ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

ROBOTICS PRACTICALS

TOPIC 8

Programming and characterization of a modular fish robot

NGUYEN Thanh Vinh Vincent
BAKKALI Yassine
FÜRST Nelson

March 2022

Contents

1 First program

1.1 Test program

The first program is pretty straightforward. The program first initializes the hardware and activates manual LED control:

```
hardware_init();
reg32_table[REG32_LED] = LED_MANUAL;
```

The LED on the robot's head then gradually changes colors in the following pattern:

1. LEDs turned off (0, 0, 0) to red (127, 0, 0);
2. Red (127, 0, 0) to yellow (127, 127, 0);
3. Yellow (127, 127, 0) to white (127, 127, 127);
4. White (127, 127, 127) to turquoise (0, 127, 127);
5. Turquoise (0, 127, 127) to blue (0, 0, 127);
6. Blue (0, 0, 127) to LEDs turned off (0, 0, 0).

When the pattern reaches its end, it loops back to the first color transition. The transition between each color is done by increments of 1 (maximum value is 255) every 10 ms.

1.2 Blinking LED

To make the LED blink in green at 1 Hz, we simply loop through the following sequence: set the LED to green, wait 500 ms, then turn the LEDs off and wait 500 ms.

```
while (1) {
  set_rgb(0, 127, 0);
  pause(5*HUNDRED_MS);
  set_rgb(0, 0, 0);
  pause(5*HUNDRED_MS);
}
```

2 Registers

When running the program on `ex2.cc`, we get the following result :

```
bakkali@BIOROBPC21 /c/Users/bakkali/Desktop/TP8-github/pc/ex2
$ ex2

Rebooting head module... ok.
get_reg_b(6) = 0
get_reg_b(21) = 66
get_reg_b(21) = 66
get_reg_b(6) = 2
get_reg_b(6) = 0
get_reg_mb(2) = 0 bytes:
get_reg_mb(2) = 8 bytes: 104, 105, 106, 107, 108, 109, 110, 111
get_reg_mb(2) = 8 bytes: 11, 22, 106, 107, 108, 109, 110, 111
get_reg_dw(2) = 2121
get_reg_dw(2) = 8128
```

Let's delve into the inner workings of the code in order to understand these results.

First of all, the program running on the pc calls a function that will print the content of various registers present within the robot which it can access via radio communication. Those functions are `get_reg_b`, `get_reg_dw()`, `get_reg_mb` and which respectively reads the contents of an 8 bits, 32 bits and mutibyte register.

The function `get_reg_b(uint16_t addr)` takes a 16-bit address as a variable and will thus the 8-bit register at said address and either returns its value if the transfer is successful or `0xffff` on failure.

Whenever the function is called, the register callback function on the robot's side will either transfer the value of a counter variable if $addr = 6$ and then clears it. Otherwise, it will increment the counter variable and then return the value `0x42` if `addr` $= 21$.

```cpp
cout << "get_reg_b(6)  = " << (int) regs.get_reg_b(6) << endl;
cout << "get_reg_b(21) = " << (int) regs.get_reg_b(21) << endl;
cout << "get_reg_b(21) = " << (int) regs.get_reg_b(21) << endl;
cout << "get_reg_b(6)  = " << (int) regs.get_reg_b(6) << endl;
cout << "get_reg_b(6)  = " << (int) regs.get_reg_b(6) << endl;
```

Since `counter` is initialized at 0 at the beginning of the code, the first line above displays 0 on the terminal. The two following lines incremented the counter twice and thus resulted in the 4th line outputting 2. Then `counter` gets cleared and the last call to the address 6 gives us 0.

The function `get_reg_mb(uint16_t addr, uint8_t* data, uint8_t& len)` reads a multibyte register at the starting address `addr` and returns the length of the register to the `len` register and its content to the data buffer`data`. The sequence of bytes is then read in the `display_multibyte_register` function which first prints out the length of the register and then its content. Similarly, the function `set_reg_mb` will send a data buffer, its size and an address in order to store it within the robot.

On the robot's side, when attempting to read a multibyte register at the address 2, the size of the last registered multibyte register and its content will be transfered one after another. Furthermore, writing into a multibyte register at the address 2 will update its size and store the data with an offset of 4.

And finally, the function `set_reg_b(uint16_t addr, uint8_t val)` will update the multibyte buffer at address `[addr - 2]` with `val`.

```
1   cout << "get_reg_mb(2) = ";
2   display_multibyte_register(regs, 2);
3
4   uint8_t buffer[8];
5   for (int i(0); i < 8; i++) {
6   buffer[i] = 100 + i;
7   }
8   regs.set_reg_mb(2, buffer, sizeof(buffer));
9
10  cout << "get_reg_mb(2) = ";
11  display_multibyte_register(regs, 2);
12
13  regs.set_reg_b(2, 11);
14  regs.set_reg_b(3, 22);
15
16  cout << "get_reg_mb(2) = ";
17  display_multibyte_register(regs, 2);
```

As we can see above, we first display the content of a multibyte register at address 2, which will naturally print out a register of size 0 and no content since nothing has been initialized yet. We then set the register with a buffer containing numbers ranging from 100 to 107 at the same address. When trying to print out the content of the register, we have to account for the (+4) shift on the buffer and thus expect to read a register of size 8 with values ranging from 104 to 111.

Lines 13 and 14 will update the multibyte buffer's first two values from 104 and 105 to respectively 11 and 22. And thus when displaying the register anew, we will obtain the following values : `11, 22, 106, 107, 108, 109, 110, 111`.

The function `set_reg_w(const uint16_t addr, const uint16_t val)`, when given the address 7, will update the content of the register `datavar` on the robot to the value $(\texttt{datavar} \cdot 3) + (\texttt{radio\_data->word})$ with `radio_data->word` being the last 16-bit value read via radio communication.

Moreover, the function `get_reg_dw(const uint16_t addr)` sends the value of `datavar` to the pc.

```
regs.set_reg_w(7, 2121);
cout << "get_reg_dw(2) = " << regs.get_reg_dw(2) << endl;
regs.set_reg_w(7, 1765);
cout << "get_reg_dw(2) = " << regs.get_reg_dw(2) << endl;
```

The program on the computer side ends with those lines. The register `datavar` is initially at 0 in the robot's program. We first set its value at $\texttt{datavar} = (0 * 3) + 2121 = 2121$ and then print it as such. Following that we set it at $\texttt{datavar} = (2121 * 3) + 1765 = 8128$ and finally print its value.

3 Modules communication

The first program reads the angle of the fish tail, and changes the color of the LED on the robot's head accordingly.

The program was then modified to constantly read the different joint angles from the robot and print them out in the terminal. In the robot, after having initialized the hardware and added the register handler as a callback function, the program initialize each body and limb module separately.

```
for(uint8_t i = 0; i < BODY_NUMBER; ++i ){
    init_body_module(MOTOR_ADDR[i]);
    for(uint8_t j = 0; j <= LIMB_NUMBER[i]; ++j){
        init_limb_module(MOTOR_ADDR[i] + j);
    }
}
```

The position of each body and limb is stored in one static `int8_t` 2D array: `pos`. Each module has its own column in the array, and every row corresponds to a DOF that the body can control. Body modules that only have access to the body DOF will only use the first row of the position array, where limb modules that also have access to the fins DOF, use 3 rows in the array. This array is constantly updated in the robot main function with the function:

```
pos[j][i] = bus_get(MOTOR_ADDR[i]+j, MREG_POSITION);
```

Two constant arrays are used to describe the structure of the robot.
LIMB_NUMBER is used to indicate how many additional limbs each module of the robot has access to. (With the boxfish robot, possible values are 1 for a body module, and 3 for a limb module).
MOTOR_ADDR gives the address of each body elements.

On the computer side, just as in exercice2, the function `get_reg_mb(uint16_t addr, uint8_t* data, uint8_t& len)` is used to get the position of the limbs of the robot from a multibyte register that contains the data for all the bodies and limbs angles. The program then prints out the values in the terminal.
The program closes when a key is pressed thanks to the `kbhit()` function.

4 Position control of a module

The goal of this exercise is to continually send setpoint angles to the robot to generate a sinusoidal motion on the robot.

4.1 Computer side

First, we define two functions to turn start/stop the robot. This is done by setting the REG8_MODE register on the robot to 1 and 0 respectively.

```
#define MODE_ON  1
#define MODE_OFF 0

void start_robot(CRemoteRegs &regs) {
  regs.set_reg_b(REG8_MODE, MODE_ON);
}

void stop_robot(CRemoteRegs &regs) {
  regs.set_reg_b(REG8_MODE, MODE_OFF);
}
```

To be able to set the position of the motor, we also define a new mode which can be activated by setting the `REG8_MODE` register to 2. This mode is used to continually read the setpoint value sent over the radio (see robot side).

```
#define IMODE_MOTOR_SETPOINT 2

void start_read_setpoint(CRemoteRegs &regs){
  regs.set_reg_b(REG8_MODE, IMODE_MOTOR_SETPOINT);
}
```

Finally, we send the setpoint as an unsigned 8-bit integer (which will be reinterpreted as a signed value on the robot side) at the address defined by `MREG_SETPOINT`.

```
#define MREG_SETPOINT 0x2F

void send_setpoint(uint8_t setpoint_deg, CRemoteRegs &regs) {
  regs.set_reg_b(MREG_SETPOINT, setpoint_deg);
}
```

To send the sinusoidal sequence of setpoints, we simply calculate the position at time t and send it to the robot. The program can be interrupted by pressing any key.

```
#define FREQUENCY 1
#define AMP_DEG   40

start_read_setpoint(regs);
while(!kbhit()){
  setpoint = (int8_t)AMP_DEG*sin(2*M_PI*FREQUENCY*time_d());
  cout << "setpoint is : " << (int) setpoint << "       \r";
  send_setpoint((uint8_t)setpoint, regs);
}
stop_robot(regs);
regs.close();
```

4.2 Robot side

On the robot side, the `main.c` file has not been modified. In the `modes.c` file, the register handler handles the following address cases: `REG8_MODE` which is used to start/stop the robot and `MREG_SETPOINT` which is used to retrieve the setpoint value sent over the radio. In our case, all values sent over the radio are 8-bit (`operation = ROP_WRITE_8`).

```c
static int8_t pos = 0;
static int8_t register_handler(uint8_t operation, uint8_t addr, RadioData* radio_data)
{
    if (operation == ROP_WRITE_8){
      switch(addr) {
        case REG8_MODE:
          reg8_table[REG8_MODE] = radio_data->byte;
          return TRUE;
        case MREG_SETPOINT:
          pos = (int8_t)radio_data->byte;
          return TRUE;
      }
    }
  return FALSE;
}
```

We have also defined a new mode which is used to initialize and set the position of the motor to the desired setpoint. This mode is enabled when the `REG8_MODE` register is set to `IMODE_MOTOR_SETPOINT`.

```c
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_MOTOR_SETPOINT:
        read_setpoint_mode();
        break;
      default:
        reg8_table[REG8_MODE] = IMODE_IDLE;
    }
  }
}
```

This mode first initializes the body module and starts its PID controller. It then sets the motor position to the desired position as long as the user hasn't requested to exit the mode (`reg8_table[REG8_MODE] == IMODE_MOTOR_SETPOINT`). When exiting the mode, we reset the motor position and set the mode back to `MODE_IDLE`.

```c
const uint8_t MOTOR_ADDR = 21;

static void read_setpoint_mode(void)
{
  init_body_module(MOTOR_ADDR);
  start_pid(MOTOR_ADDR);
  while(reg8_table[REG8_MODE] == IMODE_MOTOR_SETPOINT) {
    bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(pos));
  }
  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);
}
```

5 Trajectory generation

5.1 Simple on-board trajectory generation

The complete code with variable amplitude/frequencies will be explained in the next section. When generating the sinusoidal sequence of positions directly on the robot, the motion seemed to be a lot smoother. This can be explained by the fact that on the robot, we increment the current time by `delta_t` which is calculated at the end of every iteration of the loop. This means that the next position of the robot is calculated as soon as it is ready to move.

On the computer side however, we send the setpoints without taking into account the execution time on the robot. This means that if we send the positions at a rate that is too fast for the robot to execute, it could lead to a rougher result on the robot side.

5.2 Modulating trajectory parameters

5.2.1 Computer side

We first define two functions to send the desired frequency and amplitude to the robot. These two values are float variables encoded to 8-bit values using the `ENCODE_PARAM_8` function. To ensure that we do not send the values out of the desired limits, we cap maximum and minimum values that can be sent.

```
void send_amplitude(CRemoteRegs &regs, float amplitude){
  if (amplitude < AMPLITUDE_MIN){
    amplitude = AMPLITUDE_MIN;
  } else if (amplitude > AMPLITUDE_MAX){
    amplitude = AMPLITUDE_MAX;
  }
  regs.set_reg_b(REG_AMP, ENCODE_PARAM_8(amplitude,AMPLITUDE_MIN,AMPLITUDE_MAX));
}

void send_freq(CRemoteRegs &regs, float frequency){
  if (frequency < FREQ_MIN){
    frequency = FREQ_MIN;
  } else if (frequency > FREQ_MAX){
    frequency = FREQ_MAX;
  }
  regs.set_reg_b(REG_FREQ, ENCODE_PARAM_8(frequency,FREQ_MIN,FREQ_MAX));
}
```

The register address and min/max values are defined in `regdefs.h`:

```
// Sine amplitude register address
#define REG_AMP    1

// Sine frequency register address
#define REG_FREQ   2

#define AMPLITUDE_MAX 60.0f
#define AMPLITUDE_MIN 0.0f

#define FREQ_MAX 2.0f
#define FREQ_MIN 0.0f
```

We also define two modes: `IMODE_IDLE` and `IMODE_SINE_DEMO` which can be enabled by changing the value of the `REG8_MODE` register.

```
void start_sine_demo(CRemoteRegs &regs){
  regs.set_reg_b(REG8_MODE, IMODE_SINE_DEMO);
}

void stop_robot(CRemoteRegs &regs){
  regs.set_reg_b(REG8_MODE, IMODE_IDLE);
}
```

In the main function, we request the robot to go into the `IMODE_SINE_DEMO` mode and then go into the main loop. In this loop, the user can press the `A`, `F` and `SPACE` keys to set the amplitude, frequency and exit the program respectively.

```
float amplitude = 0;
float frequency = 0;
DWORD key = 0;

// Reboots the head microcontroller to make sure it is always in the same state
reboot_head(regs);
start_sine_demo(regs);

cout << "Press A to change the amplitude [0 to 60] of the sine,"
"F for the frequency [0 to 2] and SPACE to exit the program" << endl;
while(key != SPACE_KEYCODE){
  if( key == A_KEYCODE){
    cout << "Please enter an amplitude : " << endl;
    cin >> amplitude;
    send_amplitude(regs, amplitude);
  }
  if( key == F_KEYCODE){
    cout << endl << "Please enter a frequency :" << endl;
    cin >> frequency;
    send_freq(regs, frequency);
  }
  key = ext_key();
}

stop_robot(regs);
regs.close();
```

The keycodes are defined in `regdefs.h` and were found by printing the value returned by the `ext_key()` function.

5.2.2 Robot side

We first define some default values for the frequency and amplitude of the sinusoidal motion:

```
static uint8_t amplitude = ENCODE_PARAM_8(20, AMPLITUDE_MIN, AMPLITUDE_MAX); // 20 deg
static uint8_t frequency = ENCODE_PARAM_8(1, FREQ_MAX, FREQ_MIN); // 1 Hz
```

In the register handler, we set the set the amplitude and frequencies that are read from the radio. We also use it to change the REG8_MODE register, which defines the mode of operation. When changing the value of the amplitude or the frequency, we reset the motor position to its neutral position in order to avoid too much deviation.

```c
static int8_t register_handler(uint8_t operation, uint8_t addr, RadioData* radio_data)
{
  if (operation == ROP_WRITE_8){
    switch(addr) {
      case REG8_MODE:
        reg8_table[REG8_MODE] = radio_data->byte;
        return TRUE;
      case REG_AMP:
        amplitude = radio_data->byte;
        bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
        return TRUE;
      case REG_FREQ:
        frequency = radio_data->byte;
        bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
        return TRUE;
    }
  }
  return FALSE;
}
```

The sine_demo_mode() function is called when the REG8_MODE register is set to the value of IMODE_SINE_DEMO. In this function, we first initialize the variables that will be used to keep track of the current time and also initialize the body module and its PID controller.

```c
uint32_t dt, cycletimer;
float my_time, delta_t, l;
int8_t l_rounded;

cycletimer = getSysTICs();
my_time = 0;

init_body_module(MOTOR_ADDR);
start_pid(MOTOR_ADDR);
```

We then go into our mode's loop, from which we can exit if the value of the REG8_MODE register is no longer equal to IMODE_SINE_DEMO. Inside the body of that loop, the current time is first calculated. This is done by measuring the number of elapsed system tics during the execution of the loop and incrementing the current time with this number of tics converted back to seconds.

```c
dt = getElapsedSysTICs(cycletimer);
cycletimer = getSysTICs();
delta_t = (float) dt / sysTICSperSEC;
my_time += delta_t;
```

Then, the encoded `uint8_t` values that were sent over the radio (amplitude, frequency) are decoded back to `float` values, which are then used to calculate the current value of the sinusoidal motion. On the robot side, we also make sure to stay within the desired bounds for the amplitude and frequency.

```c
float f_amplitude = DECODE_PARAM_8(amplitude ,AMPLITUDE_MIN, AMPLITUDE_MAX);
float f_freq = DECODE_PARAM_8(frequency ,FREQ_MIN, FREQ_MAX);

if(f_freq > FREQ_MAX){
  f_freq = FREQ_MAX;
} else if (f_freq < FREQ_MIN){
  f_freq = FREQ_MIN;
}
if(f_amplitude > AMPLITUDE_MAX){
  f_amplitude = AMPLITUDE_MAX;
} else if (f_amplitude < AMPLITUDE_MIN){
  f_amplitude = AMPLITUDE_MIN;
}

// Calculates the sine wave
l = f_amplitude * sin(M_TWOPI * f_freq * my_time);
l_rounded = (int8_t) l;

// Outputs the sine wave to the motor
bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded));

// Make sure there is some delay, so that the timer output is not zero
pause(ONE_MS);
```

When exiting the loop, we set the motor back to its neutral position and go back into idle mode:

```c
bus_set(MOTOR_ADDR, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
pause(ONE_SEC);
bus_set(MOTOR_ADDR, MREG_MODE, MODE_IDLE);
```

6 LED tracking system

6.1 Combining the tracking with the radio

In this chapter, we are tasked to modulate the color of the boxfish's head LED with relation to its position within the pool. The program we wrote is pretty straightforward as all calculations are proceeded on the pc's side.

First of all, we connect the computer with the tracking system just as it was demonstrated on the base code which can be found on moodle. The function `trk.get_pos(id, x, y)` will then update the positions x and y of the robot as long as we require it. After successfully measuring the position, we compute the color of the led given its position with the following functions:

```
static uint32_t calculate_rgb_from_channels(uint8_t r, uint8_t g, uint8_t b){
  return ((uint32_t) r << 16) | ((uint32_t) g << 8) | b;
}

static uint32_t calculate_rgb_from_pos(CRemoteRegs &regs, double x, double y) {
  uint32_t rgb = 0;
  uint8_t r, g, b = 0;
  if (x < 0||x > POOL_LENGTH || y < 0 || y > POOL_WIDTH) {
    return calculate_rgb_from_channels(0, MAX_CHANNEL_VALUE, 0);
  }
  r = (uint8_t) (x/POOL_LENGTH * MAX_CHANNEL_VALUE);
  g = GREEN_VALUE;
  b = (uint8_t) (y/POOL_WIDTH * MAX_CHANNEL_VALUE);
  rgb = calculate_rgb_from_channels(r, g, b);
  cout << "x = " << x << ", y = " << y << ", R = " << (int) r << ", B = " << (int) b
       << ", RGB 0x" << hex << rgb << "    " << '\r' << flush;
  return rgb;
}
```

Since the color follows an 8-8-8 RGB color format, we first compute each color within a range of 0 to 255 and then apply bit shifting operations in order to merge them into an `uint32_t` format within the `rgb` register.

```
static void set_led_color(CRemoteRegs &regs, uint32_t rgb) {
  regs.set_reg_dw(REG_LED_COLOR, rgb);
}
```

We can then send the value of `rgb` to the address 0 which will change the value of the head module's LED.

7 Swimming and experiments

7.1 Writing a swimming trajectory generator

Our swimming trajectory generator was inspired by the work of Benjamin Frankhauser who wrote a semester project about designing, simulating and testing the boxfish robot. At first, we applied sinusoidal signals to each limb following the steady state result of the following graph :

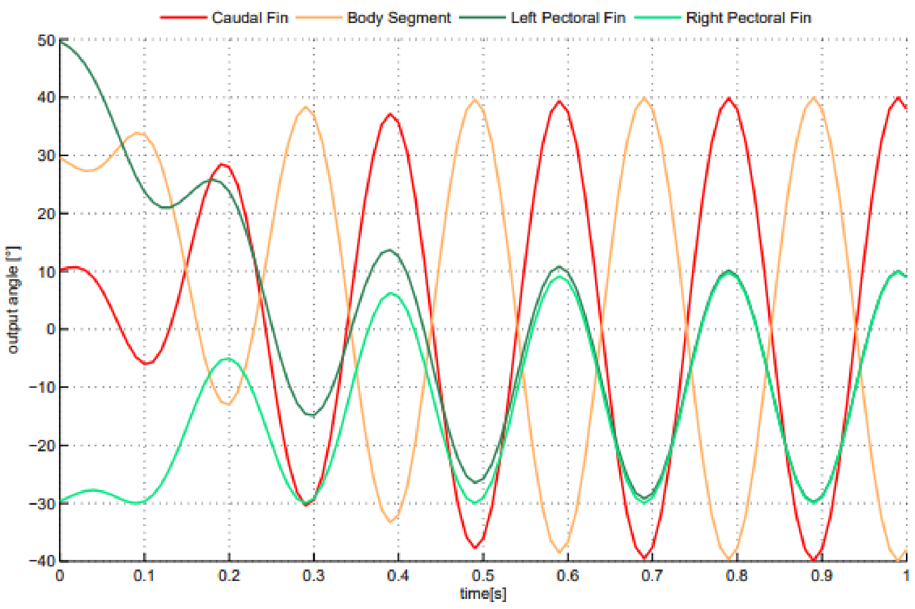


Figure 1: Desired frequency, amplitude and phase achieved by B.Frankhauser we have first tried to mimic [1].

In our case, the fins and body segment are in phase with each other while the caudal fin is not dephased. We chose to synchronize the sinusoidal trajectories since the robot remained stationary with a phase of π. The integration of our problem is similar to the code presented in subsection [5.2] with the principal difference being that we know have to initialize all body modules before moving the robot and, obviously, turn them all off once the robot finished its task.

7.1.1 Computer side

In addition to function designed to send the frequency and amplitude to the robot, we have also added one that sends the phase difference between the body and caudal modules. The phase is also sent a 8-bit encoded float variable with its minimum and maximum values capped at respectively 0 and π.

```
void send_phi(CRemoteRegs &regs, float phi) {
  if (phi < PHI_MIN){
    phi = PHI_MIN;
  } else if (phi > PHI_MAX){
    phi = PHI_MAX;
  }
  regs.set_reg_b(REG_PHI, ENCODE_PARAM_8(phi, PHI_MIN, PHI_MAX));
}
```

A new mode have also been defined in addition to `IMODE_IDLE`: `IMODE_GO_FORWARD` which, again, can be enabled by changing the value of the `REG8_MODE` register. In the main loop, we have added the ability to command the robot to move forward by the press of the `UPWARD ARROW` key or stop the robot with the `S` key.

```
void go_forward(CRemoteRegs &regs){
  cout << "Going forward" << endl;
  regs.set_reg_b(REG8_MODE, IMODE_GO_FORWARD);
}

void stop_robot(CRemoteRegs &regs){
  cout << "Stop robot" << endl;
  regs.set_reg_b(REG8_MODE, IMODE_IDLE);
}
```

7.1.2 Robot side

The `move_mode()` function is now called depending on the mode the robot is called with. The `IMODE_MOVE_FORWARD` mode will be treated in this section while the steering modes `IMODE_TURN_RIGHT` and `IMODE_TURN_LEFT` will be treated in section [8].

Within this function, we first initialize all variables body modules, limbs and PID controllers :

```
uint32_t dt, cycletimer;
float my_time, delta_t, l, l_offset;
int8_t l_rounded, l_offset_rounded;

cycletimer = getSysTICs();
my_time = 0;

```

```c
// Initialises the body module with the specified address (but do not start
// the PD controller)
uint8_t i = 0;
uint8_t j = 0;
for(i = 0; i < BODY_NUMBER; ++i ){
  init_body_module(MOTOR_ADDR[i]);
  for(j = 0; j < LIMB_NUMBER[i]; ++j){
    init_limb_module(MOTOR_ADDR[i] + j);
    start_pid(MOTOR_ADDR[i] + j);
    pause(HALF_SEC);
  }
}
```

Then, we enter the main loop which we can exit only if the value of the REG8_MODE register is equal to MODE_IDLE. Just like before, we continuously calculate the current elapsed time in the loop and then receive the values sent over the radio of the amplitude, frequency and phase which will be decoded back to float values.

We then calculate the sinusoidal trajectories for all modules.

```c
float f_amplitude = DECODE_PARAM_8(amplitude ,AMPLITUDE_MIN, AMPLITUDE_MAX);
float f_freq = DECODE_PARAM_8(frequency ,FREQ_MIN, FREQ_MAX);
float f_phi = DECODE_PARAM_8(phi, PHI_MIN, PHI_MAX);

// Amplitude and frequency are capped
if(f_freq > FREQ_MAX){
  f_freq = FREQ_MAX;
} else if (f_freq < FREQ_MIN){
  f_freq = FREQ_MIN;
}

if(f_amplitude > AMPLITUDE_MAX){
  f_amplitude = AMPLITUDE_MAX;
} else if (f_amplitude < AMPLITUDE_MIN){
  f_amplitude = AMPLITUDE_MIN;
}

if(f_phi > PHI_MAX){
  f_phi = PHI_MAX;
} else if (f_phi < PHI_MIN){
  f_phi = PHI_MIN;
}

// Calculates the sine waves
l = f_amplitude * sin(M_TWOPI * f_freq * my_time);
l_rounded = (int8_t) l;
```

```

l_offset = f_amplitude * sin(M_TWOPI * f_freq * my_time + f_phi);
l_offset_rounded = (int8_t) l_offset;
```

Two different trajectories are computed in our case : `l_rounded` corresponds to the motion of the caudal module while `l_offset` has the same amplitude and frequency as `l_rounded` but can be provided a phase difference. The amplitude on the fins were reduced using the `SIDE_FIN_AMP_RATIO` = 2 coefficient in accordance to our experimentations and Frankhauser's report [[1]].

```
switch (mode) {
  case FORWARD:
    // left fin
    bus_set(MOTOR_ADDR[0] + 1, MREG_SETPOINT,
            DEG_TO_OUTPUT_BODY(l_rounded/SIDE_FIN_AMP_RATIO));
    // right fin
    bus_set(MOTOR_ADDR[0] + 2, MREG_SETPOINT,
            DEG_TO_OUTPUT_BODY(l_rounded/SIDE_FIN_AMP_RATIO));
    // caudal fin
    bus_set(MOTOR_ADDR[1], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded));
    // body module
    bus_set(MOTOR_ADDR[0], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_offset_rounded));
    break;
```

When exiting the motors are set back to their neutral positions and we return to the idle mode:

```
for (i = 0; i < BODY_NUMBER; ++i) {
  for (j = 0; j < LIMB_NUMBER[i]; ++j) {
    bus_set(MOTOR_ADDR[i] + j, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
    pause(HALF_SEC);
  }
}
for (i = 0; i < BODY_NUMBER; ++i) {
  for (j = 0; j < LIMB_NUMBER[i]; ++j) {
    bus_set(MOTOR_ADDR[i] + j, MREG_MODE, MODE_IDLE);
    pause(HALF_SEC);
  }
}
```

7.2 Experiments

The following tests were made by calculating the distance of the robot from its initial position and extracting its speed after it was 1 meter far from it. We chose such a short distance since the radio communication frequently interrupted in between measures for higher distances and the robot deviated too much from a straight trajectory which would give us biased data.

Frequency [Hz]	**Average speed** [m/s]	**Standard deviation** [m/s]
0.6	0.0644	0.0054
0.8	0.0785	0.0039
1.0	0.0967	0.0053
1.2	0.0966	0.0039
1.4	0.1063	0.0030

Table 1: Average speed and standard deviation w.r.t frequency.

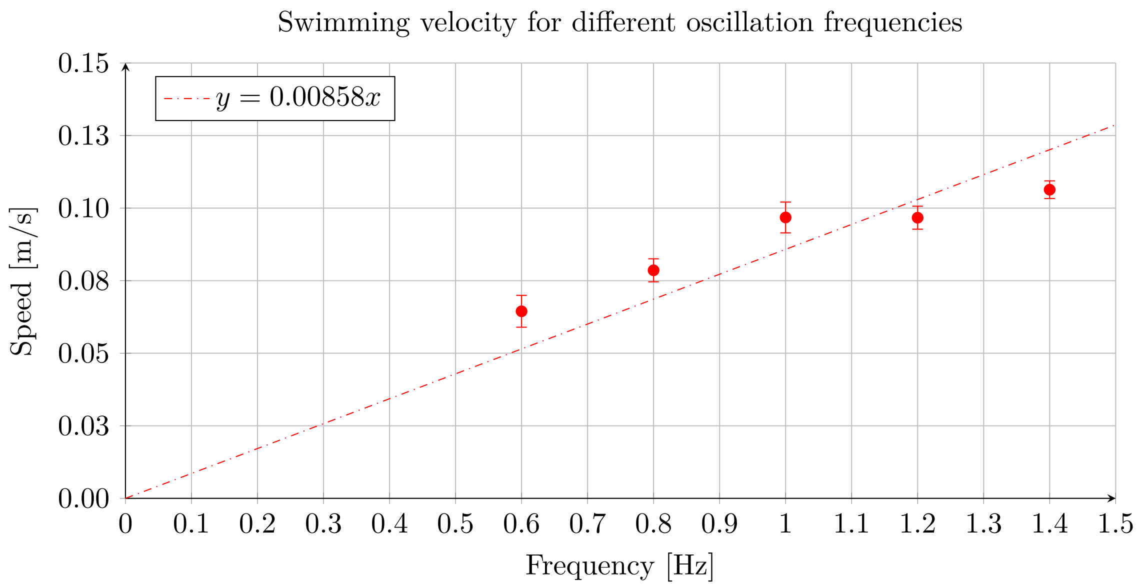
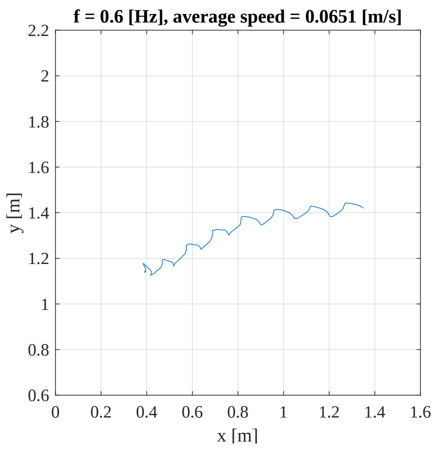
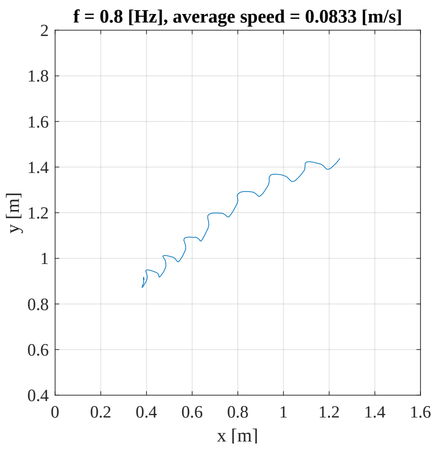


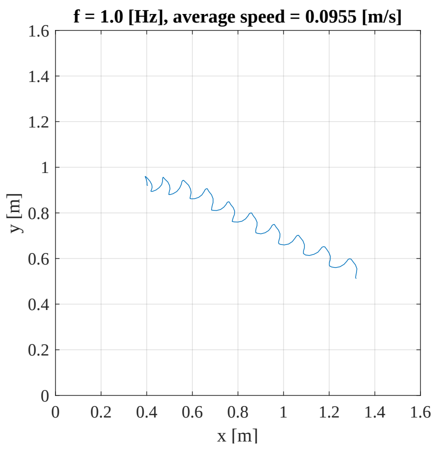
Figure 2: Influence of frequency on velocity.



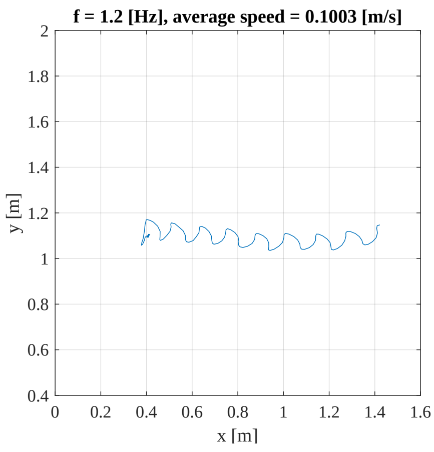
(a) Oscillation frequency = 0.6 Hz.



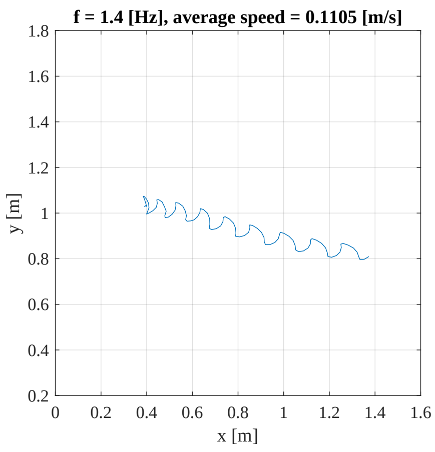
(b) Oscillation frequency = 0.8 Hz.



(c) Oscillation frequency = 1.0 Hz.



(d) Oscillation frequency = 1.2 Hz.



(e) Oscillation frequency = 1.4 Hz.

Figure 3: Example of trajectories for different oscillation frequencies.

From figure 2, it can be seen that the swimming velocity increases more or less linearly with the oscillation frequency ($R^2 = 0.9847$). It should be noted however that for some of the measurements, the trajectory was not completely straight (especially for lower oscillation frequencies, such as 0.6 Hz, which could lead to unexpected results). A way to mitigate this would have been to get a timestamp for every position, which would allow us to calculate the speed at each timestep.

8 More experiments

8.1 Steering

8.1.1 Computer side

We implemented the ability to steer the robot left and right by pressing their respective keys on the keyboard (left and right arrows). However, it should be noted that one must stop the robot (press S key) before changing directions. In our main loop, we have then defined two extra cases:

```
if (key == RIGHT_KEYCODE) {
  turn_right(regs);
}
if (key == LEFT_KEYCODE) {
  turn_left(regs);
}
```

Pressing the corresponding keys call the following functions, which set the `REG8_MODE` register to the `IMODE_TURN_LEFT` and `IMODE_TURN_RIGHT` modes.

```
void turn_left(CRemoteRegs &regs){
  cout << "Turning left" << endl;
  regs.set_reg_b(REG8_MODE, IMODE_TURN_LEFT);
}

void turn_right(CRemoteRegs &regs){
  cout << "Turning right" << endl;
  regs.set_reg_b(REG8_MODE, IMODE_TURN_RIGHT);
}
```

8.1.2 Robot side

In order to steer in one direction, we put the fin in the desired direction (e.g. left fin to turn left) at its neutral position, and make it passive. We also assign a fixed steering angle between the 2 body modules (±20°, depending on the steering direction). The moving fin and the tail oscillates with the same frequency and amplitude as in the forward mode.

```
switch (mode) {
    ... // case forward
  case LEFT:
    // left fin as passive
    bus_set(MOTOR_ADDR[0] + 1, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
    bus_set(MOTOR_ADDR[0] + 1, MREG_MODE, MODE_IDLE);
    // right fin
    bus_set(MOTOR_ADDR[0] + 2, MREG_SETPOINT,
            DEG_TO_OUTPUT_BODY(l_rounded/SIDE_FIN_AMP_RATIO));
    // caudal fin
    bus_set(MOTOR_ADDR[1], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded));
    // body module
    bus_set(MOTOR_ADDR[0], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(-STEERING_ANGLE));
    break;
  case RIGHT:
    // right fin as passive
    bus_set(MOTOR_ADDR[0] + 2, MREG_SETPOINT, DEG_TO_OUTPUT_BODY(0.0));
    bus_set(MOTOR_ADDR[0] + 2, MREG_MODE, MODE_IDLE);
    // left fin
    bus_set(MOTOR_ADDR[0] + 1, MREG_SETPOINT,
            DEG_TO_OUTPUT_BODY(l_rounded/SIDE_FIN_AMP_RATIO));
    bus_set(MOTOR_ADDR[1], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(l_rounded));
    bus_set(MOTOR_ADDR[0], MREG_SETPOINT, DEG_TO_OUTPUT_BODY(STEERING_ANGLE));
    break;
  }
}
```

8.2 Changing phase

We have also implemented a way to change the phase difference between fins and the body segment. It was observed, however, that the performance was best for a phase difference between 0 and $\pi/2$. When taking values closer to π, the swimming speed kept decreasing until it eventually started to go backwards. However, we were not able to make a detailed analysis of the influence of that parameter on the swimming speed because the robot's battery was running low and we had issues with water leakage.

And then it drowned.

Figure 4: Picture of the boxfish after it drowned. It did not taste good [2].

8.2.1 Computer side

As usual, we add an extra case in our main loop.

```
if (key == P_KEYCODE) {
  cout << endl << "Please enter a phase :" << endl;
  cin >> phi;
  send_phi(regs, phi);
}
```

Pressing ’P’ then calls the following function which sends the value of the desired phase to the robot.

```
void send_phi(CRemoteRegs &regs, float phi) {
  if (phi < PHI_MIN){
    phi = PHI_MIN;
  } else if (phi > PHI_MAX){
    phi = PHI_MAX;
  }
  regs.set_reg_b(REG_PHI, ENCODE_PARAM_8(phi, PHI_MIN, PHI_MAX));
}
```

8.2.2 Robot side

The phase is read and decoded just like the frequency and amplitude. It is then used in the calculation of the sinusoidal motion, as can be seen in section 7.1.2.

References

[1] B. Frankhauser. “BoxyBot II, the fish robot: Fin Design, Programmation, Simulation and Testing”. Master’s thesis. EPFL, 2010, p. 12.

[2] Dana Hatic. *The Chicken Wings of the Sea.* URL: https://www.eater.com/2019/4/22/18507808/blowfish-tails-puffer-fish-fugu-sugar-toads-restaurant-trend.