2019 xxx

By NTHU CS & EE Team

SMART-insole project

Integration Report

**OutLine**

1. **progress Brief summary**
2. **wireless transmission** 
   1. **Experimental Settings**
   2. **Bluetooth Transmission System**
3. **Data Encoding, Recovery, and Interpolation**
4. **Multi-thread Programming and Hand-shaking mechanism**
5. **Time Synchronization mechanism and implementation**
   1. **Uploading the Sensor Data to the Cloud Database Process**
6. **Private Cloud**
7. **Public Cloud – Amazon Web Service**
8. **progress Brief summary**

We have implemented the “Bluetooth Transmission System” which provides a wireless communication module to support up to *N* sensors data collection and transmission in a real-time manner in this project. Data sensed by *N* sensors will be collected and transmitted respectively to the end device immediately through the Bluetooth Low Energy technique. To receive every piece of data from the beginning of the connection, the hand-shaking mechanism is implemented. Since sensed data is transmitted immediately, it is impossible to retransmit the lost data during the transmission. So, the linear data interpolation technique is implemented. To balance the number of packets received by the end device, we propose a “Time Synchronization on Feet” mechanism. To conclude, we achieve 14kbps data rates for each sensor without any significant packets lost and our system is proved to run for up to 12 hours. Finally, we can upload the sensor data to the cloud database.

1. **WIRELESS TRANSMISSION**
2. **Experimental Settings**

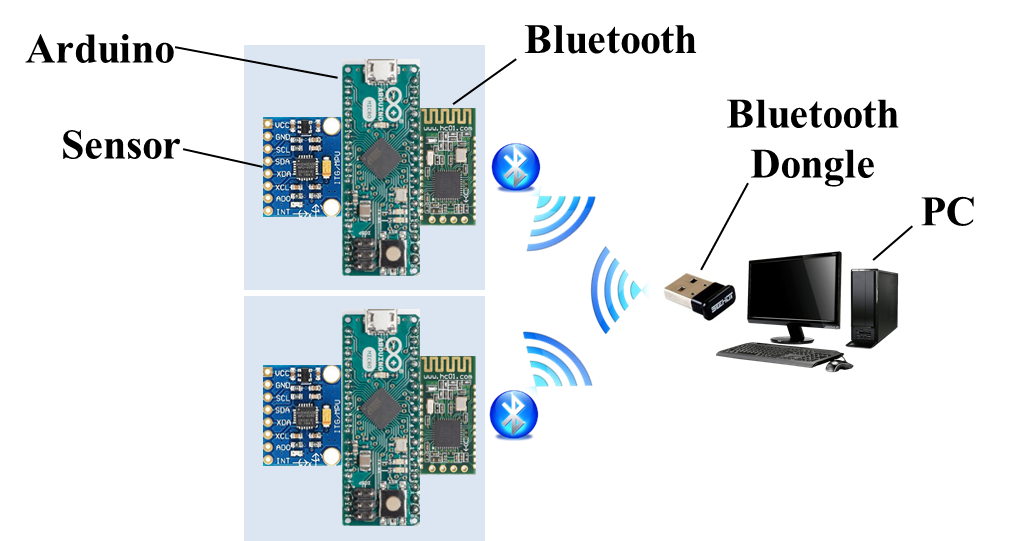
****

Figure 1. The experimental architecture of Bluetooth transmission.

The experimental equipment is shown in Figure 1 which includes:

* Development board: Arduino Micro \* 2
  + - * Microcontroller: ATmega328P
      * Operating voltage: 5V
* Sensor: MPU-6050 6-axis inertial sensor \* 2
* Bluetooth module: CC2540 HC-08 \* 2
* Bluetooth dongle: SBD-40

The packet format is shown in Figure 2. The number of sensors can be adjusted according to user-defined requirements.

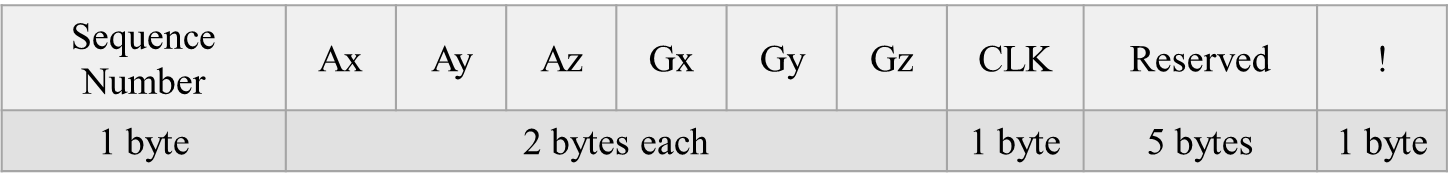


Figure 2. The packet format with a length of 20 bytes.

1. **Bluetooth Transmission System**
2. **Packet Encoding, Recovery, and Interpolation**

Our packet format is designed as shown in Figure 2. The last byte is an exclamation mark “!”, which is used as an indicator of the ending of a packet. In the implementation, “!” is sent as a byte, and the ASCII (American Standard Code for Information Interchange) code for “!” is in the decimal numeral system or in the hexadecimal numeral system. However, we are unable to correctly detect the end of a packet when the sequence number, clock or data itself contains . Thus, it is necessary to check each byte of the packet and guarantee that the sequence number, clock and data do not contain before sending the packet to the receiving end, If the sequence number or clock is , we will directly increase it by one. As for the data, we need to modify some bytes and ensure that the modification has a little impact on the original value. The modification policy is illustrated in Figure 3. The data of each axis is composed of two bytes, a higher byte and a lower byte. After the end device receives a complete packet from sensors, we will check the data of each axis. If the lower byte is , then we modify it to . If the higher byte is , then we check whether the lower byte is larger than . If the lower byte is larger than , we modify the higher byte to and lower byte to . Otherwise, we modify the higher byte to and lower byte to . To recover the missing data, we have to check the exclamation mark and the sequence number in the receiving end. First, we should detect whether missing data occurs. If the data size between two exclamation marks is equal to the exact packet length (e.g., 20 bytes in our setting for each sensor), there is no missing data. Otherwise, some data is missing and we will drop the incomplete data. Assume that the time interval between two packets is very small (i.e., ∆t→0), the data is approximately linear. We can use linear interpolation to recover the missing data given that the time interval is divided equally. Table 1 shows the correct data on the transmission end and the recovered data on the receiving end. By observing the result, the difference between the two data is small. Hence, we conclude that the linear interpolation method is acceptable.

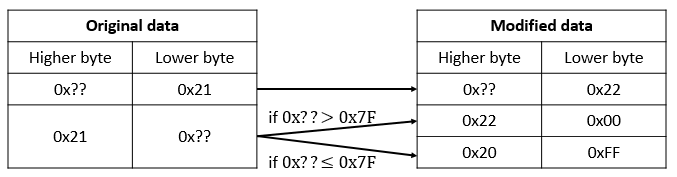


Figure 3. The modification policy for sensor data.

Table 1. An example of the correct data and the recovered data.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Ax | Ay | Az | Gx | Gy | Gz |
| Correct data | 0.015 | -0.009 | 0.954 | -1.023 | 0.030 | 0.969 |
| Recovered data | 0.010 | -0.009 | 0.961 | -0.969 | -0.248 | 0.580 |

1. **Multi-thread Programming and Hand-shaking mechanism** To achieve that one master can serve *N* slaves, we combined the Python multi-thread module with an open source “bluepy” Bluetooth module to implement our system. Note that the parameter “connection interval” is to determine how often the master will poll data from the slave. When the slave requests an update, it supplies a maximum, and a minimum wanted interval. Moreover, we found that the default setting of “connection interval” of our BLE dongle is 50ms ~ 70ms in the Linux operating system environment. These values could not afford the sample rates of the sensor in the transmission end and would cause missing data because the receiving end is in the connection interval. We set the values to 7.5ms ~ 8.5ms. In this way, we can receive the streaming packets successfully and steadily. To receive every piece of data from the beginning of the process, we added a hand-shaking process before the first piece of the sensor data transmission to ensure the connection established. The hand-shaking process is similar to the TCP (Transmission Control Protocol) three-way hand-shaking. The process is illustrated in Figure 4.

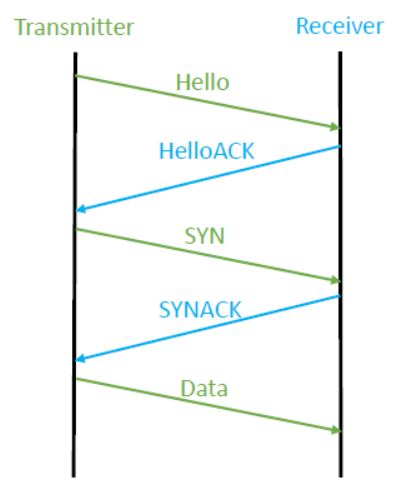


Figure 4. The hand-shaking process.

1. **Time Synchronization mechanism and implementation**

In the following, we aim to address two kinds of time synchronization problems in our Bluetooth transmission system and try to design possible solutions to these problems. First, we can see that the packet length is 20 bytes in total as shown in Figure 2. The exclamation mark “!” is put into the last byte of the packet to end the current packet. There is an attribute called “CLK” which is short for “Clock.” The function of this attribute is that we can know the exact packets generated time from different sensors. For example, if there are two packets and , which come from and and they have the same sequence number. We can know the exact generated time of these packets by looking into the “CLK” attribute. If the decimal value of ’s and ’s “CLK” attribute are 110 and 111 respectively, then we could say is generated earlier than .

Now, we look deeper into this scenario. If there are two sensors sending their packets to the end device as shown in Figure 1. The first “Time Synchronization” problem will occur if two sensors start their transmission in their local time. For example, starts its transmission 2 seconds earlier than and the sample rates is 90 (packets/s). That is, will only start its transmission after has transmitted 180 packets. Therefore, the number of packets received by the end device from two sensors are not equal. To balance the number of packets received by the end device from the two sensors, we propose a mechanism called “Time Synchronization on Feet”. As illustrated in sub-section b earlier, we have designed a hand-shaking process in our Bluetooth transmission system. If completes its hand-shaking process 2 seconds earlier than , then we “block” . After two seconds are passed, finishes its hand-shaking process and we also “block” . Then, we broadcast a packet from the end device to “notify” and that you are “unblocked”. Finally, both sensors can start their transmission and trigger their clock functions at the standard time of the end device.

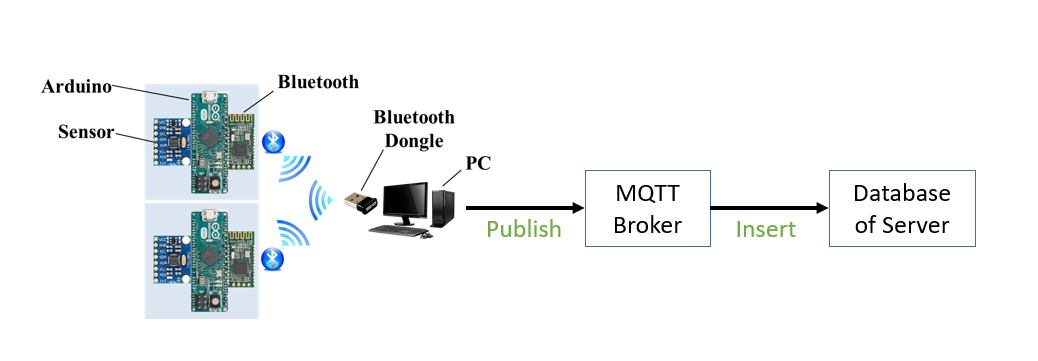
Based on our several experiments, the time difference *T* is between 0.4ms ~ 1ms which is within the range of our accuracy. Since we are using the multi-thread programming on the end device (i.e., two threads can serve different two sensors at the same time), the better way to catch the packets polled by the end device is to create two buffers (Both buffers have the same size which can be adjusted by the user) in different two threads. We call it “Producing” process (Since packets are put into buffers). Second, we take packets from the buffers in the two threads, respectively. We call it “Consuming” process (Since packets are taken from buffers). We must make sure that the “Consuming” rate is greater than the “Producing” rate. Otherwise, the buffer overflow problem will occur according to the queuing theory. Besides, if one thread finds out that there is no packet left when it wants to take packets from its buffer, it will give up this “taken a chance” in its buffer directly. The other thread will check whether the buffer is empty or not immediately.

To illustrate our implementation, for example, if “1” represents the packets taken from the buffer of “” and “2” represents the packets taken from the buffer of “”, then 1212…, 1112…, 1122…, 1211… represent some possible sequence of the taking order from two different buffers with two threads at the end device. By using the “Time Synchronization on Feet” mechanism, we can significantly balance the packets receiving rates at the end device from two sensors. However, if two sensors keep running for one day or one week long, the packets receiving rates will be unbalanced due to the “Clock Drift” effect. The longer the transmission time is, the more effective it will be. We use the clock information in the “CLK” subfield of our packet format (as shown in Figure 2) to test the clock drift effect. Since we know the packets generated time on sensors, we can calculate the accumulated time from the moment the sensor has triggered the clock function and started the transmission.

Therefore, we can know how long the sensor has elapsed at the standard time of the end device. We set the sensor’s delay to 10ms and test 3 times with each time the duration runs for 20min. Both sensors start to transmit packets based on the “Time Synchronization on Feet” mechanism we have mentioned. In average, the discrepancy between two sensors is 400ms within 20min. Since the sensor’s delay is 10ms. We need to do “Time Synchronization on Clock” mechanism per 30sec to control the discrepancy within 10ms. The solution is that we try to broadcast a control message to both sensors from end device every 30sec to retrigger the clock functions and restart the transmission. However, Arduino cannot support multi-thread programming (i.e., it is hard for sensors to transmit packets while receiving packets from the end device simultaneously). Thus, it is very hard to realize the “Time Synchronization on Clock” mechanism.

1. **Uploading the Sensor Data to the Cloud Database Process**

The goal of the cloud platform is to store and backup the 6-axis sensor data from the end device and we can retrieve the sensor data every time. Figure 5 is the architecture of the private cloud platform. The 6-axis sensor data is transmitted from Bluetooth to the end device. Then the end device will send the received data to the cloud database through MQTT and finally, the data is stored into our database. Figure 6 is the architecture of the public cloud platform. We utilize Amazon Web Services (AWS) to upload the received sensor data of end device to Amazon S3, namely Simple Storage Service, which can be simply viewed as a public cloud database. The reason we use Amazon S3 is it provides strong durability, data availability and easy-to-use interface. Amazon S3 is a suitable service for backing up the data.

Figure 5. The architecture of the private cloud platform.

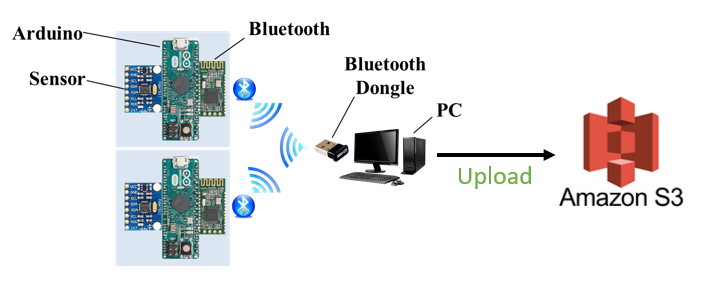


Figure 6. The architecture of the public cloud platform

1. **Private Cloud**

The data format sent from Bluetooth to the end device is shown in Figure 7. The number and the types of sensors can be adjusted according to the user-defined requirements. The counter represents the sequence number of the packet. The Ax, Ay, and Az mean the value of the 3-axis acceleration. The Gx, Gy, and Gz mean the value of the gyroscope. The “!” represents the end of a packet.

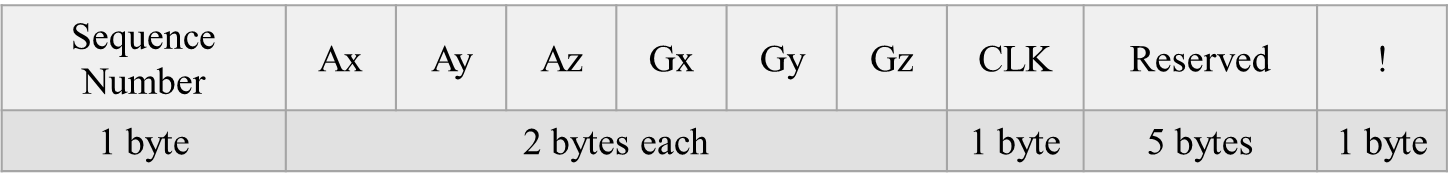


Figure 7. The packet format with the length of 20 bytes.

The data table format is shown in Figure 8. We use MySQL database, which is a relational database management systems (RDBMS) to store and manage a huge volume of data. This is called relational database because all the data is stored into different tables and relations are established using primary keys or other keys.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| BLE number | Sequence number | AX | Ay | Az | Gx | Gy | Gz |

Figure 8. The data table format in the database.

After receiving the sensor data from Bluetooth, the end device will send the received 6-axis sensor data to the cloud database through MQTT (Message Queuing Telemetry Transport) protocol. MQTT is a publish-subscribe-based messaging protocol. MQTT works on top of the TCP/IP protocol. There are three roles in MQTT protocol including publisher, subscriber, and a broker. The publish-subscribe messaging pattern requires a message broker. We describe the detail steps as follows.

First, we built a database for the system and added the data table for the 6-axis sensor data in the database. Set the data format of the column for the data table. There are the BLE number, sequence number, Ax, Ay, Az, Gx, Gy, Gz in the column of the data table. Second, the end device published the received the 6-axis sensor data to the broker. The broker would receive the 6-axis sensor data. Third, the server would check whether the broker has received the 6-axis sensor data. If the broker received the 6-axis sensor data, the server would immediately insert the 6-axis sensor data received by the broker into the database. Last, the 6-axis sensor data would store in the data table. There are the BLE number, sequence number, Ax, Ay, Az, Gx, Gy, Gz in the column of the data table.

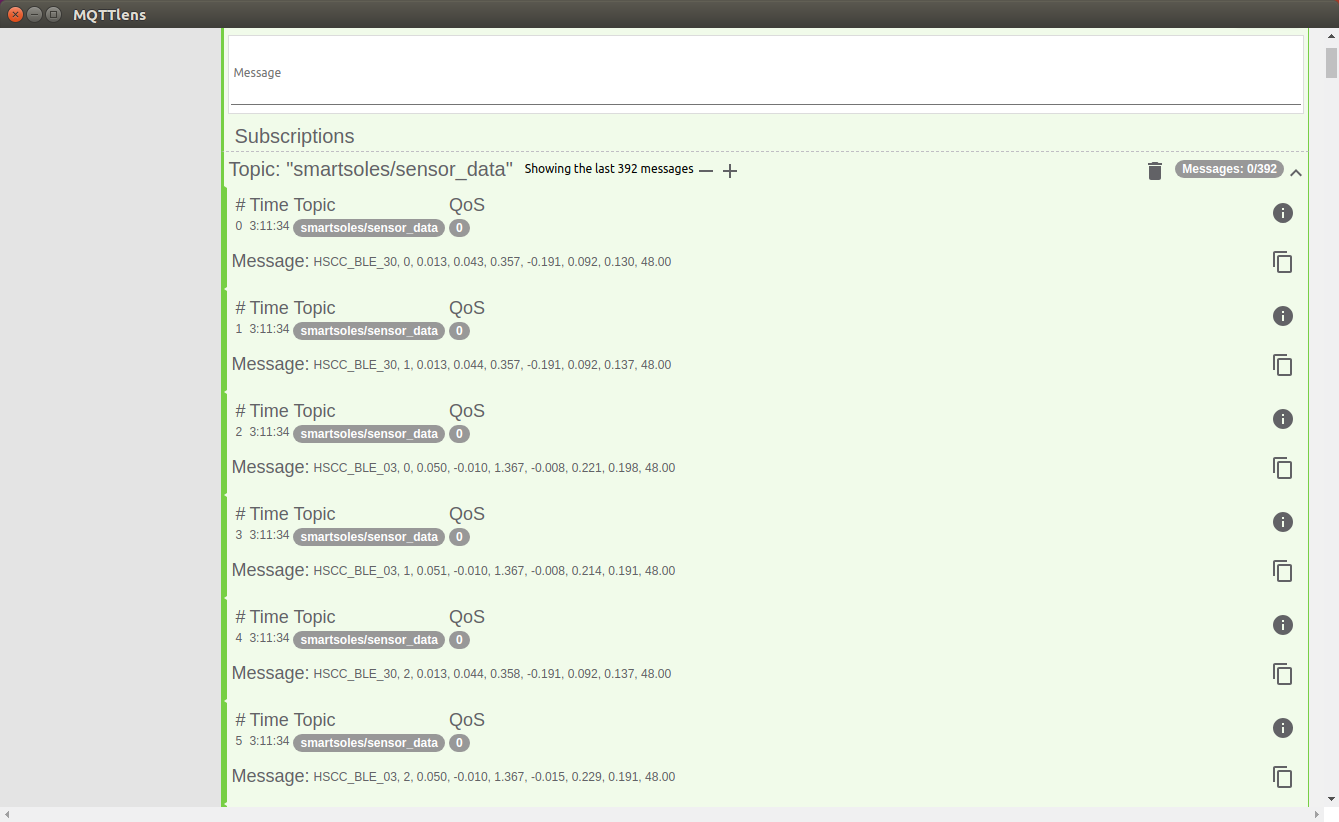
The 6-axis sensor data received by the broker is shown in Figure 9. The received 6-axis sensor data was the same as the transmitted data from the end device.

Figure 9. The data received by the broker.

The 6-axis sensor data of the data table in the cloud database is shown in Figure 10. There are the BLE number, sequence number, Ax, Ay, Az, Gx, Gy, Gz in the column of the data table. The inserted data was the same as the data from the broker.

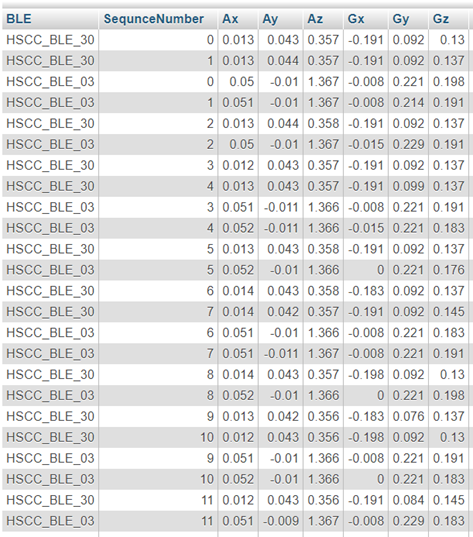
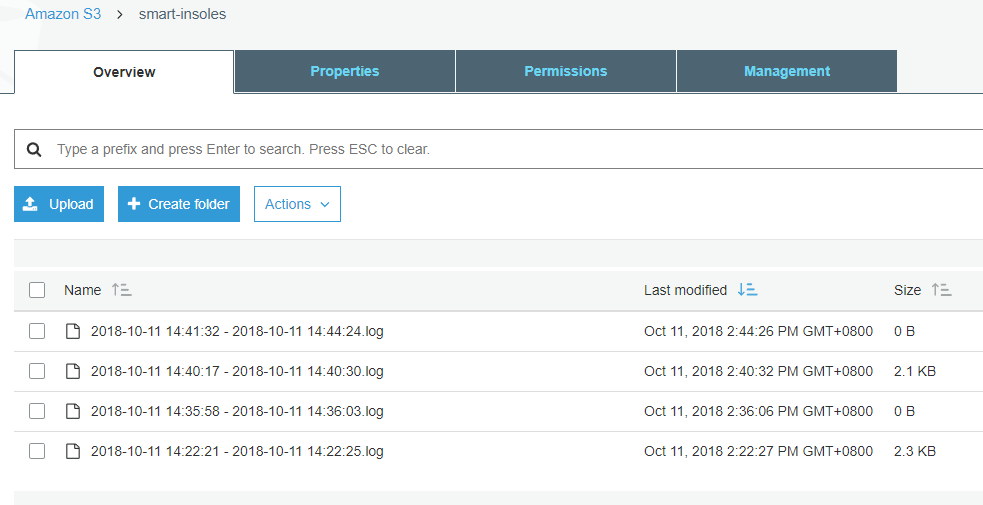


Figure 10. The result of the data table in the cloud database.

1. **Public Cloud**

We uploaded the received 6-axis sensor data to Amazon S3 database through Boto 3. Boto is the Amazon Web Services (AWS) SDK for Python, which allows Python developers to write software with Amazon services like S3. Boto provides an easy-to-use, object-oriented API as well as low-level direct service access. The more detail is as below. First, we built a bucket in Amazon S3 for the system, then we can transmit our sensor data to the bucket. Second, the end device received 6-axis sensor data from the experimental equipment, and these sensor data were written to a file simultaneously. Third, we used a file transfer method to upload the file content received from sensor data to the created bucket. Last, we can download the sensor data through a simple web service interface. The sensor data files we uploaded are shown in Figure 11 and Figure 12.



the files we uploaded using file transfer method

the bucket we created

Figure 11. The files we uploaded to Amazon S3 whose filenames are the experiment time duration.

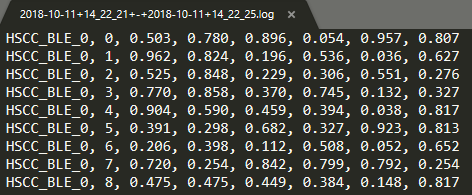


Figure 12. The file we downloaded from S3 and opened it with a text editor.