Real time sensor display using Python

Matthew Oppenheim July 2025

Lancaster University, School of Computing and Communications, United Kingdom

Testing conducted at National Star College, Gloucestershire, United Kingdom

Introduction

In this article I explain how to use the Python programming language to parse and display accelerometer data from a smart watch in real time. The software can save the accelerometer data to a file. I used the software to record accelerometer data from a programmable smart watch while recreating the hand motion of somebody with cerebral palsy. Analysing the recorded data in a Jupyter notebook enabled me to develop an algorithm to identify the hand motion. I ported the algorithm to the smart watch to enable the individual with cerebral palsy to operate environmental controls through hand motion.

I briefly talked about real time display of sensor data using the PyQtGraph library in an earlier article I wrote for Circuit Cellar in 2017[1]. I have improved the code since then and this article covers the project in greater detail. I created a YouTube video showing the software in action alongside the smart watch that produces the sensor data [2]. Illustration 1 shows an example display from the software with x, y, z accelerometer data and two graphs created by processing the accelerometer data. All of these update in real time.

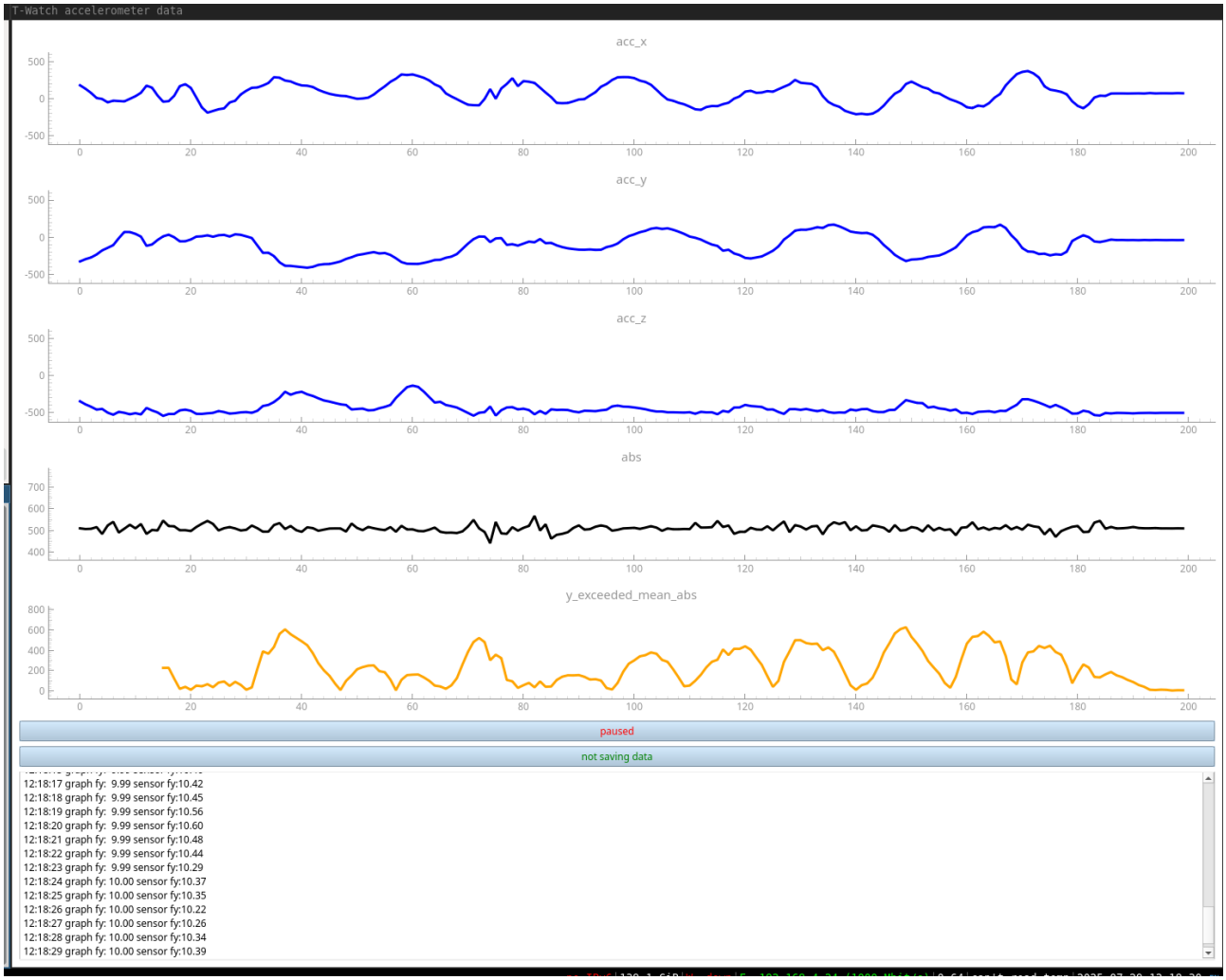

Illustration 1: Real time display of accelerometer and processed data from a Lilygo T-Watch S3

The problem I am trying to solve

I use a Lilygo T-Watch S3[3] programmable smart watch as a part of a system called handshake[4]. Handshake enables people with severe physical disability to control software that creates speech or operate environmental controls by recognising an intentional hand, arm or leg movement The T-Watch S3 contains a Bosch Sensortec BMA423 3-axis accelerometer and an Espressif ESP32-S3 MCU microcontroller along with a smorgasbord of other goodies. I wrote firmware for the ESP32-S3 that processes the accelerometer data to identify intentional hand motion. The ESP32-S3 then generates a radio trigger which is received by a Lilygo T-Embed module that also contains an ESP32-S3 MCU. The receiver module sends a trigger to a device running communication software using a standard switch controller interface. The communication software is known as ‘switchable software’ as it can be controlled with as little as a single switch or controller. There is more information about the receiver unit and how it controls communication software in an earlier article that I wrote for Circuit Cellar[5]. For testing, I use Grid communication software made by Smartbox[6] who kindly gave me a licence to use for this project.

I recently had successful real-world testing at the National Star College in Gloucester, UK with the target user-group by using a simple algorithm for detecting intentional hand motion. However, this simple algorithm failed to work reliably with one of the participants. What to do? Obviously, there

is no one size fits all solution to identifying the different types of hand motions that the participants in my target user-group make.

I needed to analyse the sensor data generated by the T-Watch S3 worn by the individual that my algorithm did not work with. I needed to create a new algorithm based on this data to recognise the intentional motion. How hard could that be?

My solution

The Lilygo T-Watch S3 has a micro-USB socket that is used for loading code and charging the device, see Illustration 2. The same micro-USB socket is used to send and receive serial data from the ESP32 S3 contained in the T-Watch S3. When I connect the T-Watch S3 to my PC using a USB cable, a new serial port appears, which I use to receive serial data from the watch.

The specialist college that I work with supplied a video showing the hand and arm motion of the participant that wants to use the handshake system. The algorithm I first implemented in the firmware on the T-Watch S3 failed to reliably identify this motion. The motion that the candidate makes is a ‘slow swinging’ motion. My initial algorithm was written to identify the ‘fast punchy’ motion that successfully identified the intentional hand movement of several other participants in the target user-group.

I recreated the slow swinging motion shown on the video while streaming accelerometer data from the T-Watch S3 through the serial port. I wrote a Python script that runs on my PC to read and parse the accelerometer data from the serial port to recover the accelerometer x, y and z axis data. The data for each of these axis is displayed real-time using the PyQtGraph[7] library. I use the PyQtGraph library as the library both displays graphs and can generate standard graphical user interface (GUI) widgets such as buttons. I added a button widget to my display to give the option of writing the accelerometer data to a text file.

I used a Jupyter notebook to analyse the accelerometer data that I recorded to file while recreating the ‘slow swinging’ hand movement. To do this analysis, I modified some of the code that I wrote for the real-time display to display the saved data in the Jupyter notebook. I leveraged the Pandas[8] library to apply different algorithms to the accelerometer data and graphed the results alongside the original data. The simple algorithm that I’d written for detecting intentional movement did not clearly identify the ‘slow swinging’ motion. However, I could see that the y-axis accelerometer shows an identifiable pattern as I recreated the motion. Please see Illustration 3 which shows the raw x, y and z accelerometer data as well as a plot of the absolute acceleration created by processing the raw accelerometer data. The red box on this plot shows the area where I recreated the motion that I want the handshake system to trigger on. The y-axis sensor data shows an identifiable feature. I tested a new algorithm written in Python that was successful at recognising this feature. I then added two graphs of processed data to my real time display software. These can be seen as the bottom two graphs in Illustration 1. These plots are used to help identify the target motion.

I ported the new algorithm to the C programming language and added it to the gesture recognition firmware running on the T-Watch S3. When I recreated the target movement, the watch detected the motion and sent a trigger signal to the receiver unit. Now I had a system ready for real-world testing!

Illustration 2: Lilygo T-Watch S3 with USB cable attached.

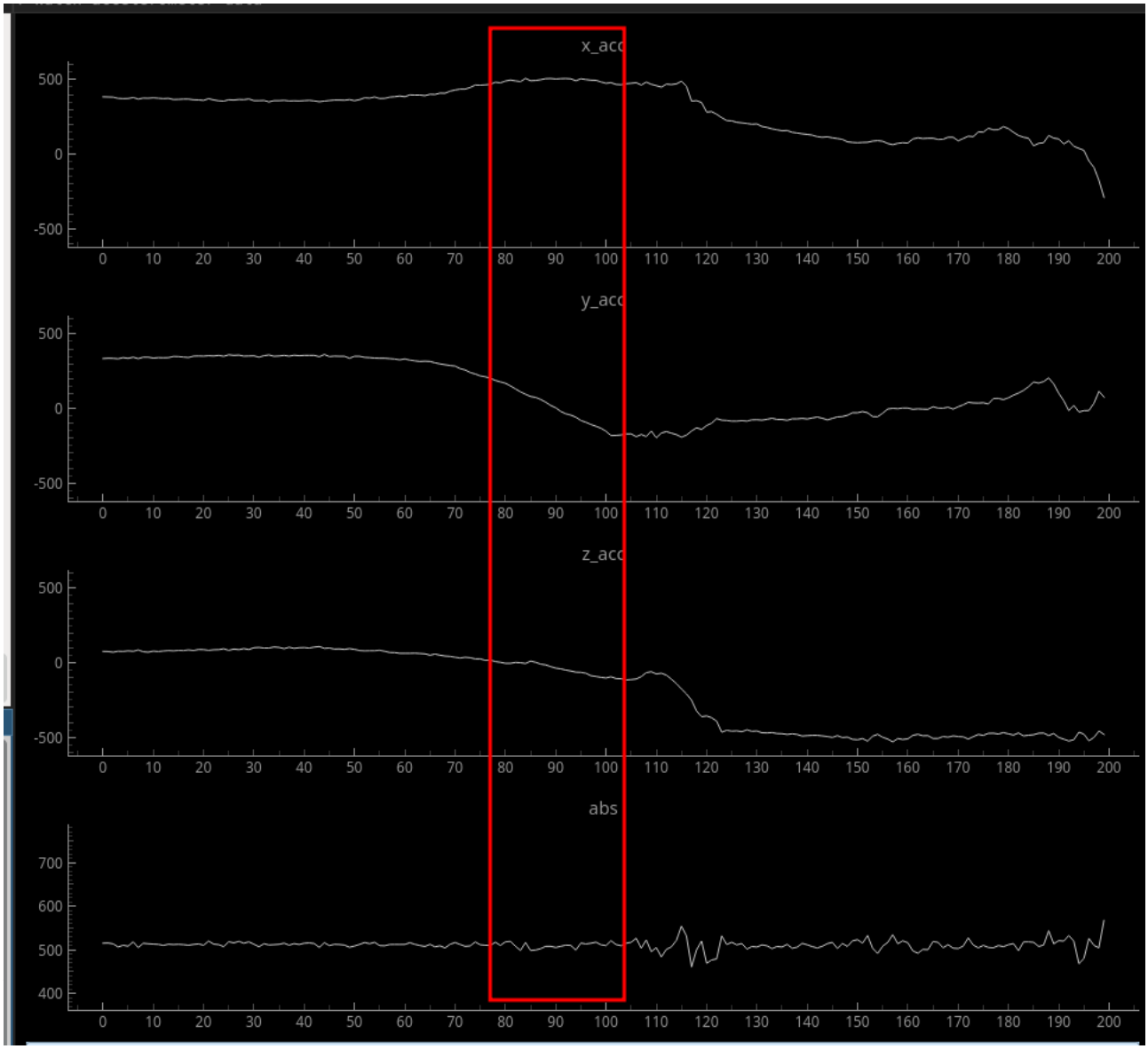

Illustration 3: Real-time and processed accelerometer data from the T-Watch S3, box indicates motion made by the participant.

How to transmit sensor data from the T-Watch S3

I use VSCode with the PlatformIO plug-in and the Arduino environment for firmware development on the ESP32-S3 which is the MCU on the T-Watch S3. **Listing 1** and **Listing 2** show the C-language code used to create and send the accelerometer data to the serial port. In Listing 1 I define a structure called ‘accelerometer’ to store the accelerometer sensor data. The x, y and z values of acceleration are read from the BMA423 accelerometer over I2C and stored in the acc_x, acc_y, acc_z elements of the accelerometer structure. An absolute value of acceleration calculated from all 3-axis can be calculated using the calc_abs function and stored in the acc_abs element of the accelerometer structure.

To write data from to the serial port I leverage the DebugLog library[9]. This library enables messages to be logged with a selectable level of importance. The different levels of importance are known as alert levels. Different alert levels can be set for different messages. I set the alert level for

logging accelerometer data to be at the DEBUG level. In normal use I don't want to stream accelerometer data to the serial port. By setting the alert level to be INFO, the DEBUG logging level is ignored as it is lower than INFO on the alert level hierarchy. I add a millisecond timestamp and a counter to the text string which contains the accelerometer data that is transmitted through the serial port. As we'll see later, the counter is useful to check that no data is missing. The timestamp is used to check the accelerometer sampling rate. The function to read data from the BMA423 accelerometer is supplied by Lilygo in their Lilygolib library. This library class is instantiated in an object called 'watch' in the Lilygolib.h file. I said that I programmed using the C programming language. However, under the Arduino framework this is not strictly true. I access C++ like classes.

Listing 3 is a highly condensed version of my main.cpp file which shows the code used for handling the accelerometer data. I define a structure of type accelerometer called watch_acc. I plan to replace the superloop structure with a simple scheduler. The code is 'good enough' for testing but I'd like to make the architecture more modular to enable activating selected functionality using a configuration file.

To send the accelerometer data as a string to the serial port, I send the memory address where the watch_acc accelerometer structure is stored to the log_acc function using:

```
log_acc(&watch_acc);
```

So long as the logging level is at DEBUG or lower a string of data is constructed and written to the serial port. The next question is, how do we enable the PC at the receiving end of the serial connection to use this string?

```
#ifndef ACCELEROMETER_H
#define ACCELEROMETER_H

#include <cstdint> // needed for int16_t

typedef struct accelerometer {
  // accelerometer x-axis value
  int16_t acc_x;
  // accelerometer y-axis value
  int16_t acc_y;
  // accelerometer z-axis value
  int16_t acc_z;
  // absolute amplitude of combined x,y,z accelerometer values
  int16_t acc_abs;
} accelerometer;

int16_t calc_acc_abs(struct accelerometer *acc);

void log_acc(struct accelerometer *acc);

#endif
```

Listing 1 accelerometer.h

```
#include "accelerometer.h"
// Arduino.h needed for sqrt
#include <Arduino.h>
#include <cstdint> // needed for int16_t
```

```
// must be defined before #include <DebugLog.h>
#define DEBUGLOG_DEFAULT_LOG_LEVEL_DEBUG
#include <DebugLog.h>
// uncomment following line to disable logging
// #include "DebugLogDisable.h"

// acc_abs is the absolute value of all 3 axis
int16_t calc_acc_abs(struct accelerometer *acc){
  int16_t acc_abs;
  // sqrt is in Arduino.h
  acc_abs = sqrt(acc->acc_x * acc->acc_x + acc->acc_y * acc->acc_y + acc->acc_z
* acc->acc_z);
  return acc_abs;
}

// log accelerometer data
void log_acc(struct accelerometer *acc) {
  static uint16_t counter=0;
  unsigned long currentMillis = millis();
  LOG_DEBUG("ST", "m: ", currentMillis, "c: ", counter, "x: ", acc->acc_x, "y:
", acc->acc_y, "z: ", acc->acc_z, "EN", "\r");
  counter += 1;
}
```

Listing 2 accelerometer.cpp

```
#include "accelerometer.h"
#include "LilyGoLib.h"
#define CHECK_INTERVAL_MS 50 // ms interval inbetween checking accelerometer
#define DEBUGLOG_DEFAULT_LOG_LEVEL_DEBUG

struct accelerometer watch_acc;

void setup()
{
  Serial.begin(115200);
  // remove serial timeout lag if no serial output available
  Serial.setTxTimeoutMs(SERIAL_TIMEOUT);
  // watch defined in LilyGoLib.h
  watch.begin();
}

void loop()
{
  // send accelerometer data to serial port
  log_acc(&watch_acc);
  // check accelerometer every CHECK_INTERVAL_MS ms
  if (lastMillis < millis()) {
    // getAccelerometer is defined in SensorBMA423.hpp as:
    // bool getAccelerometer(int16_t &x, int16_t &y, int16_t &z)
    watch.getAccelerometer(watch_acc.acc_x, watch_acc.acc_y, watch_acc.acc_z);
    // update the absolute acceleration from all 3 axis
    watch_acc.acc_abs = calc_acc_abs(&watch_acc);
    lastMillis = millis() + CHECK_INTERVAL_MS;
   }
   delay(CHECK_INTERVAL_MS);
```

```
}
```

Listing 3 snippets from main.cpp

Python real time data display

A Python script runs on the PC that is connected with the T-Watch S3. This script displays a graph that updates real-time showing raw and filtered accelerometer data from the T-Watch S3. The script runs across three threads so that the graph updates smoothly. I split the script across several classes, each one in a separate file.

The main thread is created when the Handshake class is instantiated in main.py. This thread continuously updates a graph with parsed sensor data. Instantiating this class also creates two threads called serial_connection and file_thread. Without using threads, the script would stall when e.g. new data arriving from the T-Watch S3 needed to be processed. This wait would halt the graph from updating, leading to a jerky display.

A diagram showing how the different classes and threads interact is shown in Illustration 4. The serial_connection thread monitors the serial port that the T-Watch S3 is connected to. Any incoming data is checked to ensure that it is valid accelerometer sensor data. The data can become fragmented, so the serial_connection thread assembles the fragments into complete scans. The complete scans of data are made available to the other threads using a publisher/subscriber mechanism. This publisher/subscriber inter-thread communication is handled using the PyDispatcher library. More on this later.

I added the facility to save the parsed data to a text file. This runs in a thread called file_thread. This thread instantiates the File class. A button on the graph GUI is clicked to start the parsed accelerometer data saving to a file. The button is created in the main thread. The status of the button is shared with the file_thread using a queue. The queue is the second method I use for inter-thread communication. The file_thread receives the parsed accelerometer data from the serial_connection thread using the same publisher/subscriber mechanism as the main thread uses, again created using the PyDispatcher library.

The Python code for this project can be found on my GitHub site[10]. The entry point class that starts the show is called Handshake and can be found in the file named main.py.

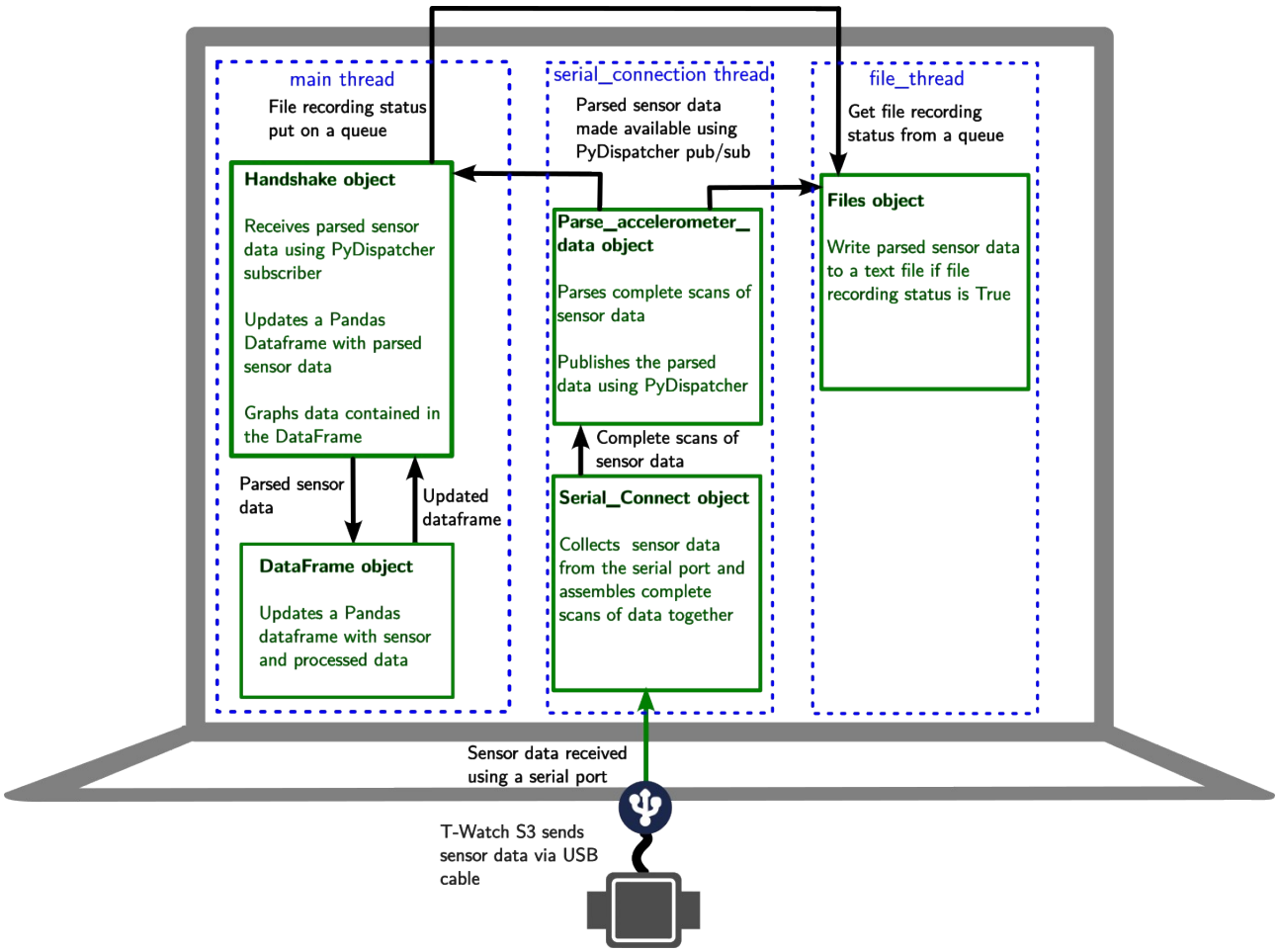

Illustration 4: Project software architecture.

Serial port monitoring thread

The Python threading library is used to create the serial_connection_thread that continuously monitors the serial port. The serial_connection thread is created in this line in main.py in this line:

```
serial_connection_thread = threading.Thread(target=Serial_Connect)
```

This thread collects any data coming over the serial port.

The serial connection is created in the class called serial_connection.py in the lines:

```
try:
        serial_connection = self.serial_connect(serial_port, baud)
        except AttributeError as e:
        utilities.exit_code('no serial connection found, error code:
\n{}'.format(e))
```

This serial_connection object is then passed to the get_bytes method. The get_bytes method continuously tries to read bytes from the serial_connection object.

An example of a complete scan of accelerometer data that is written to the serial port by the T-Watch S3 is:

ST m: 10041 c: 57 x: 228 y: 270 z: -369 EN

I placed the ‘ST’ and ‘EN’ markers to mark the start and end of a scan. The m value is the number of milliseconds since the thread started. The c value is an increasing counter. The x, y, z values are the accelerometer readings. Any bytes read by serial_connection are sent to a parser object. As the sensor data can become garbled or fragmented, the parser object checks for valid data and assembles complete scans of sensor data from fragments. Incomplete scans cause a missing scan message to be displayed. The program discounts the fragmented scan and carries on to try and read the next scan.

Displaying sensor data with PyQtGraph

The Handshake class uses a subscriber created using the PyDispatcher library. The subscriber receives complete scans of accelerometer data as soon as they are published by the serial_thread. The Handshake class instantiates a DataFrame object called self.df. This DataFrame is updated with the new sensor data in the line:

```
self.df = self.dataframe.update_dataframe(message)
```

Where ‘message’ is the sensor data received using the PyDispatcher subscriber.

X, y and z accelerometer data are extracted from the complete scans using regular expressions in the Dataframe object. The x, y and z accelerometer values are then added to a Pandas DataFrame along with any data created by processing these values. Pandas is a popular Python library for data processing. Pandas uses a structure called a DataFrame for storing and processing data. A Pandas DataFrame is like an array except that different columns can have different types. For instance, one column can be a time stamp and the next column could be an integer representing a sensor sample. The number of lines in this DataFrame is limited. When a new row of data is added to the DataFrame, the oldest row of data is removed.

The real-time graph showing the sensor and processed graphs is created using the PyQtGraph library. This runs in the main thread by default. Any plotting objects instantiated using the PyQtGraph class must run in the main thread.

PyQtGraph is based on Qt’s GraphicsView framework. PyQtGraph supports two Python wrappers for this framework, PyQt and PySide. I use PySide6. As with many graphical user interface (GUI) packages, the display is created using several classes. There are several ways we can create plots using the PyQtGraph library. However complex the following code seems, I have reduced the number of layers of widgets I use to create graphs over the last few years!

Illustration 5 shows the arrangement of the different PyQtGraph classes used to create the display. I will explain how I create and display the x-axis accelerometer data. Please see my GitHub site for code that shows how to create and display data from all three accelerometer axis and the processed data created from the x, y and z data.

First of all, I instantiate a GraphicsLayoutWidget. According to the readthedocs documentation for PyQtGraph, GraphicsLayoutWidget is a “convenience class consisting of a GraphicsView with a single GraphicsLayout as its central item”. Using a GraphicsLayoutWidget allows us to have several different graphs display using the same widget. By default, as we add plots, the plots are displayed underneath each other in rows. We can split each row into multiple columns and have graphs displayed alongside each other but I did not do this to keep things simple.

The GraphicsLayoutWidget is instantiated in the line:

```
self.win = pg.GraphicsLayoutWidget(show=True, title='T-Watch accelerometer data')
```

We now add a plot widget to the grid of plots using the addPlot() method. This line adds the plot widget which will contain the x-axis data from the accelerometer to the GraphicsLayoutWidget:

```
self.p_xacc = self.win.addPlot(title='acc_x')
```

However, we still need to add data to the plot widget. This line adds where the data can be found to the self.p_xacc plot widget:

```
self.curve_xacc = self.p_xacc.plot(pen=acc_pen)
```

Each time that we change the data in self.curve_xacc, the plot widget p_xacc will automatically update to display the new data. If we update self.curve_xacc fast enough, the plot appears to scroll smoothly. How we achieve this update speed is the crux of real-time data display. In PyQtGraph we can use the QTimer class to periodically call a method which updates self.curve_xacc. So long as the time between these updates is small enough, we see a smoothly scrolling graph line.

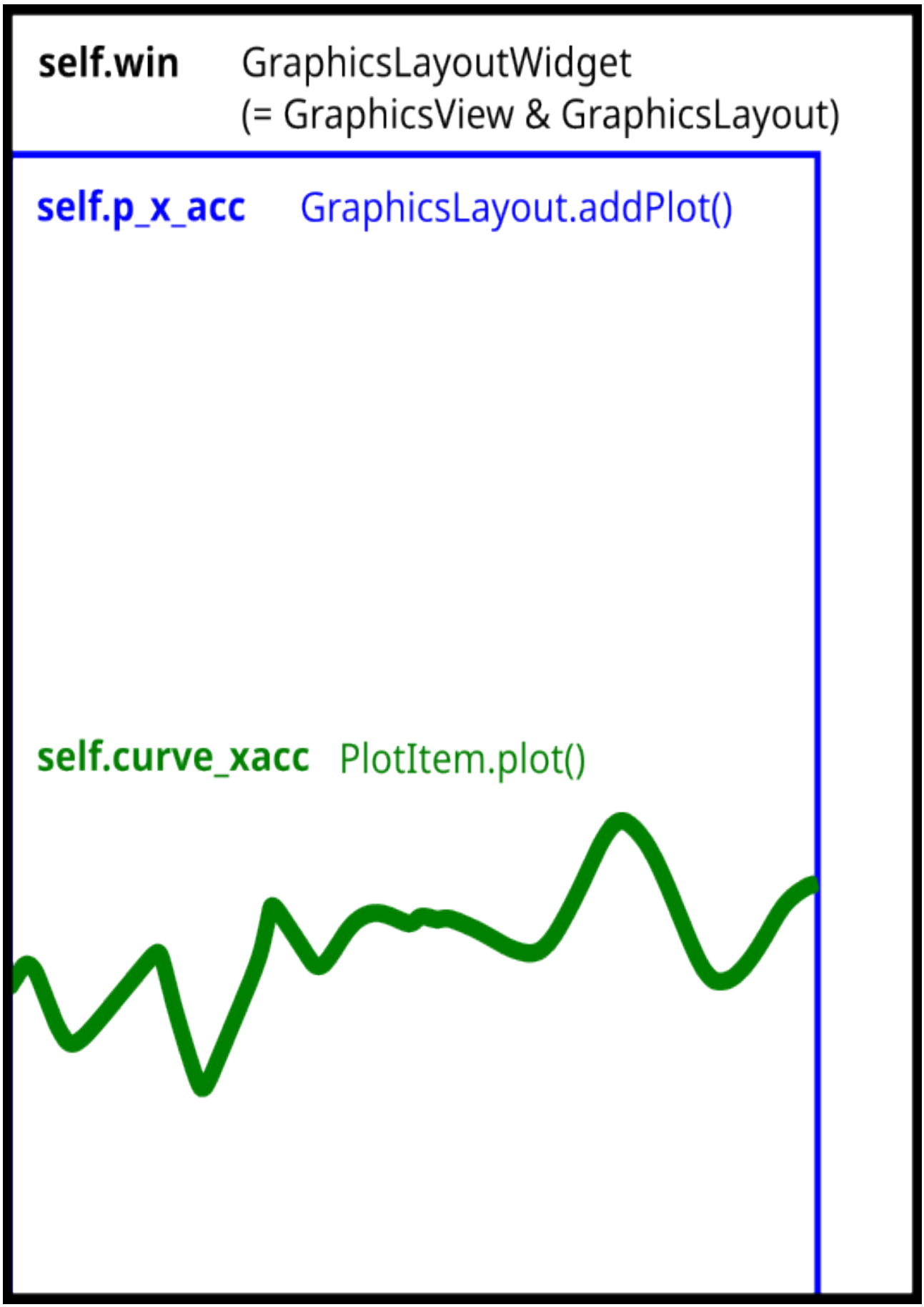

Illustration 5: How the PyQtGraph classes are arranged to display the x-axis accelerometer data.

The class that runs in the main thread and kicks everything else off is called Handshake and is in th script called main.py. This class and the QTimer object are instantiated in these lines:

```
if __name__ == '__main__' :
    handshake = Handshake()

    timer = pg.QtCore.QTimer()
    timer.timeout.connect(handshake.timer_timout)

    # timer units are milliseconds. timer.start(0) to go as fast as practical.
    timer.start(UPDATE_MS) # timer timeout in ms
    pg.exec()
```

The constant UPDATE_MS defines how often the QTimer object runs. The QTimer object runs the timer_timeout method in the Handshake class every UPDATE_MS milliseconds. The timer_timout method calls the method that updates self.curve_xacc in these lines:

```
      acc_x = self.df['acc_x'].to_numpy()
      self.curve_xacc.setData(acc_x)
```

acc_x is updated with the x-axis accelerometer data contained in the Pandas DataFrame. self.curve_xacc is then set to contain this data. The plot widget then updates and display this data. So long as we update the plot at a rate that is at least 12Hz, the graph appears to scroll smoothly. Please note that UPDATE_MS only controls the period that the graph updates at. The rate that the accelerometer is sampled at is controlled by the firmware running on the T-Watch S3. As this is likely to be higher than the rate at which the graph is updated, multiple data points are added to the graph each time that the graph updates.

y-axis and z-axis data are displayed by adding plot widgets to the same GraphicsLayoutWidget that the x-axis data is displayed on and assigning these widgets to contain the relevant sensor data from the Pandas DataFrame.

Saving sensor data to a file

The Handshake class creates a file_thread to enable parsed data to be saved to a text file in the line:

```
        self.file_thread = threading.Thread(target=Files,
args=(self.queue_out,)) # starts in a thread
```

The file_thread instantiates the class File. The code for this is in file.py. Each time that a file is created it is named with a timestamp in the filename so that old data is not over written. Once the file is created, any new sensor data is written to the file. file_thread receives new sensor data by using a subscriber created using the PyDispatcher library. The serial_thread file publishes the parsed sensor data that the file_thread subscribes to.

The file_thread reads a flag that controls if the sensor data is written to file. This flag is read from a queue object. The flag is placed on the queue by the main thread. The flag is toggled True or False whenever a button on the GUI running in the main thread is clicked.

Inter-thread communication

Examples of creating a publisher and a subscriber to sends and receives sensor data between threads using the PyDispatcher library are shown in **Listing 4** and **Listing 5**. By using unique signal and sender names we can set up a publish/subscriber messaging system between threads.

```
  def dispatcher_send_data(self, data):
        ''' Publish data '''
        dispatcher.send(signal=ds.PARSER_SIGNAL, sender=ds.PARSER_SENDER,
message=data)
```
Listing 4 Creating a publisher, from Parse_accelerometer_data.dispatcher_send_data method

```
def dispatcher_receive_data(self, message):
        ''' Received data from dispatcher set up in serial_connection.  '''
        # message is a acc_data structured tuple with data for a single
accelerometer x,y,z scan
        self.df = self.dataframe.update_dataframe(message)
        # update sensor columns to self.df
        self.df = self.imu.update_df(self.df)
        # for debugging:
        self.log_df()
```
Listing 4 Creating a subscriber, from Handshake.dispatcher_receive_data method

Discussion

At the start of this project I wrote C code that ‘should’ trigger on what I imagined the accelerometer data would look like for the gestures that I wanted to recognise. By graphing and processing the sensor data on a PC using Python, I radically reduced the development time of an algorithm that identifies the target hand motion.

The updated algorithm is successful at detecting the ‘slow rolling’ hand motion that I want to recognise. The participant is now successful at operating environmental controls and triggering recorded speech using the handshake system. This is a first step toward enabling the use of more complex communication software. The handshake system is under test with other potential users.

I created the algorithms using traditional signal processing techniques, which is a testament to reading all 100 articles of the Dark Side by Robert Lacoste.

Acknowledgements

Thanks to the team at National Star College, Gloucestershire, United Kingdom for helping with testing handshake and for their many ideas on how to progress with the project.

Resources

[1] Circuit Cellar 319 Full-Stack Python, Matthew Oppenheim

[2] https://youtu.be/m3XazLUtzLc

[3] https://lilygo.cc/products/t-watch-s3

[4] https://mattoppenheim.com/projects/2018/02/handshake/

[5] Circuit Cellar 414, Create Your Own PCBs with a CNC Milling Machine, Matthew Oppenheim

[6] https://thinksmartbox.com/grid/

[7] https://www.pyqtgraph.org

[8] https://pandas.pydata.org

[9] https://github.com/hideakitai/DebugLog

[10] https://github.com/mattoppenheim/realtime_graphing

[11] https://github.com/mcfletch/pydispatcher

Author Profile

Matthew Oppenheim works in marine geophysical survey and offshore construction. He loiters at InfoLab21, Lancaster University when not working at sea. At InfoLab21, Matthew works on assistive technology projects and how to deploy them into the Real World. Visit www.mattoppenheim.com for more details of these projects. Please email matt@mattoppenheim.com with any comments.