From Simulink to (Soft) Real Time: a step by step guide

Contents:

This document explains, in a step by step fashion, how to generate C code from an algorithm coded in Simulink, and compile it (using both the Simulink Coder and the Embedded Coder) to obtain an executable that runs in soft real time on either Windows or Linux.

The term "soft" real time is used here to indicate that, even though the underlying algorithm steps are executed, on average, at some pre-specified rate, (with respect to the computer's clock), we cannot guarantee (because neither Windows or Linux are hard real time operating systems) that errors with respect to the ideal execution rate are bounded.

That being said, the procedure is general enough to be adapted to the operating system of choice, providing one knows the corresponding timer functions.

Note that, if you are looking to just accelerate simulations, or generate an executable for an external embedded target, for a real-time microkernel sitting underneath Window or MacOS, or for a dedicated target computer hardware, Android, or iOS, then this guide is probably not what you need.

Relevant example files in the root folder

The principal files that are relevant to the example in this guide are:

- `algorithm_sl.slx`
  This is the Simulink model implementing the algorithm.
- `myprint_sys.m`, `myprint.cpp`, `myprint.h`
  The `.m` file contains the definition of a System Object which is called by `algorithm_sl.mdl`, and prints out its input, every time it is called from either MALTAB or the OS. The `.cpp` and `.h` files contain respectively declarations and code to print the input argument when the executable runs on either Windows or Linux. These files are just provided as an example of how to call OS-dependent instructions from the generated executable, but are not strictly necessary to understand the workflow. Note that this is **exactly** the same object described in the “MATLAB to Soft Real Time” guide.

Relevant example files in the “examples” folder

The `examples` folder contains the “main” C files that implement the examples of timed loop in C given in this guide, on Windows and Linux:

- `ert_main_for_fixed_spacing.c`, implements the first “fixed spacing” example, with a `for` loop, using just a `Sleep` command.
- `ert_main_for_fixed_rate.c`, implements the second “fixed rate” example, with a `for` loop, using the `GetTickCount` and the `Sleep` commands.
- `ert_main_while_fixed_rate.c`, implements the “fixed rate” example, with a `while` loop, using the `GetTickCount` and the `Sleep` commands.

- `ert_main_while_early_wakeup.c`, this is like the previous example, but shows how waking up the task before the beginning of the period and continuously checking the timing condition results in higher CPU utilization.
- `ert_main_timer.c`, this example shows how to set a “Queue Timer” (on Windows) which calls the algorithm at the desired rate.
- `ert_main_timer_linux.c`, this example shows how to set an “Interval Timer” (on Linux) which calls the algorithm at the desired rate.

The fastest way to get started is probably to run the Simulink model, generate the C code, save either of the last two files as `ert_main.c` and build the executable.

The algorithm

The algorithm is represented by the following Simulink model:

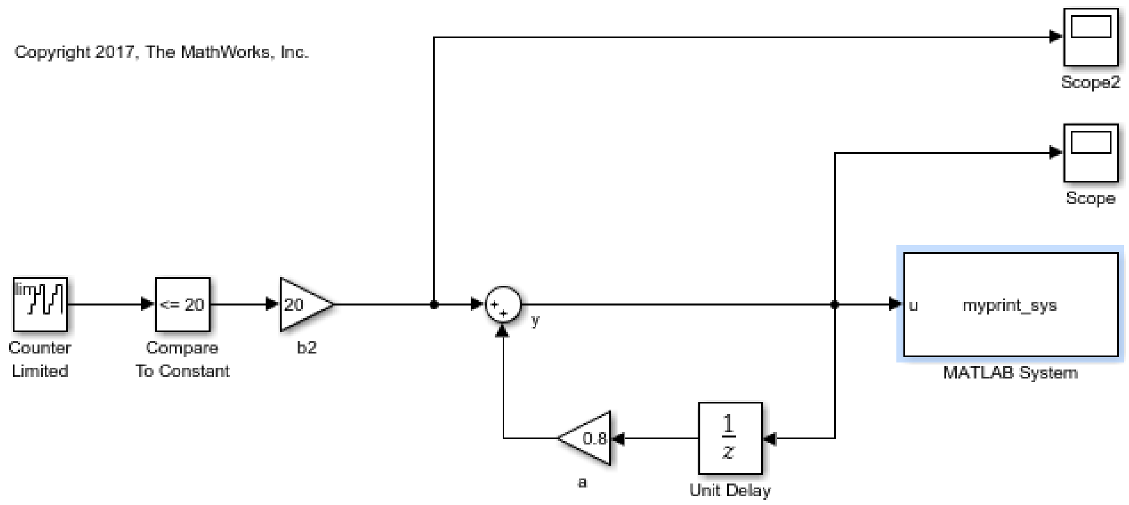


Figure 1: Simulink implementation of the algorithm

The first block is a counter which goes from 0 to 40 and then restarts from 0 again. The second block compare its input (which again goes from 0 to 40) to 20 and outputs either 1 (when its input value is lower than 20) or 0 is zero (when the input value is above 20), therefore generating a square wave. The third block multiplies its input by 20.

This is just the state space Simulink implementation of the simple low-pass IIR filter ($x^+ = x + 0.8u$) fed by a square wave signal with amplitude of 20 and period of 40 steps. Note that this is the same algorithm described in more detail in the "MATLAB to Soft Real Time" guide, and also here reported in MATLAB just for the reader's benefit:

```
function algorithm()

%% initialization
persistent x c o
if isempty(x) || isempty(c) || isempty(o)
    x=0; c=-1; o=myprint_sys;
end

%% step

% Counter, limited to 40
c=c+1; if c>40; c=0; end

% filter: next state
x=(c<=20)*20 + 0.8*x;

% output
o.step(x);
```

The two "Scope" blocks are there to visualize the filter's input and state during simulation, and the "MATLAB System" block just calls the System Object described above and defined in the files `myprint_sys.m`, `myprint.cpp`, `myprint.h`. If the execution of this block was set to Interpreted Execution this block would print out its input (using the MATLAB `disp` function), during simulation.

The "MATLAB System" block is not strictly necessary and can be deleted to start from a simpler example. On the other hand, often the final executable needs to regularly get some data or signal from the environment, process it, and output some kind of decision or result, and this is typically achieved by calling the OS Application Programming Interfaces (APIs).

This System Object is indeed provided as a starting example on how the generated executable can exchange data with the environment using some basic OS API.

The system object code

The most important part of the system object code is the one following (see the file `myprint_sys.m` for the complete code):

```
properties (Nontunable) % block parameters
    Par1 = 11
    Par2 = 9600
end

methods
    % Constructor : this must be here
    function obj = myprint_sys(varargin)
        coder.allowpcode('plain');
        % Support name-value pair arguments when constructing the
        setProperties(obj,nargin,varargin{:});
    end
end

methods (Access=protected)

    function setupImpl(obj) % Implement setup

        if coder.target('Rtw') % call external init code
            coder.cinclude('myprint.h');
            coder.ceval('myprint_init', obj.Par1,obj.Par2);
        elseif coder.target('MATLAB') % Simulation: display values
 disp(['Init: Par1=' num2str(obj.Par1) ', Par2=' num2str(obj.Par2)]);
        end

    end

    function stepImpl(~,u) % Implement step (a.k.a. output)

        if coder.target('Rtw') % call external output code
            coder.ceval('myprint_output', u);

        elseif coder.target('MATLAB') % Simulation: display values
             disp(['Step: Output=' num2str(u)]);
        end

    end

end
```

The very first part defines the two parameters `Par1` and `Par2`, (they are not really used, but just provided as a starting point in case one wants to extend the example).

The following part is the object constructor method, which returns a variable belonging to the `myprint_sys` class every time it is called.

The `setupImpl` method implements the setup part. If it is called from MATLAB (that is when `coder.target('MATLAB')==true`) then it prints the two parameters using the MATLAB "disp" function. Otherwise, if it is called from the executable (that is when `coder.target('Rtw')==true`), it called the C function `myprint_init`, declared in the included file `myprint.h` and defined in `myprint.cpp`, which prints the two parameters using the Windows or Linux `printf` function.

Finally, the `stepImpl` method implements the step part. Similarly to the setup function described above, if it is called from MATLAB, then it prints its input `u` using the `disp` function, otherwise it calls the `myprint_output` function, which again relies on the Windows or Linux `print` function.

You can refer to the documentation for a full explanation of system objects, if necessary, and on the fourth chapter of this guide for an in-detail explanation of how to use system objects to call OS or target-specific APIs.

Running the simulation in soft real time

Note that, if the simulation runs faster than real time, it is possible to use pacer blocks such as this, this, or this older one for windows, (note that this one instead is NOT recommended), to synchronize simulation time with computer time.

However, note that the accuracy and performance of these solution is of course limited by the fact that Simulink runs on top of MATLAB, which might run other tasks and use a large amount of computer resources.

Running the simulation one can easily verify that the same results seen in Figure 1 in the "MATLAB to soft real time" guide are produced.

Generating C code

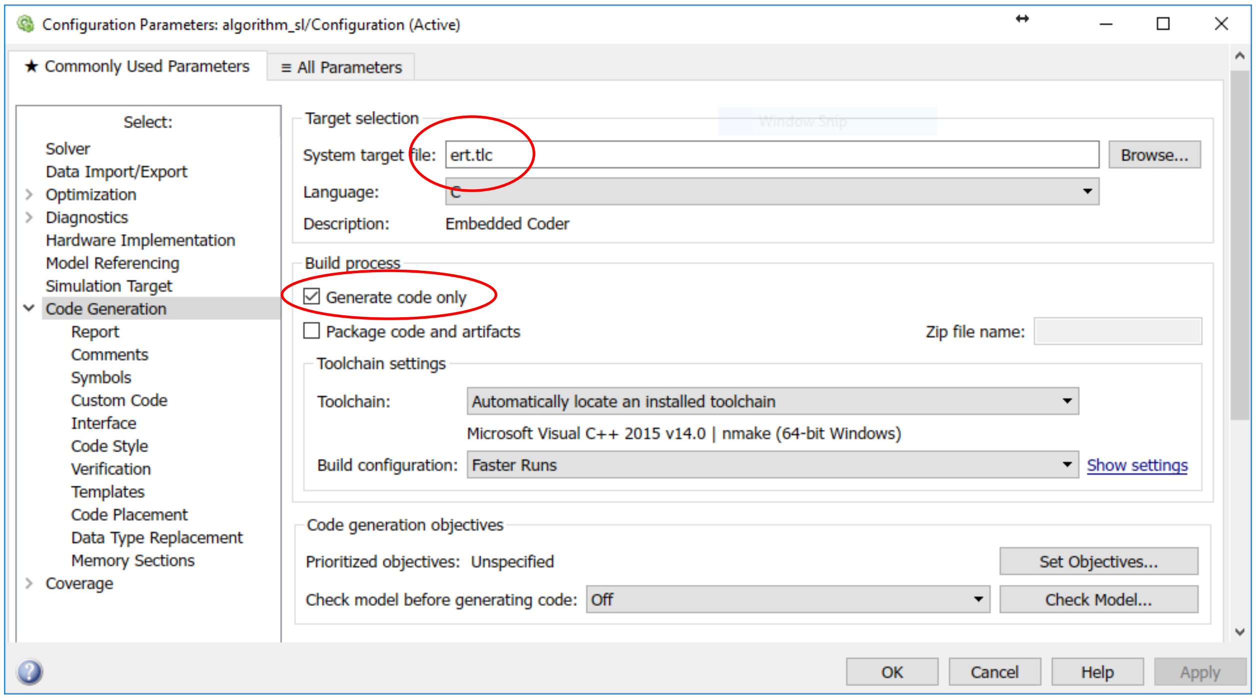
For the example shown in this guide, both the Simulink Coder and the Embedded Coder should be installed. The Embedded Coder uses a smaller memory model, and allows the user to better optimize the resulting code for speed and memory.

Relying on the Simulink code alone (and the General Real Time target) to produce executables running in soft real time on Windows, Linux, or MacOS, is also possible but not covered here in this guide for simplicity purposes.

To generate C code from the `algorithm_sl.slx` model, should be installed. One can type `ver` at the command line, and if the Simulink Coder is installed then there should be a line like this one:

```
Simulink Coder            Version 8.11         (R2016b)
Embedded Coder            Version 6.11         (R2016b)
```

Next, one should specify the target, by clicking on the "Model Configuration Parameter" button, selecting "Code Generation" on the left, and then browsing for the "Embedded Real Time" system target file, as shown in the following picture:



Configuration Parameters: algorithm_sl/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++ 2015 v14.0 | nmake (64-bit Windows)
Build configuration: Faster Runs

Code generation objectives
Prioritized objectives: Unspecified
Check model before generating code: Off

Figure 2: Selecting the system target File

Note that the "Generate code only" checkbox should be marked.

Also, one could deselect the “Create code generation report” in the “report” section of the “Code Generation” menu to streamline a bit the code generation process.

After this, the C code can be generated by either by clicking on the “Build Model” button on the right, using the keyboard shortcut “CTRL+B” or programmatically by issuing the command:

```
>> rtwbuild('algorithm_sl');
```

at the MATLAB command line.

This will generate the C code for the for the Simulink model, along with all the required object in the appropriate folders.

Assuming you have a suitable compiler installed, to generate the executable on Windows, one needs to execute the batch file `algorithm_sl.bat` located in the `algorithm_sl_ert_rtw` folder.

To do so, from the windows command line, execute the following instructions, to respectively go to the folder, execute the batch file, and return to the current folder:

```
cd algorithm_sl_ert_rtw
algorithm_sl
cd ..
```

If everything goes well, the executable `algorithm_sl.exe` should be in the current folder, and running it from the Windows command prompt should produce the output for the initialization followed by a warning indicating that the simulation will run forever unless the ‘MAT-file logging’ option is enabled.

```
C:\Users\gcampa\MATLAB Drive\srt1> algorithm_sl
Init: Par1= 11.000000
Init: Par2= 9600.000000
Warning: The simulation will run forever. Generated ERT main won't
simulate model step behavior. To change this behavior select the 'MAT-
file logging' option.
```

This can be interrupted using Ctrl-C at the Windows command shell.

Note however that the simulation is not actually running (indeed there is no Step output printed on the screen), unless there is a “Stop Simulation” block in the model.

The C code for the algorithm

The C code for the Simulink model is in the file `algorithm_sl.c`, which can be found under the `algorithm_sl_ert_rtw` folder. The core `algorithm_sl_step` function in the file is shown here as follows. You can clearly see the correspondence between the C code and the original Simulink model:

```c
void algorithm_sl_step(void)
{
  real_T rtb_y;
  uint8_T rtb_FixPtSum1;

  /* Sum: '<Root>/Sum' incorporates:
   *  Constant: '<S1>/Constant'
   *  Gain: '<Root>/a'
   *  Gain: '<Root>/b2'
   *  RelationalOperator: '<S1>/Compare'
   *  UnitDelay: '<Root>/Unit Delay'
   *  UnitDelay: '<S2>/Output'
   */
  rtb_y = (real_T)(algorithm_sl_DW.Output_DSTATE <= 20) * 20.0 + 0.8 *
    algorithm_sl_DW.UnitDelay_DSTATE;

  /* Start for MATLABSystem: '<Root>/MATLAB System' incorporates:
   *  MATLABSystem: '<Root>/MATLAB System'
   */
  myprint_output(rtb_y);

  /* Sum: '<S3>/FixPt Sum1' incorporates:
   *  Constant: '<S3>/FixPt Constant'
   *  UnitDelay: '<S2>/Output'
   */
  rtb_FixPtSum1 = (uint8_T)(algorithm_sl_DW.Output_DSTATE + 1U);

  /* Switch: '<S4>/FixPt Switch' */
  if (rtb_FixPtSum1 > 40) {
    /* Update for UnitDelay: '<S2>/Output' incorporates:
     *  Constant: '<S4>/Constant'
     */
    algorithm_sl_DW.Output_DSTATE = 0U;
  } else {
    /* Update for UnitDelay: '<S2>/Output' */
    algorithm_sl_DW.Output_DSTATE = rtb_FixPtSum1;
  }

  /* End of Switch: '<S4>/FixPt Switch' */

  /* Update for UnitDelay: '<Root>/Unit Delay' */
  algorithm_sl_DW.UnitDelay_DSTATE = rtb_y;
}
```

The main file

The `ert_main.c` file, which is used to build the executable, can be found under the in the same folder, is shown here without many comments so it could fit one page:

```c
#include <stddef.h>
#include <stdio.h>          /* This ert_main.c example uses printf/fflush */
#include "algorithm_sl.h"   /* Model's header file */
#include "rtwtypes.h"

void rt_OneStep(void);
void rt_OneStep(void)
{
  static boolean_T OverrunFlag = false;

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

  OverrunFlag = true;

  /* Step the model */
  algorithm_sl_step();

  /* Indicate task complete */
  OverrunFlag = false;

int_T main(int_T argc, const char *argv[])
{
  /* Unused arguments */
  (void)(argc);
  (void)(argv);

  /* Initialize model */
  algorithm_sl_initialize();

  printf("Warning: The simulation will run forever. "
         "Generated ERT main won't simulate model step behavior. "
         "To change this behavior select the 'MAT-file logging' option.\n");
  fflush((NULL));
  while (rtmGetErrorStatus(algorithm_sl_M) == (NULL)) {
    /*  Perform other application tasks here */
  }

  /* Disable rt_OneStep() here */

  /* Terminate model */
  algorithm_sl_terminate();
  return 0;
}
```

Note the `algorithm_sl_step()` function shown in page 9 is called by the `rt_OneStep` function declared and defined at the beginning, which also takes care of handling the overrun flag.

The `ert_main` function then calls the model initialization function, prints the warning already seen in page 8, executes an infinite while loop with an error exit condition, (which causes 100% CPU utilization and is meant to host other instructions and/or sleep commands), before calling the model termination function and exiting.

An easy way to check the behavior of the algorithm for successive iterations is to declare an unsigned integer variable, e.g. `i` and call the `rt_OneStep()` function within a `for` loop:

```
for (i=0;i<101;i++) {
       rt_OneStep();
}
```

In the following sections, we will show how very simple modifications to the `ert_main.c` file allow the algorithm to run at a certain controlled rate.

Timed execution on Windows, using `Sleep`

The easiest way to implement a timed loop is to use the Windows C API `Sleep` in the same way in which the "pause" function is used in the first MATLAB timed loop example.

To do this, we need the following modifications to the original `ert_main.c` file:

1) The `<windows.h>` library, in which the needed APIs are declared, needs to be included. This can be done by inserting the following line at the end of the include section:

   ```
   #include <windows.h> /* Using Sleep, GetTickCount, or Timers */
   ```

2) The unsigned integer variables `i` and `T` (which will serve respectively as a simple counter and sampling time) need to be declared. This can be done by

inserting the following line just after the declaration of the "main" function and before the `(void)argc;` line:

```
unsigned int i, T=100;
```

Note that the sampling time variable `T` is initialized to 100 milliseconds (0.1s).

3) The lines going from the `printf` instruction to the end of the `while` loop, need to be replaced with the `for` loop described in the previous section:

```
for (i=0;i<101;i++) {
        rt_OneStep();
        Sleep(T);
}
```

Re-building the executable (by running "algorithm_sl.bat" as shown before) with this new "ert_main.c" file in place, and running the new algorithm_sl.exe file, should show results for initialization, followed by 101 steps, while taking approximately 10 seconds:

```
Init: Par1= 11.000000
Init: Par2= 9600.000000
Step: Output: 20.000000
.
.
.
Step: Output: 98.219140
Step: Output: 98.575312
```

Note that using `Sleep(T)` in this way causes a *fixed spacing* of T seconds between *the end of* the pervious execution of `rt_OneStep()` and *the start of* the next one, resulting in a slower than 1/T execution rate, since the period will be (on average) approximately equal to $T+T_{task}$ where T_{task} is the time needed to execute the task.

Timed execution on Windows, using `Sleep` and `GetTickCount`

A more accurate way to implement a timed loop in Windows relies on the function `GetTickCount`. The following modifications to the original `ert_main.c` file are needed:

1) The `<windows.h>` library, in which the needed APIs are declared, needs to be included. This can be done by inserting the following line at the end of the include section:

```
#include <windows.h> /* Using Sleep, GetTickCount, or Timers */
```

2) The unsigned integer variables `i` and `T` (which will serve respectively as a simple counter and sampling time), as well as the double word variables `t_start` and `t_task` (which will serve respectively to measure the start time of the task and its duration) need to be declared. This can be done by inserting the following lines just after the declaration of the `main` function and before (or immediately after) the `(void)argc;` line:

```
unsigned int i, T=100;
DWORD t_start, t_task;
```

Note that the sampling time variable T is initialized to 100 milliseconds (0.1s).

3) The lines going from the `printf` instruction to the end of the `while` loop, need to be replaced with the following `for` loop:

```
for (i=0;i<101;i++) {

    /* get t_start, execute algorithm and calculate task time */
    t_start = GetTickCount();
    rt_OneStep();
    t_task = GetTickCount() - t_start;

    /* sleep for the rest of the period */
    Sleep(T-t_task);
}
```

As before, re-building the executable (by running `algorithm_sl.bat` as shown above) with this new `ert_main.c` file in place, and running the new `algorithm.exe` file, should show results for initialization, followed by 101 steps, while taking approximately 10 seconds:

```
Init: Par1= 11.000000
Init: Par2= 9600.000000
Step: Output: 20.000000
.
.
.
```

```
Step: Output: 98.219140
Step: Output: 98.575312
```

Note that using `Sleep(T-t_task)` in this way pauses execution for the remaining part of the period, therefore trying to enforce a fixed rate of exactly 1/T Hertz, since the next execution of `rt_OneStep()` should start exactly T seconds after *the start* of the previous execution.

Alternatively, we can also use a `while` loop instead of a `for` loop. To do that, in step #2 we need to define also the double word variable `t_loop` (which marks the start of the loop) and the integer variable `Tf` (the final time). This is done with the following instructions:

```
unsigned int T=100, Tf=10000;
DWORD t_start, t_task, t_loop;
```

Note that final time variable `Tf` is initialized to 10000 milliseconds (10s). Then in step #3 the code that replaces the lines going from the `printf` instruction to the end of the `while` loop, would be the following:

```
t_loop = GetTickCount();
while (GetTickCount()-t_loop < Tf) {

    /* get t_start, execute algorithm and calculate task time */
    t_start = GetTickCount();
    rt_OneStep();
    t_task = GetTickCount() - t_start;

    /* sleep for the rest of the period */
    Sleep(T-t_task);
}
```

Building the executable and running it produces the same results, except for the fact that not all 101 steps will be completed, for example if only around 90 steps are completed, the last output lines would be something like the following ones:

```
Step: Output: 79.268061
Step: Output: 83.414448
```

In general, using a while loop makes it easier to enforce a predefined total duration time for the whole process, as opposite to a predefined number of steps.

Multitasking considerations, early wakeup, and CPU usage

With the fixed spacing example that uses just `Sleep`, on page 12, timing error will tend to accumulate so the actual period will be even larger than $T+T_{task}$, especially if the OS is subject to a considerable workload and many other computational demanding processes are running concurrently.

With the second (and third) examples that also rely on `GetTickCount`, on pages 13 and 14, the actual period should better approximate T, except for (largely unpredictable but hopefully minor) OS response times due to the fact the OS cannot guarantee that the task would resume exactly at the intended time after a Sleep instruction (see the other pdf guide for a slightly more detailed explanation).

One could arguably attempt to improve the response time by awaking the process a little before the intended time, which can be done with the same initialization instructions given in the previous page, and the following loop:

```
t_loop = GetTickCount();
t_start = GetTickCount();
 while (GetTickCount()-t_loop < Tf) {

     if (GetTickCount()-t_start>=T) {

          /* get t_start, execute algorithm and calculate task time */
          t_start = GetTickCount();
          rt_OneStep();
          t_task = GetTickCount() - t_start;

          /* sleep until 5 ms before the start of the next period */
          /* removing this line will cause 100% CPU utilization */
          Sleep(T-t_task-5);
      }

 }
```

In this case the process is awaken 5 milliseconds prior to the start of the next period, and the "if" condition is continuously checked (causing 100% CPU utilization in those 5ms) until the very start of the period. Resulting improvements in response times are however debatable and dependent on many factors.

More importantly, one should not attempt to simplify the above loop by completely removing the `Sleep` instruction (together with the line calculating the task time, which would obviously be redundant in that case).

Doing so would result in a loop as the following one:

```
t_loop = GetTickCount();
t_start = GetTickCount();
 while (GetTickCount()-t_loop < Tf) {

     /* 100% CPU usage when if condition is continuously checked */
     if (GetTickCount()-t_start>=T) {

          /* get t_start, execute algorithm and calculate task time */
          t_start = GetTickCount();
          rt_OneStep();
      }

 }
```

Which, when executed, will cause 100% CPU utilization for the whole time it is running, leading to (possibly damaging) overheating, starvation for other processes, and OS loss of responsiveness or hangs.

Timed execution on Windows, the proper way (using timers).

Timers are often the best tradeoff between accuracy, performance, overhead, and ease of use. Moreover, in most practical situations in which a fixed-rate timed execution is required, one needs to run the loop indefinitely until it is externally interrupted, (as opposed to having a final time or number of iterations).

In such situations, it is easier, and less error-prone, to just set up a timer which will run the algorithm as a callback, and use the core part of the "main" function to perform other things (when necessary) or simply sleep.

This can be accomplished by performing the following modifications to the original `ert_main.c` file:

1) As seen before, the `<windows.h>` library, in which the needed APIs are declared, needs to be included. This can be done by inserting the following line at the end of the include section:

```
#include <windows.h> /* Using Sleep, GetTickCount, or Timers */
```

2) The unsigned integer variable and `T`, (the sampling time), as well as the timer handle `hTimer` need to be declared. This can be done by inserting the following lines just after the declaration of the "main" function and before the `(void)argc;` line:

```
unsigned int T=100;
HANDLE hTimer = NULL;
```

   Note that the sampling time variable `T` is initialized to 100 milliseconds (0.1s).

3) The going from the `printf` instruction to the end of the `while` loop, need to be replaced with the following set of instructions:

```
/* Set a timer to call the timer routine every T ms */
if (!CreateTimerQueueTimer( &hTimer, NULL,
     (WAITORTIMERCALLBACK)rt_OneStep, NULL , 0, T, 0)) {
printf("CreateTimerQueueTimer failed (%d)\n", GetLastError());
return 3;
}

while (1) {
  Sleep(1000*T); /* Sleep to avoid 100% CPU utilization */
}
```

Re-building the executable (by running `algorithm_sl.bat` as shown before) with this new `ert_main.c` file in place, and running the new `algorithm.exe` file, will result in the algorithm initializing, and then running at the required rate indefinitely, until interrupted by pressing Ctrl-C or until the Windows command window is closed.

The timer used in the above example belongs to the "Queue Timers" class. Windows has three a total of four different timer classes (the other three being Standard Win32, Multimedia, and Waitable Timers). The Queue timers work only Windows 2000 and later, but enjoy a relatively high-precision and a low-overhead.

Some practical considerations about rapid prototyping

Note that resulting `ert_main.c` file can be found as `ert_main_timer.c` in the examples folder coming in the zip file. If your upper lever Simulink model is called algorithm_sl then after generating the code you can simply replace the ert_main.c file with the provided one and build the executable.

Note that, differently from what happens with the codegen command, even if the ert_main.c file has been manually changed, when the C code is re-generated, following any changes in the Simulink model, the ert_main.c file is overwritten. Therefore, after changing the Simulink model (but not its name), and re-generating the code, one needs to replace the ert_main.c file with the one provided, before rebuilding the executable.

Generating C code, building and running the executable on Linux

Assuming you have Simulink, (as well as the Simulink Coder, the Embedded Coder, and a suitable compiler) running on a Linux platform, the steps to generate C code from the algorithm_sl model, build the executable, and running it, are essentially the same.

You can generate the C code for the for the algorithm_sl model, for example, with the following MATLAB command line:

```
>> rtwbuild('algorithm_sl');
```

To build the executable, one needs to execute the make file algorithm_sl.mk in the algorithm_sl_ert_rtw folder. To do so, from the Linux shell (i.e. the Linux command line), execute the following instructions, to respectively go to the appropriate folder, build the executable, and return to the current folder:

```
cd ./algorithm_sl_ert_rtw/
make -f algorithm_sl.mk
cd ..
```

If everything goes well, the executable algorithm_sl should be in the current folder, and running it from the Linux shell should produce the output for the initialization followed by a warning indicating that the simulation will run forever unless the 'MAT-file logging' option is enabled.

```
% ./algorithm> ./algorithm
Init: Par1= 11.000000
Init: Par2= 9600.000000
```

```
Warning: The simulation will run forever. Generated ERT main won't
simulate model step behavior. To change this behavior select the 'MAT-
file logging' option.
```

This can be interrupted using Ctrl-C at the Linux command shell.

Both the `algorithm_sl.c` and the `ert_main.c` files are the same as the ones seen before in the Windows case, so they don't need to be re-presented again here.

Timed execution on Linux, using timers

As discussed before in the Windows case, in many practical situations timers offer the best choice in terms of accuracy, overhead, and ease of use. Setting up the generated `rt_OneStep()` function as a timer callback can be easily accomplished in Linux by, for example, performing the following modifications to the original `ert_main.c` file:

1) Several libraries which contain the required APIs need to be included. This can be done by inserting the following lines at the end of the include section:

```
#include <sys/time.h>   /* for setitimer */
#include <unistd.h>     /* for pause */
#include <signal.h>     /* for signal */
```

2) The signal handler function needs to be re-set to `rt_OneStep` every time the latter runs, because otherwise, in some conditions, the system resets it to default handler SIG_DFL (which prints the message "Alarm Clock" and terminates the execution of the loop) after the original handler runs just one time. This can be done by inserting the line:

```
signal(SIGALRM, rt_OneStep);
```

just after the `algorithm_sl_step();` call in (and before the `OverrunFlag = false;`) of the `rt_OneStep` function definition. This will result in the following definition (without some comments):

```
void rt_OneStep(void) {
  static boolean_T OverrunFlag = false;

  /* Disable interrupts here */
```

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

    OverrunFlag = true;

    /* Step the model */
    algorithm_sl_step();

    /* Re-set the handler to rt_OneStep, just in case  */
    signal(SIGALRM, rt_OneStep);

    /* Indicate task complete */
    OverrunFlag = false;

  }
```

3) The timer structure needs to be defined. This can be done by inserting the following line just after the declaration of the "main" function and before the `(void)argc;` line:

```
/* Define the timer structure */
struct itimerval it_val;
```

4) The lines going from the `printf` instruction to the end of the `while` loop, need to be replaced with the following set of instructions:

```
/* Upon SIGALRM, call main_algorithm() */
if (signal(SIGALRM, (void(*)(int)) rt_OneStep) == SIG_ERR) {
     perror("Unable to catch SIGALRM");
     exit(1);
}

/* Define Sampling time in secs and microsecs */
it_val.it_value.tv_sec =     0;
it_val.it_value.tv_usec =    100000;
it_val.it_interval = it_val.it_value;

/* Set the timer */
if (setitimer(ITIMER_REAL, &it_val, NULL) == -1) {
     perror("error calling setitimer()");
     exit(1);
}

while (1) {
     pause();
}
```

Re-building the executable (by running `make -f algorithm_sl.mk` from the Linux shell as shown before) with this new `ert_main.c` file in place, and running the new `./algorithm_sl` file, will result in the algorithm initializing, and then running at the required rate indefinitely, until interrupted by pressing Ctrl-C or until the Linux command window is closed.

Rapid prototyping considerations

The `ert_main.c` file resulting from the above modifications can be found as `ert_main_timer_linux.c` in the examples folder coming in the zip file.

As seen in the windows case, if your upper lever Simulink model is called `algorithm_sl` then after generating the code you can simply replace the `ert_main.c` file with the provided one and build the executable.

Note that, differently from what happens with the `codegen` command, even if the `ert_main.c` file has been manually changed, when the C code is re-generated, following any changes in the Simulink model, the `ert_main.c` file is overwritten. Therefore, after changing the Simulink model (but not its name), and re-generating the code, one needs to replace the `ert_main.c` file with the one provided, before rebuilding the executable.

Finally, it's worthwhile to notice that, under Linux, one could use extensions and patches like "preempt_rt" to gain some hard real time capabilities if needed.