Simulink Code Generation Exercise
Model-Based Design Of Embedded Software

Magnus Lång

28 September 2020

1 Introduction

This lab will teach you how to generate C code from a Simulink model, how to interface with that model from C code that you write, and to compile the resulting program.

You will perform this lab on your own, but there will be time on Thursday 1:st of October for you to show me your solution, or get help if you're stuck.

These instructions is written for Linux. They should work for other operating systems, but we will not help you with any issues stemming from the use of other operating systems. Unless you are very familiar with writing and compiling C and C++ code, as well as with the use of CMake, on your OS of choice, we recommend you do the lab under Linux. If your computer does not have Linux installed, you may install it on a virtual machine (unless you have one already, of course). Ubuntu Linux is a safe choice. It is possible to generate code under windows/mac, and then bring the generated zip-file to your Linux installation.

2 Setting Up the Simulink Model

Create a simulink model, and name it "calculator"[1]. In order to configure the model for code generation, go into the Model Settings, under the "MODELLING" tab. There are a few settings you need to configure for code generation to work smoothly.

1. At the first tab, "Solver", change the type of solver to "Fixed-step".

   This has wide-ranging implications on how your simulink model is evaluated beyond just code generation, and is usually a bad idea if you are using any continuous-time features, but most code generation targets only support fixed-step solvers.

2. Expand the "Solver details" region on the same page, and change the "Fixed-step size" to 0.1.

   This is the number of seconds that should go between every evaluation of the model. If you were generating code for an embedded system, here you would enter your interrupt interval.

3. In "Code Generatrion" tab on the left, tick the check-box "Package code and artifacts".

   This useful feature, introduced very recently, makes the code generation process generate a zip-file of all the code needed to compile stand-alone, including any code files belonging to the Matlab installation.

4. Expanding "Code Generation" on the left, open the "Interface" tab on the left. At the very bottom of the tab, there are three dots ("..."), that turns into an expandable "Advanced parameters" if you hover them with your mouse. Expand that and untick the "MAT-file logging" tickbox.

   If left enabled, the generated code will attempt to write a file when simulated, and will make it more difficult to interface with and compile.

3 Make Your Simulink Model

For this lab, we will make a simple calculator, with a user interface in C and logic in Simulink/Stateflow. The calculator has two modes, sum and difference. If a number is input, it is either added to or subtracted from the current total. The user can also input "+" or "-", which will change the mode to sum and difference, respectively.

[1] Of course, code generation works for models of any name, but the name affects the names of generated code files and C functions and variables.

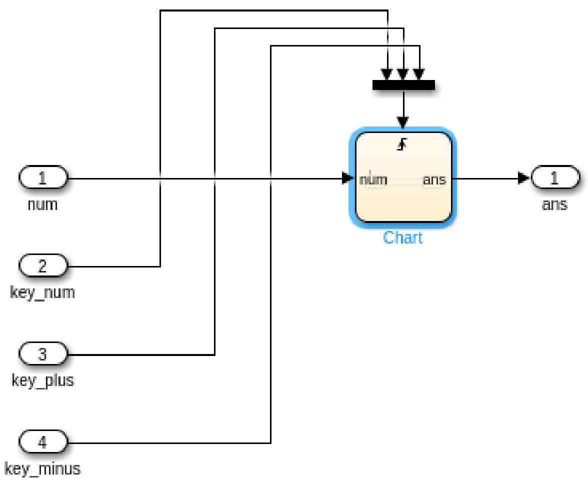

Figure 1: Example Simulink model

In order to accept input from the user interface, and output the total, we will use model inputs and outputs, just like on a subsystem. There will be four inputs in total, two for the "+" and "-" "keys", one for signalling that a number has been entered, and one to carry the number itself. The model will have one output.

Add four input blocks, "num", "key_num", "key_plus", and "key_minus". For the first block, set the "Data type" to "int32" in the "Signal Attributes" tab of the block parameters, accessible by double-clicking the block. For the other three, set the data type to "boolean". Add one output block, and make its type "int32".

Finally, add a stateflow chart to your model, and give it corresponding inputs and outputs. Make the "key_" inputs events, triggered on rising edge. You might want to make the initial value of the output "int32(0)". Your model will now look like in fig. 1.

Implement the logic of the calculator in the statechart. One way to do this is shown in fig. 2.

4 Generate C Code From Your Model

Now we're ready to generate code from your model. Under "APPS" in the top ribbon, expand the apps menu and select "Simulink Coder". In the new ribbon that appears, you have a big blue "Build" button. In the drop-down, choose "Generate Code"[2]. In the current working directory of Matlab, a bunch of files will be produced, including a file "calculator.zip". This file contains all the code for your model.

Make a new directory in which to work and unzip the file there, flattening directories (i.e., put every file in the same directory, with no subfolders. This helps us compile it later). On Linux you can do this from the terminal when you unzip. Run the following from your work directory:

```
$ unzip -j $PATH_TO_YOUR_MATLAB_WORKSPACE/calculator.zip
```

We can now inspect the generated code. Open `calculator.h` and `calculator.c` and familiarise yourself with the code that was generated. You don't need to understand all of it. Here's a section from `calculator.h`:

```
/* External inputs (root inport signals with default storage) */
typedef struct {
  int32_T num;                         /* '<Root>/num' */
```

[2]Clicking "Build" is also OK, but builds an executable we will not use.

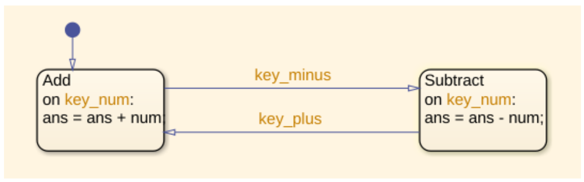

Figure 2: Example Stateflow Statechart

```c
  boolean_T key_num;                   /* '<Root>/key_num' */
  boolean_T key_plus;                  /* '<Root>/key_plus' */
  boolean_T key_minus;                 /* '<Root>/key_minus' */
} ExtU_calculator_T;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  int32_T ans;                         /* '<Root>/ans' */
} ExtY_calculator_T;

/* External inputs (root inport signals with default storage) */
extern ExtU_calculator_T calculator_U;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY_calculator_T calculator_Y;

/* Model entry point functions */
extern void calculator_initialize(void);
extern void calculator_step(void);
extern void calculator_terminate(void);
```

We can see that the interface consists of a struct of input values `calculator_U`, a struct of output values `calculator_Y`, as well as three functions `calculator_initialise()`, `calculator_step()`, and `calculator_terminate()`.

Predictably, to use the model you're supposed to fist call `initialise()`, then for some number of steps set the inputs in `_U`, call `step()`, and read outputs from `_Y`. Then, when you're done, call `terminate()`.

5 Writing our own C code

You might have noticed that the Simulink generated a file `rt_main.c`. It is not useful to us, as we will write our own main-file, so delete it. Then, create `main.c` starting from this skeleton:

```c
#include "calculator.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  // Your code here
}
```

Start by adding calls to `calculator_initialise()`, and `calculator_terminate()`. You should also call `calculator_step(`
once immediately after `initialise()`, because the model will not recognise a rising edge in the first step.

Then add a `while(1)` loop for the main logic. First, you should read a line of input from the user with something like

```c
    /* Read input */
    char buffer[128];
    char *ret = fgets(buffer, sizeof(buffer), stdin);
    if (ret == NULL || buffer[0] == '\0') {
        break;
    }
```

Then you can check for the special input "+" with a line like

```c
    if (strcmp(buffer, "+\n") == 0) {
```

And analogously for "-". You can get whatever number the user has entered using `atoi(buffer)`.

Set the inputs in `calculator_U`, and then call `step()`. You should reset the `calculator_U.key_` inputs to false and call `step()` again to send a full pulse. If you don't then the next time you set one of the `calculator_U.key_`'s to true, then the model would not recognise a rising edge (since the signal was already high).

Now, after running the model for two steps, print out the output of the model using `printf`.

Finally, you can compile your model by typing in the shell:

```
$ gcc main.c calculator.c
```

6 Conclusions

Now you should have a working calculator. Run it with

```
$ ./a.out
```

If it's all working to add up numbers, and to change modes between addition and subtraction, you're done! You have now used a simulink model from C code!