Renesas Virtual MCU Rally

Documentation and rules (draft aug. 2021)

Renesas Europe

Subject to updates

Table of contents

Renesas Virtual MCU Rally 3

Installation 4

- Installing WeBots 4
- Simulation environment 5

First steps 6

- Project setup 6
- Creating controllers 8

Programming guide 9

- API functions 9
- Line following 10
- State machine programming 11

Manuals 13

- Tuning the MCU car 13
- Building tracks 14

Competition 15

- Regulations 15
- Guide for participants 16

Renesas Virtual MCU Rally

Renesas' first virtual student contest is the perfect opportunity to combine practical learning and fun. Engineering students and their educators from universities all over Europe will build a self-guided MCU robot in the Renesas Virtual MCU Rally simulation environment and optimize the software that auto drives the robot along the white line on the track.

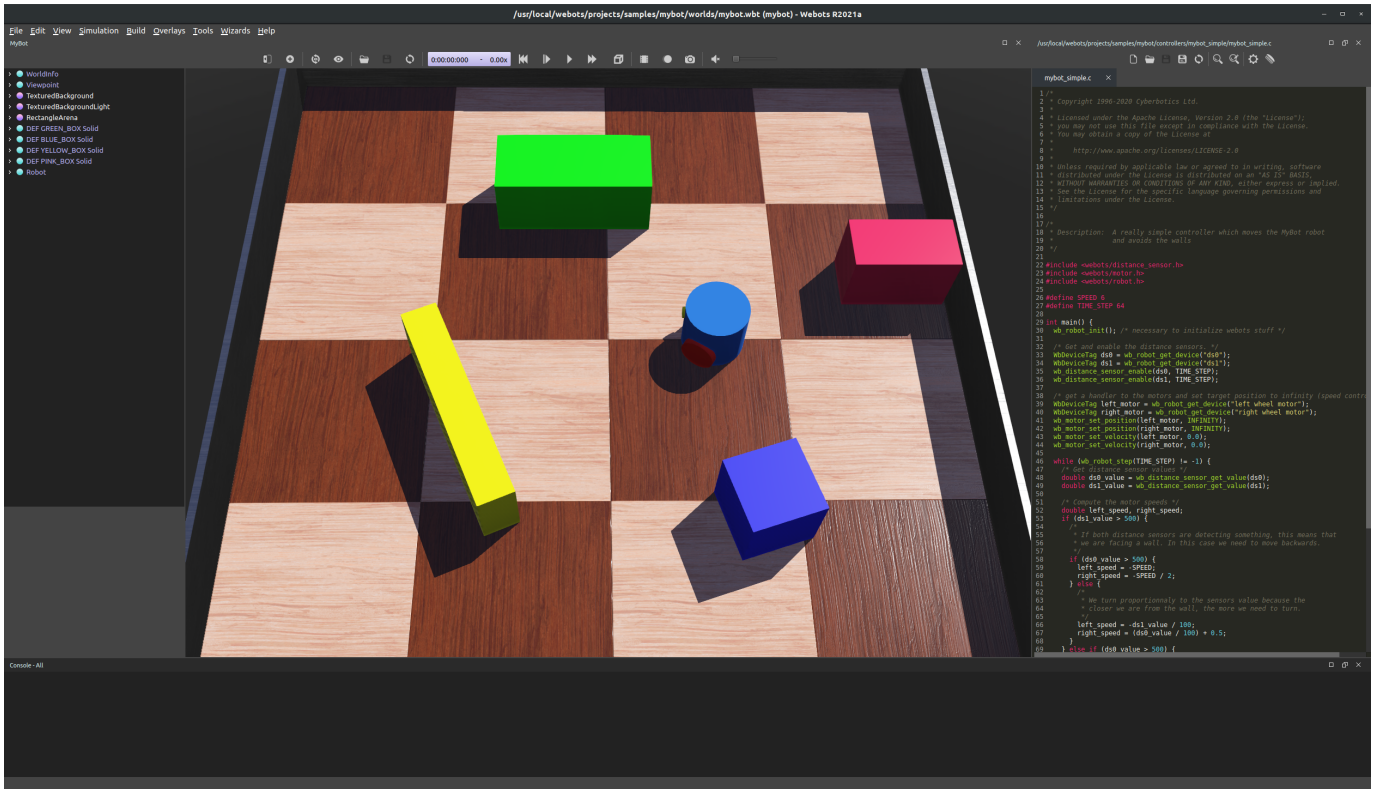
Installation

Installing WeBots

The simulation uses the open-source WeBots simulation software. WeBots is a state-of-the-art robotics simulation package, selected for the competition for its rendering capabilities, precise physics simulation, extensive documentation, seamless extensibility, and cross-platform compatibility.

The first step of the installation is to install the latest version of WeBots (2021b) using the instructions, provided on WeBots website: https://www.cyberbotics.com/doc/guide/installation-procedure.

Install and run WeBots. After initial environment setup, a default simulation world should start running. Feel free to explore the sample projects under File -> Open Sample World.



A example of a WeBots simulation environment: scene tree on the left, 3D rendered scene in the middle, and the coding environment on the right.

Simulation environment

Download the Renesas Virtual MCU Rally environment from the Git repository:

- Navigate to Latest releases
- Download the .zip file
- Windows users: renesas_virtual_mcu_rally_windows
- Linux users: renesas_virtual_mcu_rally_linux
- Advanced users: clone the repository instead, and compile the controllers yourself

Run the demo:

- Open WeBots, select File -> Open World...
- Navigate to the Simulation environment folder obtained during the installation.
- Open: embedded_world.wbt from the worlds folder

Explore the structure of the project:

- controllers: your program will go here
- referee: the automated refereeing program used by the timing gate asset
- safety_car_controller: a simple robot controller, used by the safety car
- docs: these instructions, manuals, etc.
- libraries: includes the robot programming API, used by your program
- plugins: empty, but can contain additional plugins for WeBots
- protos: assets, such as the robot model, the timing gate, etc.
- track parts: assets for building custom tracks
- worlds: contains world files, main project files that WeBots can open
- 3dmodels: 3D models used by the project
- textures: textures used by the project

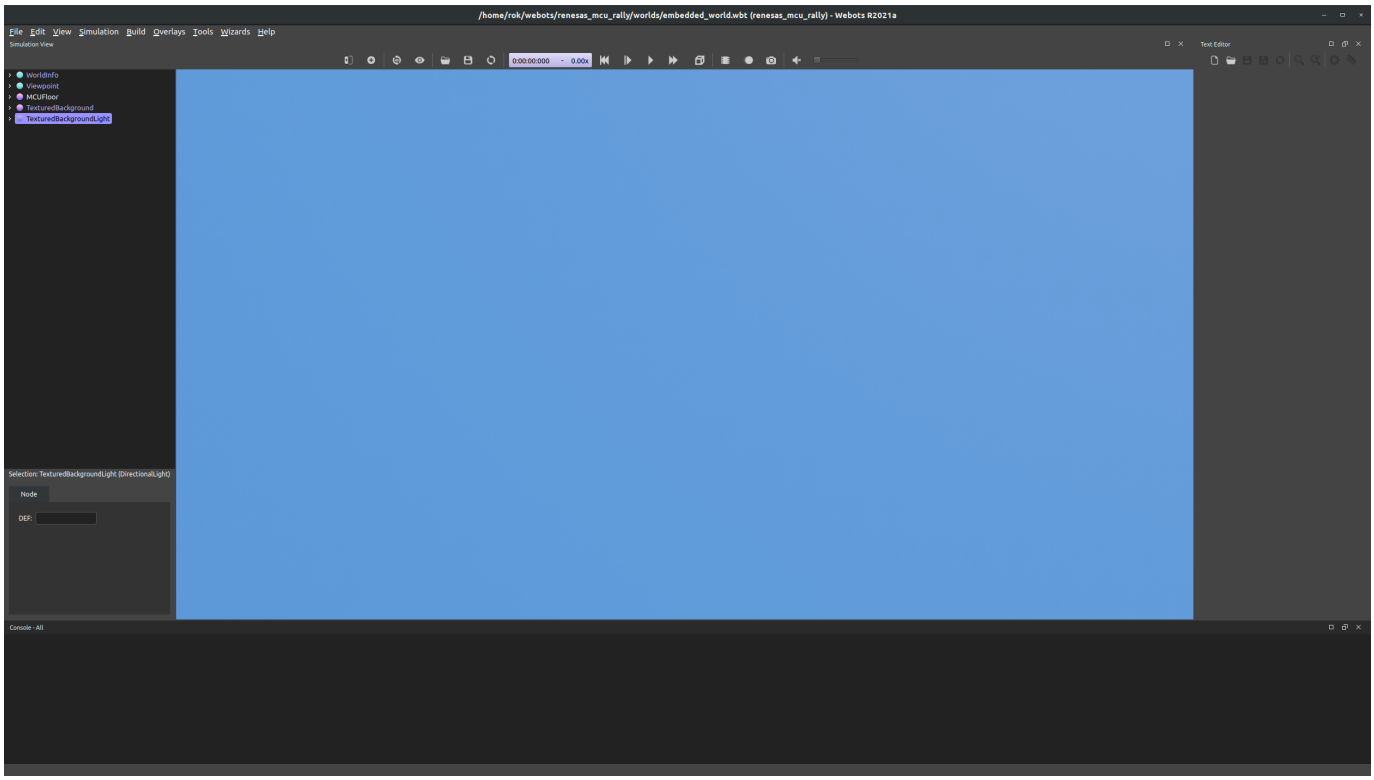
First steps

Project setup

To set up the Renesas Virtual MCU Rally simulation from scratch:

- Open WeBots, select File -> Open World...
- Navigate to the Simulation environment folder obtained during the installation.
- Open: empty_world.wbt from the worlds folder

An empty world should appear, as shown in the following figure.



The default, empty WeBots world.

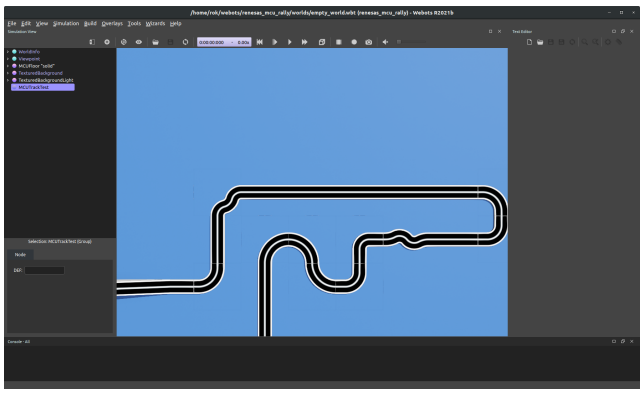
To set up your first simulation:

- assets, such as the robot and the track, need to be added, and
- the corresponding programs, called controllers, need to be compiled.

Adding assets

Assets can be added by clicking on the (+) icon (second icon in the toolbar, shortcut Ctrl-Shift-A). The assets are added to the scene tree on the left, below the currently selected asset. Start by adding the test track.

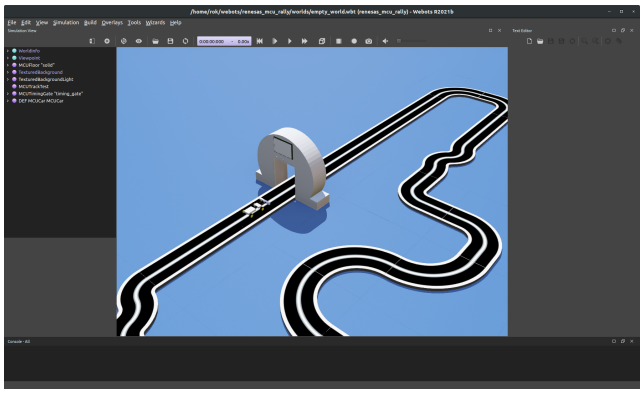
- Pause and reset the simulation (click on the pause icon || and the rewind icon <<).
- Click on the add asset icon.
- Under PROTO nodes (Current project), select tracks->MCUTrackTest and click Add.
- Save the modified world (Save icon, shortcut Ctrl-Shift-S).



The test track is now added to the world.

Repeat the procedure to add two more assets:

- MCUTimingGate
- MCUCar



All test assets are now added to the world. Save the world before continuing.

Compiling controllers

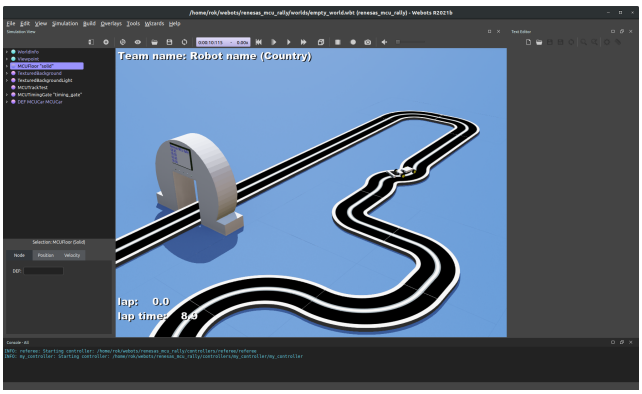
The next step is to compile all the programs (controllers). This includes the timing gate controller, which takes care of automated timing and refereeing, as well as the MCUCar controller. The controllers can be opened, edited, and compiled using the Text Editor on the right side of the WeBots environment. To compile the timing gate controller:

- Click on Open an existing text file icon (second icon in the Text Editor sub-window).
- Navigate to controllers -> referee -> referee.c.
- Click Build the current project (gear icon in the Text Editor)

Repeat the procedure for the safety_car_controller.c, located under controllers -> safety_car_controller and my_controller.c, located under controllers -> my_controller.

Running the simulation

After this, the simulation can be started with the "Run the simulation in real-time" button - the play icon in the toolbar.



An example simulation is now set up.

Creating controllers

To program the MCU Robot, a custom controller needs to be created and compiled, and then, the robot needs to point to the newly created controller executable.

To create a your robot controller, use the WeBots menu: Wizards -> New Robot Controller. Use C as the controller language. Name the controller, however you want. In this example, the name will be `my_controller`.

WeBots creates a new folder within the controllers folder of the project and two files:

- my_controller.c: The source code for the controller, to be modified.
- Makefile: The instructions for compiling the controller, to be edited once.

Modify the Makefile

Find the Makefile in the my_controller folder and modify the section before the "do not mofify" tag to include `C_SOURCES` and `INCLUDE` path definition. Make sure that you rename my_controller.c to the name you chose for your controller.

```
###-----------------------------------------------------------------------------

C_SOURCES = my_controller.c ../../libraries/renesas/renesas_api.c ../../libraries/
renesas/renesas_api_webots.c
INCLUDE = -I"include" -I"../../libraries/renesas/include"

### Do not modify: this includes Webots global Makefile.include
null :=
space := $(null) $(null)
WEBOTS_HOME_PATH=$(subst $(space),\ ,$(strip $(subst \,/,$(WEBOTS_HOME))))
include $(WEBOTS_HOME_PATH)/resources/Makefile.include
```

Default program

To start programming the MCU car, replace the code in my_controller.c with the following.

```
#include "renesas_api.h"

int main(int argc, char **argv)
{
  wb_robot_init();
  init();
  while (wb_robot_step(TIME_STEP) != -1)
  {
    update();
  };
  wb_robot_cleanup();
  return 0;
}
```

Before compiling and running the program, let us examine the function call of the default program.

`#include "renesas_api.h"` : The Renesas MCU Rally application programming interface (API) is included. The API defines the functions that you can call to control your sensors and actuators.

`wb_robot_init();` : The main program initializes the WeBots API (required by WeBots).

`init();` : Initializes the Renesas API.

`update();` : Updates the sensor data of the robot (line sensors, encoders). Needs to be called every simulation time step.

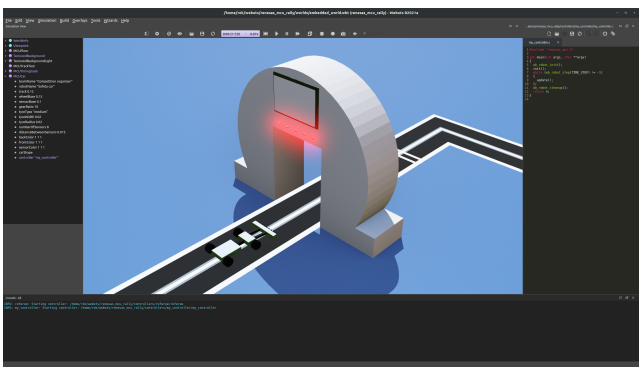
`wb_robot_cleanup();` : Frees the resources (required by WeBots).

Notice that the `update` function runs repeatedly inside a while loop. The code in the while loop is executed every controller step which is defined to be 5 ms and is the same as the simulation step. The majority of your code will need to be executed repeatedly and will therefore need to be placed after the `update` function inside the while loop.

Compiling and running

Before writing a controller that actually moves the robot, try to compile the controller and link it to the robot.

- With my_controller.c opened in the WeBots code editor, click the gear icon to compile.
- Modify the controller field in the MCUCar asset to point to my_controller.
- Expand the MCUCar asset in the scene tree view (left side of WeBots).
- Click on the controller field, Select..., my_controller.



The robot should now stay in place once the simulation is run.

Programming guide

API functions

void init()

Initializes the virtual Renesas MCU controller.

void update()

Updates sensor values, should be called in main controller while loop.

void handle(int angle)

Servo steering operation.

- `angle` : Servo operation angle: -90 (left) to 90 (right)

void motor(int back_right, int back_left, int front_right, int front_left)

Motor speed control. 0 is stopped, 100 is maximum voltage forward, and -100 is maximum voltage, but in reverse.

- `back_right` : Back right motor
- `back_left` : Back left motor
- `front_right` : Front right motor
- `front_left` : Front left motor

unsigned short *line_sensor()

Returns the values of the line sensors.

- `returns` : Sensor values as an unsigned short array.

Example that prints raw sensor values:

```
unsigned short *sensor = line_sensor();
for (int i = 0; i < 8; i++)
{
    printf("%d ", sensor[i]);
}
printf("\n");
```

double *encoders()

Returns the motor speeds (in rad/s). An array of 4 values is returned: `{back_right, back_left, front_right, front_left}`, where:

- `back_right` : Back right motor velocity
- `back_left` : Back left motor velocity
- `front_right` : Front right motor velocity
- `front_left` : Front left motor velocity

double *imu()

Returns the roll, pitch, and yaw angles of an inertial measurement unit, attached to the main platform. An array of 3 values is returned: `{roll, pitch, yaw}`.

double time()

Returns the current time in seconds since the start of the controller program.

Car parameters

The car parameters are accessible via the following functions.

```
double get_track();
double get_wheel_base();
double get_sensor_base();
double get_gear_ratio();
const char* get_tyre_type();
double get_tyre_width();
double get_tyre_radius();
int get_number_of_sensors();
double get_distance_between_sensors();
double get_weight_penalty();
```

Line following

The basic functionality of a Rensas MCU Rally Car is line following. A simple line following program will consist of the following:

- Reading raw sensor data.
- Calculating the position of the line with respect to the sensor.
- Devising a control algorithm to make the robot follow the line.

Let us first look at each of these functionalities before merging them together into a working program.

Reading raw sensor data

Line sensor data needs to be read within every step of the program, therefore, it needs to be within the `while` loop of the controller. The following two lines need to be called to access line sensor data.

```
update();
unsigned short *sensor = line_sensor();
```

`update()` updates all sensor values (including encoders and line sensors), while `line_sensor()` API function returns an array of raw sensor data.

Line sensors are simulated as an array of infrared reflectance sensors. Each sensor measures the amount of reflected light, bouncing off of the surface. In the simulated setup, a higher number corresponds to a darker surface. The raw sensor data takes the form of `unsigned short` values since each sensor value can be between 0 and 1024. Typically, the value of sensors on a black surface will be around 500, while the value of the sensors on a white line will be around 200, but this depends on many external factors. Individual sensor values can be accessed using arrays, e.g. `sensor[0]`, `sensor[1]`, etc.

Calculating the position of the line

Detecting the position of the line below the sensor can be done in many ways. For example, one can simply look for the sensor with the lowerst value. A more robust approach is to calculate the weighted average of the raw sensor data. Below is a calculation for 8 sensors, using the standard weighted average formula.

Note that for 8 sensors the `weighted_sum / sum` ratio will be between 0 and 7, therefore `3.5` is subtracted to make sure that the position of the line is negative if the line is on one side of the centre, and positive on the other.

Another important note is that the following code averages raw sensor data, therefore, the result will be the average position of black (not white) surface under the sensor.

```
float line = 0, sum = 0, weighted_sum = 0;
for (int i = 0; i < 8; i++)
{
  weighted_sum += sensor[i] * i;
  sum += sensor[i];
}
line = weighted_sum / sum - 3.5;
```

Devising a control algorithm

A very simple control algorithm would be as follows:

- move forward by applying a constant voltage to the electromotors,
- turn the handle with the servomotor to follow the line (the further away the line, the bigger the turn).

This control algorithm can be written in 2 lines of code using `motor` and `handle` function of the API.

```
motor(40, 40, 40, 40);
handle(1000 * line);
```

The final program

Copy-paste the following program into your controller code and test how it works.

```
#include "renesas_api.h"

int main(int argc, char **argv)
{
  wb_robot_init();
  init();

  while (wb_robot_step(TIME_STEP) != -1)
  {
    update();
    unsigned short *sensor = line_sensor();
    float line = 0, sum = 0, weighted_sum = 0;
    for (int i = 0; i < 8; i++)
    {
      weighted_sum += sensor[i] * i;
      sum += sensor[i];
    }
    line = weighted_sum / sum - 3.5;
    motor(40, 40, 40, 40);
    handle(1000 * line);
  }
  wb_robot_cleanup();
  return 0;
}
```

State machine programming

Line following is just one of the functionalities that the Renesas MCU Rally robot needs in order to complete a track as there are three other types of obstacles:

- 90 degree turns (marked by two white stripes across the track),
- left and right lane changes (marked by white stripes across the corresponding half of the track), and
- elevation changes (unmarked).

A structured way of solving these additional problems is to introduce *state machine* programming. The robot's behaviour can then be described as a set of states and transitions between them. For example, the robot starts in a *FOLLOW_LINE* state, in which it follows the line, but then switches to a *MAKE_90_DEGREE_TURN* state when it detects the white stripes that mark the turn.

Simple state machines in C

A rudimentary way of programming state machines in C is to use an `enum` as a data structure and `switch` statements. The following code introduces an `enum` variable `state`, and can be extended to an arbitrary number of named states.

```
enum states_t
{
  FOLLOW_LINE,
  MAKE_90_DEGREE_TURN
} state = FOLLOW_LINE;
```

Then, in the `while` loop of the main program, the robot's behaviour in each of the states and the state transitions need to be specified.

```
switch (state)
{
    case FOLLOW_LINE:
        // write the line following code
        // write the white stripe detection code
        // if white stripes detected, state = MAKE_90_DEGREE_TURN
    break;
    case MAKE_90_DEGREE_TURN:
        // follow line, but very slowly (for example)
        // detect when the turn was made
        // if turn made, state = FOLLOW_LINE
    break;
}
```

Example program

The following example program uses the state machine programming principle and represents a good basis for controller development. The program works with the default MCUCar robot with 8 line sensors. Feel free to use it as a starting point. It should, however, be studied, modified, and improved.

```
#include "renesas_api.h"

#define UPRAMP 60.0
#define FAST 30.0
#define MEDIUM 20.0
#define SLOW 10.0
#define BRAKE -40.0

#define STRICT 2000.0
#define LOOSE 20.0

enum states_t
{
  FOLLOW,
  CORNER_IN,
  CORNER_OUT,
  RIGHT_CHANGE_DETECTED,
  RIGHT_TURN,
  LEFT_CHANGE_DETECTED,
  LEFT_TURN,
  RAMP_UP,
  RAMP_DOWN
} state = FOLLOW;

double last_time = 0.0;
int left_change_pending = 0;
int right_change_pending = 0;
int detected_state = FOLLOW;

int main(int argc, char **argv)
{
  wb_robot_init(); // this call is required for WeBots initialisation
  init();          // initialises the renesas MCU controller

  while (wb_robot_step(TIME_STEP) != -1)
  {
    update();
    unsigned short *sensor = line_sensor();
    double *angles = imu();
    float line = 0, sum = 0, weighted_sum = 0;
    bool double_line = 0;
    bool left_change = 0;
    bool right_change = 0;
    for (int i = 0; i < 8; i++)
    {
      weighted_sum += sensor[i] * i;
      sum += sensor[i];
      if (sensor[i] < 500)
      {
        double_line++;
        if (i < 4)
        {
          left_change++;
        }
        else
        {
          right_change++;
        }
      }
    }
    line = weighted_sum / sum - 3.5;

    switch (state)
    {
    case FOLLOW:
      motor(FAST, FAST, FAST, FAST);
      handle(STRICT * line);
      if (angles[1] > 0.1)
      {
        last_time = time();
        state = RAMP_UP;
        printf("RAMP UP\n");
      }
      else if (angles[1] < -0.1)
      {
        last_time = time();
        state = RAMP_DOWN;
        printf("RAMP DOWN\n");
      }
      else if (double_line > 6)
      {
```

```
      if (detected_state == CORNER_IN)
      {
        last_time = time();
        state = CORNER_IN;
        printf("CORNER IN\n");
      }
      detected_state = CORNER_IN;
    }
    else if (time() - last_time > 0.2 && right_change > 3)
    {
      if (detected_state == RIGHT_CHANGE_DETECTED)
      {
        last_time = time();
        state = RIGHT_CHANGE_DETECTED;
        printf("RIGHT_CHANGE_DETECTED\n");
      }
      detected_state = RIGHT_CHANGE_DETECTED;
    }
    else if (time() - last_time > 0.2 && left_change > 3)
    {
      if (detected_state == LEFT_CHANGE_DETECTED)
      {
        last_time = time();
        state = LEFT_CHANGE_DETECTED;
        printf("LEFT_CHANGE_DETECTED\n");
      }
      detected_state = LEFT_CHANGE_DETECTED;
    }
    break;
  case CORNER_IN:
    if (time() - last_time < 0.1)
    {
      motor(BRAKE, BRAKE, BRAKE, BRAKE);
    }
    else
    {
      motor(SLOW, SLOW, SLOW, SLOW);
    }

    handle(LOOSE * line);
    if (time() - last_time > 0.2 && (left_change > 3 || right_change > 3))
    {
      last_time = time();
      state = CORNER_OUT;
      printf("CORNER OUT\n");
    }
    break;
  case CORNER_OUT:
    motor(MEDIUM, MEDIUM, MEDIUM, MEDIUM);
    handle(STRICT * line);
    if (time() - last_time > 0.70)
    {
      last_time = time();
      state = FOLLOW;
      printf("FOLLOW\n");
    }
    break;
  case RIGHT_CHANGE_DETECTED:
    motor(MEDIUM, MEDIUM, MEDIUM, MEDIUM);
    handle(LOOSE * line);
    if (double_line == 0)
    {
      last_time = time();
      state = RIGHT_TURN;
      printf("RIGHT_TURN\n");
    }
    if (time() - last_time > 2.5)
    {
      last_time = time();
      state = FOLLOW;
      printf("FOLLOW\n");
      }
      break;
    case RIGHT_TURN:
      motor(MEDIUM, MEDIUM, MEDIUM, MEDIUM);
      if (time() - last_time < 0.45)
        handle(-40);
      else if (time() - last_time < 0.6)
        handle(20);
      else if (double_line > 1)
      {
        last_time = time();
        state = FOLLOW;
        printf("FOLLOW\n");
      }
      break;
    case LEFT_CHANGE_DETECTED:
      motor(MEDIUM, MEDIUM, MEDIUM, MEDIUM);
      handle(LOOSE * line);
      if (double_line == 0)
      {
        last_time = time();
        state = LEFT_TURN;
        printf("LEFT TURN\n");
      }
      if (time() - last_time > 2.5)
      {
        last_time = time();
        state = FOLLOW;
        printf("FOLLOW\n");
      }
      break;
    case LEFT_TURN:
      motor(MEDIUM, MEDIUM, MEDIUM, MEDIUM);
      if (time() - last_time < 0.45)
        handle(40);
      else if (time() - last_time < 0.6)
        handle(-20);
      else if (double_line > 1)
      {
        last_time = time();
        state = FOLLOW;
        printf("FOLLOW\n");
      }
      break;
    case RAMP_UP:
      motor(UPRAMP, UPRAMP, UPRAMP, UPRAMP);
      handle(STRICT * line);
      if (angles[1] < 0.05)
      {
        last_time = time();
        state = FOLLOW;
        printf("FOLLOW\n");
      }
      break;
    case RAMP_DOWN:
      motor(SLOW, SLOW, SLOW, SLOW);
      handle(STRICT * line);
      if (angles[1] > -0.05)
      {
        last_time = time();
        state = FOLLOW;
        printf("FOLLOW\n");
      }
      break;
    }
  };

  wb_robot_cleanup(); // this call is required for WeBots cleanup

  return 0;
}
```

Manuals

Tuning the MCU car

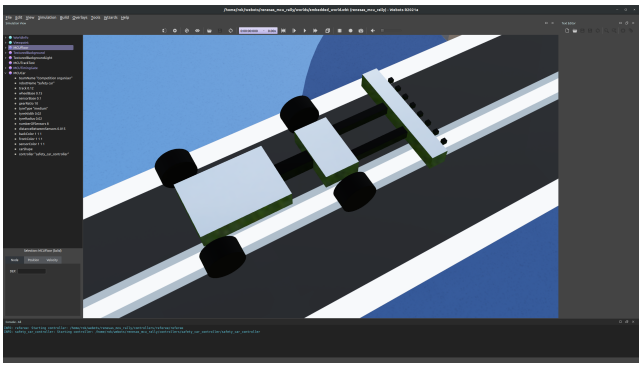
The settings

Check the parameters of the MCUCar asset and change them to see their effect. ***It is recommended to save the world and reload it after changing the settings!*** The settings include:

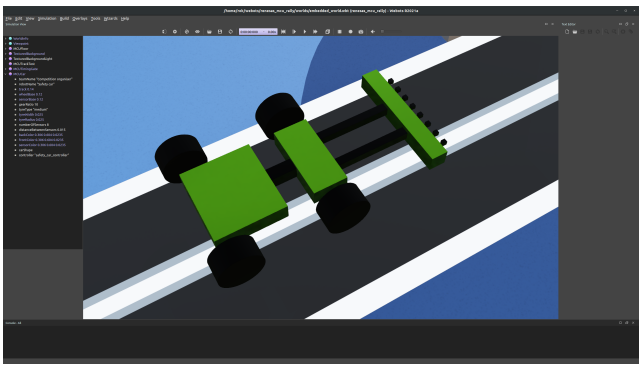
- wheel track (distance between left and right wheels)
- wheel base (distance between front and back wheels)
- sensor base (distance from base to sensor)
- gear ratio (motor gear ratio, higher means more torque but lower speed)
- tyre type (soft - sticky, but deteriorates fast, medium, and hard - slippy, but deteriorates slowly)
- tyre width
- tyre radius
- number of sensors
- distance between the sensors
- colors of the car parts
- car shape (3D Shapes can be added to the car to make it look better - we might have a separate competition for that ;))
- controller (your car controller, make sure to point it to your controller program!)

Make sure to input your team name and the robot name into the corresponding fields!

Example



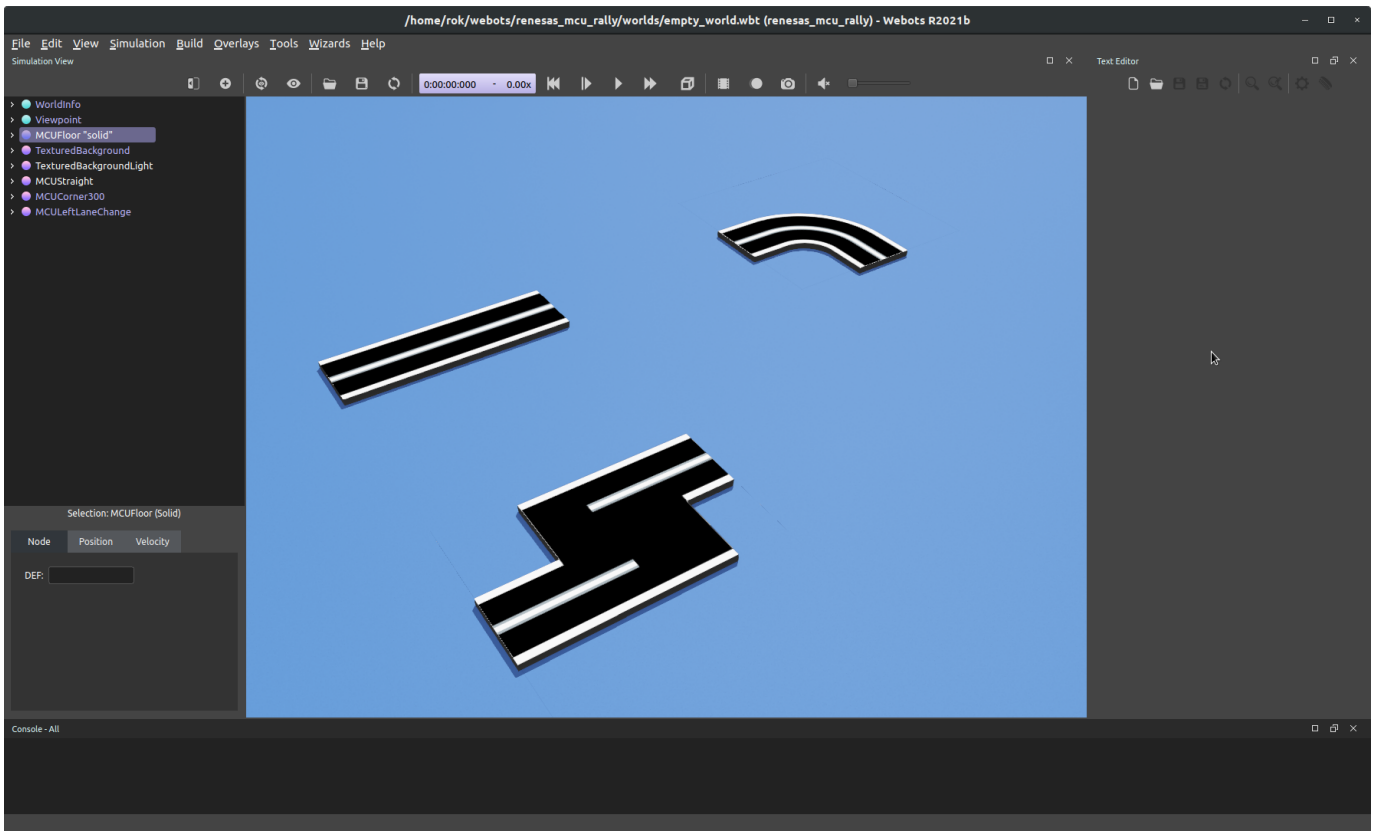
Default car settings.



An example of modified car settings.

Building tracks

Custom tracks can be built by using the provided assets.



Example use of track assets.

The assets contain tiles (1x1 m) that can be easily assembled into new configurations.

Good practice

It is recommended to use a Group node (found under Base nodes) as the root node when building tracks. To add individual assets, Add Node, then navigate to PROTO nodes (Current Project) -> track_parts.

To save a track for later use, check the PROTO manuals in the WeBots documentation. Saving custom PROTO nodes involves opening the world definition in a text editor, copying the part with the track Group node to a .proto file, and placing the proto file into the protos folder of the project.

Competition

Regulations

Important dates

Registration deadline is Sunday, October 31, 2021.

The qualifying round will be held on Monday, January 24, 2021 (file submission by Thursday, January 20, midnight CET).

The finals will be held on Thursday, January 27, 2021 (updated file submission by Tuesday, January 25, midnight CET).

Competition structure

The competition will consist of a qualifying round and the finals. The top 16 teams will advance to the finals.

Each round will take place in 3 parts: qualifying, sprint race, and feature race. For each part, a different track will be used.

Qualifying

- 1 lap
- track: to be determined
- random starting order (no penalties)
- any car setup / any controller
- up to 10 points for first 10

Sprint race

- 3 laps
- track: to be determined
- reverse grid starting order
- any tyres, but everything else must remain the same / any controller
- up to 15 points for first 10

Feature race

- 5 laps
- track: unknown upfront
- starting order based on qualifying
- same setup as qualifying, including tyres / any controller
- up to 25 points for first 10

Specific rules

Starting order penalty

The starting order, determined during the qualifying session, will influence the sprint race and the feature race. The robot starting first receives no penalty, while for every spot up to place 10, an additional weight of 0.1 kg is added to the robot, up to 0.9 kg total. Places 11+ will receive 1 kg weight penalty.

Design

Each robot must have a distinct shape, combined from added Transforms and Shapes to carShape and carShapeFront fields.

Point system

Points will be awarded as follows:

- Qualifying: 10, 8, 6, 4, 2, 1
- Spint race: 15, 12, 10, 8, 6, 4, 2, 1
- Feature race: 25, 18, 15, 12, 10, 8, 6, 4, 2, 1

Guide for participants

File submission

Send the following files to the competition organiser (email will be provided to the registered competitors):

- The robot definition file (.wbo).
- The controller source (single .c file).

Saving the robot definition file

Right click on your MCUCar asset in the scene tree -> Export, and save the .wbo file. Send the .wbo file.

Exporting the controller source

The controller source is available in the project folder under controllers/your_controller_name. Send only the corresponding .c file.