Real-Time Analytics API Details

ReTiSense Inc IP. Strictly Confidential

Components & Integration Method

- AnalyticsLib.h: ANSI-C Header File
- libStrideAnalytics.a: fat library containing 32-bit, 64-bit and watchOS build for library

Integration into Objective-C is simply through including the header file and adding the library to the linker list.

Integration into Swift:

Follow steps listed as in this web tutorial: https://medium.com/swift-and-ios-writing/using-a-c-library-inside-a-swift-framework-d041d7b701d9

We also provide web-assembly library for integration into Ionic-React hybrid development environment, or for development of web-applications that run on Google Chrome or Microsoft Edge.

Usage Process

1. When starting analytics, call the **InitUser** function with the user's details. This needs to be done **once**, at the beginning of the application. This initializes all appropriate data objects and sets the Analytics usage context. If no specific user info is collected by the application, simply pass 0 to that argument. Analytics will use appropriate defaults. While other arguments are self-explanatory and can be understood easily through header file, these two need some clarification:
   a. no_of_devices: 2 if both insoles are to be used.
   b. isShod: 1 if the person is using insoles inside shoes; 0 if using outside (as a foot pressure plate)
   c. reportPressure: Set this to 1 if you need pressure, and not load (force). Pressure is reported in KG/cm2
   d. footsize: Provide foot length (tip of toe to end of heel) in cm (again, foot size. Not shoe size). This is used for accurate calculation of weight in different foot regions.

2. After connecting to the insoles in the application, query the "Hardware Revision String" parameter for the device. Before enabling notification, ensure appropriate sensing spec is set up. Since the Analytics library is built to handle multiple versions & SKUs of hardware, we need to communicate the spec for this specific sensor hardware. This needs to be done once, before BLE notification is enabled:
   a. If Hardware Version String is `STRD_PPP_01`: call :
      ```
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 24, 1);
      ```
   b. If Hardware Version String is `STRD_PRS_01_24` or `STRD_PRS_02_24`: call :
      ```
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 23, 1);
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 23, 0);
      ```
   c. If Hardware Version String is `STRD_PRS_01_27` or `STRD_PRS_02_27`: call :
      ```
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 22, 1);
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 22, 0);
      ```
   d. If Hardware Version String is `STRD_PRS_11_27`: call :
      ```
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 30, 1);
      SetSensorSpecNew (0,0,0,0,0,0,0,0, 3, 30, 0);
      ```

3. Call **ProcessStride_FSR_grid** (or for ForcePlate, **ProcessStride_FSR_bb_grid**) for each frame of BLE data that needs to be processed. This function will process the BLE raw data, and populate the respective data structures that will contain the real-time processed values.
   a. BLE raw data needs to be passed as an array of **unsigned char** (iOS) or **byte** (Java/Android)
      i. iOS generally stores the bytes from BLE devices as NSData, which can be passed to this function through [NSData bytes] or [NSData getBytes], and the array length will be about 120. Android stores BLE raw data as ASCII string of the BLE raw data, and the array length will be 240. The "length" argument communicates to the Analytics whether this is from iOS or Android, and it will process the data appropriately (including converting ASCII to relevant data in case of Android).
   b. **speed_m_s** is speed in meters per sec**,** **d_m** is distance in meters, and **alt_m** is altitude from sea level in meters. In applications where it is not relevant, 0 can be passed for these arguments.
   c. **isLeft** is to denote whether we are currently processing left foot data or right foot. Pass 1 if this is left foot, 0 if right foot. For ForcePlate, the variable should be passed as 1 always.
   d. **mode** denotes the type of activity we are analyzing. For any non-standard application, this should be set to 0. For run/walk, this should be set to 1.
   e. **isReset** is provided by the application to demarcate an end of single "micro-analysis period", and collect data for that micro-analysis period for eventual averaging or totals. For example, in walking, this would be when a step has ended, or in case of cycling when a full rotation is complete. If not intended by application, this can always remain as 0.

    f. **noConstrain** is to turn off automated pattern-intelligence processing by analytics. Simply speaking, by default, if analytics notices unusual or unreasonable values all of a sudden (could happen spuriously for various reasons), it decides to throw that value and use established pattern to make that decision. Setting this to 1 will turn this behavior off, and will allow spurious high-variation values to be accepted.
    g. In case mode is set to 1 (run/walk), non-zero return value is obtained when a step completes, and 0 is obtained intra-step. In case mode is set to anything other than 1, return value of this function is 0 in case of bad inputs or any other bad calculations. It is recommended that application only use calculated results when the return value is 1.

4. Once analysis has been completed, and it needs to be restarted by clearing off all internal states, call **ResetStrideInfo**.
    a. **ResetStrideInfo** does NOT clear up user info, so UserInfo does not need to be initialized again.
    b. **ResetStrideInfo** takes argument isShod, which can set/reset the mode of usage (whether with shoes or without), in case user changed the mode.

Data Architecture for the sensor raw data

PRISM is packed with sensors, and it is important to understand the sensor data that is available through PRISM API.

PRISM has upto 90 pressure sensors, and two sets of 6-axis motion sensors (accelerometer and gyro) units per insole. The pressure sensor numbers vary by size of the insoles. The diagram below is for the 28.5cm insoles. The load/pressure values are available in the data structure **struct MatrixLoadInfo**, in **load_inst** array, with the indices for specific sensors as per the diagram below. The MatrixLoadInfo structure for the LEFT can be obtained by calling **GetMatrixLoadInfo** with argument **isLeft** as 1, and for RIGHT, by calling this function with **isLeft** as 0.

load_inst is a 2-dimensional array of integer numbers; the 1st (row) index corresponds to the rows, of which there are 20, and start at the BOTTOM of the insole. The 2nd (column) index corresponds to columns, of which there are 9, and start at the outer point of the heel for the LEFT and inner point of the heel for RIGHT.

The values in load_inst are stored as integers, and the unit is “grams”. This enables fast-integer computations to happen within the API library as well as in application, and then the data can be used by the application by converting to KGs as appropriate.

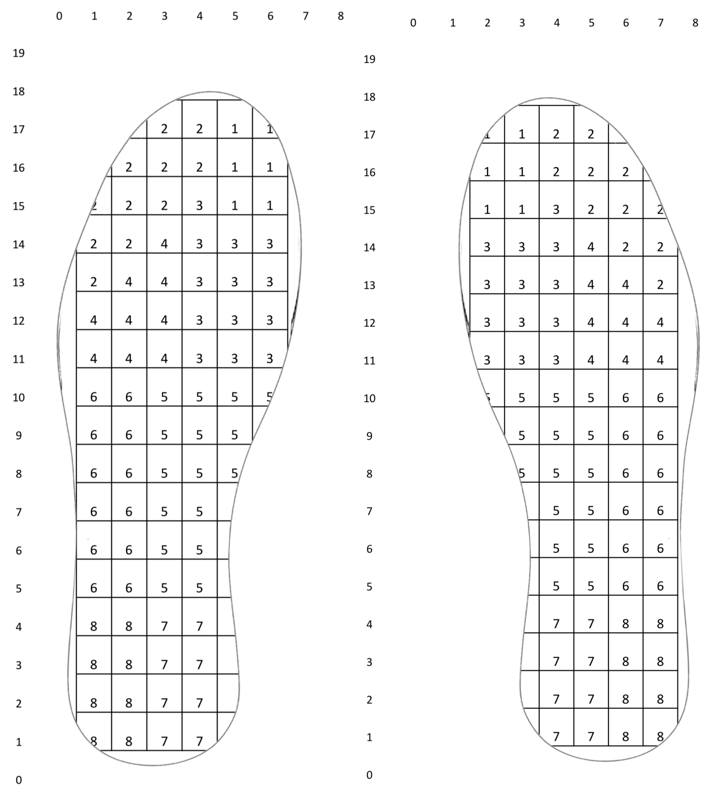


Fig. 1. Sensor positions and mapping to discrete regions

One accelerometer is located in the insole arch area, to provide info about the motion and rotation of the bottom of the foot, and one is located in the strap, to provide motion and rotation info for the topside of the foot.

Accessors for processed values

In iOS Objective-C, all accessors return the pointer to the **struct** object specific to that accessor. Analytics does not prevent mishandling of the object by the application, and in that sense, care must be taken to ensure application does not corrupt the **struct** objects.

The Analytics library also coalesces the matrix values into discrete values for easy consumption – through a mapping of matrix sensor regions to specific foot regions, in the following manner:

- 1: Big Toe/Hallux
- 2: Smaller toes
- 3: Metatarsal 1 & 2
- 4: Metatarsal 4-5
- 5: Inner (Medial) arch
- 6: Outer (Lateral) arch

- 7: Inner Heel
- 8: Outer Heel

In addition, all the existing APIs for INSIGHT are also available:

- **GetUserInfo** returns the **struct UserInfo** pointer, where user data is stored.
- **GetRunInfo** returns the **struct RunInfo** pointer, where run info is stored (not relevant for applications other than walking/running).
- **GetNewStrideInfo** returns the **struct NewStrideInfo** pointer, which always stores the CURRENT stride information. On next call to **ProcessStride_FSR**, all the fields in this struct will get updated. Any persistent storage of the data if needed, needs to be handled by the application by copying the values into it's own data objects.
- **GetStressInfo** returns the **struct StressInfo** pointer, which always stores the CURRENT stress information. On next call to **ProcessStride_FSR**, all the fields in this struct will get updated. Any persistent storage of the data if needed, needs to be handled by the application by copying the values into it's own data objects. The weight/force measured shall reflect in this data object. For INSIGHT insoles, "stress" simply refers to Load (measured in KGs).
    - **hallux** will point to Big Toe
    - **front** will point to metatarsal 1,2
    - **front_2** points to metatarsal 3,4,5
    - **mid** points to outside of the midfoot
    - **arch** points to the inside arch of the foot
    - **heel** points to the heel area of the foot. In 8-position insoles, this will be lateral (outer) heel. In 6-position, this is at the center of the heel.
    - **heel2** points to the medial (inner) heel in 8-position insoles. In 6-position insoles, this will simply be equal to "heel".
        - To get overall heel loading, heel and heel2 values must be added.
        - To get pressure on heel in 6-position insoles, heel2 can be ignored.
    - **toes** points to the smaller toes; more specifically the base of the 4th toe.
    - **lr_*** refers to Loading Rate (KG/s).
    - **peak_*** refers to peak values obtained during the analysis.
- **GetRevNumber** returns the revision number of the analytics library
- **SetDebugMode** sets printing of certain debug message in the library, if an argument of 1 is provided.

Advanced Gait Analysis parameters

Stridalyzer Analytics library calculated many gait parameters, that can be easily accessed. The analytics library maintains gait parameters for:

- Real-Time step and gait details (basic) - e.g. stride rate, gct, airtime, stepcount.

- Avg/Total step and gait details (basic) – e.g. avg. stride rate during the entire analysis, avg, gct, total stepcount, etc
- Real-Time and avg. gait phase details – e.g. Time-to-full-contact, Heel-lift-time, Toe-off-time, etc.
- Real-Time, avg, and peak load values on a per-step basis, or for the entire duration of the static loading analysis.

For the above, use the **GetStride_*** and **GetStress_*** functions. "**isCurr**" needs to be set to 1 for real-time instantaneous values, and to 0 for avg. values. "**isLeft**" needs to be set to 1 for LEFT, and 0 for RIGHT.

Center-of-Balance

Center-of-Balance in itself is a very important gait parameter, for both static gait and dynamic gait. For PRISM, center-of-balance calculation is available between both feet (what we would call as a "Global Gait Context"), as well as for each foot (what we would call as a "Local Gait Context"). Having local as well as global gait context data enables much more accurate calculations of fall-risk, physiotherapy rehabilitation, etc.

Use the **GetMatrixInfo_COB** function to get the local gait context Center-of-balance data.

- **isCurr** needs to be set to 1 for real-time (in case of static analysis that means instantaneous, and in case of dynamic analysis that means using the data of the last full step), and 0 for average (for the entire analysis period).
- **isLeft** needs to be set to 1 for LEFT foot, and 0 for RIGHT foot.
- **isRow** needs to be set to 1 to return the horizontal component, and 0 to return the vertical component. Which means to get both components, this function needs to be called twice with isRow as 1 once, and then 0 once.

The data returned is the cell-index of the Sensor-Matrix that is estimated to be the Center-of-Balance. The row value would be between 0-19, and column value would be between 0-8. The values can be mapped to obtain the relative position in the foot contact space. An example in Fig. 2 demonstrates the LEFT insole COB at (2,6) and for RIGHT insole at (4,11).

Use the **GetRunInfo_COB** function to get the global gait context Center-of-balance data.

- **isCurr** needs to be set to 1 for real-time (in case of static analysis that means instantaneous, and in case of dynamic analysis that means using the data of the last full step), and 0 for average (for the entire analysis period).
- **isRow** needs to be set to 1 to return the horizontal component, and 0 to return the vertical component. Which means to get both components, this function needs to be called twice with isRow as 1 once, and then 0 once.

- The data returned is between -100 to 100 for both horizontal and vertical. The values can be mapped to obtain the relative position in the foot contact space. An example point of (10, -20) is demonstrated below.

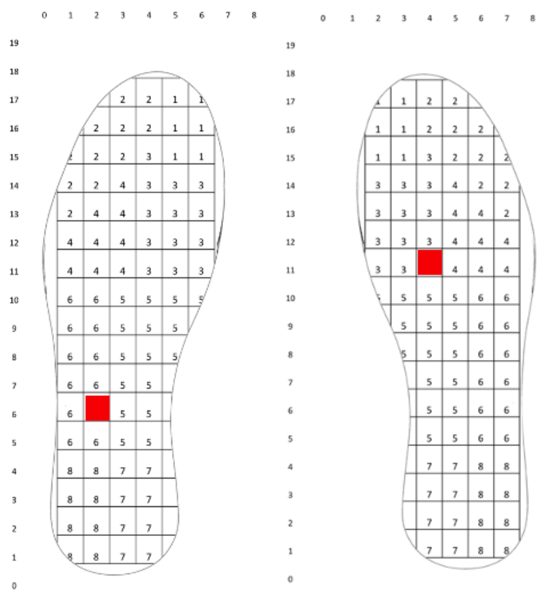


Fig. 2. Local Gait Context Center-of-balance data

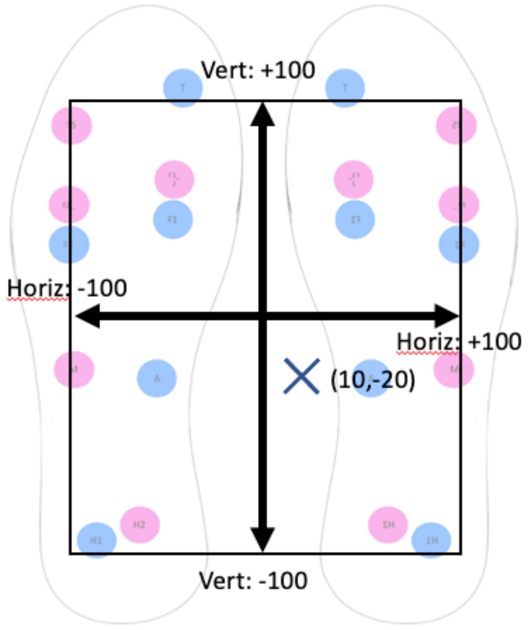


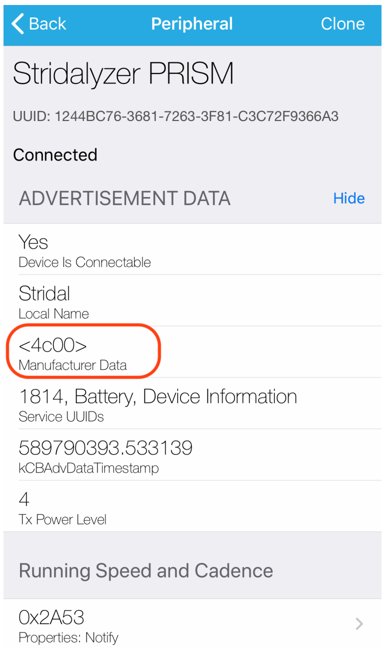
Fig. 3. Global Gait Context Center-of-balance data

Setting up the Insoles

Stridalyzer PRISM Insoles connect over BLE 4.2 protocol, and send data to the application. Here's the process to connect to the insoles and read data:

1. Set up BlueTooth LE Protocol and connection manager.
2. Scan for device with substring "Stridalyzer" in its name, and supporting service UUID 1814.
3. Connect, and discover services and characteristics.
4. The insoles send out advertising data which mark the LEFT and RIGHT specifically. Your app can read these advertising packets BEFORE connecting, so you can detect which insole this is, even before connecting. To ensure LEFT and RIGHT devices are recognized appropriately by the app, use the following logic:
    a. Read the advertising data.
    b. Check for the "Manufacturer data" field. It should be a 1 or 2 charater field depending on the OS.

c. The first character will be 'L' (ASCII value 4c) for the LEFT insole, and 'R' (ASCII value 52) for RIGHT insole.


Back Peripheral Clone

Stridalyzer PRISM

UUID: 1244BC76-3681-7263-3F81-C3C72F9366A3

Connected

ADVERTISEMENT DATA Hide

Yes
Device Is Connectable

Stridal
Local Name

<4c00>
Manufacturer Data

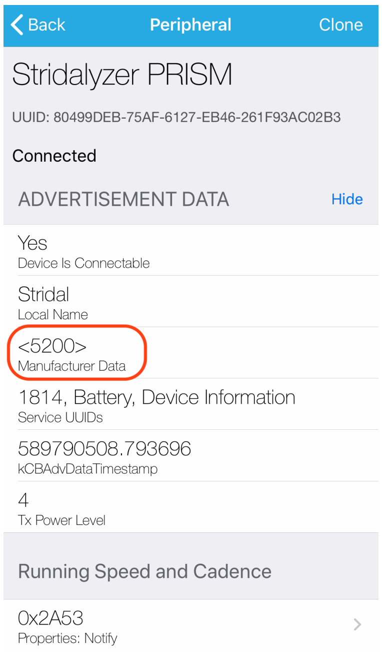
1814, Battery, Device Information
Service UUIDs

589790393.533139
kCBAdvDataTimestamp

4
Tx Power Level

Running Speed and Cadence

0x2A53
Properties: Notify


Back Peripheral Clone

Stridalyzer PRISM

UUID: 80499DEB-75AF-6127-EB46-261F93AC02B3

Connected

ADVERTISEMENT DATA Hide

Yes
Device Is Connectable

Stridal
Local Name

<5200>
Manufacturer Data

1814, Battery, Device Information
Service UUIDs

589790508.793696
kCBAdvDataTimestamp

4
Tx Power Level

Running Speed and Cadence

0x2A53
Properties: Notify

LEFT RIGHT

5. Battery data notification can be enabled at this point if application wants to show battery level.
6. When application is ready to receive and process relevant biomechanics data, set up notification for characteristic 2A53 on all connected devices.
7. For Stridalyzer PRISM default data rate is 20Hz. Device data rate is adjustable through software, through the following steps:

The characteristic **FF02** in Service UUID **1814** can be written to, to program the sampling rate.

The command for changing sampling rate is **0x000B**, and needs to be programmed to bytes 0 & 1
The opcode is data sampling period in milliseconds (represented in HEX), and should be programmed in byte 3.

An example of sending this command is below:

```
int period = 1000/dataRateInHz;
unsigned char data[3] = {[0 ... 2] = 0x00};
data[1] = 0x0B;
data[2] = (unsigned char)(period&0xFF);
```

Then you can send the char array **data** to the characteristic FF02.

8. If device disconnects for some reason, and reconnects, make sure to re-enable notification.

NOTE: Stridalyzer devices do NOT implement AES or pairing-based security. Any client can connect to the devices.

LightBlue app on iOS and BLEMonitor App on Android are good debugging apps to be able to connect to the devices and understand various supported characteristics and parameters.

Support & Help

For reporting bugs and errors, or for seeking assistance for APIs, please send an email to support@retisense.com. ReTiSense Team commits to acknowledge and respond within 24 hours, and provide an assessment of severity and potential ETA.

Terms and Conditions

This document, the API header source file and the compiled libraries are provided pursuant to the Non-Disclosure agreement. Any efforts to reverse engineer or decompile shall be considered a violation of that agreement.