From Simulink to C and Python

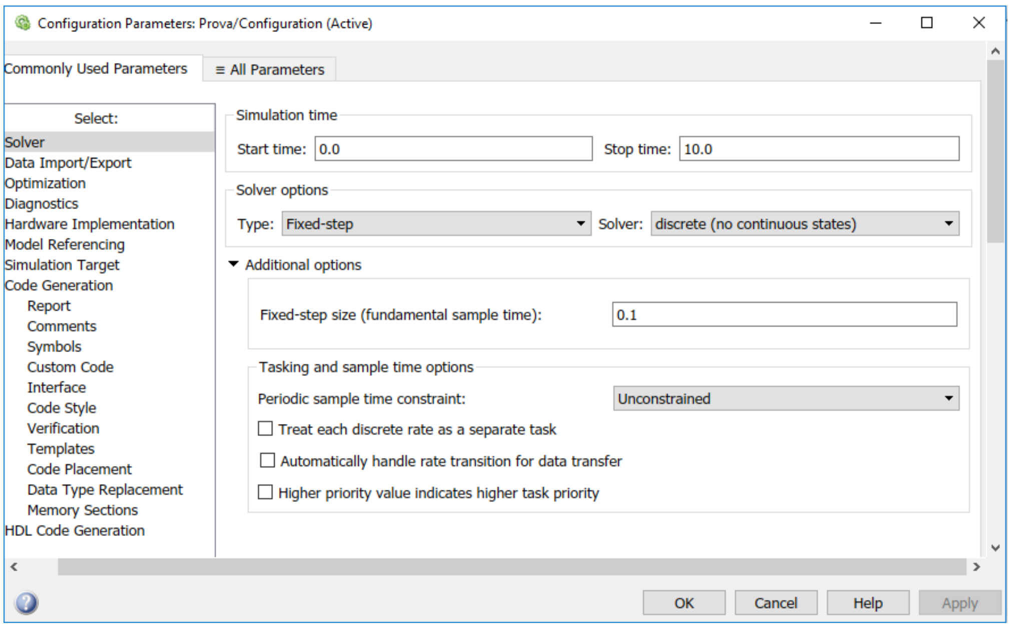
This is a Tutorial to obtain executable C and Python code, simulating an arbitrary Simulink model. This is possible using the Code Generation tools in Simulink and the Python “ctypes” library. This text assumes you are familiar with Simulink and have elementary C and Python notions.

Simulink Model and Code Generation

Firstly, the Simulink Model needs to be adapted in order to obtain an understandable C code that can be easily adapted across platforms. The code generation options in Simulink allow for many different solvers to be used and Simulink will still be able to generate the code. Keep in mind that having a continuous time solver in the Configuration Parameters will eventually lead the C code to include the variable step and solver functions. This will result in more and larger .c files, and might lead to a drop in readability and performance (execution time) when executed from some other programming language.

This section is oriented to give the steps for model in a **discrete time simulation**.

1. **Simplification of the model.** You should have a clear view of what part of your model is going to be necessary for your simulations, and you should work towards having only the absolutely necessary for your goals. The right inputs, outputs and integrators/state-holders. They should also be properly named, as the code will keep your notation and this will make the code more readable.
2. **Discretisation of the model.** Go to the Configuration Parameters Menu, change the solver to a discrete time solver and <u>choose the adequate time step</u>. You’ll need a time step that is small enough to grasp the dynamics of the system at hand, for a given final application. To do that, you’ll need to transform any integrator or continuous-time transfer function into its discrete equivalent and make them inherit their time step from the solver in their own options (-1). Any slower sampling times (measures, controller actions) can be handled later, based on the fundamental time step.



Simulation time

Start time: 0.0 Stop time: 10.0

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Additional options

Fixed-step size (fundamental sample time): 0.1

Tasking and sample time options

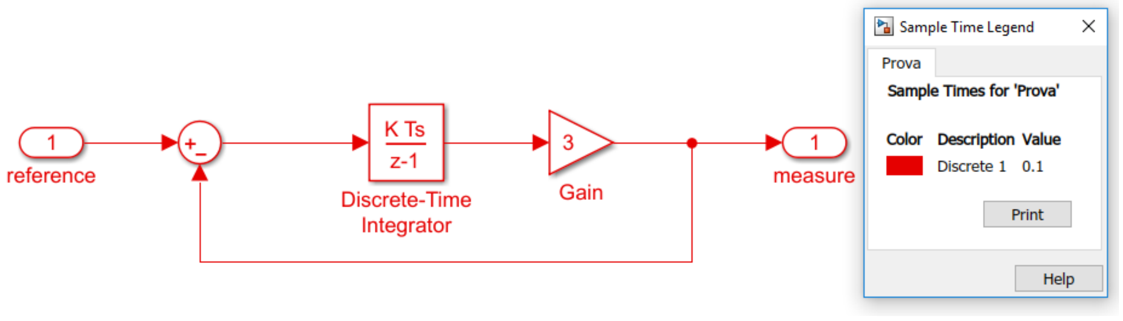
Periodic sample time constraint: Unconstrained

☐ Treat each discrete rate as a separate task

☐ Automatically handle rate transition for data transfer

☐ Higher priority value indicates higher task priority

Before generating the code, you can check sample times by going to Display>Sample Times>Colors. They should all be the same colour.



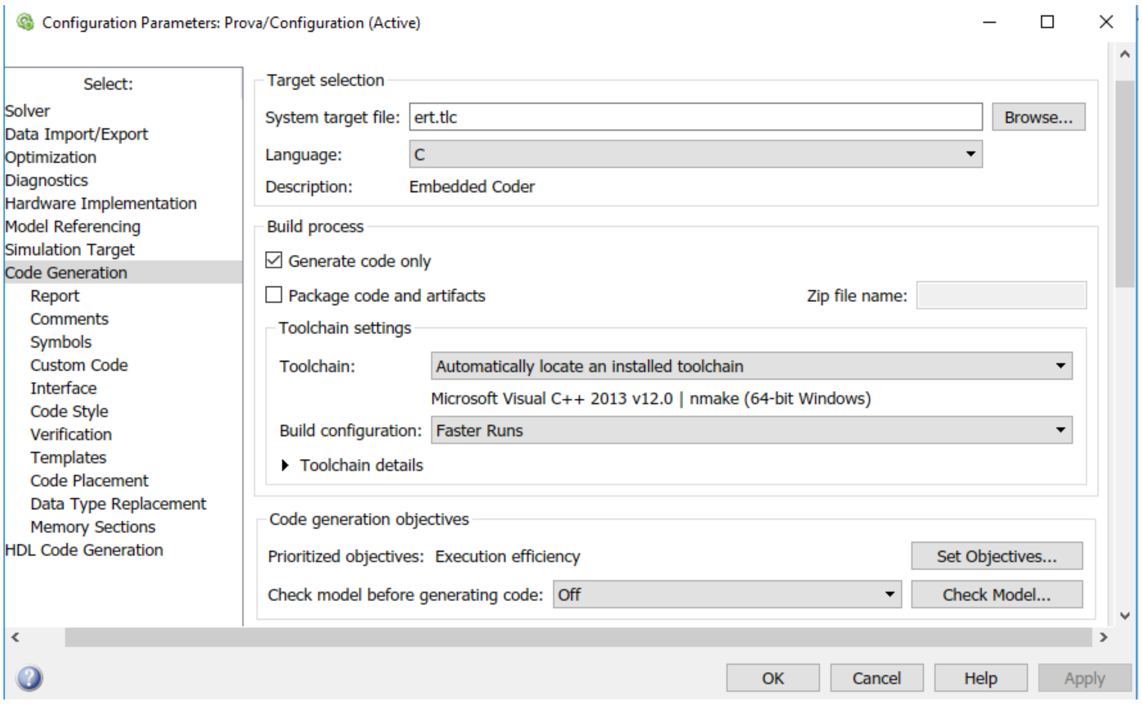
Sample Time Legend

Prova

Sample Times for 'Prova'

Color	Description	Value
🟥	Discrete 1	0.1

3. **Code Generation Options.** There are plenty of options throughout this process, and it is important not to change options from which we don't understand the effects. Before generating the code we have to go through the options menu.
   a. **System target file:** Choosing "**ert.tlc**" will generate an understandable code, that will allow us to pass parameters to specific Simulink functions, and this tutorial will follow with this type of generation in mind.
   You can either keep or change the build configuration, objectives, and check the model. Checking the model will help you working towards the Prioritized objectives.


Configuration Parameters: Prova/Configuration (Active)

Target selection

System target file: ert.tlc

Language: C

Description: Embedded Coder

Build process

☑ Generate code only

☐ Package code and artifacts   Zip file name:

Toolchain settings

Toolchain: Automatically locate an installed toolchain

Microsoft Visual C++ 2013 v12.0 | nmake (64-bit Windows)

Build configuration: Faster Runs

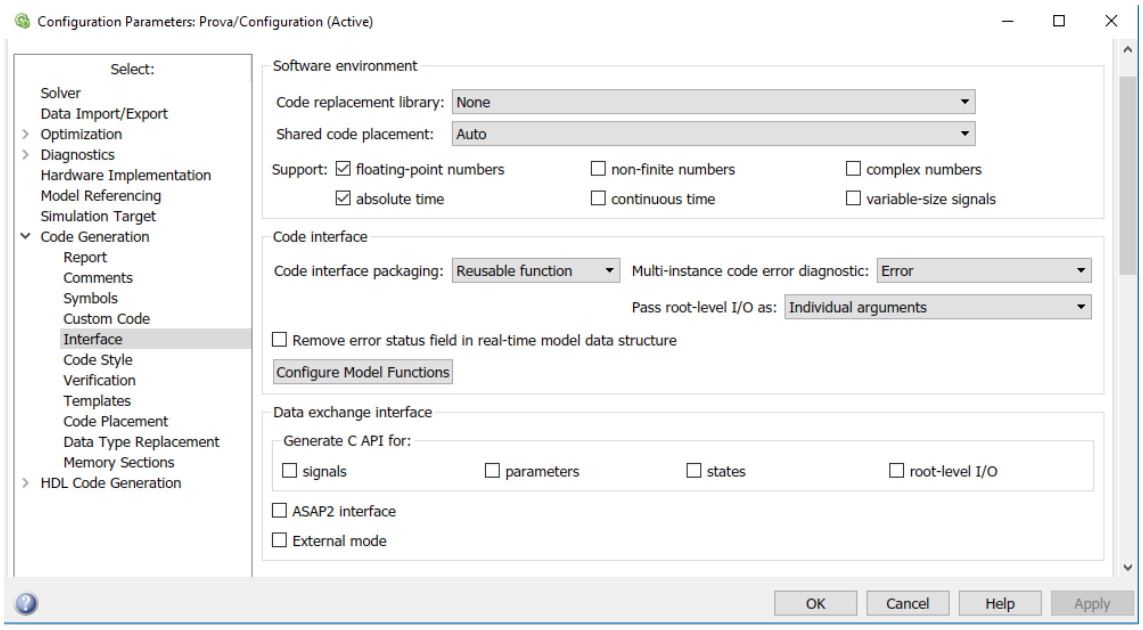
▸ Toolchain details

Code generation objectives

Prioritized objectives: Execution efficiency

Check model before generating code: Off

   b. **Interface:** Here you should pay attention to the data inside your signals to choose all the Support options. What you don't have, you should not check since it will affect readability and execution time.
   For the Code interface you HAVE TO have Reusable Function and Pass root-level I/O as individual arguments. For the error management you can choose as you please.


Configuration Parameters: Prova/Configuration (Active)

Select:
- Solver
- Data Import/Export
- Optimization
- Diagnostics
- Hardware Implementation
- Model Referencing
- Simulation Target
- Code Generation
  - Report
  - Comments
  - Symbols
  - Custom Code
  - Interface
  - Code Style
  - Verification
  - Templates
  - Code Placement
  - Data Type Replacement
  - Memory Sections
- HDL Code Generation

Software environment

Code replacement library: None

Shared code placement: Auto

Support: ☑ floating-point numbers ☐ non-finite numbers ☐ complex numbers
☑ absolute time ☐ continuous time ☐ variable-size signals

Code interface

Code interface packaging: Reusable function Multi-instance code error diagnostic: Error

Pass root-level I/O as: Individual arguments

☐ Remove error status field in real-time model data structure

Configure Model Functions

Data exchange interface

Generate C API for:

☐ signals ☐ parameters ☐ states ☐ root-level I/O

☐ ASAP2 interface

☐ External mode

c. **Check the** Create Code Generation Report **case in Report.**

d. **Further options.** All these are the basics to get the code generation going and should suffice to make it functional. <u>Further options or issues should be checked for optimality, but do it at your own risk and understanding.</u>

4. **Generating the code.** Now you are ready to generate the code. Go to Code>C/C++ Code>Build Model, or simply press Ctrl+B. It will take some time to do it and eventually the Code Generation Report will pop up.

The Generated C Code

If you have followed this tutorial up to this point, you should be facing the Code Generation Report window. <u>This is a very useful tool, since it allows you to navigate the code easily, as it has files and variables hyperlinked.</u> In this window, you can find all the **source (.c)** and **header (.h)** files generated. Up to four sections of the all the generated code can appear, and I will go through them one by one. Keep in mind that ALL .c and .h files WILL BE NEEDED for your application; the other "junk" files should be disposable for any C or Python target.

Utility files

This section is intended to be the one where all the elemental data types and Matlab constants are defined. Matlab has its own native data types and structures; these files allow referring to them and still having C code. This means that you will no longer see native C data types as they are all redefined in this section.

- **rt_defines.h:** Some mathematical and other constants are here defined.
- **rtwtyopes.h:** All data types are here redefined or declared.
- **Other files appearing might do some other specific action, not too cryptic.**

Data Files

Here you may find any user defined constants or other data inside the model. If there are any regression curves or thresholds of any kind, they will be declared and introduced in these files.

Model files

This is the core of the simulation. The ert-type generation, creates explicit functions in the **Model_name.c** file for the Simulink's 3 main simulation phases, which are:

- **Model_name_initialize**: All variables are declared, states loaded and memory is allocated. It might have other functionalities depending on the application/generation.
- **Model_name_step**: This makes the simulation advance one time step. This means that states and signals will be updated accordingly. This phase is intended to be called as many times as needed, until the simulation is finished.
- **Model_name_terminate**: This ends and closes the whole simulation, frees the memory space and might have other functionalities depending on the application/generation.

In the **Model_name.h** file, you will find the structure and type definition, as well as the data types for the system's states (they will always come as structured data).

Having a clear view of the code and its structure is necessary for any C or Python application. Data types, structures and variables need to be known.

The other files are supplementary and not very useful/explanatory.

Main file

ert_main.c is the simulation handler. It is the code in charge of calling all the functions and monitoring the simulation. In this source file we will find the template for any final application that will use the generated code. It is here where signals, timers and errors should be handled, implemented and declared.

By default, this file does nothing. Nonetheless, it has all the vital structural parts with everything else clearly indicated with auto-generated comments, and it is up to the user to mould the code to make the simulation as desired. ("Auto-generate comments" is a default option of the generation that can be unchecked).

Any future Python application should come to replace this source code, and should be able to call all the **Model_name.c** functions with the proper data types.

Compiling the Generated Code

Depending on the target of our code, several procedures are possible. If the application is C oriented, then the Main file needs to be based on the **ert_main.c,** or simply complete that code, and then be compiled as an application for the final implementation platform.

For all intends and purposes, this tutorial is conceived with the idea of a Python application, so the compiling procedure will be described for that objective in a UNIX platform.

We need to create a library that includes and links all the main functions and all the source and header files, except for the main, since we will be making our own Python version. In UNIX, those libraries are .so extension. We will be using the GCC built-in C compiler and we will build a SHARED C LIBRARY.

For that, set your path in the folder where all the generated code is. There, write the following lines in the command line:

```
gcc -shared -o libname.so -fPIC file1.c file2.c
```

In the fileX.c part you should really add ALL the .c files generated by Simulink. A big model might potentially generate many .c files that will all be needed for the simulation to work as expected.

There are plenty of ways of having a library or application compiled, and you are free to choose the one that better suits your purposes.

Python Integration

You should now have at your disposal your .so library. This will be the only file you are going to need in the Python program path from now on.

In order to be able to call compiled C functions from foreign libraries into python, we are going to need another library called "ctypes". Please refer to the doc:

https://docs.python.org/3/library/ctypes.html

The best way to convey how the Python code should be implemented is via a detailed example. This example will be equivalent to the **ert_main.c**, but instancing and declaring everything from python through ctypes. Howerver, some important points should be made:

- It is important to note that the simulation model works with two main structured variables. One is going to hold the value of every state in the system. The other one is a pointer to that first variable. Please note that they can also include some error handling variables.
  In the **ert_main.c** file those variables are declared like:

```
static RT_MODEL_NAME_T Model_NAME_M_;
static RT_MODEL_NAME_T *const Model_NAME_M = &Model_NAME_M_;/* Real-time
model */
static DW_Model_NAME_T Model_NAME_DW;/* Observable states */
```

- To be able to declare the same structures in a ctypes manner we have to know the data types and definitions and also its fields'. This information can be found in the **Model_NAME.h** file:

```
typedef struct {
  real_T state1_DSTATE;       /* '<S8>/state1 measure' */
  real_T state2_DSTATE;           /* '<S3>/state2 measure' */
  real_T DiscreteTimeIntegrator1_DSTATE;/* '<S4>/Discrete-Time
Integrator1' */
  uint8_T DiscreteTimeIntegrator_LOAD;/* '<S27>/Discrete-Time
Integrator' */
  boolean_T G1_PreviousInput;          /* '<S39>/G1' */
} DW_Model_NAME_T;

...

struct tag_RTM_Model_NAME_T { /*Just holding the pointer in this case*/
  DW_Model_NAME_T *dwork;
};
```

By going to these structures, we know the Matlab data types involved in the structured data fields. To know the actual C data type, we should go to **rtwtypes.h** file, where we would see that "`real_T`" is in fact type "`double`", "`uint8_T`" is an "`unsigned char`", and so on.

As made explicit in the documentation of the ctypes library, the equivalence is between data types is as follows:

ctypes type	C type	Python type
`c_bool`	`_Bool`	bool (1)
`c_char`	`char`	1-character string
`c_wchar`	`wchar_t`	1-character unicode string
`c_byte`	`char`	int/long
`c_ubyte`	`unsigned char`	int/long
`c_short`	`short`	int/long
`c_ushort`	`unsigned short`	int/long
`c_int`	`int`	int/long
`c_uint`	`unsigned int`	int/long
`c_long`	`long`	int/long
`c_ulong`	`unsigned long`	int/long
`c_longlong`	`__int64` or `long long`	int/long
`c_ulonglong`	`unsigned __int64` or `unsigned long long`	int/long
`c_float`	`float`	float
`c_double`	`double`	float
`c_longdouble`	`long double`	float
`c_char_p`	`char *` (NUL terminated)	string or `None`
`c_wchar_p`	`wchar_t *` (NUL terminated)	unicode or `None`
`c_void_p`	`void *`	int/long or `None`

Now we have everything needed to mimic the **ert_main.c** in Python:

```python
from ctypes import * #Import ctypes library

libc = CDLL("./libname.so") #Load library

#Rename main functions for readibility
initialize = libc.Model_NAME_initialize
step = libc.Model_NAME_step
terminate = libc.Model_NAME_terminate

#Create both ctypes structures of both state variables
class DW_Model_NAME_T(Structure):
    _fields_ = [("state1_DSTATE", c_double),
                ("state2_DSTATE", c_double),
                ("DiscreteTimeIntegrator1_DSTATE", c_double),
                ("DiscreteTimeIntegrator_LOAD", c_ubyte),
                ("G1_PreviousInput", c_ubyte)]

class RT_MODEL_Model_NAME_T(Structure):
    _fields_ = [("dwork",POINTER(DW_Model_NAME_T))]

#Now mimic their declaration in ert_main.c
Model_NAME_DW = DW_Model_NAME_T()
Model_NAME_M_ = RT_MODEL_Model_NAME_T()

Model_NAME_M = pointer(Model_NAME_M_)
Model_NAME_M_.dwork = pointer(Model_NAME_DW)
```

```
#Also declare inputs and outputs before initializing
Model_NAME_U_ref = c_double()
Model_NAME_U_In2 = c_double()
Model_NAME_U_In3 = c_int()
Model_NAME_Y_Out1 = c_double()
Model_NAME_Y_measure = c_double()

#Initialize model
initialize(Model_NAME_M,
           byref(Model_NAME_U_ref),
           byref(Model_NAME_U_In2),
           byref(Model_NAME_U_In3),
           byref(Model_NAME_Y_Out1),
           byref(Model_NAME_Y_measure));

#Define simulation parameters
timer = 0
time_step = 0.01
simulation_time = 10

while timer <= simulation_time: #Simulation loop

    #Any input manipulation or calculation should be done here
    pass

    #Input update
    Model_NAME_U_ref = c_double(1)
    Model_NAME_U_In2 = c_double(34.5)
    Model_NAME_U_In3 = c_int(2)

    #Step the model
    step(Model_NAME_M,
         Model_NAME_U_ref,
         Model_NAME_U_In2,
         Model_NAME_U_In3,
         byref(Model_NAME_Y_Out1),
         byref(Model_NAME_Y_measure))

    #Any measures or eventual manipulation should be done here
    print("Output:",Model_NAME_Y_Out1)

    #Step timer
    timer = timer+time_step

#Any plotting or post processing should be done here
pass

#Terminate the model
terminate(Model_NAME_M)
```