

Robotics practicals TP8 : Programming and characterization of a modular fish robot

Nathann Morand 296190

Felipe Ramirez 331471

Mattéo Berthet 310443

avril 2025



Contents

1	First program (1 point)	3
1.1	task	3
1.2	Expected Code behaviour	3
1.3	Comparison	4
1.4	Code modification, blink	4
2	Registers (5 points)	5
2.1	Task	5
2.2	Register Communication	5
2.3	Expected Code behaviour	7
2.4	Comparison	7
3	Communication with Other Elements (3 points)	8
3.1	Task	8
3.2	CAN Bus Communication	8
3.3	Expected Code behavior	8
3.4	Comparison	8
3.5	DOF Position Reading Implementation	9
3.6	Testing	10
4	Position Control of a Module (3 points)	11
4.1	Task	11
4.2	Verification and Basic Control	11
4.3	Sine Wave Setpoint Implementation	11
4.4	Testing and Challenges	14
5	Trajectory Generation and Control (3+3 points)	14
5.1	Task	14
5.2	Sine-Wave Generation on the Microcontroller	14
5.3	Testing and Motor Control	14
5.4	Modulating Trajectory Parameters	14
5.5	Implementation	14
5.6	Testing and Observations	19
6	Tracking System	19
6.1	Task	19
6.2	Setup and introduction	19
6.3	Implementation of LED color based on position	19
6.4	Observations and result	21
7	Swimming and Experiments (10 points)	22
7.1	Task	22
7.2	Writing a Swimming Trajectory Generator	22
7.3	Swimming speed measurement	22
7.4	Result	23
8	Speed and steering control	24
8.1	Task	24
8.2	Result	24
9	appendix	24

1 First program (1 point)

1.1 task

We are tasked with understanding the code that is provided for the first question, flash it on the robot and compare the behavior to our prediction. Finally, we are tasked to modify the code to make the green LED blink at 1Hz

1.2 Expected Code behaviour

the code will use the RGB LED to display the following sequence. After approximately 7.5 second, the cycle repeat :

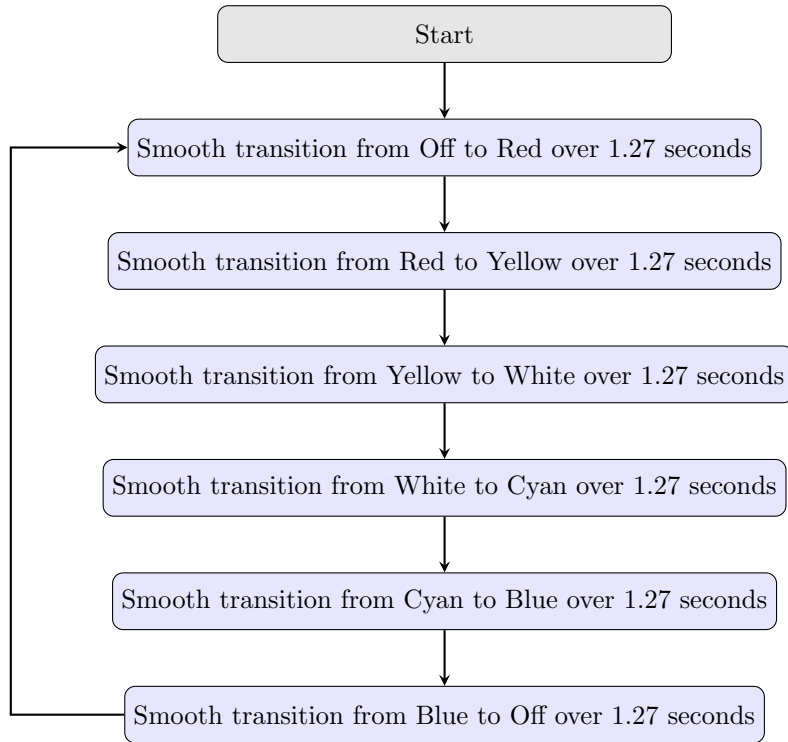


Figure 1: Color Transition Diagram
RGB LED Color Transitions

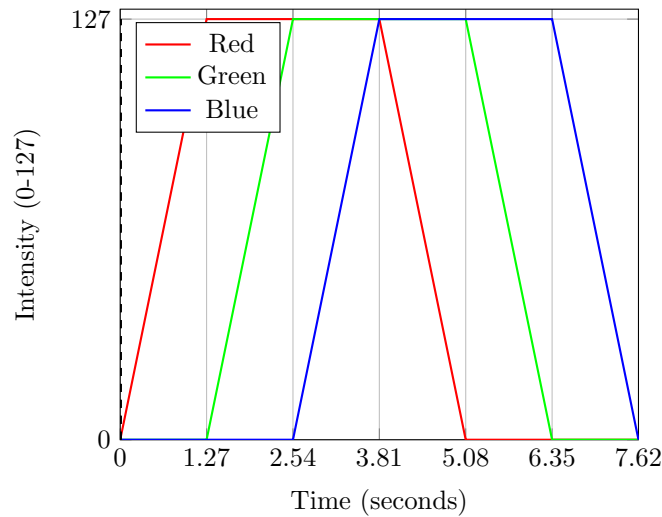


Figure 2: RGB LED Color Intensity

1.3 Comparison

The code behave as expected and follow the color pattern predicted.

Same for the blink code that make the green led blink at 1Hz.

1.4 Code modification, blink

to make the led blink, we looked into firmware/sysTime.h for longer duration then wrote a simple loop to make the LED blink at 1 Hz

```
1 #include <stdint.h>
2 #include "hardware.h"
3 #include "registers.h"
4
5 int main(void)
6 {
7     int8_t i;
8
9     hardware_init();
10    reg32_table[REG32_LED] = LED_MANUAL; // manual LED control
11
12    while (1) {
13        pause(HALF_SEC);
14        set_rgb(0, 255, 0); // turn on greenlight
15        pause(HALF_SEC);
16        set_rgb(0, 0, 0); // turn off greenlight
17    }
18    return 0;
19 }
```

2 Registers (5 points)

2.1 Task

We are tasked with reading and predicting the behavior of the code for the exercise 2. The goal is to understand how the callback system works.

2.2 Register Communication

We will talk briefly explain how the data are passed over the radio to the robot. However, we will abstract most detail of the radio communication as they are out of scope in the context of this question.

The computer use a serial link over radio to communicate with the robot. It is packet based and suffice to say that the packet have a defined format containing the kind of operation to do, the data to modify and the register.

The computer initialize the radio parameters (like data rate) and reboot the head before performing a series of operation and display the result.

On the robot side, the code implements a handler to process the register operation asynchronously whenever a data packet is received over the radio link and will blink a LED over and over from it's main loop.

The robot handler implements the following function :

8-bit Operations

- **Repeated Reads on Address 21:**

- Each read will return the fixed value 0x42 while incrementing the internal counter on the robot.

- **Read on Address 6:**

- This read returns the value of `counter` accumulated from previous reads on address 21 and then resets the counter.
- For instance, if you read address 21 three times, `counter` will be 3.
- Reading address 6 will then output 3 and reset `counter` to zero.

- **Write on Address 2, 3 or 4:**

- the data is written to the multibyte array at address n-2

16-bit and 32-bit Operations

- **Writing to Address 7:**

- Suppose the first 16-bit value written is W_1 . Since `datavar` is initially 0, the new value becomes:

$$\text{datavar} = 0 \times 3 + W_1 = W_1$$

- A subsequent write with a value W_2 will update `datavar` to:

$$\text{datavar} = W_1 \times 3 + W_2$$

- **Reading from Address 2 (32-bit):**

- Reading the 32-bit register at address 2 will return the current value of `datavar`, which reflects the cumulative updates from any previous 16-bit writes.

Multibyte Operations

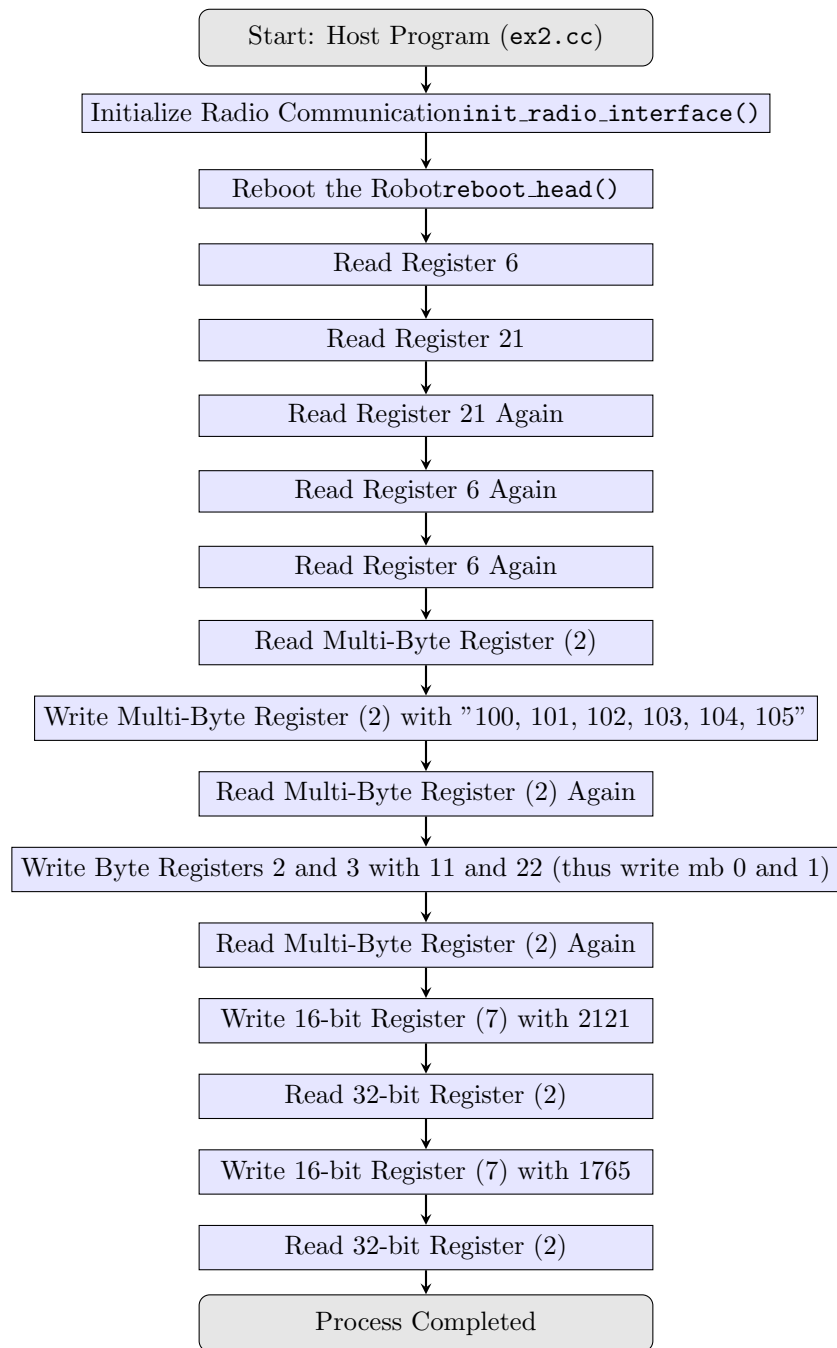
- **Writing a Multibyte Array:**

- For example, if the host writes the byte array $[1, 2, 3]$ to address 2 using a multibyte write, the robot's callback adds 4 to each byte.
- The stored array becomes $[5, 6, 7]$ and the size is set to 3.

- **Reading Back the Multibyte Array:**

- A subsequent multibyte read from address 2 will return the array $[5, 6, 7]$ along with the size (3).

The computer side code will perform the following register operation :



2.3 Expected Code behaviour

Operation	Expected Output
8-bit Operations:	
Read register 6 (initial)	<code>get_reg.b(6) = 0</code>
Read register 21 (first read)	<code>get_reg.b(21) = 66</code> <code>//=0x42</code>
Read register 21 (second read)	<code>get_reg.b(21) = 66</code>
Read register 6 (after reading 21 twice)	<code>get_reg.b(6) = 2</code>
Read register 6 (once more)	<code>get_reg.b(6) = 0</code>
Multibyte Operations:	
Read multi-byte register (2) initially	<code>get_reg.mb(2) = 0 bytes</code>
Write [100,101,...,107] to register 2, which becomes	<code>get_reg.mb(2) = 8 bytes: 104, 105, 106, 107, 108, 109, 110, 111</code>
Write 11 & 22 to register 2, which edit mb	<code>get_reg.mb(2) = 8 bytes: 11, 22, 106, 107, 108, 109, 110, 111</code>
16-bit and 32-bit Operations:	
Write 16-bit value 2121 to register 7	<code>get_reg.dw(2) = 2121</code>
Write 16-bit value 1765 to register 7 (cumulatively)	<code>get_reg.dw(2) = 8128</code>

Table 1: Register Operations and Expected Output

2.4 Comparison

the prediction was almost totally correct, with a minor difference due to a human error (can't count past 6) resulting in the predicting of value from 104 to 109 instead of 104 to 111

3 Communication with Other Elements (3 points)

3.1 Task

We are tasked with understanding the robot's CAN bus communication, analyzing the source code for Exercise 3, verifying its behavior, and implementing a program to read and display Degrees of Freedom (DOF) positions. We must be careful as the CAN bus access is interrupt-driven, meaning register functions cannot be used inside an interrupt.

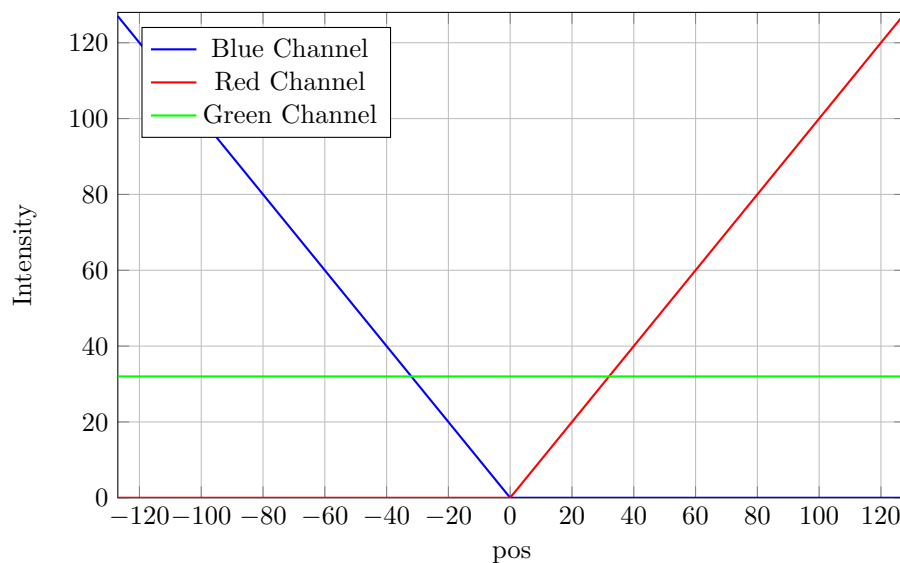
3.2 CAN Bus Communication

The robot's modules communicate via a CAN bus operating at 1 Mbps. A custom protocol is used to handle register read/write operations for 8-bit, 16-bit, and 32-bit registers, similar to the radio communication but independent of it. The key communication rules are:

- The head module initiates all CAN register operations.
- Modules can only respond to requests.
- Body elements have a unique address (found on a label).
- Limb elements have three addresses:
 - n (indicated on the label, corresponding to the body DOF)
 - $n + 1$ and $n + 2$ (corresponding to the limb DOFs)

3.3 Expected Code behavior

The code initialize the robot motor and assumes that the start position is zero. the color shown on the LED will vary from yellow to orange to green (at zero position) and continue to cyan and blue.



3.4 Comparison

The program did what was expected.

3.5 DOF Position Reading Implementation

To continuously monitor and display the Degree of Freedom (DOF) motor positions on the PC screen, a dedicated program was implemented. The communication relies on multi-byte register transactions, enabling the transmission of multiple motor positions in a single read. The architecture uses a **custom register handler** and a **local buffer** to store motor position values temporarily before transmission.

At startup, the program initializes the hardware, motors, and limbs using designated I2C addresses. A register handler function is registered to respond to 8-bit and multi-byte read/write requests. This handler allows a PC or another master device to request DOF data over a radio or bus interface.

- **Register Handler:**

- Responds to 8-bit read/write operations at a specific register address.
- Implements multi-byte read/write logic using local buffer `mb.buffer[]`.
- Enables external access to the latest DOF values by reading address 2 (multi-byte) or 6 (single-byte).

- **Local Buffering:**

- Motor positions from the bus are periodically polled and stored locally in `mb.buffer[]`.
- This decouples data acquisition from communication, ensuring consistent values are returned during a multi-byte read.

- **Main Loop Logic:**

- In each loop iteration, the program fetches current positions from four actuators and populates the buffer.
- The first motor's position is also stored in `motor_position` for single-byte access.
- LED feedback is provided by setting the RGB color intensity based on the magnitude of the first motor's position.

```
1 #include "hardware.h"
2 #include "registers.h"
3 #include "module.h"
4 #include "robot.h"
5 const uint8_t TAIL_MOTOR_ADDR = 21;
6 const uint8_t BODY_MOTOR_ADDR = 72;
7 const uint8_t BODY_MOTOR_FIN_LEFT = 73; // swap if not correct
8 const uint8_t BODY_MOTOR_FIN_RIGHT = 74;
9 static uint8_t motor_position = 0;
10 static uint8_t mb_buffer[4];
11 static uint8_t last_mb_size = 4;
12 /* Register callback function for 8-bit read/write operations */
13 static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
14 {
15     uint8_t i;
16     switch (operation) {
17     case ROP_READ_8:
18         if (address == 6) {
19             radio_data->byte = motor_position;
20             return TRUE;
21         }
22     case ROP_WRITE_8:
23         if (address == 6) {
24             motor_position = radio_data->byte; // Allow writing to register
25             return TRUE;
26         }
27     case ROP_READ_MB:
28         if (address == 2) {
29             radio_data->multibyte.size = last_mb_size;
30             for (i = 0; i < last_mb_size; i++) {
31                 radio_data->multibyte.data[i] = mb_buffer[i];
32             }
33             return TRUE;
34         }
35         break;
36     case ROP_WRITE_MB:
37         if (address == 2) {
38             last_mb_size = radio_data->multibyte.size;
39             for (i = 0; i < last_mb_size; i++) {
40                 mb_buffer[i] = radio_data->multibyte.data[i];
41             }
42             return TRUE;
43         }
44         break;
45     }
```

```

46     return FALSE;
47 }
48 int main(void)
49 {
50     hardware_init();
51     radio_add_reg_callback(register_handler); // Register the 8-bit handler
52     init_body_module(TAIL_MOTOR_ADDR);
53     init_body_module(BODY_MOTOR_ADDR);
54     init_limb_module(BODY_MOTOR_FIN_LEFT);
55     init_limb_module(BODY_MOTOR_FIN_RIGHT);
56     // Indicate boot sequence
57     set_color_i(4, 0);
58     pause(ONE_SEC);
59     set_color_i(2, 0);
60     while (1) {
61         motor_position = bus_get(TAIL_MOTOR_ADDR, MREG_POSITION); // Store position in register
62         mb_buffer[0] = bus_get(TAIL_MOTOR_ADDR, MREG_POSITION); // Store position in register
63         mb_buffer[1] = bus_get(BODY_MOTOR_ADDR, MREG_POSITION); // Store position in register
64         mb_buffer[2] = bus_get(BODY_MOTOR_FIN_LEFT, MREG_POSITION); // Store position in register
65         mb_buffer[3] = bus_get(BODY_MOTOR_FIN_RIGHT, MREG_POSITION); // Store position in register
66         set_rgb((motor_position < 0) ? -motor_position : motor_position, 32, 0);
67     }
68     return 0;
69 }

```

and on the computer side the code will be :

```

1  #include <iostream>
2  #include <thread>
3  #include <chrono>
4  #include "remregs.h"
5  #include "robot.h"
6  #include "utils.h"
7  using namespace std;
8  const uint8_t RADIO_CHANNEL = 201; ///< Robot radio channel
9  const char* INTERFACE = "COM1"; ///< Robot radio interface
10
11 // Displays the contents of a multibyte register as a list of bytes
12 void display_multibyte_register(CRemoteRegs& regs, const uint8_t addr)
13 {
14     uint8_t data_buffer[32], len;
15     if (regs.get_reg_mb(addr, data_buffer, len)) {
16         cout << (int) len << " bytes: ";
17         for (unsigned int i(0); i < len; i++) {
18             if (i > 0) cout << ", ";
19             cout << static_cast<int32_t>(static_cast<int8_t>(data_buffer[i]));
20         }
21         cout << endl;
22     } else {
23         cerr << "Unable to read multibyte register." << endl;
24     }
25 }
26
27 int main()
28 {
29     CRemoteRegs regs;
30
31     if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
32         return 1;
33     }
34
35     // Reboots the head microcontroller to ensure a consistent state
36     reboot_head(regs);
37
38     while (!kbhit()) {
39         display_multibyte_register(regs, 2);
40     }
41
42     regs.close();
43     return 0;
44 }

```

3.6 Testing

We misread the exercise and implemented the read for only one motor instead of all of them. we redid the code and it work as expected.

4 Position Control of a Module (3 points)

4.1 Task

We are tasked to control a motorized module by writing to its registers, ensuring proper startup conditions, and modifying the program to send a sine wave setpoint over the radio. We must pay attention to manually resetting elements to their zero position before starting as the modules lack absolute position sensors, they must be manually placed at zero before activation. Key precautions include:

- For body elements, zero position is the center.
- For limb elements, zero position depends on the trajectory generator. (we set it horizontally)
- PD controllers should only be activated when necessary and stopped when not in use.

4.2 Verification and Basic Control

The program is compiled and run to initiate motor movement. To activate the movement, a value of 1 must be written to REG8_MODE using a custom program based on `ex2.cc`. Before execution, the module's position is manually set to zero. To stop the movement, a value of 0 is written to the same register or the robot is turned off.

4.3 Sine Wave Setpoint Implementation

The program is modified to allow a sine wave setpoint at 1 Hz with an amplitude of $\pm 40^\circ$ is implemented and transmitted over the radio.

- Use `time_d()` from `utils.h` to obtain precise timestamps.
- Compute the sine wave using `sin()` from `<math.h>`.
- Use `M_PI` for π to ensure accurate calculations.

We implemented the following in `modes.c` to dynamically update the motor frequency and amplitude while staying backward compatible with the implementation proposed for the first part.

```
1 #include "modes.h"
2 #include "config.h"
3 #include "hardware.h"
4 #include "module.h"
5 #include "regdefs.h"
6 #include "registers.h"
7 #include "robot.h"
8 #include "sysTime.h"
9
10 const uint8_t MOTOR_ADDR = 21;
11 volatile int8_t motor_position = 0;
12
13 static int8_t register_handler(uint8_t operation, uint8_t address,
14                               RadioData *radio_data) {
15
16     if (address == REG8_MODE) {
17         switch (operation) {
18             case ROP_READ_8:
19                 radio_data->byte = reg8_table[REG8_MODE];
20                 return TRUE;
21             case ROP_WRITE_8:
22                 reg8_table[REG8_MODE] = radio_data->byte; // Allow writing to register
23                 return TRUE;
24         }
25     }
26     if (address == 0x06) {
27         switch (operation) {
28             case ROP_READ_8:
29                 radio_data->byte = motor_position;
30                 return TRUE;
31             case ROP_WRITE_8:
32                 motor_position = radio_data->byte; // Allow writing to register
33                 return TRUE;
34         }
35     }
36     return FALSE;
37 }
38
39 void motor_demo_mode() {
40     init_body_module(MOTOR_ADDR);
41     start_pid(MOTOR_ADDR);
42 }
```

```

42 set_color(4);
43 while (reg8_table[REG8_MODE] == IMODE_MOTOR_DEMO) {
44     bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(21.0));
45     pause(ONE_SEC);
46     bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-21.0));
47     pause(ONE_SEC);
48 }
49 bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
50 pause(ONE_SEC);
51 bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
52 set_color(2);
53 }
54
55 void motor_sine_demo() {
56     init_body_module(MOTOR_ADDR);
57     start_pid(MOTOR_ADDR);
58     set_color(3);
59     while (reg8_table[REG8_MODE] == IMODE_SINE_DEMO) {
60         bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(motor_position));
61         pause(TEN_MS);
62     }
63     bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
64     pause(ONE_SEC);
65     bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
66     set_color(2);
67 }
68
69 void main_mode_loop() {
70     reg8_table[REG8_MODE] = IMODE_IDLE;
71     radio_add_reg_callback(register_handler);
72     while (1) {
73         switch (reg8_table[REG8_MODE]) {
74             case IMODE_IDLE:
75                 break;
76             case IMODE_MOTOR_DEMO:
77                 motor_demo_mode();
78                 break;
79             case IMODE_SINE_DEMO:
80                 motor_sine_demo();
81                 break;
82             default:
83                 reg8_table[REG8_MODE] = IMODE_IDLE;
84         }
85     }
86 }

```

On the computer we used the following code that offer a minimal interactive prompt for part 1 and 2

```

1  #include "regdefs.h"
2  #include "remregs.h"
3  #include "robot.h"
4  #include "utils.h"
5  #include <cmath>
6  // #include <conio.h> // For kbhit() and getch()
7  #include <iostream>
8  #define IMODE_IDLE 0
9  #define IMODE_MOTOR_DEMO 1
10 #define IMODE_SINE_DEMO 2
11 // Constants
12 const uint8_t RADIO_CHANNEL = 201; // Radio channel
13 const char *INTERFACE = "COM1"; // Serial port for radio interface
14 const uint8_t MOTOR_ADDR = 21; // Motor address (from modes.c)
15 const double AMPLITUDE_DEG = 40.0; // Amplitude in degrees
16 const double FREQUENCY_HZ = 1.0; // Frequency in Hz
17 using namespace std;
18
19 int main() {
20     CRemoteRegs regs;
21
22     cout << "Initializing robot connection..." << endl;
23     if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
24         return 1;
25     }
26
27     // Reboot the head microcontroller to ensure it is in a known state.
28     reboot_head(regs);
29
30     // Show initial mode
31     uint8_t mode = regs.get_reg_b(REG8_MODE);
32     cout << "Initial mode: " << static_cast<int>(mode) << endl;

```

```

33
34 bool exitProgram = false;
35 char choice = '\0';
36
37 while (!exitProgram) {
38     cout << "\n===== \n";
39     cout << "Select an option:\n";
40     cout << "1. Motor Demo Mode\n";
41     cout << "2. Sine Wave Control Mode\n";
42     cout << "q. Quit\n";
43     cout << "Enter your choice: ";
44
45     cin >> choice;
46     // Clear any extra characters from the input buffer.
47     cin.ignore(10000, '\n');
48
49     switch (choice) {
50     case '1': {
51         cout << "\nStarting Motor Demo Mode..." << endl;
52         regs.set_reg_b(REG8_MODE, IMODE_MOTOR_DEMO);
53         cout << "Press Enter to stop motor demo." << endl;
54         cin.get();
55         regs.set_reg_b(REG8_MODE, IMODE_IDLE);
56         cout << "Motor demo stopped. Waiting for motor to return to zero..."
57             << endl;
58         Sleep(2000); // Give time for the motor to settle
59         break;
60     }
61     case '2': {
62         cout << "\nStarting Sine Wave Control Mode..." << endl;
63         regs.set_reg_b(REG8_MODE, IMODE_SINE_DEMO);
64         cout << "Press any key to stop." << endl;
65
66         double startTime = time_d(); // Get the start time
67         double currentTime = 0.0;
68         bool running = true;
69
70         while (running) {
71             // Calculate elapsed time since start
72             currentTime = time_d() - startTime;
73
74             // Calculate sine wave value (-1 to 1) and scale it to the desired
75             // amplitude
76             double angle =
77                 AMPLITUDE_DEG * sin(2.0 * M_PI * FREQUENCY_HZ * currentTime);
78
79             // Send the setpoint to the motor (assuming register 0x06 is the
80             // setpoint)
81             regs.set_reg_b(0x06, static_cast<int8_t>(angle));
82
83             // If a key is pressed, break out of the loop
84             if (kbhit()) {
85                 // Clear the key from the buffer.
86                 // getch();
87                 running = false;
88             }
89
90             // Small delay to prevent overwhelming communication
91             Sleep(10);
92         }
93         // Stop the motor and return to idle mode
94         regs.set_reg_b(0x06, 0); // Set setpoint to 0
95         regs.set_reg_b(REG8_MODE, IMODE_IDLE);
96         cout << "Sine wave control stopped." << endl;
97         break;
98     }
99     case 'q':
100     case 'Q': {
101         exitProgram = true;
102         cout << "\nExiting program." << endl;
103         break;
104     }
105     default: {
106         cout << "\nInvalid choice. Please try again." << endl;
107         break;
108     }
109 }
110 }
111 return 0;
112 }
113 }

```

4.4 Testing and Challenges

The program was tested and worked but due to lost packet over the radio link the movement was a bit jerky and not very smooth at time. To remedy this we will prefer the implementation from the next chapter.

5 Trajectory Generation and Control (3+3 points)

5.1 Task

We are tasked make the robot move using a sine-wave trajectory onboard the microcontroller, first using a predefined sine-wave generator and then extending it to control a motor. The program is further modified to allow real-time adjustment of sine wave parameters (frequency and amplitude) from the computer.

5.2 Sine-Wave Generation on the Microcontroller

We first uploaded the program to see that the led does blink following a sinusoidal pattern with frequency of 1 second. to do so we made a simple program on the computer side to set the state (IDLE/sine-Wave) while the code on the robot side

5.3 Testing and Motor Control

Then, modify it to send the sine wave to a motor (as before, with an amplitude of $\pm 40^\circ$). The motor movement is very similar for some frequency but smoother and as the frequency and amplited are increase this approach really shine compared the previous strategy.

5.4 Modulating Trajectory Parameters

The trajectory generation program is extended to allow dynamic control of frequency and amplitude from the computer. Constraints:

- Frequency is limited to a maximum of 2 Hz.
- Amplitude is limited to $\pm 60^\circ$.

To encode floating-point values in an 8-bit register, the ENCODE_PARAM_8 and DECODE_PARAM_8 macros from `registers.h` are used. On the PC side, `regdefs.h` is included to support these macros.

5.5 Implementation

On the computer side, a simple interactive shell program was implemented to allow changing the robot testing mode and edit the parameters (Amplitude & Frequency)

```
1
2 #include "regdefs.h"
3 #include "remregs.h"
4 #include "robot.h"
5 #include "utils.h"
6 #include <cmath>
7 #include <iostream>
8
9 #define IMODE_IDLE 0
10 #define IMODE_MOTOR_DEMO 1
11 #define IMODE_SINE_DEMO 2
12
13 // Constants
14 const uint8_t RADIO_CHANNEL = 201; // Radio channel
15 const char *INTERFACE = "COM1"; // Serial port for radio interface
16
17 // Define registers for frequency and amplitude control
18 #define REG8_SINE_FREQ 10 // Register for sine wave frequency
19 #define REG8_SINE_AMP 11 // Register for sine wave amplitude
20
21 // Define limits for frequency and amplitude
22 #define MAX_FREQ 2.0f // Maximum frequency in Hz
23 #define MAX_AMP 60.0f // Maximum amplitude in degrees
24
25 using namespace std;
26
27 // Function to display current settings
28 void display_settings(CRemoteRegs &regs) {
29     uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
30     uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);
31 }
```

```

32 float freq = DECODE_PARAM_8(freq_reg, 0.1f, MAX_FREQ);
33 float amplitude = DECODE_PARAM_8(amp_reg, 1.0f, MAX_AMP);
34
35 cout << "Current settings:" << endl;
36 cout << "Frequency:" << freq << "Hz(encoded:" << (int)freq_reg << ")"
37 << endl;
38 cout << "Amplitude:" << amplitude << "degrees(encoded:" << (int)amp_reg
39 << ")" << endl;
40 }
41
42 // Function to update a parameter
43 void update_parameter(CRemoteRegs &regs, const char *name, uint8_t reg,
44 float min_value, float max_value, float scale,
45 float current) {
46 float new_value;
47 cout << "Enter new" << name << "(" << min_value << "-" << max_value
48 << ")[" << current << "]:";
49
50 string input;
51 getline(cin, input);
52
53 if (input.empty()) {
54 cout << "Keeping current value." << endl;
55 return;
56 }
57
58 try {
59 new_value = stof(input);
60
61 // Validate the input
62 if (new_value < min_value || new_value > max_value) {
63 cout << "Value out of range. Using closest valid value." << endl;
64 new_value = (new_value < min_value) ? min_value : max_value;
65 }
66
67 // Encode and set the register
68 uint8_t encoded = ENCODE_PARAM_8(new_value, scale, max_value);
69 regs.set_reg_b(reg, encoded);
70
71 cout << name << " set to" << new_value << "(" << (int)encoded
72 << ")" << endl;
73 } catch (const exception &e) {
74 cout << "Invalid input. Keeping current value." << endl;
75 }
76 }
77
78 int main() {
79 CRemoteRegs regs;
80
81 cout << "Sine Wave Controller for Fish Robot" << endl;
82 cout << "-----" << endl;
83
84 cout << "Initializing robot connection on" << INTERFACE << ", channel"
85 << (int)RADIO_CHANNEL << endl;
86 if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
87 cerr << "Failed to initialize radio interface" << endl;
88 return 1;
89 }
90
91 // Reboot the head microcontroller
92 cout << "Rebooting head..." << endl;
93 reboot_head(regs);
94
95 // Main control loop
96 bool running = true;
97 while (running) {
98 // Get current parameter values
99 uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
100 uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);
101
102 float freq = DECODE_PARAM_8(freq_reg, 0.1f, MAX_FREQ);
103 float amplitude = DECODE_PARAM_8(amp_reg, 1.0f, MAX_AMP);
104
105 // Check current mode
106 uint8_t currentMode = regs.get_reg_b(REG8_MODE);
107
108 // Display menu
109 cout << "\n-----MENU-----" << endl;
110 cout << "1. Start/Stop Sine Wave Demo" << endl;
111 cout << "2. Set Frequency(current:" << freq << "Hz)" << endl;
112 cout << "3. Set Amplitude(current:" << amplitude << "degrees)" << endl;

```

```

113 cout << "4. Display Current Settings" << endl;
114 cout << "5. Exit" << endl;
115 cout << "Current mode: "
116 << (currentMode == IMODE_SINE_DEMO ? "RUNNING" : "STOPPED") << endl;
117 cout << "Selection: ";
118
119 string input;
120 getline(cin, input);
121
122 if (input.empty())
123     continue;
124
125 switch (input[0]) {
126 case '1':
127     if (currentMode != IMODE_SINE_DEMO) {
128         cout << "Starting sine wave demo..." << endl;
129         regs.set_reg_b(REG8_MODE, IMODE_SINE_DEMO);
130     } else {
131         cout << "Stopping sine wave demo..." << endl;
132         regs.set_reg_b(REG8_MODE, IMODE_IDLE);
133     }
134     break;
135
136 case '2':
137     update_parameter(regs, "frequency", REG8_SINE_FREQ, 0.1f, MAX_FREQ, 0.1f,
138                     freq);
139     break;
140
141 case '3':
142     update_parameter(regs, "amplitude", REG8_SINE_AMP, 1.0f, MAX_AMP, 1.0f,
143                     amplitude);
144     break;
145
146 case '4':
147     display_settings(regs);
148     break;
149
150 case '5':
151     // Make sure to stop the demo before exiting
152     if (regs.get_reg_b(REG8_MODE) == IMODE_SINE_DEMO) {
153         cout << "Stopping sine wave demo..." << endl;
154         regs.set_reg_b(REG8_MODE, IMODE_IDLE);
155         // Give it time to stop properly
156         Sleep(2000);
157     }
158     running = false;
159     break;
160
161 default:
162     cout << "Invalid selection. Please try again." << endl;
163     break;
164 }
165 }
166
167 cout << "Program terminated." << endl;
168 return 0;
169 }

```

On the robot side, a trajectory generation program was implemented to produce a sine-wave-based setpoint for a motor, with support for real-time modulation of frequency and amplitude. The program runs in a loop, continuously computing the desired angle using a sine function and sending the result to the motor controller.

A custom register handler is defined to handle 8-bit reads and writes for mode selection, frequency, and amplitude. When the mode is set to `IMODE_SINE_DEMO`, the robot enters a control loop where it:

- Decodes the current sine parameters from the 8-bit registers,
- Computes the sine of the current time multiplied by frequency and scaled by amplitude,
- Converts the angle into a format suitable for the motor (using `DEG_TO_OUTPUT_BODY`),
- Sends the computed setpoint to the motor over the bus,
- Updates the LED color dynamically to reflect the current frequency (red) and amplitude (green) values,
- Returns to idle and centers the motor once the mode is changed.


```

1 #include "config.h"
2 #include "hardware.h"
3 #include "modes.h"
4 #include "module.h"
5 #include "registers.h"
6 #include "robot.h"
7
8 // Define registers for frequency and amplitude control
9 #define REG8_SINE_FREQ 10 // Register for sine wave frequency
10 #define REG8_SINE_AMP 11 // Register for sine wave amplitude
11
12 // Define limits for frequency and amplitude
13 #define MAX_FREQ 2.0f // Maximum frequency in Hz
14 #define MAX_AMP 60.0f // Maximum amplitude in degrees
15
16 // Default values
17 #define DEFAULT_FREQ 1.0f // Default frequency in Hz
18 #define DEFAULT_AMP 40.0f // Default amplitude in degrees
19
20 const uint8_t MOTOR_ADDR = 21; // Motor address
21
22 uint8_t freq_enc = DEFAULT_FREQ;
23 uint8_t amp_enc = DEFAULT_AMP;
24
25
26 static int8_t register_handler(uint8_t operation, uint8_t address,
27                               RadioData *radio_data) {
28
29     switch (address) {
30         case REG8_MODE:
31             switch (operation) {
32                 case ROP_READ_8:
33                     radio_data->byte = reg8_table[REG8_MODE];
34                     return TRUE;
35                 case ROP_WRITE_8:
36                     reg8_table[REG8_MODE] = radio_data->byte; // Allow writing to register
37                     return TRUE;
38             }
39         case REG8_SINE_FREQ:
40             switch (operation) {
41                 case ROP_READ_8:
42                     radio_data->byte = freq_enc;
43                     return TRUE;
44                 case ROP_WRITE_8:
45                     freq_enc = radio_data->byte; // Allow writing to register
46                     return TRUE;
47             }
48         case REG8_SINE_AMP:
49             switch (operation) {
50                 case ROP_READ_8:
51                     radio_data->byte = amp_enc;
52                     return TRUE;
53                 case ROP_WRITE_8:
54                     amp_enc = radio_data->byte; // Allow writing to register
55                     return TRUE;
56             }
57     }
58     return FALSE;
59 }
60
61
62 // Function to initialize default parameters
63 void init_sine_params(void) {
64     // Set default values for frequency and amplitude
65     reg8_table[REG8_SINE_FREQ] = ENCODE_PARAM_8(DEFAULT_FREQ, 0.1f, MAX_FREQ);
66     reg8_table[REG8_SINE_AMP] = ENCODE_PARAM_8(DEFAULT_AMP, 1.0f, MAX_AMP);
67 }
68
69 void sine_demo_mode(void) {
70     uint32_t dt, cycletimer;
71     float my_time, delta_t, angle;
72     float freq, amplitude;
73     int8_t angle_rounded;
74
75     // Initialize and start the motor's PID controller
76     init_body_module(MOTOR_ADDR);
77     start_pid(MOTOR_ADDR);
78
79     // Set visual indicator that motor is active
80     set_color(4); // Set LED to red
81

```

```

82 // Initialize sine wave time
83 cycletimer = getSysTICs();
84 my_time = 0;
85
86 // Make sure parameters are initialized
87 if (freq_enc == 0)
88     freq_enc = ENCODE_PARAM_8(DEFAULT_FREQ, 0.1f, MAX_FREQ);
89 if (amp_enc == 0)
90     amp_enc = ENCODE_PARAM_8(DEFAULT_AMP, 1.0f, MAX_AMP);
91
92 do {
93     // Decode current parameters from registers
94     freq = DECODE_PARAM_8(freq_enc, 0.1f, MAX_FREQ);
95     amplitude = DECODE_PARAM_8(amp_enc, 1.0f, MAX_AMP);
96
97     // Apply limits to ensure safety
98     if (freq > MAX_FREQ)
99         freq = MAX_FREQ;
100     if (amplitude > MAX_AMP)
101         amplitude = MAX_AMP;
102
103     // Calculate elapsed time
104     dt = getElapsedSysTICs(cycletimer);
105     cycletimer = getSysTICs();
106     delta_t = (float)dt / sysTICsperSEC;
107     my_time += delta_t;
108
109     // Calculate the sine wave for motor angle
110     angle = amplitude * sin(M_TWOPI * freq * my_time);
111
112     // Convert angle to motor units
113     angle_rounded = DEG_TO_OUTPUT_BODY(angle);
114
115     // Send the angle to the motor
116     bus_set(MOTOR_ADDR, MREG_SETPPOINT, angle_rounded);
117
118     // Update LED for visual feedback - color indicates frequency,
119     // brightness indicates amplitude
120     uint8_t red = (uint8_t)(freq * 127.0f / MAX_FREQ);
121     uint8_t green = (uint8_t)(amplitude * 127.0f / MAX_AMP);
122
123     if (angle >= 0) {
124         // Positive angle - more green
125         set_rgb(red, green + 20, 20);
126     } else {
127         // Negative angle - more red
128         set_rgb(red + 20, green, 20);
129     }
130
131     // Small delay to ensure timer updates properly
132     pause(ONE_MS);
133
134 } while (reg8_table[REG8_MODE] == IMODE_SINE_DEMO);
135
136 // Clean up: return motor to zero position
137 bus_set(MOTOR_ADDR, MREG_SETPPOINT, 0);
138 pause(ONE_SEC); // Give the motor time to return to center
139
140 // Stop the motor
141 bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
142
143 // Return LED to normal state
144 set_color(2);
145 }
146
147 void main_mode_loop(void) {
148     // Initialize the default parameters
149     init_sine_params();
150
151     // Set initial mode
152     reg8_table[REG8_MODE] = IMODE_IDLE;
153
154     // Add the register handler
155     radio_add_reg_callback(register_handler);
156
157     while (1) {
158         switch (reg8_table[REG8_MODE]) {
159             case IMODE_IDLE:
160                 break;
161             case IMODE_SINE_DEMO:

```

```

163     sine_demo_mode();
164     break;
165     default:
166         reg8_table[REG8_MODE] = IMODE_IDLE;
167     }
168 }
169 }

```

5.6 Testing and Observations

The three mode were test and work as expected with live editing of the parameters for the Amplitude and Frequency.

6 Tracking System

6.1 Task

We are tasked to use the aquarium's LED tracking system to obtain real-time (x, y) coordinates the robot. The program in `pc/ex6/ex6.cc` establishes the connection and displays the position of a detected LED.

6.2 Setup and introduction

Before running the program, the tracking system must be confirmed as operational with the assistance of an instructor. Environmental conditions must be adjusted to minimize false detections:

- Turn off ceiling fluorescent lights.
- Partially close window blinds to reduce external light interference.

We placed a sagex bloc with an LED in the aquarium and saw that indeed the tracking system was locating the LED position. We had to recalibrate it once and learned that the positioning error is really low (1-2mm).

6.3 Implementation of LED color based on position

We modified the provided code to make the RGB LED color change depending on the position in the aquarium.

The robot code does nothing as we can natively changed the head color by writing to the correct register over radio.

```

1  #include "can.h"
2  #include "hardware.h"
3  #include "module.h"
4  #include "registers.h"
5  #include "robot.h"
6  #include <stdint.h>
7
8  // Address of the motor module
9
10 int main(void) {
11     hardware_init();
12     registers_init();
13     // Set head LED to visible for tracking (green=64)
14     // We'll let the PC control the exact color
15
16     // Main loop - just keep the system running
17     while (1) {
18         // The head will receive color commands from the PC program
19         // No additional processing needed here
20         pause(TEN_MS);
21     }
22
23     return 0;
24 }

```

while on the computer the following code is implemented to locate the position of the foam block with LED in the pool, print its position on the computer and edit the Robot LED based on that position.

```

1 #include "remregs.h"
2 #include "robot.h"
3 #include "trkcli.h"
4 #include "utils.h"
5 #include "regdefs.h"
6 #include <cstdlib>
7 #include <iostream>
8 #include <stdint.h>
9 #include <windows.h>
10
11 using namespace std;
12
13 const char *TRACKING_PC_NAME = "biorobpc6"; ///< host name of the tracking PC
14 const uint16_t TRACKING_PORT = 10502;      ///< port number of the tracking PC
15 const uint8_t RADIO_CHANNEL = 201;         ///< robot radio channel
16 const char *INTERFACE = "COM1";           ///< robot radio interface
17
18 // Aquarium dimensions in meters
19 const double AQUARIUM_WIDTH = 6.0;
20 const double AQUARIUM_HEIGHT = 2.0;
21
22 // Function to map a value from one range to another
23 double map_value(double value, double in_min, double in_max, double out_min,
24                 double out_max) {
25     return (value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
26 }
27
28 int main() {
29     CTrackingClient trk;
30     CRemoteRegs regs;
31
32     cout << "Initializing robot connection..." << endl;
33     if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
34         cerr << "Failed to initialize radio interface" << endl;
35         return 1;
36     }
37
38     // Reboot the head microcontroller to ensure it's in a known state
39     reboot_head(regs);
40
41     cout << "Connecting to tracking system..." << endl;
42     if (!trk.connect(TRACKING_PC_NAME, TRACKING_PORT)) {
43         cerr << "Failed to connect to tracking system" << endl;
44         return 1;
45     }
46
47     cout << "Connected to tracking system. Turn off other module LEDs and place"
48         << "the robot in the aquarium."
49         << endl;
50     cout << "Press any key to exit." << endl;
51
52     // Turn off the LED of another module (assuming module address 21)
53     const uint8_t MOTOR_ADDR = 21;
54     // This code doesn't work directly from PC, needs to be on the robot side
55     // Will be handled in the robot part of the solution
56
57     // Fixed green component for tracking
58     const uint8_t GREEN_COMPONENT = 64;
59
60     while (!kbhit()) {
61         uint32_t frame_time;
62         // Gets the current position
63         if (!trk.update(frame_time)) {
64             cerr << "Error updating tracking data" << endl;
65             return 1;
66         }
67
68         double x, y;
69
70         // Gets the ID of the first spot
71         int id = trk.get_first_id();
72
73         // Reads its coordinates (if (id == -1), then no spot is detected)
74         if (id != -1 && trk.get_pos(id, x, y)) {
75             // Calculate LED color based on position
76             // Red increases with x (left to right)
77             uint8_t r = (uint8_t)map_value(x, 0, AQUARIUM_WIDTH, 0, 255);
78

```

```

79 // Blue increases with y (bottom to top)
80 uint8_t b = (uint8_t)map_value(y, 0, AQUARIUM_HEIGHT, 0, 255);
81
82 // Keep green fixed for tracking
83 uint8_t g = GREEN_COMPONENT;
84
85 // Set the LED color
86 uint32_t rgb = ((uint32_t)r << 16) | ((uint32_t)g << 8) | b;
87 regs.set_reg_dw(REG32_LED, rgb);
88
89 cout << "Position:_" << fixed << x << ",_" << y << ")_m_m_|_Color:_RGB("
90 << (int)r << ",_" << (int)g << ",_" << (int)b << ")_r";
91 cout.flush();
92 } else {
93 cout << "Position:_(not_detected)_r";
94 cout.flush();
95 }
96
97 // Wait 10 ms before getting the info next time
98 Sleep(10);
99 }
100
101 // Clears the console input buffer (as kbhit() doesn't)
102
103 FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
104 cout << endl << "Program_terminated." << endl;
105 return 0;
106 }

```

6.4 Observations and result

We can see that as we move the floating LED over the pool, the robot's LED color changes correctly.

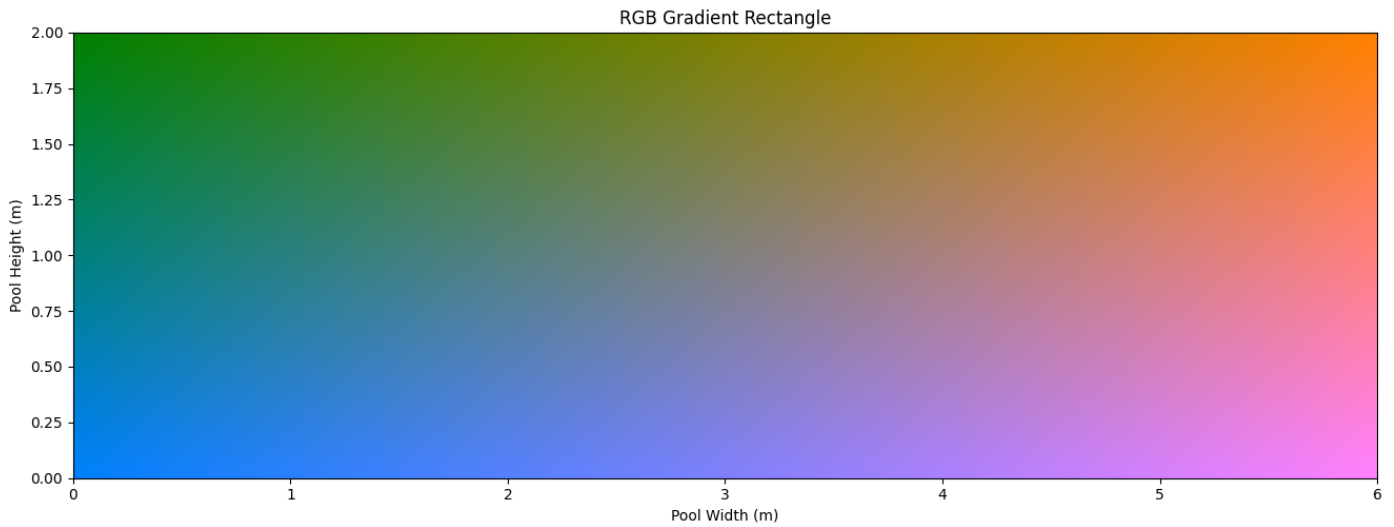


Figure 3: Color gradient

7 Swimming and Experiments (10 points)

7.1 Task

We are tasked with implementing a trajectory generator that allows swimming locomotion.

7.2 Writing a Swimming Trajectory Generator

To make the Lamprey Robot swim, we implemented a traveling wave locomotion pattern inspired by the movement of anguilliform swimmers. The core principle is to generate a sinusoidal wave that propagates from head to tail, with the amplitude gradually increasing along the body. This movement emulates the way real fish swim and is controlled through a centralized program running on the robot's microcontroller.

The oscillatory motion is defined by the following equation:

$$\theta_i(t) = A \cdot \sin \left(2\pi \cdot \nu t + \frac{i \cdot \phi}{N} + \delta \right) \quad (1)$$

where:

- A is the half-amplitude of the oscillation (in degrees),
- ν is the frequency of oscillation (in Hz),
- t is the time (in seconds),
- i is the module index (starting at $i = 0$ for the tail),
- ϕ is the total phase lag between the tail and the head,
- N is the total number of actuated modules.
- δ is the offset to enable steering

To allow dynamic tuning of the swimming parameters from a computer interface (e.g., via radio communication), we implemented four dedicated 8-bit registers:

- `REG8_SINE_FREQ` for frequency,
- `REG8_SINE_AMP` for amplitude,
- `REG8_SINE_LAG` for phase lag between adjacent segments,
- `REG8_SINE_OFF` for overall phase offset.

These registers are read and written via a central `register_handler` function triggered from the computer over radio. The values are encoded as 8-bit integers and decoded into float values using linear mappings bounded by safety limits.

The function `swim_mode()` is responsible for generating the sinusoidal wave over time and applying it to each motor. At every iteration of the main control loop:

1. The elapsed time τ is updated.
2. The frequency ν , amplitude A , lag ϕ , and offset are read from the registers.
3. A sinusoidal angle is computed for each module i according to the formula above.
4. Each angle is converted to motor units and sent to the corresponding actuator via the bus.

This produces a smooth traveling wave that animates the body of the robot. The loop continues as long as the mode register is set to `IMODE_SWIM`.

For the full code running on the robot and the computer, see the Appendix.

7.3 Swimming speed measurement

Once an effective swimming motion is achieved, a computer controller should be written to use the tracking system and measure the robot's average speed. The controller should:

- Start the swimming motion by writing a value in `REG8_MODE`,
- Stop the motion after a set time or distance.

For each parameter (frequency or phase lag), 3 to 5 measurements should be taken. The average speed and standard deviation must be reported. The LED's (x, y) coordinates should be recorded, and selected trajectories plotted.

we were careful to follow the following :

- For frequency, do not exceed 1.5 Hz to prevent motor stress.
- The total phase lag should be varied between $\phi = 0.5$ and $\phi = 1.5$.

7.4 Result

We recorded the position of the robot in the pool at each instant over multiple runs.

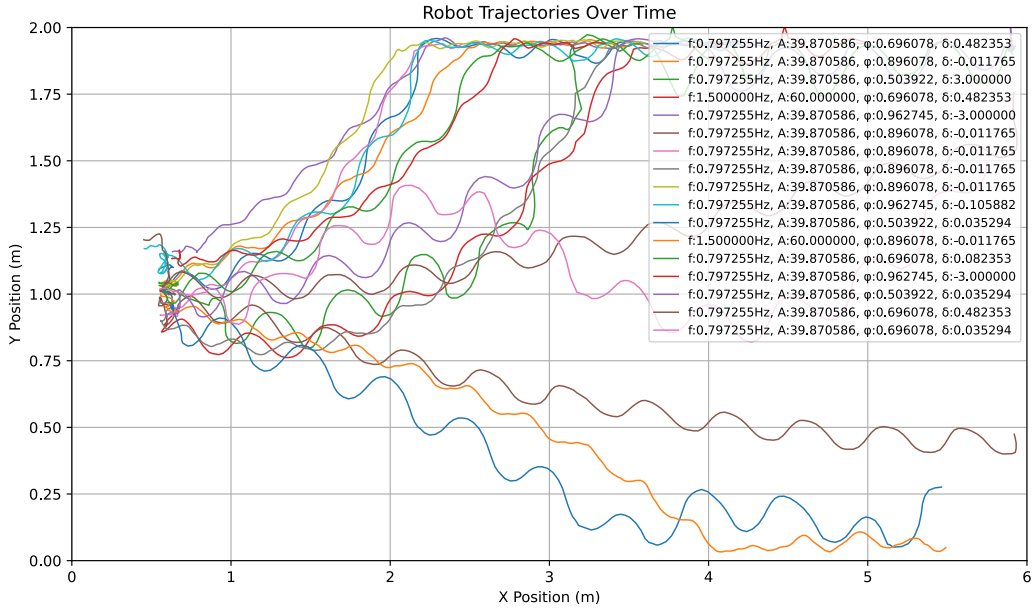


Figure 4: Pool trajectory of each run

We then plotted the relationship between the speed and the lag and get the following figure.

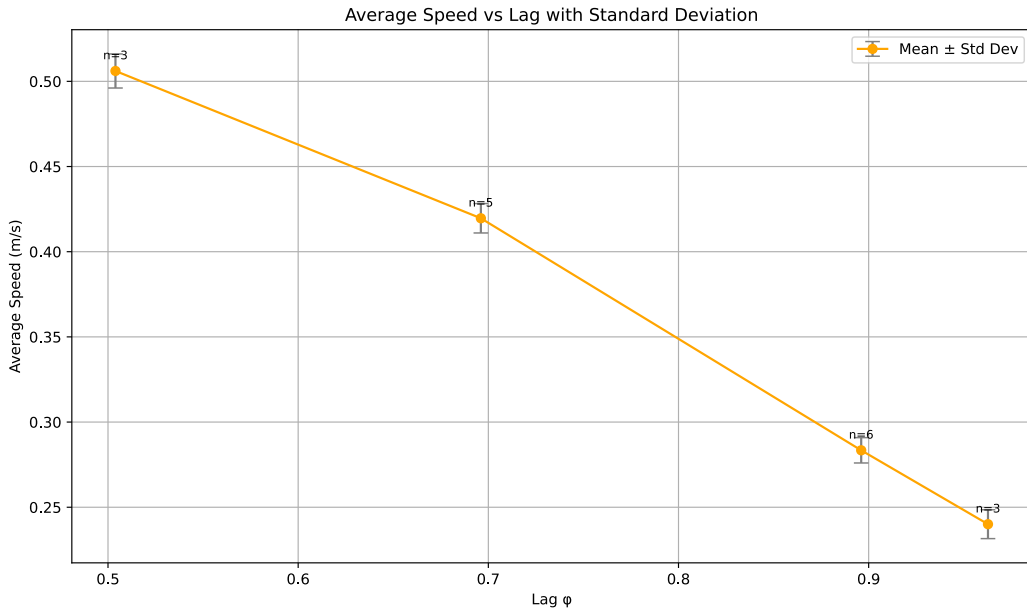


Figure 5: Plot of the robot speed over lag (radian)

We can see that the phase lag has a big effect on the speed of the robot (all other parameters staying equal). The speed decreases proportionally with the phase lag. The speed was the fastest with a phase lag of 0.5. We saw that the robot was less stable with a small phase lag than a bigger one. We think that the optimal value would be to choose a phase lag between 0.7 and 0.9 and then increase the frequency if we want to increase the speed.

8 Speed and steering control

8.1 Task

We implemented a speed and steering control of the robot from the computer keyboard. We did so by changing the offset in equation (1) to force the robot to swim more on the right or left when pressing the keys A/D. The code is the same as for Task 7. We also increase the frequency or decrease it by pressing the keys W/S.

8.2 Result

The robot was able to rotate in circles, but the dynamic steering was jerky and not very effective. Maybe a smooth transition of the offset angle would have fixed this issue. The speed control was working as expected.

9 appendix

This is the code that run on the computer for the part 7 and 8.

```
1 #include "regdefs.h"
2 #include "remregs.h"
3 #include "robot.h"
4 #include "trkcli.h"
5 #include "utils.h"
6 #include <chrono>
7 #include <cmath>
8 #include <cstdlib>
9 #include <ctime>
10 #include <fstream>
11 #include <iomanip>
12 #include <iostream>
13 #include <stdint.h>
14 #include <string>
15 #include <windows.h>
16
17 using namespace std;
18
19 /// Idle mode: do nothing
20 #define IMODE_IDLE 0
21
22 /// set the robot in a warped and rigid position (prevent capsizing)
23 #define IMODE_READY 1
24
25 /// active swimming mode
26 #define IMODE_SWIM 2
27
28 /// Define limits for frequency and amplitude
29 #define MAX_FREQ 1.5f // Maximum frequency in Hz
30 #define MAX_AMP 60.0f // Maximum amplitude in degrees
31 #define MAX_LAG 1.5f // Maximum lag between elements in degrees
32 #define MAX_OFF 3.0f // Maximum offset in degrees
33
34 #define MIN_FREQ 0.1f
35 #define MIN_AMP 1.0f // Min amplitude in degrees
36 #define MIN_LAG 0.5f // Min lag between elements in degrees
37 #define MIN_OFF -3.0f // Min offset in degrees
38
39 /// Define registers for frequency, lag, offset and amplitude control
40 #define REG8_SINE_FREQ 10 // Register for sine wave frequency
41 #define REG8_SINE_AMP 11 // Register for sine wave amplitude
42 #define REG8_SINE_LAG 12 // Register for sine wave lag between elements
43 #define REG8_SINE_OFF 13 // Register for sine wave offset
44
45 const char *TRACKING_PC_NAME = "biorobpc6"; ///< host name of the tracking PC
46 const uint16_t TRACKING_PORT = 10502; ///< port number of the tracking PC
47 const uint8_t RADIO_CHANNEL = 126; ///< robot radio channel
48 const char *INTERFACE = "COM3"; ///< robot radio interface
49
50 /// Aquarium dimensions in meters
51 const double AQUARIUM_WIDTH = 6.0;
52 const double AQUARIUM_HEIGHT = 2.0;
53
54 /// Function to map a value from one range to another
55 double map_value(double value, double in_min, double in_max, double out_min,
56                 double out_max) {
57     return (value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
58 }
59
```



```

60 // Function to display current settings
61 void display_settings(CRemoteRegs &regs) {
62     uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
63     uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);
64     uint8_t lag_reg = regs.get_reg_b(REG8_SINE_LAG);
65     uint8_t off_reg = regs.get_reg_b(REG8_SINE_OFF);
66
67     float freq = DECODE_PARAM_8(freq_reg, MIN_FREQ, MAX_FREQ);
68     float amplitude = DECODE_PARAM_8(amp_reg, MIN_AMP, MAX_AMP);
69     float lag = DECODE_PARAM_8(lag_reg, MIN_LAG, MAX_LAG);
70     float offset = DECODE_PARAM_8(off_reg, MIN_OFF, MAX_OFF);
71
72     cout << "Current settings:" << endl;
73     cout << "Frequency:" << freq << "Hz" << (int)freq_reg << " "
74         << endl;
75     cout << "Amplitude:" << amplitude << "degrees" << (int)amp_reg
76         << " " << endl;
77     cout << "Lag:" << lag << "degrees" << (int)lag_reg << " "
78         << endl;
79     cout << "Offset:" << offset << "degrees" << (int)off_reg << " "
80         << endl;
81 }
82
83 // Function to update a parameter
84 void update_parameter(CRemoteRegs &regs, const char *name, uint8_t reg,
85                     float min_value, float max_value,
86                     float current) {
87     float new_value;
88     cout << "Enter new " << name << " (" << min_value << "-" << max_value
89         << ") [" << current << "]: ";
90
91     string input;
92     getline(cin, input);
93
94     if (input.empty()) {
95         cout << "Keeping current value." << endl;
96         return;
97     }
98
99     try {
100         new_value = stof(input);
101
102         // Validate the input
103         if (new_value < min_value || new_value > max_value) {
104             cout << "Value out of range. Using closest valid value." << endl;
105             new_value = (new_value < min_value) ? min_value : max_value;
106         }
107
108         // Encode and set the register
109         uint8_t encoded = ENCODE_PARAM_8(new_value, min_value, max_value);
110         regs.set_reg_b(reg, encoded);
111
112         cout << name << " set to " << new_value << " (" << (int)encoded
113             << ")" << endl;
114     } catch (const exception &e) {
115         cout << "Invalid input. Keeping current value." << endl;
116     }
117 }
118
119 void update_parameter_force(CRemoteRegs &regs, uint8_t reg, float min_value,
120                          float max_value, float value) {
121     // Ensure value is within bounds
122     if (value < min_value)
123         value = min_value;
124     if (value > max_value)
125         value = max_value;
126
127     // Encode and set the register
128     uint8_t encoded = ENCODE_PARAM_8(value, min_value, max_value);
129     regs.set_reg_b(reg, encoded);
130 }
131
132 int main() {
133     CTrackingClient trk;
134     CRemoteRegs regs;
135
136     cout << "Initializing robot connection..." << endl;
137     if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
138         cerr << "Failed to initialize radio interface" << endl;
139         return 1;
140     }

```

```

141
142 // Reboot the head microcontroller to ensure it's in a known state
143 reboot_head(regs);
144
145 cout << "Connecting to tracking system..." << endl;
146 if (!trk.connect(TRACKING_PC_NAME, TRACKING_PORT)) {
147     cerr << "Failed to connect to tracking system" << endl;
148     return 1;
149 }
150
151 cout << "Connected to tracking system. Set the robot to ready mode"
152      << "then place it in the aquarium."
153      << endl;
154
155 bool exitProgram = false;
156 char choice = '\0';
157
158 // Initialize parameters
159 float freq = 0.8f; // Default frequency in Hz
160 float amplitude = 40.0f; // Default amplitude
161 float lag = 0.75f; // Default lag
162 float offset = 0.0f; // Default offset
163
164 // Initialize the registers with default values
165 update_parameter_force(regs, REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ, freq);
166 update_parameter_force(regs, REG8_SINE_AMP, MIN_AMP, MAX_AMP, amplitude);
167 update_parameter_force(regs, REG8_SINE_LAG, MIN_LAG, MAX_LAG, lag);
168 update_parameter_force(regs, REG8_SINE_OFF, MIN_OFF, MAX_OFF, offset);
169
170 while (!exitProgram) {
171     // Get current parameter values
172     uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
173     uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);
174     uint8_t lag_reg = regs.get_reg_b(REG8_SINE_LAG);
175     uint8_t off_reg = regs.get_reg_b(REG8_SINE_OFF);
176     uint8_t mode_reg = regs.get_reg_b(REG8_MODE);
177
178     freq = DECODE_PARAM_8(freq_reg, MIN_FREQ, MAX_FREQ);
179     amplitude = DECODE_PARAM_8(amp_reg, MIN_AMP, MAX_AMP);
180     lag = DECODE_PARAM_8(lag_reg, MIN_LAG, MAX_LAG);
181     offset = DECODE_PARAM_8(off_reg, MIN_OFF, MAX_OFF);
182
183     cout << "\n===== \n";
184     cout << "Current mode: "
185          << (mode_reg == IMODE_IDLE
186              ? "IDLE"
187              : (mode_reg == IMODE_READY
188                  ? "READY"
189                  : (mode_reg == IMODE_SWIM ? "SWIM" : "UNKNOWN")))
190          << endl;
191     cout << "Select an option: \n";
192     cout << "1. Edit frequency \n";
193     cout << "2. Edit amplitude \n";
194     cout << "3. Edit lag \n";
195     cout << "4. Edit offset \n";
196     cout << "5. Display current settings \n";
197     cout << "6. Ready mode \n";
198     cout << "7. Swim mode \n";
199     cout << "8. Interactive mode \n";
200     cout << "0. Stop (idle mode) \n";
201     cout << "q. Quit \n";
202
203     cout << "Enter your choice: ";
204
205     cin >> choice;
206     // Clear any extra characters from the input buffer.
207     cin.ignore(10000, '\n');
208
209     switch (choice) {
210     case '1':
211         update_parameter(regs, "frequency", REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ,
212                          freq);
213         break;
214
215     case '2':
216         update_parameter(regs, "amplitude", REG8_SINE_AMP, MIN_AMP, MAX_AMP,
217                          amplitude);
218         break;
219
220     case '3':
221         update_parameter(regs, "lag", REG8_SINE_LAG, MIN_LAG, MAX_LAG, lag);

```

```

222     break;
223
224 case '4':
225     update_parameter(regs, "offset", REG8_SINE_OFF, MIN_OFF, MAX_OFF,
226                     offset);
227     break;
228
229 case '5':
230     display_settings(regs);
231     break;
232
233 case '6':
234     cout << "Setting robot to ready mode..." << endl;
235     regs.set_reg_b(REG8_MODE, IMODE_READY);
236     break;
237
238 case '7': {
239     cout << "Setting robot to swim mode..." << endl;
240     regs.set_reg_b(REG8_MODE, IMODE_SWIM);
241
242     // Create a filename with timestamp
243     time_t now = time(0);
244     tm *ltm = localtime(&now);
245     string filename = "robot_position_" + to_string(ltm->tm_year + 1900) +
246                     "_" + to_string(ltm->tm_mon + 1) + "_" +
247                     to_string(ltm->tm_mday) + "_" +
248                     to_string(ltm->tm_hour) + "_" + to_string(ltm->tm_min) +
249                     "_" + to_string(ltm->tm_sec) + "_freq_" + to_string(freq) + "_amp_" +
250                     to_string(amplitude) + "_lag_" + to_string(lag) + "_off_" +
251                     to_string(offset) + ".csv";
252
253     // Write header to CSV file
254     ofstream file(filename);
255     if (file.is_open()) {
256         file << "Timestamp,X,Y" << endl;
257         file.close();
258     } else {
259         cerr << "Unable to create log file" << endl;
260     }
261
262     cout << "Press any key to stop swimming..." << endl;
263
264     bool swimming = true;
265     while (swimming) {
266         uint32_t frame_time;
267         // Gets the current position
268         if (!trk.update(frame_time)) {
269             cerr << "Error updating tracking data" << endl;
270             break;
271         }
272
273         double x = 0, y = 0;
274         // bool detected = false;
275
276         // Gets the ID of the first spot
277         int id = trk.get_first_id();
278
279         // Reads its coordinates (if (id == -1), then no spot is detected)
280         if (id != -1 && trk.get_pos(id, x, y)) {
281             // detected = true;
282
283             // Get the current time as milliseconds since epoch
284             auto now_ms = chrono::duration_cast<chrono::milliseconds>(
285                 chrono::system_clock::now().time_since_epoch());
286
287             // Log the position to file
288             ofstream datafile(filename, ios::app);
289             if (datafile.is_open()) {
290                 datafile << now_ms.count() << "," << fixed << setprecision(3) << x
291                     << "," << y << endl;
292                 datafile.close();
293             }
294
295             cout << "Position: (" << fixed << setprecision(3) << x << ", " << y
296                 << ") " << endl;
297         } else {
298             cout << "Position: (not detected) " << endl;
299         }
300         cout.flush();
301
302         // Check for key press to exit

```

```

303     if (kbhit()) {
304         swimming = false;
305         ext_key(); // Consume the key
306     }
307
308     // Small delay to prevent excessive CPU usage
309     Sleep(10);
310 }
311
312 cout << endl << "Swimming stopped." << endl;
313 regs.set_reg_b(REG8_MODE, IMODE_IDLE);
314 break;
315 }
316
317 case '8': {
318     cout << "Starting interactive mode..." << endl;
319     regs.set_reg_b(REG8_MODE, IMODE_SWIM);
320
321     cout << "Use keyboard controls:" << endl;
322     cout << "W/S: Increase/decrease speed (frequency)" << endl;
323     cout << "A/D: Turn left/right (offset)" << endl;
324     cout << "Q: Return to menu" << endl;
325
326     bool interactiveMode = true;
327     while (interactiveMode) {
328         if (kbhit()) {
329             DWORD key = ext_key();
330             char c = key & 0xFF;
331
332             switch (c) {
333                 case 'w':
334                 case 'W':
335                     freq += 0.1f;
336                     update_parameter_force(regs, REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ,
337                                             freq);
338                     cout << "Frequency: " << freq << " Hz" << "\r";
339                     break;
340
341                 case 's':
342                 case 'S':
343                     freq = max(0.1f, freq - 0.1f);
344                     update_parameter_force(regs, REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ,
345                                             freq);
346                     cout << "Frequency: " << freq << " Hz" << "\r";
347                     break;
348
349                 case 'a':
350                 case 'A':
351                     offset -= 0.1f;
352                     update_parameter_force(regs, REG8_SINE_OFF, MIN_OFF, MAX_OFF,
353                                             offset);
354                     cout << "Offset: " << offset << " degrees" << "\r";
355                     break;
356
357                 case 'd':
358                 case 'D':
359                     offset += 0.1f;
360                     update_parameter_force(regs, REG8_SINE_OFF, MIN_OFF, MAX_OFF,
361                                             offset);
362                     cout << "Offset: " << offset << " degrees" << "\r";
363                     break;
364
365                 case 'q':
366                 case 'Q':
367                     interactiveMode = false;
368                     break;
369             }
370             cout.flush();
371         }
372
373         // Update tracking display
374         uint32_t frame_time;
375         if (trk.update(frame_time)) {
376             double x, y;
377             int id = trk.get_first_id();
378             if (id != -1 && trk.get_pos(id, x, y)) {
379                 cout << "Position: (" << fixed << setprecision(3) << x << ", " << y
380                     << ") | Freq: " << freq << " Hz | Offset: " << offset
381                     << " " << "\r";
382                 cout.flush();
383             }

```

```

384     }
385
386     Sleep(10); // Small delay to prevent excessive CPU usage
387 }
388
389 cout << endl << "Interactive_mode_stopped." << endl;
390 regs.set_reg_b(REG8_MODE, IMODE_IDLE);
391 break;
392 }
393
394 case '0':
395     cout << "Stopping_robot(idle_mode)..." << endl;
396     regs.set_reg_b(REG8_MODE, IMODE_IDLE);
397     break;
398
399 case 'q':
400 case 'Q':
401     exitProgram = true;
402     cout << "\nExiting_program." << endl;
403     break;
404
405 default:
406     cout << "\nInvalid_choice._Please_try_again." << endl;
407     break;
408 }
409 }
410
411 // Make sure the robot is stopped before exiting
412 regs.set_reg_b(REG8_MODE, IMODE_IDLE);
413
414 // Clears the console input buffer
415 FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
416
417 return 0;
418 }

```

This is the code that run on the robot for the part 7 and 8.

```

1
2 #include "config.h"
3 #include "hardware.h"
4 #include "modes.h"
5 #include "module.h"
6 #include "registers.h"
7 #include "robot.h"
8
9 // Define registers for frequency and amplitude control
10 #define REG8_SINE_FREQ 10 // Register for sine wave frequency
11 #define REG8_SINE_AMP 11 // Register for sine wave amplitude
12 #define REG8_SINE_LAG 12 // Register for sine wave lag between elements
13 #define REG8_SINE_OFF 13 // Register for sine wave offset
14
15 // Define limits for frequency and amplitude
16 #define MAX_FREQ 1.5f // Maximum frequency in Hz
17 // #define MIN_FREQ 0.0f // Minimum frequency in Hz
18 #define MAX_AMP 60.0f // Maximum amplitude in degrees
19 // #define MIN_AMP 0.0f // Minimum amplitude in degrees
20 #define MAX_LAG 1.5f // Maximum lag between elements in degrees
21 // #define MIN_LAG 0.0f // Minimum lag between elements in degrees
22 #define MAX_OFF 3.0f // Maximum lag between elements in degrees (180 mean straight for float conversion)
23 // #define MIN_OFF -180.0f // Minimum lag between elements in degrees
24
25 #define MIN_FREQ 0.1f
26 #define MIN_AMP 1.0f // Min amplitude in degrees
27 #define MIN_LAG 0.5f // Min lag between elements in degrees
28 #define MIN_OFF -3.0f // Min offset in degrees
29
30 // Default values
31 #define DEFAULT_FREQ 0.8f // Default frequency in Hz
32 #define DEFAULT_AMP 40.0f // Default amplitude in degrees
33 #define DEFAULT_LAG 0.75f // Default amplitude in degrees
34 #define DEFAULT_OFF 0.0f // Default amplitude in degrees
35
36
37 const uint8_t MOTOR_ADDR_HEAD = 25; // Motor address (this is the second element, the first is the head that
38 const uint8_t MOTOR_ADDR_NECK = 22; // Motor address
39 const uint8_t MOTOR_ADDR_TORSO = 24; // Motor address
40 const uint8_t MOTOR_ADDR_HIP = 26; // Motor address
41 const uint8_t MOTOR_ADDR_TAIL = 5; // Motor address

```

```

42
43 uint8_t freq_enc = DEFAULT_FREQ;
44 uint8_t amp_enc = DEFAULT_AMP;
45 uint8_t lag_enc = DEFAULT_LAG;
46 uint8_t off_enc = DEFAULT_OFF;
47
48 static int8_t register_handler(uint8_t operation, uint8_t address,
49                               RadioData *radio_data) {
50
51     switch (address) {
52         case REG8_MODE:
53             switch (operation) {
54                 case ROP_READ_8:
55                     radio_data->byte = reg8_table[REG8_MODE];
56                     return TRUE;
57                 case ROP_WRITE_8:
58                     reg8_table[REG8_MODE] = radio_data->byte; // Allow writing to register
59                     return TRUE;
60             }
61         case REG8_SINE_FREQ:
62             switch (operation) {
63                 case ROP_READ_8:
64                     radio_data->byte = freq_enc;
65                     return TRUE;
66                 case ROP_WRITE_8:
67                     freq_enc = radio_data->byte; // Allow writing to register
68                     return TRUE;
69             }
70         case REG8_SINE_AMP:
71             switch (operation) {
72                 case ROP_READ_8:
73                     radio_data->byte = amp_enc;
74                     return TRUE;
75                 case ROP_WRITE_8:
76                     amp_enc = radio_data->byte; // Allow writing to register
77                     return TRUE;
78             }
79         case REG8_SINE_LAG:
80             switch (operation) {
81                 case ROP_READ_8:
82                     radio_data->byte = lag_enc;
83                     return TRUE;
84                 case ROP_WRITE_8:
85                     lag_enc = radio_data->byte; // Allow writing to register
86                     return TRUE;
87             }
88         case REG8_SINE_OFF:
89             switch (operation) {
90                 case ROP_READ_8:
91                     radio_data->byte = off_enc;
92                     return TRUE;
93                 case ROP_WRITE_8:
94                     off_enc = radio_data->byte; // Allow writing to register
95                     return TRUE;
96             }
97     }
98     return FALSE;
99 }
100
101 // Function to initialize default parameters
102 void init_sine_params(void) {
103     // Set default values for frequency and amplitude
104     freq_enc = ENCODE_PARAM_8(DEFAULT_FREQ, MIN_FREQ, MAX_FREQ);
105     amp_enc = ENCODE_PARAM_8(DEFAULT_AMP, MIN_AMP, MAX_AMP);
106     lag_enc = ENCODE_PARAM_8(DEFAULT_LAG, MIN_LAG, MAX_LAG);
107     off_enc = ENCODE_PARAM_8(DEFAULT_OFF, MIN_OFF, MAX_OFF);
108 }
109
110 void swim_mode(void) {
111     uint32_t dt, cycletimer;
112     float my_time, delta_t, angle0, angle1, angle2, angle3, angle4;
113     float freq, amplitude, lag, offset;
114     int8_t angle0_rounded, angle1_rounded, angle2_rounded, angle3_rounded, angle4_rounded;
115
116     // Initialize and start the motor's PID controller
117     init_body_module(MOTOR_ADDR_HEAD);
118     init_body_module(MOTOR_ADDR_NECK);
119     init_body_module(MOTOR_ADDR_TORSO);
120     init_body_module(MOTOR_ADDR_HIP);
121     init_body_module(MOTOR_ADDR_TAIL);

```

```

123
124 start_pid(MOTOR_ADDR_HEAD);
125 start_pid(MOTOR_ADDR_NECK);
126 start_pid(MOTOR_ADDR_TORSO);
127 start_pid(MOTOR_ADDR_HIP);
128 start_pid(MOTOR_ADDR_TAIL);
129
130 set_reg_value_dw(MOTOR_ADDR_HEAD, MREG32_LED, 0);
131 set_reg_value_dw(MOTOR_ADDR_NECK, MREG32_LED, 0);
132 set_reg_value_dw(MOTOR_ADDR_TORSO, MREG32_LED, 0);
133 set_reg_value_dw(MOTOR_ADDR_HIP, MREG32_LED, 0);
134 set_reg_value_dw(MOTOR_ADDR_TAIL, MREG32_LED, 0);
135
136 // Set visual indicator that motor is active
137 set_color(4); // Set LED to red
138
139 // Initialize sine wave time
140 cycletimer = getSysTICs();
141 my_time = 0;
142
143 do {
144     // Decode current parameters from registers
145     freq = DECODE_PARAM_8(freq_enc, MIN_FREQ, MAX_FREQ);
146     amplitude = DECODE_PARAM_8(amp_enc, MIN_AMP, MAX_AMP);
147     lag = DECODE_PARAM_8(lag_enc, MIN_LAG, MAX_LAG);
148     offset = DECODE_PARAM_8(off_enc, MIN_OFF, MAX_OFF);
149
150     // Apply limits to ensure safety
151     if (freq > MAX_FREQ)
152         freq = MAX_FREQ;
153     if (amplitude > MAX_AMP)
154         amplitude = MAX_AMP;
155     if (lag > MAX_LAG)
156         lag = MAX_LAG;
157     if (offset > MAX_OFF)
158         offset = MAX_OFF;
159
160     // Calculate elapsed time
161     dt = getElapsedSysTICs(cycletimer);
162     cycletimer = getSysTICs();
163     delta_t = (float)dt / sysTICSPerSEC;
164     my_time += delta_t;
165
166     // Calculate the sine wave for motor angle
167     angle0 = amplitude * sin(M_TWOPI * ((freq * my_time)+(0*lag/5)+offset));
168     angle1 = amplitude * sin(M_TWOPI * ((freq * my_time)+(1*lag/5)+offset));
169     angle2 = amplitude * sin(M_TWOPI * ((freq * my_time)+(2*lag/5)+offset));
170     angle3 = amplitude * sin(M_TWOPI * ((freq * my_time)+(3*lag/5)+offset));
171     angle4 = amplitude * sin(M_TWOPI * ((freq * my_time)+(4*lag/5)+offset));
172
173     // Convert angle to motor units
174     angle0_rounded = DEG_TO_OUTPUT_BODY(angle0);
175     angle1_rounded = DEG_TO_OUTPUT_BODY(angle1);
176     angle2_rounded = DEG_TO_OUTPUT_BODY(angle2);
177     angle3_rounded = DEG_TO_OUTPUT_BODY(angle3);
178     angle4_rounded = DEG_TO_OUTPUT_BODY(angle4);
179
180     // Send the angle to the motor
181     bus_set(MOTOR_ADDR_HEAD, MREG_SETPPOINT, angle4_rounded);
182     bus_set(MOTOR_ADDR_NECK, MREG_SETPPOINT, angle3_rounded);
183     bus_set(MOTOR_ADDR_TORSO, MREG_SETPPOINT, angle2_rounded);
184     bus_set(MOTOR_ADDR_HIP, MREG_SETPPOINT, angle1_rounded);
185     bus_set(MOTOR_ADDR_TAIL, MREG_SETPPOINT, angle0_rounded);
186
187     set_rgb(255, 255, 255);
188
189     // Small delay to ensure timer updates properly
190     pause(ONE_MS);
191
192 } while (reg8_table[REG8_MODE] == IMODE_SWIM);
193
194 // Clean up: return motor to zero position
195 bus_set(MOTOR_ADDR_HEAD, MREG_SETPPOINT, 0);
196 bus_set(MOTOR_ADDR_NECK, MREG_SETPPOINT, 0);
197 bus_set(MOTOR_ADDR_TORSO, MREG_SETPPOINT, 0);
198 bus_set(MOTOR_ADDR_HIP, MREG_SETPPOINT, 0);
199 bus_set(MOTOR_ADDR_TAIL, MREG_SETPPOINT, 0);
200
201 pause(ONE_SEC); // Give the motor time to return to center
202
203 // Stop the motor

```

```

204 bus_set(MOTOR_ADDR_HEAD, MREG_MODE, MODE_IDLE);
205 bus_set(MOTOR_ADDR_NECK, MREG_MODE, MODE_IDLE);
206 bus_set(MOTOR_ADDR_TORSO, MREG_MODE, MODE_IDLE);
207 bus_set(MOTOR_ADDR_HIP, MREG_MODE, MODE_IDLE);
208 bus_set(MOTOR_ADDR_TAIL, MREG_MODE, MODE_IDLE);
209
210 // Return LED to normal state
211 set_color(2);
212 }
213
214 void ready_mode(void) {
215     // Initialize and start the motor's PID controller
216     init_body_module(MOTOR_ADDR_HEAD);
217     init_body_module(MOTOR_ADDR_NECK);
218     init_body_module(MOTOR_ADDR_TORSO);
219     init_body_module(MOTOR_ADDR_HIP);
220     init_body_module(MOTOR_ADDR_TAIL);
221
222     start_pid(MOTOR_ADDR_HEAD);
223     start_pid(MOTOR_ADDR_NECK);
224     start_pid(MOTOR_ADDR_TORSO);
225     start_pid(MOTOR_ADDR_HIP);
226     start_pid(MOTOR_ADDR_TAIL);
227
228     set_reg_value_dw(MOTOR_ADDR_HEAD, MREG32_LED, 0);
229     set_reg_value_dw(MOTOR_ADDR_NECK, MREG32_LED, 0);
230     set_reg_value_dw(MOTOR_ADDR_TORSO, MREG32_LED, 0);
231     set_reg_value_dw(MOTOR_ADDR_HIP, MREG32_LED, 0);
232     set_reg_value_dw(MOTOR_ADDR_TAIL, MREG32_LED, 0);
233     set_rgb(255, 255, 255);
234
235
236     // Send the angle to the motor
237     bus_set(MOTOR_ADDR_HEAD, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(40)); // adopt a rigid S shape to prevent caps
238     bus_set(MOTOR_ADDR_NECK, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(40));
239     bus_set(MOTOR_ADDR_TORSO, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-40));
240     bus_set(MOTOR_ADDR_HIP, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-40));
241     bus_set(MOTOR_ADDR_TAIL, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(40));
242
243     do { // wait until we start swimming or revert to limp mode.
244         // Small delay to ensure timer updates properly
245         pause(ONE_MS);
246     } while (reg8_table[REG8_MODE] == IMODE_READY);
247
248
249     // Clean up: return motor to zero position
250     bus_set(MOTOR_ADDR_HEAD, MREG_SETPOINT, 0);
251     bus_set(MOTOR_ADDR_NECK, MREG_SETPOINT, 0);
252     bus_set(MOTOR_ADDR_TORSO, MREG_SETPOINT, 0);
253     bus_set(MOTOR_ADDR_HIP, MREG_SETPOINT, 0);
254     bus_set(MOTOR_ADDR_TAIL, MREG_SETPOINT, 0);
255
256     pause(ONE_SEC); // Give the motor time to return to center
257
258     // Stop the motor
259     bus_set(MOTOR_ADDR_HEAD, MREG_MODE, MODE_IDLE);
260     bus_set(MOTOR_ADDR_NECK, MREG_MODE, MODE_IDLE);
261     bus_set(MOTOR_ADDR_TORSO, MREG_MODE, MODE_IDLE);
262     bus_set(MOTOR_ADDR_HIP, MREG_MODE, MODE_IDLE);
263     bus_set(MOTOR_ADDR_TAIL, MREG_MODE, MODE_IDLE);
264
265     // Return LED to normal state
266     set_color(2);
267 }
268
269
270 void main_mode_loop(void) {
271     // Initialize the default parameters
272     init_sine_params();
273
274     // Set initial mode
275     reg8_table[REG8_MODE] = IMODE_IDLE;
276
277     // Add the register handler
278     radio_add_reg_callback(register_handler);
279
280     while (1) {
281         switch (reg8_table[REG8_MODE]) {
282             case IMODE_IDLE:
283                 break;
284             case IMODE_READY :

```



```
285     ready_mode();
286     break;
287 case IMODE_SWIM:
288     swim_mode();
289     break;
290 default:
291     reg8_table[REG8_MODE] = IMODE_IDLE;
292 }
293 }
294 }
```