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)

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 :

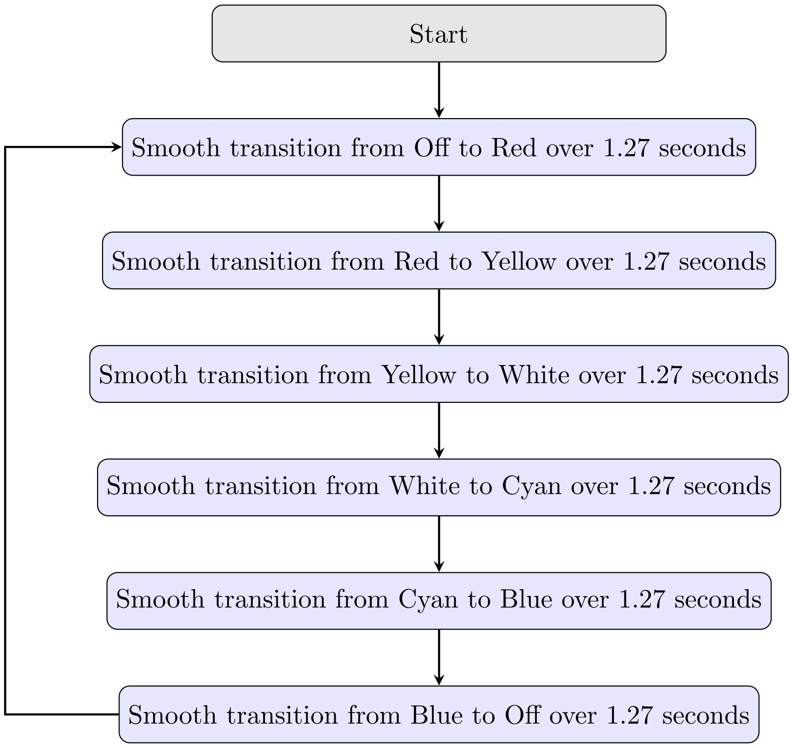


Figure 1: Color Transition Diagram

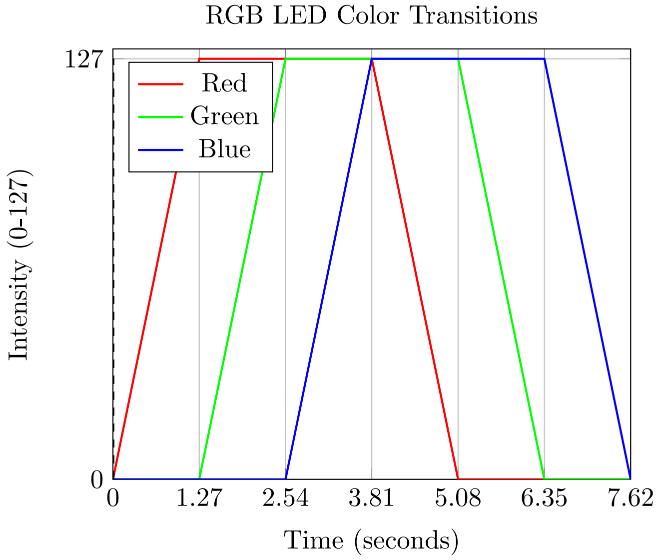


Figure 2: RGB LED Color Intensity

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

```
#include <stdint.h>
#include "hardware.h"
#include "registers.h"

int main(void)
{
  int8_t i;

  hardware_init();
  reg32_table[REG32_LED] = LED_MANUAL;  // manual LED control

  while (1) {
    pause(HALF_SEC);
    set_rgb(0, 255, 0); // turn on greenlight
    pause(HALF_SEC);
    set_rgb(0, 0, 0); // turn off greenlight
  }
  return 0;
}
```

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:

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

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

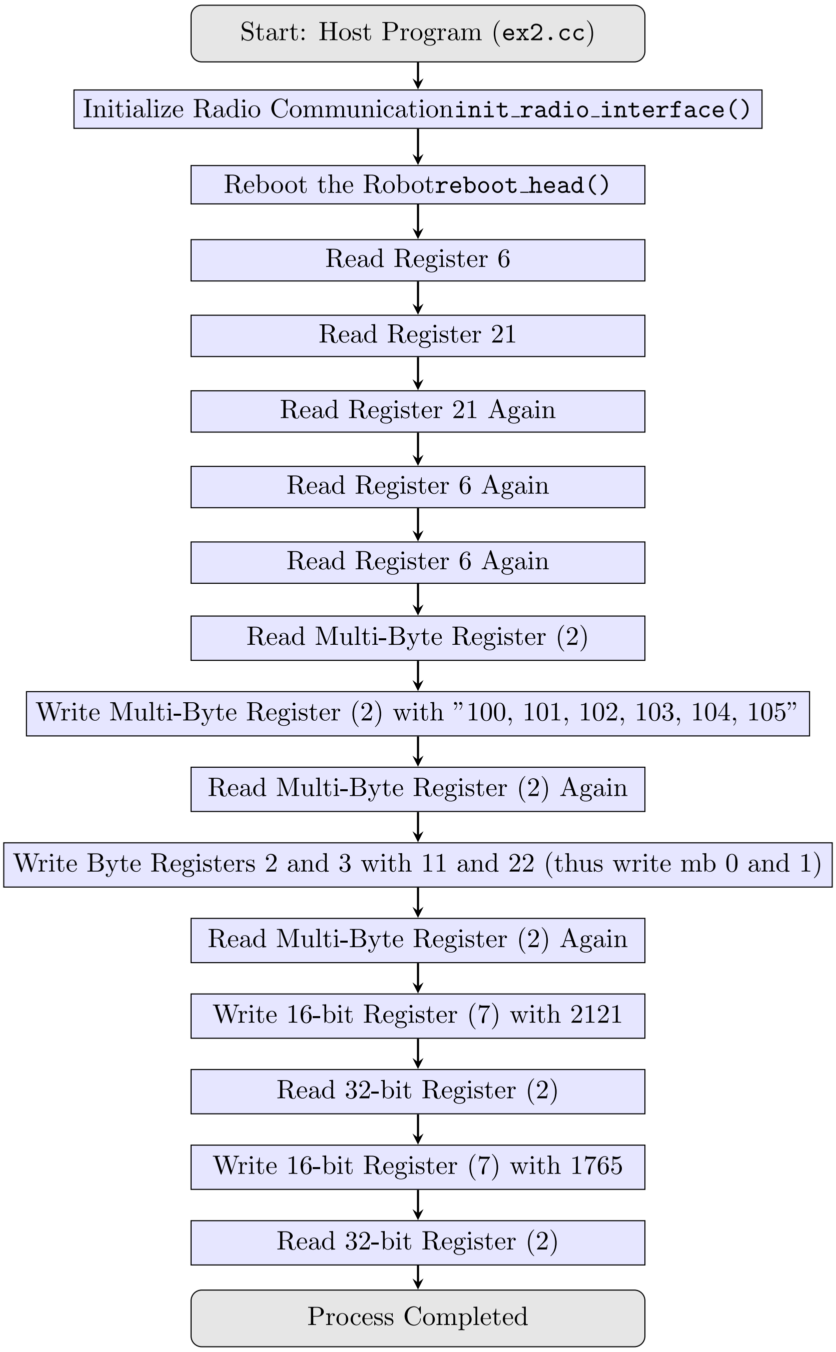
$$\texttt{datavar} = W_1 \times 3 + W_2$$

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

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.

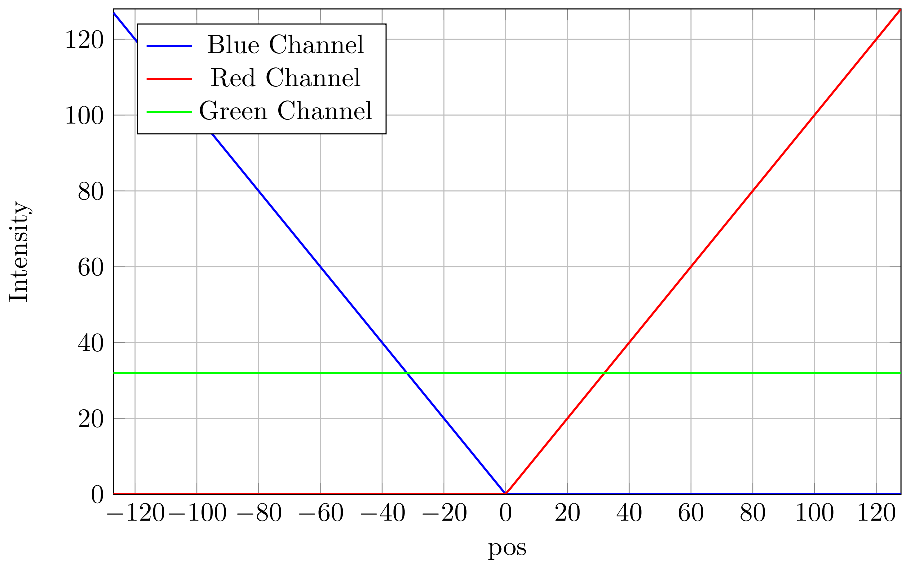
3.2 CAN Bus Communication

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.

```c
#include "hardware.h"
#include "registers.h"
#include "module.h"
#include "robot.h"
const uint8_t TAIL_MOTOR_ADDR = 21;
const uint8_t BODY_MOTOR_ADDR = 72;
const uint8_t BODY_MOTOR_FIN_LEFT = 73; // swap if not correct
const uint8_t BODY_MOTOR_FIN_RIGHT = 74;
static uint8_t motor_position = 0;
static uint8_t mb_buffer[4];
static uint8_t last_mb_size = 4;
/* Register callback function for 8-bit read/write operations */
static int8_t register_handler(uint8_t operation, uint8_t address, RadioData* radio_data)
{
    uint8_t i;
    switch (operation) {
    case ROP_READ_8:
        if (address == 6) {
            radio_data->byte = motor_position;
            return TRUE;
        }
    case ROP_WRITE_8:
        if (address == 6) {
            motor_position = radio_data->byte;  // Allow writing to register
            return TRUE;
        }
    case ROP_READ_MB:
        if (address == 2) {
            radio_data->multibyte.size = last_mb_size;
            for (i = 0; i < last_mb_size; i++) {
            radio_data->multibyte.data[i] = mb_buffer[i];
            }
            return TRUE;
        }
        break;
    case ROP_WRITE_MB:
        if (address == 2) {
            last_mb_size = radio_data->multibyte.size;
            for (i = 0; i < last_mb_size; i++) {
                mb_buffer[i] = radio_data->multibyte.data[i];
            }
            return TRUE;
        }
        break;
    }
```

```
    return FALSE;
}
int main(void)
{
    hardware_init();
    radio_add_reg_callback(register_handler); // Register the 8-bit handler
    init_body_module(TAIL_MOTOR_ADDR);
    init_body_module(BODY_MOTOR_ADDR);
    init_limb_module(BODY_MOTOR_FIN_LEFT);
    init_limb_module(BODY_MOTOR_FIN_RIGHT);
    // Indicate boot sequence
    set_color_i(4, 0);
    pause(ONE_SEC);
    set_color_i(2, 0);
    while (1) {
        motor_position = bus_get(TAIL_MOTOR_ADDR, MREG_POSITION); // Store position in register
        mb_buffer[0] = bus_get(TAIL_MOTOR_ADDR, MREG_POSITION); // Store position in register
        mb_buffer[1] = bus_get(BODY_MOTOR_ADDR, MREG_POSITION); // Store position in register
        mb_buffer[2] = bus_get(BODY_MOTOR_FIN_LEFT, MREG_POSITION); // Store position in register
        mb_buffer[3] = bus_get(BODY_MOTOR_FIN_RIGHT, MREG_POSITION); // Store position in register
        set_rgb((motor_position < 0) ? -motor_position : motor_position, 32, 0);
    }
    return 0;
}
```

and on the computer side the code will be :

```
#include <iostream>
#include <thread>
#include <chrono>
#include "remregs.h"
#include "robot.h"
#include "utils.h"
using namespace std;
const uint8_t RADIO_CHANNEL = 201;  ///< Robot radio channel
const char* INTERFACE = "COM1";     ///< Robot radio interface

// Displays the contents of a multibyte register as a list of bytes
void display_multibyte_register(CRemoteRegs& regs, const uint8_t addr)
{
  uint8_t data_buffer[32], len;
  if (regs.get_reg_mb(addr, data_buffer, len)) {
    cout << (int) len << " bytes: ";
    for (unsigned int i(0); i < len; i++) {
      if (i > 0) cout << ", ";
      cout << static_cast<int32_t>(static_cast<int8_t>(data_buffer[i]));
    }
    cout << endl;
  } else {
    cerr << "Unable to read multibyte register." << endl;
  }
}

int main()
{
  CRemoteRegs regs;

  if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
    return 1;
  }

  // Reboots the head microcontroller to ensure a consistent state
  reboot_head(regs);

  while (!kbhit()) {
    display_multibyte_register(regs, 2);
  }

  regs.close();
  return 0;
}
```

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 $\pm 40°$ 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.

```c
#include "modes.h"
#include "config.h"
#include "hardware.h"
#include "module.h"
#include "regdefs.h"
#include "registers.h"
#include "robot.h"
#include "sysTime.h"

const uint8_t MOTOR_ADDR = 21;
volatile int8_t motor_position = 0;

static int8_t register_handler(uint8_t operation, uint8_t address,
                               RadioData *radio_data) {

  if (address == REG8_MODE) {
    switch (operation) {
    case ROP_READ_8:
      radio_data->byte = reg8_table[REG8_MODE];
      return TRUE;
    case ROP_WRITE_8:
      reg8_table[REG8_MODE] = radio_data->byte; // Allow writing to register
      return TRUE;
    }
  }
  if (address == 0x06) {
    switch (operation) {
    case ROP_READ_8:
      radio_data->byte = motor_position;
      return TRUE;
    case ROP_WRITE_8:
      motor_position = radio_data->byte; // Allow writing to register
      return TRUE;
    }
  }
  return FALSE;
}

void motor_demo_mode() {
  init_body_module(MOTOR_ADDR);
  start_pid(MOTOR_ADDR);
```

```
  set_color(4);
  while (reg8_table[REG8_MODE] == IMODE_MOTOR_DEMO) {
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(21.0));
    pause(ONE_SEC);
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-21.0));
    pause(ONE_SEC);
  }
  bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
  pause(ONE_SEC);
  bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
  set_color(2);
}

void motor_sine_demo() {
  init_body_module(MOTOR_ADDR);
  start_pid(MOTOR_ADDR);
  set_color(3);
  while (reg8_table[REG8_MODE] == IMODE_SINE_DEMO) {
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(motor_position));
    pause(TEN_MS);
  }
  bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
  pause(ONE_SEC);
  bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
  set_color(2);
}

void main_mode_loop() {
  reg8_table[REG8_MODE] = IMODE_IDLE;
  radio_add_reg_callback(register_handler);
  while (1) {
    switch (reg8_table[REG8_MODE]) {
    case IMODE_IDLE:
      break;
    case IMODE_MOTOR_DEMO:
      motor_demo_mode();
      break;
    case IMODE_SINE_DEMO:
      motor_sine_demo();
      break;
    default:
      reg8_table[REG8_MODE] = IMODE_IDLE;
    }
  }
}
```

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

```
#include "regdefs.h"
#include "remregs.h"
#include "robot.h"
#include "utils.h"
#include <cmath>
// #include <conio.h> // For kbhit() and getch()
#include <iostream>
#define IMODE_IDLE 0
#define IMODE_MOTOR_DEMO 1
#define IMODE_SINE_DEMO 2
// Constants
const uint8_t RADIO_CHANNEL = 201; // Radio channel
const char *INTERFACE = "COM1";     // Serial port for radio interface
const uint8_t MOTOR_ADDR = 21;      // Motor address (from modes.c)
const double AMPLITUDE_DEG = 40.0; // Amplitude in degrees
const double FREQUENCY_HZ = 1.0;    // Frequency in Hz
using namespace std;

int main() {
      CRemoteRegs regs;

      cout << "Initializing robot connection..." << endl;
      if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
        return 1;
      }

      // Reboot the head microcontroller to ensure it is in a known state.
      reboot_head(regs);

      // Show initial mode
      uint8_t mode = regs.get_reg_b(REG8_MODE);
      cout << "Initial mode: " << static_cast<int>(mode) << endl;
```

```

    bool exitProgram = false;
    char choice = '\0';

    while (!exitProgram) {
      cout << "\n=====================================\n";
      cout << "Select an option:\n";
      cout << "1. Motor Demo Mode\n";
      cout << "2. Sine Wave Control Mode\n";
      cout << "q. Quit\n";
      cout << "Enter your choice: ";

      cin >> choice;
      // Clear any extra characters from the input buffer.
      cin.ignore(10000, '\n');

      switch (choice) {
      case '1': {
        cout << "\nStarting Motor Demo Mode..." << endl;
        regs.set_reg_b(REG8_MODE, IMODE_MOTOR_DEMO);
        cout << "Press Enter to stop motor demo." << endl;
        cin.get();
        regs.set_reg_b(REG8_MODE, IMODE_IDLE);
        cout << "Motor demo stopped. Waiting for motor to return to zero..."
             << endl;
        Sleep(2000); // Give time for the motor to settle
        break;
      }
      case '2': {
        cout << "\nStarting Sine Wave Control Mode..." << endl;
        regs.set_reg_b(REG8_MODE, IMODE_SINE_DEMO);
        cout << "Press any key to stop." << endl;

        double startTime = time_d(); // Get the start time
        double currentTime = 0.0;
        bool running = true;

        while (running) {
          // Calculate elapsed time since start
          currentTime = time_d() - startTime;

          // Calculate sine wave value (-1 to 1) and scale it to the desired
          // amplitude
          double angle =
              AMPLITUDE_DEG * sin(2.0 * M_PI * FREQUENCY_HZ * currentTime);

          // Send the setpoint to the motor (assuming register 0x06 is the
          // setpoint)
          regs.set_reg_b(0x06, static_cast<int8_t>(angle));

          // If a key is pressed, break out of the loop
          if (kbhit()) {
            // Clear the key from the buffer.
            //getch();
            running = false;
          }

          // Small delay to prevent overwhelming communication
          Sleep(10);
        }
        // Stop the motor and return to idle mode
        regs.set_reg_b(0x06, 0); // Set setpoint to 0
        regs.set_reg_b(REG8_MODE, IMODE_IDLE);
        cout << "Sine wave control stopped." << endl;
        break;
      }
      case 'q':
      case 'Q': {
        exitProgram = true;
        cout << "\nExiting program." << endl;
        break;
      }
      default: {
        cout << "\nInvalid choice. Please try again." << endl;
        break;
      }
      }
    }
    return 0;
  }
}
```

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 programm 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 ±40°). 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 ±60°.

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)

```

#include "regdefs.h"
#include "remregs.h"
#include "robot.h"
#include "utils.h"
#include <cmath>
#include <iostream>

#define IMODE_IDLE 0
#define IMODE_MOTOR_DEMO 1
#define IMODE_SINE_DEMO 2

// Constants
const uint8_t RADIO_CHANNEL = 201; // Radio channel
const char *INTERFACE = "COM1";     // Serial port for radio interface

// Define registers for frequency and amplitude control
#define REG8_SINE_FREQ 10 // Register for sine wave frequency
#define REG8_SINE_AMP 11  // Register for sine wave amplitude

// Define limits for frequency and amplitude
#define MAX_FREQ 2.0f // Maximum frequency in Hz
#define MAX_AMP 60.0f // Maximum amplitude in degrees

using namespace std;

// Function to display current settings
void display_settings(CRemoteRegs &regs) {
  uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
  uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);

```

```
  float freq = DECODE_PARAM_8(freq_reg, 0.1f, MAX_FREQ);
  float amplitude = DECODE_PARAM_8(amp_reg, 1.0f, MAX_AMP);

  cout << "Current␣settings:" << endl;
  cout << "␣␣Frequency:␣" << freq << "␣Hz␣(encoded:␣" << (int)freq_reg << ")"
       << endl;
  cout << "␣␣Amplitude:␣" << amplitude << "␣degrees␣(encoded:␣" << (int)amp_reg
       << ")" << endl;
}

// Function to update a parameter
void update_parameter(CRemoteRegs &regs, const char *name, uint8_t reg,
                      float min_value, float max_value, float scale,
                      float current) {
  float new_value;
  cout << "Enter␣new␣" << name << "␣(" << min_value << "␣-␣" << max_value
       << ")␣[" << current << "]:␣";

  string input;
  getline(cin, input);

  if (input.empty()) {
    cout << "Keeping␣current␣value." << endl;
    return;
  }

  try {
    new_value = stof(input);

    // Validate the input
    if (new_value < min_value || new_value > max_value) {
      cout << "Value␣out␣of␣range.␣Using␣closest␣valid␣value." << endl;
      new_value = (new_value < min_value) ? min_value : max_value;
    }

    // Encode and set the register
    uint8_t encoded = ENCODE_PARAM_8(new_value, scale, max_value);
    regs.set_reg_b(reg, encoded);

    cout << name << "␣set␣to␣" << new_value << "␣(encoded:␣" << (int)encoded
         << ")" << endl;
  } catch (const exception &e) {
    cout << "Invalid␣input.␣Keeping␣current␣value." << endl;
  }
}

int main() {
  CRemoteRegs regs;

  cout << "Sine␣Wave␣Controller␣for␣Fish␣Robot" << endl;
  cout << "-----------------------------------" << endl;

  cout << "Initializing␣robot␣connection␣on␣" << INTERFACE << ",␣channel␣"
       << (int)RADIO_CHANNEL << endl;
  if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
    cerr << "Failed␣to␣initialize␣radio␣interface" << endl;
    return 1;
  }

  // Reboot the head microcontroller
  cout << "Rebooting␣head..." << endl;
  reboot_head(regs);

  // Main control loop
  bool running = true;
  while (running) {
    // Get current parameter values
    uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
    uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);

    float freq = DECODE_PARAM_8(freq_reg, 0.1f, MAX_FREQ);
    float amplitude = DECODE_PARAM_8(amp_reg, 1.0f, MAX_AMP);

    // Check current mode
    uint8_t currentMode = regs.get_reg_b(REG8_MODE);

    // Display menu
    cout << "\n-----␣MENU␣-----" << endl;
    cout << "1.␣Start/Stop␣Sine␣Wave␣Demo" << endl;
    cout << "2.␣Set␣Frequency␣(current:␣" << freq << "␣Hz)" << endl;
    cout << "3.␣Set␣Amplitude␣(current:␣" << amplitude << "␣degrees)" << endl;
```

```cpp
    cout << "4. Display Current Settings" << endl;
    cout << "5. Exit" << endl;
    cout << "Current mode: "
         << (currentMode == IMODE_SINE_DEMO ? "RUNNING" : "STOPPED") << endl;
    cout << "Selection: ";

    string input;
    getline(cin, input);

    if (input.empty())
      continue;

    switch (input[0]) {
    case '1':
      if (currentMode != IMODE_SINE_DEMO) {
        cout << "Starting sine wave demo..." << endl;
        regs.set_reg_b(REG8_MODE, IMODE_SINE_DEMO);
      } else {
        cout << "Stopping sine wave demo..." << endl;
        regs.set_reg_b(REG8_MODE, IMODE_IDLE);
      }
      break;

    case '2':
      update_parameter(regs, "frequency", REG8_SINE_FREQ, 0.1f, MAX_FREQ, 0.1f,
                       freq);
      break;

    case '3':
      update_parameter(regs, "amplitude", REG8_SINE_AMP, 1.0f, MAX_AMP, 1.0f,
                       amplitude);
      break;

    case '4':
      display_settings(regs);
      break;

    case '5':
      // Make sure to stop the demo before exiting
      if (regs.get_reg_b(REG8_MODE) == IMODE_SINE_DEMO) {
        cout << "Stopping sine wave demo..." << endl;
        regs.set_reg_b(REG8_MODE, IMODE_IDLE);
        // Give it time to stop properly
        Sleep(2000);
      }
      running = false;
      break;

    default:
      cout << "Invalid selection. Please try again." << endl;
      break;
    }
  }

  cout << "Program terminated." << endl;
  return 0;
}
```

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.

```c
#include "config.h"
#include "hardware.h"
#include "modes.h"
#include "module.h"
#include "registers.h"
#include "robot.h"

// Define registers for frequency and amplitude control
#define REG8_SINE_FREQ 10 // Register for sine wave frequency
#define REG8_SINE_AMP 11  // Register for sine wave amplitude

// Define limits for frequency and amplitude
#define MAX_FREQ 2.0f // Maximum frequency in Hz
#define MAX_AMP 60.0f // Maximum amplitude in degrees

// Default values
#define DEFAULT_FREQ 1.0f // Default frequency in Hz
#define DEFAULT_AMP 40.0f // Default amplitude in degrees

const uint8_t MOTOR_ADDR = 21; // Motor address

uint8_t freq_enc = DEFAULT_FREQ;
uint8_t amp_enc = DEFAULT_AMP;


static int8_t register_handler(uint8_t operation, uint8_t address,
                               RadioData *radio_data) {

  switch (address) {
    case REG8_MODE:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = reg8_table[REG8_MODE];
        return TRUE;
      case ROP_WRITE_8:
        reg8_table[REG8_MODE] = radio_data->byte; // Allow writing to register
        return TRUE;
      }
    case REG8_SINE_FREQ:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = freq_enc;
        return TRUE;
      case ROP_WRITE_8:
        freq_enc = radio_data->byte; // Allow writing to register
        return TRUE;
      }
    case REG8_SINE_AMP:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = amp_enc;
        return TRUE;
      case ROP_WRITE_8:
        amp_enc = radio_data->byte; // Allow writing to register
        return TRUE;
      }
  }
  return FALSE;

}

// Function to initialize default parameters
void init_sine_params(void) {
  // Set default values for frequency and amplitude
  reg8_table[REG8_SINE_FREQ] = ENCODE_PARAM_8(DEFAULT_FREQ, 0.1f, MAX_FREQ);
  reg8_table[REG8_SINE_AMP] = ENCODE_PARAM_8(DEFAULT_AMP, 1.0f, MAX_AMP);
}

void sine_demo_mode(void) {
  uint32_t dt, cycletimer;
  float my_time, delta_t, angle;
  float freq, amplitude;
  int8_t angle_rounded;

  // Initialize and start the motor's PID controller
  init_body_module(MOTOR_ADDR);
  start_pid(MOTOR_ADDR);

  // Set visual indicator that motor is active
  set_color(4); // Set LED to red

```

```
  // Initialize sine wave time
  cycletimer = getSysTICs();
  my_time = 0;

  // Make sure parameters are initialized
  if (freq_enc == 0)
    freq_enc = ENCODE_PARAM_8(DEFAULT_FREQ, 0.1f, MAX_FREQ);
  if (amp_enc == 0)
    amp_enc = ENCODE_PARAM_8(DEFAULT_AMP, 1.0f, MAX_AMP);

  do {
    // Decode current parameters from registers
    freq = DECODE_PARAM_8(freq_enc, 0.1f, MAX_FREQ);
    amplitude = DECODE_PARAM_8(amp_enc, 1.0f, MAX_AMP);

    // Apply limits to ensure safety
    if (freq > MAX_FREQ)
      freq = MAX_FREQ;
    if (amplitude > MAX_AMP)
      amplitude = MAX_AMP;

    // Calculate elapsed time
    dt = getElapsedSysTICs(cycletimer);
    cycletimer = getSysTICs();
    delta_t = (float)dt / sysTICSperSEC;
    my_time += delta_t;

    // Calculate the sine wave for motor angle
    angle = amplitude * sin(M_TWOPI * freq * my_time);

    // Convert angle to motor units
    angle_rounded = DEG_TO_OUTPUT_BODY(angle);

    // Send the angle to the motor
    bus_set(MOTOR_ADDR, MREG_SETPOINT, angle_rounded);

    // Update LED for visual feedback - color indicates frequency,
    // brightness indicates amplitude
    uint8_t red = (uint8_t)(freq * 127.0f / MAX_FREQ);
    uint8_t green = (uint8_t)(amplitude * 127.0f / MAX_AMP);

    if (angle >= 0) {
      // Positive angle - more green
      set_rgb(red, green + 20, 20);
    } else {
      // Negative angle - more red
      set_rgb(red + 20, green, 20);
    }

    // Small delay to ensure timer updates properly
    pause(ONE_MS);

  } while (reg8_table[REG8_MODE] == IMODE_SINE_DEMO);

  // Clean up: return motor to zero position
  bus_set(MOTOR_ADDR, MREG_SETPOINT, 0);
  pause(ONE_SEC); // Give the motor time to return to center

  // Stop the motor
  bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);

  // Return LED to normal state
  set_color(2);
}

void main_mode_loop(void) {
  // Initialize the default parameters
  init_sine_params();

  // Set initial mode
  reg8_table[REG8_MODE] = IMODE_IDLE;


  // Add the register handler
  radio_add_reg_callback(register_handler);

  while (1) {
    switch (reg8_table[REG8_MODE]) {
    case IMODE_IDLE:
      break;
    case IMODE_SINE_DEMO:
```

```
      sine_demo_mode();
      break;
    default:
      reg8_table[REG8_MODE] = IMODE_IDLE;
    }
  }
}
```

5.6 Testing and Observations

The three mode where 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 positionning error is really low (1-2mm).

6.3 Implementation of LED color based on position

We modifed the provided code to make the RBG 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.

```
#include "can.h"
#include "hardware.h"
#include "module.h"
#include "registers.h"
#include "robot.h"
#include <stdint.h>

// Address of the motor module

int main(void) {
  hardware_init();
  registers_init();
  // Set head LED to visible for tracking (green=64)
  // We'll let the PC control the exact color

  // Main loop - just keep the system running
  while (1) {
    // The head will receive color commands from the PC program
    // No additional processing needed here
    pause(TEN_MS);
  }

  return 0;
}
```

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.

```
#include "remregs.h"
#include "robot.h"
#include "trkcli.h"
#include "utils.h"
#include "regdefs.h"
#include <cstdlib>
#include <iostream>
#include <stdint.h>
#include <windows.h>

using namespace std;

const char *TRACKING_PC_NAME = "biorobpc6"; ///< host name of the tracking PC
const uint16_t TRACKING_PORT = 10502;        ///< port number of the tracking PC
const uint8_t RADIO_CHANNEL = 201;           ///< robot radio channel
const char *INTERFACE = "COM1";              ///< robot radio interface

// Aquarium dimensions in meters
const double AQUARIUM_WIDTH = 6.0;
const double AQUARIUM_HEIGHT = 2.0;

// Function to map a value from one range to another
double map_value(double value, double in_min, double in_max, double out_min,
                 double out_max) {
  return (value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

int main() {
  CTrackingClient trk;
  CRemoteRegs regs;

  cout << "Initializing robot connection..." << endl;
  if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
    cerr << "Failed to initialize radio interface" << endl;
    return 1;
  }

  // Reboot the head microcontroller to ensure it's in a known state
  reboot_head(regs);

  cout << "Connecting to tracking system..." << endl;
  if (!trk.connect(TRACKING_PC_NAME, TRACKING_PORT)) {
    cerr << "Failed to connect to tracking system" << endl;
    return 1;
  }

  cout << "Connected to tracking system. Turn off other module LEDs and place "
          "the robot in the aquarium."
       << endl;
  cout << "Press any key to exit." << endl;

  // Turn off the LED of another module (assuming module address 21)
  const uint8_t MOTOR_ADDR = 21;
  // This code doesn't work directly from PC, needs to be on the robot side
  // Will be handled in the robot part of the solution

  // Fixed green component for tracking
  const uint8_t GREEN_COMPONENT = 64;

  while (!kbhit()) {
    uint32_t frame_time;
    // Gets the current position
    if (!trk.update(frame_time)) {
      cerr << "Error updating tracking data" << endl;
      return 1;
    }

    double x, y;

    // Gets the ID of the first spot
    int id = trk.get_first_id();

    // Reads its coordinates (if (id == -1), then no spot is detected)
    if (id != -1 && trk.get_pos(id, x, y)) {
      // Calculate LED color based on position
      // Red increases with x (left to right)
      uint8_t r = (uint8_t)map_value(x, 0, AQUARIUM_WIDTH, 0, 255);

```

```
            // Blue increases with y (bottom to top)
            uint8_t b = (uint8_t)map_value(y, 0, AQUARIUM_HEIGHT, 0, 255);

            // Keep green fixed for tracking
            uint8_t g = GREEN_COMPONENT;

            // Set the LED color
            uint32_t rgb = ((uint32_t)r << 16) | ((uint32_t)g << 8) | b;
            regs.set_reg_dw(REG32_LED, rgb);

            cout << "Position: (" << fixed << x << ", " << y << ") m  |  Color: RGB("
                 << (int)r << ", " << (int)g << ", " << (int)b << ")      \r";
            cout.flush();
        } else {
            cout << "Position: (not detected)                              \r";
            cout.flush();
        }

        // Wait 10 ms before getting the info next time
        Sleep(10);
    }

    // Clears the console input buffer (as kbhit() doesn't)

  FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));
    cout << endl << "Program terminated." << endl;
    return 0;
}
```

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.

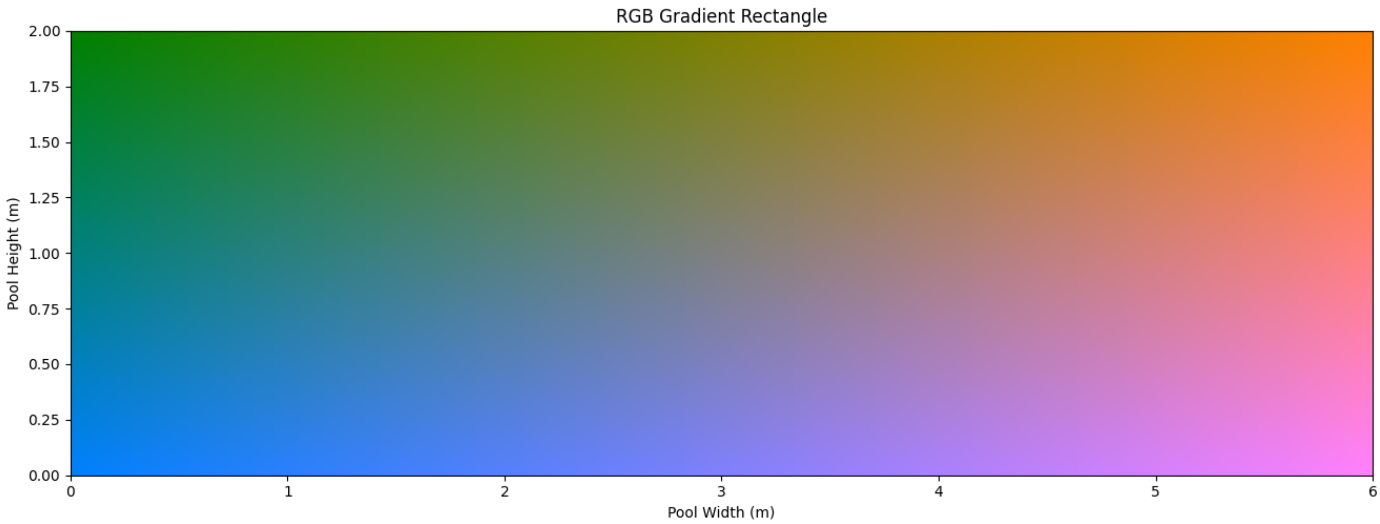


Figure 3: Color gradient

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:

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

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 `t` 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.

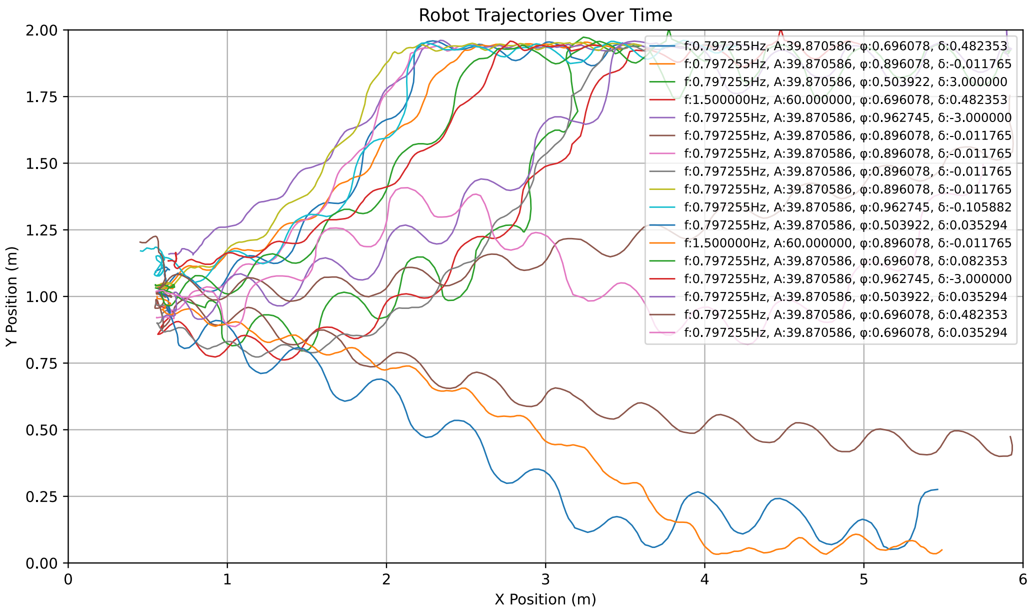


Figure 4: Pool trajectory of each run

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

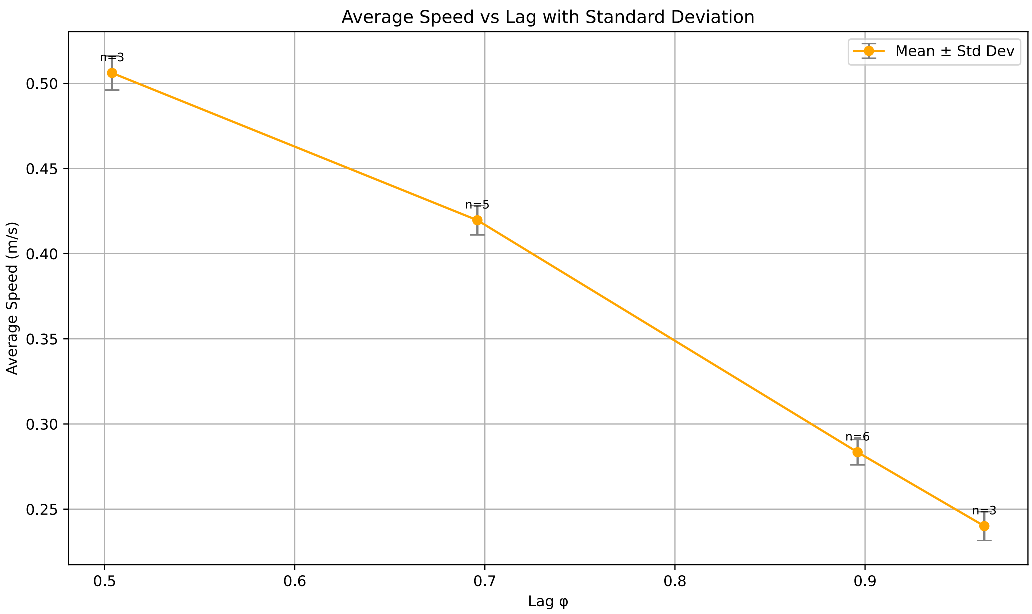


Figure 5: Plot of the robot speed over lag (radian)

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.

```
#include "regdefs.h"
#include "remregs.h"
#include "robot.h"
#include "trkcli.h"
#include "utils.h"
#include <chrono>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <stdint.h>
#include <string>
#include <windows.h>

using namespace std;

/// Idle mode: do nothing
#define IMODE_IDLE 0

/// set the robot in a warped and rigid position (prevent capsizing)
#define IMODE_READY 1

/// active swimming mode
#define IMODE_SWIM 2

// Define limits for frequency and amplitude
#define MAX_FREQ 1.5f  // Maximum frequency in Hz
#define MAX_AMP 60.0f  // Maximum amplitude in degrees
#define MAX_LAG 1.5f // Maximum lag between elements in degrees
#define MAX_OFF 3.0f // Maximum offset in degrees

#define MIN_FREQ 0.1f
#define MIN_AMP 1.0f  // Min amplitude in degrees
#define MIN_LAG 0.5f // Min lag between elements in degrees
#define MIN_OFF -3.0f // Min offset in degrees

// Define registers for frequency, lag, offset and amplitude control
#define REG8_SINE_FREQ 10 // Register for sine wave frequency
#define REG8_SINE_AMP 11  // Register for sine wave amplitude
#define REG8_SINE_LAG 12  // Register for sine wave lag between elements
#define REG8_SINE_OFF 13  // Register for sine wave offset

const char *TRACKING_PC_NAME = "biorobpc6"; ///< host name of the tracking PC
const uint16_t TRACKING_PORT = 10502;       ///< port number of the tracking PC
const uint8_t RADIO_CHANNEL = 126;          ///< robot radio channel
const char *INTERFACE = "COM3";             ///< robot radio interface

// Aquarium dimensions in meters
const double AQUARIUM_WIDTH = 6.0;
const double AQUARIUM_HEIGHT = 2.0;

// Function to map a value from one range to another
double map_value(double value, double in_min, double in_max, double out_min,
                 double out_max) {
  return (value - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

```

```cpp
// Function to display current settings
void display_settings(CRemoteRegs &regs) {
  uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
  uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);
  uint8_t lag_reg = regs.get_reg_b(REG8_SINE_LAG);
  uint8_t off_reg = regs.get_reg_b(REG8_SINE_OFF);

  float freq = DECODE_PARAM_8(freq_reg, MIN_FREQ, MAX_FREQ);
  float amplitude = DECODE_PARAM_8(amp_reg, MIN_AMP, MAX_AMP);
  float lag = DECODE_PARAM_8(lag_reg, MIN_LAG, MAX_LAG);
  float offset = DECODE_PARAM_8(off_reg, MIN_OFF, MAX_OFF);

  cout << "Current settings:" << endl;
  cout << "  Frequency: " << freq << " Hz (encoded: " << (int)freq_reg << ")"
       << endl;
  cout << "  Amplitude: " << amplitude << " degrees (encoded: " << (int)amp_reg
       << ")" << endl;
  cout << "  Lag: " << lag << " degrees (encoded: " << (int)lag_reg << ")"
       << endl;
  cout << "  Offset: " << offset << " degrees (encoded: " << (int)off_reg << ")"
       << endl;
}

// Function to update a parameter
void update_parameter(CRemoteRegs &regs, const char *name, uint8_t reg,
                      float min_value, float max_value,
                      float current) {
  float new_value;
  cout << "Enter new " << name << " (" << min_value << " - " << max_value
       << ") [" << current << "]: ";

  string input;
  getline(cin, input);

  if (input.empty()) {
    cout << "Keeping current value." << endl;
    return;
  }

  try {
    new_value = stof(input);

    // Validate the input
    if (new_value < min_value || new_value > max_value) {
      cout << "Value out of range. Using closest valid value." << endl;
      new_value = (new_value < min_value) ? min_value : max_value;
    }

    // Encode and set the register
    uint8_t encoded = ENCODE_PARAM_8(new_value, min_value, max_value);
    regs.set_reg_b(reg, encoded);

    cout << name << " set to " << new_value << " (encoded: " << (int)encoded
         << ")" << endl;
  } catch (const exception &e) {
    cout << "Invalid input. Keeping current value." << endl;
  }
}

void update_parameter_force(CRemoteRegs &regs, uint8_t reg, float min_value,
                            float max_value, float value) {
  // Ensure value is within bounds
  if (value < min_value)
    value = min_value;
  if (value > max_value)
    value = max_value;

  // Encode and set the register
  uint8_t encoded = ENCODE_PARAM_8(value, min_value, max_value);
  regs.set_reg_b(reg, encoded);
}

int main() {
  CTrackingClient trk;
  CRemoteRegs regs;

  cout << "Initializing robot connection..." << endl;
  if (!init_radio_interface(INTERFACE, RADIO_CHANNEL, regs)) {
    cerr << "Failed to initialize radio interface" << endl;
    return 1;
  }
```

```cpp

  // Reboot the head microcontroller to ensure it's in a known state
  reboot_head(regs);

  cout << "Connecting to tracking system..." << endl;
  if (!trk.connect(TRACKING_PC_NAME, TRACKING_PORT)) {
    cerr << "Failed to connect to tracking system" << endl;
    return 1;
  }

  cout << "Connected to tracking system. Set the robot to ready mode "
          "then place it in the aquarium."
       << endl;

  bool exitProgram = false;
  char choice = '\0';

  // Initialize parameters
  float freq = 0.8f;        // Default frequency in Hz
  float amplitude = 40.0f; // Default amplitude
  float lag = 0.75f;        // Default lag
  float offset = 0.0f;      // Default offset

  // Initialize the registers with default values
  update_parameter_force(regs, REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ, freq);
  update_parameter_force(regs, REG8_SINE_AMP, MIN_AMP, MAX_AMP, amplitude);
  update_parameter_force(regs, REG8_SINE_LAG, MIN_LAG, MAX_LAG, lag);
  update_parameter_force(regs, REG8_SINE_OFF, MIN_OFF, MAX_OFF, offset);

  while (!exitProgram) {
    // Get current parameter values
    uint8_t freq_reg = regs.get_reg_b(REG8_SINE_FREQ);
    uint8_t amp_reg = regs.get_reg_b(REG8_SINE_AMP);
    uint8_t lag_reg = regs.get_reg_b(REG8_SINE_LAG);
    uint8_t off_reg = regs.get_reg_b(REG8_SINE_OFF);
    uint8_t mode_reg = regs.get_reg_b(REG8_MODE);

    freq = DECODE_PARAM_8(freq_reg, MIN_FREQ, MAX_FREQ);
    amplitude = DECODE_PARAM_8(amp_reg, MIN_AMP, MAX_AMP);
    lag = DECODE_PARAM_8(lag_reg, MIN_LAG, MAX_LAG);
    offset = DECODE_PARAM_8(off_reg, MIN_OFF, MAX_OFF);

    cout << "\n====================================\n";
    cout << "Current mode: "
         << (mode_reg == IMODE_IDLE
                 ? "IDLE"
                 : (mode_reg == IMODE_READY
                        ? "READY"
                        : (mode_reg == IMODE_SWIM ? "SWIM" : "UNKNOWN")))
         << endl;
    cout << "Select an option:\n";
    cout << "1. Edit frequency\n";
    cout << "2. Edit amplitude\n";
    cout << "3. Edit lag\n";
    cout << "4. Edit offset\n";
    cout << "5. Display current settings\n";
    cout << "6. Ready mode\n";
    cout << "7. Swim mode\n";
    cout << "8. Interactive mode\n";
    cout << "0. Stop (idle mode)\n";
    cout << "q. Quit\n";

    cout << "Enter your choice: ";

    cin >> choice;
    // Clear any extra characters from the input buffer.
    cin.ignore(10000, '\n');

    switch (choice) {
    case '1':
      update_parameter(regs, "frequency", REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ,
                       freq);
      break;

    case '2':
      update_parameter(regs, "amplitude", REG8_SINE_AMP, MIN_AMP, MAX_AMP,
                       amplitude);
      break;

    case '3':
      update_parameter(regs, "lag", REG8_SINE_LAG, MIN_LAG, MAX_LAG, lag);
```

```
      break;

    case '4':
      update_parameter(regs, "offset", REG8_SINE_OFF, MIN_OFF, MAX_OFF,
                       offset);
      break;

    case '5':
      display_settings(regs);
      break;

    case '6':
      cout << "Setting robot to ready mode..." << endl;
      regs.set_reg_b(REG8_MODE, IMODE_READY);
      break;

    case '7': {
      cout << "Setting robot to swim mode..." << endl;
      regs.set_reg_b(REG8_MODE, IMODE_SWIM);

      // Create a filename with timestamp
      time_t now = time(0);
      tm *ltm = localtime(&now);
      string filename = "robot_position_" + to_string(ltm->tm_year + 1900) +
                        "_" + to_string(ltm->tm_mon + 1) + "_" +
                        to_string(ltm->tm_mday) + "_" +
                        to_string(ltm->tm_hour) + "_" + to_string(ltm->tm_min) +
                        "_" + to_string(ltm->tm_sec) + "_freq_" + to_string(freq) + "_amp_" +
                        to_string(amplitude) + "_lag_" + to_string(lag) + "_off_" +
                        to_string(offset) +".csv";

      // Write header to CSV file
      ofstream file(filename);
      if (file.is_open()) {
        file << "Timestamp,X,Y" << endl;
        file.close();
      } else {
        cerr << "Unable to create log file" << endl;
      }

      cout << "Press any key to stop swimming..." << endl;

      bool swimming = true;
      while (swimming) {
        uint32_t frame_time;
        // Gets the current position
        if (!trk.update(frame_time)) {
          cerr << "Error updating tracking data" << endl;
          break;
        }

        double x = 0, y = 0;
        // bool detected = false;

        // Gets the ID of the first spot
        int id = trk.get_first_id();

        // Reads its coordinates (if (id == -1), then no spot is detected)
        if (id != -1 && trk.get_pos(id, x, y)) {
          // detected = true;

          // Get the current time as milliseconds since epoch
          auto now_ms = chrono::duration_cast<chrono::milliseconds>(
              chrono::system_clock::now().time_since_epoch());

          // Log the position to file
          ofstream datafile(filename, ios::app);
          if (datafile.is_open()) {
            datafile << now_ms.count() << "," << fixed << setprecision(3) << x
                     << "," << y << endl;
            datafile.close();
          }

          cout << "Position: (" << fixed << setprecision(3) << x << ", " << y
               << ") m                      \r";
        } else {
          cout << "Position: (not detected)                              \r";
        }
        cout.flush();

        // Check for key press to exit
```

```
        if (kbhit()) {
          swimming = false;
          ext_key(); // Consume the key
        }

        // Small delay to prevent excessive CPU usage
        Sleep(10);
      }

      cout << endl << "Swimming stopped." << endl;
      regs.set_reg_b(REG8_MODE, IMODE_IDLE);
      break;
    }

    case '8': {
      cout << "Starting interactive mode..." << endl;
      regs.set_reg_b(REG8_MODE, IMODE_SWIM);

      cout << "Use keyboard controls:" << endl;
      cout << "  W/S: Increase/decrease speed (frequency)" << endl;
      cout << "  A/D: Turn left/right (offset)" << endl;
      cout << "  Q: Return to menu" << endl;

      bool interactiveMode = true;
      while (interactiveMode) {
        if (kbhit()) {
          DWORD key = ext_key();
          char c = key & 0xFF;

          switch (c) {
          case 'w':
          case 'W':
            freq += 0.1f;
            update_parameter_force(regs, REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ,
                                   freq);
            cout << "Frequency: " << freq << " Hz           \r";
            break;

          case 's':
          case 'S':
            freq = max(0.1f, freq - 0.1f);
            update_parameter_force(regs, REG8_SINE_FREQ, MIN_FREQ, MAX_FREQ,
                                   freq);
            cout << "Frequency: " << freq << " Hz           \r";
            break;

          case 'a':
          case 'A':
            offset -= 0.1f;
            update_parameter_force(regs, REG8_SINE_OFF, MIN_OFF, MAX_OFF,
                                   offset);
            cout << "Offset: " << offset << " degrees       \r";
            break;

          case 'd':
          case 'D':
            offset += 0.1f;
            update_parameter_force(regs, REG8_SINE_OFF, MIN_OFF, MAX_OFF,
                                   offset);
            cout << "Offset: " << offset << " degrees       \r";
            break;

          case 'q':
          case 'Q':
            interactiveMode = false;
            break;
          }
          cout.flush();
        }

        // Update tracking display
        uint32_t frame_time;
        if (trk.update(frame_time)) {
          double x, y;
          int id = trk.get_first_id();
          if (id != -1 && trk.get_pos(id, x, y)) {
            cout << "Position: (" << fixed << setprecision(3) << x << ", " << y
                 << ") m | Freq: " << freq << " Hz | Offset: " << offset
                 << "    \r";
            cout.flush();
          }
```

```
    }

    Sleep(10); // Small delay to prevent excessive CPU usage
   }

   cout << endl << "Interactive mode stopped." << endl;
   regs.set_reg_b(REG8_MODE, IMODE_IDLE);
   break;
  }

  case '0':
   cout << "Stopping robot (idle mode)..." << endl;
   regs.set_reg_b(REG8_MODE, IMODE_IDLE);
   break;

  case 'q':
  case 'Q':
   exitProgram = true;
   cout << "\nExiting program." << endl;
   break;

  default:
   cout << "\nInvalid choice. Please try again." << endl;
   break;
  }
 }

 // Make sure the robot is stopped before exiting
 regs.set_reg_b(REG8_MODE, IMODE_IDLE);

 // Clears the console input buffer
 FlushConsoleInputBuffer(GetStdHandle(STD_INPUT_HANDLE));

 return 0;
}
```

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

```

#include "config.h"
#include "hardware.h"
#include "modes.h"
#include "module.h"
#include "registers.h"
#include "robot.h"

// Define registers for frequency and amplitude control
#define REG8_SINE_FREQ 10 // Register for sine wave frequency
#define REG8_SINE_AMP 11  // Register for sine wave amplitude
#define REG8_SINE_LAG 12  // Register for sine wave lag between elements
#define REG8_SINE_OFF 13  // Register for sine wave offset

// Define limits for frequency and amplitude
#define MAX_FREQ 1.5f // Maximum frequency in Hz
//#define MIN_FREQ 0.0f // Minimum frequency in Hz
#define MAX_AMP 60.0f // Maximum amplitude in degrees
//#define MIN_AMP 0.0f // Minimum amplitude in degrees
#define MAX_LAG 1.5f // Maximum lag between elements in degrees
//#define MIN_LAG 0.0f // Minimum lag between elements in degrees
#define MAX_OFF 3.0f // Maximum lag between elements in degrees (180 mean straight for float conversion)
//#define MIN_OFF -180.0f // Minimum lag between elements in degrees

#define MIN_FREQ 0.1f
#define MIN_AMP 1.0f  // Min amplitude in degrees
#define MIN_LAG 0.5f // Min lag between elements in degrees
#define MIN_OFF -3.0f // Min offset in degrees

// Default values
#define DEFAULT_FREQ 0.8f // Default frequency in Hz
#define DEFAULT_AMP 40.0f // Default amplitude in degrees
#define DEFAULT_LAG 0.75f // Default amplitude in degrees
#define DEFAULT_OFF 0.0f // Default amplitude in degrees


const uint8_t MOTOR_ADDR_HEAD = 25; // Motor address (this is the second element, the first is the head that
const uint8_t MOTOR_ADDR_NECK = 22; // Motor address
const uint8_t MOTOR_ADDR_TORSO = 24; // Motor address
const uint8_t MOTOR_ADDR_HIP = 26; // Motor address
const uint8_t MOTOR_ADDR_TAIL = 5; // Motor address
```

```c

uint8_t freq_enc = DEFAULT_FREQ;
uint8_t amp_enc = DEFAULT_AMP;
uint8_t lag_enc = DEFAULT_LAG;
uint8_t off_enc = DEFAULT_OFF;

static int8_t register_handler(uint8_t operation, uint8_t address,
                               RadioData *radio_data) {

  switch (address) {
    case REG8_MODE:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = reg8_table[REG8_MODE];
        return TRUE;
      case ROP_WRITE_8:
        reg8_table[REG8_MODE] = radio_data->byte; // Allow writing to register
        return TRUE;
      }
    case REG8_SINE_FREQ:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = freq_enc;
        return TRUE;
      case ROP_WRITE_8:
        freq_enc = radio_data->byte; // Allow writing to register
        return TRUE;
      }
    case REG8_SINE_AMP:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = amp_enc;
        return TRUE;
      case ROP_WRITE_8:
        amp_enc = radio_data->byte; // Allow writing to register
        return TRUE;
      }
    case REG8_SINE_LAG:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = lag_enc;
        return TRUE;
      case ROP_WRITE_8:
        lag_enc = radio_data->byte; // Allow writing to register
        return TRUE;
      }
    case REG8_SINE_OFF:
      switch (operation) {
      case ROP_READ_8:
        radio_data->byte = off_enc;
        return TRUE;
      case ROP_WRITE_8:
        off_enc = radio_data->byte; // Allow writing to register
        return TRUE;
      }
  }
  return FALSE;

}

// Function to initialize default parameters
void init_sine_params(void) {
  // Set default values for frequency and amplitude
  freq_enc = ENCODE_PARAM_8(DEFAULT_FREQ, MIN_FREQ, MAX_FREQ);
  amp_enc = ENCODE_PARAM_8(DEFAULT_AMP, MIN_AMP, MAX_AMP);
  lag_enc = ENCODE_PARAM_8(DEFAULT_LAG, MIN_LAG, MAX_LAG);
  off_enc = ENCODE_PARAM_8(DEFAULT_OFF, MIN_OFF, MAX_OFF);
}

void swim_mode(void) {
  uint32_t dt, cycletimer;
  float my_time, delta_t, angle0, angle1, angle2, angle3, angle4;
  float freq, amplitude, lag, offset;
  int8_t angle0_rounded, angle1_rounded, angle2_rounded, angle3_rounded, angle4_rounded;

  // Initialize and start the motor's PID controller
  init_body_module(MOTOR_ADDR_HEAD);
  init_body_module(MOTOR_ADDR_NECK);
  init_body_module(MOTOR_ADDR_TORSO);
  init_body_module(MOTOR_ADDR_HIP);
  init_body_module(MOTOR_ADDR_TAIL);
```

```

  start_pid(MOTOR_ADDR_HEAD);
  start_pid(MOTOR_ADDR_NECK);
  start_pid(MOTOR_ADDR_TORSO);
  start_pid(MOTOR_ADDR_HIP);
  start_pid(MOTOR_ADDR_TAIL);

  set_reg_value_dw(MOTOR_ADDR_HEAD, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_NECK, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_TORSO, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_HIP, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_TAIL, MREG32_LED, 0);

  // Set visual indicator that motor is active
  set_color(4); // Set LED to red

  // Initialize sine wave time
  cycletimer = getSysTICs();
  my_time = 0;

  do {
    // Decode current parameters from registers
    freq = DECODE_PARAM_8(freq_enc, MIN_FREQ, MAX_FREQ);
    amplitude = DECODE_PARAM_8(amp_enc, MIN_AMP, MAX_AMP);
    lag = DECODE_PARAM_8(lag_enc, MIN_LAG, MAX_LAG);
    offset = DECODE_PARAM_8(off_enc, MIN_OFF, MAX_OFF);

    // Apply limits to ensure safety
    if (freq > MAX_FREQ)
      freq = MAX_FREQ;
    if (amplitude > MAX_AMP)
      amplitude = MAX_AMP;
    if (lag > MAX_LAG)
      lag = MAX_LAG;
    if (offset > MAX_OFF)
      offset = MAX_OFF;

    // Calculate elapsed time
    dt = getElapsedSysTICs(cycletimer);
    cycletimer = getSysTICs();
    delta_t = (float)dt / sysTICSperSEC;
    my_time += delta_t;

    // Calculate the sine wave for motor angle
    angle0 = amplitude * sin(M_TWOPI * ((freq * my_time)+(0*lag/5)+offset));
    angle1 = amplitude * sin(M_TWOPI * ((freq * my_time)+(1*lag/5)+offset));
    angle2 = amplitude * sin(M_TWOPI * ((freq * my_time)+(2*lag/5)+offset));
    angle3 = amplitude * sin(M_TWOPI * ((freq * my_time)+(3*lag/5)+offset));
    angle4 = amplitude * sin(M_TWOPI * ((freq * my_time)+(4*lag/5)+offset));

    // Convert angle to motor units
    angle0_rounded = DEG_TO_OUTPUT_BODY(angle0);
    angle1_rounded = DEG_TO_OUTPUT_BODY(angle1);
    angle2_rounded = DEG_TO_OUTPUT_BODY(angle2);
    angle3_rounded = DEG_TO_OUTPUT_BODY(angle3);
    angle4_rounded = DEG_TO_OUTPUT_BODY(angle4);

    // Send the angle to the motor
    bus_set(MOTOR_ADDR_HEAD, MREG_SETPOINT, angle4_rounded);
    bus_set(MOTOR_ADDR_NECK, MREG_SETPOINT, angle3_rounded);
    bus_set(MOTOR_ADDR_TORSO, MREG_SETPOINT, angle2_rounded);
    bus_set(MOTOR_ADDR_HIP, MREG_SETPOINT, angle1_rounded);
    bus_set(MOTOR_ADDR_TAIL, MREG_SETPOINT, angle0_rounded);

    set_rgb(255, 255, 255);

    // Small delay to ensure timer updates properly
    pause(ONE_MS);

  } while (reg8_table[REG8_MODE] == IMODE_SWIM);

  // Clean up: return motor to zero position
  bus_set(MOTOR_ADDR_HEAD, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_NECK, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_TORSO, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_HIP, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_TAIL, MREG_SETPOINT, 0);

  pause(ONE_SEC); // Give the motor time to return to center

  // Stop the motor
```

```
  bus_set(MOTOR_ADDR_HEAD, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_NECK, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_TORSO, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_HIP, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_TAIL, MREG_MODE, MODE_IDLE);

  // Return LED to normal state
  set_color(2);
}

void ready_mode(void) {
  // Initialize and start the motor's PID controller
  init_body_module(MOTOR_ADDR_HEAD);
  init_body_module(MOTOR_ADDR_NECK);
  init_body_module(MOTOR_ADDR_TORSO);
  init_body_module(MOTOR_ADDR_HIP);
  init_body_module(MOTOR_ADDR_TAIL);

  start_pid(MOTOR_ADDR_HEAD);
  start_pid(MOTOR_ADDR_NECK);
  start_pid(MOTOR_ADDR_TORSO);
  start_pid(MOTOR_ADDR_HIP);
  start_pid(MOTOR_ADDR_TAIL);

  set_reg_value_dw(MOTOR_ADDR_HEAD, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_NECK, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_TORSO, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_HIP, MREG32_LED, 0);
  set_reg_value_dw(MOTOR_ADDR_TAIL, MREG32_LED, 0);
  set_rgb(255, 255, 255);


  // Send the angle to the motor
  bus_set(MOTOR_ADDR_HEAD, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(40)); // adopt a rigid S shape to prevent capsi
  bus_set(MOTOR_ADDR_NECK, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(40));
  bus_set(MOTOR_ADDR_TORSO, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-40));
  bus_set(MOTOR_ADDR_HIP, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-40));
  bus_set(MOTOR_ADDR_TAIL, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(40));

  do { // wait until we start swimming or revert to limp mode.
    // Small delay to ensure timer updates properly
    pause(ONE_MS);

  } while (reg8_table[REG8_MODE] == IMODE_READY);

  // Clean up: return motor to zero position
  bus_set(MOTOR_ADDR_HEAD, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_NECK, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_TORSO, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_HIP, MREG_SETPOINT, 0);
  bus_set(MOTOR_ADDR_TAIL, MREG_SETPOINT, 0);

  pause(ONE_SEC); // Give the motor time to return to center

  // Stop the motor
  bus_set(MOTOR_ADDR_HEAD, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_NECK, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_TORSO, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_HIP, MREG_MODE, MODE_IDLE);
  bus_set(MOTOR_ADDR_TAIL, MREG_MODE, MODE_IDLE);

  // Return LED to normal state
  set_color(2);
}


void main_mode_loop(void) {
  // Initialize the default parameters
  init_sine_params();

  // Set initial mode
  reg8_table[REG8_MODE] = IMODE_IDLE;

  // Add the register handler
  radio_add_reg_callback(register_handler);

  while (1) {
    switch (reg8_table[REG8_MODE]) {
    case IMODE_IDLE:
      break;
    case IMODE_READY :
```

```
      ready_mode();
      break;
    case IMODE_SWIM:
      swim_mode();
      break;
    default:
      reg8_table[REG8_MODE] = IMODE_IDLE;
    }
  }
}
```