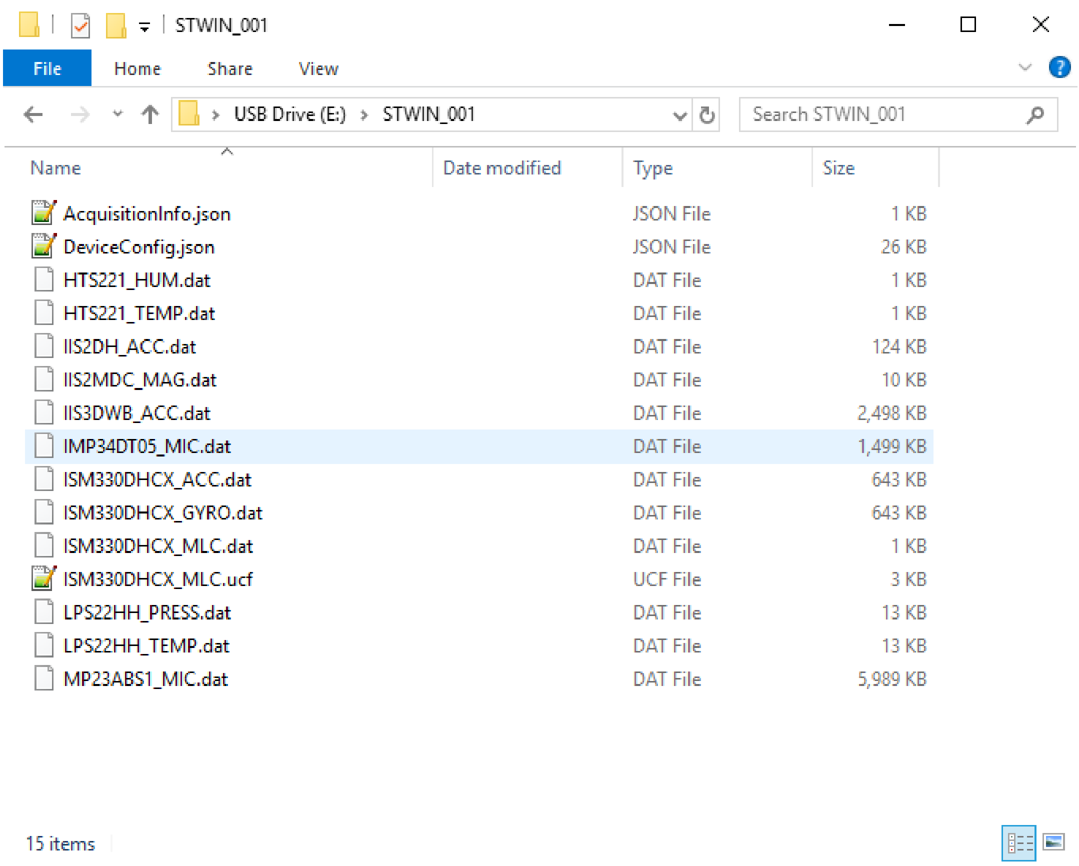


Figure 18. **SD card folder - JSON and data files**



2.5.1 DeviceConfig.json

The device consists of three attributes, `deviceInfo`, `sensor` and `tagConfig`.

Figure 19. DeviceConfig.json - device attributes

```
{
    "JSONVersion": "1.0.0",
    "UUIDAcquisition": "1330c5fb-147e-4bca-a340-6ab033ce03f0",
    "device": {
        "deviceInfo": {
        "sensor": [
        "tagConfig": {
    }
}
```

`deviceInfo` identifies the device.

Figure 20. DeviceConfig.json - deviceInfo

```
{
    "JSONVersion": "1.0.0",
    "UUIDAcquisition": "1330c5fb-147e-4bca-a340-6ab033ce03f0",
    "device": {
        "deviceInfo": {
            "URL": "www.st.com/stwin",
            "alias": "STWIN_001",
            "dataFileExt": ".dat",
            "dataFileFormat": "HSD_1.0.0",
            "fwName": "HSDatalog",
            "fwVersion": "3.0.0",
            "nSensor": 9,
            "partNumber": "STEVAL-STWINKT1",
            "serialNumber": "000E001C334E50132033334B"
        },
        "sensor": [
        "tagConfig": {
    }
}
```

`sensor` is an array of attributes to describe all the sensors available on board. Each sensor has a unique ID, a name and `sensorDescriptor` and `sensorStatus` attributes.

Figure 21. DeviceConfig.json - sensor

```
{
    "JSONVersion": "1.0.0",
    "UUIDAcquisition": "1330c5fb-147e-4bca-a340-6ab033ce03f0",
    "device": {
        "deviceInfo": {
        "sensor": [
            {
                "id": 0,
                "name": "IIS3DWB",
                "sensorDescriptor": {
                "sensorStatus": {
            },
            {
                "id": 1,
                "name": "HTS221",
                "sensorDescriptor": {
                "sensorStatus": {
            },
            {
                "id": 2,
                "name": "IIS2DH",
                "sensorDescriptor": {
                "sensorStatus": {
            },
            {
                "id": 3,
                "name": "IIS2MDC",
                "sensorDescriptor": {
                "sensorStatus": {
            },
            {
                "id": 4,
                "name": "IMP34DT05",
                "sensorDescriptor": {
                "sensorStatus": {
            },
            {
                "id": 5,
```

`sensorDescriptor` describes the main information about the single sensors through the list of its `subSensorDescriptor`. Each element of `subSensorDescriptor` describes the main information about the single sub-sensor (i.e., name, data type, sensor type, odr and full scale available, samples per unit of time supported, unit of measurement, etc.).

Figure 22. DeviceConfig.json - sensorDescriptor

```
        {
            "id": 0,
            "name": "IIS3DWB",
            "sensorDescriptor": {
                "subSensorDescriptor": [
                    {
                        "FS": [
                            2,
                            4,
                            8,
                            16
                        ],
                        "ODR": [
                            26667
                        ],
                        "dataType": "int16_t",
                        "dimensions": 3,
                        "dimensionsLabel": [
                            "x",
                            "y",
                            "z"
                        ],
                        "id": 0,
                        "samplesPerTs": {
                            "dataType": "int16_t",
                            "max": 1000,
                            "min": 0
                        },
                        "sensorType": "ACC",
                        "unit": "mg"
                    }
                ]
            },
```

`sensorStatus` describes the actual configuration of the related sensor through the list of its `subSensorStatus`. Each element of `subSensorStatus` describes the actual configuration of the single sub-sensor (i.e., whether the sensor is active or not, the actual odr, time offset, data transmitted per unit of time, full scale, etc.).

Figure 23. DeviceConfig.json - sensorStatus

```
            "sensorStatus": {
                "subSensorStatus": [
                    {
                        "FS": 16,
                        "ODR": 26667,
                        "ODRMeasured": 0,
                        "comChannelNumber": -1,
                        "initialOffset": 0,
                        "isActive": true,
                        "samplesPerTs": 1000,
                        "sdWriteBufferSize": 37000,
                        "sensitivity": 0.4880000054836273,
                        "usbDataPacketSize": 3000,
                        "wifiDataPacketSize": 0
                    }
                ]
            }
        },
```

As an example, the following figure shows the full sensor description of the STTS751 sensor available on the STWIN core system.

Figure 24. DeviceConfig.json - STTS751 sensor description example


```json
{
    "id": 8,
    "name": "STTS751",
    "sensorDescriptor": {
        "subSensorDescriptor": [
            {
                "FS": [
                    100
                ],
                "ODR": [
                    1,
                    2,
                    4
                ],
                "dataType": "float",
                "dimensions": 1,
                "dimensionsLabel": [
                    "tem"
                ],
                "id": 0,
                "samplesPerTs": {
                    "dataType": "int16_t",
                    "max": 1000,
                    "min": 0
                },
                "sensorType": "TEMP",
                "unit": "Celsius"
            }
        ]
    },
    "sensorStatus": {
        "subSensorStatus": [
            {
                "FS": 100,
                "ODR": 4,
                "ODRMeasured": 0,
                "comChannelNumber": -1,
                "initialOffset": 0,
                "isActive": true,
                "samplesPerTs": 20,
                "sdWriteBufferSize": 100,
                "sensitivity": 1,
                "usbDataPacketSize": 16,
                "wifiDataPacketSize": 0
            }
        ]
    }
}
```


The `tagConfig` attribute describes the labels activated by the user.

Figure 25. DeviceConfig.json - tagConfig attribute

```
    "tagConfig": {
        "hwTags": [
            {
                "enabled": false,
                "id": 0,
                "label": "HW_TAG_0",
                "pinDesc": ""
            },
            {
                "enabled": false,
                "id": 1,
                "label": "HW_TAG_1",
                "pinDesc": ""
            },
            {
            {
            {
                "enabled": false,
                "id": 4,
                "label": "HW_TAG_4",
                "pinDesc": ""
            }
        ],
        "maxTagsPerAcq": 100,
        "swTags": [
            {
                "id": 0,
                "label": "SW_TAG_0"
            },
            {
                "id": 1,
                "label": "SW_TAG_1"
            },
            {
            {
            {
                "id": 4,
                "label": "SW_TAG_4"
            }
        ]
    }
```

2.5.2 AcquisitionInfo.json

The AcquisitionInfo.json file contains complementary information regarding the acquisition and the list of selected labels and tags (if labelling is enabled by the user through the ST BLE Sensor app or through the CLI).

You can see an example in the following figure.

Figure 26. AcquisitionInfo.json attributes


```json
{
    "Name": "Test1",
    "Description": "New setup to be tested",
    "UUIDAcquisition": "0cabcf29-2204-42c6-81b2-e10e3761243d",
    "Tags": [
        {
            "t": 0.35236335000000005,
            "Label": "HW_TAG_0",
            "Enable": true
        },
        {
            "t": 0.35239510833333335,
            "Label": "HW_TAG_3",
            "Enable": true
        },
        {
            "t": 4.552359125,
            "Label": "HW_TAG_0",
            "Enable": false
        },
        {
            "t": 11.952358866666665,
            "Label": "HW_TAG_0",
            "Enable": true
        },
        {
            "t": 19.752360333333332,
            "Label": "HW_TAG_3",
            "Enable": false
        },
        {
            "t": 22.152360058333334,
            "Label": "HW_TAG_3",
            "Enable": true
        },
        {
            "t": 23.152358858333331,
            "Label": "HW_TAG_0",
            "Enable": false
        }
    ]
}
```


2.5.3 execution_config.json

execution_config.json configures execution contexts and phases provides the auto-mode activation at reset and its definition.

The different parameters that can be configured in this file are:

- **info**: gives the definition of auto-mode as well as each execution context; any field present overrides firmware defaults
- **version**: is the revision of the specification
- **auto_mode**: if true, auto-mode will start after reset and node initialization
- **execution_plan**: is a sequence of maximum ten execution steps
- **start_delay_ms**: indicates the initial delay in milliseconds applied after reset and before the first execution phase starts when auto-mode is selected
- **phases_iteration**: gives the number of times the execution_plan is executed; zero indicates an infinite loop
- phase step execution context settings:
  - **datalog**
  - **timer_ms**: specifies the duration in ms of the execution phase; zero indicates an infinite time
  - **idle**
  - **timer_ms**: specifies the duration in ms of the execution phase; zero indicates an infinite time

Figure 27. execution_config.json

```json
{
    "info": {
        "version": "1",
        "auto_mode": true,
        "phases_iteration": 0,
        "start_delay_ms": 3000,
        "execution_plan": [
            "datalog",
            "idle"
        ],

        "datalog": {
            "timer_ms": 7000
        },
        "idle": {
            "timer_ms": 3000
        }
    }
}
```

2.5.4 MLC configuration file (.ucf)

To set up the Machine Learning Core or the Finite State Machine, it is required a list of register configuration (register + data), saved in a text file with .ucf extension. You can build a ucf configuration file using the Unico-GUI tool or you can download a ready-to-use example from the official ST github (https://github.com/STMicroelectronics/STMems_Machine_Learning_Core).

Once the .ucf is available, you can pass this configuration file to the STWIN via Command Line Interface (see Section 2.1), via SD card (see Section 2.2) or via ST BLE Sensor app (see Section 2.3).

Figure 28. ucf configuration file

ism330dhcx_six_d_position.ucf

```
-- Machine Learning Core Tool v1.0.3.0 Beta, ISM330DHCX

Ac 10 00
Ac 11 00
Ac 01 80
Ac 05 00
Ac 17 40
Ac 02 11
Ac 08 EA
Ac 09 58
Ac 02 11
Ac 08 EB
Ac 09 03
Ac 02 11
Ac 08 EC
Ac 09 62
Ac 02 11
Ac 08 ED
Ac 09 03
Ac 02 11
Ac 08 EE
Ac 09 00
Ac 02 11
Ac 08 EF
Ac 09 00
Ac 02 11
Ac 08 F0
Ac 09 0A
Ac 02 11
Ac 08 F2
Ac 09 10
Ac 02 11
Ac 08 FA
Ac 09 3C
Ac 02 11
Ac 08 FB
```

RELATED LINKS

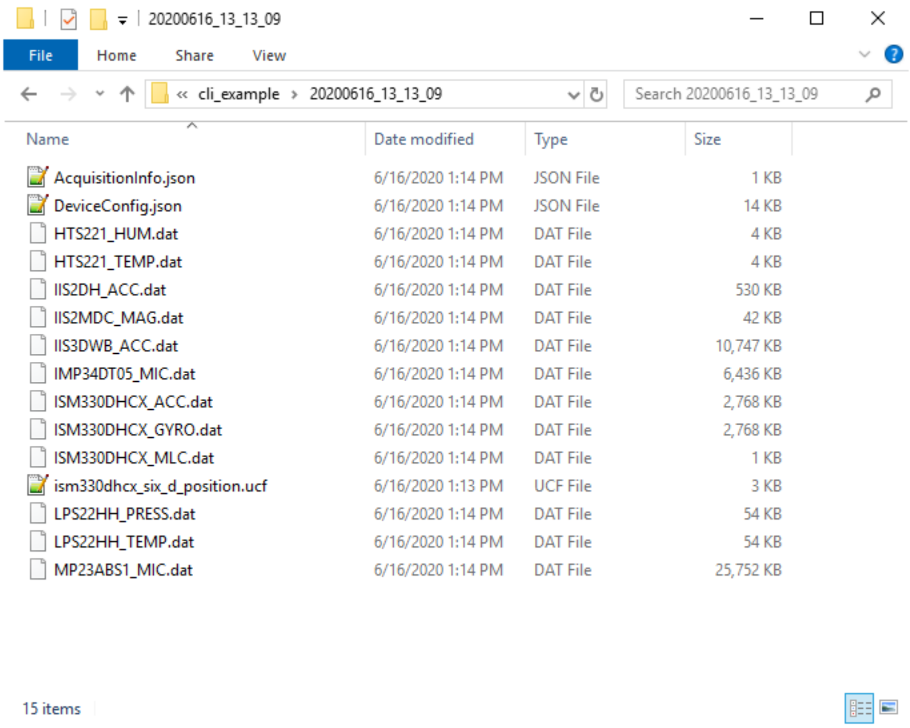
For further details on the Machine Learning Core setup, refer to AN5392

2.5.5 Raw data files (.dat)

Sensor raw data are saved in files with .dat extension. The name of the file describes the sensor part number and the sensor type, as follows:

- Name: <sensor_name>_<subsensor_type>.dat
  - <sensor_name>: component part number
  - <subsensor_type>: ACC, GYRO, MAG, HUM, TEMP, PRESS, MIC, MLC

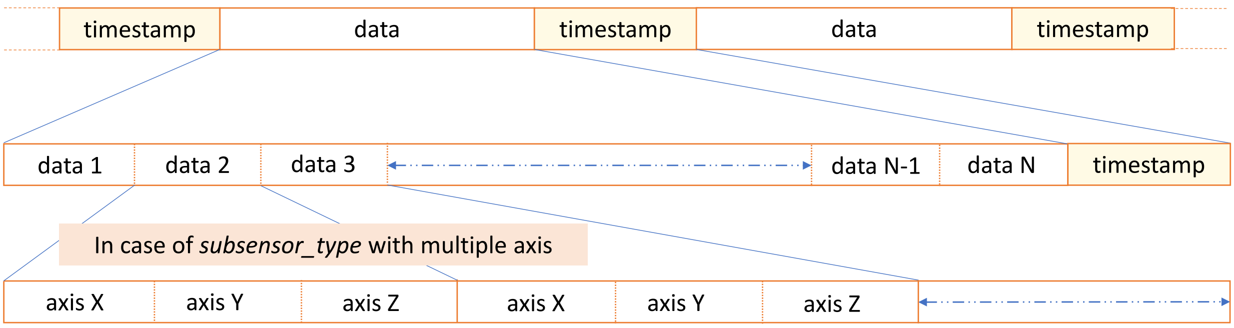
Figure 29. **Sensor raw data folder**



One file is generated for each sub-sensor. Composite sensors such as ISM330DHCX or HTS221 may thus generate multiple files. For example, HTS221_HUM.dat contains humidity raw data from the HTS221 sensor, or ISM330DHCX_GYRO.dat contains gyroscope raw data from the ISM330DHCX sensor.

A .dat file contains raw data and their timestamps. Related sensor configuration information is available in the DeviceConfig.json file. The data stream has the following structure:

Figure 30. **.dat file - data stream structure**



where

- “data k” (k = 1.. N) represents a sample generated by a `subsensor_type`.

  In case of `subsensor_type` with multiple axis, such as motion and magnetic sensors (i.e., ISM330DHCX, IIS2DH, IIS2MDC, IIS3DWB) each “data k” packet is one sample for each axis, as in the following schema: `| axis X | axis Y | axis Z |`
- length of data, in bytes (1, 2 or 4), is defined in the dataType file available in the attribute device→sensor→Descriptor→subSensorDescriptor of DeviceConfig.json
- *N* corresponds to the value of “samplesPerTs” field available in the attribute device→sensor→sensorStatus→subSensorStatus of DeviceConfig.json
- *Timestamp* is a float value calculated in seconds

2.6 PC scripts

The Utilities folder contains MATLAB and Python scripts to automatically read and plot the data saved in the log files (tested with MATLAB v2019a and Python 3.7).

A MATLAB app (`ReadSensorDataApp.mlapp`) developed and tested using the MATLAB v2019a App Designer tool is also available.

2.6.1 MATLAB scripts

The MATLAB folder contains an app (`ReadSensorData.mlapp`) and 2 scripts (`loadDatalogFiles.m` and `PlotSensorData.m`) that can be used to handle the acquired dataset in the MATLAB framework.

Both scripts use the `get_subsensorData.m` class which contains some methods used to interpret the JSON files. This class can be useful to build your standalone MATLAB application.

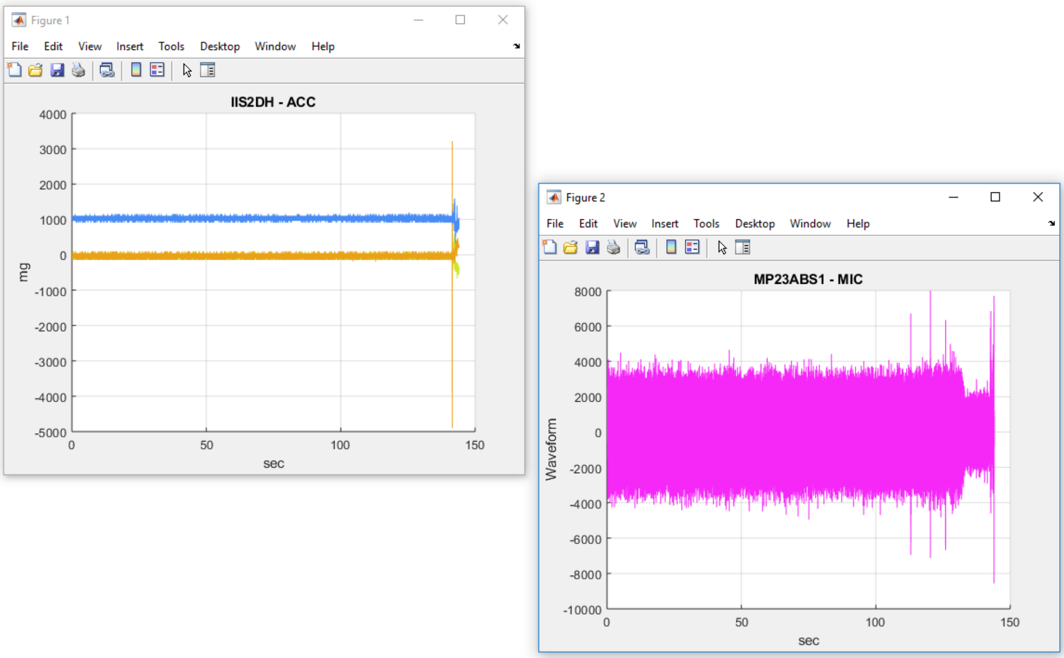
To launch the scripts:

Step 1. Open and launch `PlotSensorData` or `loadDatalogFiles` with MATLAB to automatically load or plot the data

Step 2. Once `PlotSensorData` or `loadDatalogFiles` starts, select the desired data folder from an explorer file

Step 3. Double click on the data file to interpret, in order to build the script:
- read and decode the JSON file
- read raw data and use the JSON file information to translate them into readable data (data plus timestamp)
- plot the data or load the data in a dedicated structure

Figure 31. PlotSensorData application - sensor data

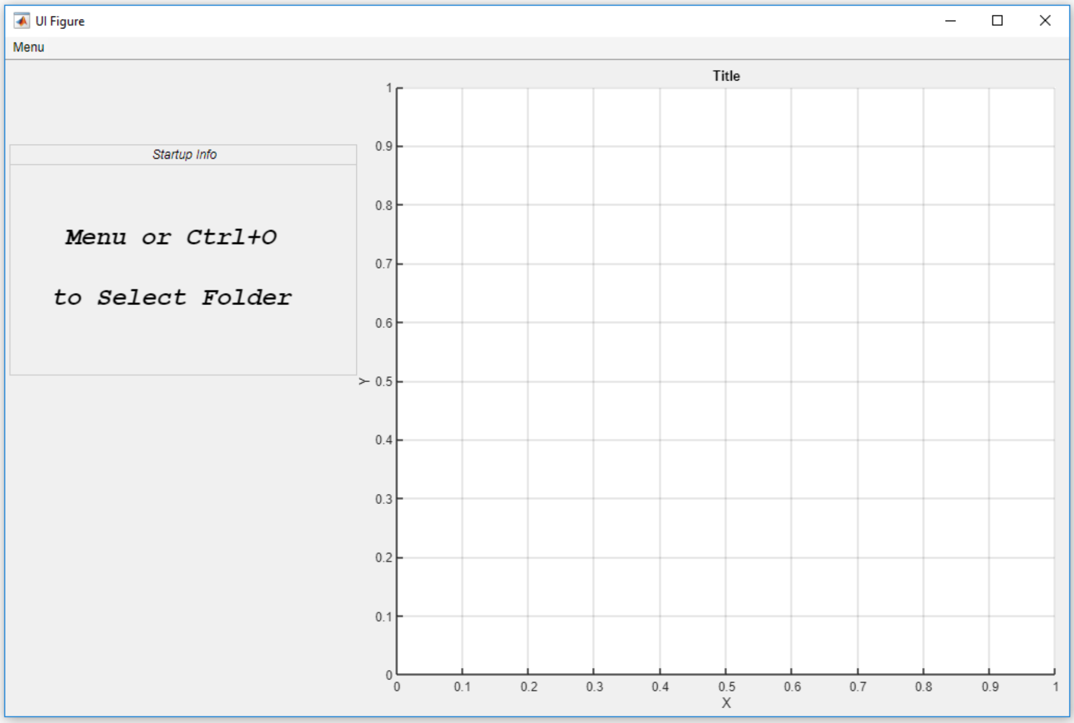


2.6.1.1 ReadSensorDataApp.mlapp

The `ReadSensorDataApp.mlapp` allows selecting the desired data through a GUI

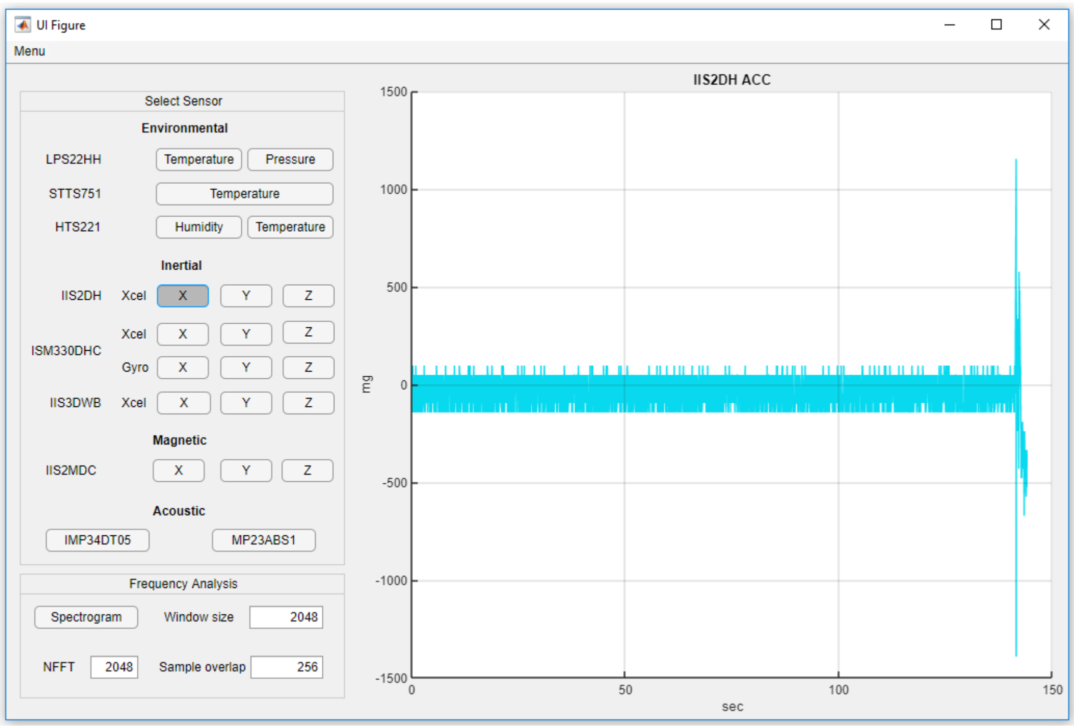
Step 1. Launch the application
The following GUI appears

Figure 32. ReadSensorDataapp.mlapp - GUI



Step 2. Select the folder and the sensor to plot

Figure 33. ReadSensorDataApp.mlapp - sensor selection



Step 3. Configure and plot the spectrogram of the signal

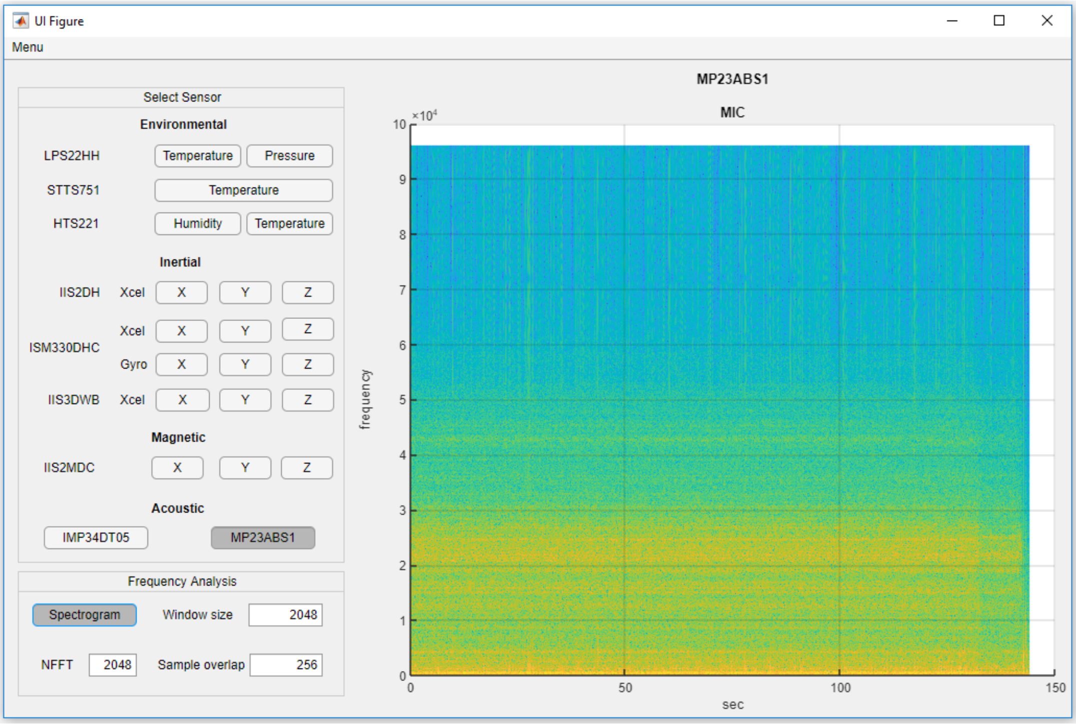


Figure 34. **ReadSensorDataApp.mlapp - sensor signal spectrogram**

2.6.2 Python SDK

The Python scripts and classes available in the HSDatalog package can be used to handle the datasets obtained by the HSDatalog firmware.

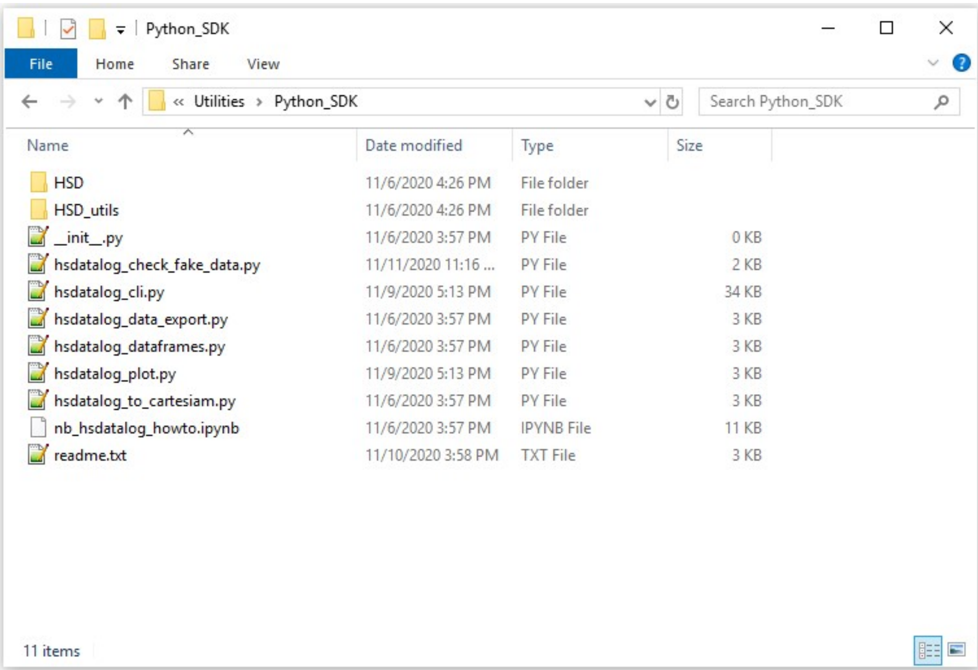
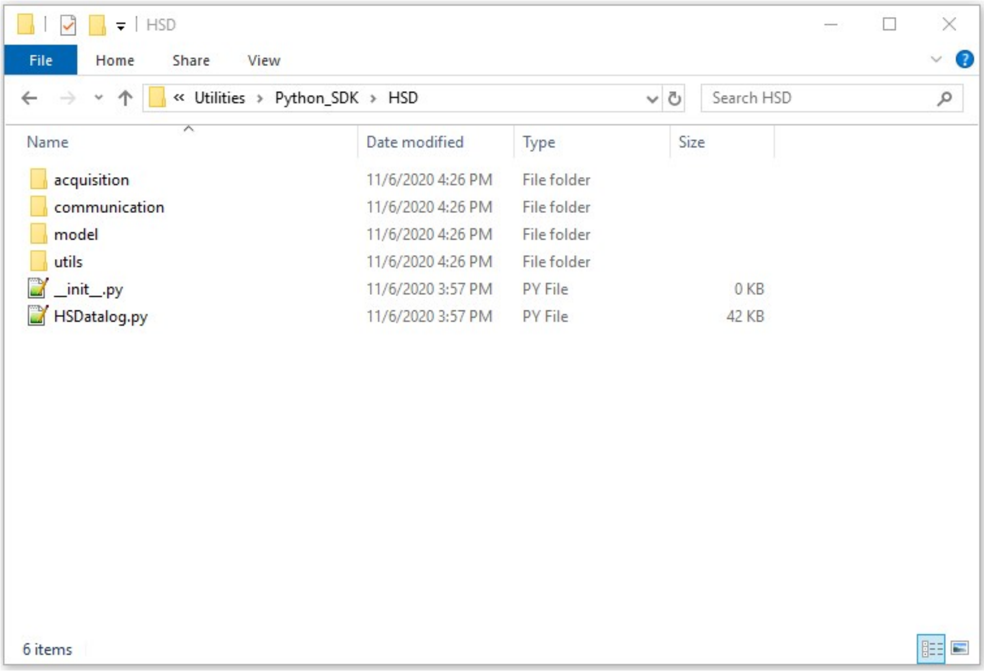


Figure 35. **Python folder**

The Python script was tested using Python 3.7 in Linux and Windows.

`HSD and HSD_utils` include some custom Python classes that you can use to build your standalone datalogger application.

Figure 36. **HSD**



The `HSDatalog.py` is the main Python module. It contains classes, methods and attributes used by the Python example scripts to properly read and handle the data logs in the format generated by HSDatalog firmware, processing information available in the JSON and raw data files.

You can start from these methods to create your own application.

HSDatalog imports the following Python libraries to be downloaded and installed (i.e: through pip command):

- numpy
- pandas
- matplotlib
- click
- colorama
- asciimatics

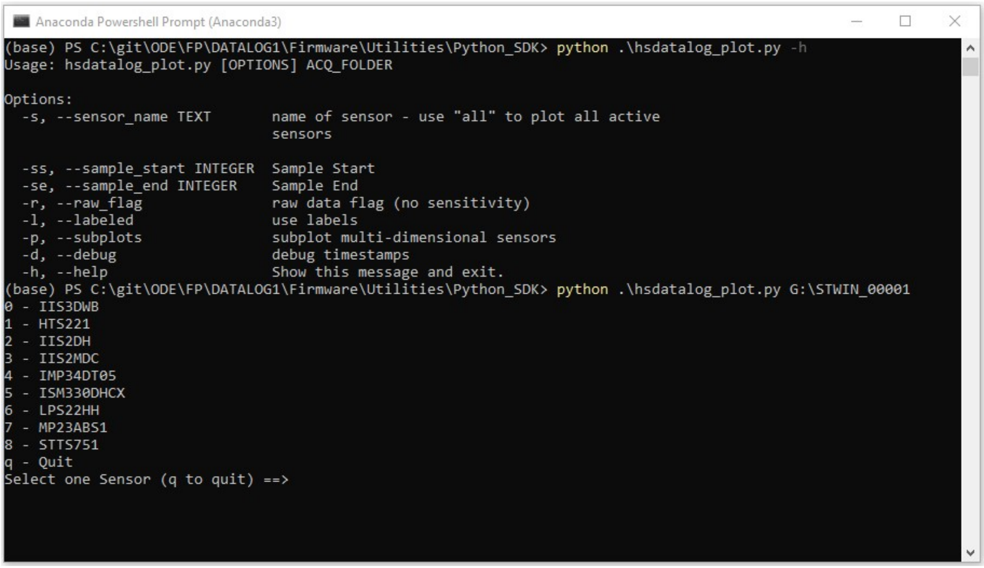
2.6.2.1 *Plot acquisition data*

hsdatalog_plot.py can be used to analyze and plot the desired data.

```
Python_SDK> python .\hsdatalog_plot.py -h

Usage: hsdatalog_plot.py [OPTIONS] ACQ_FOLDER

Options:
  -s, --sensor_name TEXT       name of sensor - use "all" to plot all active sensors
  -ss, --sample_start INTEGER  Sample Start
  -se, --sample_end INTEGER    Sample End
  -r, --raw_flag               raw data flag (no sensitivity)
  -l, --labeled                use labels
  -p, --subplots               subplot multi-dimensional sensors
  -d, --debug                  debug timestamps
  -h, --help                   Show this message and exit.
```

If the script is executed without specifying any option, on the basis of the acquisition folder, it runs in interactive mode asking the user which sensor to plot.

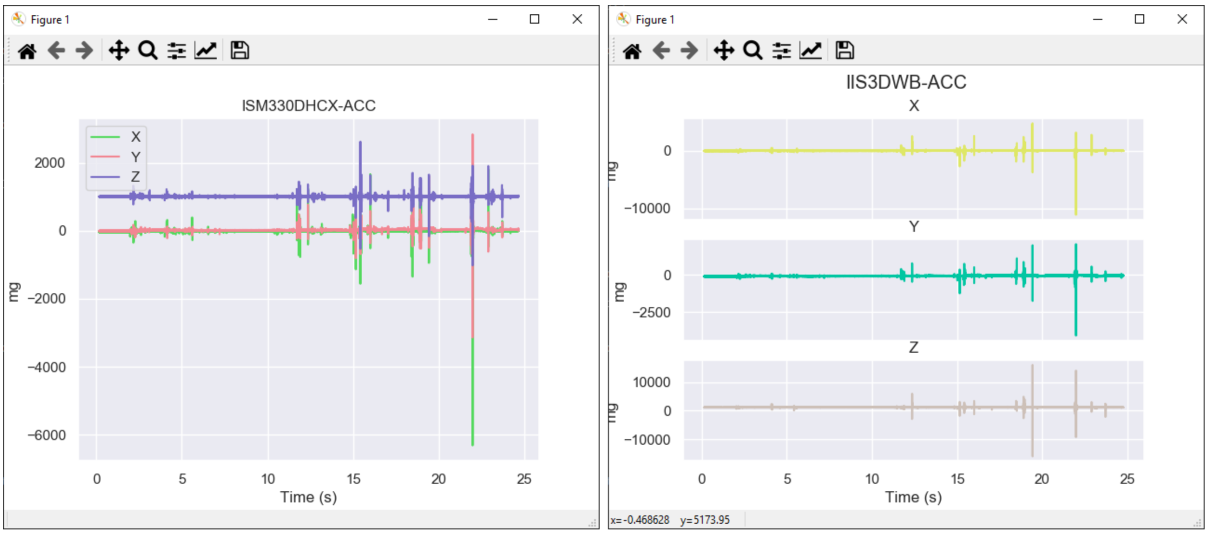
Figure 37. hsdatalog_plot application - Interactive mode



The script can also be executed in non-interactive mode. As an example, the easiest way to plot all the sensor data present in the "STWIN_00001" acquisition folder is to run:

```
python .\hsdatalog_plot.py STWIN_00001 -s all
```

Figure 38. hsdatalog_plot application - plotted data



2.6.2.2 *How to run hsdatalog_check_fake_data.py*

This script lets you verify whether the communication channel is working properly and thus if the sensor data can be streamed correctly via USB or saved correctly to the SD card.

To use the testing tool correctly, before starting any acquisition you must first recompile the HSDatalog firmware setting the `#define HSD_USE_FAKE_DATA` to 1 (in HSDCoreConfig.h file).

When `HSD_USE_FAKE_DATA` is enabled, a predefined test signal (a sawtooth signal generated with a loop counter) is streamed instead of the real sensor data.

`hsdatalog_check_fake_data` application checks if the data stored in the acquisition folder is equal to the expected test signal.

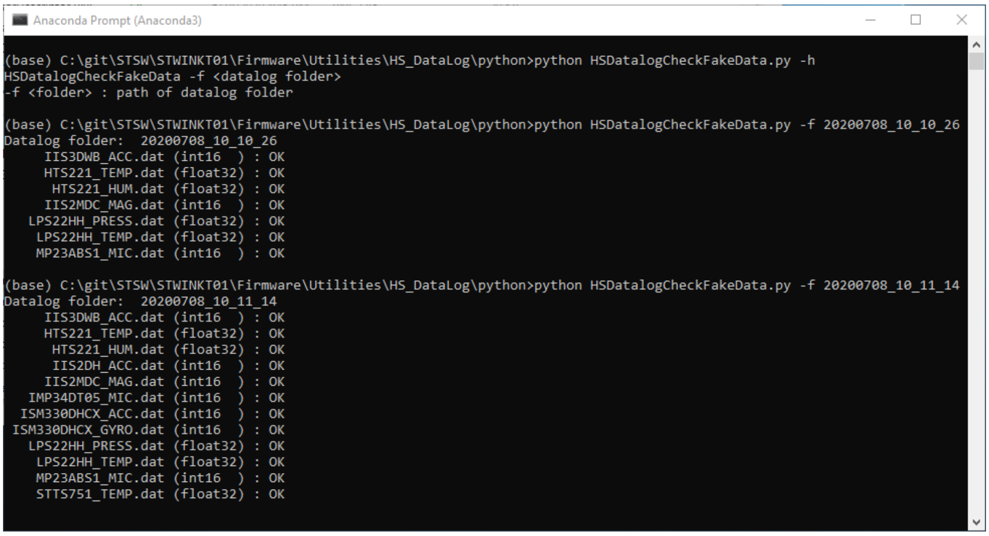
Once your testing datalog is ready:

Step 1. Copy the desired data folders.

Step 2. Run the Python script.

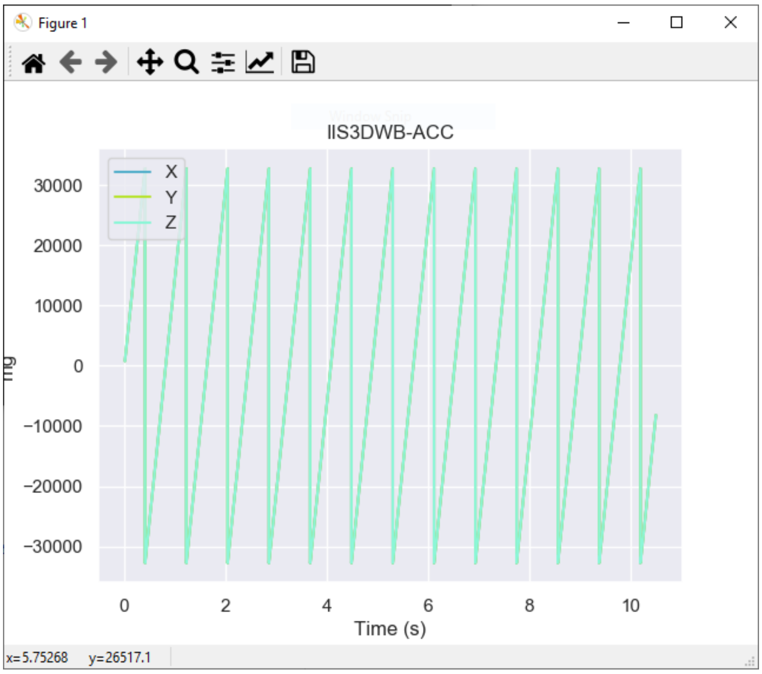
The application checks if there are any issues on the testing signals acquired for each active sensor and prints the result on the screen.

Figure 39. hsdatalog_check_fake_data.py - signal test



```
Anaconda Prompt (Anaconda3)

(base) C:\git\STSW\STWINKT01\Firmware\Utilities\HS_DataLog\python>python HSDatalogCheckFakeData.py -h
HSDatalogCheckFakeData -f <datalog folder>
-f <folder> : path of datalog folder

(base) C:\git\STSW\STWINKT01\Firmware\Utilities\HS_DataLog\python>python HSDatalogCheckFakeData.py -f 20200708_10_10_26
Datalog folder:  20200708_10_10_26
     IIS3DWB_ACC.dat (int16   ) : OK
     HTS221_TEMP.dat (float32) : OK
      HTS221_HUM.dat (float32) : OK
     IIS2MDC_MAG.dat (int16   ) : OK
   LPS22HH_PRESS.dat (float32) : OK
    LPS22HH_TEMP.dat (float32) : OK
    MP23ABS1_MIC.dat (int16   ) : OK

(base) C:\git\STSW\STWINKT01\Firmware\Utilities\HS_DataLog\python>python HSDatalogCheckFakeData.py -f 20200708_10_11_14
Datalog folder:  20200708_10_11_14
     IIS3DWB_ACC.dat (int16   ) : OK
     HTS221_TEMP.dat (float32) : OK
      HTS221_HUM.dat (float32) : OK
      IIS2DH_ACC.dat (int16   ) : OK
     IIS2MDC_MAG.dat (int16   ) : OK
   IMP34DT05_MIC.dat (int16   ) : OK
  ISM330DHCX_ACC.dat (int16   ) : OK
 ISM330DHCX_GYRO.dat (int16   ) : OK
   LPS22HH_PRESS.dat (float32) : OK
    LPS22HH_TEMP.dat (float32) : OK
    MP23ABS1_MIC.dat (int16   ) : OK
    STTS751_TEMP.dat (float32) : OK
```

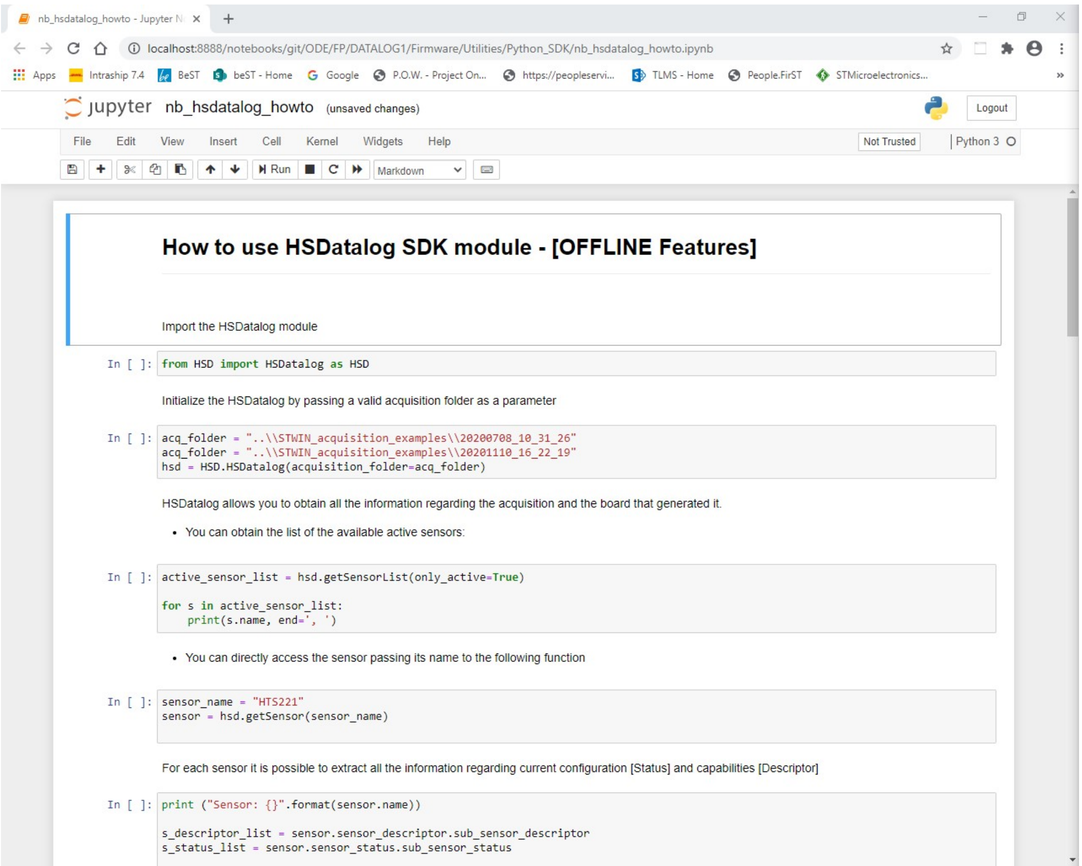
Figure 40. hsdatalog_check_fake_data.py - signal test results



2.6.2.3 *How to use HSD Python module - Jupyter Notebook*

nb_hsdatalog_howto.ipynb is a Jupyter Notebook, an open-source web application useful to create documents that contains live code and demos from Python SDK.

Figure 41. Jupyter Notebook (1 of 3)



The notebook is a step-by-step guide that shows the various functions available in the HSDatalog Python SDK.

Figure 42. Jupyter Notebook (2 of 3)

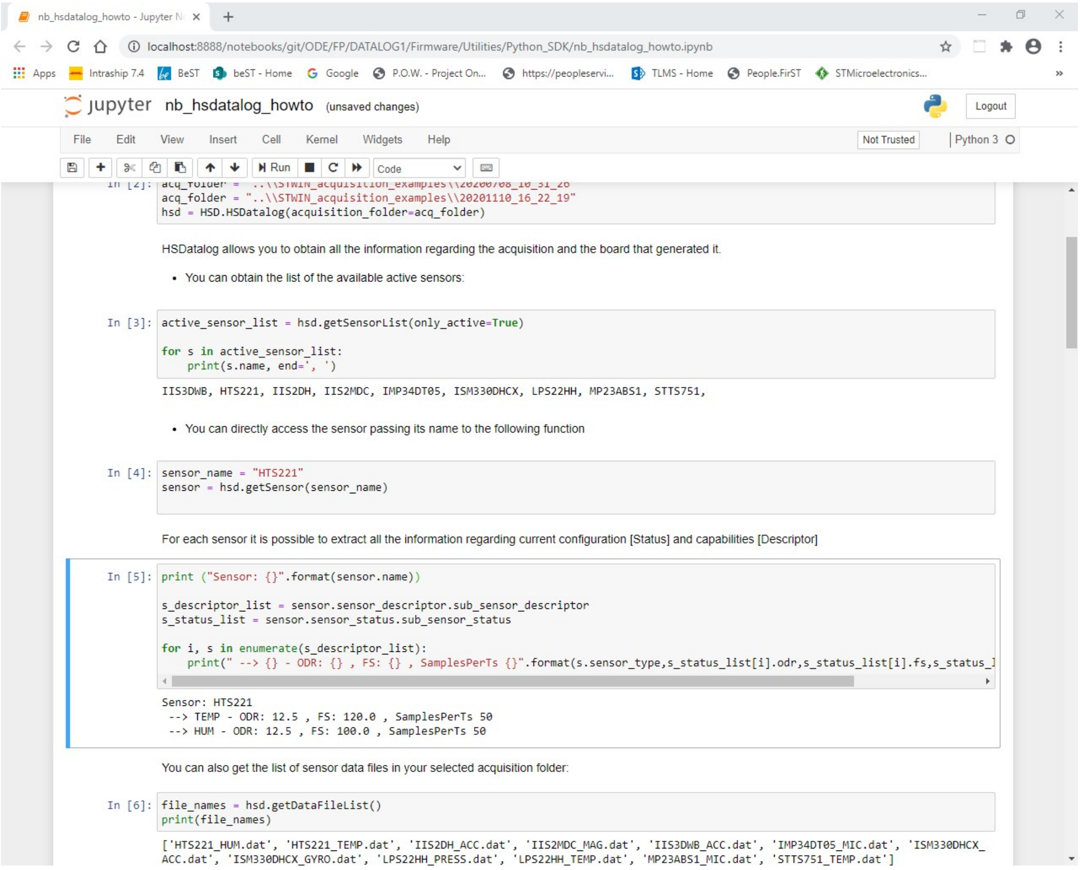
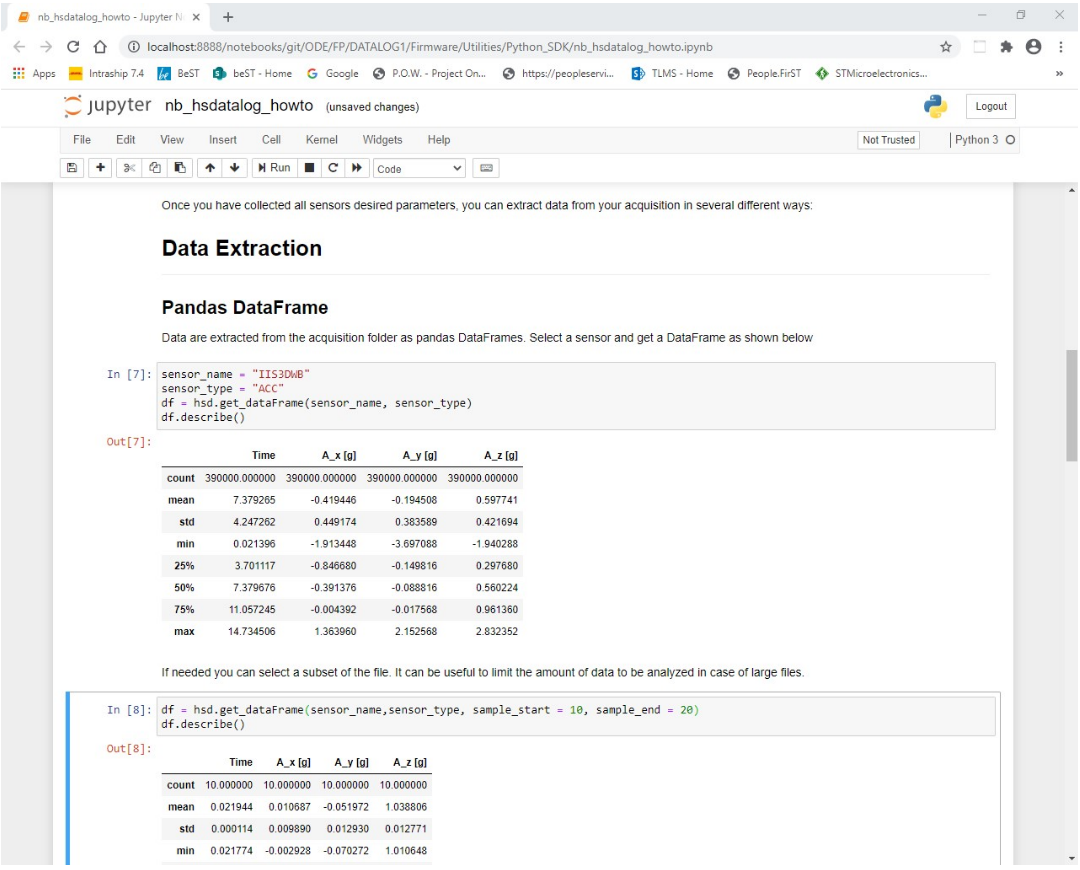


Figure 43. Jupyter Notebook (3 of 3)



Jupiter Notebook is interactive: you can easily modify the code to create your custom application, import the desired data and plot your acquisitions.

3 Firmware sensor acquisition engine

3.1 Overview

When dealing with multiple sensors running at high sampling rates on serial buses (i.e., SPI and I²C), data acquisition in blocking mode might result in long waiting times for the bus operations to end.

This processing time can be significantly reduced by the proposed software architecture, which leverages FreeRTOS and STM32 hardware capabilities.

3.2 Sensor Manager implementation

The Sensor Manager is an applicative layer based on FreeRTOS which implements the hardware specific read/write functions and the whole mechanism for managing the queue of read/writes requests for each bus (I²C/SPI) and performing the operations using DMA (non-blocking).

It implements the BUSx_Thread and BUSx message queue as shown on the right side of Figure 44. System overview.

In a standard system, several sensors can be connected to the same bus (for example, SPI or I²C) and data acquisition is performed by addressing one sensor at a time and reading data from it.

In this case, FreeRTOS manages data read requests through an SPI bus (`spi_Thread`) and an I²C (`i2c_Thread`) bus that wait for the two OS queues, `spiReqQueue_id` and `i2cReqQueue_id`, respectively. This approach can be extended to any other communication bus just by adding a new thread and a message queue.

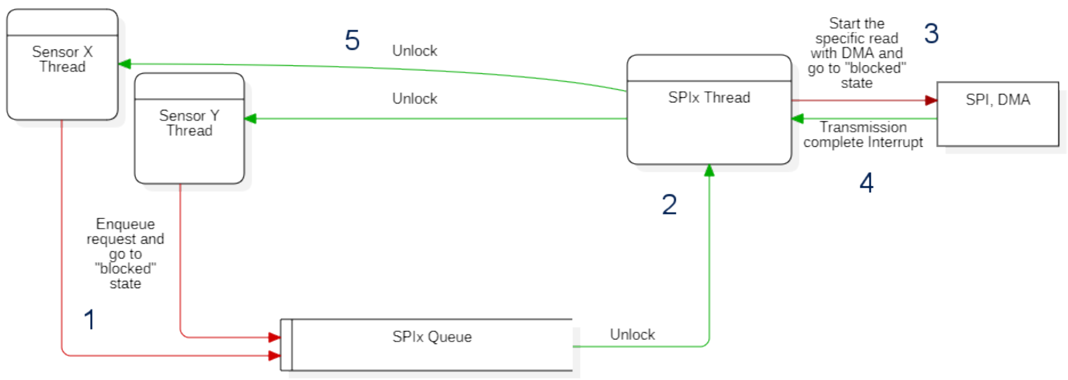
Each sensor is handled by a dedicated thread (`SensorName_Thread`) at application level to manage data acquisition from the specific sensor; when a read/write transaction is necessary, the thread appends a message to the specific bus message queue and waits for an OS semaphore, and is unblocked when the transaction is completed.

At this point, since the bus message queue is no longer empty, the bus thread wakes up and performs the actual request using DMA, after which it enters a blocked state waiting for the transaction to be completed. In this scenario, data acquisition is handled by the hardware (BUS + DMA) without any intervention of the core.

When the data transaction is completed, the DMA throws an interrupt that wakes up the bus thread, which in turn wakes up the task which originally made the request.

Reading requests from sensor threads are typically made after certain events like data ready interrupts from sensors, or as a result of timer expiration.

Figure 44. System overview



3.3 Firmware components

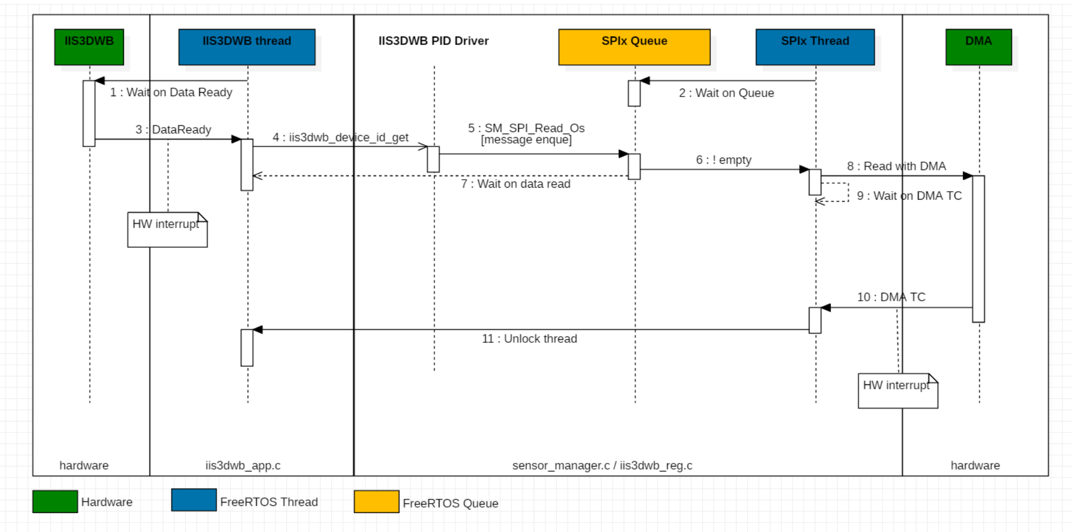
The specific implementation and the firmware example are based on the following components:

- STM32 Hardware/HAL drivers
- ST sensor PID drivers (Platform Independent Drivers)
- FreeRTOS
- Sensor Manager
- Application layer

While the example does not conform with standard BSP-based applications, it can be highly advantageous when dealing with high bandwidth sensor applications targeting low power consumption.

As an example, a more detailed overview of the IIS3DWB accelerometer acquisition sequence is shown below.

Figure 45. IIS3DWB acquisition sequence diagram



3.3.1 Platform indipendent driver (PID)

The PID driver is a low-level driver specific to each sensor to deal with all its functions and allow it to be platform-independent.

Read/write bus operations are generic and must be re-implemented for a specific board and specific hardware implementation.

The callback mechanism is used to by the PID driver to call hardware-specific functions.

3.3.2 Application layer

Each sensor has a dedicated application file called `SensorName_app.c`.

This layer is in charge of:

- linking the hardware specific functions to the PID driver
- setting up the board specific hardware needed by the sensor (for example, chip selection or interrupt pins)
- providing information to the sensor manager to perform operations correctly
- implementing the task which initializes the sensor and performs data acquisition

This layer includes many functions, from hardware configuration to data acquisition. While, it is good practice to split these functions into different layers, in this specific example they are kept together in the same file to facilitate integration into existing projects.

3.4 Data structures

The following main data structures are used to pass information among the application layers:

- Sensor context structure, part of the PID driver, which includes pointers to the read/write hardware dependent functions and a pointer to optional additional information. It is standard for all the ST PID drivers:

```
typedef int32_t (*stmdev_write_ptr) (void *, uint8_t*, uint16_t);
typedef int32_t (*stmdev_read_ptr) (void *, unint8_t*, uint16_t);
typedef struct  {
  /** Component mandatory fields  **/
  stmdev_write_ptr write_reg;
  stmdev_read_ptr  read_reg;
  /** Customizable optional pointer **/
  void  *handle;
}  stmdev_ctx_t;
```

- Sensor handler data structure, part of the sensor manager, which contains additional information on the sensor such as Chip Select PIN/PORT, I²C address, and a pointer to a freeRTOS semaphore used to block and unblock the user reading task. This data structure is linked to the void pointer of the previous structure:

```
typedef struct
{
           uint8_t WhoAmI;
           uint8_t I2C_address;
           GPIO_TypeDef*  GPIOx;
           uint16_t GPIO_pin;
           osSemaphoreId * sem;
}          sensor_handle_t;
```

The application layer, for each sensor, declares and initializes these data structures, creating a connection between the sensor PID, the sensor manager (where the functions are implemented) and the application itself:

```
sensor_handle_t  iis3dwb_hdl_instance  =  {IIS3DWB_ID, 0, IIS3DWB_SPI_CS_GPIO_Port,
IIS3DWB_SPI_CS_Pin, &iis3dwb_data_read_cmplt_sem_id};
stmdev_ctx_t iis3dwb_ctx_instance  =  {SM_SPI_Write_Os, &iis3dwb_hdl_instance};
```

The example above instantiates and first initializes a `sensor_handle_t`” for IIS3DWB (which is connected via SPI, so the `I2C_address` field is left empty); then it creates and initializes an `stmdev_ctx_t`, linking the correct sensor read/write functions and the previously declared sensor handler.

3.5 Detailed function call chain

In the function call sequence, the communication starts with a call to the PID driver from the application layer, to be performed inside a freeRTOS thread:

```
iis3dwb_device_id_get ( &iis3dwb_ctx_instance,  (unint8_t  *_ &reg0);
```

Note: *The specific sensor context is passed as a parameter.*

In general, a PID function performs a set of operations based on bus reads and writes as shown in the example below.

```
/**
            * @brief  Device Who am I. [get]
            *
            * @param  ctx    Read / wirte interface definitions. (ptr)
            * @param  buff   Buffer that stores data read
            * @retval        Interface status  (MANDATORY: return  0 → no Error).
            *
            */
int32-t |iis3dwb_device_id_get(stmdev_ctx_t  *ctx, uint8_t  *buff)
{
  int32_t ret;
  ret  =  iis3dwb_read_reg(ctx, IIS3DWB_WHO_AM_I, buff, 1);
  return ret;
}
```

This code block represents the IIS3DWB PID reading the callback call shown below.

```
/**
            * @brief  Read generic device register
            *
            * @param  ctx    read / wirte interface definitions(ptr)
            * @param  reg    register to read
            * @param  data   pointer to buffer that stores the data read(ptr)
            * @param  len    number of consecutive registers to read
            * @retval        interface status  (MANDATORY: return  0 → no Error)
            *
            */
int32-t |iis3dwb_read_reg(stmdev_ctx_t  *ctx, uint8_t reg, uint8_t* data,
                             uint16_t len)
{
  int32_t ret;
  ret  =  ctx→read_reg(ctx→handle, reg, data, len);
  return ret;
}
```

In this code fragment:

- The read function used is the one pointed to in the sensor context. In this example, it is called `SM_SPI_Read_Os` as the sensor context was initialized with this specific function (see Sensor context and handler instantiation codeblock). The read function takes the sensor handler as a parameter.
- The implementation of the `SM_SPI_Read_Os` function is based on FreeRTOS and is part of the non-blocking reading mechanism described above. This function adds a request to the reading queue and blocks the caller thread as shown below.

  The message for the reading queue also contains a pointer to the sensor handler. The reading thread can therefore perform the operation correctly (for example, it is aware of which PIN is to be used for chip selection) and unlock the correct thread at the end of the reading.

```
/**
* @brief  SPI read function: it adds a request on the SPI read queue (which will be handled
by the SPI read thread)
* @param  argument not used
* @note when the function is used and linked to the sensor context, all the calls made by
the PID driver will result in a
*       call to this function. If this is the case, be sure to make all the calls to the PID
driver functions from a freeRTOS thread
*@retval None
*/
int32_t SM_SPI_Read_Os(void * handle, uint8_t reg, uint8_t * data, uint16_t len)
{
  uint8_t autoInc = 0x00;
  SM_Message_t  * msg;

  msg = osPoolAlloc(spiPool_id);

  if  (((sensor_handle_t 8) handle) →WhoAmI  ==  IIS2DH_ID && len > 1)
  {
    autoInc = 0x40;
  }
  msg→sensor Handler = handle;
  msg→regAddr = reg | 0x80 | autoInc;
  msg→readSize = len;
  msg→dataPtr = data;

  osMessagePut (spiReqQueueid, (uint32_t) (msg), osWaitForever);
  osSemaphoreWait (* (((sensor_handle_t*)→sem), osWaitForever);

  return 0;
}
```

The following code block shows the bus reading thread.

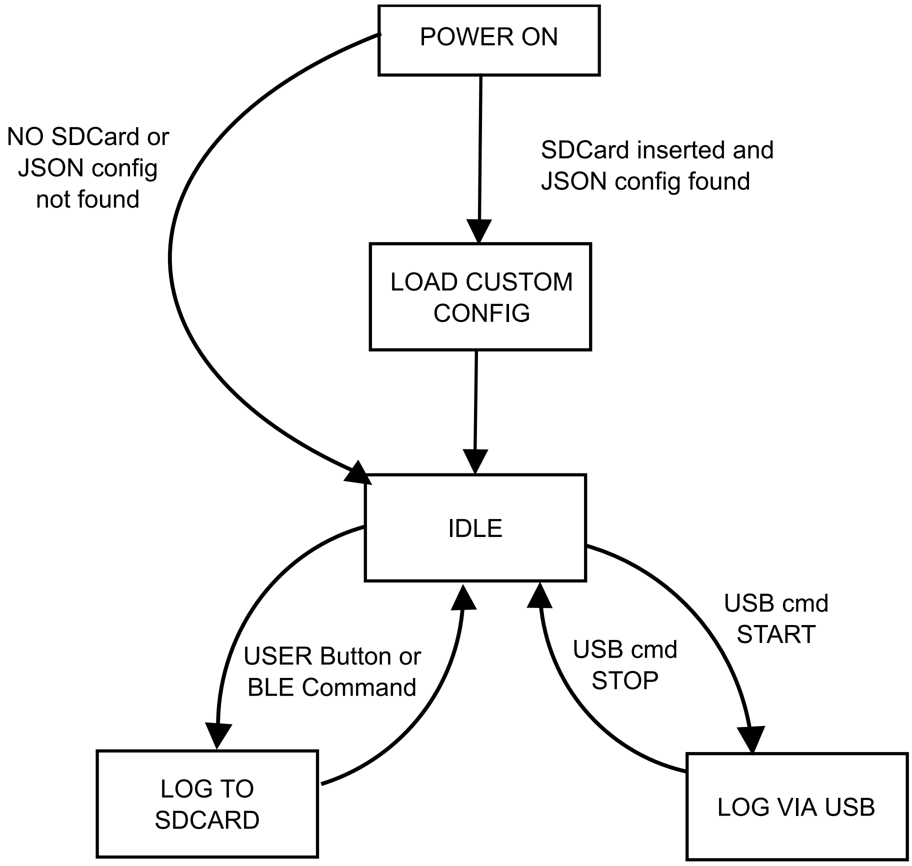
```
/**
* @brief  SPI thread: it waits for the SPI request queue, performs SPI transactions in non-
blocking mode and unlocks
*         the thread which made the request at the end of the read.
* @param  argument not used
*@retval None
*/
static void spi_Thread(void const *argument)
{
  (void) argument;

#if (configUSE_APPLICATION_TASK_TAG  ==  1 && defined(TASK_SM_SPI_DEBUG_PIN))
  vTaskSetApplicationTaskTag( NULL, (TaskHookFunction_t)TASK_SM_SPI_DEBUG_PIN );
#endif

  osEvent evt;
  for  (;;)
  {
   evt = osMessageGet(spiReqQueue_id, osWaitForever);

   SM_Message_t * msg =evt.value.p;

   HAL_GPIO_WritePin(((sensor_handle_t *)msg→sensorHandler)→GPIOx,
((sensor_handle_t*)msg→sensorHandler)→GPIO_Pin , GPIO_PIN_RESET);
   HAL_SPI_Transmit(&hsm_spi, &msg→regAddr, 1, 1000);
   HAL_SPI_TransmitReceive_DMA(&hsm_spi, msg→dataPtr, msg→readSize);

   osSemaphoreWait(spiThreadSem_id, osWaitForever);

   HAL_GPIO_WritePin(((sensor_handle_t*)msg→sensorHandler)→GPIOx, ((sensor_handle_t
*)msg→sensorHandler)→GPIO_Pin , GPIO_PIN_SET);

   osSemaphoreId * sem = ((sensor_handle_t *)msg→sensorHandler)→sem;
   osPoolFree(spiPool_id, msg);
   osSemaphoreRelease(*sem);
  }
};
```

4 Firmware data flow and device configuration

The `HSDatalog` application allows you to save data from any combination of sensors and microphones configured up to their maximum sampling rate. Sensor data are stored on a micro SD card, SDHC, formatted with the FAT32 file system, or can be streamed to a PC via USB.

At startup, the application tries to load the device configuration from the SD card (if any) and then enters Idle state, waiting for the start command either via USB or push button.

Figure 46. **Firmware data flow**



4.1 USB

4.1.1 General description

The system implements a USB-based data-logging application which allows acquisition of sensor data on a host PC.

The communication is based on USB and exploits a set of bulk endpoints exposed to the host through a custom WCID driver implemented on the firmware; this driver allows recognizing the device without any additional host drivers.

Communication functions can be split into:

- message exchange
- data transfer

For message exchange, the standard control endpoint 0 is used. JSON messages are defined and used to share information, set the devices up, configure the sensors, and so on.

For data logging, a set of bulk IN endpoints (data direction is from the device to the host) are adopted.

Data streaming channels can be:

- a physical USB endpoint: depending on the STM32 adopted, the number of endpoints can be up to 5. Data streaming is based on these endpoints. Since endpoint 0 is the standard control endpoint, endpoint numbering starts from 1 to n (0x81, 0x82 … 0x8n)
- a logical communication channel: carries data acquired from a specific sensor. Each sensor is assigned to a unique logic communication channel, identified by an ID (starting from channel 0 to channel m – 1, in case of m sensors)

When the m sensors to be logged are more than the n available endpoints, the strategy adopted is as follows:

- sensors are ordered on the basis of the required bandwidth
- logical channels are assigned to the sensors so that the sensor with the biggest bandwidth has the channel with the lowest ID
- the n USB endpoints are assigned to the m communication channels:
  - the first (n – 1) sensors are mapped to the first (n -1) endpoints to have a single endpoint dedicated to a single sensor
  - the remaining sensors are sent together to a single endpoint (the n^{th} endpoint), and to differentiate the logical channel relevant to the specific USB transaction, the first byte of the message will contain the channel number

Sensors with high bandwidth requirements have a dedicated endpoint, whereas sensors with low bandwidth requirements share a single endpoint.

The software on the host side is in charge of handling the relationship among physical/logical channels and is responsible for delivering data to the user; a dynamic library is provided within the package to interface with the firmware driver and easily exchange configuration and data between the devices and a host computer.

The driver is fully compatible with Unix-based systems.

RELATED LINKS

For further information on the WinUSB class, refer to github

4.1.2 WinUSB WCID driver

A Windows compatible ID (WCID) device is a USB device that provides extra information to a Windows system to facilitate automated driver installation and, in most circumstances, allow immediate access.

WCID allows a device to be used by a Windows application at plug in, as opposed to the standard scenario where a non-standard class USB device requires manual driver installation. WCID can extend the plug-and-play functionality of HID or Mass Storage to any USB device (that supports WCID firmware).

WCID is an extension of the WinUSB Device functionality implemented by Microsoft during the Windows 8 Developer Preview phase and which uses capabilities (Microsoft OS Descriptors, or MODs) that have been part of the operating system since Windows XP SP2.

Note: An automated WinUSB WCID driver is provided on all platforms starting from Windows Vista. On Windows 8 or later, it is native to the system, whereas for Vista and Windows 7, it can be downloaded through Windows Update. WCID devices are also supported by Linux OS.

4.1.3 WinUSB WCID driver firmware implementation

The WinUSB driver is fully compliant with the modular and hierarchic structure of the STM32 USB-FS-Device firmware library.

On top of the typical USB operations common to all USB classes (initialization, linking to the interface, etc.), some functions are provided to facilitate data transfer.

For each communication channel, you must provide:

- the packet dimension to be sent to the USB pipes; to exploit the full USB bandwidth, packets should carry at least 10 ms of data (the maximum size is fixed to 4096 bytes)
- allocated memory for internal channels

Those two requirements can be met by calling the function `USBD_WCID_STREAMING_SetTxDataBuffer(USBD_HandleTypeDef *pdev, uint8_t ch_number, uint8_t * ptr, uint16_t size)` with the appropriate parameters:

- channel number
- memory pointer
- desired size of each packet on the communication channel

Important: *The memory allocated for each channel must be at least (2 x size + 2) bytes*

The data streaming paradigm is then structured as follows:

- when data are ready, you can send them to a specific channel by calling the function `USBD_WCID_STREAMING_FillTxDataBuffer`, filling the internal buffer with the provided data.
- when the amount of provided data is at least equal to the packet dimension as previously defined by the user, the packet is sent through the assigned USB bulk endpoint

Other functions are available in the firmware to handle communication in the opposite direction (from host to device), in the following scenarios:

- message control
- data from host to device on the bulk out endpoint

Message control is performed via the USB control endpoint, exploiting well-formed USB setup messages.

Communication is always initialized by the host, which can send information to the device or ask information from the device. A variable amount of data can be attached to messages, which can be:

- get request: the host asks for information from the device, data flow is from the device to the host
- set request: the host sets parameters on the device, data flow is from the host to the device

An interface function is provided (and implemented in the firmware example) for these kinds of requests: the specific callback `(int8_t (* Control) (uint8_t, uint8_t, uint16_t, uint16_t, uint8_t * , uint16_t))` is fired when data are available on this kind of communication channel.

On top of this USB standard mechanism, a specific format is defined for exchanged messages.

Even if the most used function is data flowing from device to host via IN endpoints (for example streaming sensors data to a host), there may be cases in which the host needs to send data to the device. For this use case, an OUT endpoint is provided in the driver implementation and an interface function can be used: a specific callback `(int8_t (* Receive) (uint8_t *, uint32_t ))` is called when data are available on this kind of communication channel.

To use this function, you must assign allocated memory to the OUT endpoint through the function: `USBD_WCID_STREAMING_SetRxDataBuffer(…)`. The size of this buffer depends on the amount of data sent in each message from the host.

Note: *This function is not used in the `HSDatalog` application.*

`USBD_WCID_STREAMING_StartStreaming(…)` and `USBD_WCID_STREAMING_StopStreaming(…)` functions must be used to enable or disable the data flow.

4.1.4 USB endpoints

For this implementation, USB Full Speed is used, which supports a raw bandwidth of 12 Mbit/s, allowing roughly 5/6 Mbit/s of payload transmission.

The specific implementation is based on:

- 1 control endpoint, for the initial USB handshake procedure and control message exchange
- 1 bulk OUT endpoint, supporting data traffic from the host to the device
- n bulk IN endpoints, supporting data traffic from the device to the host

The total number of endpoints that can be used depends on the USB peripheral features of the adopted STM32.

Revision history

Table 1. Document revision history

Date	Version	Changes
24-Feb-2020	1	Initial release.
13-Jul-2020	2	Updated Introduction, Section 1.1.1 USB communication library, Section 1.1 USB mode - command line example, Section 1.2 SD card, Section 1.2.1 SD card considerations, Section 1.5 Acquisition folders, Section 1.5.1 DeviceConfig.json, Section 1.6 PC scripts, Section 1.6.1 MATLAB scripts and Section 3 Firmware data flow and device configuration. Added Section 1.3 BLE control, Section 1.4 Data labelling, Section 1.5.2 AcquisitionInfo.json, Section 1.5.3 MLC configuration file (.ucf), Section 1.5.4 Raw data files (.dat), Section 1.6.2 Python scripts and Section 1.6.2.1 How to run HSDatalogCheckFakeData.py.
13-Nov-2020	3	Updated Introduction, Section 2.1 USB mode - command line example, Section 2.2.2 SD card considerations, Section 2.3 BLE control and Section 2.5.4 MLC configuration file (.ucf), Section 2.6.2 Python SDK and Section 2.6.2.2 How to run hsdatalog_check_fake_data.py. Added Section 2.2.1 Automode, Section 2.5.3 execution_config.json, Section 2.6.2.1 Plot acquisition data and Section 2.6.2.1 Plot acquisition data.

Contents

List of figures