# Patchkeeper data format details

#### Packet head info:

typedef struct packet\_info\_struct {

uint32\_t sop;

uint32\_t timestamp\_hi; // If we represent this as 64-bit we have alignment issues with padding added

uint32\_t timestamp\_lo; // So send as two 32-bit numbers and put together in the app.

uint8\_t logger\_id;

uint8\_t custom; // Makes the structure end 32-bit aligned;

uint16\_t length;

} packet\_info\_t;

## Config packet format:

// FIRMWARE VERSION information define at the beginning of sdk\_config.h file  
# define FIRMWARE\_VERSION\_ID FIRMWARE\_MAJOR\_VERSION \* 1 00 00 00 + \  
 FIRMWARE\_MINOR\_VERSION \* 1 00 00 + \  
 FIRMWARE\_PATCH\_NUM \* 1 00 + \  
 FIRMWARE\_BUILD\_NUM

Example for version 1012: 1000102, saved as 4 bytes integral

#define LOGGER\_CONFIG\_DATA\_LENGTH\_BYTES 32  
#define LOGGER\_CONFIG\_ID 0x00

typedef struct {  
 packet\_info\_t packet\_info;  
 int8\_t data[LOGGER\_CONFIG\_DATA\_LENGTH\_BYTES]; // 32 bytes of data  
} config\_data\_t;

config\_data.packet\_info.sop = SOP\_BELLLABS;  
 config\_data.packet\_info.timestamp\_lo = timestamp.lo;  
 config\_data.packet\_info.timestamp\_hi = timestamp.hi;  
 config\_data.packet\_info.logger\_id = LOGGER\_CONFIG\_ID;  
 config\_data.packet\_info.length = LOGGER\_CONFIG\_DATA\_LENGTH\_BYTES; //32  
 config\_data.packet\_info.custom = packet\_count++;

**packet info format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -A4-EA-5C-00 | -00 | -00-00 | -20 |
| SOP\_BELLLABS:  lleB | timestamp.hi  ts\_hi: | timestamp.lo  ts\_lo: 4 bytes | logger ID: 0x00  config data | packet\_count | data length  (max 242)  0x20 : 32 bytes |
| 4 bytes | 4 bytes | 4 bytes | 1 byte | 1 byte | 2 bytes |

**242 = 242 (244 -2 because of packet\_count)**

**then followed by config data:**

**Followed by config data info (32 bytes in the following format, data length in header should be 0x20):**

// Change the pointer type so it is transferred correctly.  
 uint32\_t \*data32 = (uint32\_t\*) config\_data.data;  
 uint16\_t \*data16 = (uint16\_t\*) config\_data.data;

\*data32 = NRF\_RTC2->PRESCALER;  
 \*(data16+2) = RTC\_INPUT\_FREQ;  
 \*(data16+3) = CONFIG\_DATA\_VERSION;

\*(data32+2) = m\_timestamp\_offset.lo;  
 \*(data32+3) = m\_timestamp\_offset.hi;  
 \*(data32+4) = m\_old\_timestamp\_offset.lo;  
 \*(data32+5) = m\_old\_timestamp\_offset.hi;  
 \*(data32+6) = FIRMWARE\_VERSION\_ID;  
 \*(data32+7) = m\_last\_reset\_reason;

config data format:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| NRF\_RTC2->PRESCALER | RTC\_INPUT\_FRE | CONFIG\_DATA\_VERSION | m\_timestamp\_offset.lo | m\_timestamp\_offset.hi | m\_old\_timestamp\_offset.lo | m\_old\_timestamp\_offset.hi | FIRMWARE\_VERSION\_ID | m\_last\_reset\_reason |
| 4 bytes | 2 bytes | 2 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes | 4 bytes |

Python script for processing config data:

def process\_config\_data(SensorData):

global PrescalerValue

global RTCInputFrequency

global Ticks\_per\_second

global DataPackingVersion

# print("Config Data Handler")

PrescalerValue = (SensorData[3] << 24 | SensorData[2] << 16 << SensorData[1] << 8 | SensorData[0])

RTCInputFrequency = (SensorData[5] << 8 | SensorData[4])

DataPackingVersion = (SensorData[7] << 8 | SensorData[6])

print("Updating PrescalarValue to ", PrescalerValue)

print("Updating RTC Input Frequency to ", RTCInputFrequency)

print("Data Format Version(should be 3): ", DataPackingVersion)

Ticks\_per\_second = RTCInputFrequency / (PrescalerValue + 1)

return

## Data packet format from all Sensors

Battery 0x05 6C-6C-65-42-00-00-00-00-7F-4B-78-00-05-02-17-00-56-D8-00-00

IMU: 0x2A 6C-6C-65-42-00-00-00-00-56-4F-78-00-2A-10-C1-12-03-00-09-00-03-00-68-04-12-00-F7-BC-D6-24-5D-00

GSR:0x22 6C-6C-65-42-00-00-00-00-DD-51-78-00-22-10-60-00-79-4B-78-00-00-90-19-40-DD-51-78-00-00-90-19-40

Strain: 0x24 6C-6C-65-42-00-00-00-00-E1-51-78-00-24-10-60-00-7A-4B-78-00-01-F4-3F-40-E1-51-78-00-01-F4-3F-40

Pulse: 0x26 6C-6C-65-42-00-00-00-00-E9-51-78-00-26-10-60-00-7B-4B-78-00-01-F4-3F-40-E9-51-78-00-01-F4-3F-40

Temperture 0x28: 6C-6C-65-42-00-00-00-00-82-4B-78-00-28-10-77-00-86-CB-77-00-01-F4-3F-40-82-4B-78-00-01-F4-3F-40

ECG: 0x02: 6C-6C-65-42-00-00-00-00-B7-9A-2D-00-20-E4-C5-01-B7-9A-2D-00-01-00-7F-FF-FF-01-12-F2-61-9B-2D-00-01-01-7F-FF-FF-01-13-15-67-9C-2D-00-01-02-7F-FF-FF-01-13-70-6D-9D-2D-00-01-03-7F-FF-FF-01-13-39-74-9E-2D-00-01-04-7F-FF-FF-01-13-3E-0F-A1-2D-00-01-05-7F-FF-FF-01-13-4B-87-A1-2D-00-01-06-7F-FF-FF-01-13-17-8E-A2-2D-00-01-07-7F-FF-FF-01-13-16-94-A3-2D-00-01-08-7F-FF-FF-01-13-2D-9A-A4-2D-00-01-09-7F-FF-FF-01-13-33-7A-A7-2D-00-01-0A-7F-FF-FF-01-13-51-AE-A7-2D-00-01-0B-7F-FF-FF-01-13-64-B4-A8-2D-00-01-0C-7F-FF-FF-01-13-64-BA-A9-2D-00-01-0D-7F-FF-FF-01-13-58-C1-AA-2D-00-01-0E-7F-FF-FF-01-13-43-DB-AD-2D-00-01-0F-7F-FF-FF-01-13-35-DB-AE-2D-00-01-10-7F-FF-FF-01-13-44-E2-AF-2D-00-01-11-7F-FF-FF-01-13-79-E7-B0-2D-00-01-12-7F-FF-FF-01-13-6A

16 bytes packet\_info + subpackets

typedef struct {   
 packet\_info\_t packet\_info;   
 adc\_sensor\_data\_buffer\_t data[PATCHKEEPER\_TEMP\_BUFFER\_SIZE];   
} patchkeeper\_temp\_data\_t;

### Packet head info:

typedef struct packet\_info\_struct {

uint32\_t sop;

uint32\_t timestamp\_hi; // If we represent this as 64-bit we have alignment issues with padding added

uint32\_t timestamp\_lo; // So send as two 32-bit numbers and put together in the app.

uint8\_t logger\_id;

uint8\_t custom; // Makes the structure end 32-bit aligned;

uint16\_t length;

} packet\_info\_t;

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Bytes | bits |
| 6C-6C-65-42 | lleB | 4 | 32 |
| -00-00-00-00 | timestamp\_hi | 4 | 32 |
| -30-71-06-00 | timestamp\_lo | 4 | 32 |
| -22 (24, 26 0r 28) | logger\_id | 1 | 8 |
| -05 | data subpacket count (cycling 0-255): 0x05 | 1 | 8 |
| -E4-00 | length0x00E4(LSB first): it changes from 1 byte (V2 and before)to 2 bytes (V3) for streaming/save large amount of data without a new header. Custom for subpacket counting reduced to 1 byte:  0x10: 16 bytes for 2 readings of each sensor : ts\_lo (4) and float\_t (4)  0xE4: 228bytes for ECG  0x02: 2 bytes for battery percentage and coded voltage (padded with 2 bytes of 0 for saving and streaming)  0x10: 16 bytes for IMU: 2 bytes each value gyro.x, y, z, accel.x, y, z, ts\_lo (4)  0x04: 4 bytes for Audio mean and peak, 2 bytes each (16 bits integral) | 2 | 16 |
|  |  |  |  |
| Then followed by | 20 sub-packets of this: total 160 bytes | 16 bytes of header for all BLE packets |  |
| -DF-0C-05-00 | : timestamp\_lo of individual sub-packet | 4 | 32 |
| -CE-BC-54-40 | : float 32 bit float\_t data (reading of voltage): 3.32402 | 4 | 32 |
|  |  |  |  |
| -AC-19-05-00 | :second sub-packet, timestamp\_lo of individual packet | 4 | 32 |
| -CE-74-54-40 | :second packet,t 32 bit float\_t data (voltage) : 3.31963 | 4 | 32 |
|  |  |  |  |

**length of following data: multiples of subpackets:**

**16 bytes for analog sensor data, multiple of 8 bytes**

**228 for ECG data, multiple of 12 bytes**

**IMU and battery data are the same as in Gecko device.**

**2 bytes for battery data:**

Analog sensors: 8 bytes x 2 data points (4 bytes for ts\_lo and 4 bytes for float value)

ECG: 12 bytes x 19 supbackets

Batt: 2 bytes x 1 reading //PATCHKEEPER\_BATT\_DATA\_LENGTH\_BYTES

Audio mean and peak: PATCHKEEPER\_AUDIO\_RESULT\_DATA\_LENGTH\_BYTES\*sizeof(audio\_memory\_t);

#### IMU sensor data subpacket format:

#define PATCHKEEPER\_IMU\_DATA\_LENGTH\_BYTES 16

#define PATCHKEEPER\_IMU\_DATA\_LENGTH\_HALF\_WORDS PATCHKEEPER\_IMU\_DATA\_LENGTH\_BYTES/2

#define PATCHKEEPER\_IMU\_ID 0x2a

typedef struct {   
 packet\_info\_t packet\_info;   
 int16\_t data[PATCHKEEPER\_IMU\_DATA\_LENGTH\_HALF\_WORDS]; = data[8]  
} imu\_data\_t;

typedef struct {  
 bmi160\_sensor\_data\_simple\_t gyro\_data;  
 bmi160\_sensor\_data\_simple\_t accel\_data;  
 uint32\_t sensortime;

} imu\_data\_packed\_t;

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -A4-EA-5C-00 | -2A | -10-00 | -79 | -FE-FF | -F9-FF | -FE-FF | -5E-04 | -8C-00 | -FE-BC | -C1-EC-  -47-00 |
| lleB | ts\_hi | ts\_lo | 0x2A for IMU | length:0x10: means 16 bytes data | custom : sequence | gyro.x | gyro.y | gyro.z | accel.x | accel.y | accel.z | imu\_internal\_ts |
| 4 bytes | 4 bytes | 4 bytes | 1 byte | 2 bytes | 1 byte | 2 | 2 | 2 | 2 | 2 | 2 | 4 |

##### python scripts for processing IMU data (from Gecko):

def process\_imu\_data(SensorData):

# gx = int(SensorData[1] << 8 | SensorData[0])

# gy = int(SensorData[3] << 8 | SensorData[2])

# gz = int(SensorData[5] << 8 | SensorData[4])

# ax = int(SensorData[7] << 8 | SensorData[6])

# ay = int(SensorData[9] << 8 | SensorData[8])

# az = int(SensorData[11] << 8 | SensorData[10])

gx = 125\*3.14159/(2\*\*16\*180)\*int.from\_bytes(SensorData[0:2],byteorder='little',signed=True)

gy = 125\*3.14159/(2\*\*16\*180)\*int.from\_bytes(SensorData[2:4],byteorder='little',signed=True)

gz = 125\*3.14159/(2\*\*16\*180)\*int.from\_bytes(SensorData[4:6],byteorder='little',signed=True)

ax = 9.80665/2\*\*14\*int.from\_bytes(SensorData[6:8],byteorder='little',signed=True)

ay = 9.80665/2\*\*14\*int.from\_bytes(SensorData[8:10],byteorder='little',signed=True)

az = 9.80665/2\*\*14\*int.from\_bytes(SensorData[10:12],byteorder='little',signed=True)

imu\_internal\_ts = (SensorData[14] << 16 | SensorData[13] << 8 | SensorData[12])

return {'type':'imu\_ts','gx':gx,'gy':gy,'gz':gz,'ax':ax,'ay':ay,'az':az,'imu\_internal\_ts':imu\_internal\_ts}

#### Analog sensors data format:

**PATCHKEEPER\_GSR\_ID 0x22**

**PATCHKEEPER\_STRAIN\_ID 0x24**

**PATCHKEEPER\_PULSE\_ID 0x26**

**PATCHKEEPER\_TEMP\_ID 0x28**

#define PATCHKEEPER\_GSR\_BUFFER\_SIZE 2  
#define PATCHKEEPER\_STRAIN\_BUFFER\_SIZE 2

#define PATCHKEEPER\_PULSE\_BUFFER\_SIZE 2

#define PATCHKEEPER\_TEMP\_BUFFER\_SIZE 2

typedef struct {  
 uint32\_t ts\_lo; // 4 bytes  
 float\_t value; // 4 bytes  
} adc\_sensor\_data\_buffer\_t; // 8 bytes total

typedef struct {   
 packet\_info\_t packet\_info;   
 adc\_sensor\_data\_buffer\_t data[PATCHKEEPER\_GSR\_BUFFER\_SIZE];   
} patchkeeper\_gsr\_data\_t;

typedef struct {   
 packet\_info\_t packet\_info;   
 adc\_sensor\_data\_buffer\_t data[PATCHKEEPER\_STRAIN\_BUFFER\_SIZE];   
} patchkeeper\_strain\_data\_t;

typedef struct {   
 packet\_info\_t packet\_info;   
 adc\_sensor\_data\_buffer\_t data[PATCHKEEPER\_PULSE\_BUFFER\_SIZE];   
} patchkeeper\_pulse\_data\_t;

typedef struct {   
 packet\_info\_t packet\_info;   
 adc\_sensor\_data\_buffer\_t data[PATCHKEEPER\_TEMP\_BUFFER\_SIZE];   
} patchkeeper\_temp\_data\_t;

#define PATCHKEEPER\_GSR\_DATA\_LENGTH\_BYTES PATCHKEEPER\_GSR\_BUFFER\_SIZE \* sizeof(adc\_sensor\_data\_buffer\_t)

#define PATCHKEEPER\_STRAIN\_DATA\_LENGTH\_BYTES PATCHKEEPER\_STRAIN\_BUFFER\_SIZE \* sizeof(adc\_sensor\_data\_buffer\_t)  
#define PATCHKEEPER\_PULSE\_DATA\_LENGTH\_BYTES PATCHKEEPER\_PULSE\_BUFFER\_SIZE \* sizeof(adc\_sensor\_data\_buffer\_t)  
#define PATCHKEEPER\_TEMP\_DATA\_LENGTH\_BYTES PATCHKEEPER\_TEMP\_BUFFER\_SIZE \* sizeof(adc\_sensor\_data\_buffer\_t)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -DD-51-78-00 | -22 | -10-00 | -60 | -79-4B-78-00 | -00-90-19-40 | -DD-51-78-00 | -00-90-19-40 |
| lleB | ts\_hi | ts\_lo | 0x22/ 0x24/ 0x26/  0x28 | length:0x10: 16 bytes data | custom 1 byte: sequence No. of packets | ts\_lo1: 4 bytes | adc reading1, float\_t:  4 bytes | ts\_lo2: 4 bytes | adc reading2, float\_t, 4bytes |
| 4 bytes | 4 bytes | 4 | 1 | 2 | 1 | 4 | 4 | 4 | 4 |

#### Battery data format

There is no subpacket for battery data:

typedef struct {   
 packet\_info\_t packet\_info;   
 uint8\_t data[PATCHKEEPER\_BATT\_DATA\_LENGTH\_BYTES];   
} patchkeeper\_batt\_data\_t;

**PATCHKEEPER\_BATT\_ID 0x05**

#define PATCHKEEPER\_BATT\_BUFFER\_SIZE 1

#define PATCHKEEPER\_BATT\_DATA\_LENGTH\_BYTES PATCHKEEPER\_BATT\_BUFFER\_SIZE \* sizeof(nrf\_saadc\_value\_t) // 1X16 Bits

Battery :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -7F-4B-78-00 | -05 | -02-00 | -17 | -56 | -D8 | -00-00 |
| lleB | ts\_hi: 4 bytes | ts\_lo: 4 bytes | 0x05:  Battery\_ID | length:0x02: means 2 bytes data | custom 1 byte: sequence No. of packets | battery\_level\_Percent (86%) | coded\_vbatt\_8: 216 in decimal | padded for alignment 32 bits |
| 4 bytes | 4 | 4 | 1 | 2 | 1 | 1 | 1 | 2 |

python processing battery data (from Gecko):

def process\_battery\_data(SensorData):

battery\_percent = (SensorData[0])

if (DataPackingVersion>=1):

battery\_volt = 3.0 + SensorData[1]/200.0;

else:

battery\_volt = (SensorData[1]+300.0)/100.0;

battery\_level = battery\_volt

return {'type':'battery\_ts','battery\_percent':battery\_percent,'battery\_level':battery\_level}

#### ECG data subpacket format

Each ECG BLE packet contains: 16 bytes of header + 19 subpackets of data, total length = 12 x 19 = 228 bytes

Total bytes: 16+12 x 19 =244 each BLE packet

(To make receiving end parsing data easier, all data packets are formatted with header (16 bytes),

therefor maximum number of subpackets : 19 ( (244 -16)/12 = 19 ))

PATCHKEEPER\_ECG\_ID 0x20, data as following:

6C-6C-65-42-00-00-00-00-74-83-07-00-20-E4-47-000-74-83-07-00-01-00-7F-FF-FF-2B-B9-89-44-84-07-00-01-01-7F-FF-FF-2B-B4-E9-4B-85-07-00-01-02-7F-FF-FF-2B-B0-08-51-86-07-00-01-03-7F-FF-FF-2B-AA-9F-D9-89-07-00-01-04-7F-FF-FF-2B-9A-1D-6B-8A-07-00-01-05-7F-FF-FF-2B-95-27-71-8B-07-00-01-06-7F-FF-FF-2B-90-1B-78-8C-07-00-01-07-7F-FF-FF-2B-8A-63-7E-8D-07-00-01-08-7F-FF-FF-2B-85-1D-3E-90-07-00-01-09-7F-FF-FF-2B-7A-4B-91-90-07-00-01-0A-7F-FF-FF-2B-75-0A-97-91-07-00-01-0B-7F-FF-FF-2B-70-00-9E-92-07-00-01-0C-7F-FF-FF-2B-6B-12-A4-93-07-00-01-0D-7F-FF-FF-2B-66-13-A4-96-07-00-01-0E-7F-FF-FF-2B-5B-23-BA-96-07-00-01-0F-7F-FF-FF-2B-55-BF-BE-97-07-00-01-10-7F-FF-FF-2B-50-A8-C4-98-07-00-01-11-7F-FF-FF-2B-4B-21-CB-99-07-00-01-12-7F-FF-FF-2B-45-74

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -74-83-07-00 | -20 | -E4-00 | -47 | -74-83-07-00 | -01- | -00- | -7F-FF-FF | --2B-B9-89 |
| lleB | ts\_hi: 4 bytes | ts\_lo: 4 bytes | 0x20  ECG\_ID | length:0xE4: means 228 bytes data | custom 1 byte: sequence No. of packets | ts\_lo1: 4 bytes, time stamp of each data point | 1 byte status from ADS1992 | 1 byte subpacket count | 3 bytes of channel 1 data | 3 bytes of channel 2 data |
| 4 bytes | 4 | 4 | 1 | 2 | 1 | 4 | 1 | 1 | 3 | 3 |

The above grey part repeats 19 time in each packet, with subpacket counts changes from 0x00 to 0x12

subpacket format:

typedef struct {   
 uint32\_t timestamp\_lo;  
 uint8\_t status;  
 uint8\_t packet\_count; //for memory allocation alignment  
 uint8\_t channel1\_data[3];  
 uint8\_t channel2\_data[3];

} ads1292r\_packet\_t;

12 bytes each subpacket, since at least 10 bytes required for each data packets, use 12 bytes for struct data alignment, with status and packet count

#### Audio mean and average data subpacket format

typedef int16\_t audio\_memory\_t; // 2bytes

#define PATCHKEEPER\_AUDIO\_ID 0x04  
#define PATCHKEEPER\_AUDIO\_RESULT\_BUFFER\_SIZE 1  
#define PATCHKEEPER\_AUDIO\_RESULT\_DATA\_LENGTH\_BYTES PATCHKEEPER\_AUDIO\_RESULT\_BUFFER\_SIZE\*2\*sizeof(audio\_memory\_t) // 4 bytes

typedef struct {  
 packet\_info\_t packet\_info;  
 audio\_memory\_t data[PATCHKEEPER\_AUDIO\_RESULT\_DATA\_LENGTH\_BYTES]; = data[2]  
} audio\_data\_t;

Audio mean and peak:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -7F-4B-78-00 | -04 | -02-00 | -17 | -56-D8 | -00-00 |
| lleB | ts\_hi: 4 bytes | ts\_lo: 4 bytes | 0x04: audo ID | length:0x02: means 2 bytes data | custom 1 byte: sequence No. of packets | audio mean | audio peak |
| 4 bytes | 4 | 4 | 1 | 2 | 1 | 2 | 2 |

audio mean and peak range: from -32768 to 32767

python parsing audio data:

def process\_audio\_data(SensorData):

audio\_mean = (SensorData[1] << 8 | SensorData[0])

audio\_peak = (SensorData[3] << 8 | SensorData[2])

#### Audio raw data format

Audio raw data was continuously saved to SD card (or can be streamed if needed, not fully test, might lost packages) as .ALOG file.

Data are two channels stereo format, with sampling rate 16124 and 16bit depth.

// audio data at 16.125kHz i.e. 8062 samples at 16 bit/sample -> 16124   
#define AUDIO\_MEMORY\_SIZE\_BYTES MEMORY\_SIZE\_KB(32)   
#define AUDIO\_MEMORY\_NUM\_BUFFERS 2  
#define AUDIO\_MEMORY\_BUFFER\_SIZE AUDIO\_MEMORY\_SIZE\_BYTES/AUDIO\_MEMORY\_NUM\_BUFFERS

AUDIO\_MEMORY\_BUFFER\_SIZE 16k bytes

#### PPG data format (V10xxxxx firmware)

#define PATCHKEEPER\_PPG\_ID 0x2B

//#define PATCHKEEPER\_CONFIG\_PPG 0x2B

Each PPG data package contains a packet\_info\_t, followed by three uint32\_t for green, red and infrared reading respectively:

typedef struct {

packet\_info\_t packet\_info;

maxm86161\_ppg\_sample\_t data;

} patchkeeper\_ppg\_data\_t;

typedef struct {

uint32\_t ppg0; ///< ppg1 sample data

uint32\_t ppg1; ///< ppg2 sample data

uint32\_t ppg2; ///< ppg3 sample data

} maxm86161\_ppg\_sample\_t;

###### Default PPG sampling interval 30 seconds every 5 mins:

#define PATCHKEEPER\_DEFAULT\_PPG\_SAMPLE\_INTERVAL\_MS 50

#define PATCHKEEPER\_DEFAULT\_PPG\_DUTYCYCLE\_INTERVAL\_MS 300000 //5 mins

#define PATCHKEEPER\_DEFAULT\_PPG\_ON\_TIME\_MS 30000 //30 seconds

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -7F-4B-78-00 | -2B | -0C-00 | -17 | -56-D8-43-00 | -63-00-76-00 | -12-34-76-00 |
| lleB | ts\_hi: 4 bytes | ts\_lo: 4 bytes | 0x2B: PPG ID | length:0x0C: 12 bytes data | custom 1 byte: sequence No. of packets | 4 bytes: Green LED reading | 4 bytes: Infrared LED reading | 4 bytes:  Red reading |
| 4 bytes | 4 | 4 | 1 | 2 | 1 | 4 | 4 | 4 |

Processing with python:

MAXM86161\_REG\_FIFO\_DATA\_MASK = 0x07FFFF

MAXM86161\_REG\_FIFO\_RES = 19

MAXM86161\_REG\_FIFO\_TAG\_MASK = 0x1F

block\_buf = unpack\_from('BBBB', SensorData,offset)

temp\_data:int = block\_buf[2] << 16 | block\_buf[1] << 8 | block\_buf[0]

data\_val:int = temp\_data & MAXM86161\_REG\_FIFO\_DATA\_MASK

tag:int = (temp\_data >> MAXM86161\_REG\_FIFO\_RES) & MAXM86161\_REG\_FIFO\_TAG\_MASK

#### PPG data format

#define PATCHKEEPER\_PPG\_ID 0x2B

To speed up PPG data processing, PPG sensor data were read with I2C (TWI) interface in buck mode, about 60 data points every FIFO\_AFULL interrupt. Read data are saved/transmitted directly in one BLE packet. Each data point contains 3 bytes, first 5 bits for Tag and 19 bits for data (as described in maxm86161 datasheet p19):

Table

Description automatically generated

Number of data points in each batch read is in multiple of both 3 and 2 (30 or 60 used), this is for easy data processing as PPG is configured to use all 3 LEDs in each measurement (green, infrared and red in sequence). Therefore each timestamp corresponding to 3 data points for green, infrared and red LED.

Each PPG data package contains a packet\_info\_t, followed by three uint32\_t for green, red and infrared reading respectively:

typedef struct {

packet\_info\_t packet\_info;

maxm86161\_ppg\_bytes\_t data;

} patchkeeper\_ppg\_bytes\_t;

typedef struct {

uint8\_t ppg\_data\_byte[3\*128]; // ppg data bytes

} maxm86161\_ppg\_bytes\_t;

###### Default PPG sampling interval 30 seconds every 5 mins:

#define PATCHKEEPER\_DEFAULT\_PPG\_DUTYCYCLE\_INTERVAL\_MS 300000 //5 mins

#define PATCHKEEPER\_DEFAULT\_PPG\_ON\_TIME\_MS 30000 //30 seconds

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6C-6C-65-42 | -00-00-00-00 | -7F-4B-78-00 | -2B | -0C-00 | -17 | -56-D8-43 | -63-00-76 | -12-34-76 | -56-D8-43- |
| lleB | ts\_hi: 4 bytes | ts\_lo: 4 bytes | 0x2B: PPG ID | length:0xB4: 180 bytes data (this can change) | custom 1 byte: sequence No. of packets | 3 bytes: green LED reading | 3 bytes: Infrared LED reading | 4 bytes:  red reading | More bytes for LED readings |
| 4 bytes | 4 | 4 | 1 | 2 | 1 | 3 | 3 | 3 | 3X? |

The length of the packet (total number of bytes) could change during the reading process due to availability of data and some other possible errors. For data parsing, three data points are processed each time for same timestamp, additional individual points that can not be grouped in 3 are disregarded. Tag is checked for every data point before putting to array. The following python script is used:

block\_buf = unpack\_from('BBB', SensorData,offset)

temp\_data:int = block\_buf[0] << 16 | block\_buf[1] << 8 | block\_buf[2]

data\_val:int = temp\_data & MAXM86161\_REG\_FIFO\_DATA\_MASK

tag:int = (temp\_data >> MAXM86161\_REG\_FIFO\_RES) & MAXM86161\_REG\_FIFO\_TAG\_MASK

print("tag:value : " + repr(tag)+" : "+ repr(data\_val))

if tag == 1:

val1 = data\_val

elif tag == 2:

val2 = data\_val

elif tag == 3:

val3 = data\_val

else:

print("No correct tag, point use previous value................................" )

break

offset += No\_bytes\_each\_data