

Ref.: SA_0026_LC_DES_M Version: 02	Integration Manual	 <small>SENSORIA ANALYTICS</small>
---------------------------------------	---------------------------	--

Object

This document presents the integration manual of LibCardio.

Scope of application

This document applies only to the medical device LibCardio.

Revision history

Version	Date	Changes in the record	Editor
01	22/03/2023	Creation	Mélaine GAUTIER
02	20/06/2023	Add label and frequency information	Mélaine GAUTIER

Form ref.: SA_0001_QUA_F Version: 03	<i>This document is the property of Sensoria Analytics. Any modification, copying, printing or distribution is prohibited without the agreement of Sensoria Analytics</i>	Page 1 / 41
--	---	--------------------

Table of Contents

1. Description of the LibCardio medical device	4
2. Product description.....	5
2.1. Operating principle	5
2.1.1. Product objective.....	5
2.1.2. How does the product work?.....	5
2.2. Download the library	6
2.3. Oximeter.....	6
2.3.1. via USB cable:.....	6
2.3.2. via Bluetooth	7
2.4. Instructions for use.....	9
3. Use with a PC	10
4. Oximeter configuration parameter.....	10
4.1. Berry	10
4.1.1. Bluetooth Service Information (UUIDs).....	11
4.1.2. USB-ComPort Settings	11
4.1.3. Device Packet (Device to Host):	11
4.1.4. Host Command (Host to Device).....	11
5. Integration into a C++ project.....	12
5.1. Prerequisites	12
5.2. Example 1: with BridgeCppLibCardio	13
5.2.1. Step 1: Initialization of the process.....	13
5.2.2. Step 2: Send data to the library.....	14
5.2.3. Step 3: Stop the library	15
5.2.4. Step 4: Recovering the values of a signal.....	15
5.2.5. About.....	18
5.3. Example 2: with the BridgeLibCardio gateway	19
5.3.1. Step 1: Initialization of the process.....	19
5.3.2. Step 2: Send data to the library.....	19
5.3.3. Step 3: Stop the library	20
5.3.4. Step 4 (optional): Register callback functions.....	20
5.3.5. Step 5: Recovering the values of a signal	21
5.3.6. About	23
6. Java Integration.....	23
6.1. Prerequisites.....	23
6.2. Step 1: Initialize the Java project	23
6.3. Step 2: the C++ part	24
6.4. Step 3: Link the wrapper to the java code	25
6.5. Step 4: A sample presentation	26
7. Integration on mobile application	28
7.1. ReactXP - Djinni	28
7.1.1. Data capture	29
7.1.2. The data engine in CPP.....	30
7.1.3. Displaying data during a capture.....	31

8. Connection problem of the Berry oximeter.....34

8.1. Linux (Ubuntu).....34

8.2. Windows.....35

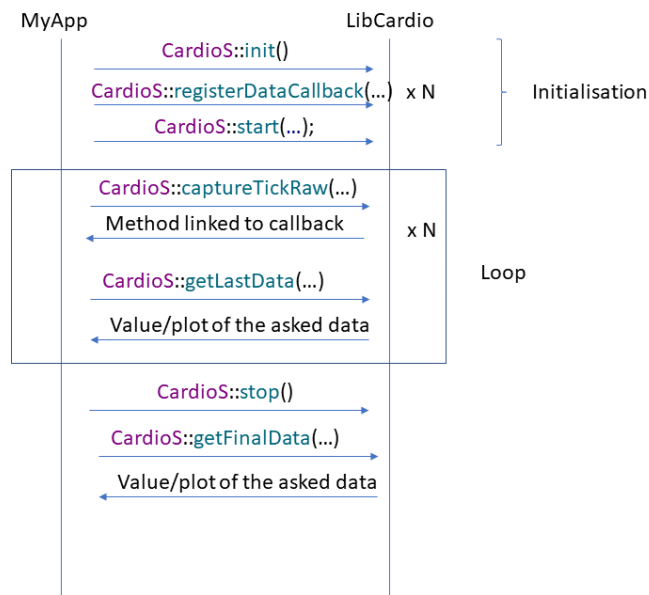
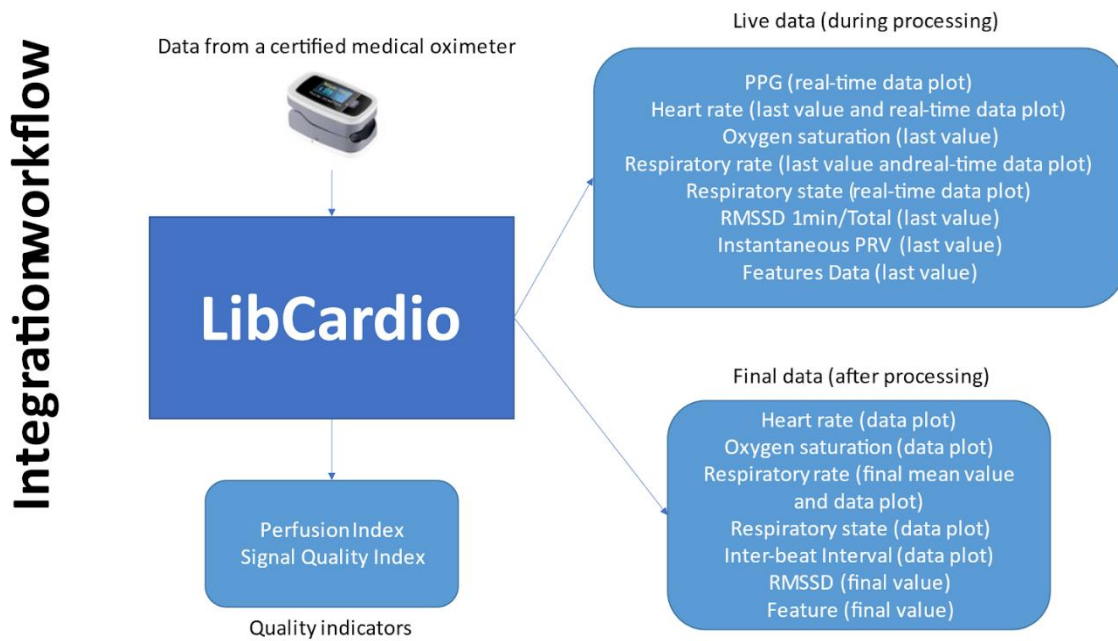
 8.2.1. Open the device manager..... 35

 8.2.2. Search for the device associated with the oximeter..... 35

9. Label..... 41

1. Description of the LibCardio medical device

This project, with its LibCardio software library, aims to significantly prevent the risk of cardio-respiratory disease by providing the most meaningful information available.



Ref.: SA_0026_LC_DES_M Version : 02	Integration Manual	 <small>SENSORIA ANALYTICS</small>
--	---------------------------	--

This manual will first present the operation of a qualified oximeter, then the implementation in C ++, Java and for mobile application.

As the library is developed in C++, it is recommended to refer to the C++ guide when writing the bridge to other platforms.

2. Product description

2.1. Operating principle

2.1.1. Product objective

The objective of LibCardio is to provide a library of simple, fast and reliable tools to be integrated into an application offering general practitioners and health professionals to detect the first signs of risk of cardio-respiratory diseases for all patients.

2.1.2. How does the product work?

The LibCardio library has input and output elements.

The input elements of the library include:

- Raw data: the fundamental data are: the photoplethysmography signal (PPG), the Heart rate (HR), the Pulse Oxygen Saturation (SpO₂), Inter-Beat Interval (IBI). These data are accepted by the library provided that they respect the configurations required (data format...) by the library;
- Data from a certified medical device (such as an oximeter, a connected watch...), tested and validated to ensure the accuracy of the results provided by the library. The connection with the library can be done by sending data packets in the required format or a Bluetooth or USB device recognized by the library.
- The input device currently used is an oximeter from the supplier Berry using a private protocol;
- An internally generated file that already has the data and configurations required by the library.

The data received by the library are then processed as they are acquired and the different algorithms are applied to generate the output data (instantaneous and at the end of the recording) among which, in addition to the input data, we have: signal quality, RMSSD, cardiac amplitude, cardiac variability, respiratory rate.

The LibCardio library is intended to be used in association with another application (middleware) which is considered as the user interface. It is in this application that the output data of the library will be interpreted according to the need. Thus, we will be able to have information such as cardiac rigidity, arterial elasticity, blood pressure, respiratory rate...

Form ref.: SA_0001_QUA_F Version: 03	<i>This document is the property of Sensoria Analytics. Any modification, copying, printing or distribution is prohibited without the agreement of Sensoria Analytics</i>	Page 5 / 41
--	---	--------------------

2.2. Download the library

The library is downloadable via the link and the password provided at the time of the purchase or the update. You will get a zip file that you will have to extract.

It will contain the ".h" file (header) and the library for the version you are using (Linux, Windows, Mac, iOS, Android, ...) as well as an example according to the integration language.

If you are still a maintenance subscriber, an email containing a new link and information about updates/evolutions will be sent to you with each new version of the library.

To know the version of the library used, chapter 5.2.5 and chapter 5.3.6 present a method/function to know the version of the current library.

2.3. Oximeter

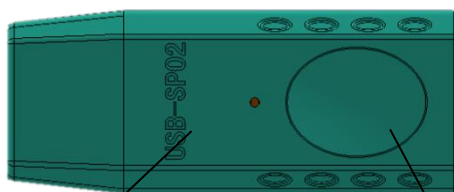
The oximeters recognized by the library are the oximeters of the Berry company which integrate the proprietary protocol SENSORIA ANALYTICS.

Other oximeters can be used by sending raw data directly according to the protocol defined by SENSORIA ANALYTICS.

They can be connected by 2 ways:

2.3.1. via USB cable:

The BM3000B is a recent pulse oximeter for measuring oxygen saturation (SpO₂) and heart rate. This product looks like a U-disk, without screen, battery and buttons. It has two connectors, one for the connection with the SpO₂ probe, the other for the connection with the PC. The necessary power is provided by PC or cell phone connected with the product. It is suitable for health care facilities, family use, as well as for physical care during sports or activities in extreme environments (You can use it before or after sports, but it is not recommended to use it during sports).



Connection Indicator

Concave shape for a good sense of touch

Figure 1

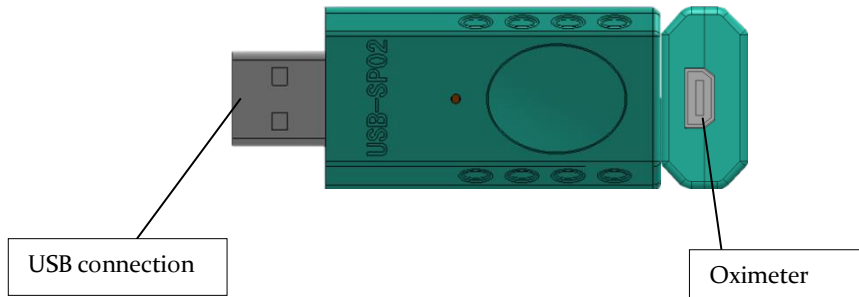
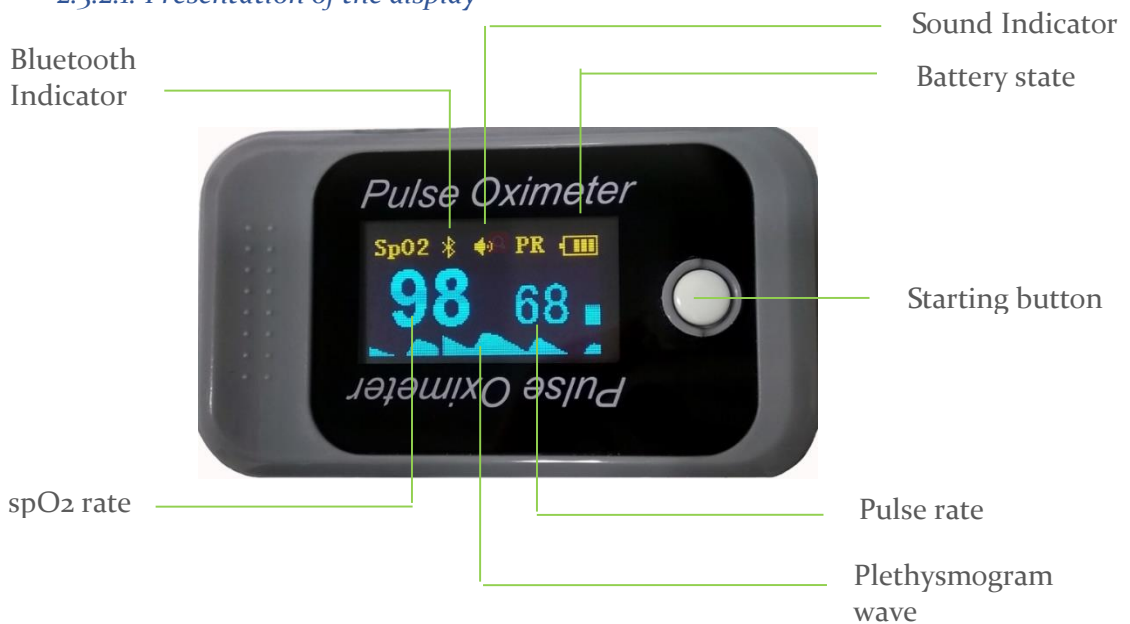


Figure 2

2.3.2. via Bluetooth

2.3.2.1. Presentation of the display




2.3.2.2. Bluetooth communication function

The BM1000D pulse oximeter is equipped with Bluetooth communication function. It can send data to a smart terminal and a computer (the associated CARDIOSENSYS software has been installed.) With a communication function through a DONGLE supplied with the oximeter.

2.3.2.3. Description of the function

- a. When the data has been displayed on the screen, press the POWER / FUNCTION button once briefly to rotate the display direction. (As shown in **Figure 12**)
- b. Then press the "POWER / FUNCTION" button again briefly to return to the previous display direction. And the ringing tone indicating that will disappear at the same time, the ringing tone will be deactivated. (As shown in **Figure 13**)
- c. Press and hold the "POWER / FUNCTION" button, the Bluetooth indication will disappear and the Bluetooth function will be disabled. (As shown in **Figure 14**)

Note: When the Bluetooth function is not successfully connected, the Bluetooth indication will flash. When the Bluetooth function is successfully connected, the Bluetooth indication will continue to be lit.

- d. When the received signal is inadequate, "  " will appear on the screen. (As shown in **Figure 15**)

- e. The product will automatically turn off when no signal is triggered for 10 seconds.



Figure 12

Figure 13



Figure 14

Figure 15

2.4. Instructions for use

1. Hold the product in one hand with the front panel facing your palm. Place the large finger of the other hand on the press sign of the battery compartment cover, press down and open the cover at the same time. The battery cabinet is open as shown in **Figure 16**.
2. Install the batteries in the slots, observing the "+" and "-" symbols, as shown in **Figure 17**.
3. Press the Clip button in Figure 1 and open the clip. Leave the test subject's finger in the rubber pads of the clip, make sure the finger is in the correct position, as shown in **Figure 18**, and then clip the finger.
4. Press the power and function button on the front panel to turn on the product. Use the first finger, middle or ring finger to perform the test. Do not hang up the finger and keep the test subject at the register during the process. The readings will be displayed on the OLED screen a moment later, as shown in **Figure 19**.

Warning:

- The positive and negative electrodes of the batteries must be installed correctly. Otherwise, the unit will be damaged.
- When installing or removing the batteries, please follow the proper sequence of operations. Otherwise, the battery compartment will be damaged.
- If the pulse oximeter is not used for a long time, please remove the batteries from it.
- Be sure to place the product on the finger in the correct direction. The LED portion of the sensor should be on the back of the patient's hand and the photodetector portion on the inside. Be sure to insert the finger at the proper depth into the sensor so that the nail is just in front of the light emitted by the sensor.
- Do not shake the finger or leave the species being tested during the process.
- The data update period is less than 30 seconds.

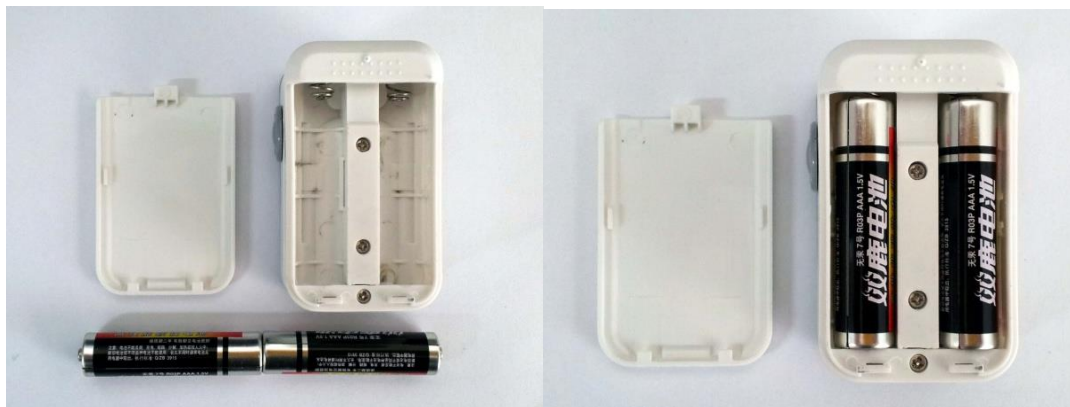


Figure 16

Figure 17



Figure 18



Figure 19

3. Use with a PC

1. Install the data manager software and the software driver on the PC. (You can obtain the software and driver from Sensoria Analytics)
2. Connect the USB-SPO₂ device and the SpO₂ sensor (or the Bluetooth dongle in case of using the Bluetooth oximeter), and then plug it into the PC. (As shown in Figure 11)



Figure 11

3. Put your finger in the oximeter and you can launch the verification test or your application.

4. Oximeter configuration parameter

4.1. Berry

4.1.1. Bluetooth Service Information (UUIDs)

Comm Service: 49535343-FE7D-4AE5-8FA9-9FAFD205E455
 Send Characteristic: 49535343-1E4D-4BD9-BA61-23C647249616
 Receive Characteristic: 49535343-8841-43F4-A8D4-ECBE34729BB3
 Rename Characteristic: 00005343-0000-1000-8000-00805F9B34FB
 MAC Address Characteristic: 00005344-0000-1000-8000-00805F9B34FB

NOTE: "Rename Characteristic" and "MAC Address Characteristic" are not supported by classic Bluetooth (e.g. BT 3.0 and below).

4.1.2. USB-ComPort Settings

Baudrate: 115200
 Bits: 8
 Stopbit: 1
 Paritybit: none

4.1.3. Device Packet (Device to Host):

Packet length: 20 bytes

Packet frequency: Adjustable (Default is 100Hz but our recommendation is to set to 200Hz)

Packet Format:

Byte0: 0xff Head mark1
 Byte1: 0xaa Head mark2
 Byte2: PktIndex Packet index
 ...
 Byte19: Checksum Checksum of the whole packet

PacketFreq: Current packets sending frequency

valid values: 1, 50, 100, 200, means 1Hz, 50Hz, 100Hz, 200Hz

Checksum Range (0~255)

Formula: Checksum = (Byte0+Byte1+...+Byte18) % 256, (% is mod operator)

4.1.4. Host Command (Host to Device)

Command length: 1 byte

Command type:

- oxfo ----- 50 Hz Package rate
- oxfi ----- 100 Hz Package rate (Default)
- oxf2 ----- 200 Hz Package rate
- oxf3 ----- 1 Hz Package rate
- oxf4 ----- ADCSample is Original sampling waveform
- oxf5 ----- ADCSample is Filtered sampling waveform
- oxf6 ----- Stop sending packet
- oxff ----- Get Software Version
- oxfe ----- Get Hardware Version
- oxfd ----- Get Bluetooth Version (Optional)
- oxfc ----- Get Unique ID (Optional)

For example:

Software Version Package Example (20 bytes):

oxff 0xaa 0x53 0x56 0x31 0x2e 0x30 0x34 0x2e 0x30 0x2e 0x33 0x36 0x00 0x00 0x00 0x00 0x00
 0x3a
 oxff 0xaa ----- Packet head mark
 0x53 ----- ASCII 'S', mark of software version
 0x56 0x31 0x2e 0x30 0x34 0x2e 0x30 0x30 0x2e 0x33 0x36 ----- ASCII string 'V1.04.00.36',
 current software version content
 0x00 0x00 0x00 0x00 0x00 ----- Padding bytes of packet
 0x3a ----- Checksum byte

Hardware Version Package Example:

oxff 0xaa 0x48 0x56 0x32 0x2e 0x30 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xd7
 oxff 0xaa ----- Packet head mark
 0x48 ----- ASCII 'H', mark of hardware version
 0x56 0x32 0x2e 0x30 ----- ASCII string 'V2.0', current hardware version content
 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 ----- Padding bytes of packet
 0xd7 ----- Checksum byte

Unique ID Package Example: (Assumed that unique id is "0x02020089541e1c17")

oxff 0xaa 0x55 0x30 0x32 0x30 0x32 0x30 0x38 0x39 0x35 0x34 0x31 0x65 0x31 0x63 0x31 0x37
 0x8e
 oxff 0xaa ----- Packet head mark
 0x55 ----- ASCII 'U', mark of unique id
 0x30 0x32 0x30 0x32 0x30 0x30 0x38 0x39 0x35 0x34 0x31 0x65 0x31 0x63 0x31 0x37 ----- ASCII
 string '02020089541e1c17', unique id content
 0x8e ----- Checksum byte

5. Integration into a C++ project

5.1. Prerequisites

- A C++ compiler

- o For windows: Cygwin or MinGW 64 to compile C++ code

The accepted frequencies are 50, 100 or 200Hz (recommended).

5.2. Example 1: with BridgeCppLibCardio

This is the easiest way to integrate the library with very general functions

5.2.1. Step 1: Initialization of the process

All boxes in this section are extracted from the *bridgeCppLibCardio.h* include file

First, you need to provide general information if the default values are not suitable for you.

```
enum ConfigItem{
    SRC=0, // source if not raw data
    PACKAGE_RATE, // int default 200
    CORE_RATE, // int default 500
    NB_CORE, // int min 1, max 8, default 4
};
void setConfiguration(const ConfigItem &item, const std::string &value);
std::string getConfiguration(const ConfigItem &item) const;

enum ProfileItem{
    GENDER=0, // 'm', 'w' or 'o'
    AGE, // age (int)
    HEIGHT, // height(int in cm)
    WEIGHT, // weight (int)
    SMOKERLEVEL, // smokerLevel('l', 'm' or 'h')
    PHYSICALACTIVITY // sportLevel('l', 'm' or 'h')
};
void setPatientData(const ProfileItem &item, const std::string &value);
std::string getPatientData(const ProfileItem &item) const;
```

The configuration allows you to specify the path/source if you use a file or a connected oximeter. You can also modify the packet frequency (by default 200Hz) and the internal frequency (not recommended). Finally, depending on your computer, you may wish to use more or less cores for your application (by default, 4 cores are used).

The patient's information can be used to refine certain results according to the profiles.

The default values are :

- For the gender 'o' (other)

- For smoking level 'l' (low)
- For activity level 'm' (medium)

The other information has no default value and it is recommended to define it.

Once this step is done, we can initialize the library.

```
enum DeviceType{
    BERRY=0,
    NONIN,
    FILE, // Read a file by following file time
    FILE_FAST, // Read a file but on one go
    RAW
};
enum Version{
    V1_4=0, // OBSOLETE
    V2_0, // OBSOLETE
    V2_02
};
// For other than the raw connection - you need to specify setConfiguration(SRC, path/source);
bool init(DeviceType type, Version v=V2_02);
```

The init function will put the library in a state of waiting for data after (re)initializing the internal parameters. **This method must be called before calling the functions that will be presented below.**

It should also be recalled after a change of parameters or patient profile or before a new measurement to reset all internal parameters.

5.2.2. Step 2: Send data to the library

```
bool tick();
bool tick(const std::vector<uint8_t>& rawData);
bool tick(const std::vector<std::string>& rawData, bool isDecimal=false);
```

The method tick(...) sends the data to the library:

- tick() will be used with all device types except raw data
- tick(...) allow to send raw data in hexadecimal or decimal format to the library (depending on the method used).

Once the data is received by the library, the actual time values are available.

5.2.3. Step 3: Stop the library

It is mandatory to stop the library before quitting the application because otherwise it will continue to run in the background and consume memory unnecessarily.

In addition, some results are only available after calling the stop function.

To stop the process, call the method :

```
bool stop();
```

5.2.4. Step 4: Recovering the values of a signal

5.2.4.1. Real time signals

Different functions allow to recover many parameters during the recording of a measurement:

- isFinger to know if the finger is well detected
- getLorenz allows you to have the instantaneous variability of the pulse rate.
- get(DataType) is used to retrieve the instantaneous values. If a value is not available -1 will be returned. Here is the list of items available in real time:
 - o SA_ARRHYTHMIA: indicator of a potential arrhythmia - value available after 30 seconds then the value changes every 20 seconds
 - o SA_BATTERY: if the device requires batteries, the charge status is indicated
 - o SA_HR: heart rate - value available after about 1 second, value is updated every second
 - o SA_PI: perfusion indicator (general blood flow) - may only provide a value after a few milliseconds
 - o SA_RMSSD: calculation of cardiac variability - available just after the first values of SA_HR, the value is updated every second
 - o SA_RR: breathing rate - value available after 30 seconds then the value changes every 3 seconds
 - o SA_SPO2: oxygen saturation level - value available after about 1 second, the value is updated every second
 - o SA_SQL: signal quality indicator - takes a few seconds to get the first value
 - o SA_TIME: gives the current time
- getPoints(DataType) retrieves a list of snapshot points formatted according to the following pattern (time, value). If a value is not available an empty vector will be returned. Here is the list of items available in real time:
 - o SA_PPG_RAW: PPG raw signal from the source - may only provide a value after a few milliseconds.

- o SA_PPG_FIL: PPG filtered signal) - may provide a value only after a few milliseconds.

```

enum DataType{
    SA_PI=0, /// Perfusion - Index value between 0 and 20 ///
    SA_SQI, /// Signal Quality Index - value between 0 and 100 ///
    SA_HR, /// Heart Rate - value between 0 and 220 - get & getPoints ///
    SA_SPO2, /// Oxygen Saturation - value between 0 and 100 - get & getPoints ///
    SA_RMSSD, /// Heart Variability of the total measurement - value between 0 and 400 ///
    SA_BATTERY, /// Current Battery level when using Raw Oximeter input - value between 0 and
100//
    SA_HR_AMP, /// heart amplitude information - 1st value between 0 and 100 -(end) get &
getPoints & getVector ///
    SA_PULSE_VAR, /// Pulse rate information - couple of value between -400 and 400 and list of
couple - getLorenz & getPoints ///
    SA_RR, /// Respiratory rate information - value between 0 and 45 - get & getPoints ///
    SA_RR_STATUS, /// Respiratory rate status information - couple of coordinate (SPO2 & RR) -
getPoints ///
    SA_STRESS_CUR, /// Current stress information - value between 0.7 and 2.5 and 0 and 1 - (end)
get & getPoints (only one element in the list val, percentage) ///
    SA_STRESS_PARA, /// parasympathic stress information - value between -5 and 5 and 0 and 1 -
(end) get & getPoints (only one element in the list val, percentage) ///
    SA_STRESS_ORTH, /// orthosympathic stress information - value between -5 and 5 and 0 and 1 -
(end) get & getPoints (only one element in the list val, percentage) ///
    SA_IBI, /// Interbeat interval - couple (unfiltered, filtered IBI) - getPoints ///
    SA_HRV, /// Heart Rate Variability information - list of 50 values - getVector ///
    SA_AE, /// Arterial Elasticity - Type and then full information - get & getVector ///
    SA_PPG_RAW, /// PPG signal from oximeter - getPoints only live ///
    SA_PPG_FIL, /// PPG signal from oximeter filtred - getPoints only live ///
    SA_ARRHYTHMIA, /// Arrhythmia evaluation - get (live & end ) & getPoints (end) ///
    SA_TIME, /// Current time - get (live)
};
// Getter - to call after a tick
// To call to get the current value of the data
bool isFinger() const; // if false need to stop recording as finger is not detected
std::pair<double,double> getLorenz() const; // -1000, -1000 if invalid value (value should be
between -400 and 400

double get(DataType type) const; // -1 if invalid value or type
std::vector<std::pair<double,double>> getPoints(DataType type) const;

```


5.2.4.2. Final signals

Once the acquisition is finished, additional values are available and mean and median values are calculated:

- get(DataType) is used to get the values. If a value is not available -1 will be returned. Here is the list of available elements:
 - o SA_AE : type of arterial elasticity (0=A, 5=F)
 - o SA_ARRHYTHMIA : final value of the indicator of a potential arrhythmia
 - o SA_HR: median heart rate value
 - o SA_HR_AMP: percentage of good amplitude
 - o SA_PI: average value of the perfusion indicator (general blood circulation)
 - o SA_RMSSD : final value recalculated after filtering the cardiac variability
 - o SA_RR : median value of the respiratory rate
 - o SA_SPO₂: median value of oxygen saturation level
 - o SA_SQI : final value of the signal quality indicator
 - o SA_STRESS_CUR: current stress level
 - o SA_STRESS_PARA: parasympathetic stress level
 - o SA_STRESS_ORTH: (ortho)sympathetic stress level
- getPoints(DataType) retrieves a list of points. If a value is not available an empty vector will be returned. Here is the list of items available in real time:
 - o SA_ARRHYTHMIA: list of arrhythmia values identified in the format (time, value)
 - o SA_HR: list of heart rate values in (time, value) format
 - o SA_HR_AMP: 3 pairs of values (good/average/bad amplitude, percentage)
 - o SA_IBI: list of intervals between beats in the form of pairs (raw value, filtered value)
 - o SA_PULSE_VAR: list of point pairs for the instantaneous variability of the pulse rate (does not work if you are doing the quick read of a file)
 - o SA_SPO₂ : list of oxygen saturation values in the format (time, value)
 - o SA_STRESS_CUR : torque (value, percentage of stress) for the current stress
 - o SA_STRESS_PARA : couple (value, percentage of stress) for parasympathetic stress
 - o SA_STRESS_ORTH: torque (value, percentage of stress) for (ortho)sympathetic stress
 - o SA_RR : list of respiratory rhythm values in (time, value) format
 - o SA_RR_STATUS : list of value pairs (SPO₂, RR)
- getVector(DataType) retrieves a list of values
 - o SA_AE : set of parameters related to the pulse wave (the first value is the identified type)
 - o SA_HR_AMP : set of amplitudes identified on the signal
 - o SA_HRV: set of parameters calculated for heart rate variability

```

enum DataType{
    SA_PI=0, /// Perfusion - Index value between 0 and 20 ///
    SA_SQI, /// Signal Quality Index - value between 0 and 100 ///
    SA_HR, /// Heart Rate - value between 0 and 220 - get & getPoints ///
    SA_SPO2, /// Oxygen Saturation - value between 0 and 100 - get & getPoints ///
    SA_RMSSD, /// Heart Variability of the total measurement - value between 0 and 400 ///
    SA_BATTERY, /// Current Battery level when using Raw Oxymeter input - value between 0 and
100//
    SA_HR_AMP, /// heart amplitude information - 1st value between 0 and 100 -(end) get &
getPoints & getVector ///
    SA_PULSE_VAR, /// Pulse rate information - couple of value between -400 and 400 and list of
couple - getLorenz & getPoints ///
    SA_RR, /// Respiratory rate information - value between 0 and 45 - get & getPoints ///
    SA_RR_STATUS, /// Respiratory rate status information - couple of coordinate (SPO2 & RR) -
getPoints ///
    SA_STRESS_CUR, /// Current stress information - value between 0.7 and 2.5 and 0 and 1 - (end)
get & getPoints (only one element in the list val, percentage) ///
    SA_STRESS_PARA, /// parasympathic stress information - value between -5 and 5 and 0 and 1 -
(end) get & getPoints (only one element in the list val, percentage) ///
    SA_STRESS_ORTH, /// orthosympathic stress information - value between -5 and 5 and 0 and 1 -
(end) get & getPoints (only one element in the list val, percentage) ///
    SA_IBI, /// Interbeat interval - couple (unfiltered, filtered IBI) - getPoints ///
    SA_HRV, /// Heart Rate Variability information - list of 50 values - getVector ///
    SA_AE, /// Arterial Elasticity - Type and then full information - get & getVector ///
    SA_PPG_RAW, /// PPG signal from oximeter - getPoints only live ///
    SA_PPG_FIL, /// PPG signal from oximeter filtred - getPoints only live ///
    SA_ARRHYTHMIA, /// Arrhythmia evaluation - get (live & end ) & getPoints (end) ///
    SA_TIME, /// Current time - get (live)
};
// Getter - to call after a stop

double get(DataType type) const; // -1 if invalid value or type
std::vector<std::pair<double,double>> getPoints(DataType type) const;
std::vector<double> getVector(DataType type) const;

```

5.2.5. About

To get the information about the library, the "about" method is available.

```

enum About{
    VERSION=0,
    COMPANY,
    UDI,

```



```
/// \return false if the tick failed without throwing an exception
///
bool tick(bool isFastReading=false);

///
/// To be called to retrieve the last packets from raw mode.
/// \param rawData All the captured data since the last tick in hexadecimal.
/// \return false if the tick failed without throwing an exception
/// Note data are managed by group of 20
///
bool tick(const std::vector<uint8_t>& rawData);

///
/// To be called to retrieve the last packets from raw mode.
/// \param rawData All the captured data since the last tick - could be in hexadecimal or decimal.
/// \return false if the tick failed without throwing an exception
///
bool tick(const std::vector<std::string>& rawData, bool isDecimal = false);

///
/// To be called to retrieve the last packets from raw mode.
/// \param rawData All the captured data since the last tick in hexadecimal.
/// \param size : Number of captured Data.
/// \return false if the tick failed without throwing an exception
///
bool tick(uint8_t* rawData, int size);
```

The method tick(...) sends the data to the library:

- tick(bool) will be used with all types of devices except raw data (if we read a file, we can take advantage of the fast read mode which will not give access to the real time values but will allow a fast calculation of the final values).
- tick(...) allow to send raw data in hexadecimal or decimal format to the library (depending on the method used).

Once the data is received by the library, the actual time values are available.

5.3.3. Step 3: Stop the library

It is mandatory to stop the library before quitting the application because otherwise it will continue to run in the background and consume memory unnecessarily.

To stop the process, you have to call the `stop()` method;

5.3.4. Step 4 (optional): Register callback functions

To access to the real-time graph display data, we can use the `LibCardio::registerDataCallback(...)` method which allows to subscribe to a notification. As soon as a value is available, the registered function is called. This method is to be inserted between the call to the `LibCardio::init()` and `LibCardio::start()` functions.

```
// static method which will be called by the callback
static void newRawPlotValues(LibCardio::LiveDataValue ldv)
{
    QVector<double> x(ldv.raw_plot.size);
    ::memcpy(x.data(), ldv.raw_plot.x, ldv.raw_plot.size*sizeof(double));
    QVector<double> y(ldv.raw_plot.size);
    ::memcpy(y.data(), ldv.raw_plot.y, ldv.raw_plot.size*sizeof(double));
    free(ldv.raw_plot.x);
    free(ldv.raw_plot.y);

    // do something now with the retrieved data
};

....
// Insert between the init and the start the registerDataCallback
void Backend::start(...)
{
    LibCardio::init(...);
    LibCardio::registerDataCallback(LibCardio::LIVE_RAW_PLOT,&newRawPlotValues);
    LibCardio::start(200, 500);
}
```

5.3.5. Step 5: Recovering the values of a signal

5.3.5.1. Real time signals

To retrieve the last available values, use the `LibCardio::getLastData(...)` method.

Here is an example of the recovery of a real time value, the perfusion index:

```
LibCardio::LiveDataValue ldv;
int pi=0;
if(LibCardio::getLastData(LibCardio::LIVE_PI, &ldv))
{
    pi = ldv.pi;
}
```

If the value you want to retrieve is a plot value, you have to be careful with memory leaks. The library allocates a space with the data that you will have to free manually when they are not used anymore.

The following example shows how to do this:

```
LibCardio::LiveDataValue ldv;  
double ppg_fil = 0;  
double ppg_fil_time = 0;  
  
if(LibCardio::getLastData(LibCardio::LIVE_FIL_PLOT, &ldv)){  
    ppg_fil = *ldv.fil_plot.y;  
    ppg_fil_time = *ldv.fil_plot.x;  
  
    free(ldv.fil_plot.y);  
    free(ldv.fil_plot.x);  
}
```

5.3.5.2. Final signals

Once the acquisition is finished, 2 additional types of data are available:

- Global data: these are the data acquired throughout the acquisition for a parameter (for example, heart rate)
- Data that is only accessible at the end of the process

They are accessible by calling the method `LibCardio::getFinalData(...)`.

This method works exactly like the previous method with the need to free up memory for the point lists.

```
LibCardio::FinalDataValue fdv;  
double *hr_x = 0; // time  
double *hr_y = 0; // hr values  
  
if(LibCardio::getLastData(LibCardio::FINAL_HR_PLOT, &fdv)){  
    hr_x = fdv.hr.x;  
    hr_y = fdv.hr.y;  
    for(int i=0; i<val.ibi.size; i++){  
        std::cout<<"HR["<<i<<"] = "<<hr_x[i]<< , "<<  
            hr_x[i]<<std::endl;  
    }  
  
    free(hr_x);  
    free(hr_y);  
}
```

5.3.6. About

To get the information about the library, the "about" function is available.

```
enum About{  
    VERSION=0,  
    COMPANY,  
    UDI,  
    RELEASE_DATE  
};  
// About  
std::string about(const About &item);
```

6. Java Integration

You will find here an example of integration in JAVA using Eclipse. There are other ways to integrate the C++ library into a JAVA project that are not described here.

6.1. Prerequisites

- Eclipse IDE with JDT and CDT plugins
- JDK for JAVA compilation
- C++ Compiler
 - For windows: Cygwin or MinGW 64 to compile C++ Code (install only one of the compilers to avoid dependency problems)

6.2. Step 1: Initialize the Java project

1. Start Eclipse IDE
2. New Java project
3. Select the name of the project:
 1. It will be named "**HelloWorld_Project**" for this example.
 2. Select **Next**, then **Finish** to create the project.
4. In the Package Explorer extend the project "**HelloWorld_Project**
5. Right click on the src folder and select **New > Package**.
6. Enter the name as **com.sensoriaanalytics.jni** and click **Finish**.
7. In the Package Explorer under the "**HelloWorld_Project>src**" right click
8. Enter the name as "**HelloWorld**" and click on **Finish**.
9. The class must be defined as the structure of the following code.

```
1
2 //Package name
3 package com.sensoriaanalytics.jni;
4
5 //Class definition
6 class HelloWorld {
7
8     //In this part you define the native methods that are going to be imported from the C++ wrapper
9     public native void sayHello();
10
11     //We load the shared library here
12     static {
13         //You should only put the name of the library not the path to it
14         System.loadLibrary("HelloWorld");
15     }
16
17     public static void main(String[] args) {
18         HelloWorld h = new HelloWorld();
19         //Call the native methods
20         h.sayHello();
21     }
22
23 }
```

10. Go to the Eclipse IDE menu and select "Run > External Tools > External Configurations".
11. Select the program from the list in the left panel.
12. Press the new button.
13. Enter the name as "**javah - C Header and Stub File Generator**".
14. Find the location of javah.exe: browse to locate it in the JDK installation folder (the file should be on a path similar to c:\Program Files\Java\jdk1.7.0\bin\javah.exe)
15. Enter the working folder as "**\${project_loc}/bin/**".
16. Enter the arguments as "**-jni \${java_type_name}**" then **apply**
17. Now, from the Common tab, select the checkbox next to **External Tools** under View in the Favorites menu, then **apply and close**
18. **Deselect** Build automatically from the project menu
19. In the **Package Explorer**, right click on "**HelloWorld_Project**" and select "**Build Project**".
20. In the package explorer, highlight **HelloWorld.java** and select "**Run > External Tools > 1 javah - C Header and Stub File Generator**"
21. This step will generate the header file for the C++ code **com_ sensoriaanalytics _jni_HelloWorld.h** placed in the bin folder of the Java project "**HelloWorld_Project**

6.3. Step 2: the C++ part

1. From the menu select **File>New>Project** then expand C/C++
2. Highlight the C++ project and click **Next**
3. Under **the project type**, expand the **shared library** and highlight the **empty project**
4. Under **Toolchains** select **Cygwin GCC (or MinGW GCC)**
5. Enter the name of the project as **C_HelloWorld** then press **Next**

6. Uncheck **Debug**. Leave only the **Release** selected, then click on **End**.
7. If you are asked to switch from Perspective to C/C++, **check and click Yes**
8. In the **Explorer project**, click on **C_HelloWorld** and select the file **New>Source**
9. Enter the **Source file com_sensoriaanalytics_jni_HelloWorld.cpp** and **finish**.
10. Now, write the C++ wrapper following this structure:

```

1 //Must include jni.h and the generated header from the java code
2 #include <jni.h>
3 #include "..\HelloWorld_Project\bin\com_sensoriaanalytics_jni_HelloWorld.h"
4 #include <stdio.h>
5
6
7 //Follow this format to create native methods
8 JNIEXPORT void JNICALL Java_com_sensoriaanalytics_jni_HelloWorld_sayHello(JNIEnv *env, jobject obj){
9     printf("Hello world!\n");
10    return;
11 }

```

11. In the Explorer project set to **C_HelloWorld**, then press **Alt +Enter** to open the properties for the project
12. Open **C/C++ Build**, highlight **Settings**, select **Includes** from the list in the Tool Settings tab on the right side.
13. **Click Add, File System**, then browse to the folder **include** in the JDK installation module and click **OK**, (should be something like **C:\Program Files\Java\jdk1.7.0**).
14. Repeat 13 but this time include the file **include \win32**
15. **Highlight Miscellaneous under Cygwin C Linker** (or **MinGW C Linker**) from the list
16. On the right side, enter the Linker flags as **-mno-cygwin -Wl,--add-stdcall-alias** if you are using Cygwin, otherwise put: **-Wl,--add-stdcall-alias**
17. **Highlight libraries under the C++ Linker**
18. Add a library (-I) : put the name of the dll file (without the .dll extension)
19. Add a library path (-L): set the directory path where the dll and .h files are located.
20. **Highlight the environment under C/C++ Build**
21. Edit the path variable by adding at the end **"; (path to the DLL and .h file) "**.
22. Open **C/C++ General**, then **Paths and Symbols**, **GNU C++ in Includes** tab. Click and **Add** button. Add directory path panel appears, then provide the path to **the dll + header files**.
23. Switch to the **Artefact Build tab** under **C/C++ Build**
24. Delete the text for the artifact name and type **in HelloWorld**
25. Delete the text for the output prefix and keep it empty
26. **Right click C_HelloWorld** in Project Explorer and select **Build Project**

6.4. Step 3: Link the wrapper to the java code

1. Switch to Java Perspective by selecting in the **Window>OpenPerspective>Other** menu, select **Java (default)** and click **OK**

2. From the menu, select Run Settings>Run...
3. From the list in the left pane highlight **HelloWorld** under **Java application**
4. On the right side, switch to **the Arguments tab**
5. Enter **the VM arguments**
like **-Djava.library.path="{workspace_loc}\C_HelloWorldRelease "**
6. In the explorer package **highlight HelloWorld_Project**
7. Select the application **Run>Run As>Java**

6.5. Step 4: A sample presentation

The code provided in the mail file contains 2 folders, one for the java part and the other for the native code part.

The native code contains a file that is the JNI wrapper and a separate folder that contains the libCardio library.

There are some methods that should be called to initialize the library and they are grouped in this method:

```
//A method that must be called first and only once to initiate the main functionalities of the lib
JNIEXPORT void JNICALL Java_com_lithiumhead_jni_HelloWorld_InitCardioS(JNIEnv *env, jobject obj, jint duration){
    int native_duration = (int)duration;

    printf("Initializing CardioS!\n");
    //Initialize the library
    CardioS::init();
    //Define the protocol
    CardioS::SerialProtocol a = CardioS::CARDIOS_PROTOCOL;
    //Start the raw data capture
    CardioS::startRaw(200, 500, native_duration, a);
    printf("Finished Initialization\n");

    return;
}
```

And to call it from Java, the following method is used:

```
//Define the native methods
public native void InitCardioS(int duration);
```

Where the duration is the duration of the recording.

In the java part, the whole input file is read, then it is sent segment by segment to the wrapper (parts of 10 lines to provide the library with enough data to calculate the parameters).

```
float[] results = h.callCardiosRaw(buffered_data);
```

The buffered data is actually an array that contains the bytes read from the file. And the type of adaptation occurs in the side of the package:

```
//A method that takes the elements of the buffer(String array) then transforms them into uint8_t elements and pass it to cardioS lib then prints the live data
JNIEXPORT jfloatArray JNICALL Java_com_lithiumhead_jni>HelloWorld_callCardiosRaw(JNIEnv *env, jobject obj, jobjectArray stringArray){
    int length = env->GetArrayLength(stringArray);

    std::vector<uint8_t> bufferUInt8 = {};
    const char* cpp_string;

    //Read the elements of the input array one by one and transforms them into uint8_t
    for (int i = 0; i < length; ++i)
    {
        //Extract the data from the input jobjectArray and put it in a jstring
        jstring jstr = (jstring) env->GetObjectArrayElement(stringArray, i);
        //Convert the jstring to a cpp native string
        cpp_string = env->GetStringUTFChars(jstr, NULL);

        //convert a const char* to an uint8_t* (Because it's the input type for the library)
        uint8_t* p = hex_str_to_uint8(cpp_string);
        bufferUInt8.push_back(*p);

        //Release the data held by the stringArray
        env->ReleaseStringUTFChars(jstr, cpp_string);
        env->DeleteLocalRef(jstr);
    }
    std::cout<<std::endl;

    //Put all the converted elements into an array
    uint8_t captureRawData[bufferUInt8.size()];
    std::copy(bufferUInt8.begin(), bufferUInt8.end(), captureRawData);
```

The data in buffered_data is transformed into a uint8_t type and then stored in a uint8_t list which is the input to libCardio.

```
//Put all the converted elements into an array
uint8_t captureRawData[bufferUInt8.size()];
std::copy(bufferUInt8.begin(), bufferUInt8.end(), captureRawData);
```

```
//Send the captured data to the library
CardioS::captureTickRaw(captureRawData, bufferUInt8.size());
```

Then, to capture the live data, we should call getlastdata and then get the value we are looking for. For example, here is how to get the ppg_filtered data:

```
CardioS::LiveDataValue dat;

if(CardioS::getLastData(CardioS::LIVE_FIL_PLOT, &dat)){
    ppg_fil = *dat.fil_plot.y;
    ppg_fil_time = *dat.fil_plot.x;

    free(dat.fil_plot.y);
    free(dat.fil_plot.x);
}
```

The Spo2:

```
if(CardioS::getLastData(CardioS::LIVE_SPO2, &dat)){  
    spo2 = dat.spo2;  
}
```

NB: spo2, ppg_fil ppg_fil_time and other extracted values are all real.

Then finally the data is stored in a jfloatArray and returned to the java side:

```
// Create an array that will hold the live data  
jfloatArray result;  
result = env->NewFloatArray(9);  
if (result == NULL) {  
    return NULL; /* out of memory error thrown */  
}  
  
//Copy the data to the output array  
std::vector<float> vec = {ppg_raw_time, ppg_raw, ppg_fil_time, ppg_fil, spo2, hr_time, hr, rr_time, rr};  
jfloat jvec[9];  
  
for(int i =0 ; i<9; i++){  
    jvec[i] = vec[i];  
}  
env->SetFloatArrayRegion(result, 0, 9, jvec);  
  
return result;
```

Then the whole process is repeated until all data is sent to the library.

NB:

- The more data is retrieved from the package the longer the computation times are, that's why a pause(10) was used between every two calls to the library on the java side knowing that JNI has some problems with successive and fast calls.

7. Integration on mobile application

We will present here the integration step using the ReactXP based application using Djinni-React-Native. There are now other more modern integration methods but we haven't tested them yet as we don't specialize in mobile - however, we can help you with integration using other methods if you need.

If another mobile development application is used, some of these steps will have to be adapted.

7.1. ReactXP - Djinni

The prerequisites to create and run an application are those of ReactXP.

The following steps will work with Android and iOS.

We use a low-level cross-platform CPP layer.

The communication with the Javascript part of ReactXP is done via a "Bridge" code that has been generated with Djinni-React-Native. The idl file and the script to generate the definition of the "Bridge" are in the generated_src folder.

If you want to connect Java and CPP, you can use Djinni with the IDL file or other tools at your convenience and expertise.

Please note that the library integration is hybrid in that before the LIBCARDIO protocol existed, we needed a basic approach (Start / Send Oximeter Data / Stop / Receive Final Data). Therefore, the application uses only a small subset of the available API surface. All the data to be displayed on the screen was available via the incoming packets.

With the LIBCARDIO protocol, PlethWave data is no longer available and must be requested from the LibCardio CPP library. Fortunately, we were already using a flexible approach to retrieve a Javascript map of pulse rate objects, so it is easy to add PlethWave data to this map.

If you want to bridge Java and CPP, you could use Djinni with the IDL file, or other tools.

Please note that the library integration is hybrid in the sense that before the existence of the LIBCARDIO protocol, we needed a basic approach (Start/Send Oximeter Data/Stop/Receive Final data). This is why the app only use a tiny subset of the API surface available. All the data required to be displayed on screen was available in the incoming packets.

7.1.1. Data capture

Once you are connected to the device whether it is USB, BLE or a replayed file :

The process used to decode the data and determine by examining the packet header.

```
// Berry protocol 1.3
// tslint:disable-next-line:no-bitwise
if (value[0] & 0xff && value[1] & 0xaa) {
} else {
    // BCI protocol 1.2
    // No checksum
```

}

The proprietary protocol integrated in LIBCARDIO is detected by a request to the oximeter. In contrast to the BCI/BRI protocols, the PlethWave data is no longer accessible as input data. The packets must be sent to the library and in return, the PlethWave data will be accessible (as presented below).

7.1.2. The data engine in CPP

The data engine needs at least 50 packets of 20 bytes of data for its algorithms. Once we have accumulated enough data, it is sent for processing. The engine returns immediately with the current respiration rate, if it has accumulated enough data to calculate it. If the LIBCARDIO protocol is used, we also retrieve the PlethWave data:

The following is the code for the CPP / Javascript "Bridge":

```
// Send the data coming from the JS side to the engine
LibCardio::captureTick(data, nb);
// Prepare for returning data to the consuming application
auto map = mBridge->createMap();
// Query for Respiration rate
LibCardio::LiveDataValue liveRR;
if (LibCardio::getLastData(LibCardio::LIVE_RR, &liveRR)) {
} else {
    liveRR.rr = 0;
}
map->putInt("rr", liveRR.rr);
// Query for Pleth Wave plot data
LibCardio::LiveDataValue liveFillPlot;
if (LibCardio::getLastData(LibCardio::LIVE_FIL_PLOT, &liveFillPlot)) {
    // sounds good
    LOG_VERBOSE("< DataBridgeImpl::provideOxymeterData returning RR "
        "[%d] FillPlot size [%d]"
        "FillPlot X [%f] FillPlot Y [%f]",
        liveRR.rr, liveFillPlot.fil_plot.size,
        *liveFillPlot.fil_plot.x, *liveFillPlot.fil_plot.y);
} else {
    liveFillPlot.fil_plot.size = 0;
    liveFillPlot.fil_plot.x = nullptr;
    liveFillPlot.fil_plot.y = nullptr;
}
```

```

}
// Set the size of the arrays returned
if (liveFillPlot.fil_plot.size != 0) {
    map->putInt("fillPlotSize", liveFillPlot.fil_plot.size);
}
// populate the arrays
auto arrayliveFillPlotX = mBridge->createArray();
auto arrayliveFillPlotY = mBridge->createArray();
if (liveFillPlot.fil_plot.x != nullptr && liveFillPlot.fil_plot.y != nullptr) {
    for (int i = 0; i < liveFillPlot.fil_plot.size; ++i) {
        arrayliveFillPlotX->pushDouble(liveFillPlot.fil_plot.x[i]);
        arrayliveFillPlotY->pushDouble(liveFillPlot.fil_plot.y[i]);
    }
}
else
{
    LOG_VERBOSE("No Data for PlethWave");
}
map->putArray("fillPlotX", arrayliveFillPlotX);
map->putArray("fillPlotY", arrayliveFillPlotY);
// Resolve the promise to return the data to the consumin app.
promise->resolveMap(map);

```

However, for the Javascript part, we have to process each received packet as presented in the next section.

7.1.3. Displaying data during a capture

Everything that is received is decoded and processed in real time. But the refresh rate of the data on the screen is fixed at 1 second. We do not use a timer to know when to switch the screen state. We use the capture rate to know when 1 second has been reached.

This means that data can be accumulated in buffers if necessary (mainly for the graphical display of PlethWave).

7.1.3.1. The PlethWave graph

The displayed data is managed by a linked list whose length is equal to the capture frequency * 5. At 200 Hz, this means that we have a maximum of 1000 points. To mimic the fact that the

Ref.: SA_0026_LC_DES_M Version : 02	Integration Manual	 SENSORIA ANALYTICS
--	---------------------------	---

graph moves to the left when the list is full, we remove the oldest value from the list and add the new one.

Remember that when you use the LibCardio protocol, the data comes from the CPP engine and not from the device.

7.1.3.2. The heart rate graph (Pulse Rate)

The displayed data is managed by a linked list whose length is equal to 60 points (1 minute of data). To mimic the fact that the graph moves to the left when the list is full, we remove the oldest value from the list and add the new one.

7.1.3.3. The Lorenz graph

The displayed data is supported by a linked list whose length is equal to 20. A maximum of 20 bubbles are displayed).

7.1.3.4. Breathing animation

This part can be based on CSS and the recommended duration is 10s.

7.1.3.5. All gauges

The value that would be displayed is updated in real time. When the state of the screen is changed, the current value is used.

7.1.3.6. Saving patient information at the end of the capture

At the end of the capture, you must:

- Send configuration data to CPP engine (subject_age / subject_height)
- Stop the CPP engine. It triggers an event to return the radar data to the application.
- The data is stored on the phone.
- The data is then sent to the AWS S3 cardiosensys-mobile compartment
- The accumulated data is then saved in a CSV file
- The CSV file is then sent to the cardiosensys-mobile AWS S3 compartment.
- Access the radar view

7.1.3.7. Display of the final radar

The radar data is a JSON fragment that contains everything we need to display the radar and each associated individual graph.

Form ref.: SA_0001_QUA_F Version: 03	<i>This document is the property of Sensoria Analytics. Any modification, copying, printing or distribution is prohibited without the agreement of Sensoria Analytics</i>	Page 32 / 41
--	---	------------------------

It is returned automatically when you stop the engine in the bridge code by triggering the NEW_RADAR_DATA event.

```
// Stop the engine (this should be done first, otherwise the following code
// would crash the app)
LibCardio::stop();
// Get radar data
LibCardio::FinalDataValue val;
// set subject age and height
setRadarAge(mSubjectAge);
setRadarHeight(mSubjectHeight);
bool retval = LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_HR, &val);
LOG_VERBOSE("LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_HR, &val) "
"returned [%s]",
retval ? "true" : "false");
setRadarHR(val.hr.x, val.hr.y, val.hr.size);
retval = LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_RR, &val);
LOG_VERBOSE("LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_RR, &val) "
"returned [%s]",
retval ? "true" : "false");
setRadarRR(val.rr.x, val.rr.y, val.rr.size);
retval = LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_SPO2, &val);
LOG_VERBOSE("LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_SPO2, "
"&val) returned [%s]",
retval ? "true" : "false");
setRadarO2(val.rro2.x, val.rro2.y, val.rro2.size);
retval = LibCardio::getFinalData(LibCardio::FINAL_RRO2, &val);
LOG_VERBOSE("LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_SPO2, "
"&val) returned [%s]",
retval ? "true" : "false");
setRadarRRO2(val.rro2.x, val.rro2.y, val.rro2.size);
retval = LibCardio::getFinalData(LibCardio::FINAL_RMSSD, &val);
LOG_VERBOSE("LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_RMSSD, "
"&val) returned [%s]",
retval ? "true" : "false");
setRMSSD(val.rmssd);
retval = LibCardio::getFinalData(LibCardio::FINAL_FEATURES, &val);
LOG_VERBOSE("LibCardio::getFinalData(LibCardio::FinalDataType::FINAL_FEATURES, "
"&val) returned [%s]",
retval ? "true" : "false");
setRadarEAType(val.features.angle, val.features.cVal, val.features.bVal,
val.features.eaType);
setRadarIRA(val.features.ppgai, val.features.tsDic);
```

```
setRadarTA(val.features.pai);  
// conversion: RadarData -> json  
json j = radarData_;  
std::string stringified = j.dump();  
LOG_VERBOSE("RadarData JSON string is [%s]", stringified.c_str());  
// Emit an event to the UI  
auto ret = mBridge->createMap();  
ret->putString("radarData", stringified);  
mBridge->emitEventWithMap(DataBridge::NEW_RADAR_DATA, ret);  
promise->resolveInt(o);
```

8. Connection problem of the Berry oximeter

8.1. Linux (Ubuntu)

Most of the time the problem comes from the access rights to the driver.

(you must have administrator rights to perform these manipulations presented in the rest of this document)

The best way to solve this problem is to determine the group the device belongs to and add your account to it.

It is assumed that the device connects to ttyACMo

```
# exploring the problem  
ls -l /dev/ttyACMo  
man ls  
# problem solving  
sudo adduser $USER $(stat --format="%G" /dev/ttyACMo )
```

This solution solves the problem for one account. If you want other accounts to connect to the device, you will have to repeat the operation for each of them.

An alternative would be to add a rule in the `/etc/udev/rules.d` directory.

You will first need to run the `dmesg` function to get the vendor information and product ID.

Then you can create a file named `99-sensoria.rules` (if there is already a rule with the number 99, you can change it by a used name number, you can also choose a different name of sensoria if it is more meaningful for your project).

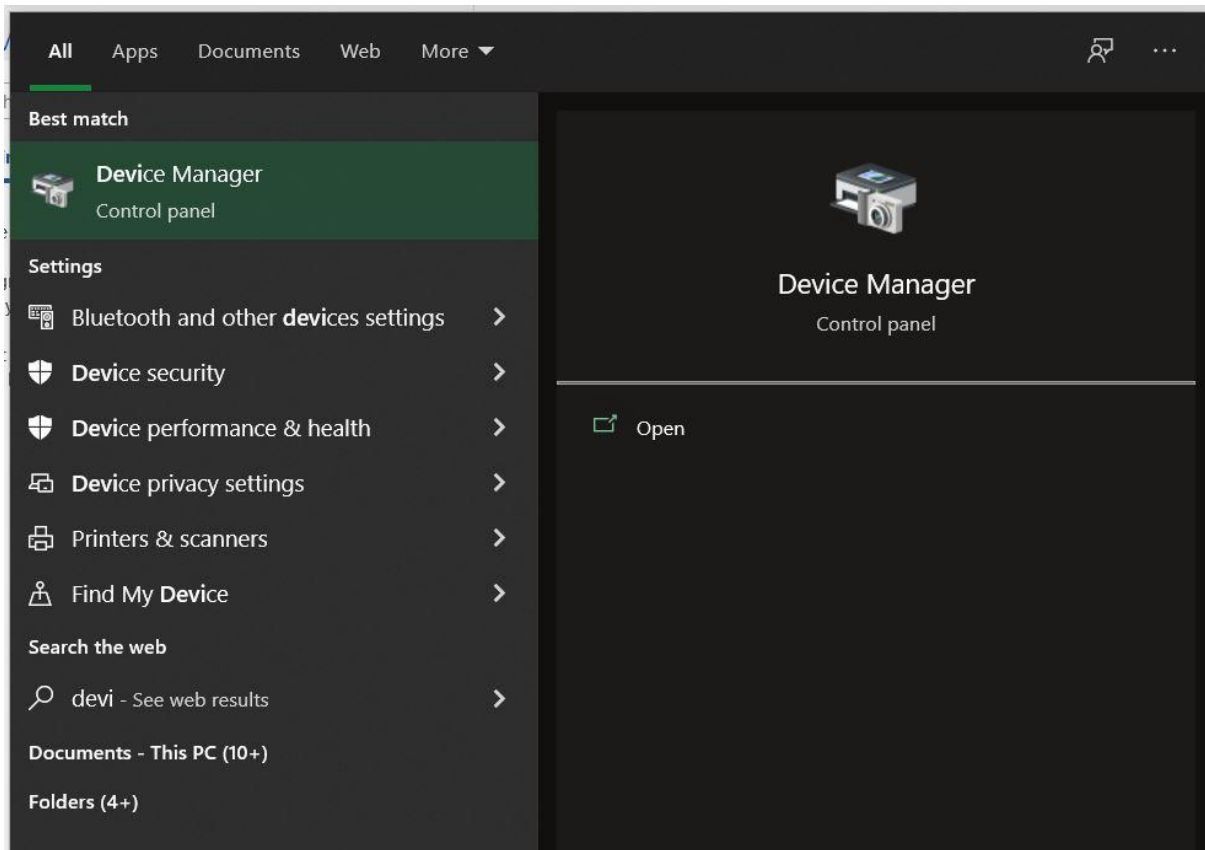
Content of the file *99-sensoria.rules*

```
# USB-Sensoria
SUBSYSTEM=="usb", ATTR{idVendor}=="XXXX", ATTR{idProduct}=="*", \
ENV{DEVTYPE}=="usb_device",MODE="o666"
```

XXXX will be replaced by the vendor's identifier

8.2. Windows

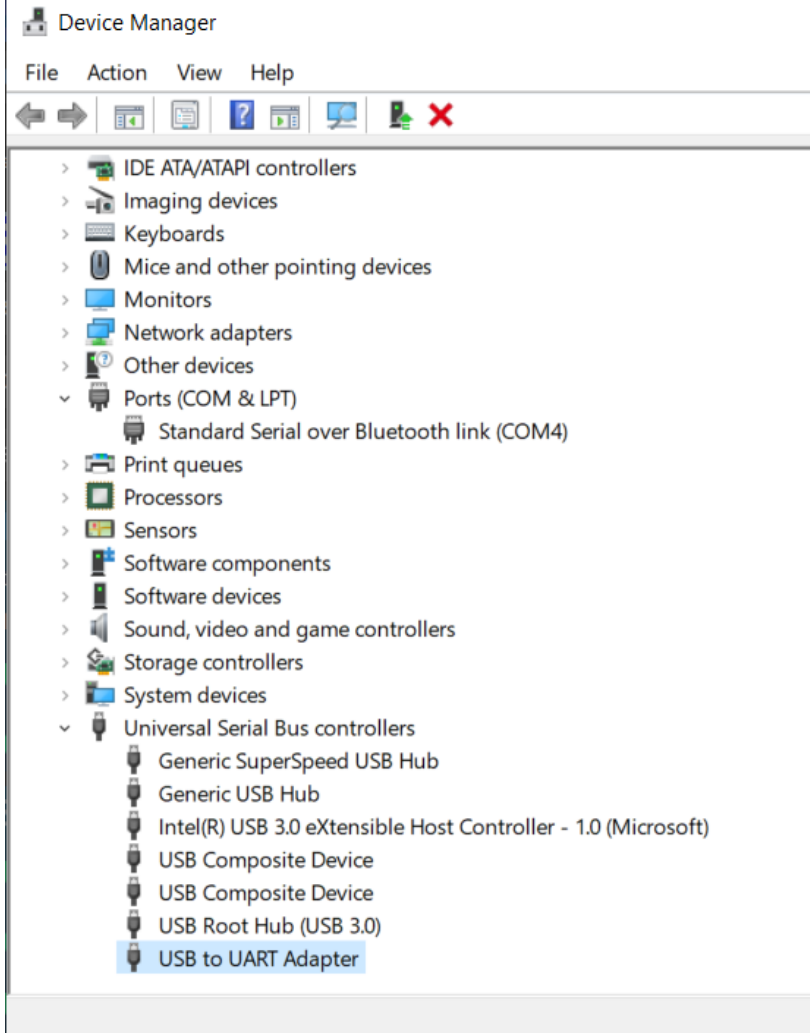
8.2.1. Open the device manager



8.2.2. Search for the device associated with the oximeter

- If it appears in USB/Universal Serial Bus controllers (look for USB to UART Adapter), you have to update the driver

Right click on the USB to UART Adapter -> Property

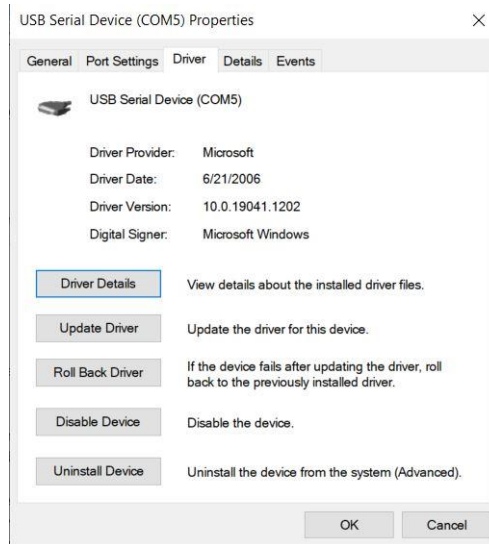



- Otherwise it should appear in Ports (COM & LPT) under the name USB Serial Device (COM X), with X a COM number. If you have multiple devices in the list disconnect and then reconnect the oximeter to identify the correct port.

- > Ordinateur
- > Périphériques de sécurité
- > Périphériques logiciels
- > Périphériques système
- ✓ Ports (COM et LPT)
 - ↳ Périphérique série USB (COM3)
- > Processeurs

Right click on the USB to COM X -> Property

- Go to the *Driver* section and click on *Update Driver*



←  Update Drivers - USB Serial Device (COM5)

How do you want to search for drivers?

→ Search automatically for drivers

Windows will search your computer for the best available driver and install it on your device.

→ Browse my computer for drivers

Locate and install a driver manually.

Cancel

- Select : *Let me pick from a list of available drivers on my computer / Laisser moi choisir à partir de la liste des drivers disponibles sur mon ordinateur.*



←  Update Drivers - USB Serial Device (COM5)

Browse for drivers on your computer

Search for drivers in this location:

Include subfolders

→ **Let me pick from a list of available drivers on my computer**

This list will show available drivers compatible with the device, and all drivers in the same category as the device.

- Select the first device in the list and click on Next



←  Update Drivers - USB Serial Device (COM5)



Select the device driver you want to install for this hardware.




Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.

Show compatible hardware

Model

-  USB Serial Device
-  USB Serial Device
-  USB to UART Adapter

 This driver is digitally signed.

[Tell me why driver signing is important](#)

Have Disk...

Next
Cancel

9. Label



 **LibCardio**
©Sensoria Analytics 

MD LibCardio 2.0.0

UDI (01)3770030728003(8012)V2001(11)230322

 22/03/2023

  contact@sensoriaanalytics.com

 +33 (0)4 22 46 24 20

 1047 route des Dolines, Business Pole
06560 Valbonne
FRANCE

 **Sensoria Analytics**

End of the document