Gyroscope tracking 3D-motion via WIFI

Volker Ziemann

Uppsala University, Uppsala, Sweden
Gyroscope
tracking 3D-motion via WIFI

V. Ziemann
Uppsala University, Uppsala, Sweden
May 29, 2017

Abstract
By connecting a MPU-6050 or MPU-9250 accelerometer with built-in angular velocity measurement capabilities to a ESP8266 WIFI dongle it is possible to track motion in three dimensions on any browser connected to a network. This system can be used to visualize for example the motion of rigid bodies in lab courses.

1 Introduction
In our physics lab courses students use various devices to explore the motion of spinning tops or other rigid bodies in three dimensions. Often the devices have some sensor based on a detecting the rotation around some axis by a rotary encoder that produces two on-off signals that are 90 degrees out of phase which permits to determine the sense and speed of rotation. Being an electronics hobbyist I thought that building a somewhat modernized system using a motion-sensor circuit hooked up to a small micro-controller with WIFI capabilities might result in a useful device that permits to track the 3D motion of spinning tops and other things on a web browser. In that way the system is platform-independent and will work on any type of computer.

I built the system around an Invensense motion sensor MPU-6050 [1] and its more capable brethren MPU-9250 [2] which also provides magnetic fields in three dimensions apart from acceleration and angular velocities. As micro-controller I picked the ESP-01, which is the smallest bread-boarded version of the ESP8266 [3] that has built-in WIFI support. In this project the sensor talks to the micro-controller via the I2C bus and the micro-controller talks to the web-browser via websockets. The web-page via which the communication is conducted can either be copied to the computer running the browser directly or served from a web-server. In my system I use a Raspberry Pi as intermediary to serve the web-page.
In the next few sections I will discuss the different subsystems.

## 2 Hardware

The MPU-6050 and MPU-9250 sensors are based on a micro-machined (MEMS) process according to the manufacturers web site. According to the data sheet, internally they measure the variation of the capacitance of a capacitor due to the Coriolis force. My interpretation of this scarce information is that a small mass periodically creates a mechanical imbalance by pushing one of the ‘plates’ of the capacitor resulting in a periodic variation of the capacitance. If the capacitor is part of a resonant $LC$ circuit, the variation will cause a modulation of the voltage across the $LC$ circuit, that is amplified and demodulated, before being digitized by a 16-bit analog-to digital converter (ADC) and placed in a register that is accessible from the I2C bus. The available ranges for the detection of the angular velocity range from $\pm 250$ to $\pm 2000$ degrees per second.

The accelerometer is based on a small mass pressing against capacitors. After an initial calibration the changes in capacitance are measured in a similar way to the one described above and the determined acceleration are placed in another set of registers. The range of detectable accelerations in three dimensions is from $\pm 2\, \text{g}$ to $\pm 16\, \text{g}$.

The MPU-9250 integrates an AK8963 magnetometer [4] that is based on Hall sensors measuring the magnetic induction $B$ in three dimensions within a range of $\pm 4800\, \mu\text{T}$ with either 14 or 16-bit accuracy. In the latter case the resolution is $0.15\, \mu\text{T}$ per bit. By configuring the MPU-9250 appropriately, the magnetometer...
is made accessible on the I2C bus as a separate device with its own I2C address.

The ESP-01 micro-controller is a small breakout board based on the ESP8266 [3]. Internally, the ESP chips sport a 32-bit RISC CPU that normally operates at 80 MHz or 160 MHz and has 64 kB of RAM for instructions and 96 kB for data. There are 16 general-purpose IO (GPIO) pins that can be configured as input or output pin as well as a single analog input with a 10-bit ADC. The controller supports I2C, SPI communication as well as RS-232. On the ESP-01 there are only 8 pins exposed and accessible. Additionally, it has built-in support for WIFI according to IEEE 802.11 b/g/n including authentication with WPA or WEP. Finally, ESP8266 can be programmed very conveniently through the Arduino development system [5], after installing a support package. This combination of powerful features with an easy to use development environment makes it the perfect controller for this project.

The raw chips are very small, as can be seen on Fig. 1. Luckily there are small breakout boards available with the chip already soldered on, while the boards have normal 0.1” spacing for external pins that are connected to the power lines and to the communication bus, here I2C. Thus only four external lines are needed for interfacing: GND, VCC, and the I2C clock SCL, and data line SDA.

As mentioned above the ESP-01 has eight pins exposed, of which four can be used for general applications. Two pins RX and TX, are typically used for serial communication, and two others, GPIO0 and GPIO2, are available. The latter we use for the I2C communication as SCL and SDA lines, respectively.

A prototype system is shown on the left of Fig. 2. A MPU-9250 sensor is visible on the red breakout board on the lower left right next to the blue ESP-01 board with the meander-antenna. I power the system from a Li-ion battery that nominally delivers 3.7 V and use a MCP-1700 low-dropout regulator to produce
3.3 V for the rest of the circuits. There are also a few capacitors visible on the system to buffer the rather high current of over 100 mA that the ESP-01 chip occasionally needs. For the I2C lines I use 3.3 kΩ pull-up resistors, that are located under the ESP-01 board and not visible on Fig. 2. Pin headers for a switch and a LED complete the system. On the right of Fig. 2 we see the circuit enclosed in a 50 mm plastic pipe with 3D-printed end-caps and the switch.

The hardware is rather simple, just connect the few wires and solder a few external components, there is not much to it. Next, we address programming the micro-controller first and later the Javascript on the browser to receive the data sent from the controller.

3 Micro-controller Programming

The task of the micro-controller is to connect to an available WLAN and then listen to a TCP port, I use port 81, and wait for an external process to establish contact via the websocket protocol. The external process typically is a Javascript program, running on a computer connected to the same network. The micro-controller then accepts commands to configure it and to acquire measurement data and send the data to the browser. Once the acquisition is started the controller periodically reads all available sensor data via the I2C bus, normally at 100 Hz and places the data into a sample buffer. Once the buffer is full it sends the contents via the websocket protocol to the controlling browser. Normally this process of acquiring samples and sending them to the browser repeats automatically until a stop request is received. I also prepared a second mode where the sensors are queried at a slower rate and each acquisition is immediately sent to the browser.

In the following I discuss the program that runs on the micro-controller that implements the above requirements.

```cpp
// Gyroscope, V. Ziemann, 170303
// based on ESPWebSock.ino from https://gist.github.com/bbx10/
const char* ssid = "messnetz";
const char* password = "........";
const int port=81;
#include <ESP8266WiFi.h>
#include <ESP8266WiFiMulti.h>
#include <WebSocketsServer.h>
#include <ArduinoJson.h>
ADC_MODE(ADC_VCC);
ESP8266WiFiMulti WiFiMulti;
WebSocketsServer webSocket = WebSocketsServer(port);
#include <ArduinoJson.h>
ADC_MODE(ADC_VCC);
```
At the top of the program that I wrote in the Arduino-IDE I declare the name and password of the wireless network as well as the port number on which the websocket server will listen. These declarations are followed by including the header-files needed for the the WIFI functionality and eventually the server for the websocket named webSocket is instantiated. Finally, I include support for JSON data encapsulation and configure the internal ADC to read the supply voltage.

In the next few lines I define the communication on the I2C bus with the sensors.

```c
#include <Wire.h>
const int MPU6050=0x68;
const int AK8963=0x0C;
uint8_t statusByte=0x00;
float temperature;
unsigned long last=0,last_action=0,goto_sleep=0;
void I2Cwrite(int addr, int reg, int value) {
    Wire.beginTransmission(addr);
    Wire.write(reg);
    Wire.write(value & 0xFF);
    Wire.endTransmission(true);
}
uint8_t I2Cread(int addr, int reg) {
    Wire.beginTransmission(addr);
    Wire.write(reg);
    Wire.endTransmission(false);
    Wire.requestFrom(addr,1);
    delayMicroseconds(5);
    return Wire.read();
}
//.................MPU6050 specific code
#define LED_BUILTIN 16
#define SDA 0
#define SCL 2
void mpu6050_init(int addr) {
    uint8_t b;
    Wire.begin(SDA,SCL);
    b=I2Cread(addr,0x75);
    Serial.print("MPU-9250 WHO_AM_I_byte (should be 0x71) = 0x");
    Serial.println(b,HEX);
    if (0x71 == b) {statusByte |= 0x01;}
    if (0x68 == b) {statusByte |= 0x04;}
}````
I2Cwrite(addr,0x37,0x02); // enable access to magnetometer
b=I2Cread(0x0C,0x00); // check WHOAMI of magnetometer
Serial.print("AK8963 WHO_AM_I_byte (should be 0x48) = 0x");
Serial.println(b,HEX);
if (0x48 == b) {statusByte |= 0x02;}
Serial.print("statusByte = "); Serial.println(statusByte,BIN);
Wire.beginTransmission(addr); // wake up device
Wire.write(0x6B);
Wire.write(0);
Wire.endTransmission(true);

void mpu6050_read_int(int addr, int16_t data[7]) {
    int16_t intval;
    float fval;
    Wire.beginTransmission(addr);
    Wire.write(0x3B);
    Wire.endTransmission(false);
    Wire.requestFrom(addr,14);
    intval=Wire.read()<<8 | Wire.read();
    fval=intval/16.384; data[0]=(int)fval;
    intval=Wire.read()<<8 | Wire.read();
    fval=intval/16.384; data[1]=(int)fval;
    intval=Wire.read()<<8 | Wire.read();
    fval=intval/16.384; data[2]=(int)fval;
    intval=Wire.read()<<8 | Wire.read(); fval=intval/333.87+21.0;
    data[3]=(int)fval; temperature=fval;
    intval=Wire.read()<<8 | Wire.read();
    fval=intval/131.0; data[4]=(int)fval;
    intval=Wire.read()<<8 | Wire.read();
    fval=intval/131.0; data[5]=(int)fval;
    intval=Wire.read()<<8 | Wire.read();
    fval=intval/131.0; data[6]=(int)fval;
    Wire.endTransmission();
}

void getBfield(float B[3]) {
    const float raw2muTesla=4912./32760.0;
    uint8_t data[7];
    int16_t intval;
    uint8_t i=0;
    B[0]=0; B[1]=0; B[2]=0;
    if (!((statusByte>1) & 0x01)) return;
    I2Cwrite(0x0C,0x0A,0x12); // continuous measurements, 16 bit
    while (!(I2Cread(0x0C,0x02) & 0x01)) {} // wait to complete
}
Wire.beginTransmission(0x0C);
Wire.write(0x03); // request 6 bytes from address 3 onwards
Wire.endTransmission(false);
Wire.requestFrom(0x0C,6);
for (i=0;i<6;i++) {data[i]=Wire.read();}
intval=((data[1]<<8) | data[0]); B[1]=intval*raw2muTesla;
}

First up are the I2C addresses of the sensors and service routines that implement reading and writing single bytes. The mpu6050_init() function initializes the I2C bus with the wire.begin() function call where also the used pins are declared. The first reading of the I2C bus determines whether an MPU-6050 is really connected by querying the WHOAMI address. If it is 0x68 the sensor is an MPU-6050, if it is 0x71 it is an MPU-9250. Then we repeat the same test for the magnetometer to determine its presence. The result of the queries is stored in the variable statusbyte. Before leaving the function we enable the sensor.

Obtaining the raw sensor data is accomplished in the mpu_read_int() function. Writing 0x3B to the sensor determines the starting address from where to read. Then requesting 14 consecutive addresses configures the sensor to deliver the contents of 14 consecutive registers, starting at address 0x3B on successive Wire.read() calls. From the data-sheet we know that those registers contain two bytes each for the three acceleration values, the temperature and three rotation sensors. The construction with the shift left operator builds a two-byte unsigned integer from two consecutive bytes. The numerical manipulations are used to convert the raw ADC values to milli-g for the accelerometer, to degrees Celsius for the temperature and to degrees per second for the rotation sensor.

The function getBfield() returns the magnetic measurement values for the $B$ field. After defining some variables the statusbyte is checked for the presence of the magnetometer. If present, the measurement is triggered by writing 0x12 to register 0x0A to the magnetometer at I2C address 0x0C. After waiting for the measurement to complete, writing the starting address 0x0C and requesting 6 consecutive bytes retrieves the raw magnetic measurement values, which we immediately convert to $\mu$T before exiting the function.

Now that we have functions to read the sensors, we need a programming infrastructure to do so periodically at a fixed rate. That is accomplished by the following functions.

```c
#include <Ticker.h>
Ticker SampleFast,SampleSlow,Beacon;
const int npts=500;
volatile uint16_t isamp=0;
```
int sample_period=10,sample_buffer_ready=0,info_available=0;
int data_available=0;
int16_t sample_buffer[npts][6];
volatile uint8_t websock_num=0;
char info_buffer[80];
void beacon_action() { //..........................beacon_action
    if ((millis()-last_action) > 180000) {goto_sleep=1;}
    for (int k=1;k<10;k++) {Serial.println("****************
                                    ****************************I am alive an kicking!");}
}
void sampleslow_action() { //.................sampleslow_action
    data_available=1;
}
void samplefast_action(uint8_t num) { //........samplefast_action
    int16_t data[7];
    mpu6050_read_int(MPU6050,data);
    sample_buffer[isamp][0]=data[0];
    sample_buffer[isamp][1]=data[1];
    sample_buffer[isamp][2]=data[2];
    sample_buffer[isamp][3]=data[4];
    sample_buffer[isamp][4]=data[5];
    sample_buffer[isamp][5]=data[6];
    isamp++;
    if (npts == isamp) {
        SampleFast.detach();
        isamp=0;
        sample_buffer_ready=1;
    }
}

I employ the Ticker library to generate periodic calls to functions and instantiate three Tickers: SampleFast, SampleSlow, and Beacon. Later in the program I attach functions to these Tickers, but those functions I need to define first. The function beacon_action is used to check whether any network action is in progress and sets the variable goto_sleep to unity unless that is the case. It also writes many characters to the Serial output, because the LED lights up as a reminder that the device is turned on. Basically this should remind the user to turn off the device to save the battery since the ESP-01 normally consumes between 80 and 100 mA and empties batteries rather quickly. The variable goto_sleep is tested in the main loop and in case it is non-zero, sends the device to sleep mode to save battery. The sample_slow() function also sets a variable that is checked in the main program and in case it is non-zero starts the acquisition. The reason to defer the sending-to-sleep and the acquisition to the main loop is that sleep
mode did not work when engaged from within and interrupt handler and that sending via websockets did not work entirely reliably from within. I therefore only flag the need to perform some action, but the action itself only takes place in the main function `loop()` that we discuss below.

The `samplefast_action()` function is treated differently. It calls the function `mpu6050_read_int()` to obtain the data from the sensor and then packs it into the `sample_buffer`. Only when the buffer is full, the Ticker is turned off with the call to the `SampleFast.detach()` function, the sample counter `isamp` is reset to zero and the variable `sample_buffer_ready` is set to indicate to the main program to pack the buffers in a JSON package and send it to the browser, but more on that later, when I discuss the `loop()` function further below.

The service functions to orchestrate the communication with the browsers are next in line.

```c
//...................................................................
void sendMSG(char *nam, const char *msg) {
(void) sprintf(info_buffer,"{"%s":"%s"}",nam,msg);
info_available=1;
}
//...................................................................
void webSocketEvent(uint8_t num, WStype_t type, uint8_t * payload, size_t length) {
  Serial.printf("webSocketEvent(%d, %d, ...
", num, type);
  websocket_num=num;
  switch(type) {
    case WStype_DISCONNECTED:
    Serial.printf("[%u] Disconnected!
", num);
    if (10 == sample_period) {
      SampleFast.detach();
    } else {
      SampleSlow.detach();
    }
    sample_period=10;
    break;
    case WStype_CONNECTED:
    {
      IPAddress ip = webSocket.remoteIP(num);
      Serial.printf("[%u] Connected from %d.%d.%d.%d url: %s\r\n", num, ip[0], ip[1], ip[2], ip[3], payload);
      char msg[100]="Connected to sensor";
      if (statusByte & 0x01) {
        strcat(msg," and MPU-9250 installed");
      } else {
        strcat(msg," and MPU-9250 not installed");
      }
      if (((statusByte >> 1) & 0x01) {
        strcat(msg," and MPU-9250 installed")
      } else {
        strcat(msg," and MPU-9250 not installed")
      }
      webSocket.sendJSON(msg, strlen(msg));
    }
```
strcat(msg, ", Magnetometer present");
}
}
if ((statusByte >> 2) & 0x01) {
    strcat(msg, " and plain MPU-6050 installed");
}
char buf[20], out[100];
dtostrf(ESP.getVcc()/1024.0f, 7, 2, buf);
(void) sprintf(out, "{\"%s\":\"%s\"}" , "VCC", buf);
webSocket.sendTXT(websock_num, out, strlen(out));
sendMSG("INFO", msg);
break;
case WStype_TEXT:
{
    Serial.printf("[%u] get Text: %s\r\n", num, payload);
    //........................................parse JSON
DynamicJsonBuffer json_input;
JsonObject& root = json_input.parseObject((const char *)payload);
const char *cmd = root["cmd"];
const int val = root["val"];
if (strstr(cmd,"START")) {
    sample_period=val;
    if (10 == sample_period) {
        SampleFast.attach_ms(sample_period, samplefast_action, num);
    } else {
        SampleSlow.attach_ms(sample_period, sampleslow_action);
        Serial.print("sample_period = ");
        Serial.println(sample_period);
    }
} else if (strstr(cmd,"STOP")) {
    if (10 == sample_period) {
        SampleFast.detach();
    } else {
        SampleSlow.detach();
    }
} else if (strstr(cmd,"CALIBRATE")) {
    sendMSG("INFO","CALIBRATE not yet implemented!");
    // calibrate();
} else if (strstr(cmd,"ACCRANGE")) {
    uint8_t controlbyte=val<<3;
    I2Cwrite(MPU6050, 0x1C, controlbyte);
} else if (strstr(cmd,"GYRORANGE")) {
uint8_t controlbyte=val<<3;
I2Cwrite(MPU6050,0x1B,controlbyte);
} else if (strstr(cmd,"DLPF")) {
    uint8_t controlbyte=val;
    I2Cwrite(MPU6050,0x1A,controlbyte);
} else {
    Serial.println("Unknown command");
    sendMSG("INFO","Unknown command received");
}
break;
}
default:
    Serial.printf("Invalid WStype [%d]\r\n", type);
    break;
}

The first function sendMSG() simply packs the string msg into a JSON string with the name nam and sets the variable info_available that is checked in the main program and triggers sending the INFO message. This turned out to be rather convenient to send various debugging messages, but also other messages, asynchronously with the main task of sending accelerometer data.

The main work-horse of the communication is the function webSocketEvent that is called automatically every time something happens on or with the websocket, such as disconnecting, which is the first case in the switch construction. In that case-branch it simply reports that the browser has disconnected and stops any Ticker events and sets the default period to 10 ms or 100 Hz acquisition rate.

In case the browser initiates the connection the branch with WStype_CONNECTED is executed. First the IP number of the connecting browser is reported on the serial line and then the information in the statusbyte is used to build a string msg that describes the sensors available in the particular system. Just before sending the msg we query the supply voltage on the ESP with the ESP.getVcc() function, properly scale it and send the information in a separate JSON packaged message.

In case the event type is WStype_TEXT a command is received from the browser in the form of a JSON-encoded string. First we unpack the JSON string into the root object, which we subsequently parse for entries cmd and val, for command and value. The subsequent construction with the strstr() function calls checks what command was received. If the command was START the value val is placed in the variable sample_period which is the time between acquisitions. Depending on whether it is 10 ms or longer the Tickers SampleFast or SampleSlow are started with their respective attach_ms() method. The first argument is the time between calls to the function declared as the second argument. The third
argument is passed as an argument to the\texttt{action()} function. If the command is \texttt{STOP} the respective Tickers are detached. \texttt{CALIBRATE} is not yet implemented while \texttt{ACCRANGE} and \texttt{GYRORANGE} extract the desired range from \texttt{val}, shifts it to the proper position in the control byte and sends it to the sensor via the I2C bus. \texttt{DLPF} sets the digital low-pass filter to smooth the values at the expense of a fast response to changes.

The \texttt{default} case is triggered if some unknown command is received. Now we have all the service functions available to initialize the system in the \texttt{setup()} function.

\begin{verbatim}
void setup() { //............................................setup
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);
  Serial.begin(115200);
  last_action = millis();
  Beacon.attach(3, beacon_action);
  delay(1000);
  WiFiMulti.addAP(ssid, password);
  while (WiFiMulti.run() != WL_CONNECTED) {
    Serial.print('.'); delay(500);
  }
  Serial.print("\nConnected to "); Serial.print(ssid);
  Serial.print(" with IP address: ");
  Serial.println(WiFi.localIP());
  webSocket.begin();
  webSocket.onEvent(webSocketEvent);
  mpu6050_init(MPU6050);
  delay(2000);
  digitalWrite(LED_BUILTIN, HIGH);
}
\end{verbatim}

As the first action the LED is defined and the serial line is initialized before starting the \texttt{Beacon} ticker to blink the LED every 3 seconds. After a short waiting time we connect to an access point with the \texttt{addAP()} function that takes the network name and the password as arguments and wait again until the connection is established, whence we print the network name and the IP number that was obtained via DHCP. Once the network is up and running we start the websocket server and register the event-handler \texttt{webSocketEvent} before initializing the sensor.

At this point we have a controller that is connected to the network and has the sensor initialized. Thus we are ready discuss the main function \texttt{loop()} that runs continuously and responds to internal triggers from the Ticker and to external triggers via the websocket. How that is done we'll cover next.
void loop() { //...........................................loop
    webSocket.loop();
    if (sample_buffer_ready) {
        last_action=millis();
        digitalWrite(LED_BUILTIN,LOW);
        sample_buffer_ready=0;
        char out[2570];
        int points_to_send=npts;
        for (int i=0;i<6;i++) {
            char chstr[6];
            sprintf(chstr,"WF%d",i);
            DynamicJsonBuffer jsonBuffer;
            JsonObject& json = jsonBuffer.createObject();
            JsonArray& ddd = json.createNestedArray(chstr);
            for (int k=0;k<points_to_send;k++) {
                ddd.add(sample_buffer[k][i]);
                yield();
            }
            json.printTo(out,sizeof(out));
            webSocket.sendTXT(websock_num,out,strlen(out));
        }
        digitalWrite(LED_BUILTIN,HIGH);
        char buf[20];
        dtostrf(temperature,7,1,buf);
        sendMSG("TEMP",buf);
        float B[3];
        getBfield(B);
        Serial.print("Bfield/uT = ");
        Serial.print(B[0]);
        Serial.print("\t");
        Serial.print(B[1]);
        Serial.print("\t");
        Serial.println(B[2]);
        {
            DynamicJsonBuffer jsonBuffer;
            JsonObject& json = jsonBuffer.createObject();
            JsonArray& ddd = json.createNestedArray("BFIELD");
            ddd.add(double_with_n_digits(B[0],2));
            ddd.add(double_with_n_digits(B[1],2));
            ddd.add(double_with_n_digits(B[2],2));
            json.printTo(out,sizeof(out));
            webSocket.sendTXT(websock_num,out,strlen(out));
        }
    }
    if (goto_sleep==1) {
        Beacon.detach();
    }
}
delay(100);
while (1) {ESP.deepSleep(1000000); delay(100);}
if (info_available==1) {
  info_available=0;
  webSocket.sendTXT(websock_num,info_buffer,strlen(info_buffer));
}
if (data_available==1) {
  last_action=millis();
  data_available=0;
  int16_t data[7];
  mpu6050_read_int(MPU6050,data);
  float B[3];
  getBfield(B);
  DynamicJsonBuffer jsonBuffer;
  JsonObject& json = jsonBuffer.createObject();
  JsonArray& ddd = json.createNestedArray("ALLDATA");
  for (int i=0;i<7;i++) { ddd.add(data[i]);}
  ddd.add(double_with_n_digits(temperature,1));
  ddd.add(double_with_n_digits(B[0],2));
  ddd.add(double_with_n_digits(B[1],2));
  ddd.add(double_with_n_digits(B[2],2));
  char out[200];
  json.printTo(out,sizeof(out));
  webSocket.sendTXT(websock_num,out,strlen(out));
  if ((millis()-last) > 5000) {
    char buf[20];
    dtostrf(temperature,7,1,buf);
    (void) sprintf(out,"{%s":"%s"}","TEMP",buf);
    webSocket.sendTXT(websock_num,out,strlen(out));
    dtostrf(ESP.getVcc()/1024.0f,7,2,buf);
    (void) sprintf(out,"{%s":"%s"}","VCC",buf);
    webSocket.sendTXT(websock_num,out,strlen(out));
    last=millis();
  }
}

The first command in the loop() function is the call to webSocket.loop() which handles anything related to the websocket, such as placing the variables received from the network buffers into user-space and triggering the event-handler for the websocket events. What follows is a sequence of checks whether certain variables are non-zero. First out is the variable sample_buffer_ready that was
set in the samplefast_action() function indicating that the required number of samples are ready to be packaged and sent. Since we have six wave forms, three accelerometer and three rotation sensors, we loop over those, package them into a JSON buffer named WF0 to WF5 and send them one at a time to the browser with the call to websocket.sendTXT(). The ample use of the yield() function grants a little time to the background processes for network and other activities on the controller. Next we package the temperature in a JSON package with the name TEMP before obtaining the magnetic field value and packaging it into a JSON package with name BFIELD. This branch thus handles the fast acquisition at 100 Hz but only reads the accelerometer and rotation sensor at that rate. The temperature and magnetic field are only sent to the browser once at the end of the acquisition.

The next variable to be tested in the loop() function is goto_sleep which activates the deepSleep() function for 1000000 microseconds or one second. In this mode the system stops responding and power needs to be cycled to turn it on again. It does, however, use considerably less power compared to normal idling. If info_available is non-zero the contents of the variable info_buffer is sent via the websocket to the controlling browser.

In case the variable data_available is non-zero, which happens in the sample slow_action() function, the sensor is read with the call to the mpu6050_read_int() and the getBfield() functions and a JSON package with the name ALLDATA is prepared that sends one set of measurements to the browser. Furthermore, in case that 5 seconds have elapsed since the last time, the temperature and voltage Vcc are sent to the browser.

This completes the program that runs on the ESP-01 micro-controller and we now have to address the controlling program in the browser that uses Javascript to communicate with the controller.

4 Javascript in Browser

The interaction between the browser and the micro-controller is most easily described with the help of a screen shot of the user interface, which can be seen in Fig. 3. The interface is based on a web page constructed using HTML5, Javascript and SVG. It can either be saved as a single file to a computer or retrieved from a web server. Connecting to the micro-controller is accomplished by giving the web page address an argument, separated by a question mark, specifying the IP and port number of the controller discussed in the previous section. An example is the following line.

```javascript
.../gyroscope-v2.html?ip=192.168.20.194:81
```

which assumes that the micro-controller with the sensor has IP number 192.168.20.194 and listens on port 81, just as we defined in the previous section. In Fig. 3
Figure 3: The user interface running in a browser window.
I use the address ip=192.168.2.146:20194 which is directed by a Raspberry Pi to the controller by a simple port-forwarding rule.

Once we open the web page we see the web page shown in Fig. 3 in the browser window. In the first row at the top the name and version of the program is reported, followed by buttons to start and stop the acquisition. To the right of the stop button pull-down menus are located to select the range of the accelerometer from ±2 g to ±16 g, the range of the rotation-sensor from ±250 to ±2000 degrees per second, the bandwidth of a low-pass filter to smooth the curves, the sample rate, which can be either 500 samples acquired at 100 Hz or variable time between acquisitions, ranging from 20 ms to 2 seconds. In the latter cases the display is updated one sample at a time, which is similar to the roll-mode of oscilloscopes. The variable MagMult can be used to increase the vertical scale of the magnetic field display. The Calibrate button is not implemented at this time, and the DataLogger writes the data from the accelerometer and rotation sensor in tabular form, separated by commas, at the bottom of the webpage such that it can be easily copied to a spreadsheet program.

The second row contains a number of button to turn on and off the visibility of the traces in the windows that are visible below. The three dimensions of the acceleration, rotation speed, an magnetic field are color-coded in red, green, and blue with the absolute value coded in black. Pressing the buttons simply makes the trace visible or not. This permits to focus on one variable at a time and removes the clutter from the display. The upper display shows the acceleration, the next the rotation speed, and below that the magnetic field (not shown in Fig. 3). The small arrows next to the text above the graph can be used to make the graph visible or to hide it.

Below the graphs the system reports when the last data sample was acquired and the IP number of the connected controller. The status line has a slightly darker background color and is used to report status changes and info about the most recently pressed button. At startup information about the attached sensor is reported and whether a magnetometer is present. Also the temperature of the sensor is reported at the right-most position in the status line. As last item logging information, such as that from the DataLogger button, is presented at the bottom of the page.

This describes the functionality of the controlling web page on the browser, but as the next topic we need to discuss how to implement it. As a matter of fact the entire functionality is available as a single file containing HTML, Javascript and SVG commands.

```html
<!DOCTYPE HTML>
<html lang="en">
<head>
  <title>GyroScope V2</title>
  <meta charset="UTF-8">
  <style>
```

18
At the head of the file we first declare the DOCTYPE to be HTML which implicitly defines that the contents follows the HTML5 rules. After starting the HTML section we give the page a title, define the character set, and define CSS style properties of the entities that appear on the web page. #accelerometer is the top window with the accelerometer data and defines that it should be enclosed in a black border, similar for the rotation and magnetic data. Then we define the colors of the traces for the respective data traces and the color of the button to toggle the visibility of respective trace with the same color. Finally we define that the IP number and temperature data displayed in the status lines should appear flush-right.

After these initial declarations that determine the appearance of things visible (or not) on the web page we need to define the user-interface with the buttons that we later hook up to Javascript functions to provide the interactivity with the controller.
<OPTION value="1">500 Hz</OPTION>
<OPTION value="2">1000 Hz</OPTION>
<OPTION value="3">2000 Hz</OPTION>
</SELECT>

SELECT onchange="setDLPF(this.value);">
<OPTION value="0" selected="selected">BW=260 Hz</OPTION>
<OPTION value="1">BW=184 Hz</OPTION>
<OPTION value="2">BW=94 Hz</OPTION>
<OPTION value="3">BW=44 Hz</OPTION>
</SELECT>

SELECT onchange="setSamplePeriod(this.value);">
<OPTION value="10" selected="selected">500 samples @ 100 Hz</OPTION>
<OPTION value="20">Roll-mode with 20 ms</OPTION>
<OPTION value="50">Roll-mode with 50 ms</OPTION>
<OPTION value="100">Roll-mode with 100 ms</OPTION>
<OPTION value="500">Roll-mode with 0.5 s</OPTION>
<OPTION value="1000">Roll-mode with 1 s</OPTION>
<OPTION value="2000">Roll-mode with 2 s</OPTION>
</SELECT>

SELECT onchange="setMagnetMultiplier(this.value);">
<OPTION value="0.1">MagMult=0.1</OPTION>
<OPTION value="0.3">MagMult=0.3</OPTION>
<OPTION value="1" selected="selected">MagMult=1</OPTION>
<OPTION value="3">MagMult=3</OPTION>
<OPTION value="10">MagMult=10</OPTION>
</SELECT>
<button id="calibrate" type="button" onclick="calibrate();">Calibrate</button>
<button id="logger" type="button" onclick="datalogger();">DataLogger</button>

Toggle trace visibility:
<button id="toggleAx" type="button" onclick="toggletrace('trace0');">AccX</button>
<button id="toggleAy" type="button" onclick="toggletrace('trace1');">AccY</button>
<button id="toggleAz" type="button" onclick="toggletrace('trace2');">AccZ</button>
<button id="accTot" type="button" onclick="accTotal();">AccTotal</button>
<button id="togglePhix" type="button" onclick="toggletrace('trace3');">PhiX</button>
<button id="togglePhiy" type="button" onclick="toggletrace('trace4');">PhiY</button>
In the above HTML code we define the buttons in the top two rows of the web page. The normal procedure is to define each button in <button> and </button> tags that enclose the visible text on the button. In the first tag we give the button a name with the id= declaration, declare the type of button, and hook it to a Javascript function with the onclick= declaration. The first two buttons are there to start and stop the acquisition. The next button is the selection menu for the acceleration range that is declared between the <SELECT> and </SELECT> tags. The Javascript function that is called at the onchange event takes the value and the text in the respective option as argument. It will send the values to the controller where it will be used to set the range of the accelerometer. The next four <SELECT> tags are used to communicate the range of the rotation sensor, the bandwidth of the low-pass filter, the sampling speed, and the multiplier for the magnetic display, respectively. The last two <button> tags trigger the Javascript functions specified in the onclick= directive. Note that all buttons and select statements are enclosed in <P> tags to specify that they should be grouped together. If the window is wide enough they populate one row.

The contents of the second row with the visibility buttons for the individual traces is defined in the next <P> and </P> block. Note that all buttons have their own id= which makes it possible to refer to them from other places in the code. At the end a <label> is defined that will display the measured supply voltage from he controller.

The above declarations take care of the upper rows of buttons. Next we need to discuss the areas where the data is plotted.
The plot areas are enclosed in `<details>` tags which makes them individually collapsible. The text that is visible with the small black arrowhead to the left is enclosed in `<summary>` tags and the area where the data is plotted is enclosed in `<svg>` tags. Note that the `id=` and the size of the area is defined in the opening `<svg>` tag. Within the tags four traces, or `path` in SVG lingo, are defined, one for each dimension and one for the total. The starting point of each path is already declared by the statement `d="M0 200"` which means that the painting cursor must move-to point (0,200). We later add additional points with statements such as `L2 250`, which draws a line-to point (2,250). The first `<details>` section declares the drawing area for the acceleration, and there are two more for the rotation sensor and the magnetometer. The first two are `open` at the start and the last one lacks that statement and is collapsed at the start.
Below the definitions needed for the drawing areas we place several `<div>` tags for elements such as the status line and the logging data. They contain some initial explanatory text, but that will be overwritten, as soon as contact is established to the controller. Information sent via the websocket is routed to the appropriate locations on the web page using Javascript, because each widget has its own name and can be addressed individually. That is what we cover next.

```javascript
var counter = 0;
var val0;
var ipaddr='192.168.20.135:81';
var step = 2;
var acc_multiplier=1;
var gyro_multiplier=1;
var magnet_multiplier=1;
var current_position=0;
var sample_period=10;
var getQueryString= function (field) {
  var reg = new RegExp("[?&"] + field + "=([^&#]*)", "i");
  var string=reg.exec(window.location.href);
  return string ? string[1] : null;
};
iii=getQueryString("ip"); // address in query string?
if (iii != null) {ipaddr=iii;}
document.getElementById('ip').innerHTML=ipaddr;
var websock = new WebSocket('ws://'+ ipaddr);
websock.onopen = function(evt) {console.log('websock open');
websock.onclose = function(evt) {
  console.log('websock close');
toStatus('websock close');
};
websock.onerror = function(evt) {
  console.log(evt); toStatus(evt)
};
```

All Javascript code needs to be enclosed in `<script>` tags where the opening tag is right at the top of the previous code segment. Following it a number of variables are declared and a function `getQueryString` that performs some regular expression magic to extract a particular variable following a question-mark in the line that is used to retrieve the web page. In that way we find the IP-address and the port number in the variable `iii` of the controller. If that is not declared the default address stored in the variable `ipaddr` is used. Now that we know the IP address of the controller we display it in the tag that has the `id='ip'` and that is accomplished by the line `document.getElementById(ip).innerHTML=`
which searches the document to find a widget with the specified \texttt{id} and changes the displayed text, which is called \texttt{innerHTML} by the string containing the IP number.

After having an IP and port number we are ready to open the websocket connection called \texttt{websocket} which is done in the next line. We only need to specify the protocol type \texttt{ws://} and the IP and port number. All the necessary hand-shaking is done behind the scenes completely automatically. If all went well, we have established a two-way communication channel between the web-browser and the controller. The controller should report on its serial line by virtue of the case statement with \texttt{WType\textunderscore CONNECTED} that a connection is up and running. It will also immediately send some information back to the browser about the installed hardware. In order to handle that in the browser we need to declare event handlers for different things that can happen to the websocket. The first event is opening the websocket and is called \texttt{websocket\_onopen}. In the attached function we simply log that to the \texttt{console.log}. The next even is \texttt{onclose} which also causes a log entry and '\texttt{websocket close}' written to the status display using the \texttt{toStatus()} function that is declared further down in the file. The \texttt{onerror} event is treated in much the same way.

The event handlers we just discussed are more of the housekeeping type. The event that is triggered by the controller sending a message is treated in the \texttt{onmessage} event handler that we discuss in the following.

```javascript
//...................................................onmessage
websocket.onmessage=function(event) {
  var stuff=JSON.parse(event.data);

  //..............................................info
  var val=stuff['INFO'];
  if ( val != undefined ) {
    toStatus(val);
  }

  //..............................................ALLDATA
  var val=stuff['ALLDATA'];
  if ( val != undefined ) {
    //.............................acceleration
    dd=document.getElementById('trace0').getAttribute('d');
    dd += ' L ' + current_position + ' ' + (200-val[0]/10);
    document.getElementById('trace0').setAttribute('d',dd);
    dd=document.getElementById('trace1').getAttribute('d');
    dd += ' L ' + current_position + ' ' + (200-val[1]/10);
    document.getElementById('trace1').setAttribute('d',dd);
    dd=document.getElementById('trace2').getAttribute('d');
    dd += ' L ' + current_position + ' ' + (200-val[2]/10);
    document.getElementById('trace2').setAttribute('d',dd);

    //.............................angular velocity
    dd=document.getElementById('trace3').getAttribute('d');
    ```
dd += ' L' + current_position + ' ' + (250-val[4]);
document.getElementById('trace3').setAttribute('d',dd);
document.getElementById('trace4').setAttribute('d',dd);
dd += ' L' + current_position + ' ' + (250-val[5]);
document.getElementById('trace5').setAttribute('d',dd);
document.getElementById('trace5').setAttribute('d',dd);

//.............................magnetic field
dd=document.getElementById('traceB0').getAttribute('d');
dd += ' L' + current_position + ' ' + (250-val[8]*magnet_multiplier);
document.getElementById('traceB0').setAttribute('d',dd);
document.getElementById('traceB1').setAttribute('d',dd);
document.getElementById('traceB2').setAttribute('d',dd);
document.getElementById('traceBtot').setAttribute('d',dd);

current_position += 1;
if (current_position > 1000) {
  document.getElementById('trace0').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('trace1').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('trace2').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('trace3').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('trace4').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('trace5').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('traceB0').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('traceB1').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
document.getElementById('traceB2').setAttribute('transform','translate(' + (1000-current_position) + ',0)');
}
('transform','translate(' + (1000-current_position) + ',0'));
document.getElementById('traceBtot').setAttribute
('transform','translate(' + (1000-current_position) + ',0'));
}
}
//..............................................Bfield
var val=stuff['BFIELD'];
if ( val != undefined ) {

}
//..............................................temperature
var val=stuff['VCC'];
if ( val != undefined ) {

document.getElementById('VCC').innerHTML="V<sub>cc</sub>= " + val + " V";
}
//..............................................temperature
var val=stuff['TEMP'];
if ( val != undefined ) {

document.getElementById('temperature').innerHTML="Temp= " + val + " °C";
}
//...........................................trace 0
var val=stuff['WF0'];
if ( val != undefined ) {

document.getElementById('trace6').setAttribute('d','M0 0');
document.getElementById('trace6').setAttribute('visibility','null');
var path = document.getElementById('trace0');
var d =path.getAttribute('d');
d="M0 200"; step=1000/val.length;
for (i=0; i<val.length; i++) {
	d += ' L' + (step*i) + ' ' + (200-val[i]/10);
}
path.setAttribute('d',d);
val0=val;
}
//...........................................trace 1
var val=stuff['WF1'];
if ( val != undefined ) {

var path = document.getElementById('trace1');
var d =path.getAttribute('d');
d="M0 200"; step=1000/val.length;
for (i=0; i<val.length; i++) {
	d += ' L' + (step*i) + ' ' + (200-val[i]/10);
path.setAttribute('d',d);
val1=val;
}

//...........................................trace 2
var val=stuff['WF2'];
if ( val != undefined ) {
    var path = document.getElementById('trace2');
    var d =path.getAttribute('d');
    d="M0 200"; step=1000/val.length;
    for (i=0; i<val.length; i++) {
        d += ' L' + (step*i) + ' ' + (200-val[i]/10);
    }
    path.setAttribute('d',d);
    val2=val;
}

//...........................................trace 3
var val=stuff['WF3'];
if ( val != undefined ) {
    document.getElementById('trace7').setAttribute('d','M0 0');
    document.getElementById('trace7').setAttribute('visibility','null');
    var path = document.getElementById('trace3');
    var d =path.getAttribute('d');
    d="M0 250"; step=1000/val.length;
    for (i=0; i<val.length; i++) {
        d += ' L' + (step*i) + ' ' + (250-val[i]);
    }
    path.setAttribute('d',d);
    val3=val;
}

//...........................................trace 4
var val=stuff['WF4'];
if ( val != undefined ) {
    var path = document.getElementById('trace4');
    var d =path.getAttribute('d');
    d="M0 250"; step=1000/val.length;
    for (i=0; i<val.length; i++) {
        d += ' L' + (step*i) + ' ' + (250-val[i]);
    }
    path.setAttribute('d',d);
    val4=val;
}

//...........................................trace 5
var val=stuff['WF5'];
if ( val != undefined ) {
The first thing we do in the function that treats the onmessage event is to parse the data, that we agreed upon to be in JSON form and place the data in a dictionary that we call stuff. It is a dictionary in the sense that we access an entry with the name "INFO" simply through stuff["INFO"] and that is immediately used in the following lines where we check whether there is actually an INFO stanza in the received message and if that is the case, we print the contents, or the value, of that stanza to the status line.

The next name we check is ALLDATA which indicates that the message contains all sensor values from a single time instance when slow acquisition is in progress. Check the construction of the JSON package in the code that runs on the micro-controller for the ALLDATA package. If such a stanza was present in the message we obtain a handle dd to the trace information, which is contained in d= of the SVG statements with the getAttribute() function, add a line segment to point (x,y) with the L x y and write the updated path dd back with the setAttribute() function. This we repeat for all traces for acceleration, rotation speed and magnetic data. In the end we need to check whether we reached the end of the display and if that is the case we simply translate the trace to the left such that it rolls off to the left.

Next we check whether BFIELD data is inclosed in the message and display it in the log window. In case the supply voltage is available, indicated by the presence of a VCC entry we write it to the HTML tag with name VCC. Temperature data, indicated by TEMP receives the same treatment.

In case the message contained waveform data, as indicated by WF0 to WF5, we know that this is a full waveform from the fast acquisition. The main task is to assemble the SVG path d= for the displayed trace and writing it to the proper trace data with the setAttribute() function. For WF0 and WF3 we also initialize the black traces for the total acceleration and rotation speed.

At the end of the onmessage handler we update the text displayed on the Start and Stop buttons as well as the time stamp for the last update. The above
routines define how the web page reacts to messages from the controller. Next we need to address how the interaction with the user who clicks on the buttons is taken care of by defining the actions that happen when a button is clicked.

```javascript
function start() {
    websock.send(JSON.stringify(
        {"cmd" : "START", "val" : sample_period }));
    document.getElementById('stop').innerHTML="Stop me";
    document.getElementById('start').innerHTML="Running";
    document.getElementById("log").innerHTML="";
}
function stop() {
    websock.send(JSON.stringify({ "cmd" : "STOP", "val" : "-1" }));
    document.getElementById('stop').innerHTML="Stopped";
    document.getElementById('start').innerHTML="Start me";
}
function toStatus(txt) {
    document.getElementById('status').innerHTML=txt;
}
function toLogWin(txt) {
    document.getElementById('log').innerHTML=txt;
}
function calibrate(v) {
    toStatus("Calibrating");
    websock.send(JSON.stringify({ "cmd" : "CALIBRATE", "val" : v }));
}
function datalogger(v) {
    logwindow=document.getElementById("log");
    logwindow.innerHTML="#Time[s], AccX[mg], AccY[mg], AccZ[mg], PhiX[\degree/s], PhiY[\degree/s], PhiZ[\degree/s]<BR>");
    for (i=0; i<val0.length; i++) {
        logwindow.innerHTML += (i*0.01).toPrecision(4) + ' , ' +
            (val0[i]*acc_multiplier) + " , " +
            (val1[i]*acc_multiplier) + " , " +
            (val2[i]*acc_multiplier) + " , " +
            (val3[i]*gyro_multiplier) + " , " +
            (val4[i]*gyro_multiplier) + " , " +
            (val5[i]*gyro_multiplier) + "<BR>";
    }
}
```

First we declare the function `start()` that is called when the start button is clicked. The main task is to send a websocket message to the controller and that is done with the `websock.send()` functions. As argument it takes a string that in this case format as a JSON string with two entries: `cmd` and `val`; the first with
argument START and the second with the contents of the variable sample_period. This message is decoded on the controller in the WStype_TEXT section of the webSocketEvent() function. All further commands from the browser to the controller use the same mechanism. Apart from sending the message to the controller the start() function also updates the text displayed on the start and stop buttons. The stop() function, that is executed by pressing the stop button, behaves in much the same way. In it we send a JSON message with STOP as command and -1 as value.

The functions toStatus() and toLogWin() are simply aliases to the longer versions that are in their respective function bodies. The calibrate() function sends the command CALIBRATE together with a value v to the controller. The datalogger() function prints the content of the waveforms to the logging window at the bottom of the web page. Presently it only works for the data from the fast acquisition.

In the following section we show the code to toggle the visibility of the traces and how to handle mouse-clicks inside the plot area.

document.getElementById("trace0").setAttribute('visibility','visible');
document.getElementById("trace1").setAttribute('visibility','visible');
document.getElementById("trace2").setAttribute('visibility','visible');
document.getElementById("trace3").setAttribute('visibility','visible');
document.getElementById("trace4").setAttribute('visibility','visible');
document.getElementById("trace5").setAttribute('visibility','visible');
document.getElementById("traceB0").setAttribute('visibility','visible');
document.getElementById("traceB1").setAttribute('visibility','visible');
document.getElementById("traceB2").setAttribute('visibility','visible');
document.getElementById("traceBtot").setAttribute('visibility','visible');
function toggletrace(v) {
  t1=document.getElementById(v);
  s=t1.getAttribute('visibility');
  if ( s == 'visible' ) {t1.setAttribute('visibility','hidden');}
  if ( s == 'hidden' ) {t1.setAttribute('visibility','visible');}
}
document.getElementById("accelerometer").addEventListener('mousedown',showCoordinates, false);
function showCoordinates(event) {
    rect=document.getElementById("accelerometer").getBoundingClientRect();
    toStatus("Acceleration at cursor = "+((200-(event.clientY-rect.top-1))
    *acc_multiplier/100).toPrecision(4) + " g");
}
document.getElementById("gyro").addEventListener
('mousedown',showCoordinates2, false);
function showCoordinates2(event) {
    rect=document.getElementById("gyro").getBoundingClientRect();
    toStatus("Angular rotation speed at cursor = "+((250-(event.clientY-rect.top))*gyro_multiplier)
    .toPrecision(4) + " deg/s");
}
document.getElementById("magnet").addEventListener
('mousedown',showCoordinates3, false);
function showCoordinates3(event) {
    rect=document.getElementById("magnet").getBoundingClientRect();
    toStatus("Magnetic field at cursor = "+((250-(event.clientY-rect.top))/magnet_multiplier)
    .toPrecision(3) + " &#181;T");
}

First we make all traces visible by declaring the respective attribute. The function
toggletrace() is hooked up to the buttons in the second row of the web page and
toggles the visibility of a trace between visible and hidden. The following three
sections add an EventListener to mousedown events in the plot areas and defines
functions that display the value at the clicked location in the status window.
An finally we have a number of service routines.

function setAccRange(v,h) {
    toStatus("Setting Acceleration range to option " + v);
    acc_multiplier=Math.pow(2,parseInt(v));
    websocket.send(JSON.stringify({ "cmd" : "ACCRANGE", "val" : v }));
}
function setGyroRange(v) {
    toStatus("Setting Gyro range to option " + v);
    gyro_multiplier=Math.pow(2,parseInt(v));
    websocket.send(JSON.stringify({ "cmd" : "GYRORANGE", "val" : v }));
}
function setDLPF(v) {
    toStatus("Setting digital low-pass filter to option "+ v);
    websocket.send(JSON.stringify({ "cmd" : "DLPF", "val" : v }));
}
function setSamplePeriod(v) {
    toStatus("Setting sample period to "+ v + " ms");

sample_period=v;
document.getElementById('trace0').setAttribute('d','M0 200');
document.getElementById('trace1').setAttribute('d','M0 200');
document.getElementById('trace2').setAttribute('d','M0 200');
document.getElementById('trace3').setAttribute('d','M0 250');
document.getElementById('trace4').setAttribute('d','M0 250');
document.getElementById('trace5').setAttribute('d','M0 250');
document.getElementById('traceB0').setAttribute('d','M0 250');
document.getElementById('traceB1').setAttribute('d','M0 250');
document.getElementById('traceB2').setAttribute('d','M0 250');
document.getElementById('traceBtot').setAttribute('d','M0 250');
current_position=0;
if (parseInt(sample_period) < 11) { //
    document.getElementById('trace0').setAttribute
        ('transform','translate(0,0)');
document.getElementById('trace1').setAttribute
        ('transform','translate(0,0)');
document.getElementById('trace2').setAttribute
        ('transform','translate(0,0)');
document.getElementById('trace3').setAttribute
        ('transform','translate(0,0)');
document.getElementById('trace4').setAttribute
        ('transform','translate(0,0)');
document.getElementById('trace5').setAttribute
        ('transform','translate(0,0)');
document.getElementById('traceB0').setAttribute
        ('transform','translate(0,0)');
document.getElementById('traceB1').setAttribute
        ('transform','translate(0,0)');
document.getElementById('traceB2').setAttribute
        ('transform','translate(0,0)');
document.getElementById('traceBtot').setAttribute
        ('transform','translate(0,0)');
}
} 
function setMagnetMultiplier(v) {
    toStatus("Setting magnet multiplier to "+ v);
magnet_multiplier=v;
}
function accTotal() {
    var path = document.getElementById('trace6');
s=path.getAttribute('visibility');
if ( s == 'visible' ) {
    path.setAttribute('visibility','hidden');
    return;
}
} else if ( s == 'hidden' ) {
    path.setAttribute('visibility','visible');
    return;
} else {
    val6=val0;
    for (i=0;i<val2.length;i++) {
        val6[i]=Math.sqrt(val0[i]*val0[i]+val1[i]*val1[i] +val2[i]*val2[i]);
    }
    var d =path.getAttribute('d');
    d="M0 200"; step=1024/val6.length;
    for (i=0; i<val6.length; i++) {d += ' L' + (step*i) + ' '+ (200-val6[i]/10);}
    path.setAttribute('d',d);
    path.setAttribute('visibility','visible');
}
}
function phiTotal() {
    var path = document.getElementById('trace7');
    s=path.getAttribute('visibility');
    if ( s == 'visible' ) {
        path.setAttribute('visibility','hidden');
        return;
    } else if ( s == 'hidden' ) {
        path.setAttribute('visibility','visible');
        return;
    } else {
        val7=val0;
        for (i=0;i<val2.length;i++) {
            val7[i]=Math.sqrt(val3[i]*val3[i]+val4[i]*val4[i] +val5[i]*val5[i]);
        }
        var d=path.getAttribute('d');
        d="M0 250"; step=1024/val7.length;
        for (i=0; i<val7.length; i++) {d += ' L' + (step*i) + ' '+ (250-val7[i]);}
        path.setAttribute('d',d);
        path.setAttribute('visibility','visible');
    }
}
function cleanup() {
    console.log("cleaning up");
    websocket.close();
}
window.addEventListener("beforeunload",cleanup);
The functions starting with set send the respective commands to set ranges and values of a selected parameters on the controller and report the new value in the status line at the bottom of the web page. In the setSamplePeriod() function we need to initialize the path variables and undo any transform that was used in roll mode to prevent from plotting new data off-screen.

The functions accTotal() and phiTotal() calculate the data for the black traces with the total acceleration and rotation speed and displays the traces. Internally it uses the same mechanisms as the other plotting functions. It accesses the d= to define the path with a number of M x y and L x y statements before writing d back with the setAttribute() function.

The last two actions are the definition of an EventListener to the beforeunload event which is executed as the last thing before the web page is left, for example by closing it. In that event the function cleanup() is called which closes the websocket connection and reports to the console log.

At the end of the file we need to provide the closing tags for </script>, </body> and </html> which completes the description of the user interface to interact with the controller.

5 Other Prototype Systems

I built several systems, one of which is shown in Fig. 2 which uses a MPU-9250 sensor and a somewhat bulky Li-ion battery. I built a similar system that I
managed to squeeze into a 40 mm pipe thanks to a flat Lipo battery which is smaller at the expense of a smaller capacity. This pipe I placed into the inside of a soft foam ball that I cut open. The ball and the ball cut open with the pipe containing the controller is visible on Fig. 4. It is quite entertaining to bounce the ball around and observe the acceleration it experiences.

Another early prototype using a MPU-6050 breakout board is shown in Fig. 5. It is built into a cable canal but the canal can not be closed because the sensor sticks up. Note the Lipo battery mounted below with the switch dangling on the side. Clearly an early version!

This concludes the description of the Gyroscope and I’ll try it out the next time I teach motion of spinning tops.

References