SpacePilot Pro: STM32 Real-Time Controller
Group 2

廖盛弘
R10922135
資工所碩三

楊乃明
B09901174
電機系大四

周哲瑋
B09502132
機械系大四

ABSTRACT

This project aims to achieve precise control of the classic game Space Pilot using an STM32 microcontroller and an accelerometer, addressing the shortcomings of previous student projects such as simplistic game graphics and low requirements for real-time performance and precision. We employed the Unity platform and its free asset library to enhance the quality of the game graphics and utilized various technologies to optimize data transmission performance and real-time responsiveness. The system primarily involves accelerometer data acquisition, data transmission, and processing on the computer side, followed by system integration and testing. We designed a timer to periodically acquire accelerometer data and transmit it to the computer via Wi-Fi, where the data is then processed using Python scripts or directly within Unity. The test results indicate that selecting the appropriate tools and technologies is crucial for reducing latency, with the Queue module and pynput library demonstrating outstanding performance. Ultimately, we successfully implemented an efficient, low-latency game control system that meets high real-time and precision requirements, providing valuable insights for future embedded system development.

WORK DIVISION

In this section, we outline the contributions of each team member to provide a clear understanding of the distribution of tasks and responsibilities throughout the project.

Here is our github final project repo link : https://github.com/5000user5000/ESLab2024Spring/tree/main/final%20project

Table 1. Work division table

Member	Contribution
廖盛弘	1. **Unity Development**: Modify C# scripts in Unity to receive STM32 data and convert it into game control commands. 2. **C++ Script Testing**: Test and debug C++ scripts to ensure accurate and stable data transmission. 3. **Data Collection and Algorithm Adjustment**: Collect STM32 movement data and adjust movement algorithms to improve control precision and real-time response. 4. **Research I/O multiplexing** 5. **Network Delay time measurement**
楊乃明	1. **C++ Development**: Use Mbed Studio to develop C++ programs, extending from Assignment 2. 2. **Code Optimization**: Streamline original code and add Timer for stable data transmission. 3. **Python Support**: Assist in researching and modifying mouse control algorithms in Python to ensure precise and real-time spaceship movement.
周哲瑋	1. **Python Development**: Develop Python scripts to receive and process STM32 data, converting it into keyboard commands. Research and implement multithreading for improved performance. Utilize pynput for low-latency input event handling and Queue for thread-safe data structures. 2. **C++ Script Testing**: Test and debug C++ scripts to ensure accurate and stable data transmission 3. **Action Delay time measurement** 4. **Main writer of the report**

1 MOTIVATION

When reviewing the work of previous students, we noticed that many chose to develop games as their project. However, these games often had the following shortcomings:

Simplistic Game Graphics: Since these games were usually developed from scratch, the limited development time and resources resulted in simple and unattractive graphics.
Low Requirements for Real-Time Performance and Precision: Many of these games were casual, so the precision and real-time responsiveness of the game object movements were not a priority, requiring only basic operations.
To overcome these shortcomings, we decided to adopt different methods to improve the quality and challenge of our project. First, we chose to use the Unity platform and its rich library of free assets. This approach allowed us to significantly reduce the time and effort required for game development and focus on embedded system development. Additionally, the high-quality assets provided by Unity greatly enhanced the visual appeal and professionalism of the game.
Secondly, we selected the classic game Space Pilot as our project theme. This game requires players to navigate through various obstacles, with the speed increasing as the game progresses. This demands higher real-time performance and precision, meaning our control system must respond accurately and promptly to player inputs. This presented a significant challenge for our embedded system design and development. Our goal was to achieve a gameplay experience as smooth as directly operating the game on a computer.
By choosing a game that demands high precision and real-time performance, we aimed to showcase our capabilities in embedded system development and validate our system's performance under high-pressure and high-demand conditions. Additionally, we hope this project will provide future students with a higher-quality and more challenging example, inspiring more innovation and progress.

2 METHODS

In this project, we used an STM32 microcontroller to obtain accelerometer data, which controls the movement direction and distance of the game spaceship based on tilt direction and magnitude (gravitational acceleration). The implementation of the system consists of the following main parts:

2.1 Accelerometer Data Acquisition

We selected the STM32 microcontroller as the primary embedded platform. The accelerometer is connected to the STM32 via the I2C interface, continuously capturing the device's tilt data. The specific methods are as follows:
Tilt Detection: The accelerometer measures acceleration values on the XYZ axes. These values are used to calculate the device's tilt direction and angle.
Data Processing: The processed data is converted into specific movement commands, such as moving left, right, up, or down.

2.2 Data Transmission

To achieve real-time control, we designed a timer within the STM32 to periodically acquire accelerometer data and transmit it to the computer. The specific steps are as follows:
Using the Timer: The STM32 timer triggers an interrupt at fixed intervals (e.g., 10 times per second) to read the accelerometer data.
Wi-Fi Communication: Based on the instructor's recommendation, we chose Wi-Fi sockets for data transmission, as they theoretically offer faster speeds and lower latency compared to BLE (Bluetooth Low Energy). The processed tilt data is transmitted to the connected computer via Wi-Fi sockets.

2.3 Computer-Side Processing

On the computer side, we implemented two approaches to handle the data received from the STM32:
Approach 1 (Direct Processing in Unity): We modified the Unity C# game script to include socket communication and data processing algorithms. This approach integrates all processing within the game, resulting in lower latency. However, we put more focus on the next approach because python is easier to debug and we did not want to modify the original game scripts.
Approach 2 (Processing with Python): In this method, we used a Python script to open a socket and receive data from the STM32. The script processes the data to determine direction and distance, then simulates mouse movements to control the game spaceship. This approach simplifies the logic and facilitates debugging and modification.

2.4 System Integration and Testing

After developing each module, we integrated the STM32 hardware with the computer-side software to ensure continuous and stable data transmission and control flow.
System Integration: We combined the STM32 hardware and computer-side software to ensure the continuity and stability of data transmission and control processes.
Testing and Debugging: We tested system performance in actual gameplay, checking the real-time responsiveness and precision of spaceship movements. Based on the test results, we made adjustments and optimizations. Additionally, we measured latency to objectively assess the transmission efficiency.
Through these steps, we successfully used the STM32 microcontroller and accelerometer to achieve precise control of the game spaceship, meeting the high real-time and precision requirements.

3 BACKGROUND

3.1 Timer Mechanism

Using the Timer in the STM32 to acquire accelerometer data periodically and transmit it to the computer via Wi-Fi ensures the timing and stability of data acquisition. The main features are:

Timed Interrupts: The STM32 Timer can trigger interrupts at preset intervals, ensuring the stable and accurate acquisition of accelerometer data.

Real-time Performance: Using the Timer ensures real-time acquisition of accelerometer data, suitable for applications requiring high precision and low latency.

Stability: Timed interrupts avoid the instability of manually scheduled data acquisition, enhancing system reliability.

3.2 Wi-Fi Communication

Wi-Fi communication was chosen for its high speed and low latency, enabling fast and reliable data transmission to the computer, outperforming BLE (Bluetooth Low Energy). Below is the main features:

High Speed: Wi-Fi offers higher data transmission speeds, capable of quickly transmitting large amounts of data, suitable for high-bandwidth applications.

Low Latency: Compared to BLE, Wi-Fi transmission latency is lower, ensuring immediate data transmission, suitable for high real-time requirements.

Stability: Wi-Fi provides a more stable signal transmission, offering continuous and stable data connections, reducing data loss due to signal interruptions.

3.3 I/O multiplexing

I/O multiplexing offers several advantages over the traditional blocking I/O model, especially in scenarios where an STM32 frequently disconnects from a TCP server. Here are the main features and benefits of I/O multiplexing:

Flexible Monitoring: I/O multiplexing can monitor multiple file descriptors interested simultaneously, allowing the TCP server to listen to multiple client sockets at once. This flexibility ensures that the server is not restricted to a single client connection.

Efficient Resource Utilization: Unlike the non-blocking I/O model, which can lead to busy waiting, I/O multiplexing is a busy-wait-free approach. This means the process does not continuously occupy the CPU to check for status changes.

3.4 Queue Module

The Queue module in the Python standard library is designed for safe data exchange and communication in multithreaded environments. It provides a first-in, first-out (FIFO) data structure with built-in multithreading support, avoiding race conditions that can occur with manually managed data structures. Below is the main features:

Thread Safety: The Queue module uses internal locks to ensure safe data operations in multithreaded environments, preventing data competition and corruption.

Multiple Modes: The Queue module offers FIFO, LIFO (last-in, first-out), and priority queue modes to meet different application needs.

Blocking Operations: The Queue module supports blocking operations, such as the put() and get() methods, which can be set with timeout parameters to prevent threads from waiting indefinitely.

Ease of Use: Using the Queue module simplifies multithreaded programming, eliminating the need for manual lock management and synchronization, reducing code complexity and potential errors.

3.5 Pyautogui Library

Pyautogui is a Python library for controlling the mouse and keyboard, suitable for automating graphical user interface (GUI) operations. It provides a series of simple and easy-to-use functions to simulate user operations, such as moving the mouse, clicking, and typing. Below is the main features:

Cross-Platform Support: pyautogui supports Windows, macOS, and Linux, offering good compatibility.

Simplicity and Intuitiveness: It provides a straightforward API that allows quick implementation of basic GUI automation tasks.

Image Recognition: Pyautogui supports image-based screen recognition and interaction, suitable for tasks that require interaction with specific interface elements.

Limitations: Despite its usefulness, pyautogui has several limitations. Its performance can be slow for high-frequency input operations or real-time applications due to its reliance on higher-level OS operation. It will restrict its effectiveness in time-sensitive applications. Most operations in pyautogui are blocking, meaning the program execution halts until the operation completes, which can affect the performance and responsiveness of automation scripts.

3.6 Pynput Library

Pynput is a Python library specifically designed for monitoring and controlling input devices (keyboard and mouse), aimed at providing efficient, low-latency input control. Below is the main features:

High Performance: Pynput focuses on input device control, optimizing the handling of a large number of input events to provide extremely low latency.

Flexibility: Pynput supports all major functions of keyboard and mouse monitoring and control, allowing easy extension and customization.

Non-Blocking Operations: Pynput supports non-blocking operation modes, allowing simultaneous monitoring of multiple input events without blocking the main program.

3.7 Multithreading

Multithreading is a concurrent programming technique that allows a program to run multiple threads simultaneously, each of which is an independent execution unit within the program. Multithreading leverages multi-core processor performance, improving program execution efficiency and responsiveness. Below is the main features:

Concurrent Execution: Multithreading allows multiple tasks within a program to run concurrently, reducing wait times and improving overall efficiency**.**

Resource Sharing: Threads can share resources (such as memory) within the same process, facilitating data exchange and communication.

Asynchronous Operations: Multithreading enables asynchronous operations, such as handling I/O operations in one thread while other tasks continue in another, avoiding main program blockage.

3.8 Thread Pool

Thread Pool is a design pattern used to manage and reuse a set of pre-created threads. Its purpose is to reduce the overhead of creating and destroying threads and effectively manage thread usage. Below is the main features:

Resource Reuse: Thread Pool reuses pre-created threads to execute multiple tasks, reducing the overhead of creating and destroying threads.

Load Balancing: Thread Pool usually has a load balancing mechanism that reasonably distributes tasks among threads, preventing certain threads from being overloaded.

Simplified Management: Thread Pool provides a unified interface for submitting tasks, simplifying the design and management of multithreaded programs.

4 EVALUATION

To evaluate the performance of our control system, we shifted our focus from subjective game scores to an objective metric: we measured the **action delay**, which is the time taken from when Python receives the information to when the action is completed. We also measured the **network delay** to observe the total latency from the STM32 transmitting data to the computer. This comprehensive approach allowed us to analyze the overall system delay accurately.

For action delay, we conducted a series of tests using different Python methods to measure data transmission latency, focusing on both dual-thread and single-thread scenarios. These tests utilized different libraries (pyautogui and pynput) and synchronization mechanisms to compare their performance. The average, maximum, and minimum delays were recorded for each test to comprehensively evaluate the performance of various methods.

By combining these metrics, we obtained a holistic view of the system's performance. This detailed evaluation not only highlighted the differences in performance across various methods and tools but also provided valuable insights into optimizing real-time control systems in embedded applications.

4.1 Environmental Setup

CPU: AMD Athlon x4 845
RAM: 12.0 GB DDR3
STM32: B-L475E-IOT01A1

4.2 Action Delay With Pyautogui

In the dual-thread scenario using **pyautogui**, we observed significant delays, with an average delay of 3.977909 seconds, a maximum delay of 7.707328 seconds, and a minimum delay of 0.271053 seconds. Replacing manual list management with the Queue module reduced these delays substantially, resulting in an average delay of 0.585117 seconds, a maximum delay of 0.902399 seconds, and a minimum delay of 0.110214 seconds. This improvement highlights the importance of using thread-safe data structures to avoid race conditions and enhance performance.

4.3 Action Delay With Pynput

Switching to the **pynput** library, we observed further improvements. The dual-thread scenario with pynput and thread pool yielded an average delay of 0.000598 seconds, a maximum delay of 0.014850 seconds, and a minimum delay of 0.000000 seconds. Without the thread pool, the average delay was 0.000490 seconds, the maximum delay was 0.006995 seconds, and the minimum delay remained at 0.000000 seconds. In single-thread scenarios, pynput consistently outperformed pyautogui, with significantly lower average, maximum, and minimum latencies.

4.4 Network Delay

Next, we measured the delay time for transmitting data from the STM32 microcontroller to the computer. Initially, we used the interval time of data reception in Python for calculating the delay time. This method yielded a measurement of 0.69 seconds. However, this value is inaccurate. As illustrated in Figure 2 , this approach cancels out the propagation delay [4] and transmission delay, leaving only the computer’s processing and queuing delay. Consequently, the delay time is underestimated.

Subsequently, we refined our method by considering the time at which data was sent from the STM32 and the time at which it was received by the computer. The interval divided by two gives the actual total delay time, which is 26.506 milliseconds. Detailed information on this method can be found in the TestLatency branch of our GitHub repository.

4.5 Discussion

The experimental results demonstrate that the choice of libraries and synchronization mechanisms has a profound impact on the performance of the control system. The pyautogui library, while simple and easy to use, exhibited considerable delays in dual-thread scenarios, especially when manual list management was used.

A key optimization involved replacing Python's built-in list with the **Queue** module. The primary reason for this change is that Python lists are implemented as dynamic arrays, which pose a significant overhead in multi-threaded environments, especially in producer-consumer patterns. When using a list, the consumer thread repeatedly removes the top element, necessitating the shifting of subsequent elements forward. This continuous element rearrangement results in substantial overhead. In contrast, the Queue module, specifically designed for thread-safe, eliminates this overhead by efficiently handling element removal without the need for shifting.

The pynput library outperformed pyautogui in all tested scenarios, showcasing its suitability for high-frequency, real-time input control. The non-blocking operations and low-latency handling of input events in pynput proved crucial for achieving the desired responsiveness in our system. Additionally, using double threads with pynput further optimized performance by efficiently managing concurrent tasks.

Our findings suggest that for applications requiring real-time performance and precision, such as game control systems, the pynput library combined with thread-safe data structures and multithreading techniques offers a superior solution. The results also highlight the importance of careful library selection and synchronization mechanism implementation in developing high-performance, responsive systems. These insights provide valuable guidance for future embedded system development projects, emphasizing the need for efficient input handling and data processing to meet stringent real-time requirements.

Table 1. Action Delay table

Scenario	Avg Delay (seconds)	Max Delay (seconds)	Min Delay (seconds)
py single-thread (pyautogui)	0.123027	0.367265	0.105081
py dual-thread (pyautogui+ List)	3.977909	7.707328	0.271053
py dual-thread (pyautogui+Queue)	0.585117	0.902399	0.110214
py single-thread (pynput)	0.043509	0.103515	0.020854
py dual-thread (pynput+List)	0.000516	0.008003	0
py dual-thread (pynput+Queue)	0.00049	0.006995	0
py dual-thread (pynput+Queue + thread pool)	0.000598	0.01485	0

Figure 1. Average delay comparison figure

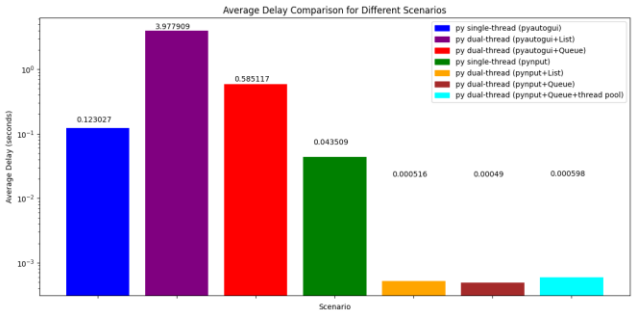
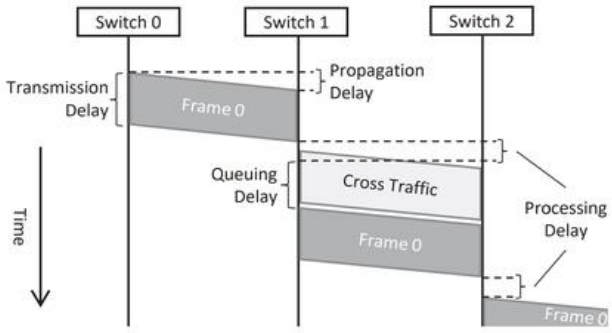


Figure 2. Network delay depiction figure



5 CONCLUSION

This project demonstrates the capability of an embedded system in high-pressure, high-demand environments by implementing a game control system based on the STM32 microcontroller. We utilized accelerometer data to control the movement direction and distance of the game spaceship, ensuring real-time and stable data acquisition and transmission through the Timer mechanism and Wi-Fi communication. Our comprehensive evaluation revealed significant differences in performance between various Python libraries and synchronization mechanisms.

Initially, our code suffered from frequent disconnections due to poor connection quality, requiring a complete restart each time, which was highly inefficient. By incorporating I/O multiplexing, the STM32 was able to automatically reconnect after a disconnection, greatly enhancing development efficiency and user convenience.

We also measured the network delay and action delay. The results indicate that using the Queue module instead of manually managing lists effectively avoids data race conditions and significantly reduces latency. Specifically, the pyautogui library, while easy to use, exhibited considerable delays, particularly in dual-thread scenarios with manual list management. By contrast, the pynput library consistently outperformed pyautogui, demonstrating its superiority in handling high-frequency, real-time input events.

Furthermore, implementing multithreading and thread pool techniques enhanced the system's parallel processing capability and responsiveness, leading to lower latency and improved real-time performance. The combination of pynput with thread-safe data structures and efficient threading models proved to be the most effective solution for achieving the desired level of control precision and responsiveness.

Our game's delay time, including both network and action delays, was approximately 0.033 seconds (0.026 + 0.007 seconds), which is a substantial improvement over our initial target of 1 second.

In addition to the substantial reduction in objective latency times, the subjective gaming experience also significantly improved compared to earlier progress reports. The gameplay now closely mimics the experience of using a physical mouse, offering a highly responsive and precise control system. Players can execute more immediate, intricate, and complex maneuvers, making it possible to avoid previously unavoidable obstacles. This enhanced control translated into a remarkable increase in gameplay scores, from 3,000 to over 300,000 points, demonstrating the dramatic improvement in user experience.

Our project, compared to previous students' work, benefits from the use of free assets from Unity, resulting in better graphics and game design. Additionally, we chose to tackle the Space Pilot game, which demands high precision and low latency, presenting a greater challenge than casual games or simple side-scrollers.

Ultimately, we successfully implemented an efficient, low-latency game control system that meets the high real-time and precision requirements of the Space Pilot game. This project not only highlights the importance of selecting appropriate tools and technologies for embedded system development but also provides a high-quality and challenging example for future projects. Our findings and methodology offer valuable insights and inspiration for future innovation and progress in the field of embedded systems.

REFERENCE

[1] *pyautogui's documentation!*¶. Welcome to PyAutoGUI's documentation! - PyAutoGUI documentation. (n.d.). https://pyautogui.readthedocs.io/en/latest/

[2] *Pynput documentation*. pynput Package Documentation - pynput 1.7.6 documentation. (n.d.). https://pynput.readthedocs.io/en/latest/#

[3] Brownlee, J. (2023, November 23). Python threadpool: The CompleteGuide Super Fast Python. https://superfastpython.com/threadpool-python

[4] Wikimedia Foundation. (2023, June 19). Network delay. Wikipedia. https://en.wikipedia.org/wiki/Network_delay