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 v1 of data sending formatting(float string)

```
/* 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 v2 of data sending formatting(int string)

```
/* 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: try modifying data unpacking timing

```
/* 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: try raising flag after finished receiving

```
/* Commit hash: 2f0f60e, main_servo.c */
/* in the main while loop */
while (true) {
    if (isFinished) {
        degreeArr[0] = atoi(charYaw);
        degreeArr[1] = atoi(charPitch);
        yaw_duty_cycle = angleToPWMDutyCycle(degreeArr[0]);
        PWMPulseWidthSet(PWM1_BASE, PWM_OUT_1, PWMGenPeriodGet(
    PWM1_BASE, PWM_GEN_0) * yaw_duty_cycle);
        pitch_duty_cycle = angleToPWMDutyCycle(degreeArr[1]);
        PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, PWMGenPeriodGet(
    PWM1_BASE, PWM_GEN_0) * pitch_duty_cycle);
    }
}

/* other code in the middle */

void UART5IntHandler(void) {
    uint32_t ui32Status = UARTIntStatus(UART5_BASE, true);
    UARTIntClear(UART5_BASE, ui32Status);
    isFinished = 0;
    uint32_t charCount = 0;
    /* same while loop as before */
    isFinished = 1;
}
```

In this attempt, we added a flag called `isFinished` to indicate the end of the data receiving process. And theoretically, the instructions that write the angle to the servo motors are only executed when `isFinished` is true. However, we found that the data

received was still unstable, and the servo motors were still having weird movements. Starting from this attempt, we are investigating a new way to transfer data, which could be more stable.

3.1.5 v3 of data sending formatting(ascii encoding)

```
/* Commit hash: 5a403e6, main_mpu.c */
void sendData(int yawAngle, int pitchAngle) {
    int putA[2];
    putA[0] = yawAngle;
    putA[1] = pitchAngle;
    int i = 0;
    while(i < 2) {
        while(!UARTSpaceAvail(UART5_BASE)) {}
        UARTCharPut(UART5_BASE, putA[i++]);
    }
}
```

After previous attempts, we decided to choose a new way to send data, which could minimize the steps for encoding and decoding the trasmitting data. Inspired by the mechanism of C language handling characters as a 1-byte integer, we know that a character can represent a number from 0 to 255. Which our angle data is 0°-180° for yaw and 15°-90° for pitch, which is well within the range of a character. Therefore, we can send the yaw and pitch angles as two characters, which is a lot more concise and atomic to handle during the transmission. Moreover, we no longer need to do extra steps to encode and decode the data, which reduces the overhead of both the master and slave ends. After implementing this version, the data received was much more stable, and the servo motors were able to move to the correct positions most of the time.
However, we found that the data received was still not 100% stable, and the servo motors would sometimes flick even if the mpu was not moved. We noticed that it may have received the tilt data as the yaw angle, since the minimum tilt angle to send is 20°

3.1.6 Data integrity checking

```
/* Commit hash: ceb7c95, main_servo.c */
/* In the main while loop*/
idx = 0;
while (UARTCharsAvail(UART5_BASE) || b != END_INDICATOR) {
    b = UARTCharGet(UART5_BASE);
set:
    receiveArr[idx] = b;
    idx = (idx + 1) % 4;
}
// check data validity
if (receiveArr[0] == START_INDICATOR && receiveArr[3] == END_INDICATOR)
    isValid = 1;
else {
    while (UARTCharsAvail(UART5_BASE)) {
        char temp = UARTCharGet(UART5_BASE);
        if (temp == START_INDICATOR) {
            idx = 0;
            clearArray(receiveArr);
            goto set;
        }
    }
```

```
}
if (isValid) {
    /* write data to servo motors */
}
```

In this part, we aimed to ensure that each data packet has a clear start and terminate with an indicator. Therefore we added two constants in the beginning of both ends,

```
char const START_INDICATOR = 254;
char const END_INDICATOR = 255;
```

Which these two indicators are not going to be messed up with the data since the range is 0°-180° for yaw and 15°-90° for pitch. In the slave checking logic, we are not terminating the data receiving process until we receive the `END_INDICATOR` and all data received, then we check if both the start and end indicators are correct, if not, we will wait until the next `START_INDICATOR` to restart the data receiving process by jumping back to the loop. The data will only be written to the servo motors when the data is valid.

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: effa073, main_mpu.c */
g_bMPU6050Done = false;
MPU6050Init(&sMPU6050, &g_sI2CMSimpleInst, 0x68, MPU6050Callback, &
    sMPU6050);
while (!g_bMPU6050Done) {
    // Waiting for initialization to complete
}
```

We are aware of the problem, and we know that if we are using the method of manipulating I2C instead of using library would mitigate the issue, but since we have implemented the whole project already, we afraid migrating the implementation would be time-consuming and raise incompatibilities. So we just flash the code until it works. Haha

4 Conclusion

To sum up, we have successfully implemented a responsive laser turret that can capture the motion of hand and reflect it to the laser turret wirelessly.
Before working on this project, we have expected that the most challenging part would be the motion capturing part. But surprisingly, we found that the most challenging part was actually the data transmission part, which we have to ensure the data integrity and stability. Through this iterative process of design, implementation, and debugging, we have gained valuable insights into embedded systems development. These experiences and lessons learned will prove invaluable for future embedded systems projects encountered.