ECSE 444 Project: Gaming Controller

Thibaut Chan Teck Su *Student ID: 261120277*
Faiza Ambreen Chowdhury *Student ID: 260952568*
Namir Habib *Student ID: 261115885*
Haoyuan Sun *Student ID: 261120493*

***Abstract*—This project aims to develop a Unity space-shooter game using a microcontroller as a motion controller. The microcontroller's accelerometer measures motion and estimates tilt angles, processed through a Kalman filter for enhanced accuracy. Real-time UART communication with Unity enables gameplay control, while a pushbutton triggers weapon firing with DAC-generated sound effects. The system demonstrates the potential for creating immersive gaming experiences using microcontroller-based motion controls.**

I. Introduction

The integration of motion-based controls has revolutionized the gaming industry, providing immersive user experiences. This project aims to create a Unity-based space-shooter game controlled by a microcontroller that serves as a motion controller. Key features include ADC for sensor data acquisition, UART for real-time communication with Unity, Kalman filter for raw data processing, and DAC for sound generation. The microcontroller detects player movements via its onboard accelerometer, processes this data through a Kalman filter for noise reduction, and includes a pushbutton for weapon triggering with DAC-generated sound effects.

This report outlines the design, implementation, and testing processes in building a responsive motion-controlled gaming system.

II. Motivation

Traditional gaming controllers rely on buttons, joysticks, and triggers for input, offering precise control but often lacking immersive interaction. These controllers can feel static and disconnected from the gameplay. In contrast, motion-based controls provide a more natural and intuitive experience, allowing players to interact with the game through physical movements like tilting or rotating the controller. *Table I* compares these controller with more traditional ones.

TABLE I
Comparison with Traditional Gaming Controllers

	Traditional Gaming Controllers	**Motion-Based Controllers**
Input	Buttons, joysticks	Gestures
Precision	High precision	Relies on filtering
Immersion	Static interaction	Natural interaction
User-experience	Fixed, repetitive inputs	Dynamic, interactive gameplay

III. Game Development

The space-shooter game was developed using Unity, a widely used cross-platform game engine for creating 2D and 3D interactive experiences with C#. To streamline the visual design, we utilized premade assets for the front end, allowing us to focus on functionality and gameplay mechanics. The game features a straightforward yet engaging gameplay loop, where players control a spaceship, search for specific objects in space, and shoot them to complete objectives.

The core challenge of this section was integrating the game with the hardware. Using the UART protocol, Unity communicated with the microcontroller to receive real-time motion data, enabling precise control of the spaceship. Tilt angles (pitch and roll) calculated by the microcontroller were mapped to the spaceship's movements, while a button press on the microcontroller triggered the weapon-firing mechanism. This integration ensured a smooth and responsive gaming experience.

IV. Technical Implementation

- System Initialization The system initialization process configures essential hardware components to make the system ready for operation:
  - `HAL_Init()`: Initializes the hardware abstraction layer (HAL), which includes setting up the system clock, configuring the Systick timer, and preparing the system for peripheral initialization.
  - `BSP_ACCELERO_Init()`: Initializes the accelerometer. The board support package (BSP) function initializes the accelerometer hardware, enabling the device to read accelerometer data for motion tracking.
  - `SystemClock_Config()`: Configures the system clock, setting up the core clock, peripheral clocks, and ensuring that the microcontroller runs at the desired frequency.
  - `MX_GPIO_Init()`: Configures general-purpose input/output (GPIO) pins.
  - `MX_DMA_Init()`: Initializes the Direct Memory Access (DMA) controller for data transfer without involving the CPU.
  - `MX_I2C2_Init()`: Initializes the I2C interface, which might be used to communicate with the accelerometer.

- `MX_USART1_UART_Init()`: Initializes the UART interface for serial communication with Unity.
- `MX_DAC1_Init()`: Initializes the Digital-to-Analog Converter (DAC), used to generate sound effects.
- `MX_TIM2_Init() and MX_TIM3_Init()`: Initialize timers TIM2 and TIM3 for time-based events.

- Arrays and variables are initialized to store sound waveforms, button states, and motion data:
  - acceleroVal[3]: An array used to store the raw accelerometer readings for the x, y, and z axes.
  - pitch, roll: Variables to store the calculated tilt angles (pitch and roll) based on the accelerometer readings.
  - output[100]: A character array used to format the output string containing tilt data and button state for transmission to Unity.
  - Kalman Filter Structures (kstate): roll_filter and pitch_filter are structures initialized with parameters for the Kalman filter (process noise q, measurement noise r, state estimate x, error covariance p, and Kalman gain k). These filters are used to reduce noise in the calculated pitch and roll angles.
  - Waveform Arrays (sawtoothWave_1, sawtoothWave_2, sawtoothWave_3): Arrays that store precomputed sawtooth waveforms, which will be used for generating sound effects.

  For full initialization steps see the code snippet in Appendix A.
- Waveform Calculation The code precomputes sawtooth waveforms of different lengths and stores them in the sawtoothWave_1, sawtoothWave_2, and sawtoothWave_3 arrays. This is done in three loops:
  - First waveform (sawtoothWave_1): The loop runs for 110 iterations, generating a waveform for a frequency of 400Hz. Each value is calculated by j* 37.
  - Second waveform (sawtoothWave_2): A similar loop is used for a shorter waveform (55 iterations), corresponding to a frequency of 800Hz. Each value is calculated by j* 74.
  - Third waveform (sawtoothWave_3): This waveform is calculated with even fewer iterations (28 iterations), generating a waveform for a frequency of 1575 Hz. Each value is calculated by j* 146.
- Sound Output Configuration Sound effects are generated dynamically using the DAC:
  - `HAL_DAC_Start(&hdac1, DAC_CHANNEL_1)` Starts the DAC to output sound on the specified channel. The DAC will be used to play the precomputed sawtooth waveforms.
  - `HAL_TIM_Base_Start_IT(&htim2) and HAL_TIM_Base_Start_IT(&htim3)`: Start the timers TIM2 and TIM3 in interrupt mode. These timers will control periodic events, such as triggering waveform playback and periodic switching of the active waveform.
- Main Loop (Real-Time Data Processing) Sound effects are generated dynamically using the DAC. Key points include:
  - BSP_ACCELERO_AccGetXYZ(acceleroVal): The accelerometer values (x, y, z axes) are retrieved and stored in the acceleroVal array. This data is used to calculate the pitch and roll angles.
  - Pitch and Roll Calculations: The pitch and roll angles are calculated using the atan2f() function, which computes the arctangent of two values (in this case, using accelerometer data for x, y, and z axes). The calculated angles are converted from radians to degrees.
  - The Kalman filter update function (kalmanFilter_update()) is called for both the pitch and roll filters. The function refines the tilt angle estimates by incorporating new accelerometer data and previous estimates.
  - The output string (output) is formatted with the refined roll and pitch values, along with the button state. This string is then transmitted to Unity using the HAL_UART_Transmit() function. The transmission is done via the UART interface.
  - HAL_Delay(10) introduces a delay of 10 ms between each iteration, ensuring that the system does not overwhelm the UART communication or accelerometer reading.

  See Appendix B for the implementation of the main loop.
- Pushbutton Interrupt for Weapon Firing
  - The interrupt triggers whenever the button is pressed, starting sound playback by enabling the DAC to output sound.
  - The state of the button (buttonPressed) is monitored and can be used to toggle between different sound effects or system modes.
- Kalman Filter for Noise Reduction The Kalman filter is a central component of the motion tracking system:
  - State Initialization: Each filter state includes parameters for process noise (q), measurement noise (r), estimate (x), and error covariance (p).
  - Update Function: The filter updates the tilt angles in each iteration, refining the estimates based on raw measurements and previous predictions. The Kalman Filter implementation is from Lab 1.
- Timers for Waveform Playback Timers are used for scheduling tasks, particularly sound playback:
  - TIM3: Handles periodic waveform switching, ensuring variation in sound effects during gameplay.
  - TIM2: Used for system-level timing tasks, such as sound wave playback intervals.

V. Configuration

- *General Purpose Input-Output (GPIO)*
  - Pushbutton connected to pin 13 of port C (PC13).
  - Configured pushbutton to GPIOEXTI13 (to use global interrupt)
  - GPIO mode of PC13 is External Interrupt Mode with Rising/Falling edge trigger detection (so we can create interrupts when button is pressed and released)
- *Inter-Integrated Circuit (I2C) Sensors*
  - The LSM6DSL sensor was used to retrieve data from the accelerometer.
  - The I2C2 was configured with the following pin assignments: PB10 for the Serial Clock Line and PB11 for the Serial Data Line.
- *Universal Asynchronous Receiver/Transmitter (UART)*
  - USART1 (Universal Synchronous/Asynchronous Receiver/Transmitter) was enabled and configured in asynchronous mode, which allows it to function as a UART for communication.
  - The following parameters were set to ensure proper transmission with the virtual COM port:
    - Baud Rate: 9600 bits per second.
    - Word Length: 8 bits (including parity).
    - Parity: 'none'.
    - Stop Bits: '1'.
  - Moreover, the PB6 and PB7 pins were respectively enabled for the TX (transmit data output) and RX (receive data input) lines of USART1.
- *Digital-to-Analog Converter (DAC)*
  - Out1 was enabled and set to "Only to external pin" for connecting the DAC output to a speaker.
  - The waveform and trigger settings are as follows:
    - The sample and hold mode are disabled to ensure the generation of smoother waveforms.
    - The output buffer was enabled to handle larger loads without signal degradation.
    - The user trimming was set to factory trimming to make use of pre-calibrated values from the manufacturing process.
    - The trigger was set to "Timer 2 Trigger Out event" to allow the DAC to update its output value with the timer.
  - The DMA settings are as follow:
    - A circular DMA request was enabled for continuous signal generation.
    - The data width was set to word.
- *Timer*
  - TIM2 is used as a trigger for DMA.
    - The counter period is calculated using the following equation:

$$\text{Counter Period} = \frac{\text{System Clock Frequency}}{\text{Sampling Frequency}}$$

With a system clock of 120 MHz and a sampling frequency of 44.1 kHz, the resulting counter period is 2721 clock cycles, which was configured in the timer.

    - The Trigger Event Selection (TRGO) is set to "Update Event", which enables the timer to create a trigger signal when the counter period is reached, resetting then the counter to zero
  - TIM3 is used as a global interrupt.
    - While holding down the button, it enables the sound to change when the timer interrupt occurs.
    - To achieve an interrupt every 100 ms, the counter period, with the equation mentionned above, is calculated to be 12 million cycles by using 100 ms (10 Hz) for sampling rate and 120 MHz for the system clock. However, since this value is too large, a pre-calibration is used to divide the system clock before feeding it to the timer and is calculated with the following equations:

$$\text{Timer Clock} = \text{System Clock Prescaler} + 1$$

$$\text{Counter Period} = \text{Timer Clock} \times \text{Sampling Frequency}$$

Setting the pre-calibration to 11999, the timer clock is reduced to 10 kHz. As a result, the counter period is 1000 clock cycles, which corresponds to an interrupt frequency of 10 Hz.

VI. Testing

TABLE II
Test Scenarios and Results

Test Scenario	Outcome	Pass/Fail
Accelerometer response	Pitch and roll angles updated as expected based on accelerometer values.	Pass
Kalman Filter	Tilt angles showed reduced noise after filtering.	Pass
Sound Generation	Sound output was triggered with each button press.	Pass
Pushbutton Interrupt	Weapon firing was triggered by button press with sound feedback.	Pass
Real time data transmission	Data was received by Unity with no significant delay.	Pass

Overall, the system passed all major functional and integration tests, with accurate motion tracking, real-time communication, sound output, and responsive gameplay controls.

VII. User Feedback

In addition to integration testing, user feedback was collected to assess the system's user experience, ease of use and overall enjoyment of the motion-based controls. A group of 10 test users played the space-shooter game and completed a short survey. The results of the survey are summarized in the table below:

TABLE III
User Feedback Survey Results

Aspect	Scale	Average Rating
Responsiveness	1 (Very Unresponsive) to 5 (Very Responsive)	4.83
Ease of Use	1 (Very Difficult) to 5 (Very Easy)	4.0
Enjoyment	1 (Not Enjoyable) to 5 (Very Enjoyable)	4.5
Immersion	1 (Not Immersive) to 5 (Very Immersive)	4.5

The survey focused on the following aspects, with the key findings summarized below:

- **Responsiveness:**
  Question: *How accurately did the motion controls translate your physical movements to in-game actions?*
  With an average score of 4.83/5, test users reported that the motion controls were highly accurate, contributing to a seamless gaming experience.
- **Ease of Use:**
  Question: *Were the controls intuitive and easy to learn?*
  While controls were generally intuitive, some users mentioned a slight learning curve while first adjusting to the mechanics, leading to an average rating of 4.0/5.
- **Enjoyment:**
  Question: *Rate your overall satisfaction with the game.*
  The majority of users found the gameplay engaging and fun, resulting in an average score of 4.5/5.
- **Immersion:**
  Question: *How was the level of engagement and sense of control provided by the gameplay?*
  The combination of responsive controls and interactive gameplay contributed to a highly immersive experience, with an average score of 4.5/5.

VIII. Conclusion

In this project we successfully developed a motion-based control system for a space-shooter game. Testing confirmed that the system met key functional requirements. Real-time data transmission between the hardware and Unity ensured smooth gameplay. User feedback highlighted the responsiveness of the motion controls, ease of use, and high levels of enjoyment and immersion. In conclusion, the system offers a highly engaging and intuitive gaming experience, with potential for further development.

Appendix

A. Initialization

```
MX_GPIO_Init();
MX_DMA_Init();
MX_I2C2_Init();
MX_USART1_UART_Init();
MX_DAC1_Init();
MX_TIM2_Init();
MX_TIM3_Init();
/* USER CODE BEGIN 2 */
int16_t acceleroVal[3];

float pitch, roll;

char output[100];

struct kstate {
    float q;
    float r;
    float x;
    float p;
    float k;
};
// Adjust parameters for precision
struct kstate roll_filter = {0.01f, 0.1f, 0.0f, 1.0f, 0.0f};
struct kstate pitch_filter = {0.01f, 0.1f, 0.0f, 1.0f, 0.0f};

// Sawtooth waveform generation
for (uint32_t j = 0; j < 110; j++){ // 400 Hz
    sawtoothWave_1[j] = j * 37;
}

for (uint32_t j = 0; j < 55; j++){ // 800 Hz
    sawtoothWave_2[j] = j * 74;
}

for (uint32_t j = 0; j < 28; j++){ // 1575 Hz
    sawtoothWave_3[j] = j * 146;
}

HAL_DAC_Start(&hdac1, DAC_CHANNEL_1);
HAL_TIM_Base_Start_IT(&htim2);
HAL_TIM_Base_Start_IT(&htim3);
```

B. Main Loop

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
    // Get accelerometer data (X, Y, Z axes) and store it in acceleroVal array
    BSP_ACCELERO_AccGetXYZ(acceleroVal);

    // Convert raw accelerometer readings to floating-point values for calculations
    float ax = (float)acceleroVal[0];
    float ay = (float)acceleroVal[1];
    float az = (float)acceleroVal[2];

    // Compute denominator for pitch calculation: sqrt(ay^2 + az^2)
    float pitch_denom = sqrtf(ay * ay + az * az);
    // Compute denominator for roll calculation: sqrt(ax^2 + az^2)
    float roll_denom = sqrtf(ax * ax + az * az);

    // Calculate pitch angle (in degrees) using arctangent formula
    pitch = atan2f(-ax, pitch_denom) * (180.0f / M_PI);
    // Calculate roll angle (in degrees) using arctangent formula
    roll = atan2f(ay, roll_denom) * (180.0f / M_PI);

    // Update Kalman filters for roll and pitch angles (only update if no floating p
    if (kalmanFilter_update(&roll_filter, roll) == 0) {
        roll = roll_filter.x;
    }
    if (kalmanFilter_update(&pitch_filter, pitch) == 0) {
        pitch = pitch_filter.x;
    }

    // Format the roll, pitch, and buttonPressed status into a string for UART trans
    sprintf(output, "Roll: %.2f, Pitch: %.2f, Button: %d\r\n", roll, pitch, buttonPr

    // Transmit the formatted string over UART (timeout of 10 seconds)
    int16_t len = strlen(output);
    HAL_UART_Transmit(&huart1, (uint8_t*)output, len, 10000);

    HAL_Delay(10);
}
/* USER CODE END 3 */
}

/**
```