Lab Tutorial and Exercise 3

Using TrueTime to Simulate Embedded Control Systems

Developing embedded systems requires many types of skills and knowledge, such as C programming, configuration and optimization of the compiler, the operating system, pins of the target microprocessor, interfaces to the sensor and the actuator, driver programs of the devices, the network communication protocol, etc. It is challenging to master all these knowledge in a short time, and, worse, one has to learn them again if the target platform is changed. In the tight schedule of the course, we spend more time on learning these hands-on skills of the platform rather than the system approach of designing and analysing embedded control systems.

One learning objective of the course is to use the model-based technologies to reduce the development efforts on real hardware and system implementation. In the labs, we shall work with a Matlab based simulator to learn and practice the knowledge on distributed real-time embedded control system. The advantage is that we can skip the tedious tasks of configuring the specific operating system and the hardware while focusing on the generic principles of developing real-time embedded systems. Consequently we can readily practice the theory learned in the lectures and have fun to experiment our own designs.

The simulator is called *TrueTime*, developed by the Department of Automatic Control at Lund University[1]. TrueTime is a Simulink toolbox that can simulate the timing effects on embedded software of real-time OS, microprocessor, time delays, scheduling strategies, and network communication. The configuration process and task execution commands are very similar to common RTOS, such as Rubus, but more concise. The experiences gained through the TrueTime simulations are beneficial when you develop real embedded systems.

The objectives of this tutorial are:

- to learn the functions of the TrueTime toolbox and understand its working principle,
- to build a TrueTime model for a single-node multiple task real-time application,
- to investigate how time delays may deteriorate the control performance,

[1] http://www.control.lth.se/truetime/

- to monitor deadline overrun in TrueTime simulation.

The TrueTime tutorial in this hand-out is adapted from the online tutorial developed by the Department of Automatic Control at Lund University. We use TrueTime version 2.0 beta 7. This lab only covers the basics. For further knowledge on TrueTime, please read the TrueTime reference manual.

Imagine a simple position control system with an ideal motor, i.e., the motor speed command is immediately achieved by the motor. Taking the speed command as input and position as output, the model of the ideal motor is simply an integrator, as in **Figure** 1. The target position is a square wave between 0 and 1. Figure 2 shows the parameters of the square wave signal. The controller is a simple proportional control with the sampling period of Ts = 0.5 second. The control input to the motor is contaminated by a white noise, which simulates the quantization error of A/D and D/A converters. The parameters of the "Band-Limited White Noise" block are in the following table.

You may build the model yourself or download it from the course website. Run the motor and plot the motor position

Noise power	**Sample time**	**Seed**
[0.1]	Ts	[23341]

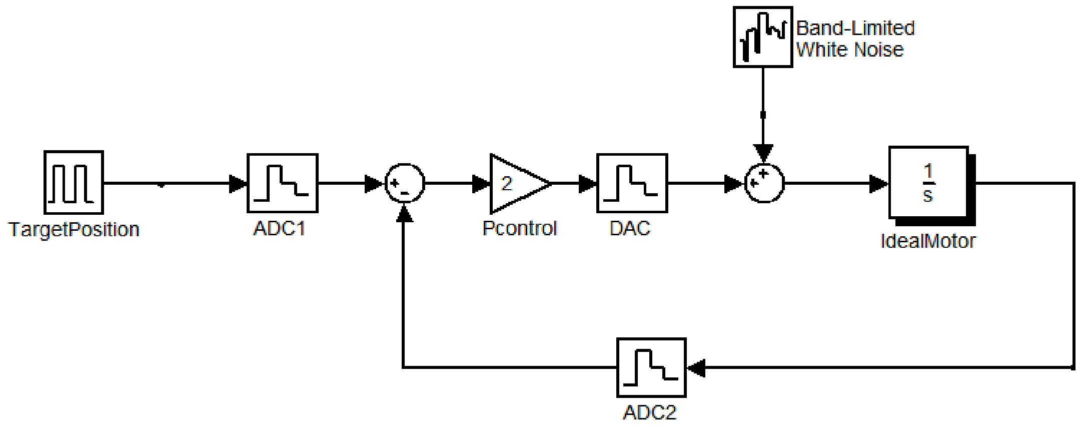
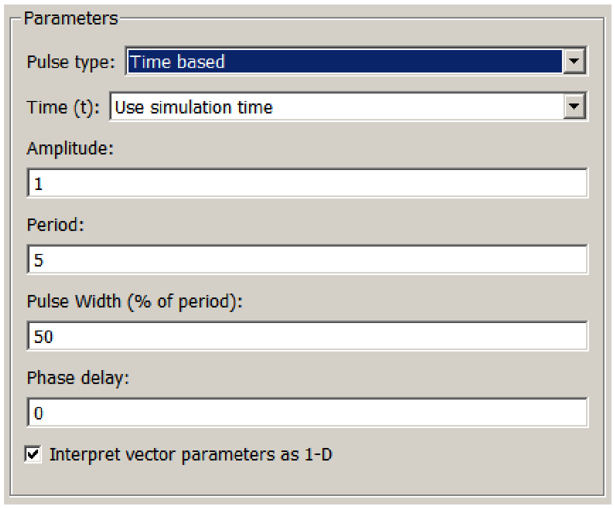


Figure 1: Position Control of an Ideal Motor


Parameters

Pulse type: Time based

Time (t): Use simulation time

Amplitude: 1

Period: 5

Pulse Width (% of period): 50

Phase delay: 0

[✓] Interpret vector parameters as 1-D

Figure 2: Parameters of the Square Wave

Through the following steps, you will build a TrueTime model of the same controller and observe the difference between the ideal controller and its implementation. The primary reason for the difference is the unavoidable time delay in digital processors.

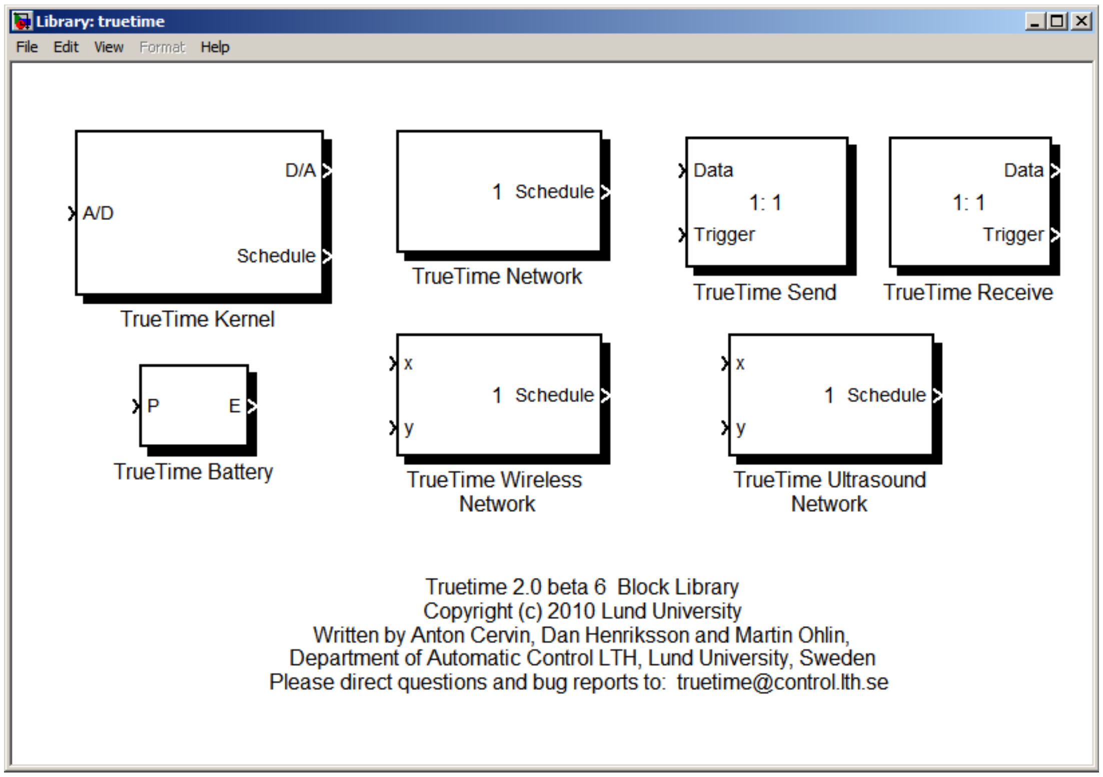
1 TrueTime Block Library

Step 1: Open the TrueTime block library

Start Matlab and type the command

```
>> truetime
```

The TrueTime block library will then prompt as in Figure 3. If an error occurs, the reason is probably due to defected installation of the toolbox. Please contact the lab assistant for help. The functions of two most relevant blocks are briefly introduced below. Deeper knowledge on them will be gained through a series of labs. The working principles of other blocks are very similar to these two.



Truetime 2.0 beta 6 Block Library
Copyright (c) 2010 Lund University
Written by Anton Cervin, Dan Henriksson and Martin Ohlin,
Department of Automatic Control LTH, Lund University, Sweden
Please direct questions and bug reports to: truetime@control.lth.se

Figure 3: TrueTime Block Library

- *TrueTime Kernel* simulates a single micro-computer equipped with a generic real-time operating system (RTOS). All RTOS features discussed in the course are supported by it.
- *TrueTime Network* simulates medium access control and data transmission in a local area network such as CAN and Ethernet. Multiple TrueTime kernels, namely nodes, may be connected to one TrueTime network.

Step 2: Use the TrueTime Kernel block

Drag the TrueTime kernel block from the TrueTime library window, illustrated in Figure 3, to the Simulink model in **Figure** 1. This kernel simulates the computer executing the control tasks. The input and output ports of the TrueTime kernel block are explained below.

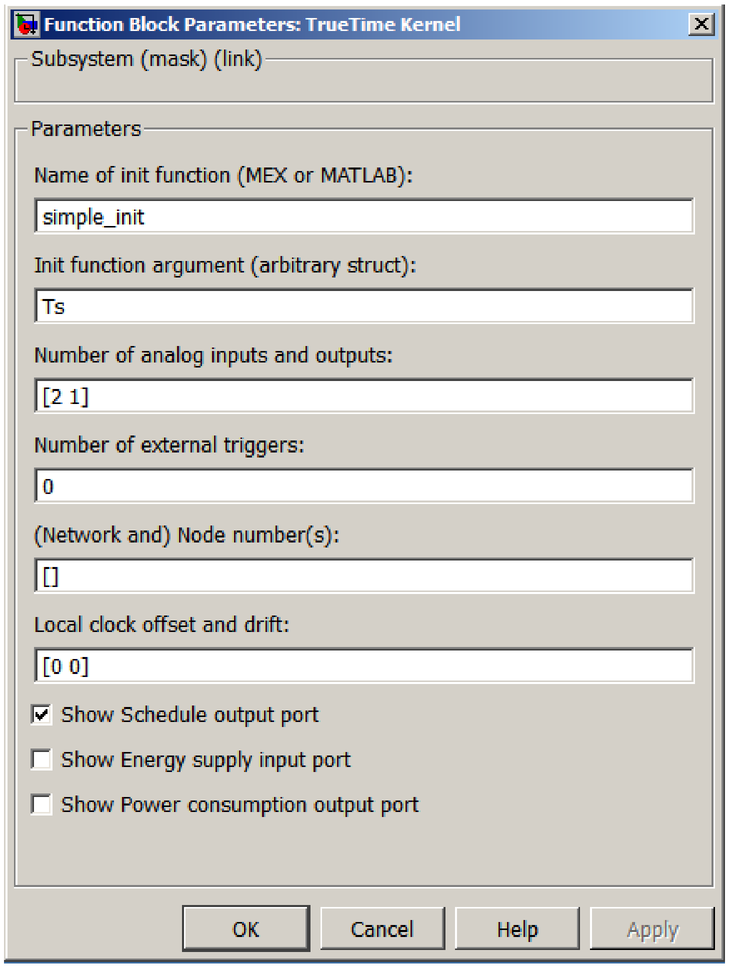
- A/D: The Analog to Digital converter provides sensor data and operator commands to the microprocessor. If you have multiple analog inputs, you can specify the number of inputs in the dialog window in **Figure 4** and use a multiplexer (mux) to collect all inputs as a vector.
- D/A: The Digital to Analog converter to transmit the digital control command to the analog signal applied to the actuators. The number of outputs is specified in **Figure 4** and you use a demux block to access each output signal.

- Schedule: The port plots the task execution status on the real-time kernel. Three different values are used for each task, representing three different statuses. A high value indicates that the task is running, a medium value indicates that the task is ready but preempted, and a low value means that the task is idle and not ready to run.

Step 3 Customize the TrueTime kernel block

Double click on the TrueTime kernel block. A dialog window will appear like the one in Figure 4. We customize the block with the following parameters.

- "Name of init function" The name of an m-file at the same path of the Simulink model. The m-file specifies the task scheduling method on the kernel, such as fixed-priority (FP), rate monotonic/deadline monotonic (DM), and earliest-deadline first (EPF). In this tutorial, the m-function is "simple_init", which will be elaborated shortly.
- "Init function argument": An arbitrary data passed into the above-specified init function, e.g., simple_init, as the input argument. In this example, the sampling period Ts (= 0.5) is the argument.
- "Number of analog inputs and outputs": A vector of 2 scalars. The first one is the number of input ports and the second one the number of output ports.
- "Number of external triggers": An external trigger simulates an interrupt source to the kernel.
- "(Network and) Node number(s)": If the kernel is connected to a network, either TrueTime Network, TrueTime Wireless Network, or TrueTime Ultrasound Network, then the parameter indicates the node ID number of the kernel.
- "Local clock offset and drift": A vector of 2 scalars. The first one, clock offset, indicates a constant time offset of the local clock on the kernel with respect to the simulation time; the second one, clock drift, indicates if the local clock runs faster or slower than the simulation time.


Function Block Parameters: TrueTime Kernel

Subsystem (mask) (link)

Parameters

Name of init function (MEX or MATLAB): simple_init

Init function argument (arbitrary struct): Ts

Number of analog inputs and outputs: [2 1]

Number of external triggers: 0

(Network and) Node number(s): []

Local clock offset and drift: [0 0]

[✓] Show Schedule output port

[ ] Show Energy supply input port

[ ] Show Power consumption output port

Figure 4: Dialog to Customize a TrueTime Kernel Block

2 TrueTime Simulation Model

In this part we build the TrueTime simulation model with the TrueTime kernel block, which represents a computer with RTOS and necessary A/D and D/A converters to communicate with the plant. (Imagine it as a real embedded control system)

Figure 5 shows the TrueTime control model below the ideal proportional control model, with the TrueTime kernel highlighted in green. There are two input signals to the proportional controller: the target position and the motor position measurement. As illustrated in the figure, both signals enter the A/D port of the TrueTime kernel block. The proportional controller has only one control output: the motor speed demand. As in Figure 5, the D/A port of the TrueTime kernel has only one output signal.

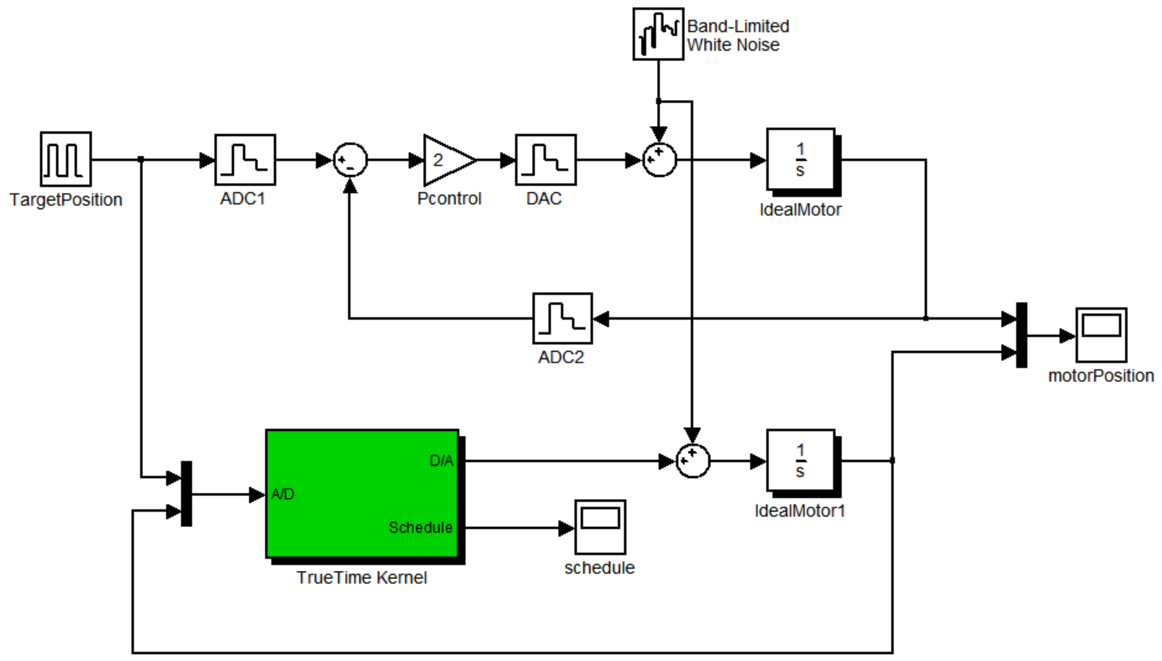


Figure 5: Motor Control with the TrueTime Kernel

Imagine the TrueTime kernel as a real computer. We have set up the sensor input (A/D ports) and actuator output (D/A ports), and have connected the computer to the control plant, i.e., the ideal motor. How can we instruct the computer to perform the feedback proportional control given in **Figure** 1? The answer is program. We must supply the computer with the code that realizes the control function.

3 TrueTime Code

When programing an embedded control system, two types of programs are needed.

1. The initialization program to configure the real-time operation system (RTOS). The program instructs the RTOS on the number and names of executing tasks, their periods, deadlines, priorities, and other task parameters, interrupt triggers, interrupt handlers, task communication methods, the scheduling strategy, etc.
2. The application program realizing the functions of the tasks and interrupt handlers. These programs read the sensor inputs, compute the control outputs, and command the actuators.

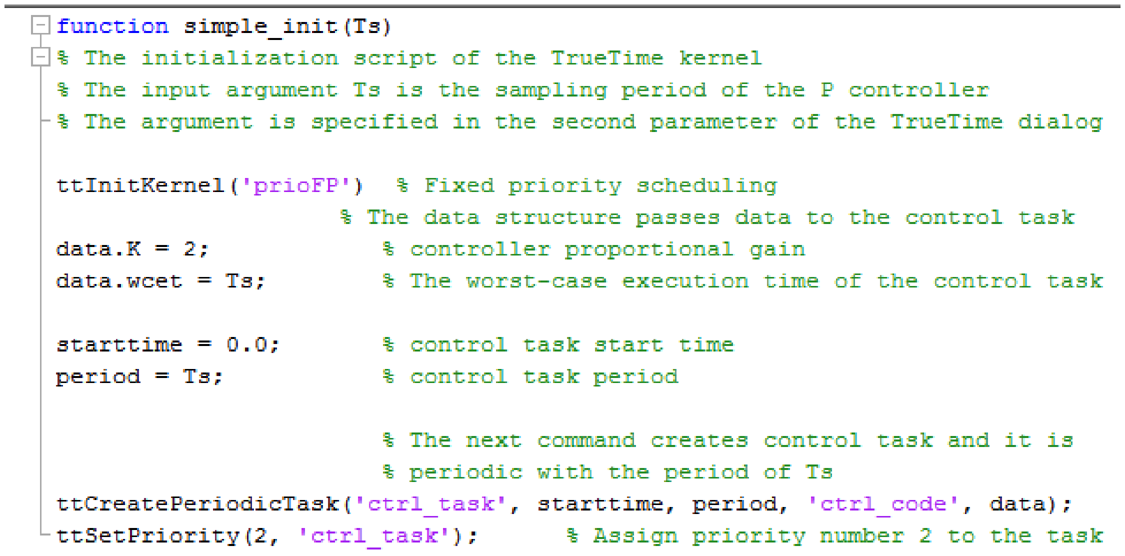
With TrueTime simulation, these programs may be written in either Matlab or C++. The application program may even directly call Simulink diagrams. This tutorial teaches Matlab only. The interested reader can refer to the TrueTime manual for other options.

We first write the initialization script for the ideal motor control problem. Recall the customization of the TrueTime kernel in Figure 4. The init function is named “`simple_init`”.

Create a new m-file in the same folder as the Simulink mode and save it as “simple_init.m”. The code of the file is shown in Figure 6. The first line `ttInitKernel` specifies the fixed-priority scheduling (FP), which always executes the highest-priority one among all ready tasks. Priorities are indicated by positive integer numbers. A smaller number indicates a higher priority. TrueTime supports three scheduling strategies.

1. ‘prioFP’: fixed-priority scheduling
2. ‘prioDM’: deadline-monotonic scheduling
3. ‘prioEDF’: earliest-deadline-first scheduling

The interpretations of these scheduling strategies are taught in the lecture. You may find that the rate-monotonic (RM) scheduling is missing in the list, but it can be equally realized by DM if we assume the deadline of a periodic task as its period.



```
function simple_init(Ts)
% The initialization script of the TrueTime kernel
% The input argument Ts is the sampling period of the P controller
% The argument is specified in the second parameter of the TrueTime dialog

ttInitKernel('prioFP')   % Fixed priority scheduling
                        % The data structure passes data to the control task
data.K = 2;                % controller proportional gain
data.wcet = Ts;            % The worst-case execution time of the control task

starttime = 0.0;           % control task start time
period = Ts;               % control task period

                           % The next command creates control task and it is
                           % periodic with the period of Ts
ttCreatePeriodicTask('ctrl_task', starttime, period, 'ctrl_code', data);
ttSetPriority(2, 'ctrl_task');      % Assign priority number 2 to the task
```

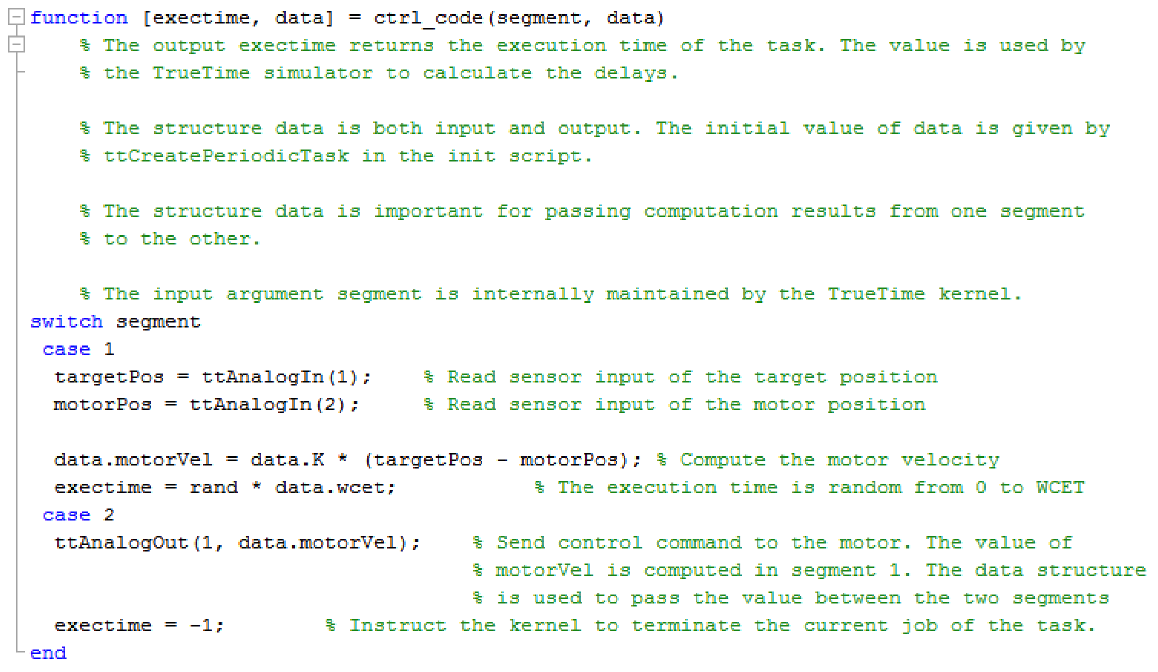
Figure 6: The Code of simple_init.m

The second and third lines create a new structure named data with two attributes: K and wcet. The structure is later passed to the control task code as an argument. In this example, the control gain and the worst-case execution time are provided to the control task. In particular, the execution time of the control task has to be supplied by us because the Matlab simulation does not know the execution time of the control task. This must be measured or estimated by the code developer on the actual platform.

The function `ttCreatePeriodicTask` creates the control task, named '`ctrl_task`', and specifies its period as Ts (= 0.5). The code to implement the task is specified as the m-function '`ctrl_code`' saved in the m-file 'ctrl_code.m' at the same folder. Subsequently we assign the priority number 2 to the control task by function `ttSetPriority`.

Now that we have created the control task and its execution environment with the RTOS, we proceed to supply the details of the control task, i.e., the code function 'ctrl_code.m' illustrated in Figure 7. The fundamental concept with the code function is the variable `segment`. The writer of the simulation code may subjectively split the function into multiple segments and specify the execution time for every segment. The sum of the execution times of all segments is the execution time of the task if not pre-emption occurs. This allows the user to investigate the timing effect of each code segment.

The most important usage of the segment structure is to simulate the blocking operations, such as locking a semaphore or writing a full message queue, which may temporarily block a running task until a relevant event arises. We shall elaborate this topic in a future lab.



```
function [exectime, data] = ctrl_code(segment, data)
    % The output exectime returns the execution time of the task. The value is used by
    % the TrueTime simulator to calculate the delays.

    % The structure data is both input and output. The initial value of data is given by
    % ttCreatePeriodicTask in the init script.

    % The structure data is important for passing computation results from one segment
    % to the other.

    % The input argument segment is internally maintained by the TrueTime kernel.
switch segment
 case 1
  targetPos = ttAnalogIn(1);     % Read sensor input of the target position
  motorPos = ttAnalogIn(2);      % Read sensor input of the motor position

  data.motorVel = data.K * (targetPos - motorPos); % Compute the motor velocity
  exectime = rand * data.wcet;            % The execution time is random from 0 to WCET
 case 2
  ttAnalogOut(1, data.motorVel);     % Send control command to the motor. The value of
                                     % motorVel is computed in segment 1. The data structure
                                     % is used to pass the value between the two segments
  exectime = -1;        % Instruct the kernel to terminate the current job of the task.
end
```

Figure 7: The Code of ctrl_code.m

The value of the variable `segment` is internally maintained by the TrueTime kernel. When the task is ready, `segment` is 1. The kernel executes the task if it has the highest priority; otherwise, the task is preempted. When the task runs, it executes all code between "`case 1`" and "`case 2`" and then exits the function according to the definition of the switch block. Now the value of `segment` is increased by 1 and the task is still ready. The TrueTime kernel again chooses the highest-priority task among all ready tasks. In this example, "`ctrl_task`" resumes to execute the code from "`case 2`" to "`end`". In the last segment the execution time is set negative, which indicates the end of the task execution, i.e., no more segments exists and the task becomes idle.

We have now supplied the "computer" with the necessary "software". It is the time to examine the difference between the TrueTime control block and the plain Simulink control block.

4 Simulation and Comparison

Run simulation with the default solver configuration. Check the plots of `motorPosition` and `schedule` in Figure 5. Explain the meaning of the plots.

1. How do you verify if the control task "`ctrl_task`" really executes every Ts (= 0.5) second?

2. How do you determine the execution time of every instance of the control task?

3. With respect to overshoot and rise time, which controller has better performance? What is the reason for this result?

The Simulink controller is ideal because there is no time delay between the sensor input and the actuator output. The computation time of the controller is ignored. In contrast, the TrueTime controller is more realistic, because various time delays are considered in TrueTime simulation. For example, the execution time delay is specified by the variable `exectime` in the code of the control function in Figure 7.

Experiment: Please reduce the worst-case execution time (WCET) of the control task and see if the control performance is improved.

5 Time Delay Caused by CPU Sharing

In most cases the computer contains multiple concurrent tasks sharing the same CPU. At each time instant, the CPU can be allocated to only 1 task, and the others, if ready, must wait. This is called pre-emption and will be taught in the lecture. Pre-emption is another major source of time delay in control implementation by the digital computer.

This lab briefly touches this aspect. Intensive study will follow in the next TrueTime lab. In addition to `ctrl_task` defined by the initialization code in Figure 6, we introduce a disturbance task, `disturb_task`, which does nothing but occupies CPU time. Add the following two lines at the end of the function `simple_init`.

```
ttCreatePeriodicTask(‘disturb_task’, 0, Ts, ‘distrub_code’);
ttSetPriority(1, ‘disturb_task);
```

1. What is the start time of the disturbance task? What is its period?

2. What is the priority of the disturbance task? Is it more important than the control task `ctrl_task`?

3. Write the code of `disturb_code` so that the execution time is always 0.1 second. The function must be saved in a separate file named "disturb_code.m" in the same folder of the model and files.

```
function [exectime, data] = disturb_code(segment, data)
switch segment
  case 1
    % Your code
  case 2
    % Your code
end
```

Run simulation with the new code and analyse the results.

4. After adding `disturb_task`, is the control performance of the TrueTime controller better or worse? Please explain the reason.

5. Please explain the plots of the `schedule` scope. How many plots can you see? Which one corresponds to `disturb_task`/`ctrl_task`? Explain why the plots may show *low*, *middle*, or *high* values.

6. Does `disturb_task` or `ctrl_task` miss some deadlines? Please justify your answer.

Experiment: Please modify the code so that the priorities of the tasks `disturb_task` and `ctrl_task` are reversed. Answer questions 4 and 6 again.

6 Deadline Overrun Handler

An important service from all decent RTOS is the runtime monitoring of tasks' timing constraints, e.g., deadline and worst-case execution time (WCET). The RTOS monitors the execution status of all tasks. If a task violates a timing constraint, the RTOS triggers an interrupt handler that has higher priority than other tasks. The action of the handler is programmed by the developer.

The TrueTime kernel faithfully simulates this mechanism. This lab discusses the deadline overrun only, but the same procedure is applicable to WCET overrun by using the corresponding TrueTime functions. To monitor the deadline overrun of a task, say `ctrl_task`, we must attach a *deadline overrun handler* (DL) to the task. The DL is another concurrent thread that must be created in the TrueTime kernel.

To attach a DL to `ctrl_task`, we must perform the following 4 actions in any order:

Create the deadline overrun handler in the init function, `simple_init` by anyone of the following commands.

```
ttCreateHandler(handlerName, priority, codeFcn)
ttCreateHandler(handlerName, priority, codeFcn, data)
```

- `handlerName`: Name of handler. Must be a unique, non-empty string.
- `priority`: Priority of the handler must be non-negative. Note that in TrueTime kernel the interrupt handler always pre-empts ordinary tasks regardless of the priority number.
- `codeFcn`: The code function of the handler. It is the non-empty string of the m-file name.
- `data`: An arbitrary data structure representing the local memory of the handler.

Attach the DL to the monitored task `ctrl_task` in `simple_init` with the command

```
ttAttachDLHandler(taskName, handlerName)
```

- `taskName`: The name of the monitored task, i.e., `ctrl_task` for this lab.
- `handlerName`: The name of the interrupt handler specified above.

Write the code function of the DL, who does nothing but print a warning message on the Matlab command window. Set the execution time of the DL as 0.001 second. Remember to follow the code structure as `disturb_code` and `ctrl_code`.

Experiment: Run the simulation with the setup that `disturb_task` has higher priority than `ctrl_task`. Does the DL catch any deadline overrun? Please specify when the deadline overruns happen on the `schedule` plots.

7 Summary

In this lab, we have learned the basics of using TrueTime to digital control implementation, whose major difference from the ideal controller is various time delays introduced by computation time and resource sharing. This lab is focused on the computation time delay and also briefly discusses the CPU sharing of multiple tasks and interrupt handler.

After the lab, you should master the basic skills of

- connecting the TrueTime kernel block with other Simulink function blocks;
- initializing the TrueTime kernel block by both customizing the dialog parameters and writing the initialization function;
- selecting the scheduling strategy with the command `ttInitKernel`;
- defining the tasks and interrupt handlers in the initialization function;
- assigning priorities to the tasks and interrupt handlers;
- writing the code function of tasks and handlers with the mandatory code structure;
- interpreting the plots of the `schedule` outport of the TrueTime kernel block.

In future labs, we shall learn more advanced functions of TrueTime. The skills leaned in this lab is the foundation for your further study.