Book Report

"Elevator Operation Health Diagnosis using Vibration Region Segmentation Algorithm via Internet"

Brahmaji Chukka

4th Aug, 2025

Contents in the Book

Required Components

Step 1: Power Supply Setup

🔌 Materials:

- **AMS1117 3.3V Voltage Regulator**

    The AMS1117 3.3V is a low dropout (LDO) voltage regulator, specifically designed to provide a stable 3.3V output from a higher input voltage. It's a popular choice for powering electronic circuits requiring a 3.3V supply, especially those using microcontrollers and other low-voltage components. This regulator is known for its simplicity, cost-effectiveness, and ability to deliver up to 1A of current.

Here's a more detailed breakdown:

Fixed Voltage Output:
The AMS1117-3.3 provides a fixed 3.3V output, meaning it's designed to output that specific voltage consistently.
Low Dropout:
It's a low dropout regulator, meaning it can operate with a small voltage difference between the input and output. This is important for efficient operation, especially when the input voltage is close to the desired output voltage.
Current Capacity:
The AMS1117-3.3 can deliver up to 1A of current, which is suitable for powering a variety of low-voltage devices.
Applications:
It's commonly used in projects like:

- Powering microcontrollers (e.g., Arduino, ESP32).
-
- Sensors and other low-voltage components.

- Post-regulation for switching power supplies.
- Battery-powered devices.

Simple Design:
The AMS1117 is known for its ease of use, often requiring only a few external capacitors for input and output filtering.
Packages:
It's available in various packages, including SOT-223 and TO-252, making it adaptable to different board layouts.

Protection Features:

It typically includes thermal shutdown and current limiting features to protect the regulator and the connected circuitry from overloads.

- Capacitors (10µF, 0.1µF)

Capacitors with values of 10µF and 0.1µF (also written as 100nF) are commonly used in electronic circuits for filtering and decoupling. The 10µF capacitor is typically used for low-frequency filtering and energy storage, while the 0.1µF capacitor is used for high-frequency filtering and noise suppression.

Here's a more detailed explanation:

10µF Capacitors:
These are generally larger capacitors, often electrolytic or tantalum types. They are well-suited for handling larger fluctuations in power requirements and act as a buffer in the circuit. They are good for low-frequency filtering and energy storage, meaning they can store a larger amount of electrical charge and release it more slowly.
0.1µF (100nF) Capacitors:
These are often smaller, ceramic capacitors, such as MLCC (Multi-Layer Ceramic Capacitors). They have lower series resistance and can respond quickly to fast changes in voltage or current. This makes them ideal for high-frequency filtering and suppressing noise in the circuit.
Combined Use:
It's common to see both 10µF and 0.1µF capacitors used together in parallel in a circuit. This combination allows the circuit to handle both large, slow changes in voltage and fast, high-frequency noise. The 0.1µF capacitor filters out high-frequency noise, while the 10µF capacitor provides a stable voltage source for the circuit.

Example Applications:

These capacitors are frequently found in power supplies, voltage regulators, and decoupling circuits to ensure stable and clean power delivery to sensitive electronic components.

- **Battery Pack or USB 5V input**

🔍 What It Does:

- Converts 5V (from USB or battery) to stable 3.3V.
- Almost all components (sensor, microcontroller, RTC, EEPROM) need **3.3V**, not 5V.
- Capacitors stabilize voltage and prevent noise.

🛠️ Why It's Important:

- Microcontrollers and sensors are sensitive to voltage.
- Incorrect voltage may **damage** components or cause **unstable readings**.

Step 2: Connect the LSM6DSR Accelerometer

🎯 Component: LSM6DSR (3D Accelerometer + Gyroscope)

The LSM6DSR is a system-in-package featuring a 3D digital accelerometer and a 3D digital gyroscope, also known as an inertial measurement unit (IMU). It's designed to provide high-performance motion sensing for various applications, including augmented and virtual reality, optical image stabilization (OIS), and motion-based gaming controllers.

Here's a more detailed breakdown:

Combined Sensor:
It integrates both an accelerometer and a gyroscope into a single package.

High Performance:
The LSM6DSR offers extended full-scale ranges for the gyroscope (up to 4000 dps) and high stability over temperature and time.

Applications:
It's well-suited for:

- **Augmented and Virtual Reality:** Providing accurate motion tracking for immersive experiences.
- **Optical Image Stabilization (OIS):** Supporting camera stabilization through both gyroscope and accelerometer data.
- **Motion-based Gaming:** Enabling more sophisticated game controls and interactions.
- **General Motion Sensing:** Detecting orientation, gestures, and other movements for various applications.

Key Features:

- **Extended gyroscope full-scale range:** Supports up to 4000 dps.
- **High stability:** Maintains performance accuracy over temperature and time.
- **Smart FIFO:** Utilizes a FIFO (First-In, First-Out) buffer for efficient data management.
- **Android Compliance:** Meets the requirements for Android-based devices.
- **Auxiliary SPI:** Provides a dedicated interface for OIS data output.
- **Programmable Finite State Machine:** Allows for custom logic and state management.
- **Multiple Interfaces:** Supports SPI, I2C, and MIPI I3CSM interfaces for communication with a main processor.

Sensor Fusion:
The LSM6DSR can be combined with other sensors, like magnetometers, using its Sensor Hub feature, allowing for more complex motion analysis and tracking.

🔍 What It Does:

- Measures **acceleration** in X, Y, Z axes.

- Internally includes a gyroscope (not used in your case).
- Outputs data over I2C or SPI.

🛠️ Why It’s Important:

- The **core sensor** of your system.
- You use acceleration data to compute **jerk** (rate of change of acceleration), which indicates abnormal elevator behavior.

📢 Bonus Tip:

- You can mount the LSM6DSR inside the elevator cabin for real motion sensing.

Step 3: dsPIC33CK64MP506 Microcontroller

🔧 Component: dsPIC33CK64MP506

The dsPIC33CK64MP506 is a 16-bit digital signal controller (DSC) from Microchip Technology. It features a dsPIC33CK core, 64KB of Flash memory, 8KB of RAM, and operates at a maximum clock frequency of 100MHz. It is available in a 64-pin QFN package and offers various analog and communication peripherals, including up to three operational amplifiers (OpAmps), comparators, and CAN-FD (Flexible Data-rate CAN).

Here's a more detailed breakdown:

Core:
The device is built around Microchip's dsPIC33CK core, which is designed for high-performance digital signal processing and control applications.

Memory:
It includes 64KB of Flash memory for program storage and 8KB of RAM for data storage.

Clock Speed:
The maximum clock frequency is 100MHz, enabling fast processing of data.

Package:
The dsPIC33CK64MP506 is available in a 64-pin Quad Flat No-leads (QFN) package.

Analog Features:
It incorporates several analog components, including up to three OpAmps and comparators.

Communication:
The device supports various communication interfaces, such as CAN-FD, UART, SPI, and I2C.

Advanced Features:
It includes a high-speed ADC with configurable resolution and flexible trigger sources, along with other features like a Programmable Cyclic Redundancy Check (CRC) and DMA channels.

Safety Features:
It incorporates a Clock Monitor System with a backup oscillator and a Deadman Timer.
Applications:
This DSC is suitable for a wide range of applications, including motor control, power conversion, and other embedded control systems.

🔍 What It Does:

- Main brain of the system.
- Reads sensor data, calculates jerk, detects abnormal patterns.
- Stores and transmits data to PC.

🛠️ Why It's Important:

- Performs all real-time data processing.
- Has built-in support for:
  - **I2C communication** (for sensor, EEPROM, RTC)
  - **Timers and interrupts**
  - **UART communication** (for USB/PC)

Step 4: Connect EEPROM (e.g., AT24C256)

💾 Component: EEPROM (AT24C256)

The AT24C256 is a 256-Kbit (32KB) I2C EEPROM (Electrically Erasable Programmable Read-Only Memory) chip. It's a type of non-volatile memory, meaning it retains data even when power is removed. The AT24C256 is commonly used in microcontroller-based projects to store data that needs to be preserved, such as configuration settings, calibration data, or small amounts of program code.

Here's a more detailed breakdown:

Key Features:

- **256Kbit (32KB) Storage:** Offers a significant amount of storage for various applications.
- **I2C Interface:** Uses the I2C (Inter-Integrated Circuit) communication protocol for easy integration with microcontrollers.
- **Non-Volatile:** Data is preserved even when the device is powered off.
- **Write Protection:** Can be configured to protect data from accidental writes.
- **Low Power Consumption:** Suitable for battery-powered devices.
- **Wide Operating Voltage:** Typically compatible with 2.7V to 5.5V systems.
- **Endurance:** Can typically withstand 1 million write cycles per byte.
- **Data Retention:** Data can be retained for up to 100 years.

How it works:

The AT24C256 communicates with a microcontroller using two wires: SDA (Serial Data) and SCL (Serial Clock). Data is written to and read from specific memory addresses within the EEPROM. The I2C protocol allows multiple devices to share the same bus, making it efficient for systems with multiple memory chips.

Common Applications:

- **Data Logging:** Storing sensor readings, event logs, and other time-stamped data.
- **Configuration Storage:** Saving user preferences, device settings, and application parameters.
- **Calibration Data:** Storing calibration values for sensors and other devices.
- **Firmware Storage:** In some cases, a small amount of firmware code can be stored on the EEPROM.
- **Embedded Systems:** Widely used in various embedded systems that require persistent storage.

In essence, the AT24C256 is a reliable and versatile EEPROM chip widely used in embedded systems for storing essential data that needs to be retained even when the power is off.

🔍 What It Does:

- Stores sensor data when power is off.
- Keeps logs of vibration events locally.

🛠️ Why It's Important:

- Prevents data loss in case of power failure.
- Allows offline storage before uploading to PC.

Step 5: Connect RTC (Real-Time Clock)

🕒 Component: DS3231 RTC Module

The DS3231 is a highly accurate real-time clock (RTC) module, commonly used in electronics projects to track time and date. It features a built-in temperature-compensated crystal oscillator (TCXO) for enhanced accuracy and a battery backup to maintain timekeeping even when the main power is off.

Here's a more detailed explanation:

Key Features and Functionality:

Accurate Timekeeping:
The DS3231 chip includes a TCXO, which minimizes the impact of temperature variations on the clock's accuracy, resulting in more precise timekeeping.
Battery Backup:
The module has a space for a backup battery (like a CR2032), allowing it to maintain the time and date even when the main power supply is interrupted.
I2C Interface:
It communicates with microcontrollers (like Arduino or Raspberry Pi) using the I2C protocol, which requires only two wires (SDA and SCL) for data transfer.
Time and Date Tracking:
The DS3231 keeps track of seconds, minutes, hours, day, date, month, and year.
Alarm and Square Wave Output:
It also features two programmable alarms and a programmable square-wave output.

Applications:

- **Clocks:** The DS3231 is ideal for building accurate clocks and time displays.
- **Data Logging:** It can be used to timestamp data collected by other devices.
- **Timers and Alarms:** Its alarm functions make it useful for creating timed events and alarms.

- **Embedded Systems:** The module's small size and low power consumption make it suitable for various embedded systems.

How it works:

- The DS3231 chip contains a crystal oscillator that provides a stable clock signal.
- The built-in temperature sensor monitors the ambient temperature.
- The control logic adjusts the clock frequency based on temperature readings to compensate for any drift.
- The module uses an I2C interface to communicate with a microcontroller, sending time and date information upon request.
- When main power is lost, the backup battery takes over to keep the time accurate.

🔍 What It Does:

- Keeps track of real-world time.
- Adds **timestamp** to each sensor reading or jerk event.

🛠️ Why It's Important:

- Enables you to know **when** an abnormal event occurred.
- Helps with **data logging** and **cloud monitoring**.

Step 6: USB Communication

🔌 Component: USB-to-Serial Converter (CP2102 or FT232RL)

USB-to-Serial converters, like those based on the CP2102 or FT232RL chips, are devices that enable communication between a computer's USB port and a UART (Universal Asynchronous Receiver/Transmitter) serial interface, commonly found in microcontrollers and other embedded systems. They translate the USB protocol into a serial communication protocol (UART) that can be understood by these devices, allowing them to interact with a computer.

How they work:

USB Interface:
The converter plugs into a USB port on the computer, and the computer recognizes it as a virtual COM port.
UART Interface:
On the other side, the converter provides a UART interface, typically with pins for transmit (TX), receive (RX), ground (GND), and sometimes power (VCC), and potentially others like RTS and CTS for handshaking.
Conversion:
The converter's chip handles the conversion between the USB and UART protocols, allowing data to flow seamlessly between the computer and the connected device.

Why use them?

Microcontroller Programming:
They are commonly used to program microcontrollers like Arduino or those found in Raspberry Pi or other embedded systems that often lack built-in USB connectivity.
Serial Communication:
They facilitate serial communication between a computer and devices that use UART, such as GPS modules, GSM modules, and other embedded systems.

Debugging:
They can be used to monitor serial data and debug communication issues in embedded systems.

Specific chips:

CP2102:
A popular and cost-effective chip for USB to UART conversion, known for its reliability and ease of use.
FT232RL:
Another widely used chip for USB to UART conversion, often favored for its robust performance and compatibility.

Key features of these converters:

Baud Rate:
Support a wide range of baud rates for serial communication (e.g., 300 bps to 1.5 Mbps).
Voltage Levels:
Support both 3.3V and 5V logic levels for compatibility with different devices.
LED Indicators:
Often include LEDs to indicate data transmission and reception, aiding in debugging.
Auto-Reset:
Some modules, particularly those used with Arduino, include an auto-reset feature for simplified programming.
Self-Recovery Fuse:
Some modules incorporate a self-recovery fuse to protect the computer's USB port from short circuits.

🔍 What It Does:

- Acts as a bridge between the microcontroller and the computer.
- Transfers data from dsPIC to PC via USB.

🛠️ Why It's Important:

- Sends processed jerk/acceleration data to the **PC UI**

- Useful for **debugging**, **visualization**, and **cloud upload**

Step 7: PC Interface & SQL Server

💻 Software Tools:

- Python (for PC interface)
- SQL Server or MySQL (for cloud database)
- Serial monitor software (like PuTTY or TeraTerm)

🔍 What It Does:

- Reads data sent via USB
- Visualizes data (graphs of acceleration, jerk)
- Stores results in a cloud database for analysis

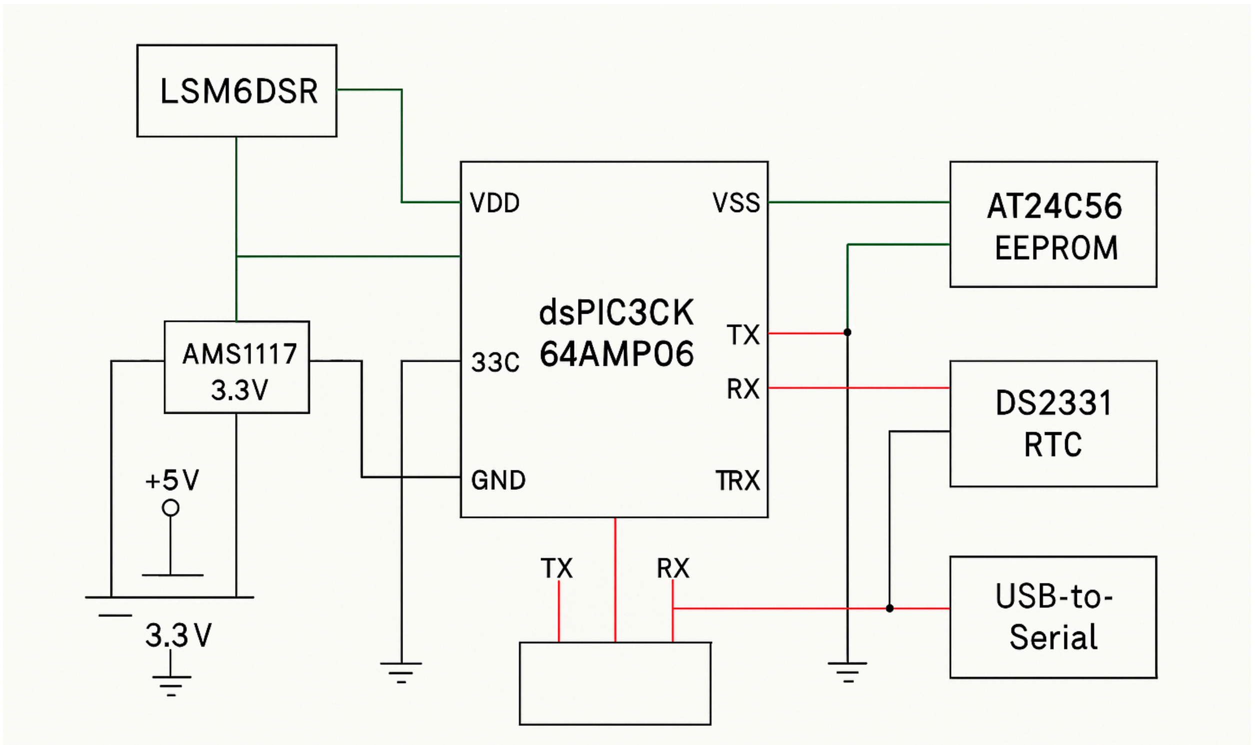
🛠️ Why It's Important:

- Lets you monitor elevator performance in real-time or remotely.
- Ensures **long-term recordkeeping** and comparison with commercial tools (like EVA625).

Summary Table of Components & Their Functions

Component	Function	Why It's Needed
AMS1117	5V to 3.3V regulator	Power supply for all 3.3V components
LSM6DSR	3D acceleration sensing	Core sensor for motion data
dsPIC33CK64MP506	Central controller	Data collection, processing, and communication
EEPROM	Data storage	Keeps logs even if power is lost
RTC (DS3231)	Real-time clock	Adds timestamps to events
USB to Serial Module	PC communication	Transfers data to computer
PC Software	Visualization & SQL upload	See results and monitor over time
LED/Buzzer/OLED	Status indicators	Make device user-friendly

Actual **circuit schematic diagram**



Detectcing Abnmal Elevator Vibrations Actoms

```
+----------------------------+
|        Power Supply        |
|   5V --> AMS1117 --> 3.3V   |
+----------------------------+
          |
          V
+----------------------------+
|      dsPIC33CK64MP506       |
| - I2C to LSM6DSR, EEPROM, RTC|
| - UART to USB Module        |
| - 3.3V, GND shared to all    |
+----------------------------+
          |
  +-------+-------+--------+
  |       |       |        |
  V       V       V        V
LSM6DSR  EEPROM   RTC     USB
Accel    Storage  Clock   Comm
```

🔌 Power Supply Block

🔧 AMS1117 3.3V Regulator

- **Input:** +5V (from USB or battery)
- **Output:** 3.3V (used for all components)
- **Capacitor Filter:** Ensures clean power.

Purpose: Provides a stable 3.3V power supply to the microcontroller, sensor, EEPROM, and RTC.

🧠 dsPIC33CK64MP506 Microcontroller (Center of the Diagram)

This is the **main processing unit** of the circuit. It:

- Reads data from the LSM6DSR sensor
- Calculates jerk
- Flags abnormal vibrations
- Communicates with EEPROM, RTC, and PC

Important Pins:

- **VDD / VSS →** Power supply (3.3V / GND)
- **SDA / SCL (I2C) →** Connected to sensor, EEPROM, and RTC
- **TX / RX (UART) →** Connected to USB-to-Serial for PC communication

---

📦 LSM6DSR Accelerometer

- **Connected to the I2C bus** shared with EEPROM and RTC.
- Sends real-time 3-axis acceleration data to the dsPIC.
- Powered by 3.3V and GND.

Purpose: This sensor is the core data source for detecting motion and calculating jerk.

💾 AT24C56 EEPROM

- I2C device (connected to same SDA/SCL lines).
- Stores vibration data and system logs in non-volatile memory.

Purpose: Keeps a local record of abnormal events, even if power is lost.

🕒 DS3231 RTC Module

- Also on the I2C bus.

- Provides **timestamps** for all readings and jerk events.

Purpose: Allows you to know *when* each vibration happened.

🔌 USB-to-Serial Module

- Connected to **TX/RX** pins of the microcontroller.
- Converts UART signals to USB for PC connection.

Purpose: Sends processed data to a PC interface or cloud system for visualization, logging, or alerting.

🧠 How the Whole System Works Together

1. **Power on the circuit →** 3.3V distributed to all modules.
2. **The sensor reads acceleration data** in real time.
3. **dsPIC calculates jerk** using acceleration readings.
4. **If TQ** > **4**, it:
    - Flags "abnormal"
    - Saves event in EEPROM
    - Sends result + timestamp to PC
5. **The PC receives data** via USB for display and storage.

The dsPIC firmware to read LSM6DSR and calculate jerk

complete **dsPIC33CK64MP506 firmware outline** to:

1. **Read acceleration** from the **LSM6DSR** over I2C
2. **Calculate jerk** (rate of change of acceleration)
3. **Detect abnormal vibrations** based on a TQ threshold
4. **Send data via UART** to the PC

✅ Tools Required

- **MPLAB X IDE** (Microchip)
- **XC16 Compiler**
- **I2C and UART libraries**
- **Pickit4 or ICD programmer**

✅ Assumptions

- **I2C** used to interface with **LSM6DSR**

- **UART** used for PC communication
- Using **only one axis (Z)** for simplicity
- **Sampling rate:** 100 Hz
- **TQ threshold:** `TQ > 4` is abnormal

🧠 Algorithm Summary

1. Initialize peripherals (I2C, UART, Timer)
2. Read Z-axis acceleration at fixed intervals
3. Calculate **jerk** = **$(a_{\square} - a_{\square-1}) / \Delta t$**
4. Convert jerk signal to binary: 1 (jerk > threshold), 0 (else)
5. Count transitions (TQ)
6. If TQ > 4, mark as **abnormal**
7. Send result via UART

Firmware Code (XC16 / MPLAB X)

```
#include <xc.h>

#include <stdint.h>

#include <stdbool.h>

#include <stdio.h>
```

```
#include <math.h>


// I2C & UART config headers assumed

#include "i2c.h"

#include "uart.h"


// Constants

#define LSM6DSR_ADDR 0x6A

#define OUTZ_L_G 0x26

#define OUTZ_H_G 0x27

#define ACCEL_SENSITIVITY 0.061f   // mg/LSB

#define JERK_THRESHOLD 500.0f      // mg/s

#define SAMPLE_INTERVAL_MS 10      // 100 Hz

#define MAX_SAMPLES 100


// Globals

float acc_prev = 0.0;

uint8_t jerk_bin[MAX_SAMPLES];

uint16_t TQ = 0;


void delay_ms(uint16_t ms);

void read_accel_z(float *accel_z);


void main(void) {
```

```
SYSTEM_Initialize();

I2C_Init();

UART_Init();


__delay_ms(500);  // Startup delay


float acc_current, jerk;

uint8_t binary_prev = 0, binary_now;

TQ = 0;


for (uint8_t i = 0; i < MAX_SAMPLES; i++) {

    read_accel_z(&acc_current);


    // Calculate jerk

    jerk = (acc_current - acc_prev) / (SAMPLE_INTERVAL_MS / 1000.0f);

    acc_prev = acc_current;


    // Binarization

    binary_now = (fabs(jerk) > JERK_THRESHOLD) ? 1 : 0;

    jerk_bin[i] = binary_now;


    // Count transitions (TQ)

    if (binary_now != binary_prev)

        TQ++;
        binary_prev = binary_now;


        delay_ms(SAMPLE_INTERVAL_MS);
    }


    // Classification
    if (TQ > 4)
        UART_SendString("Abnormal Vibration Detected\r\n");
    else
        UART_SendString("Normal Operation\r\n");
}
```

`read_accel_z()` Function

```
void read_accel_z(float *accel_z) {
    uint8_t lsb, msb;
    int16_t raw_z;


    I2C_Start();
    I2C_Write((LSM6DSR_ADDR << 1) | 0);      // Write address
    I2C_Write(0x2C);                         // OUTZ_L_XL (accel)
    I2C_Restart();
    I2C_Write((LSM6DSR_ADDR << 1) | 1);      // Read
    lsb = I2C_Read_ACK();
    msb = I2C_Read_NACK();
    I2C_Stop();

    raw_z = (int16_t)((msb << 8) | lsb);
    *accel_z = raw_z * ACCEL_SENSITIVITY;     // Convert to mg
}
```

Supporting Functions (Placeholders)

```
void delay_ms(uint16_t ms) {
    for (uint16_t i = 0; i < ms; i++)
        __delay_ms(1);
}
```

🧪 Output (via UART)

- `"Normal Operation\r\n"` — if TQ ≤ 4
- `"Abnormal Vibration Detected\r\n"` — if TQ > 4

This is visible on your **PC terminal** via USB-Serial.

1. Real-Time Logging with EEPROM and RTC Timestamp

📦 Required:

- **EEPROM (e.g., AT24C256 over I2C)**
- **RTC (e.g., DS3231 over I2C)**

🧠 What You'll Do:

- At each sample, fetch timestamp from RTC.
- Save: `[timestamp][acc_x][acc_y][acc_z][jerk_x][jerk_y][jerk_z]` to EEPROM.

🔧 Key Functions to Implement:

- `rtc_get_time()` → returns timestamp string or BCD values.
- `eeprom_write_bytes(address, data[], length)` → save one record.
- Manage EEPROM address so it doesn't overwrite old data (use pointer tracking).

2. PC Interface in Python for Graphing

🛠️ Tools:

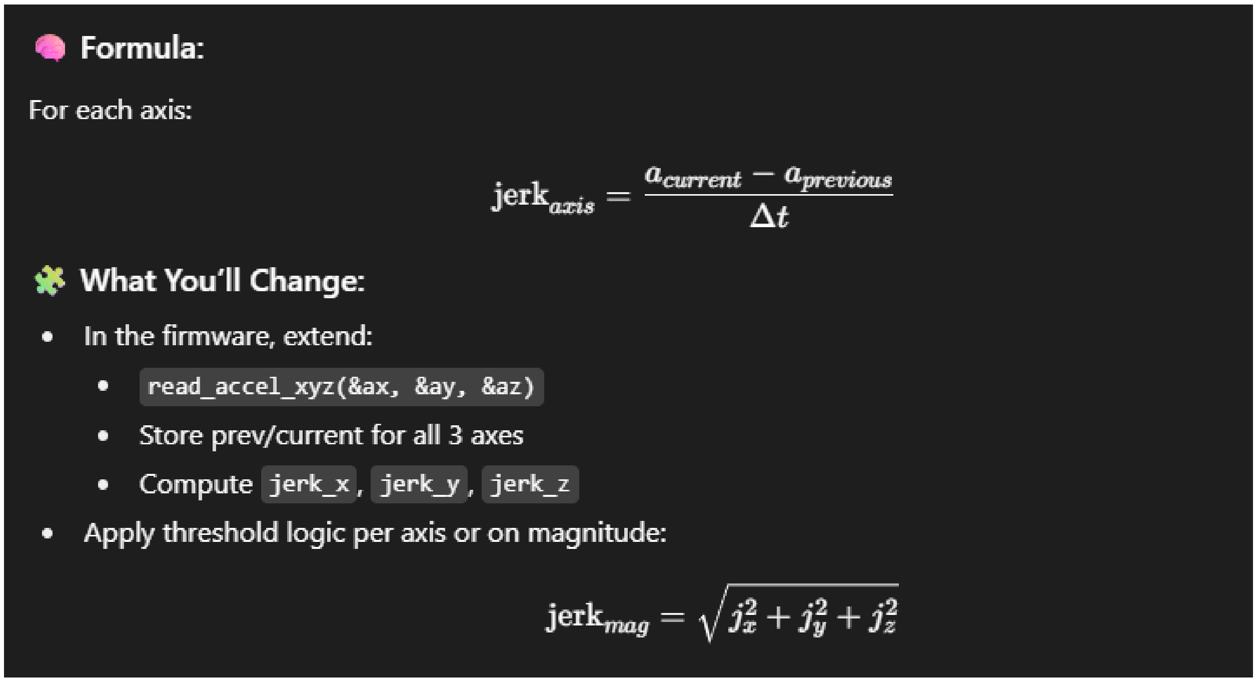
- **Python**
- **PyQt5 or Tkinter** (UI)
- **matplotlib** or **PyQtGraph** (real-time plotting)
- **pyserial** (read from USB/Serial)

📈 Features:

- Start/Stop button to record data
- Live plots of:
  - Acceleration (x, y, z)
  - Jerk (x, y, z)
- Label result: “Normal” or “Abnormal”
- Save to CSV or upload to SQL database

I can give you ready-to-run Python code next.

3. Jerk Calculation on All 3 Axes



🧠 **Formula:**

For each axis:

$$\text{jerk}_{axis} = \frac{a_{current} - a_{previous}}{\Delta t}$$

🧩 **What You'll Change:**

- In the firmware, extend:
  - `read_accel_xyz(&ax, &ay, &az)`
  - Store prev/current for all 3 axes
  - Compute `jerk_x`, `jerk_y`, `jerk_z`
- Apply threshold logic per axis or on magnitude:

$$\text{jerk}_{mag} = \sqrt{j_x^2 + j_y^2 + j_z^2}$$

Part 1: Updated Firmware for 3-Axis Jerk Calculation

This version:

- Reads X, Y, Z acceleration
- Calculates jerk on all 3 axes
- Sends acceleration & jerk via UART to PC

🔁 Main Loop Overview

1. Read `acc_x`, `acc_y`, `acc_z` from LSM6DSR
2. Calculate `jerk_x`, `jerk_y`, `jerk_z`
3. Check if jerk exceeds thresholds
4. Count transitions (TQ)
5. Print data to UART for PC plotting

🧠 Global Variables & Constants

c

CopyEdit

```c
#define SAMPLE_INTERVAL_MS 10     // 100Hz
#define JERK_THRESHOLD 500.0f     // mg/s
#define MAX_SAMPLES 100

float acc_prev[3] = {0.0, 0.0, 0.0};
uint8_t jerk_bin[3][MAX_SAMPLES] = {0};
uint16_t TQ[3] = {0, 0, 0};
```

📦 Function to Read All 3 Axes

```
void read_accel_xyz(float *ax, float *ay, float *az) {
    uint8_t raw_data[6];
    int16_t raw_x, raw_y, raw_z;

    I2C_Start();
    I2C_Write((LSM6DSR_ADDR << 1) | 0);       // Write address
    I2C_Write(0x28);                          // OUTX_L_XL
    I2C_Restart();
    I2C_Write((LSM6DSR_ADDR << 1) | 1);       // Read
    for (uint8_t i = 0; i < 6; i++)
        raw_data[i] = (i < 5) ? I2C_Read_ACK() : I2C_Read_NACK();
    I2C_Stop();

    raw_x = (int16_t)(raw_data[1] << 8 | raw_data[0]);
    raw_y = (int16_t)(raw_data[3] << 8 | raw_data[2]);
    raw_z = (int16_t)(raw_data[5] << 8 | raw_data[4]);

    *ax = raw_x * ACCEL_SENSITIVITY;
    *ay = raw_y * ACCEL_SENSITIVITY;
    *az = raw_z * ACCEL_SENSITIVITY;
}
```

🚀 Main Loop

```
for (uint8_t i = 0; i < MAX_SAMPLES; i++) {

    float acc_now[3], jerk[3];

    read_accel_xyz(&acc_now[0], &acc_now[1], &acc_now[2]);


    for (uint8_t axis = 0; axis < 3; axis++) {

        jerk[axis] = (acc_now[axis] - acc_prev[axis]) /
(SAMPLE_INTERVAL_MS / 1000.0f);

        acc_prev[axis] = acc_now[axis];


        // Binarization & TQ

        uint8_t bin = (fabs(jerk[axis]) > JERK_THRESHOLD) ? 1 : 0;

        if (i > 0 && bin != jerk_bin[axis][i - 1])

            TQ[axis]++;

        jerk_bin[axis][i] = bin;

    }
    // Send data to PC
    char buf[128];
    sprintf(buf, "ACC: %.2f,%.2f,%.2f | JERK: %.2f,%.2f,%.2f\r\n",
            acc_now[0], acc_now[1], acc_now[2],
            jerk[0], jerk[1], jerk[2]);
    UART_SendString(buf);

    delay_ms(SAMPLE_INTERVAL_MS);
}
```

🏁 Final Classification

```
UART_SendString("TQ values:\r\n");
for (uint8_t axis = 0; axis < 3; axis++) {
    char out[32];
    sprintf(out, "Axis %d TQ = %d\r\n", axis, TQ[axis]);
    UART_SendString(out);

    if (TQ[axis] > 4)
        UART_SendString("Abnormal vibration detected on this
axis\r\n");
}
```

✅ Goals for the Python GUI

Feature	Description
Serial Communication	Read real-time data from dsPIC (UART via USB)
Live Plotting	Acceleration and jerk for X, Y, Z axes
Classification	Display "Normal" or "Abnormal" status
CSV Logging	Save incoming data for later analysis

🧰 Required Libraries

Install these first if you haven't already:

bash

CopyEdit

```
pip install pyserial matplotlib pyqt5
```

🖥️ Python GUI Code

Here’s a working **PyQt5** GUI that reads and plots live serial data.

python

CopyEdit

```
import sys

import serial

import csv

from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel,
QPushButton, QFileDialog

from PyQt5.QtCore import QTimer

from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg

from matplotlib.figure import Figure

from collections import deque


# SERIAL PORT SETTINGS

PORT = 'COM3'      # Change as needed

BAUD = 9600        # Match with dsPIC UART speed


# === GUI Class ===

class ElevatorMonitor(QMainWindow):

    def __init__(self):

        super().__init__()

        self.setWindowTitle("Elevator Vibration Monitor")

        self.setGeometry(100, 100, 1000, 600)
        self.serial = serial.Serial(PORT, BAUD, timeout=1)
        self.data_buffer = deque(maxlen=100)
        self.jerk_buffer = deque(maxlen=100)
        self.csv_data = []

        self.init_ui()
        self.start_timer()

    def init_ui(self):
        self.status_label = QLabel("Status: ---", self)
        self.status_label.setGeometry(800, 50, 180, 30)

        self.save_btn = QPushButton("Save to CSV", self)
        self.save_btn.setGeometry(800, 100, 120, 40)
        self.save_btn.clicked.connect(self.save_csv)

        # Matplotlib Figure
        self.canvas = FigureCanvasQTAgg(Figure())
        self.ax1 = self.canvas.figure.add_subplot(211)
        self.ax2 = self.canvas.figure.add_subplot(212)
        self.canvas.setGeometry(20, 20, 750, 550)
        self.setCentralWidget(self.canvas)

    def start_timer(self):
        self.timer = QTimer()

        self.timer.timeout.connect(self.update_data)

        self.timer.start(100)


    def update_data(self):

        if self.serial.in_waiting:

            line = self.serial.readline().decode().strip()

            if "ACC:" in line and "JERK:" in line:

                try:

                    acc_raw = line.split("|")[0].replace("ACC:",
"").strip()

                    jerk_raw = line.split("|")[1].replace("JERK:",
"").strip()

                    acc = [float(x) for x in acc_raw.split(",")]

                    jerk = [float(x) for x in jerk_raw.split(",")]


                    self.data_buffer.append(acc)

                    self.jerk_buffer.append(jerk)

                    self.csv_data.append(acc + jerk)


                    self.plot_data()


                    # Simple rule: abnormal if any axis > threshold

                    if any(abs(j) > 500 for j in jerk):

                        self.status_label.setText("Status: Abnormal")
                    self.status_label.setStyleSheet("color: red;")
                else:
                    self.status_label.setText("Status: Normal")
                    self.status_label.setStyleSheet("color: 
green;")
            except:
                pass

    def plot_data(self):
        if len(self.data_buffer) == 0:
            return

        ax_data = list(zip(*self.data_buffer))  # (X, Y, Z)
        jrk_data = list(zip(*self.jerk_buffer))

        self.ax1.clear()
        self.ax1.plot(ax_data[0], label="X")
        self.ax1.plot(ax_data[1], label="Y")
        self.ax1.plot(ax_data[2], label="Z")
        self.ax1.set_title("Acceleration (mg)")
        self.ax1.legend(loc='upper right')

        self.ax2.clear()
        self.ax2.plot(jrk_data[0], label="JX")
        self.ax2.plot(jrk_data[1], label="JY")

        self.ax2.plot(jrk_data[2], label="JZ")

        self.ax2.set_title("Jerk (mg/s)")

        self.ax2.legend(loc='upper right')


        self.canvas.draw()


    def save_csv(self):

        filename, _ = QFileDialog.getSaveFileName(self, "Save CSV",
"", "CSV Files (*.csv)")

        if filename:

            with open(filename, 'w', newline='') as file:

                writer = csv.writer(file)

                writer.writerow(['Acc_X', 'Acc_Y', 'Acc_Z', 'Jerk_X',
'Jerk_Y', 'Jerk_Z'])

                writer.writerows(self.csv_data)


# === Run App ===

if __name__ == "__main__":

    app = QApplication(sys.argv)

    win = ElevatorMonitor()

    win.show()

    sys.exit(app.exec_())
```

🔌 How to Use

1 Connect dsPIC via USB and make sure it sends lines like:

yaml
```
ACC: 123.45, 234.56, 87.90 | JERK: 101.1, 205.3, 401.0
```

1. **2 Update `PORT = 'COM3'`** to match your system
2. 3 Run the script — it opens a GUI with:
   - Real-time plots of acceleration and jerk
   - Status label: *Normal / Abnormal*
   - Button to save all data to CSV

the **EEPROM + RTC logging** in your dsPIC firmware. This will allow your device to:

1. Read real-time timestamps from the RTC (DS3231)
2. Log timestamped data to external EEPROM (e.g., AT24C256)
3. Store `[timestamp][acc_x][acc_y][acc_z][jerk_x][jerk_y][jerk_z]` efficiently

✅ EEPROM + RTC Logging Overview

🧩 Modules:

Module	Interface	Purpose
DS3231	I2C	Real-time timestamping
AT24C256	I2C	32KB non-volatile data storage

📦 1. Required Libraries (or Functions)

Make sure you already have working **I2C master** routines:

- `I2C_Start()`, `I2C_Stop()`, `I2C_Write()`, `I2C_Read_ACK()`, `I2C_Read_NACK()`

🕒 2. RTC Interface (DS3231)

📘 Get Time from RTC

c

CopyEdit

```
void rtc_get_time(uint8_t *hours, uint8_t *minutes, uint8_t *seconds)
{

    I2C_Start();

    I2C_Write(0xD0);        // DS3231 write

    I2C_Write(0x00);        // Start at seconds register

    I2C_Restart();

    I2C_Write(0xD1);        // DS3231 read
    *seconds = I2C_Read_ACK();
    *minutes = I2C_Read_ACK();
    *hours   = I2C_Read_NACK();

    I2C_Stop();
}
```

🧠 DS3231 time is in BCD format (e.g., 0x45 = 45)

✅ BCD to Decimal Conversion

c

CopyEdit

```
uint8_t bcd_to_dec(uint8_t val) {
    return ((val >> 4) * 10) + (val & 0x0F);
}
```

💾 3. EEPROM Write (AT24C256)

📝 Write 16 bytes (e.g., one record)

c

CopyEdit

```
#define EEPROM_ADDR 0xA0  // AT24C256 I2C address
uint16_t eeprom_write_ptr = 0x0000;
```

```
void eeprom_write_block(uint16_t addr, uint8_t *data, uint8_t len) {
    I2C_Start();
    I2C_Write(EEPROM_ADDR);            // write mode
    I2C_Write(addr >> 8);              // MSB
    I2C_Write(addr & 0xFF);            // LSB
    for (uint8_t i = 0; i < len; i++)
        I2C_Write(data[i]);
    I2C_Stop();
    __delay_ms(10);  // Wait for write cycle
}
```

🧠 4. Prepare Data for Logging

📚 Convert float to int16 format (multiplied by 100)

c

CopyEdit

```
void log_to_eeprom(float acc[3], float jerk[3]) {
    uint8_t h, m, s;
    rtc_get_time(&h, &m, &s);

    uint8_t buffer[16];
    buffer[0] = bcd_to_dec(h);
    buffer[1] = bcd_to_dec(m);
    buffer[2] = bcd_to_dec(s);

    for (uint8_t i = 0; i < 3; i++) {
        int16_t a = (int16_t)(acc[i] * 100);
        int16_t j = (int16_t)(jerk[i] * 100);
        buffer[3 + i*2] = (a >> 8);       // high byte
        buffer[4 + i*2] = (a & 0xFF);     // low byte
        buffer[9 + i*2] = (j >> 8);       // high byte
        buffer[10 + i*2] = (j & 0xFF);    // low byte
    }

    eeprom_write_block(eeprom_write_ptr, buffer, 16);
    eeprom_write_ptr += 16; // Advance pointer
}
```

🔁 5. Example: Logging Inside Main Loop

c

CopyEdit

```
read_accel_xyz(&acc_now[0], &acc_now[1], &acc_now[2]);
// ... calculate jerk[0..2] ...
```

```
log_to_eeprom(acc_now, jerk);  // Save one line
```

> You could log every 100 ms (10 Hz), and 32KB EEPROM can store 2048 entries (i.e., ~3.5 minutes of continuous logging).

✅ Summary

- ✅ **RTC (DS3231)** provides timestamp (HH:MM:SS)
- ✅ **EEPROM (AT24C256)** logs 16 bytes per entry
- ✅ Data includes acceleration + jerk on all 3 axes
- ✅ Easy to offload and analyze later on PC

Full working mainloop

Here is a full working `main.c` firmware template for your **dsPIC33CK64MP506** project that:

✅ Reads LSM6DSR accelerometer
✅ Calculates **jerk** for all 3 axes
✅ Logs data with **RTC timestamp** to **EEPROM**
✅ Sends data via UART to a PC (for your Python GUI)

✅ Assumptions

- **I2C connected**: LSM6DSR (0x6A), DS3231 (0x68), AT24C256 (0x50 or 0xA0)
- **UART initialized** for PC communication (e.g., 9600 baud)
- Sampling rate: 100 ms (10 Hz)
- Using MPLAB X with XC16

📁 `main.c`

```
#include <xc.h>

#include <stdint.h>

#include <stdio.h>

#include <math.h>

#include <libpic30.h>  // For __delay_ms


#define _XTAL_FREQ 8000000UL  // 8 MHz for delays


// === I2C ADDRESSES ===

#define LSM6DSR_ADDR  0xD6  // Assuming SA0=1

#define RTC_ADDR      0xD0
```

```
#define EEPROM_ADDR   0xA0

// === GLOBAL ===
float acc[3], acc_prev[3], jerk[3];
uint16_t eeprom_ptr = 0x0000;

// === UART PRINT ===
void uart_write(char *s) {
    while (*s) {
        while (!U1STAbits.TRMT);
        U1TXREG = *s++;
    }
}

// === I2C Functions === (Assume already implemented)
void I2C_Start();
void I2C_Stop();
void I2C_Write(uint8_t);
uint8_t I2C_Read_ACK();
uint8_t I2C_Read_NACK();
void I2C_Restart();

// === RTC ===
uint8_t bcd_to_dec(uint8_t val) {
    return ((val >> 4) * 10) + (val & 0x0F);
}

void rtc_get_time(uint8_t *h, uint8_t *m, uint8_t *s) {
    I2C_Start();
    I2C_Write(RTC_ADDR);  // Write mode
    I2C_Write(0x00);      // Start from seconds
    I2C_Restart();
    I2C_Write(RTC_ADDR | 0x01);  // Read mode
    *s = I2C_Read_ACK();
    *m = I2C_Read_ACK();
    *h = I2C_Read_NACK();
    I2C_Stop();
}

// === EEPROM ===
void eeprom_write_block(uint16_t addr, uint8_t *data, uint8_t len) {
    I2C_Start();
    I2C_Write(EEPROM_ADDR);         // Device address
    I2C_Write(addr >> 8);           // MSB
    I2C_Write(addr & 0xFF);         // LSB
    for (uint8_t i = 0; i < len; i++)
        I2C_Write(data[i]);
    I2C_Stop();
    _delay_ms(10);
}

// === LSM6DSR ===
void lsm6dsr_init() {
    I2C_Start();
    I2C_Write(LSM6DSR_ADDR);
    I2C_Write(0x10);  // CTRL1_XL
    I2C_Write(0x60);  // 416 Hz, ±2g
    I2C_Stop();
}

void read_accel_xyz(float *x, float *y, float *z) {
    uint8_t xl, xh, yl, yh, zl, zh;
    int16_t raw_x, raw_y, raw_z;

    I2C_Start();
    I2C_Write(LSM6DSR_ADDR);
    I2C_Write(0x28);  // OUTX_L_A
    I2C_Restart();
    I2C_Write(LSM6DSR_ADDR | 0x01);
    xl = I2C_Read_ACK();
    xh = I2C_Read_ACK();
    yl = I2C_Read_ACK();
    yh = I2C_Read_ACK();

    zl = I2C_Read_ACK();

    zh = I2C_Read_NACK();

    I2C_Stop();


    raw_x = (int16_t)((xh << 8) | xl);

    raw_y = (int16_t)((yh << 8) | yl);

    raw_z = (int16_t)((zh << 8) | zl);


    *x = raw_x * 0.061;  // mg/LSB for ±2g

    *y = raw_y * 0.061;

    *z = raw_z * 0.061;

}


// === JERK ===

void calculate_jerk(float *curr, float *prev, float *jerk, float dt) {

    for (int i = 0; i < 3; i++) {

        jerk[i] = (curr[i] - prev[i]) / dt;

    }

}


// === LOGGING ===

void log_to_eeprom(float *acc, float *jerk) {

    uint8_t h, m, s;
    uint8_t buf[16];
    int16_t a, j;

    rtc_get_time(&h, &m, &s);
    buf[0] = bcd_to_dec(h);
    buf[1] = bcd_to_dec(m);
    buf[2] = bcd_to_dec(s);

    for (int i = 0; i < 3; i++) {
        a = (int16_t)(acc[i] * 100);
        j = (int16_t)(jerk[i] * 100);
        buf[3 + i * 2] = (a >> 8);
        buf[4 + i * 2] = (a & 0xFF);
        buf[9 + i * 2] = (j >> 8);
        buf[10 + i * 2] = (j & 0xFF);
    }

    eeprom_write_block(eeprom_ptr, buf, 16);
    eeprom_ptr += 16;
}

// === MAIN LOOP ===
int main(void) {
    // Init clocks, I2C, UART, etc.
    lsm6dsr_init();


    for (int i = 0; i < 3; i++) acc_prev[i] = 0;


    while (1) {
        read_accel_xyz(&acc[0], &acc[1], &acc[2]);
        calculate_jerk(acc, acc_prev, jerk, 0.1);


        log_to_eeprom(acc, jerk);


        char out[128];
        sprintf(out, "ACC: %.2f,%.2f,%.2f | JERK: %.2f,%.2f,%.2f\r\n",
                acc[0], acc[1], acc[2], jerk[0], jerk[1], jerk[2]);
        uart_write(out);


        for (int i = 0; i < 3; i++) acc_prev[i] = acc[i];
        __delay_ms(100);  // 10 Hz
    }
    return 0;
}
```

✅ Next Steps

- ✅ Load this firmware on your dsPIC
- ✅ Connect to your PC Python GUI — plots and logs live
- ✅ EEPROM stores history, even without USB connected

dump all logged EEPROM data over UART for later viewing or saving to a `.CSV` file from your PC.

✅ EEPROM Dump Overview

Each EEPROM entry is:

- 16 bytes:

  `[hh][mm][ss][axH][axL][ayH][ayL][azH][azL][jxH][jxL][jyH][jyL][jzH][jzL]`

We'll read back in 16-byte chunks and print them in human-readable format.

🧩 1. EEPROM Read Function

c

CopyEdit

```
void eeprom_read_block(uint16_t addr, uint8_t *data, uint8_t len) {
    I2C_Start();
    I2C_Write(EEPROM_ADDR);        // Write mode
    I2C_Write(addr >> 8);          // MSB
    I2C_Write(addr & 0xFF);        // LSB
    I2C_Restart();
    I2C_Write(EEPROM_ADDR | 0x01); // Read mode

    for (uint8_t i = 0; i < len - 1; i++)
        data[i] = I2C_Read_ACK();
    data[len - 1] = I2C_Read_NACK();

    I2C_Stop();
}
```

🧪 2. Dump EEPROM Entries Over UART

c

CopyEdit

```
void dump_eeprom_log() {
    uint8_t buf[16];
    char line[128];
    uart_write("Time,AccX,AccY,AccZ,JerkX,JerkY,JerkZ\r\n");

    for (uint16_t addr = 0x0000; addr < eeprom_ptr; addr += 16) {
        eeprom_read_block(addr, buf, 16);

        uint8_t h = buf[0], m = buf[1], s = buf[2];

        int16_t ax = (buf[3] << 8) | buf[4];
        int16_t ay = (buf[5] << 8) | buf[6];
        int16_t az = (buf[7] << 8) | buf[8];
        int16_t jx = (buf[9] << 8) | buf[10];
        int16_t jy = (buf[11] << 8) | buf[12];
        int16_t jz = (buf[13] << 8) | buf[14];

        sprintf(line,
"%02d:%02d:%02d,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f\r\n",
                h, m, s,
                ax / 100.0, ay / 100.0, az / 100.0,
                jx / 100.0, jy / 100.0, jz / 100.0);

        uart_write(line);
        __delay_ms(5); // Give time for UART
    }
    uart_write("=== End of Log ===\r\n");
}
```

🎯 How to Trigger Dump?

You can use a **button press**, **UART command**, or **timer**.

📍 Option A: UART Command

Add to main loop:

c

```
if (U1STAbits.URXDA) {
    char cmd = U1RXREG;
    if (cmd == 'D') {
        dump_eeprom_log();
    }
}
```

> 🔌 Then from your PC (e.g., Python or serial terminal), send a single `D` to download the full log.

✅ Output Format Example

makefile

```
Time,AccX,AccY,AccZ,JerkX,JerkY,JerkZ
```

```
10:25:01,0.12,0.08,-0.03,0.50,0.40,-0.10

10:25:01,0.13,0.09,-0.04,0.10,0.10,-0.01

...
```

Can be easily saved and graphed in Excel or your **Python GUI**.

updated `main.c` file with:

- ✅ Accelerometer & jerk reading
- ✅ Logging to EEPROM with RTC timestamp
- ✅ UART command `'D'` to **dump EEPROM** log to your PC in CSV format

---

✅ Final `main.c` for dsPIC33CK64MP506

🧠 Assumes:

- `LSM6DSR` at I2C `0xD6`,
- `DS3231 RTC` at `0xD0`,
- `AT24C256 EEPROM` at `0xA0`
- UART1 initialized at 9600 baud

- I2C functions (`I2C_Start`, `I2C_Write`, etc.) already implemented

c

```c
#include <xc.h>

#include <stdint.h>

#include <stdio.h>

#include <math.h>

#include <libpic30.h>


#define _XTAL_FREQ 8000000UL


#define LSM6DSR_ADDR   0xD6

#define RTC_ADDR       0xD0

#define EEPROM_ADDR    0xA0


float acc[3], acc_prev[3], jerk[3];

uint16_t eeprom_ptr = 0x0000;


// UART Write

void uart_write(char *s) {

    while (*s) {

        while (!U1STAbits.TRMT);

        U1TXREG = *s++;
    }
}

// RTC (DS3231)
uint8_t bcd_to_dec(uint8_t val) {
    return ((val >> 4) * 10) + (val & 0x0F);
}

void rtc_get_time(uint8_t *h, uint8_t *m, uint8_t *s) {
    I2C_Start();
    I2C_Write(RTC_ADDR);
    I2C_Write(0x00);
    I2C_Restart();
    I2C_Write(RTC_ADDR | 0x01);
    *s = I2C_Read_ACK();
    *m = I2C_Read_ACK();
    *h = I2C_Read_NACK();
    I2C_Stop();
}

// EEPROM
void eeprom_write_block(uint16_t addr, uint8_t *data, uint8_t len) {
    I2C_Start();
    I2C_Write(EEPROM_ADDR);
    I2C_Write(addr >> 8);

    I2C_Write(addr & 0xFF);

    for (uint8_t i = 0; i < len; i++) I2C_Write(data[i]);

    I2C_Stop();

    __delay_ms(10);

}


void eeprom_read_block(uint16_t addr, uint8_t *data, uint8_t len) {

    I2C_Start();

    I2C_Write(EEPROM_ADDR);

    I2C_Write(addr >> 8);

    I2C_Write(addr & 0xFF);

    I2C_Restart();

    I2C_Write(EEPROM_ADDR | 0x01);

    for (uint8_t i = 0; i < len - 1; i++) data[i] = I2C_Read_ACK();

    data[len - 1] = I2C_Read_NACK();

    I2C_Stop();

}


// LSM6DSR

void lsm6dsr_init() {

    I2C_Start();

    I2C_Write(LSM6DSR_ADDR);

    I2C_Write(0x10);     // CTRL1_XL
    I2C_Write(0x60);     // 416 Hz, 2g
    I2C_Stop();
}

void read_accel_xyz(float *x, float *y, float *z) {
    uint8_t xl, xh, yl, yh, zl, zh;
    int16_t raw_x, raw_y, raw_z;

    I2C_Start();
    I2C_Write(LSM6DSR_ADDR);
    I2C_Write(0x28);  // OUTX_L_A
    I2C_Restart();
    I2C_Write(LSM6DSR_ADDR | 0x01);
    xl = I2C_Read_ACK();
    xh = I2C_Read_ACK();
    yl = I2C_Read_ACK();
    yh = I2C_Read_ACK();
    zl = I2C_Read_ACK();
    zh = I2C_Read_NACK();
    I2C_Stop();

    raw_x = (int16_t)((xh << 8) | xl);
    raw_y = (int16_t)((yh << 8) | yl);
    raw_z = (int16_t)((zh << 8) | zl);
    *x = raw_x * 0.061;

    *y = raw_y * 0.061;

    *z = raw_z * 0.061;

}


// Jerk

void calculate_jerk(float *curr, float *prev, float *jerk, float dt) {

    for (int i = 0; i < 3; i++) {

        jerk[i] = (curr[i] - prev[i]) / dt;

    }

}


// EEPROM Logging

void log_to_eeprom(float *acc, float *jerk) {

    uint8_t h, m, s;

    uint8_t buf[16];

    int16_t a, j;


    rtc_get_time(&h, &m, &s);

    buf[0] = bcd_to_dec(h);

    buf[1] = bcd_to_dec(m);

    buf[2] = bcd_to_dec(s);
    for (int i = 0; i < 3; i++) {
        a = (int16_t)(acc[i] * 100);
        j = (int16_t)(jerk[i] * 100);
        buf[3 + i * 2] = (a >> 8);
        buf[4 + i * 2] = (a & 0xFF);
        buf[9 + i * 2] = (j >> 8);
        buf[10 + i * 2] = (j & 0xFF);
    }

    eeprom_write_block(eeprom_ptr, buf, 16);
    eeprom_ptr += 16;
}

// EEPROM Dump over UART
void dump_eeprom_log() {
    uint8_t buf[16];
    char line[128];

    uart_write("Time,AccX,AccY,AccZ,JerkX,JerkY,JerkZ\r\n");

    for (uint16_t addr = 0; addr < eeprom_ptr; addr += 16) {
        eeprom_read_block(addr, buf, 16);

        uint8_t h = buf[0], m = buf[1], s = buf[2];
        int16_t ax = (buf[3] << 8) | buf[4];
        int16_t ay = (buf[5] << 8) | buf[6];
        int16_t az = (buf[7] << 8) | buf[8];
        int16_t jx = (buf[9] << 8) | buf[10];
        int16_t jy = (buf[11] << 8) | buf[12];
        int16_t jz = (buf[13] << 8) | buf[14];

        sprintf(line,
"%02d:%02d:%02d,%.2f,%.2f,%.2f,%.2f,%.2f,%.2f\r\n",
                h, m, s,
                ax / 100.0, ay / 100.0, az / 100.0,
                jx / 100.0, jy / 100.0, jz / 100.0);
        uart_write(line);
        __delay_ms(5);
    }

    uart_write("=== End of Log ===\r\n");
}

// === Main ===
int main(void) {
    // Init UART, I2C, timers, etc.
    lsm6dsr_init();
    for (int i = 0; i < 3; i++) acc_prev[i] = 0;
```

```
while (1) {
    // UART dump command
    if (U1STAbits.URXDA) {
        char cmd = U1RXREG;
        if (cmd == 'D') {
            dump_eeprom_log();
        }
    }

    read_accel_xyz(&acc[0], &acc[1], &acc[2]);
    calculate_jerk(acc, acc_prev, jerk, 0.1);

    log_to_eeprom(acc, jerk);

    char out[128];
    sprintf(out, "ACC: %.2f,%.2f,%.2f | JERK: %.2f,%.2f,%.2f\r\n",
            acc[0], acc[1], acc[2], jerk[0], jerk[1], jerk[2]);
    uart_write(out);

    for (int i = 0; i < 3; i++) acc_prev[i] = acc[i];
    __delay_ms(100);
}
```

```
    return 0;
}
```

✅ What's Next?

From your PC (Python GUI or serial terminal like TeraTerm/Putty):

- Send `D` **over UART**
  ➡ EEPROM log prints in CSV format