Real Time Systems
Lab 2: Programming a Webots Robot in Ada

Introduction

In this assignment, you will learn how to do basic real–time programming on an embedded device with a runtime that supports real–time tasking.

For this assignment we are using the Webots robot simulator. The tasks you write in Ada will communicate with Webots (which is handled by the API we provide to you using TCP/IP). The simulation will not be running in real–time; however, you will be writing real–time code that is very similar to what you would write on a real embedded device. You do not need to do any programming in Webots since we will provide you with all necessary files.

The main goal of this lab is to introduce some concepts of real–time systems and programming in a practical application. The lab has three parts:

- Part 1 serves as warm–up to the lab. The goal is to introduce the lab environment and ensure that everything is working in a simple setup.
- Part 2 requires you to implement a fully controllable car (using the directional keys on the keyboard). This part aims to highlight the principles of event–driven scheduling.
- Part 3 requires you to implement a line follower robot with obstacle detection. You need to use either event–driven scheduling (described in part 2) or periodic scheduling (described in part 3) to implement this part of the lab.

Evaluation

Demonstration

You need to demonstrate your solutions for parts 2 and 3 and have them both approved by at least one of the teaching assistants (TAs). This should be done either during one of the help sessions for this lab (lab 2) or during the demonstration slot specified in the schedule.

The lab will be done in groups (same groups will be used as in Lab 1), and all students participating in the group should be present during demonstration and independently be able to describe all parts of the solution (i.e., do not divide the parts within the group). The TAs might ask different group members to continue the explanation of their solution and/or direct questions at them.

Hand–in

You must submit the Ada source files (`.adb` and `.ads` only) of your solutions for parts 2 and 3. You should place each part in a separate folder, compress these two folders into a single archive, and then upload this single archive to Studium by the deadline. Also hand in a document briefly outlining the group's work distribution as per the general lab instructions.

> Please note that in addition to the individual requirements of each part, the technical quality of your solution is also evaluated. We can ask for supplementation if we find the quality to be insufficient. Some common examples are: non–indented code, badly structured code, lack of comments etc.

1 Part 1: Hello Webots!

This first part is supposed to get you used to compiling and running programs, together with simple input/output operations using the Webots API. The program you will write will print the sensed light value on the console.

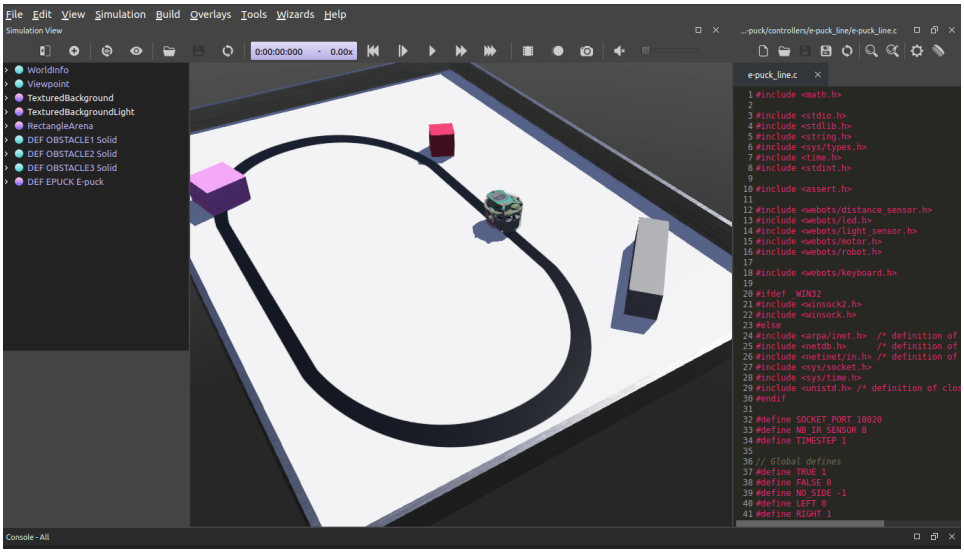
1.1 Setting up Webots

This part assumes that you have Webots installed on your system: either using the provided virtual machine (VM) or by installing it natively. Even if you are using the VM, you still need to follow below steps as the VM does not include the required project files.

Download the program skeleton (`lab2-skeleton.zip`) from Studium and extract it. The extracted files should contain a folder named `e-puck`.

To open the Webots simulation, run Webots. Then select `File/Open World` in Webots. Browse to where you have extracted the folder, and select `e-puck/worlds/e-puck_line.wbt` to open the simulation world. Note that you need to put this folder somewhere where Webots will have write access.

You should see a single e-puck robot in an arena.



Do not start the simulation yet, but try building the controller by selecting `Build/Build` (or by pressing `F7`) in Webots. If the build operation fails on your system, we encourage you to use the provided VM.

1.2 Setting up the Ada project

This part assumes that your system has the necessary tools to compile and run Ada programs: either using the provided VM or by installing natively.

Extract the provided skeleton program (under the `skeleton` folder of `lab2-skeleton.zip`) to your system:

`main.adb` Contains the entry point of the program along with `SyncTask` which synchronises with Webots. You do not need to change anything in this file, but you might want to slightly increase the `Period` value if you have performance issues while running the simulation.

Note that we assigned lowest priority (`System.Priority'First`) to the main procedure by using attribute `'First` which indicates the first value of a range. `SyncTask` is assigned the highest priority, meaning that it will preempt all other tasks of lower priority.

`tasks.ads` Contains the specifications of your tasks that are accessible from other files, as well as any variables you wish to specify. It is a good idea to declare program–wide variables which you want to be easily changeable (such as task periods) here.

`tasks.adb` This is the body (or implementation) of your tasks. This file is where you will be doing most of the work.

`webots_api.ads` Contains the specification of API functions and procedures for accessing Webots input/output data. You should use the functions and procedures specified here to read the sensors and set motor speeds.

> On a real embedded device, you would need to use the provided drivers for the input/output operations. Since we are using a simulator, all input/output operations that you need in this lab are provided to you in `webots_api.ads` as part of the program skeleton.

`webots_api.adb` Contains the implementation of the Webots API which communicates with Webots using TCP/IP ports. Do not modify anything in this file.

1.3 Writing the code

Your code should do the following:

- Set one or both motor speeds to some value within the range [-999,999] (negative values change direction), so that your robot has some visible movement in the arena.
- Read one of the three ground-facing light sensor values and display it repeatedly with a delay of 100 ms in–between.

Compile your code with the command: `gnatmake main.adb`

Note that the provided skeleton program should compile out of the box without errors (but will not print anything).

1.4 Running the simulation

If you receive no errors and can generate an executable for your Ada program (the controller must also have been built in Webots), you can test your program by first running the Webots simulation, and then running the executable (on Unix–like systems execute `./main` from a terminal; on Windows you can simply double–click `main.exe`).

You should see the light values changing when the robot is over a black line, and see it moving with the speed you set in the motors. You can also move the robot (and other objects) manually around by left click dragging it while holding the shift keyboard button.

In order to ensure smooth operation while testing your programs (i.e., not to wait for the TCP/IP port to timeout), pause Webots simulation before stopping the Ada program. The Ada program can then be stopped by pressing ctrl+c in the terminal.
To run your program again, first reset the Webots simulation either by pressing the "Reset Simulation" button or by using its shortcut ctrl+shift+T in Webots. After the simulation is started you can run your Ada program again.

If there are performance issues while running Webots, preventing you from working on this lab, we encourage you to work together with a groupmate who is able to run it. If you are unable to solve this within your group, get in touch with the course TAs as soon as possible so that we try to find a solution.
Some things which might help increase performance:

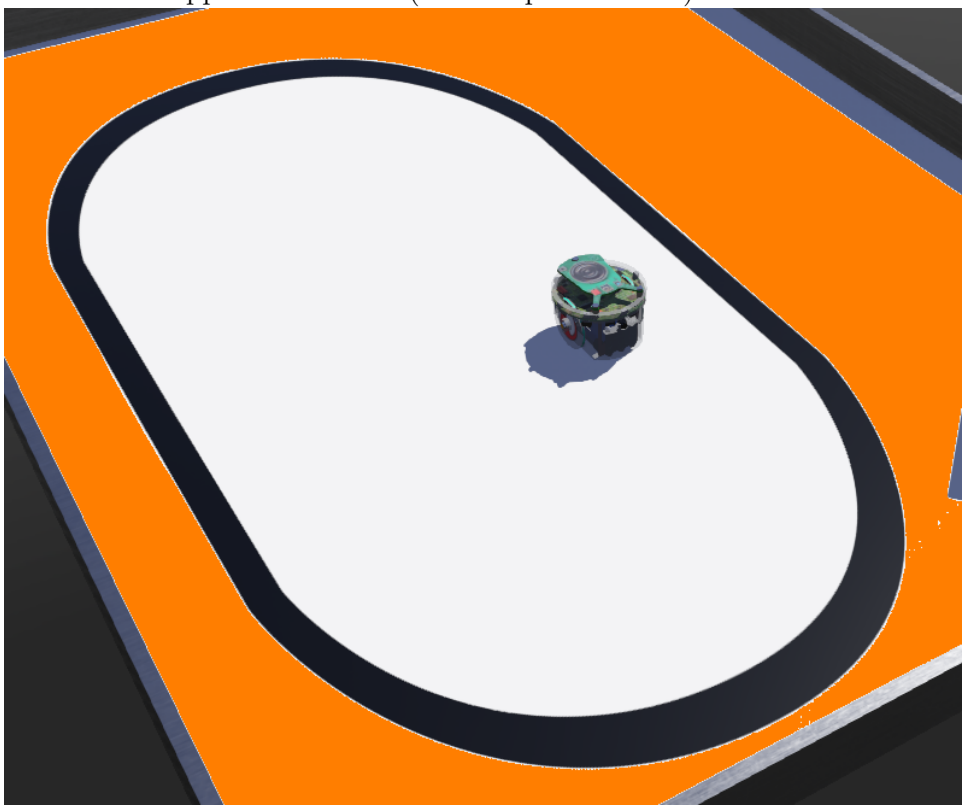
1. Disable any visual effects your operating system might have (these might be turned on by default in the provided VM). According to Webots documentation, these might cause a performance drop of up to 10x on a Linux OS.

2. Turn down (or off) the graphics quality settings in Webots, under Tools/Preferences/OpenGL.

2 Part 2: Event–driven scheduling

In this part, you will learn how to program event–driven schedules. The target application will be a Webots robot that can be manually controlled using the keyboard directional keys (as long as at least one of the keys is pressed) and while it is not over a black line (in which case it should immediately stop).

The robot must not be able to be driven past a black line. You are free to come up with an implementation that comes to a total halt as soon as the black line is detected, or an alternative implementation where it could be driven in the opposite direction (but not past the line).



(In the image above, if the robot started in the white-shaded area, it should not be possible to drive it to the outer orange-shaded area.)

2.1 Handling Events

Ideally events generated by external sources are detected by the interrupt service routines (ISRs). This allows to react immediately to signals from various sources. Unfortunately, the Webots API is working in polling mode: the sensors need to be asked for their state again and again, instead of updating the state themselves when something interesting happens.

Our workaround for this is to create a small, second task that checks

the sensors periodically (about every 10 ms). If the state of the sensor has changed, it generates the appropriate event for us.

First we need to create a protected object named `Event` with a single entry (and an enumeration of possible `EventID`s):

```
type EventID is (UpPressed, DownPressed, ...) -- events

protected Event is
       entry Wait(id : out EventID);
       procedure Signal(id : in EventID);
private
        -- assign priority that is ceiling of the
        -- user tasks' priorities
       current_id : EventID;     -- event data
       signalled : Boolean := false;  -- flag for event signal
end Event;

protected body Event is
      entry Wait(id : out EventID) when Signalled is
      begin
         id := current_id;
         signalled := false;
      end Wait;

      procedure Signal(id : in EventID) is
      begin
         current_id := id;
         signalled := true;
      end Signal;
end Event;
```

This protected object can be used by different tasks to communicate between them. For example, a task can block until it receives an event:

```
Event.wait(received_event);
```

An event dispatcher task can notify the blocked task by sending an event:

```
Event.signal(event_id);
```

In order to do this, declare and implement a task `EventDispatcherTask`. It should call the appropriate Webots API function to read the button presses (`button_pressed`) and compare it to its old state. If the state has changed, it should release the corresponding event by using `signal` procedure of the `Event` protected object. As an example, you may define event ids like `UpButtonPressed` and `UpButtonReleased`. Just as in part 1, put your code in an infinite loop with a delay in the end of the loop body.

It is up to you to decide whether the robot can move diagonally or not (i.e., you can choose if the robot should react to concurrent directional button presses such as up+left or down+right). You might consider some states to be invalid for simplicity (e.g. by allowing only one button press at a time), which would reduce the number of possible events.

Now create a new task `MotorControlTask` that does the following in an infinite loop:

- Wait for an event
- Update robot motor speeds based on the received event

In order for `MotorControlTask` to have priority over `EventDispatcherTask`, make sure to assign a lower priority to the latter. Otherwise, the infinite loop containing the sensor reading would might make the system completely busy and it could never react to the generated events. You are recommended to print some status output on the console (such as events and sensor data).

Compile and test the program to see if the car reacts to your commands.

2.2 Extending The Program

Extend the program to also react to one of the three ground-facing light sensors (light sensor 2 is in the middle front part, so it should suffice). The car should stop not only when no button is being pressed, but also when the light sensor detects that the car is on top of a black line. (You may need to play a little bit with the "Hello Webots!" program from part 1 in order to find appropriate light levels.). The car should only start moving again when the car is back on the white-shaded area and one of the directional keys is pressed again (alternatively you can make it so that the car can be moved only in the reverse direction when it detects a line). It should not be possible for the car to cross over a line completely.

The line detection should happen in `EventDispatcherTask` and be communicated to `MotorControlTask` via the `Event` protected object. Use two new events for that purpose. Make sure you define and use all events properly. The console should also provide some useful information about the state of the car.

Some clarifications

- The job of `EventDispatcherTask` is just to create the events signaling key presses/releases and line detection, *independently* from each other.
- All logic should happen in `MotorControlTask`, i.e., when to move or not, depending on the current state. This task must not read the sensors by itself and must not communicate with `EventDispatcherTask` by any other means except through the `Event` protected object, i.e., shared variables etc. are not allowed.
- A frequent mistake is that the periodic `EventDispatcherTask` generates an event each time it is executed, reporting the current state of the sensors. Instead, it should only generate events at *state changes*: one (and only one) in the moment when a button is pressed down and one when it is released[a]. The same is true for the light sensor: when the black line is detected, generate *one* event. Doing otherwise might spam the event system with redundant information.

[a] Note that you are free to choose the events for the key presses differently as explained before, e.g., you can consider pressing concurrent buttons such as up+left a valid state, or you can choose to constrain the implementation to only four distinct directional button presses. Having states for only four directional button presses only would lead to a simpler implementation, and would be a good starting point even if you want to add more states.

3 Part 3: Line Tracker

In the last part you need to create a robot that can simultaneously:

- Follow the black line that is drawn on the floor using the light sensors.
- Keep a constant distance to detected objects in front of it using the distance sensors.

In order to pass this part, your robot must succeed in both points, and your implementation should either use periodic scheduling (as described in Section 3.1), or event–driven scheduling as described in Section 2. As an example, putting everything in a single task would not be sufficient to pass this part (even if your robot follows the line and avoids obstacles).

Some clarifications

- You may use all three available light sensors.
- The obstacles will be the solid boxes already present in the arena. During demonstration, you will be asked to manually place these boxes (and possibly move them) in front of your robot. Alternatively, we may also provide a script to do this automatically.
- The robot does not have to drive backwards. The obstacles it has to follow will not move backwards, and if your robot gets too close, it is sufficient to just slow down/stop.
- Your robot does not need to increase its normal speed to catch up to an obstacle if it is moving too fast; however, if the obstacle is moving at a slower constant speed, your robot should also keep moving behind it while keeping a constant visible distance[a] behind the obstacle.
- Don't assume a direction on the track; however, your robot will be placed alongside a black line. Your robot must work both clock- and counterclockwise equally well.

[a] The distance sensor in the simulation cannot detect objects until they are quite close; however, a visible gap should still be achievable.

3.1 Periodic Scheduling

Real–time schedulers usually schedule most of their tasks periodically. This usually fits the applications: sensor data needs to be read periodically and reactions in control loops are also calculated periodically and depend on a constant sampling period. Another advantage over purely event–driven

scheduling is that the system becomes much more predictable, since load bursts are avoided and very sophisticated techniques exist to analyse periodic schedules (i.e. response–time analysis).

A high level structure for the system of this part can be as follows: We have several tasks that are scheduled periodically, with different periods:

- A task `MotorControlTask` that only takes care of setting the speeds of the motors.
- A task `LineFollowingTask` that reads the light sensors and sends commands to `MotorControlTask`.
- A task `DistanceTask` that reads the distance sensor and sends commands to `MotorControlTask`.
- A task `DisplayTask` that displays some status information.

Obviously, the tasks can have different periods, since while we would like the car to react fast to changes in light sensor readings, we can't (optically, as humans) read updated information from the display faster than in certain intervals anyway.

Before we implement the actual tasks, we need a way for them to communicate. A straightforward way of doing this is to make the tasks communicate directly with each other; however, this kind of implementation does not scale up well and makes the program more complicated (i.e., more bugs and deadlocks).

Another way would be to use global data structures which can be modified by some tasks, and read by others. In this example, the `LineFollowingTask` and `DistanceTask` tasks can update a driving command, which can be read by the `MotorControlTask`. Since this can result in unwanted race conditions, you can define a *protected object* that contains the data you want to communicate. The tasks can then pass information by accessing this protected object. An example protected object is given below:

```
protected SharedData is
   procedure Set (V : Integer);
   function Get return Integer;
private
   --  Data goes here
   some_data : Integer := 0;
end Obj;

protected body SharedData is
   --  procedures can modify the data
   procedure Set (new_data : Integer) is
   begin
```

```
      some_data := new_data;
   end Set;

   --  functions cannot modify the data
   function Get return Integer is
   begin
      return some_data;
   end Get;
end SharedData;
```

The tasks can at any time write a new *command* into this (global) protected object. Using this, you can now define the tasks in your system.