DT0063
Design tip

Bluetooth Low Energy network: time-stamping and sample-rate-conversion

By Andrea Vitali

Main components	
BLUENRG-MS	Bluetooth Low Energy Network Processor supporting Bluetooth 4.1 core specification
BLUENRG-1	Bluetooth Low Energy System On Chip
STEVAL-STLKT01V1	SensorTile development kit

Purpose and benefits

This design tip explains how to apply and use time-stamps for data sent over a BLE wireless link. It also explains how to process data to compensate for any sampling frequency mismatch, a process known as Sample-Rate-Conversion (SRC).

Benefits:

- Remote sensor units may not have an accurate crystal oscillator; instead, they may have an RC oscillator which can be off +/-5% with respect to the nominal desired frequency. Data is sent over the BLE wireless link from sensor node to sensor hub. By time stamping upon reception, it is possible to perform sample-rate-conversion to compensate.
- Linear interpolation is explained: this is the simplest sample-rate-conversion technique with minimal latency. A special note dedicated to averaging quaternions is also included.
- Data loss concealment is explained: this is actually a side effect of linear interpolation, which can fill gaps in the received data.

BLE wireless link overview

Connections are point to point. The central node is the master and controls the network. The peripheral is the slave and joins the network. BLUENRG/BLUENRG-MS as a master can handle up to 8 slaves. BLUENRG-MS can concurrently be master of up to 4 slaves, and slave of another master.

The central node and peripherals can be servers, clients, or both. Servers offer a list of services, made of characteristics, which clients can read upon request, read without request (by enabling notifications or indications), or write. Read and write can be with acknowledge (e.g. indication) or without acknowledge (notification).

Services and characteristics are listed in a table, which is known as the *attribute* list. Entries in the list follow this pattern: primary service declaration (0x2800), one or more characteristic declaration (0x2803) followed by the characteristic itself (standard 16-bits or custom 128-bits UUID), and possibly the characteristic configuration for the service (0x2902). The pattern repeats for each service. There are at least two services: Generic Access (0x1800) and Generic Attribute (0x1801). A given *attribute* has a pointer to this table, the pointer is also known as handle.

Whenever a BLE server notifies a BLE client, the following happens on the client:

1. The network processor BLUENRG receives the packet and holds it in a buffer, an interrupt request is sent to the host microcontroller STM32
2. The host microcontroller STM32 receives the interrupt; the interrupt handler runs HCI_Isr() in hci.c; the data packet is transferred over the SPI and, if there is enough space, it is stored in a queue (an empty node is taken from hciReadPktPool, it is filled with data, and it is put into hciReadPktRxQueue)
3. The host microcontroller STM32 periodically runs HCI_Process() in hci.c; any packet in the queue is taken and processed (the node is taken from hciReadPktRxQueue, processed, and it is put back into hciReadPktPool)
    a. HCI_Process() calls HCI_Event_CB() in ble_service.c, the event associated with the packet triggers the execution of a specific callback
    b. For notifications: EVT_BLUE_GATT_NOTIFICATION triggers the execution of GATT_Notification_CB() in ble_service.c, this is where the packet is parsed.
    c. For each packet, the **parser function** has access to pointer to connection (conn_handle), pointer to *attribute* (attr_handle), attribute value (attr_value) and attribute length (attr_len).

Step 1: parsing and reassembling based on timestamps

The **parser function** is called whenever a BLE notification data packet is received. If the data packet is known (attr_handle), it is parsed and data is copied to a local buffer. The local buffer is then processed.

The function is written carefully to guarantee that different types of data sent in different packets but with the same timestamp can be reassembled correctly. The typical case is when the raw data from the sensor is sent as a given *attribute* (e.g. ‘rawdata’ made of 3 triplets: XYZ values from accelerometer, gyroscope and magnetometer) and the sensor fusion output is sent as another *attribute* (e.g. ‘quaternion’ made of 1 quadruplet: WXYZ).

The function is also written in order to guarantee that no data is overwritten and lost, even if the timestamps are wrong (not monotonically increasing, or if wraparound happens). This is the sequence of operations:

1. The timestamp inside the packet, the **remote timestamp**, is converted from a fixed point RTC value to a common unit to make processing easier (as an example: msec). This timestamp drives the reassembly of the data.

2. If the buffer timestamp is not present (e.g. if it is negative), then the timestamp is copied into the buffer timestamp.

3. If the timestamp is strictly greater than the buffer timestamp, the buffer is processed before it is overwritten by new data. Then the buffer is cleared, its parts marked as empty, and the new timestamp is copied into the buffer timestamp. Note that timestamps are computed from the RTC and the RTC can wraparound. A ‘smaller’ timestamp is to be considered ‘greater’ if the difference is greater than half the maximum value it can have.

4. If the same data is already present in the buffer, then their timestamp is strictly lower (old data has been received) or equal (timestamp are not increasing) to the buffer timestamp. Old data can be discarded. Data with same timestamp can be kept. If discarded, execution stops here. If data is kept, then the buffer is processed before it is overwritten by new data. Then the buffer is cleared, its parts marked as empty, and the timestamp is copied into the buffer timestamp.

5. Data is copied into the buffer and the corresponding part is marked as present

6. If the buffer is complete, it is processed (see next paragraph). Then the buffer is cleared, its parts are marked as empty; the buffer timestamp must also be cleared (e.g. by setting it to a negative value).

Step 2: processing the data buffer

Data can be written to a local log file. Data can also be resampled to compensate for a frequency drift of the remote device connected via BLE wireless link.

1. The most important step: the master reads the local time and acquires the **local timestamp**. The quality of the sample-rate-conversion critically depends on the resolution of the local timestamp. As an example, under Windows the best function is QueryPerformanceCounter(). Under Linux, iOS, or Android, the accuracy of timer-related functions should be carefully checked.

2. Filter the timestamp (see next paragraph): the acquisition of the timestamp can be interrupted in a multi-task environment; this means that the timestamp will have a spurious time-varying offset which must be eliminated by filtering.

3. Sample-rate-conversion (see final paragraph): it is suggested to call this function only if the buffer has all the data, and not if it is partially full. In this way, it is possible to exploit the resampling to get **loss-concealment by linear interpolation**.

4. Optional steps: the buffer can be written to a log file; the buffer can be inserted into a FIFO to enable filtering; the buffer can also be cleared (actually this is also done in the parsing function when it is really needed).

Step 3: filtering local timestamps to get high quality timestamps

Timestamps are affected by an unknown offset due to interrupts: T=Ttrue+ofs. This is especially true in a multi-task environment where the acquisition of timestamps can be interrupted and delayed by other concurrent tasks.

If the offset is always the same, then it can be eliminated by subtraction: T1=T1true+ofs, T2=T2true+ofs, and T2-T1=T2true-T1true=TrueDelta.

If the offset is time-varying, then the subtraction cannot eliminate it, but the hope is that *on average* the offset has a given value, therefore by low-pass filtering Tdelta=T2-T1, it is possible to estimate TtrueDelta=filter(Tdelta).

Once TtrueDelta is estimated, it is possible to recompute the correct high quality timestamp as a multiple of TtrueDelta: Thq=n*TtrueDelta. This high quality timestamp will actually drive the sample-rate-conversion process.

The function is written carefully to manage correctly the case where one or more packets have been lost over the air and the Tdelta is greater than expected. If not managed properly, this would lead to incorrect computation. As an example: the expected time interval between consecutive packets is 40msec, but Tdelta=T2-T1=80msec, then one packet has been lost. Tdelta may not be accurate but that inaccuracy should not be as large as the expected time interval.

Filtering can be applied to remote and local timestamps. However, resampling is driven only by filtered local timestamps. This is the sequence of operations:

1. Delta time interval: Tdelta=T2-T1. Remote timestamps are computed from RTC and RTC can wraparound: if the delta is negative, then to make it positive, add the maximum value that the timestamp can have before wraparound happens.

2. Number of lost packets: Tdelta from remote timestamps should be preferred because they have less variance; while Tdelta is greater than 150% of expected delta, increment the lost packet counter and decrement Tdelta by the expected delta; empty or NaN lines can be written to log file to make losses evident.

3. Low-pass filtering: delta time intervals are averaged with the highest precision by keeping a cumulative sum and a counter. The cumulative sum of Tdelta is the numerator. The counter is the denominator. The counter is incremented by 1 plus the lost packet counter. The ratio is the average delta time.

4. Additional low-pass filtering: TavgDelta = 95% TavgDelta + 5% ratio. This further smoothes variations. TavgDelta is the filter output, which should be near TrueDelta.

5. High quality timestamp: Thq=previousThq+TavgDelta*(1+loss counter). This high quality timestamp drives the resampling function.

Step 4: sample-rate-conversion (SRC) by linear interpolation

Linear interpolation is a weighted average of previous data (d1) and current data (d2); it is based on the target timestamp (t), the high quality timestamp of previous data (t1) and the high quality timestamp of current data (t2).

1. If target t is less or equal to t2 then resample
    a. Compute alpha = (t-t1)/(t2-t1), alpha goes from 0 to 1
    b. Compute resampled data d=d1*(1-alpha)+d2*(alpha), two multiplications. Alternatively, d=d1+alpha*(d2–d1), which is equivalent but uses only one multiplication. Resampled data can be written to a secondary log file.
    c. Increment t by desired target delta time (re-sampled data can have a different rate with respect to data) and loop until t > t2.
2. Copy current data buffer and high quality timestamp (d2 and t2) into previous data buffer and high quality timestamp (d1 and t1) to prepare next iteration.

About **weighted average**: There is one notable exceptions **for quaternions**. Q and -Q representing the same orientation. Their average is 0, which is obviously wrong because it should be Q or –Q. Quaternions dot-product must be checked before averaging: if it is negative then one of the two quaternions must be negated: s=w1*w2+x1*x2+y1*y2+z1*z2; if s<0, then use –w2,-x2,-y2,-z2.

Support material

Related design support material
STEVAL-STLKT01V1: SensorTile development kit
Documentation
PM0237 Programming manual: BlueNRG, BlueNRG-MS stacks programming guidelines
UM1865 User manual: BlueNRG-MS Bluetooth® LE stack application command interface (ACI)
AN4494 Application note: Bringing up the BlueNRG and BlueNRG-MS devices

Revision history

Date	Version	Changes
13-Jul-2016	1	Initial release

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2016 STMicroelectronics – All rights reserved