The Chinese University of Hong Kong
Department of Computer Science and Engineering

CENG2400 Embedded System Design

Final Project - Laser Turret

Report

Project Demo Date: 13 Dec 2024

Project Report Deadline: 20 Dec 2024 23:59

1155193237 - Yu Ching Hei (chyu2@cse.cuhk.edu.hk)
1155223226 - Tam Yiu Hei (1155223226@link.cuhk.edu.hk)
1155194650 - Leung Chung Wang (1155194650@link.cuhk.edu.hk)

Source code and project history is available on GitHub:
`https://github.com/Jellyfish227/CENG2400_Embedded_System_Design.git`

Under the kind guidance of
Prof. Ming-Chang YANG

Also thank you for the help provided by teaching assistants
Mr. Chenchen ZHAO
Mr. Kezhi LI
Mr. Han ZHAO
Mr. Zhirui ZHANG

Contents

1 Introduction

The CENG2400 final project involved implementing a gyroscope-controlled laser turret that uses the MPU6050 to direct the laser. Our goal was to create a responsive turret for maximizing hits.
The workflow consists of two phases:

1. **Master End:** We collect and process gyro and accelerometer data from the MPU6050, then transmit the calculated angles via UART.

2. **Slave End:** The angles are received and converted into PWM signals to control the servo motors.

This report outlines our challenges faced and insights gained during implementation, aiming to deliver a functional motion-controlled laser turret.

2 Division of Work

Task	Person In Charge
Master End	**Yu Ching Hei, Tam Yiu Hei, Leung Chung Wang**
Slave End	**Yu Ching Hei, Tam Yiu Hei**
Report	**Yu Ching Hei, Tam Yiu Hei, Leung Chung Wang**

3 Chanllenges during implementation

3.1 Transmission of Data

3.1.1 Master end: First edition of data transmission formatting

```
/* Commit hash: c6e8ed6, main_mpu.c*/
void sendData(float yawAngle, float pitchAngle) {
    char data[22];
    // format the data as a string
    sprintf(data, "%.10f,%.10f", yawAngle, pitchAngle);
    char* chp = data;
    while (*chp)
        UARTCharPut(UART5_BASE, *chp++);
}
```

In this initial implementation, we formatted the yaw and pitch angles into a string using sprintf. While this allowed us to send the angles with high precision (10 decimal places), the long string to be sent over UART wirelessly increases the risk of data loss. And turns out during the testing, we found that the data received was very unstable, we have presumed that the problem is due to the data being sent is too long, which means there are more characters that can potentially be lost, and just one character lost can cause the whole data to be corrupted.

3.1.2 Master end: Second edition of data transmission formatting

```
/* Commit hash: af09000, main_mpu.c */
void sendData(float yawAngle, float pitchAngle) {
    char data[7];
    sprintf(data, "%03d%03d\0", (int)yawAngle, (int)pitchAngle);
    /* same send looping as before*/
    UARTCharPut(UART5_BASE, '\n'); // '\n' to mark end of transmission
}
```

In this update, we have made 2 important changes:

1. We have changed the data string to be a string of 7 characters, which is a lot shorter than the previous 22 characters.

2. We have added a newline character to indicate the end of the transmission, which is a simple way to check the data integrity.

We decided to sacrifice the precision of the data to just send the integral part of the angle, in order to reduce the risk of data corruption due to data loss during transmission. And in the previous implementation, we found that we have no way to check the data integrity, so we have to add a newline character to indicate the end of the transmission. However, the problem was not fixed, in the slave end, we found that the data received was still very unstable, we noticed that sometimes the data received would swap their positions(i.e. the yaw angle would be the pitch angle and vice versa), or sometimes some digits that should be in the pitch angle would appears in the array of yaw angle characters received. Therefore, we started to investigate in the slave end.

3.1.3 Slave end: wrongly assumed the problem is during data unpacking

```
/* Commit hash: 7f6934b, main_servo.c */
void UART5IntHandler(void) {
    uint32_t ui32Status = UARTIntStatus(UART5_BASE, true);
    UARTIntClear(UART5_BASE, ui32Status);
    uint32_t charCount = 0;
```

```
    while (UARTCharsAvail(UART5_BASE)) {
        char b = UARTCharGet(UART5_BASE);
        if (charCount < 3) {
            charYaw[charCount] = b;
        } else if (charCount < 6) {
            charPitch[charCount - 3] = b;
        }
        charCount++;
    }
}
```

In the original version of slave end code, we used an interrupt to receive data from the UART. We just called the interrupt handler to put the received character into two separate arrays. And we traslate the character array into integer in the main loop by calling atoi(). When we are testing the code in debug mode, we found that the integral data was wrong, so we assumed it is the problem to translate it separately in other loop, so we moved the

```
degreeArr[0] = atoi(charYaw);
degreeArr[1] = atoi(charPitch);
```

into the interrupt handler, and we found that the integral data was still incorrect, which means that the wrong data was not due to the conversion process but the character receiving process instead.

3.1.4 Slave end: investigating receive logic by locking or raising flags

```
/* Commit hash: , main_servo.c */
void UART5IntHandler(void) {
    uint32_t ui32Status = UARTIntStatus(UART5_BASE, true);
    UARTIntClear(UART5_BASE, ui32Status);
    isFinished = 0; // Reset flag

    uint32_t charCount = 0;
    if (processLock) {
        char b = UARTCharGet(UART5_BASE);
        charCount++;
    }
    isFinished = 1;
    processLock = 0; // Release lock
}
```

In this attempt, we introduced a locking mechanism to handle data reception more effectively. This approach aimed to prevent data loss by ensuring that the system only processed one set of data at a time. However, we still faced issues with data integrity, especially when the UART buffer was filled with new incoming data before the previous data was fully processed.

3.1.5 Slave end: use polling to receive data instead of interrupt

```
/* Commit hash: , main_mpu.c */ //FIXME: wrong code included
void sendData(int yawAngle, int pitchAngle) {
    char data[8]; // Increased buffer size for safety
    while (!UARTSpaceAvail(UART5_BASE)) {}
```

```
    snprintf(data, sizeof(data), "%03.0f,%03.0f", roundf(yawAngle),
    roundf(pitchAngle));
    char *chp = data;
    while (*chp) {
        UARTCharPut(UART5_BASE, *chp++);
    }
    UARTCharPut(UART5_BASE, '/'); // End of data
}
```

We decided to use a polling mechanism instead of relying on interrupts to avoid data integrity issues. By sending the yaw and pitch angles as ASCII characters followed by a delimiter (/), we aimed to ensure that each data packet was correctly terminated. However, ensuring that no data was lost during transmission remained a challenge.

3.1.6 Slave end: data integrity checking

```
/* Commit hash: , main_servo.c */
while (true) {
    while (UARTCharsAvail(UART5_BASE)) {
        char b = UARTCharGet(UART5_BASE);
        degreeArr[isPitch] = b;
        isPitch = !isPitch;
        idx++;
        if (b == '/' && idx % 3 != 0) {
            // Logic to ensure data is valid
        }
    }
}
```

In this code, we implemented checks to verify the integrity of received data. We ensured that only complete and correctly formatted data packets were processed by checking for start and end indicators. However, we still faced challenges in ensuring that the data received from the MPU6050 sensor was not corrupted by noise, often leading to missed or incorrect angle readings.

3.1.7 Final adjustment

```
/* Commit hash: , main_mpu.c */
char const START_INDICATOR = 254;
char const END_INDICATOR = 255;
void sendData(int yawAngle, int pitchAngle) {
    while (!UARTSpaceAvail(UART5_BASE)) {}
    UARTCharPut(UART5_BASE, START_INDICATOR);
    UARTCharPut(UART5_BASE, (yawAngle >> 8) & 0xFF);
    UARTCharPut(UART5_BASE, yawAngle & 0xFF);
    UARTCharPut(UART5_BASE, (pitchAngle >> 8) & 0xFF);
    UARTCharPut(UART5_BASE, pitchAngle & 0xFF);
    while (!UARTSpaceAvail(UART5_BASE)) {
        UARTCharPut(UART5_BASE, END_INDICATOR);
    }
}
```

In the final adjustments, we emphasized starting and ending indicators for each data transmission. By sending the yaw and pitch angles as separate bytes, we improved the clarity of the data being transmitted. This method aimed to reduce confusion and ensure

that the receiving end could accurately reconstruct the angles. Through this iterative process, we learned the importance of robust error handling and effective communication strategies in embedded systems.

3.2 MPU Connection

In our project, we faced challenges due to noise interference in receiving data from the MPU6050 sensor. The flag g_bMPU6050Done is used to indicate when the sensor has been successfully initialized, but due to noise, this flag often failed to be set, causing the system to hang indefinitely.

```
/* Commit hash: , main_mpu.c */
g_bMPU6050Done = false;
MPU6050Init(&sMPU6050, &g_sI2CMSimpleInst, 0x68, MPU6050Callback, &
    sMPU6050);
while (!g_bMPU6050Done) {
    // Waiting for initialization to complete
}
```

On the above code, the while loop waits for g_bMPU6050Done to turn true. However, due to noise, this flag might never be set, leading to a situation where the system appears unresponsive. To mitigate this issue, we implemented a delay of 500 ms. This was chosen through binary search. A value too low results in unresponsive behavior, while a value too high make the system too sensitive to noise. Ultimately, this approach ensures reliable data reception from the MPU6050, balancing responsiveness and stability in the presence of noise.

4 Conclusion

To sum up, our last work was to create a working prototype of a laser turret that utilizes a gyroscopic system based off the MPU6050 sensor and the Tiva C LaunchPad. This achievement was possible due to the structured sequence of activities which included collecting, processing and transmitting data in such a way that the system was able to point a laser at any direction accordingly to the movement.

Even with problems arising with data consistency, the timing of communications and noise sources, we were still able to rectify problems by adopting a more pliable approach to problems during the iterations which then greatly increased system performances. The given project increased our level of development in embedded systems, while participating as well and working in a team. Overall, the hardware-software integration in mobile systems was a great experience for us, and, no doubt, will lay down a solid base for us in similar projects shortly.