Simple Type-Length-Value Pair Protocol over Bluetooth

1. **Limitations**

* **Max Packet Size <= 43 bytes**

The Max Packet Size can be adjusted slightly depends on various devices and their hardware. Actually, we can negotiate the actual packet size during the establishing period of the connection, but, in the initial version of the protocol, we use 43 bytes as a constant.

* **No connection information offered from Low level:**

There is guarantee that request between endpoints can be paired with each other. In short, the communication between client and server is half-duplex.

1. **Protocol Stack**

The table below demonstrates the protocol stack of the STLV protocol over Bluetooth used in Kreyos watch. The stack is setup from bottom to top:

|  |  |
| --- | --- |
| **File Transferring** | A special application command used to transfer even larger data blocks like file. |
| **Application Commands** | The counterpart application layer of IOS in STLV protocol. In this layer, application commands are defined as element types and their sub-types. |
| **Simple TLV Layer** | This layer introduces a high efficient framework to represent the application data structure. |
| **Simple Transport Layer** | To exceed the limitation of lower level Bluetooth, this layer is invented to help transferring data block larger than 43 bytes. |
| **Bluetooth RFCOMM** | This layer is a wrapper to all lower Bluetooth stack. It isolates all the hardware and air protocol from the application usage. |

**Protocol Payload**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Simple Transport Head** | **Simple TLV Head** | **Application Command 0** | **Application Command 1** | **…** | **Application Command n** |

1. **Bluetooth Transport Packet Layout**

|  |  |  |
| --- | --- | --- |
| **Simple Transport Head** | | **Payload** |
| **Flag (1 Byte)** | **Length (1 Byte)** | **Data (Length Bytes)** |

* **Flag:**

0 ~ there is continuous transport units.

1 ~ there is continuous transport units.

* **Length:**

Integer indicate the length of the Data in bytes.

1. **Simple Type-Length-Value Packet Layout**

**Overall**

|  |  |  |  |
| --- | --- | --- | --- |
| **Simple Type-Length-Value Head** | | | |
| **Version (1 Byte)** | **Flag (1 Byte)** | **Length(1 Byte)** | **Reserved(1 Byte)** |
| **Body (Length Bytes)** | | | |

**Simple Type-Length-Value Header Detail**

|  |  |  |  |
| --- | --- | --- | --- |
| **Spec** | | | **Comments** |
| Byte 0 | Version | | Should be 1 |
| Byte 1 | Bit 0 | Request Flag | Should be ignored in current version |
| Bit 1 | Response Flag |
| Bit 2 ~ 7 | Reserved |
| Byte 2 | Length of the body | |  |
| Byte 3 | Sequence Number | | Should be ignored in current version |

**Body Detail**

|  |  |
| --- | --- |
| **Element 0** | **Variable Length** |
| **Element 1** | **Variable Length** |
| **…** | **…** |
| **Element n** | **Variable Length** |

**Element Detail**

|  |  |  |
| --- | --- | --- |
| **Byte 0** | **Bit 0** | **Flag indicates that if the element is extent element type** |
|  | **Bit 1 ~ 7** | **Element Type** |
| **Byte 1** | **Element Type** | **If Byte 0 Bit 0 == 1** |
|  | **Element Length** | **If Byte 0 Bit 0 == 0** |
| **Byte 2** | **Element Type** | **If Byte 0 Bit 0 == 1 && Byte 1 Bit 0 == 1** |
|  | **Element Length** | **If Byte 0 Bit 0 == 1 && Byte 1 Bit 0 == 0** |
|  | **Element Data** | **Otherwise** |
| **Byte 3** | **Element Type** | **If Byte 0 Bit 0 == 1 && Byte 1 Bit 0 == 1 && Byte 2 Bit 0 == 1** |
|  | **Element Length** | **If Byte 0 Bit 0 == 1 && Byte 1 Bit 0 == 1 && Byte 2 Bit 0 == 0** |
|  | **Element Data** | **Otherwise** |
| **…** | | |

1. **Sports Data Format**

The data file parse has been implemented on both WP and Android Protocol Class. Give the format here to help implement the iOS version parser.

Sports Data Format is used to organize both daily activity data and sports workout data as files on the watch. One file per day. The file will be stored in watch under the folder /DATA/.

**File Name:**

The file will stored each day. File name will be the format as “YY-MM-DD”. The entire path name is “/DATA/YY-MM-DD”

**File Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| Signature(4 Bytes) | | | |
| Version (1 Byte) | Year (1 Byte) | Month (1 Byte) | Day (1 Byte) |
| Sports Data Array (Various Length) | | | |

**The Sports Data Array is composed from multiple Sports Data Row defined as below:**

|  |  |  |  |
| --- | --- | --- | --- |
| Row Mode (1 Byte) | | Hour (1 Byte) | |
| Minute (1 Byte) | | Entry Count (1 Byte) | |
| Entry Type 0 (4 bits) | Entry Type 1 (4 bits) | Entry Type 2 (4 bits) | Entry Type 3 (4 bits) |
| Entry Type 4 (4 bits) | Entry Type 5 (4 bits) | Entry Type 6 (4 bits) | Entry Type 7 (4 bits) |
| Data Entry 0 (4 byte) | | | |
|  | | | |
| Data Entry 1 (4 byte) | | | |
|  | | | |
| Data Entry 2 (4 byte) | | | |
|  | | | |
| Data Entry 3 (4 byte) | | | |
|  | | | |
| … | | | |

**Note:**

* There should be only as many as entry count valid entry types in the blue part. Entry type index over the limitation will be padded as zero. Totally, the blue part known as “Sports Data Meta” is fixed size as 4 bytes.
* There should be only as many as entry count Data Entry attached at the end of each Sports Data Row. That is, the length of Sports Data Row is a variable depending on the Entry Count.

**Row Mode:**

The row mode indicates what the sports type is during this record. Below is the available value and meaning:

|  |
| --- |
| #define DATA\_MODE\_NORMAL 0x00  #define DATA\_MODE\_RUNNING 0x01  #define DATA\_MODE\_BIKING 0x02  #define DATA\_MODE\_WALKING 0x03 |

Each workout mode has correspondent paused one defined as below:

|  |
| --- |
| #define DATA\_MODE\_RUNNING\_PAUSED 0x11  #define DATA\_MODE\_BIKING\_PAUSED 0x12  #define DATA\_MODE\_WALKING\_PAUSED 0x13 |

**Entry Type:**

The entry type indicates the meaning for correspondent Data Entry below the Meta part.

|  |
| --- |
| #define DATA\_COL\_INVALID 0x00  #define DATA\_COL\_STEP 0x01  #define DATA\_COL\_DIST 0x02  #define DATA\_COL\_CALS 0x03  #define DATA\_COL\_CADN 0x04  #define DATA\_COL\_HR 0x05 |

**Data Entry:**

Since the watch does not support float number, data can only be passed as 32 bits integer and has different meaning for each different type of the Entry Type.

|  |
| --- |
| DATA\_COL\_STEP – Steps taken  DATA\_COL\_DIST – Distance in Centimeters  DATA\_COL\_CALS – Calories Burned  DATA\_COL\_CADN - Count  DATA\_COL\_HR - Heart Beat Count |

**Android Interface:**

To simplify the parsing process, I have already implement the parser inside Android Bluetooth Module.

Once the file received, a message of Protocol.MessageID.MSG\_FILE\_RECEIVED will be raised with the parsed data structure.

The parsed result will be stored as a Java ActivityDataDoc object.

In ActivityDataObject, the member data is an ArrayList<ActivityDataRow> so that each ActivityDataRow represents a Sports Data Row defined above.

* Inside the ActivityDataRow, the ActivityDataRow.mode indicates the row mode defined above.
* The ActivityDataRow.hour and ActivityDataRow.minute are the time from 00:00.
* The ActivityDataRow.data is a SparseArray<Double>. The Key of the SparseArray is the ActivityDataRow.DataType, which is the same as Entry Type defined above, the value of the SpraseArray is correspondent data collected from watch.

Normally, the ActivityDataRow array can be treated as below table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Mode | Hour | minutes | Step | Distance | Calories | HeartRat |
|  |  |  |  |  |  |  |

1. **Bluetooth Module Architecture**

**According to the protocol stack, a typical design of the Bluetooth Module in Watch/Phone App should be like below.**

Note:

* Encoder is in responsible for STLV packet encoding and Decoder, as its counterpart, is for STLV packet parsing.
* The Reader should hold a thread waiting on low level packet event and maintains the internal status machine to help build up the STLV packet from those 43 bytes packet. In other word, Reader is used to handle transport layer.
* The Writer should also use a thread waiting on the Request Queue. It should pick request from the queue filling the raw data into simple transport packet and send them out though low level Bluetooth interface.
* Application module like UI and schedule tasks should use the STLV encoder to build requests and push the requests into Request Queue.
* Multiple Application modules can be waiting on the same type of Bluetooth Message. To approach this, they should create and register a message Listener to the Bluetooth Server. Thus, the Server can broadcast the incoming message to all those Listeners.
* Due to the version 1 do not support sequential transport packet, this is essentially due to the hardware limitation, if the application wants to send data, it should lock the request queue and push all the data connectively into the queue. This behavior can be encapsulated into the STLV protocol encoder.

1. **Android Example**

**Android Bluetooth framework is essentially an implementation of the common architecture.**



Note:

* The Bluetooth modules were split into 2 major parts – the Protocol and the BluetoothAgent.
* The BluetoothAgent will enumerate all Bluetooth endpoints from system and setup Bluetooth socket to the particular one.
* The Reader thread will listen on the socket and be wake up if there incoming a packet. The status of transport packets handling is also located inside the Reader.
* The Writer is also helping process transport layer. It is waiting on the Request Queue and pick up STLV packet from it. Later on, the Client split larger STLV packet into smaller transport packets with the flag set to proper value and send them out.
* Protocol will parse and compose STLV packet after the packet is rebuilt from transport packet or based on the application unit’s request.

1. **Interaction among Android Application Units and Bluetooth Modules**

Activities and KreyosService can send out a Bluetooth Command to the device by simply 2 steps:

1. Create or reference a Protocol instance.
2. Call the proper member of the instance.

Meanwhile, Activities and KreyosService can pick up a Bluetooth Command from Watch by just registering the message listener to the BluetoothAgent.

BluetoothAgent is a singleton.

We now have 4 major categories of Bluetooth commands:

Configurations

* To write the configurations to the watch.
* To read the configurations from watch.

Activity

* To sync activity data when user is running or biking.

Notifications

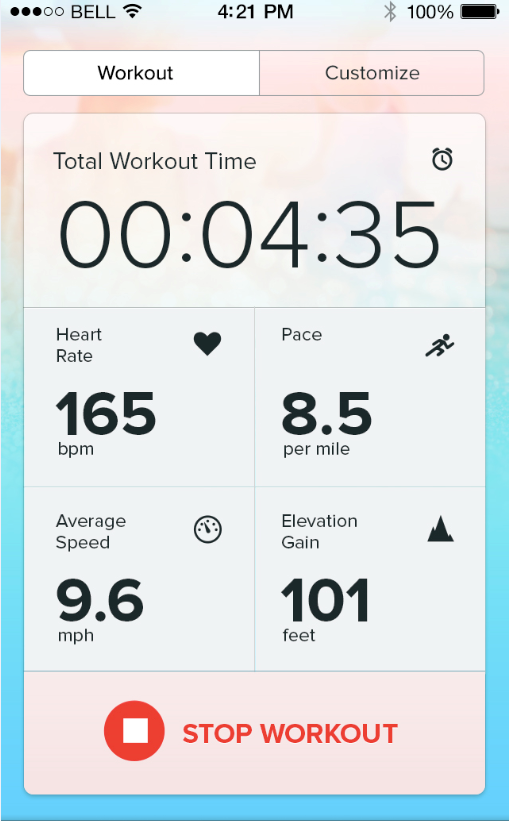
* To send a notification to the watch.

File

* To read historic activity data from watch.

**UI and Bluetooth Module Interaction Detail**

1. **Sports Page:**



* 1. **Workout**

In SportsActivity, handleBTData() will update local copy of current active workout data from watch. displaySportsData() is for display the runtime data from device.

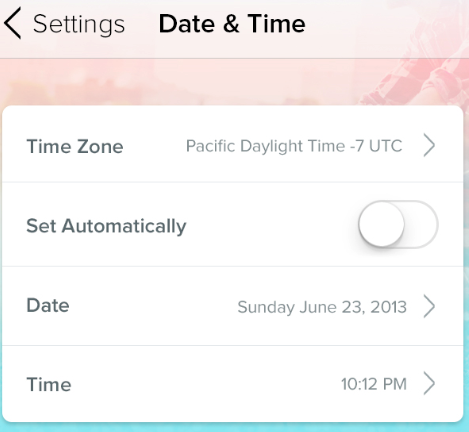
* 1. **Customize Watch Grid**

Also in SportsActivity, once the watch grid configured, the grids information will be stored in the global SharedPreferences. Meanwhile, it will also call

KreyosActivity. writeWatchUIConfig() to write this configuration to the watch.

1. **More Page:**
   1. **Date & Time**

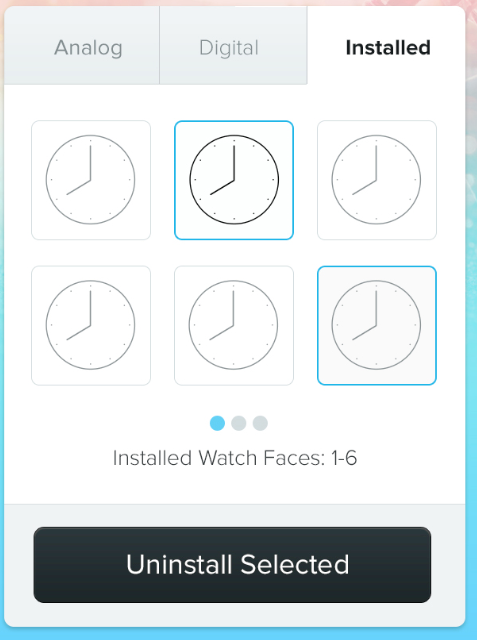
In DateTimeActivity, the textViewSync’s onClick event will trigger clock sync from phone to watch.



* 1. **Watch Faces**

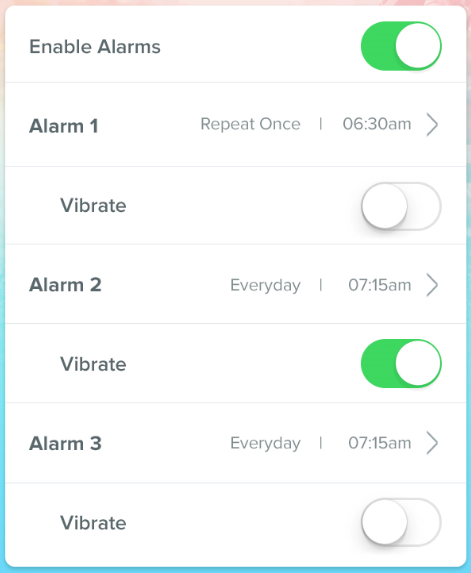
In WatchFaceActivity, if user select one uninstalled gadget, then click the bottom button, the onClick event of R.id.btnUploadWatchFace will trigger a Protocol.sendFile() action via WatchFaceActivity.installGadget() to download the Gadget file to watch. Similarly, WatchFaceActivity.uninstallGadget() will call Protocol.removeFile() to uninstall a particular gadget.

The WatchFaceActivity should load snapshot of all gadget from some online service, this is right now a temporary URL set at the begin of the class, and update the gadget status via loadInstalledWatchFaces(). The function will call Protocol.listFile() to enumerate all gadget files. The returned file list will be received in WatchFaceActivity.handleBTMessage();



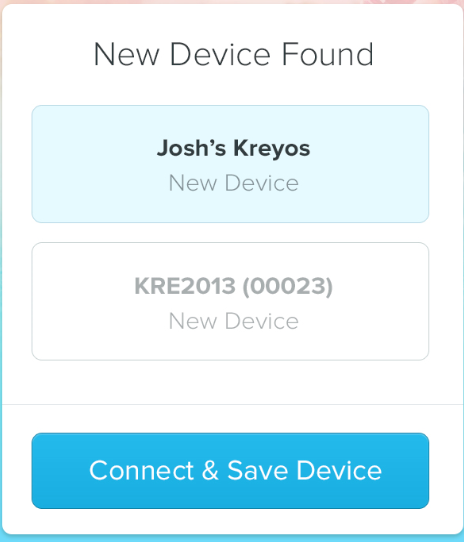
* 1. **Watch Alarms**

WatchAlarmActivity uses Protocol.setWatchAlarm() in syncWatchAlarms() to set the alarms on watch.



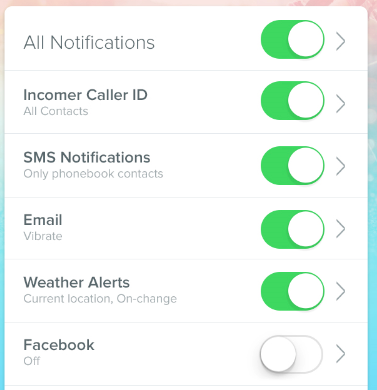
* 1. **Bluetooth**

DeviceListActivity is in response of this page. In the class, if a device is selected and the button is clicked, connectDevice() function will store the device name to global SharedPreferences for later use and initialize the Bluetooth delegate of the KreyosService.



* 1. **Notification**

This page will not send data to watch directly. But the notification settings will be configured and saved to Global SharedPreferences. In the meantime, the correspondent KreyosService listeners or tasks will be started. Correspondent notification will be handled in the KreyosService.



1. **Kreyos Service:**

This service is important for notification and workout data sync. There is 3 major parts:

* Notification listeners and Tasks

There are many notification listeners or push tasks:

SMS, Call, Facebook Feed, Twitter, Reminder, Low Battery.

All of them will use Protocol.notifyMessage() to transfer an vibrate notification to the watch.

* Workout data sync task

This is mainly used to obtain GPS information to the watch if watch is currently in workout status. The LocationUpdater will retrieve GPS info and notify the listener function. The result listener function gotLocation() will use Protocol.sendGPSInfo() to update GPS info on watch.

* None-workout data sync task

A silent task named TimeSyncThread will timely read historic data from watch via Protocol.readFile().

1. **Windows Phone Bluetooth Module**
   1. **Architecture**



* 1. **Interfaces**

**BluetoothAgent**

|  |  |
| --- | --- |
| **Name** | **Functionality** |
| Connected | Check the Bluetooth status |
| InnnerProtocol | Get or set the embedded protocol instance.  Event handler must be set on this protocol object. |
| Instance | Return the BluetoothAgent singleton |
| GetPairedDevices() | Get a list of paired Kreyos Devices |
| Start() | Start the BluetoothAgent internal processor on target device by device name |
| Stop() | Stop current BluetoothAgent session. |
| SendBytes() | Send a bytes array to target |
| BatchRegisterJob() | Commit a batch of sending job atomically. These batch jobs will be sent one by one without break. |

**Protocol**

|  |  |
| --- | --- |
| **Name** | **Functionality** |
| Protocol() | The constructor to create an instance of protocol. To do this need a BluetoothAgent reference. |
| echo() | Send an echo packet to watch |
| readFile() | Send a request to read a file from watch. This function just sends the request. Caller must register an OnFileReceived event handler for coming file data. |
| sycTime() | Sync phone time to watch |
| notifyMessage() | Send a notification to watch. |
| getDeviceID() | Send a request to read Device ID from watch. Caller must register an OnDeviceIDRead event handler for the returning device ID |
| sendFile() | Send file to watch. |
| setWatchAlarm() | Send watch alarm info to watch. |
| setGestureControl() | Set gesture configuration to watch. |
| syncWatchConfig() | Sync all configuration. |
| sendGPSInfo() | Sync GPS info from phone to watch |
| OnFileReceived | Be triggered once file data received. |
| OnDeviceIDRead | Be triggered when device id returned from watch. |
| OnActivityDataRead | Be triggered when an activity is ready to start. |
| OnActivityDataEnd | Be triggered when an activity is ended. |
| OnActivityDataPrepared | Be triggered when an activity is started. |
| OnActivityDataSync | Be triggered when an activity data is coming. |

* 1. **Flowcharts**

**Activity Data Sync Flow Chart**



1. **BLE Service/Characteristic Definition**

BLE is used in iOS and will be applied on Android 4.4 and later version.

BLE is simpler than STLV over RFCOMM. It is defined based on services and their characteristic.

We can simply treat this services/characteristic structure as kind of tree. Service is a category of properties and characteristic is the property under a service.

In Kreyos watch firmware, we support only one service since the architecture limitation.

23 dynamic read/write characteristics are defined so far under this service.

In Kreyos watch, each characteristic can be only one of the following 4 types:

* BYTE Array – a bunch of byte with predefined fix size.
* Short Array – an array of 16 bit integer with predefined fix size.
* Integer Array – an array of 32 bit integer with predefined fix size.
* String – a null terminated string with maxim size.

**The detail format of the 32 characteristics are defined in table below:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **UUID** | **Name** | **Type** | **Size** | **Comments** |
| FFF1 | TEST\_READ | byte | 1 | For test. |
| FFF2 | TEST\_WRITE | byte | 1 | For test. |
| FFF3 | DATETIME | byte | 6 | Split date time information. |
|  | Byte[0] : year – year miners 2000, for example, 2013 – 13  Byte[1] : month – 0~11  Byte[2] : month day – 0~30  Byte[3] : hour – 0~23  Byte[4] : minutes – 0~59  Byte[5] : seconds – 0~59 | | | |
| FFF4 | ALARM\_0 | byte | 3 | Byte[0] : 0 – disable, 1 – enable  Byte[1] : hour to trigger alarm  Byte[2] : minute to trigger alarm |
| FFF5 | ALARM\_1 | byte | 3 |
| FFF6 | ALARM\_2 | byte | 3 |
| FFF7 | SPORTS\_GRID | byte | 4 | Indicates the meanings for each grid |
| GRID type is defined as below:  private static final byte DATA\_WORKOUT = 0;  private static final byte DATA\_SPEED = 1;  private static final byte DATA\_HEARTRATE = 2;  private static final byte DATA\_CALS = 3;  private static final byte DATA\_DISTANCE = 4;  private static final byte DATA\_SPEED\_AVG = 5;  private static final byte DATA\_ALTITUTE = 6;  private static final byte DATA\_TIME = 7;  private static final byte DATA\_SPEED\_TOP = 8;  private static final byte DATA\_CADENCE = 9; | | | |
| FFF8 | SPORTS\_DATA | Int32 | 5 | Sports data sync to phone. |
|  | Detail meaning for each integer is:  Int32[0] :  SPORTS\_DESC.Int32[0] == 1 – Seconds Elapse from the activity begin  SPORTS\_DESC.Int32[0] == 2 – Timestamp from 2000-01-01 00:00:00  If it is -1, the sync progress is ended.  Int32[1] : sports data according to SPORTS\_GRID.byte[0]  Int32[2] : sports data according to SPORTS\_GRID.byte[1]  Int32[3] : sports data according to SPORTS\_GRID.byte[2]  Int32[4] : sports data according to SPORTS\_GRID.byte[3] | | | |
| FFF9 | SPORTS\_DESC | Int32 | 2 | Describe the sports data. |
|  | Int32[0] : 0 – no valid sports data, 1 – workout mode, 2 – sync mode  Int32[1] : if it is in workout mode, this integer indicates the start timestamp of the current workout. | | | |
| FF10 | DEVICE\_ID | Int32 | 1 | The unique identify of the watch. |
| FF11 | FILE\_DESC | Complex | 20 | Describe the File Data |
|  | FILE\_DESC.Str[0]:  Set by App:  ‘I’ – Investigate Available Sports Data File  ‘R’ – Read specific Sports Data File  ‘W’ – Write File down to watch  ‘S’ – Send Block to watch  ‘C’ – Write File Completed  ‘X’ – Reset the file transferring handler  Set by Watch:  ‘F’ – File name offered as  ‘N’ – No available files  ‘D’ – Data arrived in FILE\_DATA  ‘E’ – End of all data blocks  ‘H’ – Handle for data transferring is ok  ‘P’ – The Watch is prepared for data block transferring.  ‘O’ – The current data block transferring is Over.  To understand the detail of this part, see the next table. | | | |
| FF12 | FILE\_DATA | byte | 20 | File Data Block |
| FF13 | GPS\_INFO | Int32 | 3 | GPS Information sync to watch. |
|  | The GPS information is defined as below:  Int32[0] : Speed in meter \* 100/second  Int32[1] : Altitude in meter  Int32[2] : Distance in meter \* 10 | | | |
| FF14 | GESTURE | byte | 5 | The gesture configuration sync to watch. |
|  | Detail for each byte are defined here:  Byte[0] : bit 0 – enable/disable, bit 1 – left hand/right hand  Byte[1] : Action for gesture “swipe right”  Byte[2] : Action for gesture “swipe left”  Byte[3] : Action for gesture “twist right”  Byte[4] : Action for gesture “twist left”  The actions are defined as below:  1) Music Control - Next Song  2) Music Control - Previous Song  3) Music Control - Play/Pause  4) Music Control - Volume Up  5) Music Control - Volume Down  6) Phone: Pick Up  7) Phone: Reject | | | |
| FF15 | WORLDCLOCK\_0 | string | 10 | Strings to each world clock name. Here is some examples:  "Shanghai", "London", "New York". |
| FF16 | WORLDCLOCK\_1 | string | 10 |
| FF17 | WORLDCLOCK\_2 | string | 10 |
| FF18 | WORLDCLOCK\_3 | string | 10 |
| FF19 | WORLDCLOCK\_4 | string | 10 |
| FF20 | WORLDCLOCK\_5 | string | 10 |
| FF21 | WATCHFACE | byte | 2 | Watch face configuration. |
|  | Byte[0] : 0 – analog watch face, 1 – digital watch face  Byte[1] : index of the watch face | | | |
| FF22 | SPORTS\_GOALS | short | 3 | Goals set in home page “set goals” button. |
|  | Detail usage for each slot inside the array is:  Short[0] : Steps  Short[1] : Distance in meter  Short[2] : Calories | | | |
| FF23 | USER\_PROFILE | byte | 2 | User height and weight in metrics unit. |
|  | Byte[0] : height in cm  Byte[1] : weight in kg  Byte[2] : Circumference | | | |

**FILE\_DESC Symbol Meaning:**

Since the FILE\_DESC is kind of complex data and it might be in various format depending on different scenarios, I list the meaning for the symbol used in the characteristic below:

|  |  |
| --- | --- |
| **Symbol** | **Comments** |
| FILE\_DESC.Str[n] | Cast the FILE\_DESC returned NSObject to a char array and pick only the first char. |
| FILE\_DESC.Str[n-m] | Cast the FILE\_DESC returned NSObject to a char array and pick the range from n to m to build up a NSString. |
| FILE\_DESC.Int8[n] | Cast the FILE\_DESC returned NSObject to a byte array and pick the nth byte, then cast it to an integer. |
| FILE\_DESC.Int16[n] | Cast the FILE\_DESC returned NSObject to a byte array. Pick continuous 2 bytes at n \* 2 and n \* 2 + 1. Cast each of them into integers and then adding them up by the following way:  (Integer from byte[n\*2]) \* 16 + (Integer from byte[n\*2 + 1]) |

Normal cases of FILE\_DESC’s binary formats are listed below:

The most often used format:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte0 | Byte1 | Byte2 – Byte3 | Byte 4 – Byte 19 |
| Flag | Size | Block id | File name |

Sample code to read:

|  |
| --- |
| - (void) valueChanged:(NSData\*)value fromCharacteristic:(NSString \*)characteristic  {  if ([characteristic isEqual:BLE\_HANDLE\_FILE\_DESC]) {  int8\_t cursor = 0;  int8\_t flag = 0;  int8\_t size = 0;  int16\_t blockid = 0;  NSString\* filename = nil;  [value getBytes:&flag range:NSMakeRange(cursor, sizeof(flag))];  cursor = cursor + sizeof(flag);  [value getBytes:&size range:NSMakeRange(cursor, sizeof(size))];  cursor = cursor + sizeof(size);  [value getBytes:&blockid range:NSMakeRange(cursor, sizeof(blockid))];  cursor = cursor + sizeof(blockid);  filename = [[NSString alloc]initWithData:  [value subdataWithRange:NSMakeRange(  cursor, [value length] - cursor)] ncoding:NSUTF8StringEncoding];  cursor = [value length];  }  } |

Sample code to write:

|  |
| --- |
| NSData \*data = [[NSData alloc] init];  int8\_t flag = (int8\_t)’W’;  int8\_t size = 0;  int16\_t blockid = 0;  NSString\* filename = @"FIRMWARE";  [data appendBytes:&flag length:sizeof(flag)]; //append flag  [data appendBytes:&size length:sizeof(size)]; //append size  [data appendBytes:&blockid length:sizeof(blockid)]; //append blockid  [data appendData:[filename dataUsingEncoding:NSUTF8StringEncoding]]; //file name  //now send data via FILE\_DESC |

Investigating Request to Watch:

|  |  |
| --- | --- |
| Byte0 | Byte1 - 19 |
| ‘I’ | N/A |

Data File found Response from Watch:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte0 | Byte1 | Byte2 – Byte3 | Byte 4 – Byte 19 |
| ‘F’ | 0 | 0 | File name |

Read block request to watch:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte0 | Byte1 | Byte2 – Byte3 | Byte 4 – Byte 19 |
| ‘R’ | 0 | Block ID | File name |

Block data ready to transferring Response from watch:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte0 | Byte1 | Byte2 – Byte3 | Byte 4 – Byte 19 |
| ‘D’ | Size | Block ID | File name |

Write File request to watch:

|  |  |
| --- | --- |
| Byte0 | Byte1 – Byte19 |
| ‘W’ | File name |

Handler for file transferring ready response from watch:

|  |  |
| --- | --- |
| Byte0 | Byte1 – Byte19 |
| ‘H’ | N/A |

Send block request to watch:

|  |  |  |  |
| --- | --- | --- | --- |
| Byte0 | Byte1 | Byte2 – Byte3 | Byte 4 – Byte 19 |
| ‘S’ | Size | Block ID | File name |

Block reading is prepared response from watch:

|  |  |
| --- | --- |
| Byte0 | Byte1 – Byte19 |
| ‘P’ | N/A |

**Sports Data Sync Flow**

**Step 1:**

In sports page, phone app should periodically query the status of sports activity via the SPORTS\_DESC characteristic.

**Step 2:**

If SPORTS\_DESC.Int32[0] become 1, the watch gets into sports workout mode. Thus the SPORTS\_DATA characteristic become available. Phone app should query sports data record via the SPORTS\_DATA characteristic and display them to UI as described in Step 3.

If SPORTS\_DESC.Int32[0] is 0, the watch is idle and no data is available, go back to Step 1 and keep tracking the SPORTS\_DATA flag.

If SPORTS\_DESC.Int32[0] is 2, the watch has none-activity sports data to sync to watch. Phone app should thus read them from SPORTS\_DATA, store them in phone for Home Page using and upload them to cloud db. Go to Step 4 for detail.

**Step 3:**

If SPORTS\_DATA.Int32[0] is -1, the sports data sync might be ended or not started. So, go back to step 2. Otherwise, the phone app should keep reading data from the SPORTS\_DATA and display them to UI.

Additional work in this step is to start GPS tracker, and sync GPS information to watch via GPS\_INFO characteristic.

**Step 4:**

If SPORTS\_DATA.Int32[0] is -1, the sync is not available, go back to step 2. Otherwise, read data from SPORTS\_DATA and store them locally.

**Flow chart:**



**Daily Activity Data Sync Flow (File Read Flow):**



**Firmware Transferring Flow (Upload File Flow):**

