Model-driven design of embedded and real-time systems

Exercise 14 – Generating code for the robot and uploading it there

Copy the mdders/examples/lab4 folder into your workspace. Open the file TrackerPlant.slx. This contains a simple simulation of the robot on the red-line. However, the controller for the robot is contained in the referenced model (TrackerNXT). If you launch the simulation, it will use TrackerPlant for simulating the environment and TrackerNXT for simulating the robot controller.

Open the TrackerNXT file. It is made in such a way so that code for the robot can be generated from it. On the top-level, it contains special blocks which determine what are the devices used by the robot controller. Furthermore, it contains the Controller subsystem that implements the control logic. In simulation, the control subsystem is called according to the settings of the Simulink solver (in s); at runtime, the generated code executes with the period set in the Simulation > Model Configuration Parameters > Code Generation > OIL generation > Cycle time (in ms). Make sure the two values correspond to each other.

To see the implementation of the controller, double-click on Controller subsystem.

You can see that the TrackerNXT model contains special blocks to read the motor encoders, color sensor and it writes to motors and display. If you go a step further and open the Controller subsystem, you can see that it just simply applies power to the motors. Thus, the Controller subsystem is where you are later going to insert the control logic. (However, not for now.)

The aim of this task is to generate code for the robot and upload it into the robot. Before generating the code, make sure you use only integer arithmetic, have only one sample time type in the TrackerNXT model (Display > Sample Time > Colors/Legend), and set the right cycle time in the OIL configuration in the model parameters.

To generate the code, press the ‘Build Model’ (or Code > C/C++ Code > Build Model) button in the TrackerNXT model (doing this for the TrackerPlant would result in errors). You should see that it starts generating the code and producing some messages on the MATLAB console. In fact, the whole model is turned into a function that is periodically called via a generated real-time tasks of the nxtOSEK OS.

It’s good to take a few moments and look on the sources it has generated in the TrackerNXT_nxt_rtw directory (the sources for the model – TrackerNXT.c, Makefile, and glue code that integrates it with the real-time operating system – nxt_main.c).

To compile the sources, either run the “./build” script from the lab4 directory, or run “make -f TrackerNXT.mk all” in the TrackerNXT_nxt_rtw directory (make sure you set your environment via “./setenv” beforehand).
Once the code is compiled and generated. You should upload the resulting image (main_rom.bin – stored in TrackerNXT_nxt_rtw directory) to the robot. This is done in the following steps:

1) Switch on the robot by pressing the orange button
2) Set it to upload mode by simultaneously pressing the orange and the left button until it switches off.
3) Connect it to the computer via the USB cable.
4) Switch on the robot again by pressing the orange button
5) Run the "./flash" script from the lab4 directory, or run "./appflash.sh main_rom.bin" in the TrackerNXT_nxt_rtw directory (make sure the generated appflash.sh has the execute permissions). The code should get uploaded into the robot, which is indicated by a progress bar.
6) Switch off the robot using the dark middle button (this and the next step might happen automatically).
7) Switch on the robot using the orange button.
8) Press the right button to start your tasks.
9) Have fun.

Exercise 15 – Stopping on the line

Change the controller subsystem in such a way that the robot stops on a red line. Note that for any computation you have to use integer/fixed-point arithmetic, which means types int8, int16, int32 for integer and fixdt(<signedness>, <total bit length>, <fraction bit length>) for fixed-point.

Test your program in the simulation first to see that your integer/fixed-point arithmetic works as expected.

Exercise 16 – Following the line

Change the controller subsystem in such a way that it drives the robot along the red line (reuse the model from exercise 13). Note that for any computation you have to use integer/fixed-point arithmetic, which means types int8, int16, int32 for integer and fixdt(<signedness>, <total bit length>, <fraction bit length>) for fixed-point.

Test your program in the simulation first to see that your integer/fixed-point arithmetic works as expected.

Test your program in the real environment.
Exercise 17 – Including Stateflow

Enhance your controller from Exercise 16 in such a way that when the robot encounters a red line, it goes for a short while in reverse and then with a lower speed towards to line in order not to miss it. Realize this logic using a Stateflow chart.

Here are some hints:

- You will need a couple of states (e.g. going forward with high speed, going backward, going forward low speed, following the line).
- Do not forget the entry to the initial state.
- To go back for a short while, you should use something as an event “after(80,tick)” on the transition (see Pulse generator Simulink block).
- To detect the line, you will need something like a guard “[redLevel >= 0.25]”.
- If needed, you may create the hierarchical transition system.
- Inputs, outputs and local data are set in the Model Explorer.
- Stateflow uses different syntax for expression (compared to Matlab). For example, array indexing is done via square brackets “a[0]”, with index starting from 0.
- Stateflow expressions do not support well the fixed-point arithmetics, thus:
  - Specify fixed-point types explicitly for input, output and local data in Model Explorer
  - Rather than using Stateflow expressions for computation, declare internal Simulink block, which you will call from Stateflow. The Simulink block allows you to specify fixed-point in the usual Simulink way.
- If you keep the Stateflow model opened during the simulation, you can see active state and transitions.