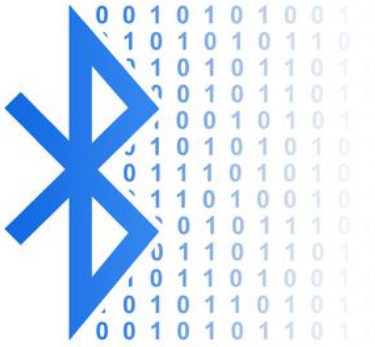
BLE Handler



Version:	1.1
Date:	May 26, 2020
Author:	Fred Dijkstra
Doc. id:	rwtej5x

Document history

Version	Date	Author(s)	Description
1.1	May 26, 2020	Fred Dijkstra	• Update for firmware 1v3 in enableSensor() and disableSensor() methods. • Changed constructor to take an interface as an argument. • Added comment about using noble-mac in the constructor. • Added missing sendBleEvents() method and changed calls in other methods to use this. • In setBleEventHandlers() removed the section about the “internal event handler”. • In connectSensor() added the call to disconnect() method when Services Discovery failed.
1.0	March 29, 2020	Fred Dijkstra	First version

Table of contents

Introduction

This document describes the *BLE Handler*, a JavaScript class that can be used to create a handler for BLE events created by using Noble.

The class is implemented in the Node.js module **`bleHandler.js`**.

BLE protocol

The sensor hosts a number of BLE services. For this demo application only the "Measurements" service is used as described in this chapter.

Base UUID

All attributes, i.e. services and characteristics have a universally unique identifier, or UUID. All attributes have a UUID that is formatted as a hexadecimal number:

`1517xxxx494711e98646d663bd873d93`

The bold characters are those that differ between the attributes. As such, the short notation of the UUID can be used.

Measurements service

The sensor hosts a number of services, but for this demo application only the "*Measurements*" service is required.

This service enables the Application to start and stop measurements on the sensor and receive sensor data comprising the orientation and the acceleration vector.

The UUID of this service is **`0x2000`** and relevant characteristics are given in table 1.

Table 1: Characteristics of the Configuration Service.

Name	UUID	Description	Length	Properties
Control	`0x2001`	Configuring and controlling the measurements.	3	● Read ● Write
Measurement	`0x2004`	Receiving the measurements.	36	● Notify

Control Characteristic

The "*Control*" characteristic is a 3-bytes data structure with the fields with their values as used in this Application as specified in table 2.

Table 2: Control characteristic structure.

Field name	Size	Description	Values
`type`	1	Type of the control target.	1
`action`	1	Start or stop the operation.	0/1
`mode`	1	Measurement mode used; orientation (quaternion).	6

Measurement Characteristic

The Measurement Characteristic is a 20-bytes data structure with the fields as specified in table 3.

Table 3: Measurement Characteristic structure.

Field name	Size	Description	Values
`timestamp`	4	Timestamp on the Sensor in microseconds.	-
`orientation`	16	The orientation expressed as a quaternion.	w,x,y,z IEEE-754 32-bit floating point

Constants

To process and create the orientation object, the Quaternion package is used.

```
var Quaterionion = require('quaternion');
```

The constants defined in the module are listed in table 4.

Table 4: Constants of the BLE Handler.

Constant	Value	Description
`SENSOR_NAME`	`"Xsens DOT"`	The default name of the sensor and used to filter discovered peripherals.
`SENSOR_ENABLE`	`1`	Enable measurements.
`SENSOR_DISABLE`	`0`	Disable measurements.
`BLE_UUID_CONTROL`	`"15172001494711e98646d663bd873d93"`	The UUID of the “Control” characteristic.
`BLE_UUID_MEASUREMENT`	`"15172002494711e98646d663bd873d93"`	The UUID of the “Measurement” characteristic.
`ROLLOVER`	`4294967295`	Rollover value for a 32-bit unsigned integer.
`CLOCK_DELTA`	`0.0002`	Offset to the clock per interval in µs for synchronization.

The clock delta is calculated as follows.

Suppose that a maximum worst case of 200 ppm (parts-per-million) relative clock drift can be expected between the system and the sensor.

This means a drift of 200 µs/second, or 0.0002 µs/µs.

Constructor

A BLE Handler is constructed by supplying the reference to the interface (i.e. an instance of EventEmitter[1]) which is required to have an **`eventHandler()`** function.

```
constructor( bleEventsInterface )
{
    ....
}
```

The supplied **`bleEventsInterface`** argument will be stored in the corresponding property

```
this.bleEventsInterface = bleEventsInterface;
```

Next the relevant properties must be initialized to be able to interface with the Noble object, starting with loading the Noble module and initializing it as the **`central`** property.

```
this.central = require('noble-mac');
```

Note that this is using a version of noble that also supports running the script on a MacOS, hence “noble-mac”. On non-Mac platforms this will use the regular noble implementation and on MacOS it will use the native binding using the official CoreBluetooth API[2].

One consequence of running on MAcOS is that on MacOS, the address of the sensors are only available after the sensors have been connected. To be able to identify the sensors on MacOS, a simple counter is used that is used to give the sensor an artificial ID.

```
this.discoveredSensorCounter = 0;
```

Finally, the handlers are set.

```
this.setBleEventHandlers();
```

The methods of the BLE Handler are discussed in the next chapter.

[1] https://nodejs.org/api/events.html
[2] https://github.com/Timeular/noble-mac

Methods

Overview

Method	Arguments	Description
setBleEventHanders	• -	Set the handlers for the central.
startScanning	• -	The scanning for new sensors can be started.
stopScanning	• -	Scanning for new sensors can be stopped.
connectSensor	• sensor	Starts the connection process for the sensor which includes connecting to the sensor and retrieving the characteristics.
enableSensor	• sensor	Starts the measurements on the sensor and enables the notifications for the sensor data.
disableSensor	• sensor	Stops the measurements on the sensor.
disconnectSensor	• sensor	Disconnects the indicated sensor.
sendBleEvents	• -	Emitting the BLE events on the supplied interface..

Although the functions are methods of the BLE Handler class, the first thing that needs to be done in most functions is retaining the reference to the object since the 'this' keyword cannot be used in the callback functions that are going to be used.

```
var bleHandler = this;
```

Set BLE event handlers

To set the handlers of the initialized central module, the **`setBleHandlers()`** method must be called.

```
setBleHandlers()
{
    var bleHandler = this,
        central    = this.central;
    ...
}
```

The "*stateChange*" event has the **`state`** as the parameter. When this parameter is equal to "poweredOn", then the BLE radio is ready which needs to be sent to the system.

```
central.on( 'stateChange', function(state)
{
    if( state == 'poweredOn' )
    {
        bleHandler.sendBleEvent( 'blePoweredOn' );
    }
});
```

The "*scanStart*" event indicates that scanning was properly started.

```
central.on( 'scanStart', function()
{
    bleHandler.sendBleEvent( 'bleScanningStarted' );
});
```

The "*scanStop*" event indicates that scanning stopped.

```
central.on( 'scanStop', function()
{
    bleHandler.sendBleEvent( 'bleScanningStopped' );
});
```

When a new peripheral is discovered (i.e. an advertisement packet was received), the "*discover*" event is generated.

```
central.on( 'discover', function(peripheral)
{
    ...
});
```

The 'local name' is present in the advertising packet and can be received.

```
var localName = peripheral.advertisement.localName;
```

Only sensors need to be detected where the **`localName`** is defined and equal to the specified **`SENSOR_NAME`** constant.

```
if( localName && localName == SENSOR_NAME ) return;
```

For some operating systems (e.g. OSX), the address is not available unless a connection has been established first. If this is the case, the address of the peripheral is set to a counter. This address will be used until the server is restarted and the sensor is discovered again.

```
if( peripheral.address == undefined || peripheral.address == "" )
{
    peripheral.address = (bleHandler.discoveredSensorCounter++).toString(16);
}
```

When a sensor is detected, the reference is copied for convenience.

```
var sensor = peripheral;
```

Next a number of properties are added to the object to create an abstraction from the actual peripheral and be able to synchronize the timestamps of the sensor.

```
sensor.name = peripheral.advertisement.localName;
sensor.characteristics = {};
sensor.systemTimestamp = 0;
sensor.sensorTimestamp = 0;
```

With this set, the system can be informed of the discovery.

```
bleHandler.sendBleEvent( 'bleSensorDiscovered', {sensor:sensor} );
```

Start scanning

To start scanning for peripherals the **`startScanning()`** method must be called. This is simply a wrapper around the **`startScanning()`** method of the Noble object.

```
startScanning()
{
    this.central.startScanning( [], false );
}
```

Note that the actual start of the scanning results in the "*scanStart*" event for which a handler was set.

Stop scanning

To stop scanning the **`stopScanning()`** method must be called. This is simply a wrapper around the **`stopScanning()`** method of the Noble object.

```
stopScanning()
{
    this.central.stopScanning();
}
```

Note that the actual stop of the scanning results in the "*scanStop*" event for which a handler was set.

Connect sensor

To start the connection process for a sensor, the **`connectSensor()`** method must be called, with the reference to the sensor as the argument.

```
connectSensor( sensor )
{
    var bleHandler = this;
    ...
}
```

Before starting the connection process it must be ensured that no event handlers were set from any previous connections.

```
sensor.removeAllListeners();
```

To have the BLE stack start the connection process, the **`connect()`** method of the sensor must be called.

```
sensor.connect( function(error)
{
    ...
});
```

If an error occurred, then this means that the connection failed and the "*bleSensorError*" event is generated.

```
if( error )
{
    bleHandler.sendBleEvent( 'bleSensorError', { sensor:sensor, error:error } );
    return;
}
```

When the sensor is properly connected, the next step is to perform a *Services Discovery* to discover all the services and characteristics hosted by the sensor. This can be done by calling the **`discoverAllServicesAndCharacteristics()`** method of the sensor.

Note that it is also an option to first discover the specific service by calling the **`discoverServices()`** method of the sensor and then specific characteristics by calling the **`discoverCharacteristics()`** method of the service. This can be more efficient when the protocol contains a lot of services and characteristics, however it does require handling two callbacks and thus more code.

```
sensor.discoverAllServicesAndCharacteristics( function(error, services, characteristics)
{
    ...
});
```

If an error occurred, then this means that the connection failed and the "*bleSensorError*" event is generated. Since the sensor was connected from the BLE Handler's perspective, it is disconnected again. Since the 'disconnect' handler was not set yet (see further on), this will not cause an unwanted 'disconnect' event.

```
if( error )
{
    bleHandler.disconnectSensor( sensor );
    bleHandler.sendBleEvent( 'bleSensorError', { sensor:sensor, error: error } );
    return;
}
```

In the callback function the references to all the characteristics can be saved by looping through the received characteristics and using it to populate the **`characteristics`** property of the sensor.

```
sensor.characteristics = {};
characteristics.forEach( function( characteristic )
{
    sensor.characteristics[characteristic.uuid] = characteristic;
});
```

A characteristic is now available via **`sensor.characteristics[<uuid>]`**.

This finishes the connection process of the sensor and therefore at this point the disconnection handler can be set as well to handle unexpected disconnections which are indicated by the "*bleSensorConnectionLost*".

```
sensor.on( 'disconnect', function()
{
    bleHandler.sendBleEvent( 'bleSensorConnectionLost', { sensor:sensor } );
});
```

Note that there is a possibility that the sensor disconnected during Services Discovery. However then the error would have been generated.

As a final step, the "*bleSensorConnected*" event must be handled to inform the system that the sensor connected.

```
bleHandler.sendBleEvent( 'bleSensorConnected', { sensor:sensor } );
```

Enable sensor

Enabling a sensor means that the "Control" characteristic must be set and the notifications for the "Measurement" characteristic must be enabled. This is implemented by the **`enableSensor()`** method which takes the reference to the sensor (i.e. a Peripheral object) as the argument.

```
enableSensor( sensor )
{
    var bleHandler = this;
    ...
}
```

The first step is to retrieve the reference to the characteristics as was described in "Connect sensor".

```
var controlCharacteristic     = sensor.characteristics[BLE_UUID_CONTROL],
    measurementCharacteristic = sensor.characteristics[BLE_UUID_MEASUREMENT];
```

To write a characteristic, its **`write()`** method must be called as described in more detail in the chapter "Writing a characteristic" in which it is described that the actual value to be written is to be supplied as a Buffer object[3]. Since the "Control" is constructed by 3-bytes, the buffer can be filled as such.

```
var buffer = new Buffer(3);
buffer[0] = 0x01;
buffer[1] = SENSOR_ENABLE;
buffer[2] = 0x05;
```

The buffer can now be used as an argument when calling the **`write()`** method.

```
controlCharacteristic.write( buffer, false, function(error)
{
    ...
});
```

If an error occurred, then this means that writing to the "Control" characteristic failed and the "*bleSensorError*" event is generated.

```
if( error )
{
    bleHandler.sendBleEvent( 'bleSensorError', { sensor:sensor, error: error } );
    return;
}
```

[3] Nodejs.org - "How to use buffers"

To subscribe to the notifications of the “Measurement” characteristic the **`subscribe()`** method of the Characteristic object must be called.

```
measurementCharacteristic.subscribe( function(error)
{
    ...
});
```

If an error occurred, then this means that subscribing writing to the “Measurement” characteristic failed and the “*bleSensorError*” event is generated.

```
if( error )
{
    bleHandler.sendBleEvent( 'bleSensorError', { sensor:sensor, error: error } );
    return;
}
```

When a notification is received for the characteristic, a “*data*” event is generated. So if there is no error, the handler must be set for this event when it is not set already, which is checked by retrieving the number of listeners via the **`listenerCount()`** method. The received data still needs to be processed and converted, which is done in the `convertSensorData()` method.

```
if( measurementCharacteristic.listenerCount('data') == 0 )
{
    measurementCharacteristic.on('data', function(data)
    {

        bleHandler.sendBleEvent
        (
            "bleSensorData",
            convertSensorData( sensor, data )
        );
    });
}
```

With this set, the sensor is now enabled which is indicated by generating the “*bleSensorEnabled*” event.

```
bleHandler.sendBleEvent( 'bleSensorEnabled', { sensor:sensor } );
```

Disable sensor

Disabling a sensor can be done by setting the "Control" characteristic accordingly. This is implemented by the **`disableSensor()`** method which takes the reference to the sensor (i.e. a Peripheral object) as the argument.

```
enableSensor( sensor )
{
    var bleHandler = this;
    ...
}
```

The first step is to retrieve the reference to the characteristics as was described in "Connect sensor".

```
var controlCharacteristic = sensor.characteristics[BLE_UUID_CONTROL];
```

In the same way as was described in "Enable sensor", the "Control" characteristic is now written to disable the operation.

```
var buffer = new Buffer(3);
buffer[0] = 0x01;
buffer[1] = SENSOR_DISABLE;
buffer[2] = 0x05;
```

The buffer can now be used as an argument when calling the **`write()`** method.

```
controlCharacteristic.write( buffer, false, function(error)
{
    ...
});
```

If an error occurred, then this means that writing to the "Control" characteristic failed and the "*bleSensorError*" event is generated.

```
if( error )
{
    bleHandler.sendBleEvent( 'bleSensorError', { sensor:sensor, error: error } );
    return;
}
```

With this set, the sensor is now disabled which is indicated by generating the "*bleSensorDisabled*" event.

```
bleHandler.sendBleEvent( 'bleSensorDisabled', { sensor:sensor } );
```

Disconnect sensor

To disconnect a sensor, the **`disconnectSensor()`** method must be called, with the reference to the sensor as the argument.

```
disconnectSensor( sensor )
{
    ...
}
```

The disconnection can be simply done by only calling the **`disconnect()`** method of the sensor since there is still an event handler set for the "*disconnect*" event of the sensor.

```
sensor.disconnect();
```

Send BLE events

To send events on the supplied interface, the **`sendBleEvents()`** method is used, which takes the name of the event and the parameters as arguments.

```
sendBleEvents( eventName, parameters )
{
    ...
}
```

When the name of the event is “bleSensorError” then this event cannot be sent right away. The reason being that the error could be the result of a function call by the user of the BLE Handler (the Sensor Server). In case of an empty event queue, emitting a new event will result in immediately calling the event handler, thereby interrupting the function flow of the caller. Therefore a “bleSensorError” is wrapped in a small timeout.

```
if( eventName == 'bleSensorError' )
{
    setTimeout( function()
    {
        bleHandler.bleEvents.emit( 'bleEvent', eventName, parameters );
    },10);
    return;
}
```

In all other cases, the call is asynchronous and the event can be simply placed in the event-queue.

```
bleHandler.bleEvents.emit( 'bleEvent', eventName, parameters );
```

Local functions

Overview

Method	Arguments	Description
convertSensorData	• sensor • data	Convert the sensor data to a proper format that can be sent as the parameters of an event.
getSynchronizedTimestamp	• sensor • data	Determine and set the synchronized timestamp of the received data.
getOrientation	• data	Returns the orientation stored in the data as a quaternion.
getSensorTimestamp	• data	Returns the sensor timestamp as stored in the data.

Convert sensor data

When the data is received, it must be convert into an object with the following properties:

Property	Type	Description
`sensor`	sensor	Reference to the sensor.
`orientation`	quaternion	Orientation of the sensor.
`systemTime`	Unsigned integer	Synchronized system time of the data in µs.
`sensorTime`	Unsigned integer	Timestamp of the data in ms expressed in the sensor time.

This is done by calling the **`convertSensorData()`** function which takes the sensor and the received data buffer as arguments.

```
function convertSensorData( sensor, data )
{
    ...
}
```

The first step is to precisely determine the current system time. This is done using the **`process.hrtime()`** JavaScript function which returns the number of seconds and nanoseconds. This is used to calculate the number of µs.

```
const hrTime = process.hrtime();
var systemtime = hrTime[0] * 1000000 + hrTime[1] / 1000;
```

This is then used to set the synchronized timestamps of the sensor by calling the setSynchronizedTimestamp().

```
setSynchronizedTimestamp( sensor, data, systemtime );
```

Next the object can be returned. The orientation is retrieved by the **`getOrientation()`** function.

```
var result =
{
    sensor:      sensor,
    orientation: getOrientation(data),
    systemTime:  sensor.systemTimestamp,
    sensorTime:  sensor:sensorTimestamp
};
return result;
```

Set synchronized timestamp

To determine and set the synchronized timestamp of the received data, the **`getSynchronizedTimestamp()`** function is called. This function takes the reference to the sensor, the received data and the system time at which the data was received as arguments.

```
function setSynchronizedTimestamp( sensor, data, systemtime )
{
    ...
}
```

The first step is to retrieve the sensor timestamp which is part of the received data. This is done by calling the `getSensorTimestamp()` function.

```
var sensorTimestamp = getSensorTimestamp( data );
```

To synchronize the system timestamp, the first step is to check whether this was the first measurement, which is the case when the systemTimestamp of the sensor is 0.. If this is the case, the system timestamp and sensor timestamp of the sensor are set and the function is ready.

```
if( sensor.systemTimestamp == 0 )
{
    sensor.systemTimestamp = systemtime;
    sensorTimestamp = sensorTimestamp;
    return;
}
```

If this is not the first measurement, the interval of the measurement must be determined and expressed in the sensor time. This must be done taking a possible rollover into account.

```
var sensorTimeDiff = sensorTimestamp - sensor.sensorTimestamp;
if( sensorTimeDiff < 0 )
{
    sensorTimeDiff += ROLLOVER;
}
sensor.sensorTimestamp = sensorTimestamp;
```

Next the time difference is used to determine the new system timestamp.

```
sensor.systemTimestamp =
    Math.min( sensor.systemTimestamp+sensorTimeDiff*(1+CLOCK_DELTA), systemtime );
```

Note that the system clock is artificially ‘sped-up’ to allow to compensate for clock drift.

Get orientation

To retrieve the orientation stored in the data, the `getOrientation()` function is called. This function reads the 32-bit floating points using the `readFloatLE()` function from the Buffer object.

```
function getOrientation(data)
{
    var w,x,y,z;

    w = data.readFloatLE(4);
    x = data.readFloatLE(8);
    y = data.readFloatLE(12);
    z = data.readFloatLE(16);

    return new Quaternion(w, x, y, z);
}
```

Get sensor timestamp

The sensor timestamp is a 4-byte unsigned integer and is extracted by calling the `getSensorTimestamp()` function.

```
function getSensorTimestamp(data)
{
    return data.readUInt32LE(0);
}
```

Writing a characteristic

To write a specific value to a characteristic its **`write()`** method must be called. The arguments for this method are given in the table below.

Argument	Type	Description
`data`	Buffer[4]	The actual bytes of data.
`withoutResponse`	Boolean	• `false`: send a write request, used with "authenticated write" characteristic property. • `true`: send a write command, used with "write" characteristic property
`callback`	error	Callback function.

Note that BLE is Little Endian[5]. This means that when adding multiple bytes values to the buffer, the corresponding write-methods must be used.

An example to store a 32-bit unsigned integer is given below.

```
var buffer = new Buffer(4);
buffer.writeUInt32LE( value);
```

When the bytes of a buffer, simply the bytes can be accessed by the index.

[4] Node.js - "How to use buffers"
[5] https://en.wikipedia.org/wiki/Endianness