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

Contents:

From MATLAB to (Soft) Real Time: a step by step guide 1
- Relevant example files in the root folder 2
- Relevant example files in the `ML_examples` folder 2
- The algorithm 3
- The System Object code 4
- Calling the algorithm from MATLAB 6
- A quick look at the algorithm output 6
- Generating C code 8
- The C code for the algorithm function 9
- The main file 10
- Timed execution on Windows, using `Sleep` 11
- Timed execution on Windows, using `Sleep` and `GetTickCount` 13
- Multitasking considerations 15
- Waking up the process early causes higher CPU usage 16
- Timed execution on Windows, the proper way (using timers). 18
- Some practical considerations about rapid prototyping 19
- Generating C code, building and running the executable on Linux 20
- Timed execution on Linux, using timers 21
- Rapid prototyping considerations 22

This document explains, in a step by step fashion, how to generate C code from an algorithm coded in MATLAB, and compile it (using MATLAB Coder) to obtain an executable which 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, in practice and 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 deviations of the actual execution rate with respect to the ideal one will be bounded.

giampiero.campa@mathworks.com

That being said, the procedure is general enough to be adapted to the operating system of choice, providing one knows the corresponding timer functions. Also note that, if you are looking to just speed up MATLAB code, (even by generating MEX functions from MATLAB code), 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.m`
  Contains the main algorithm function.
- `myprint_sys.m`, `myprint.cpp`, `myprint.h`
  The `.m` file contains the definition of a System Object which is called by `algorithm.m`, and prints out its input, either on MATLAB or in the Operating System (OS), every time it is called. The `.cpp` and .h files contain declarations and code to print the input argument when the executable runs on either Windows or Linux. These files are provided as an example of how to call OS-dependent instructions from the generated executable, but are not strictly necessary to understand the workflow.

Relevant example files in the `ML_examples` folder

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

1. `main_for_fixed_spacing.c`, implements the first "fixed spacing" example, with a `for` loop, using just a `sleep` command.
2. `main_for_fixed_rate.c`, implements the second "fixed rate" example, with a `for` loop, using the `GetTickCount` and the `sleep` commands.
3. `main_while_fixed_rate.c`, implements the "fixed rate" example, with a `while` loop, using the `GetTickCount` and the `sleep` commands.
4. `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.

5. `main_timer.c`, this example shows how to set a "Queue Timer" (on Windows) which calls the algorithm at the desired rate.
6. `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 to try the algorithm in MATLAB, generate the C code, save either of the last two files (items 5 or 6 above) as `main.c` and build the executable.

The algorithm

It is not necessary to fully understand the algorithms, but nevertheless, for the reader's benefit, the MATLAB function implementing it is shown below:

```
function algorithm()
% Example of IIR filter fed by a square wave, with output print
% Copyright 2017, The MathWorks, Inc

%% initialization: effective only the first calling time
persistent x c out
if isempty(x) || isempty(c) || isempty(out)
    x=0;c=-1;out=myprint_sys;
end

%% step: this part updates the input c, the state x, and prints x

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

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

% out(x) calls out.step(x), which prints the value of x when called
% from either MATLAB (in interpreted mode) or the OS (as executable)
out(x);
```

It has an initialization part, which is executed only once to initialize the three variables `c`, `x`, and `out` if either of them are empty. The following part performs three actions:

1) The first line increments the counter variable `c`, and resets it to 0 to prevent it from reaching values above 40. So the counter goes from 0 to 40 and then restarts from 0 again.

2) The following line implements a low-pass IIR filter ($x^+ = x + 0.8u$) fed by a square wave with a period of 40, having amplitude equal to 20 for the first 20 steps and 0 for the following 20 steps ($u = 20(c \leq 20)$).
3) The final line executes the step method of the System Object variable `out`. This prints the filter state variable `x`, (which is fed as input for the step function), either on MATLAB (using the `disp` command when this instruction is executed on MATLAB), or on the operating system (using the functions in `myprint.cpp` and `myprint.h`, which in turn call `printf`, when the executable generated from the MATLAB code runs in either Windows or Linux.

The System Object variable `out`, along with its class file `myprint_sys.m`, and the OS support files `myprint.cpp` and `myprint.h`, are only provided as a starting example on how the generated executable can exchange data with the environment using some basic OS API.

While one does not have to use a System Object to do the above (i.e. technically the instructions `coder.target`, `coder.cinclude`, and `coder.ceval` can be called from anywhere in the algorithm to respectively detect the coder target, include a C file, and evaluate a C function), System Objects provide a general and consistent framework (e.g. the same exact code can be called from both MATLAB and Simulink) which also allows one to neatly separate the pure algorithmic and numerical parts of the code (which should be totally abstracted from the OS) from its "input/output" parts (which might rely on different OS API and services).

The System Object code

The most important part of the System Object code is the 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 (in which case `coder.target('Rtw')==true`), it calls 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 `printf` 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.

Calling the algorithm from MATLAB

The above algorithm can be called from MATLAB (just to make sure it works before generating code) simply by typing `algorithm` at the command line a few times:

```
>> algorithm
Init: Par1=11, Par2=9600
Step: Output=20
>> algorithm
Step: Output=36
>> algorithm
Step: Output=48.8
```

Note that the first time the function algorithms is executed, the variables `c`, `x`, and `out` are empty ad are initialized before performing the first step. Also, the method `out.setup` (which prints the parameters) is automatically called (only) before the first `out.step(x)` call (which prints the input variable `x`).

In the following `algorithm` calls, the `x` variable is updated and `out(x)` (a proxy for `out.step(x)`) prints out the value of `x`.

A quick look at the algorithm output

Calling the algorithm form MATLAB in a for loop as following:

```
T=0.1;Tf=10;
for i=1:1+Tf/T;
    algorithm();
end
```

produces the following output:

```
Init: Par1=11, Par2=9600
Step: Output=20
Step: Output=36
.
.
.
Step: Output=98.2191
Step: Output=98.5753
```

It is also possible (by adding the line `y=x` at the end of **`algorithm.m`** and using `y(i)=algorithm();` in the for loop) to save the state values in the workspace so that it can be easily plotted using the "`plot(0:T:Tf,y)`" command:

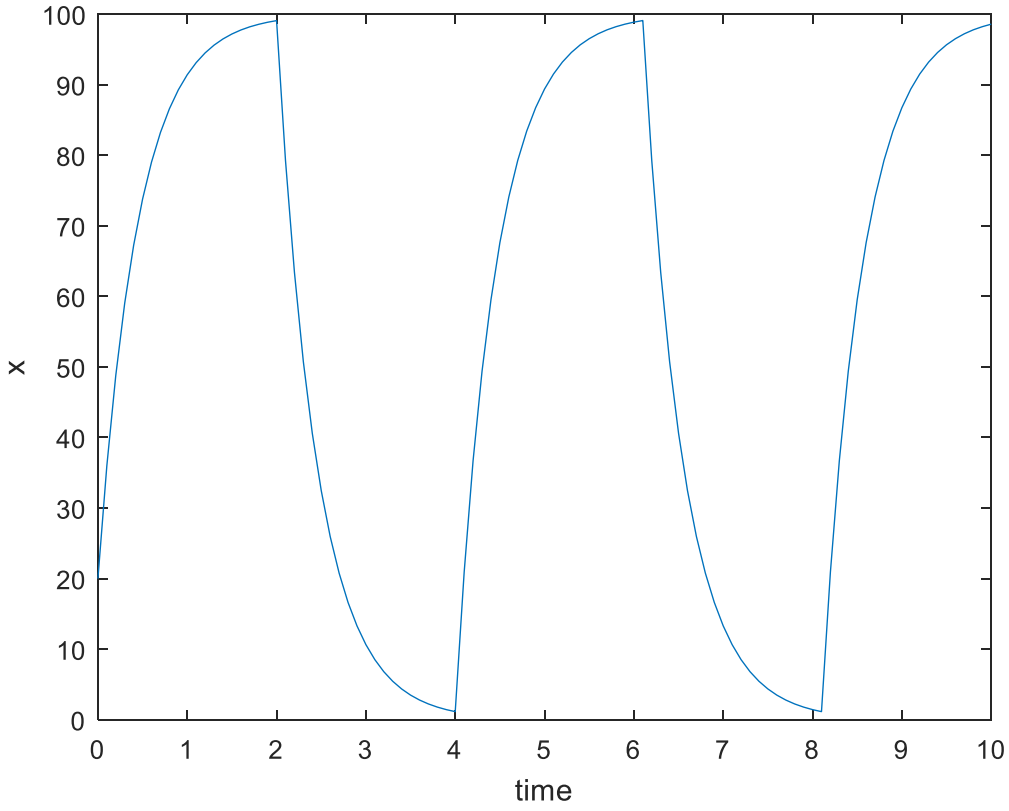

Figure 1: Algorithm output for the first 10 seconds

The low pass filtering behavior (on a square wave) of the main part of the algorithm is clearly visible from the figure.

Generating C code

To generate C code from the `algorithm()` function, MATLAB Coder should be installed. One can type `ver` at the command line, and if MATLAB coder it is installed then there should be a line like this one:

```
MATLAB Coder            Version 3.2         (R2016b)
```

Next, an appropriate coder configuration object must be created:

```
cfg = coder.config('exe');
cfg.CustomSource = 'main.c';
cfg.CustomInclude = '.\codegen\exe\algorithm\examples';
```

The first line specifies that the target is an executable running on the same OS as MATLAB, the second and third lines specify that we want an example `main.c` to be generated as well, and placed in the `examples` folder under the folder `codegen\exe\algorithm`.

After this, the following line generates the C code for the MATLAB `algorithm` function, along with all the required object in the appropriate folders:

```
codegen algorithm -c -config cfg
```

Assuming you have a suitable compiler installed, to build the executable, one needs to execute the batch file `algorithm_rtw.bat` in the `codegen\exe\algorithm` folder. To do so, from the windows command line, execute the following instructions, to respectively go to the appropriate folder, execute the batch file, and return to the current folder:

```
cd .\codegen\exe\algorithm
algorithm_rtw
cd ..\..\..
```

If everything goes well, the executable `algorithm.exe` should be in the current folder, and running it from the (Windows) command prompt should produce the output for the initialization followed by just one step:

```
C:\Users\gcampa\MATLAB Drive\srt1>algorithm
Init: Par1= 11.000000
Init: Par2= 9600.000000
Step: Output: 20.000000
```

The C code for the algorithm function

The C code for the algorithm function is in the file `algorithm.c`, which can be found under the `.\codegen\exe\algorithm\` folder.

The main `algorithm` function in the file is shown here as follows. You can clearly see the initialization section, as well as the step section. The code is readable, and there is a very clear correspondence between the C code and the original MATLAB one:

```
void algorithm(void)
{
  double varargin_1;
  if ((!x_not_empty) || (!c_not_empty) || (!out_not_empty)) {
    x = 0.0;
    x_not_empty = true;
    c = -1.0;
    c_not_empty = true;
    out.isInitialized = 0;
    out_not_empty = true;
  }

  /*  step */
  /*  Counter, limited to 40 */
  c++;
  if (c > 40.0) {
    c = 0.0;
  }

  /*  filter: next state */
  x = (double)(c <= 20.0) * 20.0 + 0.8 * x;

  /*  output */
  varargin_1 = x;
  if (out.isInitialized != 1) {
    out.isInitialized = 1;
    myprint_init(11.0, 9600.0);
  }

  myprint_output(varargin_1);
}
```

As it will be shown next, this function is called from the `main_algorithm()` C function, which is declared, defined, and called used in the `main.c` file.

The main file

The `main.c` file, which is used to build the executable, can be found under the `.\codegen\exe\algorithm\examples` folder:

```
/* Include Files */
#include "rt_nonfinite.h"
#include "algorithm.h"
#include "main.h"
#include "algorithm_terminate.h"
#include "algorithm_initialize.h"

/* Function Declarations */
static void main_algorithm(void);

/* Function Definitions */

/*
 * Arguments    : void
 * Return Type  : void
 */
static void main_algorithm(void)
{
  /* Call the entry-point 'algorithm'. */
  algorithm();
}

/*
 * Arguments    : int argc
 *                const char * const argv[]
 * Return Type  : int
 */
int main(int argc, const char * const argv[])
{
  (void)argc;
  (void)argv;

  /* Initialize the application.
     You do not need to do this more than one time. */
  algorithm_initialize();

  /* Invoke the entry-point functions.
     You can call entry-point functions multiple times. */
  main_algorithm();

  /* Terminate the application.
     You do not need to do this more than one time. */
  algorithm_terminate();
  return 0;
}
```

As you can see in the main function, after the initialization performed by `algorithm_initialize();` function the `main_algorithm();` function is only called once, which is why running the generated executable produced the output for the initialization and just one step, as shown before.

An easy way to check the behavior of the algorithm for successive iterations is to declare a unsigned integer variable, e.g. `i` and call the `main_algorithm();` function within a `for` loop (instead of just once as in the original main file):

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

In the following sections, we will show how very simple modifications to the main 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`.

To do this, we need the following modifications to the original main 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 in the main file 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 before (or just after) 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 `main_algorithm();` line, needs to be replaced with the `for` loop described in the previous section, that is:

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

Re-building the executable (by running `algorithm_rtw.bat` as shown before) with this new `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)` in this way tries to enforce a (roughly) a *fixed spacing* of T seconds between *the end of* the previous execution of the task (that is the function `main_algorithm()`) 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.

Just as a side note, a similar behavior can be obtained within the MATLAB environment with the following initialization line (to set sampling and final time) and `for` loop:

```
%% sampling time and final time
T=0.1; Tf=10;

%% simple loop
tic
for i=1:1+Tf/T;

    algorithm();
    pause(T);

end
toc
```

where the `pause` command is used in place of the C Windows API `Sleep`.

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

A more accurate way to implement a timed loop in Windows is to use the function `GetTickCount` as well as `Sleep`:.

To do this, we need the following modifications to the original main 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 in the main file 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 (or just before) 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 `main_algorithm();` line, needs 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();
    main_algorithm();
    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_rtw.bat` as shown above) with this new `main.c` file in place, and running the new `algorithm.exe` file, should show results for initialization, followed by 101 steps, while taking just about 10 seconds:

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

Here the `GetTickCount` instruction is used twice before and after calling `main_algorithm()`, to measure its execution time (`t_task`). The instruction `Sleep(T-t_task)` then pauses execution of the thread for the remaining part of the period. Note that this amount to trying to enforce a *fixed rate* of exactly 1/T Hertz, since the next execution of `main_algorithm()` 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 `main_algorithm()` would be the following:

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

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

Just as a side note, a similar behavior can be obtained within the MATLAB environment with the following initialization line (to set sampling and final time) and `for` loop:

```
%% sampling time and final time
T=0.1; Tf=10;

%% timed loop
tic
while toc<Tf

    t_start=toc;
    algorithm();
    t_task=toc-t_start;

    pause(T-t_task);
end
toc
```

where the `toc` instruction is used in place of the C Windows API `GetTickCount`.

Multitasking considerations

It is important to remember that in any multitasking OS, higher priority process might interrupt the `algorithm.exe` process at any time (and since neither Windows nor Linux can be classified as a "Real Time" operating system, it is not even guaranteed that the process will resume in any fixed pre-established amount of time).

Using the first "fixed spacing" timed loop example, if any other task interrupts the execution of `main_algorithm()` for a certain time T_{others}, the total time elapsed from the beginning of one execution of the algorithm to the beginning of the next one will be $T_{task}+T_{others}+T+T_{response}$. The final factor $T_{response}$ takes into account the time that the OS might need to switch to the current task from a previous one.

So, in summary, with the simple "fixed spacing" loop these kind of timing errors will tend to accumulate, resulting in actual period intervals that are even longer than $T+T_{task}$.

With the second (and third) "fixed rate" examples on pages 13 and 14, which also rely on `GetTickCount`, 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. In other words, in these cases the total time elapsed from the beginning of one execution of the algorithm to the beginning of the next one will be $(T_{task}+T_{others}) + T -(T_{task}+T_{others}) + T_{response} = T+T_{response}$, which should be a good approximation of T.

Waking up the process early causes higher CPU usage

Assuming the `algorithm.exe` process has a higher priority than most other processes, one possible attempt to decrease $T_{response}$ is to make sure that `algorithm.exe` is running and ready to execute the next iteration of the algorithm right before the start of the next period T. This can be attempted with the following timed loop example:

```
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();
          main_algorithm();
          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 during those 5ms) until the very start of the period. Resulting improvements in response times are however debatable and dependent on many factors.

An apparently logical simplification of the above loop, which is unfortunately still used in some cases, is the following one, in which the `Sleep` instruction has been removed (together with the line calculating the task time, which would obviously be redundant in that case):

```
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();
          main_algorithm();
      }

 }
```

While the above loop works, it causes 100% CPU utilization for the whole time it is running. This causes serious overheating, it does not leave enough CPU time to other processes, and often causes the OS to become unresponsive and/or hang.

Just as a side note, a similar behavior can be obtained within the MATLAB environment with the following initialization line and `for` loop:

```
T=0.1; Tf=10;

tic;t_start=toc;
while toc<Tf

    if toc-t_start>=T
        t_start=toc;
        algorithm();
        t_task=toc-t_start;
    end

    pause(T-t_task-0.01); % removing this line causes 100% CPU usage

end
toc
```

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 main 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 in the main file at the end of the include section:

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

2) The unsigned integer variable `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 before (or just after) 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 `main_algorithm();` line, needs 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)main_algorithm, 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_rtw.bat` as shown before) with this new `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.

Just as a side note, a similar behavior can be obtained within the MATLAB environment with the following instructions:

```
%% MATLAB timer
tr=timer;tr.ExecutionMode='FixedRate';
tr.TimerFcn='algorithm';tr.Period=T;

tic
start(tr);
pause(T+Tf);
stop(tr);
toc
```

where the timer object `tr` is used.

Some practical considerations about rapid prototyping

Note that the resulting `main.c` file can be found as `main_timer.c` in the examples folder coming in the zip file. If your MATLAB application is enclosed in a function called "`algorithm`" then after generating the code, you can simply replace the main file with the provided one and build the executable.

Also, once the `main.c` file has been replaced, if the C code is re-generated by the `codegen` command, following some changes in the algorithm, the `main.c` file is not overwritten. Therefore, (provided that the name the function to be coded stays the same), one can just make changes in the algorithm, and the process to recompile, rebuild, and re-run the algorithm stays exactly the same.

Generating C code, building and running the executable on Linux

Assuming you have MATLAB, (as well as the MATLAB Coder and a suitable compiler) running on a Linux platform, the steps to generate C code from the MATLAB `algorithm()` function, build the executable, and running it, are essentially the same.

In MATLAB, the coder configuration object can be created as follows:

```
cfg = coder.config('exe');
cfg.CustomSource = 'main.c';
cfg.CustomInclude = './codegen/exe/algorithm/examples';
```

As previously explained, the first line specifies that the target is an executable running on the same OS as MATLAB, the second and third lines specify that we want an example `main.c` to be generated as well, and placed in the "examples" folder under the folder `codegen/exe/algorithm/`. An important, and unfortunately very easy to forget, difference with the Windows case is that the slash character "/" must be used instead of the backslash one "\".

After this, the following line generates the C code for the MATLAB `algorithm`() function, along with all the required object in the appropriate folders:

```
codegen algorithm -c -config cfg
```

To build the executable, one needs to execute the make file `algorithm_rtw.mk` in the `codegen/exe/algorithm` 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 ./codegen/exe/algorithm
make -f algorithm_rtw.mk
cd ../../..
```

If everything goes well, the executable `algorithm` should be in the current folder, and running it from the Linux shell should produce the output for the initialization followed by just one step:

```
% ./algorithm> ./algorithm
Init: Par1= 11.000000
Init: Par2= 9600.000000
Step: Output: 20.000000
```

Both the `algorithm.c` and the `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 `main_algorithm()` function as a timer callback (i.e. signal handler) can be easily accomplished in Linux by, for example, performing the following modifications to the original main file:

1) Several libraries which contain the required APIs need to be included. This can be done by inserting the following lines in the main file 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 `main_algorithm()` 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, main_algorithm);
```

just after the `algorithm();` call in (and before the end) of the "`static void main_algorithm(void)`" function definition. This will result in the following definition:

```
static void main_algorithm(void)
{
    /* Call the entry-point 'algorithm'. */
    algorithm();
    signal(SIGALRM, main_algorithm);
}
```

3) The timer structure needs to be defined. This can be done by inserting the following line just after (or just before) the `(void)argc;` line:

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

4) The `main_algorithm();` line, needs to be replaced with the following set of instructions:

```
/* Upon SIGALRM, call main_algorithm() */
if (signal(SIGALRM, (void(*)(int)) main_algorithm) == 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_rtw.mk` from the Linux shell as shown before) with this new `main.c` file in place, and running the new `./algorithm` 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 `main.c` file resulting from the above modifications can be found as `main_ml_timer_linux.c` in the examples folder coming in the zip file.

If your MATLAB application is enclosed in a function called `algorithm`, then after generating the code you can simply replace the main file with the provided one and build the executable.

Also, as mentioned for the windows case, once the `main.c` file has been replaced, if the C code is re-generated by the `codegen` command, following some changes in the algorithm, the `main.c` file is not overwritten. Therefore, if the name the function to be coded (in this case "`algorithm`") stays the same, one can just make changes in the algorithm, and then recompile, rebuild, and re-run the algorithm using exactly the same instructions.

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.