UM2496

User manual

Getting started with the BlueST protocol and SDK

Introduction

BlueST-SDK is a multi-platform library (Android/iOS/Python) that enables easy access to the data exported by a Bluetooth low energy (BLE) device implementing the BlueST protocol.

The protocol is easily extensible to support user-defined data and manages various data coming from inertial sensors (accelerometer, magnetometer, and gyroscope), environmental sensors (luminosity, temperature, pressure, humidity), battery information (voltage, current, charge state), and DC and stepper motors (state and commands).

1 Advertising data

The library shows just the devices with a vendor-specific field formatted as follows:

Table 1. **BlueST-SDK vendor-specific field formatting**

Length	Name	Value
1	Length	0x07/0x0D
1	Field type	0xFF
1	Protocol version	0x01
1	Device ID	0xXX
4	Feature mask	0xXXXXXXXX
6	Device MAC (optional)	0xXXXXXXXXXXXX

The field *Length* must be 7 or 13 bytes long.
The *Device Id* is a number that identifies the type of device:
- “0x00” for a generic device
- “0x01” is reserved for the STEVAL-WESU1 board
- “0x02” is reserved for the STEVAL-STLKT01V1 (SensorTile) board
- “0x03” is reserved for the STEVAL-BCNKT01V1 (BlueCoin) board
- “0x04” is reserved for the STEVAL-IDB007V2 (BlueNRG-1) and STEVAL-IDB008V2 (BlueNRG-2) boards
- “0x05” is reserved for the STEVAL-BCN002V1B (BlueNRG-Tile)
- “0x80” for a generic STM32 Nucleo board

To define your own custom boards, use Device Id values not yet assigned, if any.

Note: Values between “0x80” and “0xFF” are reserved for STM32 Nucleo boards.

The *Feature Mask* is a bit field that provides information about the features exported by the board.
The table below lists the mapping between bits and features.

Table 2. **BlueST-SDK feature mask bits and features**

Bit	Feature
0	Pedometer
1	MEMS Gesture
2	Proximity Gesture
3	Carry Position
4	Activity
5	Compass
6	Motion Intensity
7	Sensor Fusion
8	Sensor Fusion Compact
9	Free Fall
10	Accelerometer Event
11	Beam forming

Bit	Feature
12	SD Logging
13	Stepper Motor
14	DC Motor – Not yet formalized
15	CO Sensor
16	Second Temperature
17	Battery
18	Temperature
19	Humidity
20	Pressure
21	Magnetometer
22	Gyroscope
23	Accelerometer
24	Luxmeter
25	Proximity
26	Microphone Level
27	ADPCM Audio
28	Direction of arrival
29	Switch
30	ADPCM Sync
31	Analog – Not yet formalized

To understand the way the data are exported by predefined features, refer to the Feature class method `extractData()` within the feature class definition.

The *Device MAC* address is optional (used just for iOS).

2 Characteristics and features

The characteristics managed by the SDK must use a UUID as follows:
`XXXXXXXX-0001-11e1-ac36-0002a5d5c51b`
The SDK scans all the services, searching for characteristics that match the pattern.
The first part of the UUID has bits set to "1" for each feature exported by the characteristics.
In case of multiple features mapped in a single characteristic, the data must be in the same order as the bit mask.
A characteristic data format must be the following:

Table 3. **BlueST-SDK characteristic data format**

Length	Name
2	Timestamp
>1	First feature data
>1	Second feature data
...	...

The first 2 bytes are used to send a timestamp. This is especially useful to recognize any loss of data.
Since the BLE packet maximum length is 20 bytes, the maximum size of a feature data field is 18 bytes.

3 Special services

3.1 Debug

If available, the debug service has to use the following UUID:

`00000000-000E-11e1-9ab4-0002a5d5c51b`

Moreover, it contains the following characteristics:

- `00000001-000E-11e1-ac36-0002a5d5c51b` (Notify/Write): used to send string commands to the board and notify the user about the result;
- `00000002-000E-11e1-ac36-0002a5d5c51b` (Notify): used by the board to notify the user about an error message.

3.2 Configuration

If available, the configuration service has to use the following UUID:

`00000000-000F-11e1-9ab4-0002a5d5c51b`

Moreover, it contains the following characteristics:

- `00000002-000F-11e1-ac36-0002a5d5c51b`: (Notify/Write) used to send commands/data to a specific feature.

Table 4. **BlueST-SDK configuration service: request message format**

Length	Name
4	Destination feature mask
1	Command ID
0-15	Command data

The first 4 bytes select the recipient of the command/data package.

Table 5. **BlueST-SDK configuration service: optional command answer**

Length	Name
2	Timestamp
4	Sender feature mask
1	Command ID
0-13	Answer data

Messages are sent with the feature class method `sendCommand()` and answers are processed by the feature class method `parseCommandResponse()`.

If this characteristic does not exist, but the characteristics that export the desired feature are in write mode, the *Command ID* and the *Command Data parameters* are sent directly to these characteristics. In this case, it is not possible to answer the command.

- `00000001-000F-11e1-ac36-0002a5d5c51b` (Read/Write/Notify): if available, it is used to access the board configuration register that can be modified using the `ConfigControl` class.

4 Firmware and applications

The main function packs compatible with the BlueST protocol are:

- FP-SNS-MOTENV1: STM32Cube function pack for IoT node with BLE connectivity plus environmental and motion sensors;
- FP-SNS-ALLMEMS1: STM32Cube function pack for IoT node with BLE connectivity, digital microphone, environmental and motion sensors;
- FP-SNS-FLIGHT1: STM32Cube function pack for IoT node with BLE connectivity, environmental and motion sensors, time-of-flight sensors (please remove NFC when used with Python SDK);
- FP-NET-BLESTAR1: STM32Cube function pack for creating a BLE star network connected via Wi-Fi to IBM Watson IoT cloud (Android and iOS SDK only).

The protocol is also adopted in several mobile applications (Android and iOS), such as:

- STBLESensor
- STBLEStarNet

The protocol Linux version comes with significant sample applications covering different scenarios.

5 Main classes

5.1 Manager

This is a singleton class that starts/stops the discovery process and stores the retrieved nodes.

Before starting the scanning process, it is also possible to define a new device ID and to register/add new features to already defined devices.

The `Manager` notifies a new discovered node through the `ManagerListener` class. Each callback is performed asynchronously by a thread running in background.

5.2 Node

This class represents a remote device.

It allows recovering the features exported by a node and reading/writing data from/to the device.

The node exports all the features whose corresponding bit is set to “1” within the advertising data message. Once the device is connected, scanning and enabling the available characteristics can be performed. After that, it is possible to request/send data related to the discovered features.

A node notifies its RSSI (signal strength) when created.

A node can be in one of the following status:

- **Init**: dummy initial status.
- **Idle**: the node is waiting for a connection and sending an advertising data message.
- **Connecting**: a connection with the node was triggered, the node is performing the discovery of device services/characteristics.
- **Connected**: connection with the node was successful.
- **Disconnecting**: ongoing disconnection; once disconnected, the node goes back to the Idle status.
- **Lost**: the device has sent an advertising data, but it is not reachable anymore.
- **Unreachable**: the connection with the node was in place, but it is not reachable anymore.
- **Dead**: dummy final status.

Each callback is performed asynchronously by a thread running in background.

5.3 Feature

This class represents the data exported by a node.

Each feature has an array of field objects that describes the exported data.

Data are received from a BLE characteristic and are contained in a Sample class.

The user is notified about new data through a listener.

Note: Each callback is performed asynchronously by a thread running in background.

Available features can be retrieved from the *Features* package.

5.4 Logging

The SDK defines a way to log feature data. Using the class `FeatureLogCSVFile`, each feature has its own log file, and the logged data are:

- Node address (for Android) or name (for iOS);
- Timestamp (for example, the message identifier);
- Raw data which are the data received by the feature class method `extractData()`;
- One column for each data extracted by the feature.

6 Format of characteristics

All the data are in Little Endian format.

6.1 Accelerometer

Default characteristic: `0x00800000-0001-11e1-ac36-0002a5d5c51b`
Feature mask bit: 23
Description: accelerometer data on the three axes

Table 6. Accelerometer data format

Bytes	Description
0	Timestamp
1	
2	X (int16)
3	
4	Y (int16)
5	
6	Z (int16)
7	

6.2 Accelerometer event

Default characteristic: `0x00000400-0001-11e1-ac36-0002a5d5c51b`
Feature mask bit: 10
Description: event detected by the accelerometer. It can contain the event, the number of steps or both.
The content of the notification changes depending on the length of the package.

Table 7. Accelerometer event data format (2-byte length)

Bytes	Description
0	Timestamp
1	
2	Event

Table 8. Accelerometer event data format (3-byte length)

Bytes	Description
0	Timestamp
1	
2	# Steps (Uint16)
3	

Table 9. **Accelerometer event data format (4-byte length)**

Bytes	Description
0	Timestamp
1	
2	Event
3	# Steps (Uint16)
4	

Event contains the type of event detected; it is an OR operation of the values listed below.

Table 10. **Accelerometer event detection**

Value	Event
0x00	No event
0x01	Orientation top right
0x02	Orientation bottom right
0x03	Orientation bottom left
0x04	Orientation top left
0x05	Orientation up
0x06	Orientation
0x08	Tilt
0x10	Free fall
0x20	Single tap
0x40	Double tap
0x80	Wake up

6.3 Activity detection

Default characteristic: `0x00000010-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 4

Description: the type of activity the user is doing

Table 11. **Activity detection data format**

Bytes	Description
0	Timestamp
1	
2	Activity

Table 12. **Activity detection allowed values**

Value	Activity
0x00	No activity
0x01	Stationary

Value	Activity
0x02	Walking
0x03	Fast Walking
0x04	Jogging
0x05	Biking
0x06	Driving

6.4 Analog

Default characteristic: `0x80000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 31

Description: getting/setting “continuous” values in a range.

Table 13. **Analog data format**

Bytes	Description
0	Timestamp
1	
2	Analog

At the moment this feature has not been formalized yet.

6.5 Audio ADPCM

Default characteristic: `0x08000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 27

Description: audio streaming at 8 bit/8 KHz encoded with the ADPCM codec.

Table 14. **Audio ADPCM data format**

Bytes	Description
0	Audio data
...	
19	

Note: For a proper decoding, you also have to use the Audio ADPCM Sync feature.

6.6 Audio ADPCM Sync

Default characteristic: `0x40000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 30

Description: sync parameters used for a proper ADPCM decoding

Table 15. **Audio ADPCM Sync data format**

Bytes	Description
0	Index (Int16)
1	

Bytes	Description
2	Previous sample (int32)
3	
4	
5	

6.7 Battery

Default characteristic: `0x00020000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 17

Description: battery state from the board (level, voltage, current, and status)

Table 16. Battery data format

Bytes	Description
0	Timestamp
1	
2	Percentage *10
3	Voltage * 1000 (int16)
4	
5	Current (int16) Current *10 (int16) if high current bit is enabled
6	
7	Status

Table 17. Battery status values

Value	Status
0x00	Low battery
0x01	Discharging
0x02	Plugged not charging
0x03	Charging
0x04	Unknown

In the case of `((status & 0x80) != 0)`, the value of the current is sent with one decimal position.

6.8 Beam forming

Default characteristic: `0x00000800-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 11

Description: it reports the current direction selected for the beamforming algorithm

Table 18. Beam forming data format

Bytes	Description
0	Timestamp
1	

Bytes	Description
2	Direction

Table 19. **Beam forming available directions**

Value	Direction
1	Top
2	Top right
3	Right
4	Bottom right
5	Bottom
6	Bottom left
7	Left
8	Top left

6.9 Carry position

Default characteristic: `0x00000008-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 3

Description: indicates how the user is carrying the device

Table 20. **Carry position data format**

Bytes	Description
0	Timestamp
1	
2	Carry position

Table 21. **Carry position value**

Value	Activity
0x00	Unknown
0x01	On desk
0x02	In hand
0x03	Near head
0x04	Shirt pocket
0x05	Trousers pocket
0x06	Arm swing

6.10 Compass

Default characteristic: `0x00000040-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 5

Description: gets the angle from the North magnetic pole

Table 22. **Compass data format**

Bytes	Description
0	Timestamp
1	
2	Carry position

6.11 CO sensor

Default characteristic: `0x00008000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 15

Description: gets the concentration of CO particle in [ppm]

Table 23. **CO sensor data format**

Bytes	Description
0	Timestamp
1	
2	CO ppm *100 (UInt32)
3	
4	
5	

6.12 DC motor

Default characteristic: `0x00004000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 14

Description: gets the status of the DC motor and sets a command for it

Table 24. **DC motor data format**

Bytes	Description
0	Timestamp
1	
2	DC motor

6.13 Direction of arrival

Default characteristic: `0x10000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 28

Description: gets the sound direction of arrival

Table 25. **Direction of arrival data format**

Bytes	Description
0	Timestamp
1	
2	Angle (int16)
3	

The angle is in the range [-360, +360].

6.14 Free fall

Default characteristic: `0x00000200-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 9

Description: signals the free fall event

Table 26. **Free fall data format**

Bytes	Description
0	Timestamp
1	
2	Free fall[1]

1. Free Fall: "0" = False, "1" = True.

6.15 Gyroscope

Default characteristic: `0x00400000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 22

Description: gyroscope data on the three axis

Table 27. **Gyroscope data format**

Bytes	Description
0	Timestamp
1	
2	X*10 (int16)
3	
4	Y*10 (int16)
5	
6	Z*10 (int16)
7	

6.16 Humidity

Default characteristic: `0x00080000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 19

Description: percentage of humidity in the air

Table 28. **Humidity data format**

Bytes	Description
0	Timestamp
1	
2	Humidity*10 (int16)
3	

6.17 Luxmeter

Default characteristic: `0x01000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 24

Description: luminosity in the room

Table 29. **Luxmeter data format**

Bytes	Description
0	Timestamp
1	
2	Lux (int16)
3	

6.18 Magnetometer

Default characteristic: `0x00200000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 21

Description: magnetometer data on the three axis

Table 30. **Magnetometer data format**

Bytes	Description
0	Timestamp
1	
2	X (int16)
3	
4	Y (int16)
5	
6	Z (int16)
7	

6.19 MEMS gesture

Default characteristic: `0x00000002-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 1

Description: detects a gesture from MEMS data

Table 31. **MEMS gesture data format**

Bytes	Description
0	Timestamp
1	
2	Gesture

Table 32. **MEMS gesture available values**

Values	Activity
0x00	Unknown
0x01	Pick up
0x02	Glance

Values	Activity
0x03	Wake up

6.20 MEMS sensor fusion

Default characteristic: `0x00000080-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 7

Description: quaternions computed by the MEMS sensor fusion algorithm

Table 33. **MEMS sensor fusion data format**

Bytes	Description
0	Timestamp
1	
2	Qi (float32)
...	
5	
6	Qj (float32)
...	
9	
10	Qk (float32)
...	
13	
14	Qs (float32)
...	
19	

Note: Values are normalized by the application to get a vector with module equal to “1”.

6.21 MEMS sensor fusion compact

Default characteristic: `0x00000100-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 8

Description: quaternions computed by the MEMS sensor fusion algorithm, sent as three values at a time, that must be already normalized

Table 34. **MEMS sensor fusion compact data format**

Bytes	Description
0	Timestamp
1	
2	Qi*10000(int16)
3	
4	Qj*10000(int16)
5	

Bytes	Description
6	Qk*10000(int16)
7	
8	Qi*10000(int16)
9	
10	Qj*10000(int16)
11	
12	Qk*10000(int16)
13	
14	Qi*10000(int16)
15	
16	Qj*10000(int16)
17	
18	Qk*10000(int16)
19	

Note: *Values are computed by the application.*

6.22 Microphone level

Default characteristic: `0x04000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 26

Description: detects an action using MEMS data

Table 35. Microphone level data format

Bytes	Description
0	Timestamp
1	
2	Microphone level
...	...

Note: *The length is not fixed and the bytes sent are considered microphone levels of different microphones.*

6.23 Motion intensity

Default characteristic: `0x00000020-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 6

Description: motion intensity index computed by the MotionID library

Table 36. Motion intensity data format

Bytes	Description
0	Timestamp
1	
2	Motion level

Note: The motion level is a number between "0" and "10".

6.24 Pedometer

Default characteristic: `0x00000001-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 0

Description: pedometer information computed by the MotionPR library

Table 37. **Pedometer data format**

Bytes	Description
0	Timestamp
1	
2	N steps (Uint32)
...	
5	
6	Step frequency (Uint16)
7	

6.25 Pressure

Default characteristic: `0x00100000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 20

Description: atmospheric pressure

Table 38. **Pressure data format**

Bytes	Description
0	Timestamp
1	
2	Pressure*100 (int32)
...	
5	

6.26 Proximity

Default characteristic: `0x02000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 25

Description: distance measured by a time-of-light sensor

Table 39. **Proximity data format**

Bytes	Description
0	Timestamp
1	
2	Distance (Uint16)
3	

Note: There are two sensors: short range and long range. If the first bit is "0", the short range is used, and the out-of-range value is "0xFE"; otherwise, the long range is used and the out-of-range value is "0x7FFE". The SDK automatically manages this difference using an out-of-range value of "0xFFFF".

6.27 Proximity gesture

Default characteristic: `0x00000004-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 2

Description: gesture detected using a set of proximity sensors

Table 40. **Proximity gesture data format**

Bytes	Description
0	Timestamp
1	
2	Gesture

Table 41. **Proximity gesture values**

Value	Activity
0x00	Unknown
0x01	Tap
0x02	Left to right
0x03	Right to left

6.28 SD logging

Default characteristic: `0x00001000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 12

Description: starts the data logging in the SensorTile board

Table 42. **SD logging data format**

Bytes	Description
0	Timestamp
1	
2	Is Enabled (UInt8)(1)
3	Feature mask (uint32)(2)
...	
6	
7	Time interval (uint32)(3)
...	
10	

1. *Is Enabled: "0" = False, "1" = True.*
2. *Bit mask with the data to enable, as specified within the advertising data.*
3. *Logging time in [seconds].*

6.29 Stepper motor

Default characteristic: `0x00002000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 13

Description: gets the status of the stepper motor and sets a command for it

Table 43. **Stepper motor data format**

Bytes	Description
0	Timestamp
1	
2	Stepper motor

Table 44. **Stepper motor allowed values when reading the status**

Value	Status
0x00	Inactive
0x01	Running

Table 45. **Stepper motor allowed values when writing commands**

Value	Command
0x00	Stop running without torque applied
0x01	Stop running with torque applied
0x02	Run forward indefinitely
0x03	Run backward indefinitely
0x04	Move steps forward
0x05	Move steps backward

6.30 Switch

Default characteristic: `0x20000000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 29

Description: gets and sets the switch status

Table 46. **Stepper motor data format**

Bytes	Description
0	Timestamp
1	
2	Switch status[1]

1. *Switch status: "0" = Off. "1" = On.*

6.31 Temperature/second temperature

Default characteristic: `0x00040000-0001-11e1-ac36-0002a5d5c51b`/`0x00010000-0001-11e1-ac36-0002a5d5c51b`

Feature mask bit: 18/16

Description: get the temperature

Table 47. Temperature data format

Bytes	Description
0	Timestamp
1	
2	Temperature*10
3	

7 Extending the SDK

To add a new feature to the SDK, two actions are required:

1. Creating the feature
2. Registering the feature

7.1 Android

7.1.1 Creating the feature

To create a valid extension of the feature class, follow the procedure below.

Step 1. Create a constructor with a node as single parameter.

The feature class has a constructor that accepts a name, a node and field description. So the code to write should be something like:

```
public MyFeature(Node n) {
    super(FEATURE_NAME, n, new Field[]{
        new Field(FEATURE_DATA_NAME, FEATURE_DATA_UNIT,FEATURE_DATA_TYPE,
                  DATA_MAX,DATA_MIN),
    });
}//MyFeature
```

A feature should be self-describing, but if you have a generic feature, you can query it to determine which field is being exported using the field array.

Step 2. Implement the `extractData` method.

This method is called each time the SDK receives a notification (or a read) from your feature. It parses the bytes and extracts the actual values sent by the node, packing them into a sample object that is notified to the user through the `Feature.FeatureListener` class. Single characteristics can export multiple features, so this method has an offset parameter that describes where the unparsed data start.

A simple implementation could be:

```
@Override
protected ExtractResult extractData(long timestamp, @NonNull byte[] data,
                                    int dataOffset) {
    if (data.length - dataOffset < 1)
        throw new IllegalArgumentException("There are no 1 byte available to read");

    Sample temp = new Sample(timestamp,new Number[]{
        data[dataOffset]
    },getFieldsDesc());

    return new ExtractResult(temp,1);
}//extractData
```

Note: The first two bytes of the notification are used to extract a timestamp. This field should be an increasing number that can be used to determine whether a sample is more recent than another or when the sample was acquired. You can extend the `DeviceTimestampFeature` class instead of the feature class: in this case, the timestamp value is the current time and you can also parse the first two bytes of the notification (the offset starts from 0 instead of 2).

Step 3. Optionally, you can create some static methods to help the user to obtain the needed value from a sample, without knowing how the sample is organized.

```
public static byte getMotionIntensity(Sample sample){
    if(hasValidIndex(sample,0))
        return sample.data[0].byteValue();
    return -1;
}//getMotionIntensity
```

7.1.2 Registering the feature

To register the feature:

- when using a BlueST characteristic like `XXXXX-0001-11e1-ac36-0002a5d5c51b`, you have to use the `addFeatureToNode` before starting the discovery:

```
SparseArray<Class<? extends Feature>> temp = new SparseArray<>();
temp.append(0x00080000, MyFeature1.class);
temp.append(0x00040000, MyFeature2.class);
try {
    Manager.addFeatureToNode(NODE_ID,temp);
} catch (InvalidFeatureBitMaskException e) {
    //you have a mask with more than 1 bit to 1
}
```

Note: Each key associated to a feature must have only 1 bit to 1

- when using a different characteristic, you have to register the characteristics/feature pair directly to the node before connecting to it:

```
UUIDToFeatureMap temp = new UUIDToFeatureMap();
temp.put(UUID.fromString("0000fe41-8e22-4541-9d4c-21edae82ed19"),
                Arrays.asList(FeatureControlLed.class,FeatureThreadReboot.class));
temp.put(UUID.fromString("0000fe42-8e22-4541-9d4c-21edae82ed19"),
    FeatureSwitchStatus.class);
…
ConnectionOption options = ConnectionOption.builder()
    .setFeatureMap(temp);
    .build()
…
node.connect(context,options)
```

7.2 iOS

7.2.1 Creating the feature

To create a custom feature you have to extend the `BlueSTSDKFeature` class and implement its abstract methods.

Step 1. Create a constructor with a node as single parameter.
The `BlueSTSDKFeature` class has a constructor that accepts a node and a name:

```
-(instancetype) initWhitNode:(BlueSTSDKNode *)node{
    self = [super initWhitNode:node name:FEATURE_NAME];
    return self;
}
```

In Swift:

```
public override init(whitNode node: BlueSTSDKNode) {
    super.init(whitNode: node, name: MyFeature.FEATURE_NAME)
}
```

Step 2. Implement the `getFieldDesc` method.
This method is not used inside the SDK, but, if you have a generic feature, you can query it to know which field is being exported.

```
-(NSArray<BlueSTSDKFeatureField*>*) getFieldsDesc{
    return @[[BlueSTSDKFeatureField createWithName:FEATURE_DATA_NAME
                                    unit:FEATURE_UNIT
                                    type:FEATURE_TYPE
                                    min:@FEATURE_MIN
                                    max:@FEATURE_MAX]];
}
```

In Swift:

```
public override func getFieldsDesc() -> [BlueSTSDKFeatureField] {
    return [BlueSTSDKFeatureField(name: MyFeature.DATA_NAME,
                                  unit: MyFeature.DATA_MIN,
                                  type: .uInt8,
                                   min: NSNumber(value: MyFeature.DATA_MIN),
                                   max: NSNumber(value: MyFeature.DATA_MAX))]
}
```

Step 3. Implement the `extractData` method.

This method is called each time the SDK receives a notification (or a read) from your feature, parses the bytes and extracts the actual values sent by the node, packing them into a `BlueSTSDKSample` object that will be notified to the user through the `BlueSTSDKFeatureDeletage`.

A single characteristic can export multiple features, so this method has an offset parameter that describes where the unparsed data start.

```
-(BlueSTSDKExtractResult*) extractData:(uint64_t)timestamp
                                  data:(NSData*)rawData
                            dataOffset:(uint32_t)offset{
    if(rawData.length-offset < 2){
        @throw [NSException
                exceptionWithName:BLUESTSDK_LOCALIZE(@"Invalid data",nil)
                reason:BLUESTSDK_LOCALIZE(@"The feature need almost 2 bytes",nil)
                userInfo:nil];
    }//if

    float angle = [rawData extractLeUInt16FromOffset:offset]/100.0f;

    NSArray *data = @[@(angle)];
    BlueSTSDKFeatureSample *sample =
         [BlueSTSDKFeatureSample sampleWithTimestamp:timestamp data:data ];
    return [BlueSTSDKExtractResult resutlWithSample:sample nReadData:2];
}
```

In Swift:

```
public override func extractData(_ timestamp: UInt64,
                                        data: Data,
                            dataOffset offset: UInt32) -> BlueSTSDKExtractResult {
    if ((data.count - Int(offset)) < 2) {
        NSException(name: NSExceptionName(rawValue: "Invalid Data"),
                  reason: "No Bytes",
                userInfo: nil).raise()
        return BlueSTSDKExtractResult(whitSample: nil, nReadData: 0)
    }
    let angle = NSNumber(value:
       (data as NSData).extractLeUInt16(fromOffset: UInt(offset)/100.0))
    let sample = BlueSTSDKFeatureSample(timestamp: timestamp, data: [angle])
    return BlueSTSDKExtractResult(whitSample: sample, nReadData: 2)
}
```

Note: *The first two bytes of the notification are used to extract a timestamp. This field should be an increasing number that can be used to know if a sample is more recent than another or when the sample was acquired. You can extend the `BlueSTSDKDeviceTimestampFeature` class instead of the `BlueSTSDKFeature` class: in this case, the timestamp value is the current time and you can parse also the first two bytes of the notification (the offset will start from 0 instead of 2).*

Step 4. As an option, you can create some static methods to help the user to get the needed value from a sample, without knowing how the sample is organized.

```
+ (int32_t)getHeartRate:(BlueSTSDKFeatureSample *)sample {
    if(sample.data.count>=HEART_RATE_INDEX)
        return [sample.data[HEART_RATE_INDEX] intValue];
    return -1;
}
```

7.2.2 Registering the feature

To register the feature:

- when using a BlueST characteristic like `XXXXX-0001-11e1-ac36-0002a5d5c51b`, you have to use the `addFeatureToNode` before starting the discovery:

```
let charMap = [
    0x00080000 : MyFeature1.self,
    0x00040000 : MyFeature2.self,
    0x00008000 : MyFeature3.self
]
let nodeId = 0x02
BlueSTSDKManager.sharedInstance().addFeature(forNode: nodeId, features: charMap)
```

Note: Each key associated to a feature must have only 1 bit to 1

- when using a different characteristic, you have to register the characteristics/feature pair directly to the node before connecting to it:

```
(NSDictionary<CBUUID *, NSArray<Class> * > *)getManagedCharacteristics {
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    CBUUID *uuid = [CBUUID UUIDWithString:@"0a000000-000D-11e1-ac36-0002a5d5c51b"];
    [dict add:uuid feature:[MyFeature1 class]];
 // add all your Features
    return dict;
}

....
    [node addExternalCharacteristics:[ self getManagedCharacteristics]];
...
```

In Swift:

```
public func getManagedCharacteristics() -> [CBUUID:[AnyClass]]{
    var temp:[CBUUID:[AnyClass]]=[:]
    temp.updateValue([MyFeature1.self],
        forKey: CBUUID(string: "0a000000-000D-11e1-ac36-0002a5d5c51b"))
    //add all your Features
        return temp;
    }
...

node.addExternalCharacteristics(getManagedCharacteristics());
...
```

7.3 Python

7.3.1 Creating the feature

To add a new feature to the SDK, extend the Feature class by following the procedure below.

Step 1. Create an array of `Field` objects that describe the data exported by the feature.

Step 2. Create a constructor that accepts only the node as a parameter. From this constructor, call the superclass constructor, passing the feature name and the feature fields.

Step 3. Implement the method `Feature.extract_data(timestamp, data, offset).`

Step 4. Implement a class method that allows to get data from a `Sample` object.

7.3.2 Registering the feature

To register a feature:

- when using BlueST bitmask for features within the advertising data, you just have to register the new feature before performing the discovery process:

```
# Adding a 'MyFeature' feature to a Nucleo device and mapping it to a custom
# '0x10000000-0001-11e1-ac36-0002a5d5c51b' characteristic.
mask_to_features_dic = {}
mask_to_features_dic[0x10000000] = my_feature.MyFeature
try:
    Manager.add_features_to_node(0x80, mask_to_features_dic)
except InvalidFeatureBitMaskException as e:
    print e
# Synchronous discovery of Bluetooth devices.
manager.discover(False, SCANNING_TIME_s)
```

- otherwise, you can register the feature after discovering a node and before connecting to it:

```
# Adding a 'FeatureHeartRate' feature to a Nucleo device and mapping it to
# the standard '00002a37-0000-1000-8000-00805f9b34fb' Heart Rate Measurement
# characteristic.
map = UUIDToFeatureMap()
map.put(uuid.UUID('00002a37-0000-1000-8000-00805f9b34fb'), feature_heart_rate.FeatureHea
rtRate)
node.add_external_features(map)
# Connecting to the node.
node.connect()
```

8 Installing the SDK

8.1 Android

8.1.1 Installation as external library

To install the SDK as an external library:

Step 1. Clone the repository.

Step 2. Add the BlueSTSDK directory as a submodule of your project: [**File**]>[**Import Module**].

8.1.2 Installation as Git submodule

To install the SDK as a Git submodule:

Step 1. Add the repository as a submodule:

```
$ git submodule add https://github.com/STMicroelectronics-
CentralLabs/BlueSTSDK_Android.git BlueSTSDK
```

Step 2. Add the SDK as a project submodule in the settings.gradle file, adding the line:

```
include ':BlueSTSDK:BlueSTSDK'
```

8.2 iOS

8.2.1 Installation as external library

To install the SDK as an external library:

Step 1. Clone the repository or add it as a Git submodule:

```
$ git submodule add https://github.com/STMicroelectronics-
CentralLabs/BlueSTSDK_iOS.git BlueSTSDK
```

Step 2. In the application project, “General” tab, open the BlueSTSDK project file as “Linked external frameworks or library”.

Step 3. Add the BlueSTSDK framework as an embedded library.

8.3 Python

8.3.1 Installation as a Python package

To make the SDK working:

Step 1. Ensure the following preconditions are met:

- the `bluepy` Python interface to Bluetooth low energy on Linux is installed:

  ```
  $ sudo pip install bluepy
  ```

- the `concurrent.futures` module to run threads is installed:

  ```
  $ sudo pip install futures
  ```

Step 2. Install the SDK:

```
$ sudo pip install blue_st_sdk
```

8.3.2 Running the SDK application examples (optional)

To run the application samples that come with the SDK:

Step 1. Clone the repository (which contains the SDK already installed as described in Section 8.3.1 Installation as a Python package and the application examples):

```
$ git clone https://github.com/STMicroelectronics-
CentralLabs/BlueSTSDK_Python.git BlueSTSDK_Python
```

Step 2. Become a super user:

```
$ sudo su
```

Step 3. Enter the `blue_st_examples` folder and run the desired script:

```
$ python example_ble_x.py
```

A References

- For Android: https://stmicroelectronics-centrallabs.github.io/BlueSTSDK_Android/javadoc/
- For iOS: https://stmicroelectronics-centrallabs.github.io/BlueSTSDK_iOS/doc/html
- For Python: https://stmicroelectronics-centrallabs.github.io/BlueSTSDK_Python/index.html

Revision history

Table 48. **Document revision history**

Date	Version	Changes
31-Oct-2018	1	Initial release.

Contents

List of tables