Tutorial of the C code, dll and Sfunction generation

The first step of Autocoding is to create the model you want to autocode. To show how it is done, we will use a first order discrete model.

I – Creation of the model in Simulink

As you can on the figure below, the model is composed by one input (input1) and one output (output1)

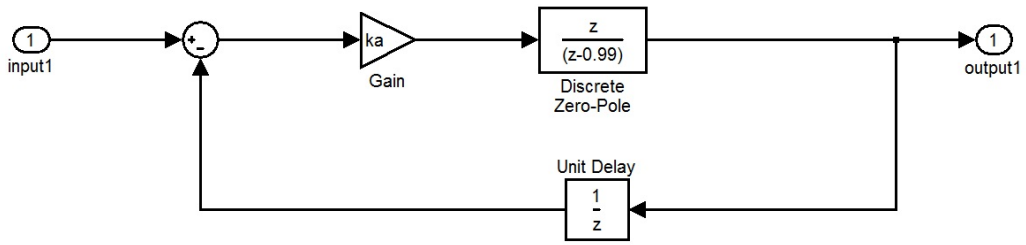

Figure 1: First order model

The system must be discrete to generate a C code in order to be implemented on the microcontroller. We choose a sample period of Te = 10^{-4}s.

II - Generation of the different codes

The approach to generate codes of the Simulink model is the following.

1) Choice of the solver

The solver that must be used is a discrete solver.

Go in Simulation/Configuration parameter -> Solver-> Solver option

type: fixed_step

solver: discrete no continuous state

fixed step size: Te

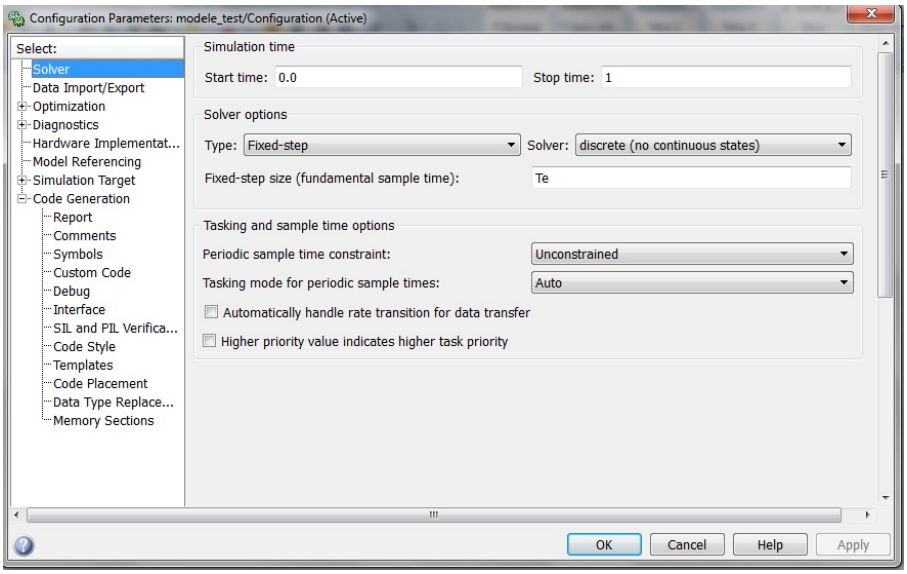

Figure 2: Solver Configurations

2) Configuration of the inputs/outputs

At this step you have two choices.

- If the model's inputs you are working on will be set directly in the code by another function or program for example and if the outputs will not be saved in the Matlab workspace, then go to the next step Choice of the tunable parameters.
- If you are working on Matlab through the workspace, you need to specify the data.

To do so, go in **Simulation/Configuration parameter** -> Data Import/Export

Tick Input and enter the name of your inputs [t,input1] (t is present by default)

Tick output and enter [yout]

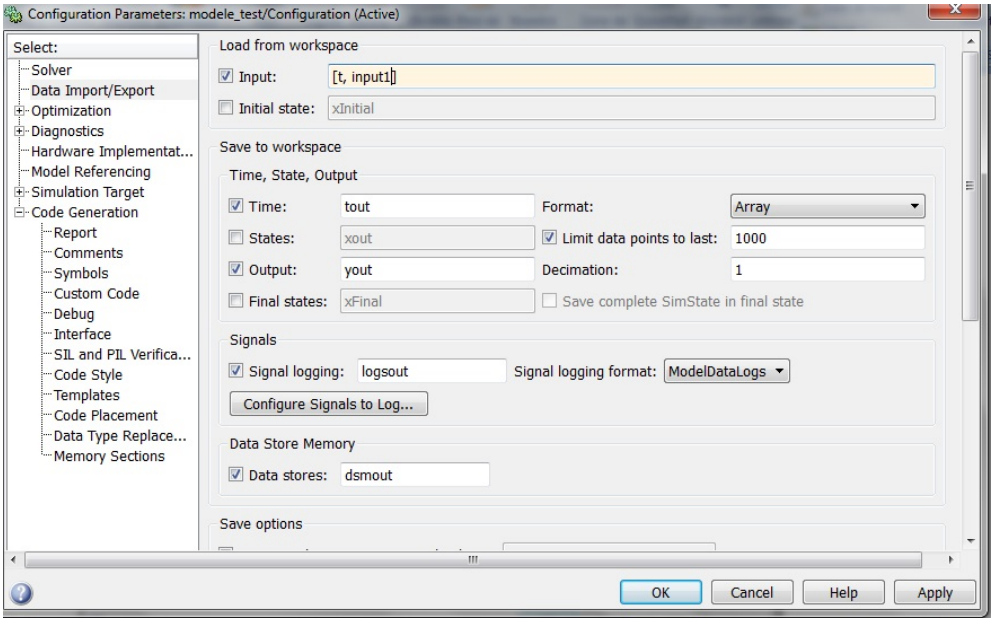

Figure 3:Inputs and Outputs specification

3) Choice of the tunable parameters

The inline parameters (parameters used in the Simulink model) are configured in this section.

Go on Simulation/Configuration parameter /Optimization/Signal and parameters -> configure

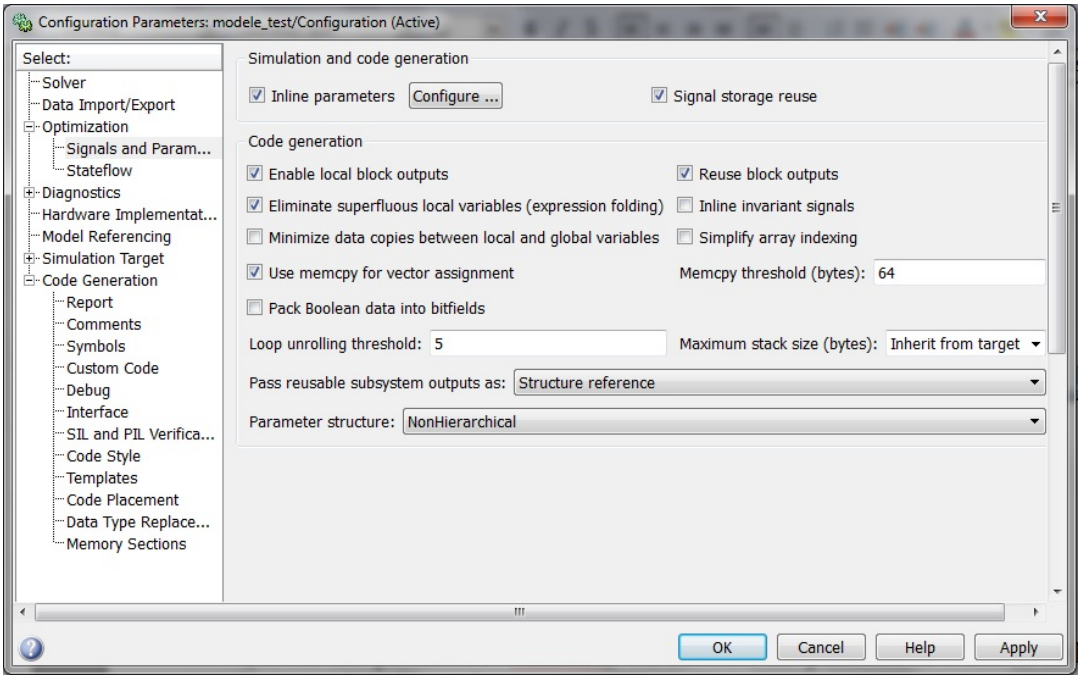

Figure 4: Inline parameters configuration

First create in the workspace the variables you want to add by running your script. For our example, the Inline parameters are Te and ka.
Select the variable to add in the left side and click on **add to the table**.

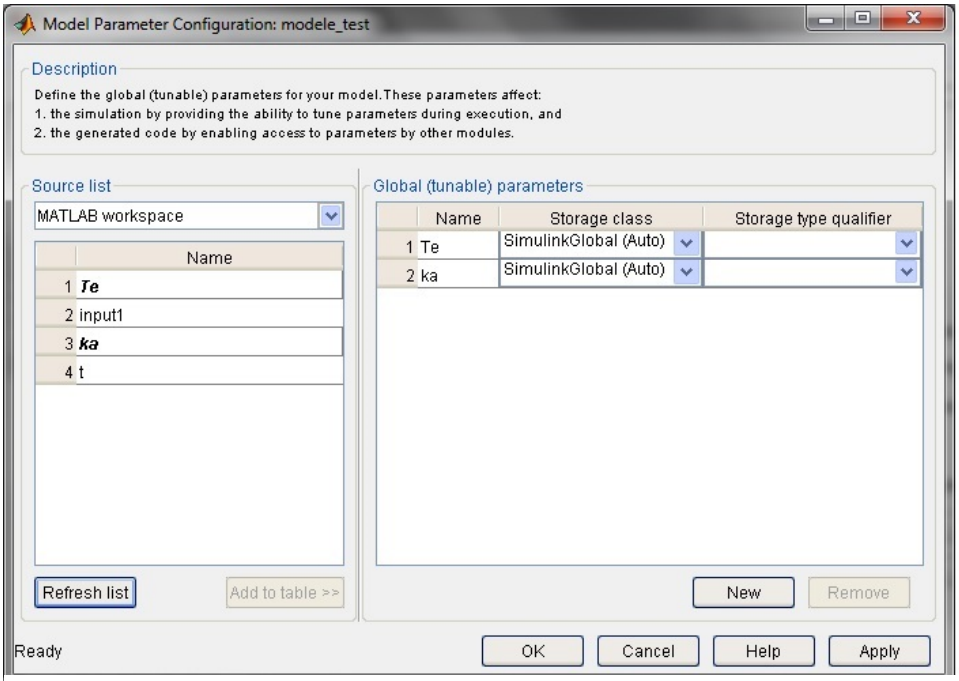

Figure 5: Adding the Inline parameters

4) Configuration of the code generation

Different configuration for the code generation can be implemented. You have to go in **Simulation/Configuration parameter** and click on **code generation**. Then you just have to choose the **system target file** according to what you want.

Create directly an S_function is useless for us because we want an approach step by step as it is shown below:

1 - Create the C code
2 - Create the dll
3 - Create the S_function with the C code and the dll

To create the C codes go in **Simulation/Configuration parameter** -> code generation
Then click on browse and select the option ert.tlc (to create a C++ code).
Be careful, the Language C doesn't work because C++ library are used by default.
So you have to choose the C++ language "ert.tlc Embedded Coder".

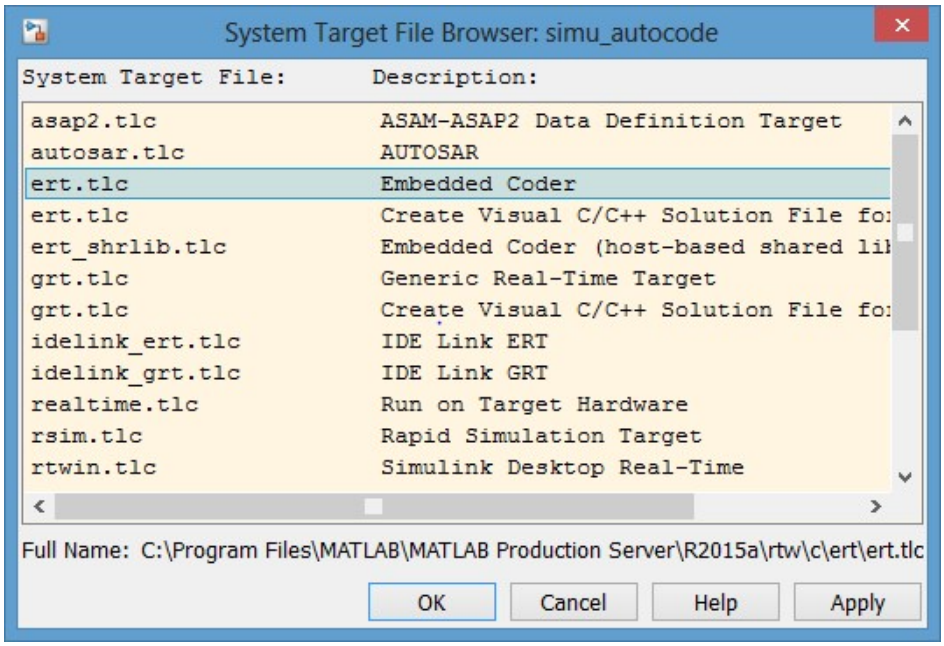

Figure 6: Code generation target file

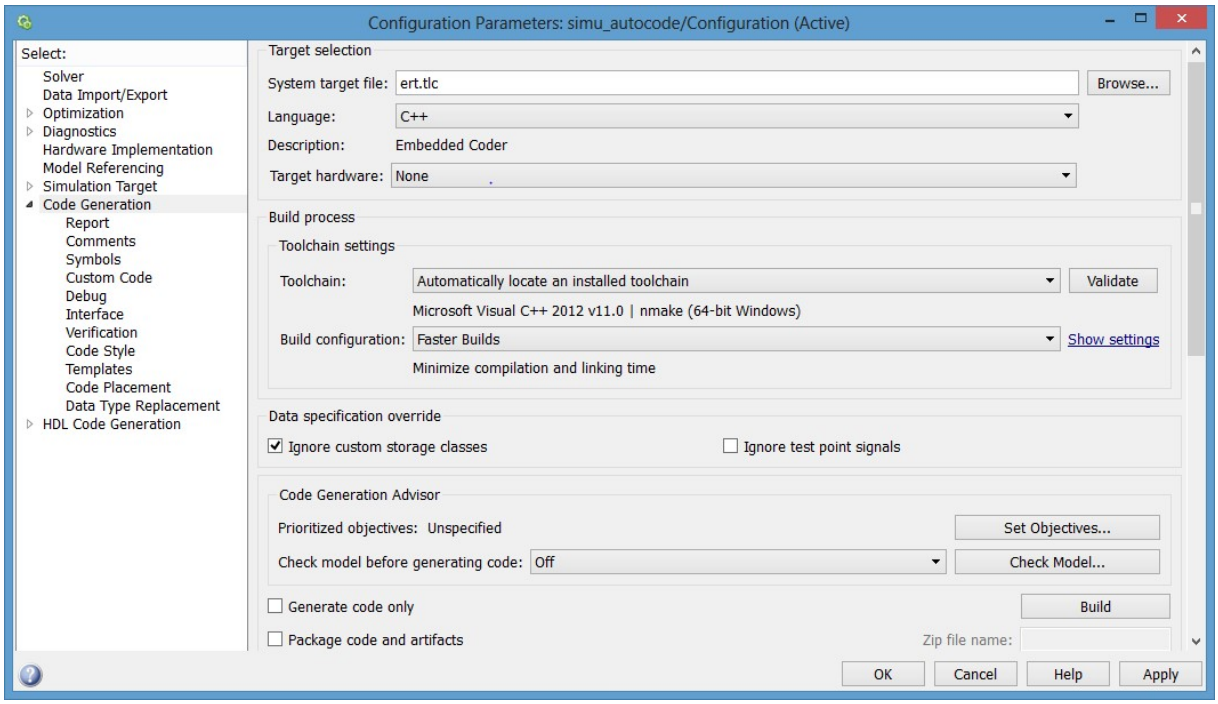

Figure 7: Code Generation Parameters

Then click on Build. The code generation is done.

If you want a better readability you can chose the following option:
In **Simulation/Configuration parameter** -> code generation -> Interface Software environment: Select Code Replace Library C89/C90 (ANSI)

In **Simulation/Configuration parameter** -> code generation -> ode style Parentheses level: Nominal optimize for readability

You may encounter a failure while building the Code because of the GNU Compiler of Matlab which may not be installed on your computer. If so, install it and restart the Autocoding procedure.

III - Analysis of the codes generated

For a future work, an analysis of a C++ code is necessary. The analysis may include how the file (.cpp and .h) are connected.

After the *build* you have :
2 files .c pp
ert_main.cpp
your_name.cpp (your_name is the name of the model you just autocoded)

3 files .h
your_name.h
your_name_private.h
your_name_types.h

If you choose some tunable parameters 1 file .cpp and 1 file .h will be generated.

To open the code and test it, you need to have a C++ software. We recommend “Code Blocks” which is free to download.
Firstly you have to create a new project on Code Blocks or any C++ software.
Here is the example on Code Blocks

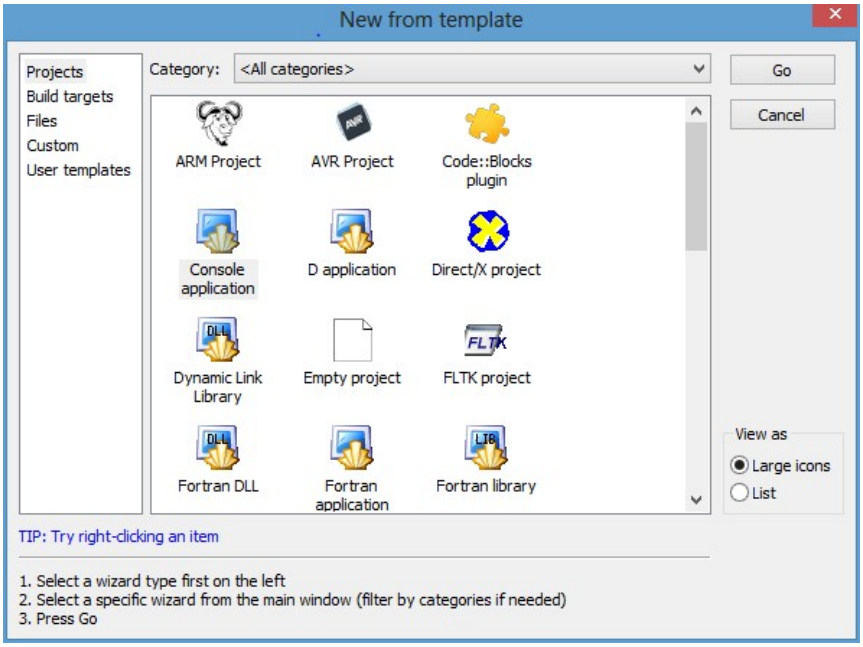

Figure 8: Create a C++ project

Choose the “Console application” and C++ Langage. You will have to name your project and indicate the directory.
Finally, you have to add the 5 files (.cpp and .h) generated by the Autocoding in this project as shown below :

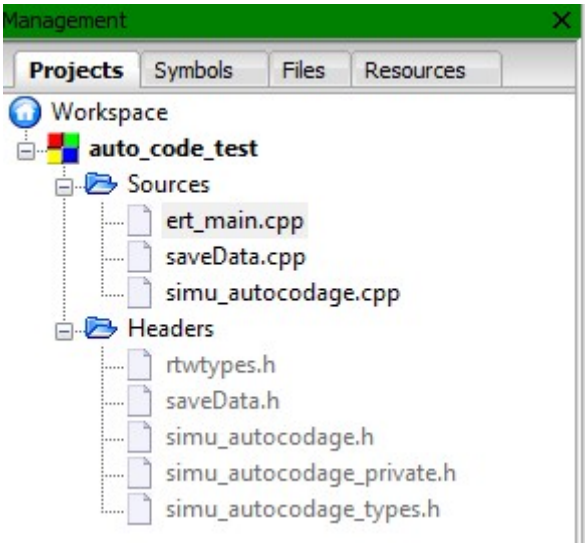

Figure 9: The C++ project with all the files

The files .h contain the structures and the different names present in the Simulink model. The file ert_main.cpp contains an example of main in order to use the model.
A function one_step which created the output at time i considering the input and the output at time i-n (n depends on the order and the complexity of the system).

You have to choose your input value and create a loop *for* in order to create the output (a function saveData can be implemented by the user to save the output, see code in annex1)

So you are going to add some code in order to create the input and save the corresponding output. The red lines is the ones added by the user

```
/*
 * File: ert_main.cpp
 *
 * Code generated for Simulink model 'Modele_1er_ordre'.
 * Model version                  : 1.14
 * Simulink Coder version         : 8.5 (R2013b) 08-Aug-2013
 * C/C++ source code generated on : Mon Feb 16 16:46:06 2015
 * * Target selection: ert.tlc
 * Embedded hardware selection: 32-bit Generic
 * Code generation objectives: Unspecified
 * Validation result: Not run
 */

#include <stdio.h>                /* This ert_main.c example uses printf/fflush */
#include "Modele_1er_ordre.h"         /* Model's header file */
#include "saveData.h"  /* Be careful you have to create saveData.h and saveData.cpp, you can find them in annex1*/
#include "rtwtypes.h"

/*
 * Associating rt_OneStep with a real-time clock or interrupt service routine
 * is what makes the generated code "real-time".  The function rt_OneStep is
 * always associated with the base rate of the model.  Subrates are managed
 * by the base rate from inside the generated code.  Enabling/disabling
 * interrupts and floating point context switches are target specific.  This
 * example code indicates where these should take place relative to executing
 * the generated code step function.  Overrun behavior should be tailored to
 * your application needs.  This example simply sets an error status in the
 * real-time model and returns from rt_OneStep.
 */
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = 0;

  /* Disable interrupts here */

  /* Check for overrun */
  if (OverrunFlag) {
    rtmSetErrorStatus(Modele_1er_ordre_M, "Overrun");
    return;
  }

  OverrunFlag = TRUE;

  /* Save FPU context here (if necessary) */
  /* Re-enable timer or interrupt here */
```

```
  /* Step the model */
  Modele_1er_ordre_step();

  /* Get model outputs here */

  /* Indicate task complete */
  OverrunFlag = FALSE;

  /* Disable interrupts here */
  /* Restore FPU context here (if necessary) */
  /* Enable interrupts here */
}
/*
 * The example "main" function illustrates what is required by your
 * application code to initialize, execute, and terminate the generated code.
 * Attaching rt_OneStep to a real-time clock is target specific.  This example
 * illustates how you do this relative to initializing the model.
 */
int_T main(int_T argc, const char *argv[])
{
  /* Unused arguments */
  (void)(argc);
  (void)(argv);

  /* Initialize model */
  Modele_1er_ordre_initialize();

  /* Attach rt_OneStep to a timer or interrupt service routine with
   * period 0.0001 seconds (the model's base sample time) here.  The
   * call syntax for rt_OneStep is
   */

  const int taille = 150;   //length for the output and input vectors you can choose what you want
  Modele_1er_ordre_U.input_C=1.0;
  printf("Simulation du modele pour une entree = %f\n",Modele_1er_ordre_U.input_C);

//creation of output and input vectors
double out_tab[taille];
 double inp_tab[taille];

//filling of vectors
  for(int i=0 ; i<taille ;i++){
              rt_OneStep();
              inp_tab[i]=Modele_1er_ordre_U.input_C;
              out_tab[i]=Modele_1er_ordre_Y.ouput_simu;
  }

  printf("Modele_1er_ordre_Y.ouput = %f\n",Modele_1er_ordre_Y.ouput_simu);


  saveData(taille,inp_tab,"input_C.txt");
  saveData(taille,out_tab,"output_C.txt");
```

```
 fflush((NULL));
 while (rtmGetErrorStatus(Modele_1er_ordre_M) == (NULL)) {
   /*  Perform other application tasks here */
 }

 /* Disable rt_OneStep() here */

 /* Terminate model */
 Modele_1er_ordre_terminate();
 return 0;
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

The file your_name.cpp contains the parameters, inputs and outputs of your system
There are 2 functions:
your_name_step() does a step on the system
your_name_initialize() initializes the inputs and outputs of the system (0 by default)

```
/*
 * File: Modele_1er_ordre.cpp
 *
 * Code generated for Simulink model 'Modele_1er_ordre'.
 *
 * Model version                  : 1.14
 * Simulink Coder version         : 8.5 (R2013b) 08-Aug-2013
 * C/C++ source code generated on : Mon Feb 16 16:46:06 2015
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: 32-bit Generic
 * Code generation objectives: Unspecified
 * Validation result: Not run
 */

#include "Modele_1er_ordre.h"
#include "Modele_1er_ordre_private.h"

/* Block states (auto storage) */
DW_Modele_1er_ordre_T Modele_1er_ordre_DW;

/* External inputs (root inport signals with auto storage) */
ExtU_Modele_1er_ordre_T Modele_1er_ordre_U;

/* External outputs (root outports fed by signals with auto storage) */
ExtY_Modele_1er_ordre_T Modele_1er_ordre_Y;
```

```
/* Real-time model */
RT_MODEL_Modele_1er_ordre_T Modele_1er_ordre_M_;
RT_MODEL_Modele_1er_ordre_T *const Modele_1er_ordre_M = &Modele_1er_ordre_M_;

/* Model step function */
void Modele_1er_ordre_step(void)
{
 real_T rtb_DiscreteTransferFcn;

 /* DiscreteTransferFcn: '<S1>/Discrete Transfer Fcn' */
 rtb_DiscreteTransferFcn = Modele_1er_ordre_P.DiscreteTransferFcn_NumCoef *
  Modele_1er_ordre_DW.DiscreteTransferFcn_states;

 /* Outport: '<Root>/ouput_simu' */
 Modele_1er_ordre_Y.ouput_simu = rtb_DiscreteTransferFcn;

 /* Update for DiscreteTransferFcn: '<S1>/Discrete Transfer Fcn' incorporates:
  *  Update for Inport: '<Root>/input_C'
  *  Sum: '<S1>/Add'
  *  UnitDelay: '<S1>/Unit Delay'
  */

/*Equation of the system*/
 /* Update for UnitDelay: '<S1>/Unit Delay' */
 Modele_1er_ordre_DW.UnitDelay_DSTATE = rtb_DiscreteTransferFcn;
}

/* Model initialize function */
void Modele_1er_ordre_initialize(void)
{
 /* Registration code */

 /* initialize error status */
 rtmSetErrorStatus(Modele_1er_ordre_M, (NULL));

 /* states (dwork) */
 (void) memset((void *)&Modele_1er_ordre_DW, 0,
           sizeof(DW_Modele_1er_ordre_T));

 /* external inputs */
 Modele_1er_ordre_U.input_C = 0.0;

 /* external outputs */
 Modele_1er_ordre_Y.ouput_simu = 0.0;

 /* InitializeConditions for DiscreteTransferFcn: '<S1>/Discrete Transfer Fcn' */
 Modele_1er_ordre_DW.DiscreteTransferFcn_states =
  Modele_1er_ordre_P.DiscreteTransferFcn_InitialStat;

 /* InitializeConditions for UnitDelay: '<S1>/Unit Delay' */
 Modele_1er_ordre_DW.UnitDelay_DSTATE =
  Modele_1er_ordre_P.UnitDelay_InitialCondition;
```

```
}

/* Model terminate function */
void Modele_1er_ordre_terminate(void)
{
  /* (no terminate code required) */
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

All the parameters are declared as structures in the .h.

Example:

```
  typedef struct {
  real_T input_C;              /* '<Root>/input_C' */   (name of your input in your simulink model)
} ExtU_Modele_1er_ordre_T;

/* External outputs (root outports fed by signals with auto storage) */
typedef struct {
  real_T ouput_simu;        /* '<Root>/ouput_simu' */ (name of your output in your simulink model)
} ExtY_Modele_1er_ordre_T;


/* External inputs (root inport signals with auto storage) */
  extern ExtU_Modele_1er_ordre_T Modele_1er_ordre_U;

  /* External outputs (root outports fed by signals with auto storage) */
  extern ExtY_Modele_1er_ordre_T Modele_1er_ordre_Y;
```

IV- Test

To see if our Code Generation was correct, we compared the output given by the code and the one given by the Simulink model.
To do so the input and the output generated by the C++ code are stored in 2 text files input_C.txt and output_C.txt using the function saveData().
We wrote a small script which:

- Load the input_C and the output_C in the script.
- Set an input model at input=1
- Simulate the system with this input and plot the 2 outputs (with the C++ Code and with the Simulink).

Here is the result.

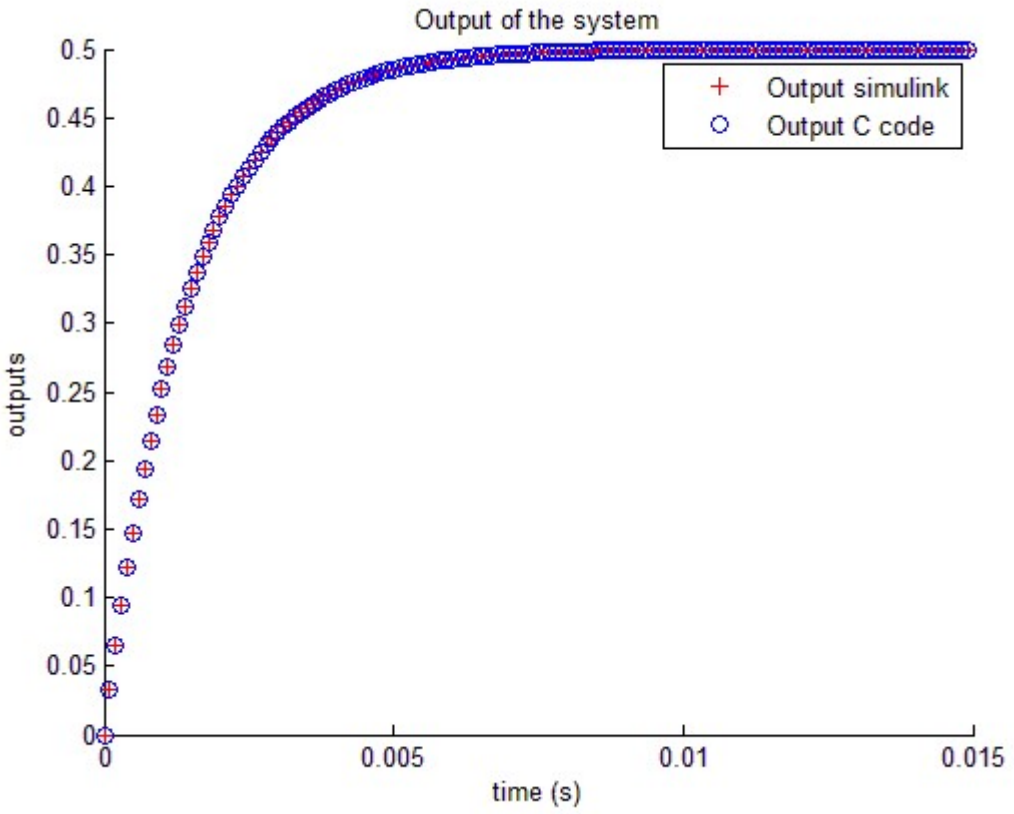

Figure 10: Comparison between the model and the C++ code

The values of the outputs generated by the C code and by Simulink are perfectly matching. But some differences can be found at the 5th decimal.

Annex

saveData.cpp

```
#include "saveData.h"
#include <iostream>     //Pour afficher les messages (cout)
#include <fstream>      //Pour la lecture des fichier en C++
using namespace std;

float *saveData(int taille ,double *sum_save,string filename_save) {

ofstream f(filename_save.c_str());

if(f) {
        for (int i=0 ; i<taille ; i++){
        f << sum_save[i] << endl;
        }
        }
return 0;
}
```

saveData.h

```
#ifndef SAVEDATA_H
#define SAVEDATA_H
#include <string>
#include <iostream>     //Pour afficher les messages (cout)
#include <fstream>      //Pour la lecture des fichier en C++

using namespace std;

float *saveData(int taille ,double *sum_save,string filename_save);

#endif
```