CODE EXPLAINATION

This PDF contains codes which were tested for each of sensor and modules to check how the basic system is work, and how to develop further.

GPS Code

```
#include <Arduino.h>
#include <TinyGPS++.h>

TinyGPSPlus gps;
HardwareSerial SerialGPS(USART2); //RX=PA3, TX=PA2

double totalDistance = 0.0;
double lastLat = 0.0, lastLon = 0.0;
bool hasStartingPoint = false;

void setup() {
   Serial.begin(115200);
   while (!Serial);
   Serial.println("FUELGUARD");
   Serial.println("");

   SerialGPS.begin(115200);
}

void gps_data() {
   while (SerialGPS.available() > 0) {
      char c = SerialGPS.read();
      gps.encode(c);

      if (gps.location.isUpdated()) {
         double currentLat = gps.location.lat();
         double currentLon = gps.location.lng();

         if (!hasStartingPoint) {
            lastLat = currentLat;
            lastLon = currentLon;
            hasStartingPoint = true;
            Serial.println("Starting point set.");
         } else {
            double segmentDistance = TinyGPSPlus::distanceBetween(
               lastLat, lastLon, currentLat, currentLon);

            totalDistance += segmentDistance;

            lastLat = currentLat;
            lastLon = currentLon;

            Serial.print("Segment Distance: ");
            Serial.print(segmentDistance, 2);
            Serial.println(" meters");

            Serial.print("Total Distance Traveled: ");
            Serial.print(totalDistance / 1000, 3);
            Serial.println(" kilometers");
         }
      }
```

GPS Code

```
        if (gps.speed.isUpdated()) {
            Serial.print("Speed (km/h): ");
            Serial.println(gps.speed.kmph(), 2);
        }

        if (gps.date.isUpdated() && gps.time.isUpdated()) {
            Serial.print("Date: ");
            Serial.print(gps.date.day());
            Serial.print("/");
            Serial.print(gps.date.month());
            Serial.print("/");
            Serial.print(gps.date.year());
            Serial.print(" Time: ");
            Serial.print(gps.time.hour());
            Serial.print(":");
            Serial.print(gps.time.minute());
            Serial.print(":");
            Serial.println(gps.time.second());
        }
    }
}

void loop() {
    gps_data();
}
```

GPS Code - Explaination

This code is designed to track location, calculate distance traveled, and display speed and time using GPS data. It continuously reads data from the GPS module and processes it in real-time.

At the beginning, the system initializes the serial monitor for debugging and establishes communication with the GPS module. A startup message is displayed to confirm that the system is running.

The main function reads incoming GPS data and checks if a new location update is available. If it's the first valid location, it sets it as the starting point. For every new location update, it calculates the distance traveled from the previous position and adds it to the total distance.

The system also monitors speed updates from the GPS data and displays the current speed in kilometers per hour. Additionally, if the date and time information is updated, it prints the current date and time from the GPS signal.

The process runs continuously, ensuring that location, speed, and time data are updated in real-time. This system can be useful for tracking movement, monitoring speed, and keeping records of travel history. It can be further improved by applying filters to reduce errors or by storing data for offline access.

GSM Code

```
#include <SPI.h>
#include <Wire.h>

#define SerialMon Serial   // USB Serial Monitor
#define SerialSim Serial1  // A7672S Serial on PA9 (TX), PA10 (RX)
#define PIN_GSM_PWRK PA8   // GSM Power Key


void sendATCommand(const char* cmd) {
   SerialSim.println(cmd);
   delay(1000);
   String response = "";
   while (SerialSim.available()) {
      response += (char)SerialSim.read();
   }
   SerialMon.print("Command: ");
   SerialMon.println(cmd);
   SerialMon.print("Response: ");
   SerialMon.println(response);
}


void updateSerial() {
   delay(200);  // Give some time for data to arrive
   String smsResponse = "";  // Reset for each function call

   while (SerialSim.available()) {
      char c = (char)SerialSim.read();
      smsResponse += c;
   }

   if (smsResponse.length() > 0) {
      smsResponse.trim();  // Remove whitespace and newlines
      SerialMon.println("\nSMS Response:");
      SerialMon.println(smsResponse);

      if (smsResponse.indexOf("Hello") != -1) {
         SerialSim.println("AT+CMGS=\"+91XXXXXXXXXX\"");
         delay(500);
         SerialSim.print("Hello from Direct UART");
         SerialSim.write(26);
         delay(5000);
         SerialMon.println("SMS Sent Successfully.");
      }

      // Reset smsResponse after processing to prevent looping
      smsResponse = "";
      while (SerialSim.available()) SerialSim.read();  // Clear serial buffer
   }

   smsResponse = "";
```

GSM Code

```
}



void setup() {
   SerialMon.begin(115200);
   SerialSim.begin(115200, SERIAL_8N1);

   SerialMon.println("Initializing A7672S Module...");
   delay(2000);

   sendATCommand("AT");                // Test communication with GSM
   sendATCommand("AT+CMGF=1");         // Set SMS to text mode
   sendATCommand("AT+CSCA=\"+917010075009\"");  // Set SMSC (India example)
   sendATCommand("AT+CNMI=1,2,0,0,0"); // Configure SMS notifications

   SerialMon.println("Ready to receive SMS.");
}


void loop() {
   updateSerial();  // Continuously check for SMS
}
```

GSM Code - Explaination

This code is designed to communicate with a GSM module (A7672S) using AT commands. It enables sending and receiving SMS messages through a serial connection. The system continuously listens for incoming messages and responds when a specific keyword is detected.

At the beginning, the system initializes the serial monitor for debugging and establishes communication with the GSM module. The communication is done via a UART serial connection, which allows the module to send and receive data.

After initialization, a set of AT commands is sent to configure the GSM module. The command AT checks if the module is responding. Then, AT+CMGF=1 sets SMS mode to text format, making it easier to read and send messages. The command AT+CSCA="NUMBER" sets the SMS center number (SMSC), which is required to send messages. Lastly, AT+CNMI=1,2,0,0,0 enables new SMS notifications, ensuring that any incoming message is immediately available for processing.

A function is created to send AT commands and read the module's response. This function sends a command, waits for a response, and prints both the command and response to the serial monitor for debugging purposes. This helps ensure that communication with the GSM module is working correctly.

The main loop continuously checks for incoming SMS messages. If a message is received, it is read and processed. If the message contains the word "Hello", the system automatically responds by sending an SMS back to a predefined number using the AT+CMGS="NUMBER" command. After writing the message content, a special CTRL+Z (ASCII 26) character is sent to indicate the end of the message.

To prevent repeated processing of the same message, the received SMS data is cleared from memory after processing. This ensures that each message is handled only once and avoids unnecessary repeated responses.

This system provides a basic way to interact with a GSM module for SMS-based communication. It can be further enhanced by adding additional AT commands to handle call functions, network status checks, or error handling mechanisms.

Fluid Flow Sensor Code

```
#include “Arduino.h"

#define FLOW_SENSOR_PIN PA0  // YF-SR04 signal pin

volatile uint32_t pulseCount = 0;
uint32_t lastTime = 0;
float flowRate = 0;
float totalFuelPassed = 0;  // Total volume of fuel passed in liters

// Interrupt service routine for counting pulses
void countPulse() {
   pulseCount++;
}

void setup() {
   Serial.begin(115200);

   pinMode(FLOW_SENSOR_PIN, INPUT_PULLUP);
   attachInterrupt(FLOW_SENSOR_PIN, countPulse, RISING);

   lastTime = millis();
}

void loop() {
   uint32_t currentTime = millis();

   if (currentTime - lastTime >= 1000) {  // Every 1 second
      noInterrupts();  // Disable interrupts while reading pulseCount
      uint32_t pulses = pulseCount;
      pulseCount = 0;
      interrupts();  // Re-enable interrupts

      // Convert pulses to flow rate in L/min (YF-SR04 = 450 pulses per L)
      flowRate = (pulses / 450.0) * 60.0;

      // Calculate fuel passed in the last second (L/sec)
      float fuelPassed = flowRate / 60.0;
      totalFuelPassed += fuelPassed;

      // Print data
      Serial.print("Flow Rate: ");
      Serial.print(flowRate);
      Serial.print(" L/min, ");
      Serial.print("Total Fuel Passed: ");
      Serial.print(totalFuelPassed);
      Serial.println(" L");

      lastTime = currentTime;
   }
}
```

Fluid Flow Sensor Code - Explaination

This code is designed to measure the fuel flow rate using a YF-SR04 flow sensor. It calculates both the instantaneous flow rate (L/min) and the total fuel passed (liters) over time. The sensor works by generating pulses as fuel flows through it, and the number of pulses is used to determine the fuel flow rate.

At the beginning, a pin is defined to read signals from the flow sensor. A volatile variable (pulseCount) is used to store the number of pulses counted by the sensor. The variables flowRate and totalFuelPassed are used to store the calculated fuel flow rate and total volume of fuel that has passed through the sensor.

An interrupt service routine (ISR) is defined to count pulses whenever the flow sensor generates a signal. This ensures that pulses are counted accurately, even if they occur very quickly. The interrupt is triggered on a RISING edge, meaning it detects when the signal changes from LOW to HIGH.

In the setup() function, serial communication is initialized for debugging. The sensor pin is set as an input with a pull-up resistor, and an interrupt is attached to the pin so that every pulse generated by the flow sensor is counted automatically. The lastTime variable stores the current time to keep track of when to update the flow rate calculations.

In the loop() function, the system checks if 1 second has passed since the last measurement. If so, it temporarily disables interrupts to safely read the pulse count without any interference. Then, the number of pulses counted in the last second is used to calculate the fuel flow rate in liters per minute. Since the YF-SR04 sensor generates 450 pulses per liter, the formula used is:

$$\text{Flow Rate (L/min)} = \left(\frac{\text{pulse}}{450}\right) \times 60$$

Next, the amount of fuel passed in the last second is calculated and added to the total fuel consumption using:

$$\text{Flow Rate (L/sec)} = \frac{\text{Flow Rate (L/min)}}{60}$$

Finally, the flow rate and total fuel consumed are displayed on the serial monitor. The lastTime variable is updated to ensure the next reading occurs exactly after 1 second.This system provides a simple but effective way to measure fuel consumption in real-time. It can be enhanced by adding data logging, error handling, or filtering techniques to improve accuracy.

IMU Code

```
#include “Arduino.h"
#include “Wire.h"
#include <math.h>

#define I2C_SDA PB7
#define I2C_SCL PB6
#define ADXL345_ADDR 0x53

float xOffset = 0;
float velocityX = 0; // Velocity in m/s
float distanceX = 0; // Distance in meters (not cm)
unsigned long prevTime = 0;
const float accelThreshold = 0.005; // Reduced threshold for better sensitivity
const float velocityDamping = 0.99; // Reduced damping for faster response
const float alpha = 0.2; // Increased complementary filter weight

void writeRegister(uint8_t reg, uint8_t value) {
   Wire.beginTransmission(ADXL345_ADDR);
   Wire.write(reg);
   Wire.write(value);
   Wire.endTransmission();
}

void configureADXL345() {
   writeRegister(0x2D, 0x08); // Power Control: Enable measurement mode
   writeRegister(0x31, 0x0B); // Data Format: Full resolution (4mg/LSB), ±16g range
   writeRegister(0x2C, 0x0D); // BW_RATE: Set to 800Hz output rate
   writeRegister(0x38, 0x00); // FIFO: Bypass mode
}

void readRawAcceleration(float &ax) {
   Wire.beginTransmission(ADXL345_ADDR);
   Wire.write(0x32);
   Wire.endTransmission(false);
   Wire.requestFrom(ADXL345_ADDR, 6, true);

   int16_t x = Wire.read() | (Wire.read() << 8);
   Wire.read(); Wire.read(); // Ignore Y-axis
   Wire.read(); Wire.read(); // Ignore Z-axis

   ax = x * 9.81 * 0.0039; // Convert to m/s²
}

void calibrate() {
   Serial.println("Calibrating... Please keep the sensor still.");
   long numSamples = 1000;
   float sumX = 0;

   for (long i = 0; i < numSamples; I++) {
      float ax;
      readRawAcceleration(ax);
```

IMU Code

```
      sumX += ax;
        delay(1);
    }

    xOffset = sumX / numSamples;
    Serial.println("Calibration complete.");
}

void setup() {
    Serial.begin(115200);
    Wire.setSDA(I2C_SDA);
    Wire.setSCL(I2C_SCL);
    Wire.begin();
    configureADXL345();
    Serial.println("ADXL345 Configured and Initialized");
    calibrate();
    prevTime = millis();
}

void loop() {
    unsigned long currentTime = millis();
    if (currentTime - prevTime < 10) return; // Ensure 10ms delay between readings
    float dt = (currentTime - prevTime) / 1000.0; // Convert to seconds
    prevTime = currentTime;

    float accelX;
    readRawAcceleration(accelX);
    accelX -= xOffset;

    // Apply complementary filter for smoothing
    static float filteredAccelX = 0;
    filteredAccelX = alpha * accelX + (1 - alpha) * filteredAccelX;

    // Apply noise threshold
    if (fabs(filteredAccelX) < accelThreshold) filteredAccelX = 0;

    // Ensure acceleration is always positive
    filteredAccelX = fabs(filteredAccelX);

    // Integrate acceleration to get velocity (m/s)
    velocityX = velocityX * velocityDamping + filteredAccelX * dt;

    // If velocity is below a very small threshold, reset to zero
    if (velocityX < 0.001) velocityX = 0;

    // Integrate velocity to get distance (m)
    distanceX += velocityX * dt;

    Serial.print("Accel X (m/s²): "); Serial.print(filteredAccelX, 4);
    Serial.print(" | Velocity X (km/h): "); Serial.print(velocityX * 3.6, 4); // Convert m/s to
km/h
    Serial.print(" | Distance X (km): "); Serial.println(distanceX / 1000.0, 6); // Convert m to
km
}
```

IMU - Explaination

This code measures acceleration, velocity, and distance using an ADXL345 accelerometer. It continuously integrates acceleration to estimate velocity and then integrates velocity to determine distance along the X-axis.

First, the I2C communication is set up, and the ADXL345 is configured to measurement mode with a ±16g range and 800 Hz output rate. The calibration process then calculates an offset value to remove sensor drift.

The function readRawAcceleration() reads raw acceleration data from the X-axis, ignoring Y and Z. It then converts the raw value into m/s^2 using a scaling factor.

A complementary filter smooths the acceleration data, and a threshold is applied to filter out noise. The code then integrates acceleration over time to compute velocity and further integrates velocity to estimate distance traveled.

Every 10 milliseconds, the program updates and prints acceleration (m/s^2), velocity (km/h), and distance (km) to the serial monitor.