Introduction

The aim of this practical work is to program and test a *boxfish* or *lamprey* robot (depending on the week) made using the *Salamandra robotica II* modules. These elements have been designed to be completely independent (each module contains its own battery, motor, and local controller, and is individually waterproof) and can be assembled in several ways to create different types of robots, as shown in Figure 1.

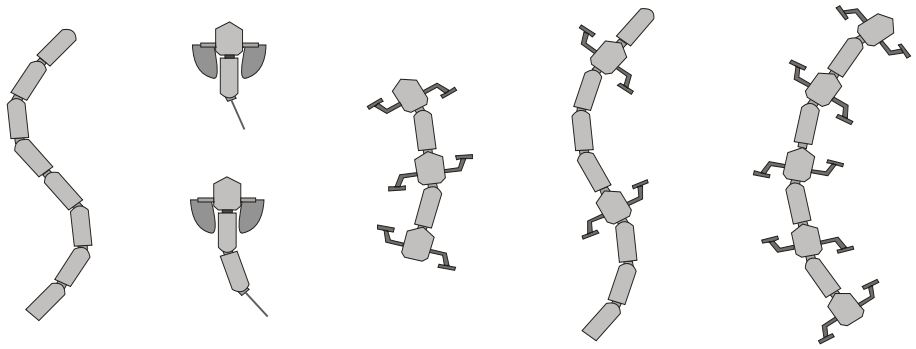

Figure 1: Examples of robots that could be built using the modules.

There are two types of active modules: *body* and *limb*. Body modules have a single degree of freedom (see Figure 2), whereas limb modules have three DOF (one of which is the same of the body modules; the two others are rotating limbs, that could be for example legs or fins). The boxfish robot uses both types of modules, while the lamprey robot only uses body modules.

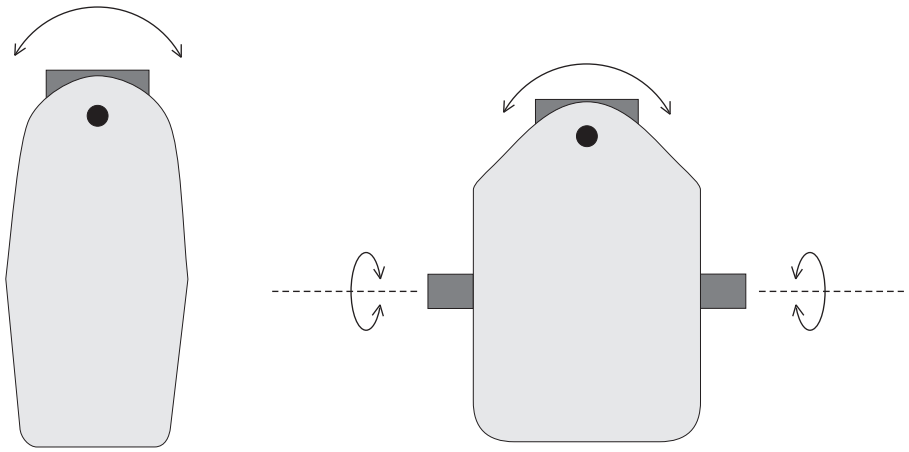

Figure 2: Body (left) and limb (right) elements, seen from the top.

In addition to these two module types, a third one is generally used: a head. It has the same shape of a body element, but contains different electronics and has no moving mechanical parts. The modules are made out of three parts: a body and two covers, which are fixed together with four couples of magnets. Greased O-rings placed in-between ensure the waterproofing. **Please do not open the modules**, as good waterproofing requires careful cleaning and greasing before closing the covers again. If a strong blue light (and/or a red and blue blinking light) appears while using the modules, this means that a water leakage has been detected: please disconnect the charger if it is connected, then immediately report the problem to the assistant and do not operate the modules anymore to avoid any damages.

The output axis of each module has a connection piece that can be screwed to the rear of the next module using four screws. This piece contains six contacts, which are used to connect electrically the different modules of the robot (power supply for battery charging, CAN bus, power on/off switch control).

Depending on the electronics inside them, modules can be either on or off by default: those of the boxfish robot are **off** by default, while those of the lamprey robot are on by default—this also means that the modules of the two robots cannot be mixed. The top cover of the head module has a slot in which you can place a small magnet. When the magnet is in place, the head and all the connected modules will turn on (boxfish) or off (lamprey). Please remember that **it is very easy to lose the magnet**. To prevent the batteries from discharging, **keep the robot off while you are not working with it** (or connect it to a power source, please ask your assistant to show you how).

Some background info about the robot control board: the head element contains a NXP LPC2129 microcontroller. It has a 32-bit ARM7TDMI core running at 60 MHz. When writing programs that have to run on it, keep in mind that it has only 16 kB of RAM and 256 kB of flash (program) memory, part of which are already used by the firmware framework in which you will put your own program. You should therefore optimize your code and the memory usage as much as possible. Dynamic memory allocation (i.e., `malloc` or `new`) is not available. The main microcontroller is connected through a serial port (UART) to a small PIC microcontroller, which is in turn connected by SPI to an 868 MHz radio transceiver. The PIC implements the radio communication protocol (allowing a computer to read data from the ARM or to send data to it), and is able to reprogram the ARM using its integrated bootloader.

The radio interface you will need to control the robot is already connected to the computer you will use. If you have any troubles while communicating with the robot, check the connections of the interface, and try to reset it (either by disconnecting and reconnecting it, or by cycling the power, depending on which interface is available on the PC you use).

Time and report

Exercises 1 to 5.2 are supposed to be done during the first session, and the rest the following one. Please make sure that you prepare the exercises *before* the sessions, otherwise it will be very difficult to finish within the allocated time. In particular, exercise 2 can be done almost entirely by *reading* and *understanding* the source code. You should also arrive at the second session with an idea of how to make the robot swim (see Section 7.1), **otherwise it will be impossible to finish the experiments in time**.

Please write a complete report, documenting all your work, e.g., how the programs you wrote are working, or how you are taking speed measures (and not only answering explicit questions!). No need to print out long excerpts of the code you received; please rather include quotations of the code you *wrote*. If you include code, **please insert it as text**, not as screenshots, and **do not use dark backgrounds**. Remember that the quality of the report is part of the evaluation!

Remember to explicitly indicate the units on the axes of your plots (very common problem). Remember to **justify any choices you made**. Also ensure that what you did and what you took elsewhere (e.g., source code, images) could be easily distinguished.

Do not forget to keep all the programs you write (i.e., don't overwrite them to make new ones), as you will have to submit all source codes with your report (in compilable form: do not remove Makefiles or the source files you did not modify). Object code and dependency files are not needed and can use quite some disk space; to remove them, you can use `make clean` *in each of your directories before creating the ZIP file to hand in.*

Source files

There are two main directories in the source files you will download: `pc` and `robot`. The first one contains all the C++ programs you will run on the PC, the libraries to access the robot radio interface and the tracking system, as well as the compiled robot flashing tool. The second one contains the C code that will run inside the robot. All the "low-level" part contained in the `firmware` subdirectory can be used as-is; **you are not supposed to modify it**. If you think that you need to modify something there please ask beforehand, as all the exercises can be done without any modifications. For example, **do not** add static registers to the `registers.h` firmware file, as you are supposed to implement your registers using a register handler callback (see exercise 2). **When editing existing code, make sure that variable and function names still make sense**! For example, please do not leave the name `read_registers` to a function that would write a register to start the robot!

1. First program (1 point)

Open the source code in `robot/ex1/main.c` and read it. What will happen when running this code on the microcontroller? If you want to see what the functions called from `main()` do, you can find all the source codes in `robot/firmware` (the header files contain comments that explain what each function do and what the parameters are).

From the command line, go to the source directory containing the file, and run `make`. Once the program is compiled, ensure that the radio interface is connected to the computer, turn on the head module by putting the magnet in place (or by removing it, if you are using the lamprey robot), and run `make program` to send the compiled program over the radio link. If the programming succeeds (look at the messages on the screen), the microcontroller will reset and run your program: is it doing what you predicted by reading the source code? If programming did not succeed, just retry (perhaps keeping the robot closer to the radio interface).

Now modify the program to make the LED blink in green at 1 Hz.

2. Registers (5 points)

This exercise is longer than most other ones, because understanding how to properly communicate with the robot is essential for all the following exercises. The particular communication protocol used here is proprietary (i.e., only used on this family of robots), but the concepts (register banks, callback functions, etc.) are commonly used.

The radio communication between the computer and the robot's head is based on registers, which are of four different types: 8-bit, 16-bit, 32-bit and *multibyte* (which have a variable length between 0 and 29 bytes). Each register has an address between 0 and 255[1]. **Each register type has its own address space, meaning for example that the 8-bit register at address 21 has nothing to do with the 32-bit register having the same address**. All register operations (reads or writes) are initiated from the computer, and are handled inside an *interrupt service routine* (ISR) in the head's microcontroller. The ISR will in turn call all the registered callback functions[2] that handle the actual implementation of register operations.

Open the source code in `robot/ex2/main.c`. You will see the declarations of some register callback functions. The program in `pc/ex2/ex2.cc` will read from and write values to these registers through the radio interface. Read both files (and read the documentation comments inside the included `.h` files in `pc/common` to find out what the called functions do). What will `ex2.cc` output on the screen (i.e., what do you expect the register contents to be)? Why? Explain in detail. Compile both programs using `make` in the corresponding directories, send the firmware to the robot, and then run `ex2`. Does the output correspond to what you expected? If not, why?

3. Communication with other elements (3 points)

The different modules of the robot are connected together with a CAN bus having a speed of 1 Mbps. Over the CAN bus, a custom protocol is used to define how the elements communicate. This protocol implements register read/write primitives for 8-bit, 16-bit and 32-bit registers (similar to those used by the radio communication, **but unrelated**). The head module initiates all CAN register operations, while modules can only answer. Body elements have their unique address (indicated on a label on the side and/or back), whereas limb elements have three distinct addresses: the first one (n) being indicated on its label and corresponding to the body DOF, and the other two ($n + 1$ and $n + 2$) corresponding to the limb DOFs.

Read the source code in `robot/ex3/main.c`: what does it do? Compile the program, run it, and verify that it indeed does what you expected it to do. Now implement a program that can read back the positions of each DOF and continuously display it on the PC screen (*hint: use a multi-byte register*). Test your program.

---

[1] The actual address range is 0–1023, but registers over 255 are reserved (e.g., for the bootloader or radio interface).

[2] See `http://en.wikipedia.org/wiki/Callback_(computer_programming)`

Warning: *the CAN bus access is interrupt-driven, meaning that an interrupt service routine will be triggered by the CAN hardware; therefore,* ***you cannot use register functions for the CAN bus inside an interrupt*** *(and thus inside the callback for the radio registers), as no other interrupts will be allowed to run at the same time! If you do, the program in the robot will get stuck each time you read the concerned register.*

4. Position control of a module (3 points)

Important*: elements do not have an absolute position sensor, and therefore have to be reset manually at zero position at startup. You should place the output axis of all elements at zero before starting the controller (for the body, this is the center position; for the limbs this depends on the implemented trajectory generator, as the shaft has no mechanical limit). When writing programs controlling the motors, pleas also make sure that the PD controllers are started* ***only*** *when needed (i.e., not immediately when turning on the robot). It is also good practice to stop them when the robot does not need to be actively moved anymore.*

Look at the sources in `robot/ex4`. They will make a motor move: verify the address to make sure that the motor is the last one of your chain (the default address will possibly work on the boxfish robot but not on the lamprey). Compile the program and run it. To enter the function that will make the robot move, you will need to write a value to a specific register: based on `ex2.cc`, write a program that writes the value *1* to `REG8_MODE` (include `regdefs.h` on PC side to define this constant). Make sure that the element that will move is at its zero position before starting the program! Once the element moves, stop it either by writing a *0* to the same register, or by turning off the whole robot. Now, modify the programs to be able to send the *setpoint* for the element from the computer, and send a sine wave at 1 Hz over the radio connection. The oscillation amplitude should be ±40°. *Hints*: in `utils.h` (located in `pc/common`) there is a function named `time_d()` (which returns the current time in seconds, including the fractional part), the `sin()` function is declared in `<math.h>`, and π is `M_PI`.

5.1. Simple on-board trajectory generation (3 points)

The source codes in `robot/ex5` implement a sine-wave generator (which is enabled by writing a *2* to `REG8_MODE`) with a period of 1 s. The program demonstrates how to use the integrated timer of the microcontroller. The output sine wave is visible as intensity on the LED. Compile the program and test it. Then, modify it to send the sine wave to a motor (as before, with an amplitude of ±40°). Does the obtained sine wave look the same as the one obtained when the controller was on the computer? If not, why? If yes, will that always be true?

5.2. Modulating trajectory parameters (3 points)

Modify your trajectory generation program so that you can specify the frequency and amplitude of the sine wave from the computer. Limit the frequency to 2 Hz and the amplitude to ±60°. You might want to use the `ENCODE_PARAM_8` and `DECODE_PARAM_8` macros, declared in `registers.h`, to encode floating-point numbers on a given range inside an 8-bit register (to use those macros on the PC side, include `regdefs.h`).

6. Using the LED tracking system

The aquarium where your robot will swim is equipped with a simple spot tracking system, that can give in real time the *x* and *y* coordinates of one or more spots (generally LEDs) in the image. The program in `pc/ex6/ex6.cc` shows how to connect to the tracking system and displays the current position of a spot (LED) in the aquarium. Please check with the assistant that the tracking system is ready to use, then compile and run the program. Check on the tracking PC screen that there are no false detections owing to ambient light (the fluorescent lights at the ceiling must be turned off, and the window blinds partially closed). Put a plate of expanded polystyrene with a LED in the aquarium, and make it move under the tracking system, checking that the displayed current position is correct.

6.1. Combining the tracking with the radio (2 points)

The programs you used for the robot until now (e.g., `robot/ex5`) support changing the color of the head LED through the radio by writing a RGB value on the 32-bit register at address 0. Given three unsigned 8-bit values `r`, `g` and `b`, the LED color can for example be changed as follows:

```
uint32_t rgb = ((uint32_t) r << 16) | ((uint32_t) g << 8) | b;
regs.set_reg_dw(0, rgb);
```

While keeping the green value at 64 so that the LED is always detected by the tracking system, write a program that changes the red and blue components of the LED color depending on its *x* and *y* coordinates in the aquarium (which size is approximately 6.0 × 2.0 m). Test the program by putting the module on the expanded polystyrene plate and moving it in the aquarium.

To have only one spot detected in the system, you should turn off the LEDs of the other elements composing the robot. For that, you can use the following code **in the robot's head** (please notice that you can't communicate with the modules directly from the computer through the radio!):

```
set_reg_value_dw(eladdr, MREG32_LED, 0);
```

where `eladdr` is the address of the motor module whose LED has to be turned off (for this to work, you will have to include `"can.h"` and `"module.h"`).

Hint: on the PC side, you will now be using both the radio library (`remregs.cc`) and the tracking library (`trkcli.cc`) in the same program. You will therefore have to modify the `Makefile` to link both of them to the program you are compiling.

7. Swimming and experiments (10 points)

7.1. Writing a swimming trajectory generator

You should now be able to write a trajectory generator for the robot, to achieve swimming locomotion. Please include a register in the program, so that you can set the oscillation frequency from the computer.

- For the boxfish robot, you can read part of the semester projects reports of Benjamin Fankhauser, Daisy Lachat and Ariane Pasquier (see `http://biorob2.epfl.ch/pages/studproj/?id=birg71745`, `http://biorob2.epfl.ch/pages/studproj/?id=birg57462` and `http://biorob2.epfl.ch/pages/studproj/?id=birg61246`) to find some ideas of how to move the fins to obtain swimming with a similar robot.
- For the lamprey robot, you can simply generate a *travelling wave* from head to tail, possibly with its amplitude increasing towards the tail. A travelling wave is simply generated by sending a sine wave to each module, and introducing a constant phase lag between each element:

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

  where A is the oscillation half-amplitude, ν the oscillation frequency, t the time, i the element number (starting at 0 from the tail), ϕ the total phase lag, and N the total number of active elements in the robot.

7.2. Experiments

Once you obtain a satisfying swimming motion, write a controller program on the computer that uses the tracking system to measure the average speed of the robot. The program should make the robot swim (e.g., by writing a specific value in `REG8_MODE`), and stop it after a given time or travelled distance.

The scientific question to study is:

- for the **boxfish robot**: what is the influence of the oscillation frequency on the speed?
- for the **lamprey robot**: what is the influence of the total phase lag on the speed?

You should take 3 to 5 measures for each data point (*i.e.*, frequency or phase lag), and give the average speed and its standard deviation. Record the x and y coordinates of the LED, and plot some of the trajectories. For the frequency, do not operate the robot at more than 1.5 Hz to avoid stressing the motors. The total phase lag can be studied between $\phi = 0.5$ and $\phi = 1.5$.

8. More experiments (up to 5 bonus points)

If you have time left, you can carry out some additional experiments, for example (but not limited to): implementing steering (direction control) controlled by a register, investigating the dependency of speed on other parameters (*e.g.*, oscillation amplitude), implementing some behaviour using the tracking system as feedback (*e.g.*, going back and forth in the aquarium), interactively controlling the robot trajectory with the keyboard of the PC…