The Real-Time Control Systems Simulator -Reference manual

Cervin, Anton

2000

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Cervin, A. (2000). *The Real-Time Control Systems Simulator -Reference manual*. (Technical Reports TFRT-7592). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
1

ISSN 0280–5316
ISRN LUTFD2/TFRT--7592--SE

The Real-Time Control Systems Simulator

Reference Manual

Anton Cervin

Department of Automatic Control
Lund Institute of Technology
April 2000

Department of Automatic Control **Lund Institute of Technology** **Box 118** **SE-221 00 Lund Sweden**	*Document name* INTERNAL REPORT *Date of issue* April 2000 *Document Number* ISRN LUTFD2/TFRT--7592--SE
Author(s) Anton Cervin	*Supervisor* *Sponsoring organisation*

Title and subtitle

The Real-Time Control Systems Simulator—Reference Manual

Abstract

A Matlab/Simulink-based simulator for real-time control systems is described. The simulator facilitates co-simulation of plant dynamics, controller code execution, and real-time scheduling. Three examples are given. The kernel primitives are described in detail.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280–5316		*ISBN*
Language English	*Number of pages* 25	*Recipient's notes*
Security classification		

Contents

1. Introduction

This report describes in further detail the usage of the real-time control systems simulator presented in [Eker and Cervin, 1999]. Notice that some minor differences from the paper exist—some fields of the real-time kernel data structures have changed names, the contents of the intitialization script is slightly changed, etc.

2. Using the Toolbox

2.1 Creating a Computer-Controlled System

The Real-Time Kernel block models a computer with a real-time kernel. By connecting the inputs and outputs to a model of a physical plant, a computer-controlled system is formed, see Figure 1. As also shown in the figure, it is often convenient

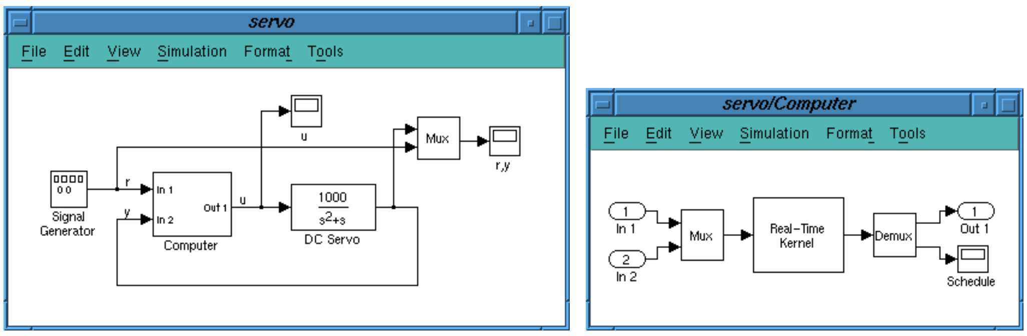

Figure 1 The inputs and outputs of the Real-Time Kernel blocks are connected to form a computer-controlled system. The Real-Time Kernel block is conveniently placed in subsystem.

to put the Real-Time Kernel block and its connectors in a subsystem (here named "Computer").

Since all information about a Real-Time Kernel block is stored locally, it is possible to include several kernel blocks in the same SIMULINK model.

The number of inputs and outputs of the kernel block is dynamic and depends on the contents of the initialization function, which is given as a parameter to the kernel block. The Mux and Demux blocks in Figure 1 must agree with the system structure returned by the initialization function, `rtsys`. The number of inputs of the kernel block is equal to `rtsys.nbrOfInputs` (given by the user). The number of outputs of the kernel block is equal to

$$\texttt{rtsys.nbrOfOutputs} + \texttt{rtsys.nbrOfTasks}(1 + \texttt{rtsys.nbrOfMutexes})$$

(`rtsys.nbrOfOutputs` is given by the user, while `rtsys.nbrOfTasks` and `rtsys.nbrOfMutexes` are determined by the simulator.) In short, the schedule generates `rtsys.nbrOfTasks` extra outputs, and so does each mutex graph. As an example, consider again the system in Figure 1. If there were four tasks, three output channels, and two mutexes, the number of outputs of the Demux block could be specified as `[1 1 1 4 4 4]`. Details about the `rtsys` structure are found in Section 7.

2.2 Running a Simulation

When running a simulation, the Real-Time Kernel block is treated like any other discrete-time block in the SIMULINK model. At the start of a simulation, the initialization function of the kernel block is evaluated. During simulation, the code segment functions of the tasks in the kernel are executed repeatedly.

The kernel block produces a schedule graph, where the execution of each task is described by a signal. When the signal of a task is high, the task is running, and when the signal is medium, the task is in the ready queue (but not running). When the signal is low, the task is sleeping (or, to be more precise, not in the ready queue).

If there are mutexes in the system, each mutex produces a mutex graph, where the locking of the mutex of each task is described by a signal. When the signal of a task is high, the mutex is locked by the task, and when the signal is medium, the task is waiting to lock the mutex. When the signal is low, the task is not attempting to lock the mutex.

2.3 Hints

You might experience that nothing changes in the simulations, even though you have made changes in the initialization or the code segment files. To make the changes take effect, issue the command

```
>> clear functions
```

To force MATLAB to reload all functions at the start of each simulation, issue the command (assuming that the model is named servo)

```
>> set_param('servo','StartFcn','clear functions')
```

and save the model.

Similarly, you can attach plot functions and other functions at the end of a simulation:

```
>> set_param('servo','StopFcn','plotresults')
```

3. Writing a Code Segment

The code of a task is built from a number of code segments, which are executed in sequence every period. Often, one or two code segments are enough to model the timely behavior of a control task.

A code segment is implemented as a MATLAB function. The function is called twice during the simulated execution of a segment, once to execute the `enterCode`, and once to execute the `exitCode`.

The syntax of a code segment is best illustrated by a small example. Consider the following segment that implements a P-controller:

```
function [exectime,states] = pController(flag,states,params)
switch flag,
  case 1, % enterCode
```

```
    r = analogIn(1);
    y = analogIn(2);
    states.u = params.K*(r-y);
    exectime = 0.002;
  case 2, % exitCode
    analogOut(1,states.u)
end
```

The `flag` indicates which part should be executed, while `states` and `params` are user-defined data structures. In this example, the control signal `u` is a state, and the controller gain `K` is a parameter.

When `flag=1` (`enterCode`), the function should return `[exectime,states]`, i.e. the execution time of the segment and the new states. When `flag=2` (`exitCode`), the function should return nothing. Note that assigning new states in the `exitCode` has no effect.

`analogIn` and `analogOut` are real-time primitives. Further primitives include functions to change the task attributes dynamically, and resource-access primitives. A complete list is found in Section 9.

Generally, all computations should be placed in the `enterCode`. The `exitCode` could be used to write new output signals, unlock resources, and perhaps to change task attributes (priority, deadline, etc.) before the next segment.

4. Writing an Initialization Function

The initialization function defines the tick-size of the kernel (i.e. the resolution of the simulation), what scheduling policy should be used, the number of input and output channels, and the tasks that should execute in the kernel. Optionally, the user may also define mutexes and events, and set the initial values of the output channels.

Consider the following example, where two PID control tasks are created:

```
function rtsys = mymodel_init
%% General settings
rtsys.tickSize = 0.001;
rtsys.prioFun = 'prioEDF';
rtsys.nbrOfInputs = 4;
rtsys.nbrOfOutputs = 2;
%% Create control tasks
T = 0.005; % Sampling period = deadline
states = ...
params = ...
pidCode1 = code({'pid'},states,params);
pidTask1 = task('pidTask1',pidCode1,T,T);
T = 0.010; % Sampling period = deadline
states = ...
params = ...
pidCode2 = code({'pid'},states,params);
pidTask2 = task('pidTask2',pidCode2,T,T);
```

```
rtsys.tasks = {pidTask1,pidTask2};
```

The initialization function returns a data structure, `rtsys`. A complete description of this structure is found in Section 7.

A task is created in two steps. First, a code structure is created by a call to `code`. The first argument is a list of code segments. The second and third arguments are arbitrary data structures that give the initial states and the parameters of the code.

Next, a task structure is created by a call to `task`. The supplied arguments are a name, the code structure, the period, and the relative deadline. As optional arguments, a release offset and a fixed priority may also be given. Further details about the `code` and `task` functions are found in Section 8.

5. Writing a Priority Function

At each clock tick, the kernel sorts the tasks in the ready queue and selects the highest-priority task to be the running task. The priority of each task is determined dynamically through a call to priority function, which is specified in the initialization script. The function may perform any calculation on the task attributes, including the code attributes. For a complete description of the data structures, see Section 7.

It should be noted that a low priority value denotes a high priority.

Consider for instance the predefined priority-function for earliest-deadline-first scheduling, `prioEDF`:

```
function prio = prioEDF(task)
prio = task.release + task.deadline;
```

The function returns the absolute deadline of the task, which is computed as the release-time plus the relative deadline. The earlier the absolute deadline, the higher the priority will be.

As another example, assume that all tasks in a system has a state `error` that keeps track of the current control error. The user could define a priority function that gives the highest priority to the task with the largest control error:

```
function prio = prioError(task)
prio = -task.code.states.error;
```

(It turns out that this is generally not a very good scheduling policy for a set of control tasks.)

6. Examples

Three examples are included in the simulator archive. The first example concerns PID control of DC servos, the second describes sub-task scheduling of control tasks, and the third describes the use of mutexes and events.

6.1 PID Control of DC Servos

Consider PID control of a DC servo, described by the continuous-time transfer function

$$G(s) = \frac{1000}{s(s+1)}.$$

One possible discrete-time implementation of the PID controller, that includes filtering of the derivative part, is

$$\begin{aligned} P(t) &= K(r(t) - y(t)), \\ I(t) &= I(t-h) + \frac{Kh}{T_i}(r(t) - y(t)), \\ D(t) &= a_d D(t-h) + b_d(y(t-h) - y(t)), \\ u(t) &= P(t) + I(t) + D(t), \end{aligned}$$

where $a_d = \frac{T_d}{Nh+T_d}$ and $b_d = \frac{NKT_d}{Nh+T_d}$.

The controller is designed (i.e. K, T_i, and T_d are chosen) to give the system the closed-loop bandwidth $\omega_c = 20$ rad/s and the relative damping $\zeta = 0.7$.

Code Segment The controller is implemented as a single code segment (`pid.m`):

```
function [exectime,s] = pid(flag,s,p)
switch flag,
  case 1, % enterCode
    r = analogIn(p.rChan);
    y = analogIn(p.yChan);
    P = p.K*(r-y);
    I = s.Iold+p.K*p.h/p.Ti*(r-y);
    D = p.Td/(p.N*p.h+p.Td)*s.Dold+p.N*p.K*p.Td/(p.N*p.h+p.Td)*(s.yold-y);
    s.u = P + I + D;
    s.Iold = I;
    s.Dold = D;
    s.yold = y;
    exectime = 0.002;
  case 2, % exitCode
    analogOut(p.uChan,s.u);
end
```

Initialization Function In the SIMULINK model `servo.mdl`, a single DC servo should be controlled by a single PID control task. The model in shown in Figure 2. The corresponding initialization function looks like this (`servo_init.m`):

```
function rtsys = servo_init

%% General settings
rtsys.tickSize = 0.001;
rtsys.prioFun = 'prioRM';
rtsys.nbrOfInputs = 2;
rtsys.nbrOfOutputs = 1;
```

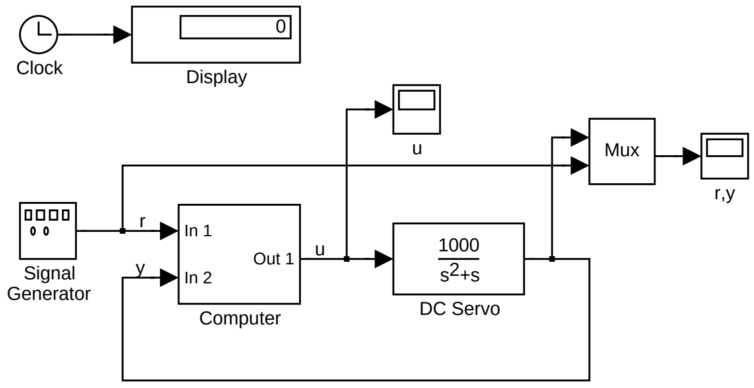

Figure 2 Example 1a. One DC Servo.

```
%% Create control task
T = 0.006; % Sampling period
states.Iold = 0;
states.Dold = 0;
states.yold = 0;
params.K = 0.96;
params.Ti = 0.12;
params.Td = 0.049;
params.N = 10;
params.rChan = 1;
params.yChan = 2;
params.uChan = 1;
params.h = T;
pidCode = code({'pid'}, states, params);
pidTask = task('pidTask', pidCode, T, T);
rtsys.tasks = {pidTask};
```

Single-Servo Experiments The following experiments illustrate the effects of sampling period and computational delay on control performance. Try the following:

1. Simulate the system for 2 seconds. Verify that the controller behaves as expected. Notice the computational delay of 2 ms in the control signal.
2. Increase the execution time of the PID controller to 6 ms (edit `pid.m`) and run another simulation. Notice that the performance gets worse.
3. Change the execution time back to 2 ms, and instead change the sampling period, first to 12 ms, then to 18 ms and run new simulations (edit `servo_init.m`). Notice that the response is oscillatory when $T = 12$ ms and that the system is very close to unstable when $T = 18$ ms.

Multiple-Servo Experiments The following experiments illustrate that control performance and scheduling performance (i.e. the ability to meet deadlines) are completely different things. In the SIMULINK model `threeservos.mdl`, three DC servos should be controlled by three PID control tasks executing in a single CPU. The model is shown in Figure 3. The corresponding initialization function is called `threeservos_init.m`. The controllers have the sampling periods 6 ms, 5 ms, and

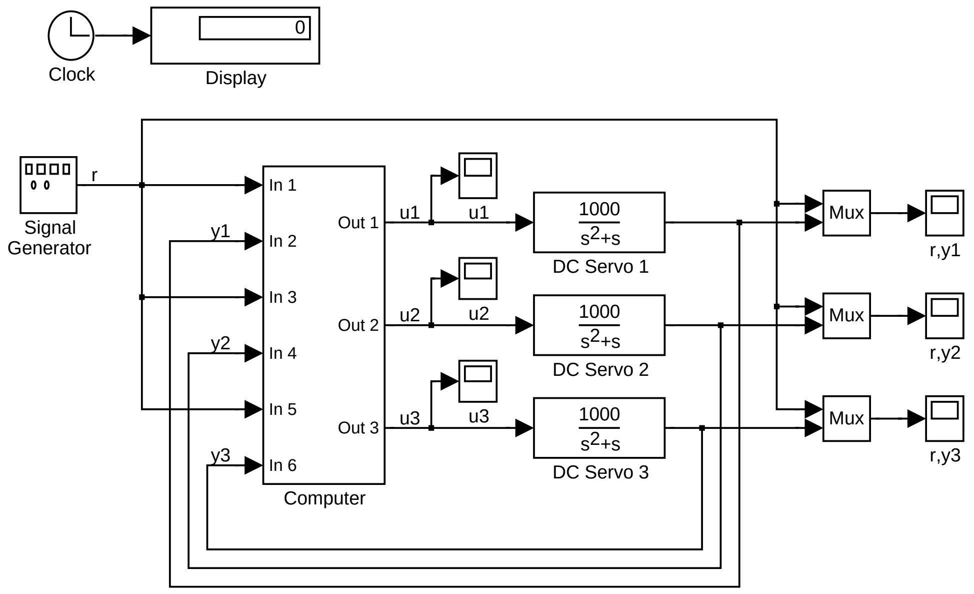

Figure 3 Example 1b. Three DC servos.

4 ms respectively. Since the execution time of the PID controller is 2 ms, the system is overloaded:

$$U = \sum_i \frac{C_i}{T_i} = \frac{2}{6} + \frac{2}{5} + \frac{2}{4} = 1.23 > 1.$$

Try the following:

1. Set the scheduling type to rate-monotonic (`prioRM`) and simulate the system for 2 seconds. Notice that Task 1 misses its deadlines and that the corresponding control loop becomes unstable. A plot of the results from the simulation is shown in Figure 4.
2. Change the scheduling type to earliest-deadline-first (`prioEDF`) and run another simulation. After an initial transient, *all* tasks miss their deadlines. The performance of all control loops is satisfactory, however. A plot of the results from the simulation is shown in Figure 5.

6.2 Sub-Task Scheduling of Control Tasks

This example illustrates that by scheduling the two main parts of a control algorithm—*Calculate Output* and *Update State*—as separate tasks, the performance of a set of control tasks may be improved, see [Cervin, 1999].

In the SIMULINK model `improved.mdl`, three inverted pendulums should be controlled by three control tasks executing in the computer. The model is shown in Figure 6.

The pendulum process is given by the continuous-time state-space description

$$\frac{dx}{dt} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$y = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

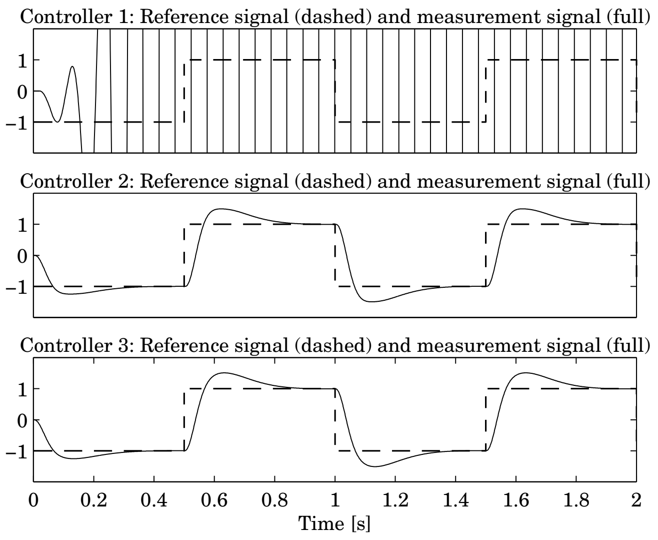

Figure 4 Simulation of the three servo controllers under rate-monotonic scheduling. Task 1 misses all its deadlines and the corresponding control loop is unstable.

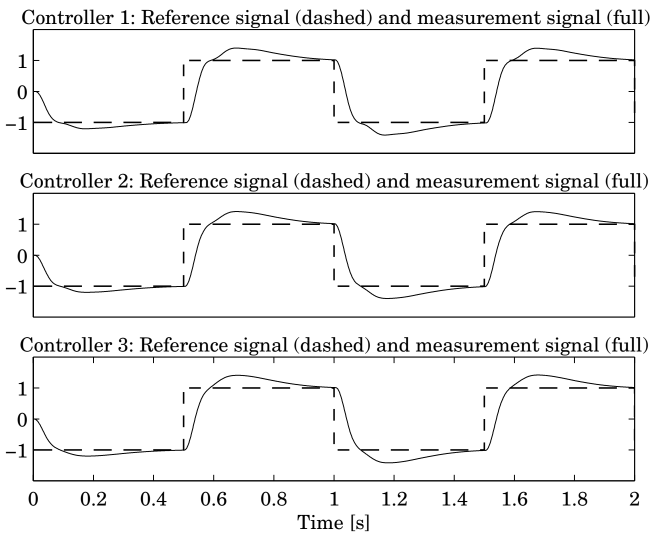

Figure 5 Simulation of the three servo controllers under earliest-deadline-first scheduling. After an initial transient, all tasks miss all their deadlines, but the performance is satisfactory for all controllers.

Each pendulum controller has a different closed-loop specification and a different sampling interval. A discrete-time state-feedback controller with an observer (with direct term) is given by

$$\begin{aligned} \hat{x}(k|k) &= (I - KC)(\Phi\hat{x}(k-1|k-1) + \Gamma u(k-1)) + Ky(k) \\ u(k) &= -L\hat{x}(k|k) \end{aligned}$$

The calculations can be rearranged to minimize the computational delay. Introduce the controller state $w(k) = \hat{x} - Ky(k)$. The controller can now be written (see Problem 4.7 in [Åström and Wittenmark, 1997])

$$\begin{aligned} w(k+1) &= \Phi_0 w(k) + \Gamma_0 y(k) \\ u(k) &= C_0 w(k) + D_0 y(k) \end{aligned}$$

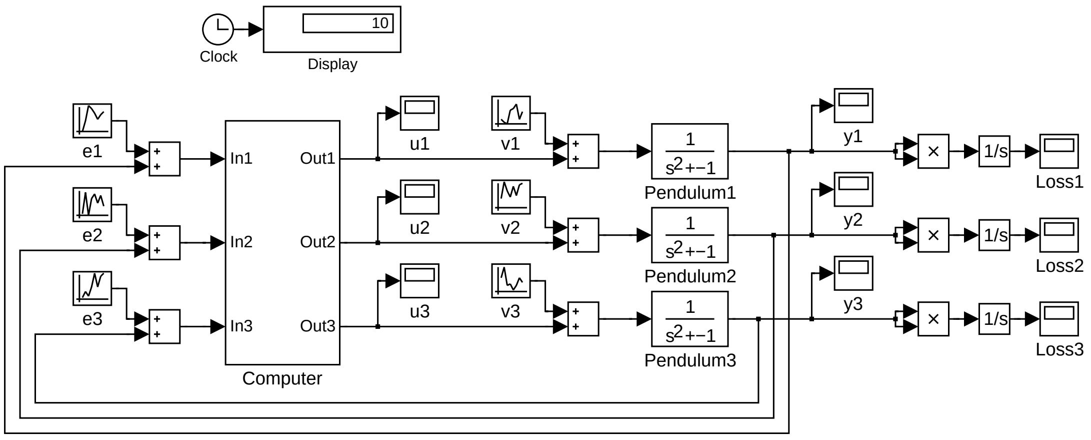

Figure 6 Example 2. Sub-task scheduling of control tasks.

where

$$\Phi_0 = (I - KC)(\Phi - \Gamma L) \qquad \Gamma_0 = (I - KC)(\Phi - \Gamma L)K$$
$$C_0 = -L \qquad D_0 = -LK$$

Code Segments Two code segments capture the timely behavior of the controller. In the first segment, `calculateOutput`, the measurement signal is read, the control signal is computed, and the control signal is sent to the process. The execution time of the segment is 10 ms:

```
function [exectime,states] = calculateOutput(flag,states,params)
switch flag,
  case 1, % enterCode
    states.y = analogIn(params.inChan);
    states.u = states.C0w + params.D0*states.y;
    exectime = 0.010;
  case 2, % exitCode
    analogOut(params.outChan,states.u);
    setPriority(params.P_US);
end
```

Under arbitrary fixed-priority (FP) scheduling, the `setPriority` kernel call is used to set the priority of the next segment (`updateState`). Note that the priority number is used *only if* FP scheduling has been specified. If rate-monotonic (RM) scheduling is used, it is the period alone that decides the priority of the task.

In the second segment, `updateState`, the states of the controller are updated. The execution time of this segment is 18 ms:

```
function [exectime,states] = updateState(flag,states,params)
switch flag,
  case 1, % enterCode
    states.w = params.Phi0 * states.w + params.Gamma0 * states.y;
    states.C0w = params.C0 * states.w;
    exectime = 0.018;
  case 2, % exitCode
```

```
    setPriority(params.P_CO);
end
```

Again, under FP scheduling, `setPriority` is used to set the priority of the next segment (`calculateOutput`).

Initialization Function In the initialization function `improved_init.m`, three pendulum controllers are created. If rate-monotonic scheduling (`prioRM`) is specified, both code segments will execute at the same priority (according to the task period). If fixed-priority scheduling (`prioFP`) is specified, all Calculate Output segments get higher priorities (`P_CO`) than the Update State parts (`P_US`).

```
function rtsys = improved_init

%% General settings
rtsys.tickSize = 0.001;
rtsys.prioFun = 'prioRM'; % Normal, rate-monotonic scheduling
% rtsys.prioFun = 'prioFP'; % Improved, dual-priority scheduling
rtsys.nbrOfInputs = 3;
rtsys.nbrOfOutputs = 3;

%% Create control tasks
omega = [3 5 7];            % Closed-loop bandwidths
T = [0.167 0.100 0.071];    % Sampling periods
P_CO = [3 2 1];             % Priorities for Calculate Output
P_US = [6 5 4];             % Priorities for Update State
rtsys.tasks = {};
for i = 1:3
  % Design controller
  [Phi,Gamma,C,L,K] = penddesign(omega(i),T(i));
  % Initialize the controller
  states.w = [0 0]';
  states.COw = 0;
  params.Phi0 = (eye(2)-K*C)*(Phi-Gamma*L);
  params.Gamma0 = params.Phi0*K;
  params.C0 = -L;
  params.D0 = -L*K;
  params.P_CO = P_CO(i);
  params.P_US = P_US(i);
  params.inChan = i;
  params.outChan = i;
  rcode = code({'calculateOutput','updateState'}, states, params);
  rtask = task(['Regul' num2str(i)], rcode, T(i), T(i), 0, P_CO(i));
  rtsys.tasks = {rtsys.tasks{:} rtask};
end
```

Experiements The following experiments verify that smaller loss can be obtained using sub-task scheduling.

1. Set the scheduling type to `prioRM`. This causes both segments of each controller, Calculate Output and Update State, to execute at the same priority.

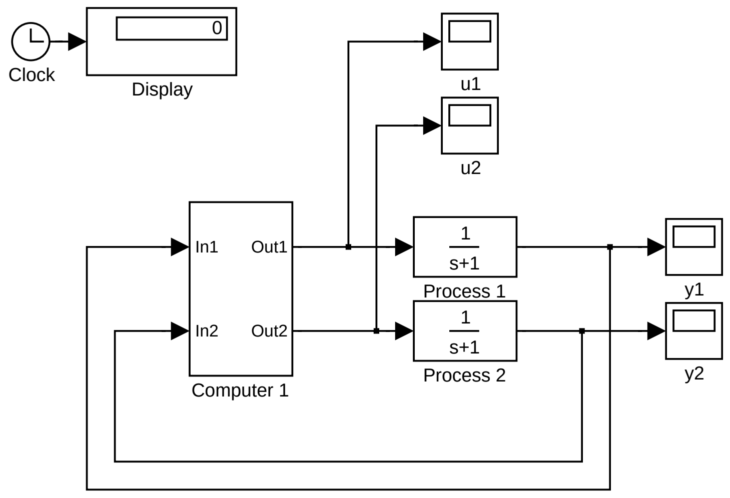

Figure 7 Example 3. Resources.

Simulate the system for 10 seconds and record the control performance loss J for the different controllers.

2. Change the scheduling type to `prioFP`. This causes the different segments to execute at different priorities. Run another simulation and record J. Verify that the loss is significantly smaller for Controller 1 and 2. (Controller 3 has the same loss, since it its Calculate Output part always has the highest priority.)

6.3 Resources

This example illustrates how monitors can be implemented in the simulator. Two first-order systems should be controlled by two control tasks, `rTask` and `oTask`. The model is shown in Figure 7. The tasks share a common resource, the `data` variable, which is protected by a mutex, `Mutex1`.

Code Segments The code of the first task, `rTask`, consists of two code segments, `rSeg1` and `rSeg2`. In the first segment, the task attempts to lock the mutex and then read the value of the `data` variable. If the value is less than 2, the task waits for the monitor event `Event1` before attempting to read the value again. The second segment implements a simple P controller.

```
function [exectime,states] = rseg1(flag,states,params)
switch flag,
  case 1, % enterCode
    if lock('Mutex1') == 0
      exectime = 0;
      return
    end
    if readData('Mutex1') < 2
      await('Event1')
      exectime = 0;
      return
    end
    exectime = 0.003;
  case 2,  % exitCode
    unlock('Mutex1')
end

function [exectime,states] = rseg2(flag,states,params)
switch flag,
  case 1, % enterCode
    y = analogIn(params.inChannel);
    states.u = -50*y;
    exectime = 0.003;
  case 2, % exitCode
    analogOut(params.outChannel,states.u)
end
```

The second task, `oTask`, also consists of two segments. The first segment, `oseg1`, implements a simple code segment. In the second segment, `oseg2`, the task attempts to lock the mutex and then increase the value of the `data` variable. Before leaving the monitor, the task causes the event `Event1`, signaling to the other task that the data value has changed.

```
function [exectime,states] = oseg1(flag,states,params)
switch flag,
  case 1, % enterCode
    y = analogIn(params.inChannel);
    states.u = -20*y;
    exectime = 0.002;
  case 2, % exitCode
    analogOut(params.outChannel,states.u)
end

function [exectime,states] = oseg2(flag,states,params)
switch flag,
  case 1, % enterCode
    if lock('Mutex1') == 0
      exectime = 0;
      return
    end
    data = readData(1);
    writeData(1,data+1)
    exectime = 0.003;
  case 2, % exitCode
    cause('Event1')
    unlock('Mutex1')
end
```

Initialization Function The initialization function is listed below:

```
function rtsys = resources_init

%% General settings
rtsys.tickSize = 0.001;
rtsys.prioFun = 'prioRM';
rtsys.nbrOfInputs = 2;
rtsys.nbrOfOutputs = 2;

%% Create tasks
rsegs = {'rseg1' 'rseg2'};
rParams.inChannel = 1;
rParams.outChannel = 1;
rcode = code(rsegs,[],rParams);
rtask = task('Regul', rcode, 0.012, 0.012);
osegs = {'oseg1' 'oseg2'};
oParams.inChannel = 2;
oParams.outChannel = 2;
ocode = code(osegs,[],oParams);
otask = task('OpCom', ocode, 0.017, 0.017);
rtsys.tasks = {rtask otask};

%% Create mutexes
data = 0;
m1 = mutex('Mutex1',data);
rtsys.mutexes = {m1};

%% Create events
e1 = event('Event1','Mutex1');
rtsys.events = {e1};
```

7. The Kernel Data Structures

The main data structure of the kernel is called `rtsys`. When the simulation starts, the `rtsys` structure is initialized by a call to the user-supplied initialization function. Further initialization is performed by the kernel S-function. Between simulation steps, the `rtsys` structure is stored in the `UserData` field of the Real-Time Kernel SIMULINK block.

7.1 The rtsys structure

The following fields of the `rtsys` structure must be supplied by the user in the initialization function:

`tickSize`	The tick-size of the kernel, in seconds.
`prioFun`	The name of the priority function used for scheduling. Predefined priority functions are `prioRM` for rate-monotonic, `prioDM` for deadline-monotonic, `prioFP` for arbitrary fixed-priority, and `prioEDF` for earliest-deadline-first scheduling.
`nbrOfInputs`	The number of input channels.
`nbrOfOutputs`	The number of output channels.
`tasks`	A cell array containing `task` structures.

The following fields of the `rtsys` structure are optionally created by the user in the initialization function:

`mutexes`	A cell array containing `mutex` structures.
`events`	A cell array containing `event` structures.
`outputs`	A vector containing initial outputs. If this field is not supplied by the user, the initial outputs are assumed to be zero.

The following fields of the `rtsys` structure are supplied and maintained by the kernel itself:

`nbrOfTasks`	The number of tasks in the kernel.
`nbrOfMutexes`	The number of mutexes.
`nbrOfEvents`	The number of events.
`timeQ`	A vector containing the indexes of the tasks that are waiting for the next release.
`readyQ`	A vector containing the indexes of the tasks that are ready to execute, including the running one.
`running`	The index of the running task, 0 if none.
`inputs`	A vector containing the inputs from the environment.

7.2 The task Structure

A `task` structure has the following fields:

`name`	The name of the task.
`code`	The `code` structure
`period`	The period, in seconds.
`deadline`	The relative deadline, in seconds.
`release`	The time of the current release of the task. It is updated by the kernel after each completion of the task.
`priority`	The priority. This field must be present only if arbitrary fixed-priority (`prioFP`) scheduling is specified.

7.3 The code Ctructure

A `code` structure has the following fields:

`segs`	A cell array containing the names of the files in which the code segments are located.
`nbrOfSegs`	The number of code segments.
`currentSeg`	The index of the current segment.
`nextSeg`	The index of the next segment to be executed.
`execTime`	Execution time left before the current segment has completed.
`states`	A user-defined data structure containing the controller states.
`params`	A user-defined data structure containing the controller parameters.

7.4 The mutex Structure

A `mutex` structure has the following fields:

`name`	The name of the mutex.
`heldBy`	The index of the task holding the mutex, 0 if none.
`waiting`	An array containing the indexes of the tasks waiting to lock the mutex.

7.5 The event Structure

An `event` structure has the following fields:

`name`	The name of the event.
`mutex`	The mutex associated with the event, if any.
`waiting`	An array containing the indexes of the tasks awaiting the event.

8. Initialization Functions

The following functions are used to create the initial data structures in the initialization script.

8.1 code

Purpose Create a code structure.

Syntax `c = code(segments,states,params)`

Description `code` is used to create a code structure in the initialization script. `segments` is a cell array containing the names of the code segments that are to be executed. `states` and `params` are any user-defined data structures.

Example `regulCode = code({'calculate','update'},states,params);`

8.2 task

Purpose Create a task structure.

Syntax

```
o = task(name,code,period,deadline)
o = task(name,code,period,deadline,offset)
o = task(name,code,period,deadline,offset,priority)
```

Description `task` is used to create a task structure in the initialization script. `code` is a code structure that is to be associated with the task. `period` and `deadline` give the period and the relative deadline of the task, in seconds. `offset` specifies a release offset for the task, in seconds. The default offset is zero. If arbitrary fixed-priority scheduling (`prioFP`) is used, `priority` should contain the priority of the task. Note that a low number denotes a high priority.

Example `regulTask = task('Regul 1',regulCode,0.020,0.020);`

8.3 mutex

Purpose	Create a mutex structure.

Syntax	`o = mutex(name)`
	`o = mutex(name,data)`

Description	`mutex` is used to create a mutex structure in the initialization script. The mutex is initially unlocked. `data` is used to supply a user-defined data structure that is to be protected mutex.
Example	`parMutex = mutex('parMutex',regPars);`

8.4 event

Purpose	Create an event structure.
Syntax	`o = event(name,mutex)`
Description	`event` is used to create an event structure in the initialization script. The event is associated with the mutex structure `mutex`.

Example	`parMutex = mutex('parMutex',regPars);`
	`parChange = event('parChange',parMutex);`

Limitations	There is currently no support for "free" events, i.e. events that are not associated with any mutex.

9. Real-Time Primitives

The following functions may be called from the used-defined code segments.

9.1 analogIn

Purpose	Read an input signal.
Syntax	`y = analogIn(inputNbr)`
Description	This function is typically called from the `enterCode`. `analogIn` reads an input signal from the environment. `inputNbr` must be a number between 1 and the `rtsys.nbrOfInputs`.
Example	`y = analogIn(params.yChan);`

9.2 analogOut

Purpose Write an output signal.

Syntax `analogOut(outputNbr, u)`

Description This function is typically called from the `exitCode`.

`analogOut` writes a new output signal to the environment. `outputNbr` must be a number between 1 and `rtsys.nbrOfOutputs`.

Example `analogOut(params.outChan, states.u)`

9.3 lock

Purpose Attempt to lock a mutex.

Syntax `status = lock(mutexName)`

Description This function must be called from the `enterCode`.

If the lock succeeded, `lock` returns 1, and the task may continue to execute the code segment.

If the lock failed, `lock` returns 0. The task is removed from the ready queue and is inserted into the waiting queue of the mutex. The code segment function must immediately return with an execution time of 0 since the task is no longer running. When the mutex is later unlocked, the task will be inserted into the ready queue again. When the task eventually becomes running, the same code segment will be executed again, causing a new attempt to lock the mutex.

Example

```
function [exectime,states] = mySeg(flag,states,param)
switch flag,
  case 1, % enterCode
    if lock('Mutex1') == 0
      exectime = 0;
      return
    end
    % Lock succeeded, we may continue...
```

Limitations Several tasks waiting for a mutex are handled in a FIFO manner. No further resource access protocols have been implemented.

Execution times different from zero are not allowed when the lock fails.

9.4 unlock

Purpose Unlock a mutex.

Syntax `unlock(mutexName)`

Description This function must be called from the `exitCode`.

The mutex must be held by the task itself. `unlock` causes all tasks waiting to lock the mutex to be moved to the ready queue. The first task that becomes running will be able to lock the mutex again.

Example `unlock('Mutex1');`

9.5 await

Purpose Await an event.

Syntax `await(eventName)`

Description This function must be called from the `enterCode`.

`await` is used to await an event inside a `lock`-`unlock` construct. The task is removed from the running queue and is inserted into the waiting queue of the event. The code segment function must immediately return with an execution time of 0 since the task is no longer running. When the event is later caused, the task will be inserted into the waiting queue of the associated mutex. When the mutex is unlocked, the task will be inserted into the ready queue again. When the task eventually becomes running, the same code segment will be executed again, causing a new attempt to lock the mutex.

Example

```
function [exectime,states] = mySeg(flag,states,param)
switch flag,
  case 1, % enterCode
    if lock('Mutex1') == 0
      exectime = 0;
      return
    end
    if readData('Mutex1') < 2
      await('Event1');
      exectime = 0;
      return
    end
    % Condition fulfilled, we may continue...
```

Limitations Execution times different from zero are not allowed after an `await`.

9.6 cause

Purpose	Cause an event.
Syntax	`cause(eventName)`
Description	This function must be called from the `exitCode`. `cause` is used to cause an event inside a `lock-unlock` construct. The mutex associated with the event must be held by the task itself. It causes all tasks waiting for the event to be moved to the waiting queue of the associated mutex.
Example	`cause('Event1');`

9.7 readData

Purpose	Read the data associated with a mutex.
Syntax	`data = readData(mutexName)`
Description	This function is typically called from the `enterCode`. `readData` reads the data associated with a mutex inside a `lock-unlock` construct. The mutex must be held by the task itself. If no data has been associated with the mutex, the function returns an empty vector.
Example	`data = readData('Mutex1');`

9.8 writeData

Purpose	Associate data with a mutex.
Syntax	`writeData(mutexName,data)`
Description	This function is typically called from the `exitCode`. `writeData` associates user-defined data with a mutex inside a `lock-unlock` construct. This provides a mean to communicate data to other tasks. Any previous data is overwritten.
Example	`writeData('Mutex1',regPars);`

9.9 currentSegment

Purpose Get the number of current segment.

Syntax `cs = currentSegment`

Description `cs = currentSegment` gets the number of the current code segment. This function could be used to implement several code segments in a single M-file.

Example In the initialization script:

```
regulCode = code({'allInOne','allInOne'},states,params);
```

In the code segment:

```
function [exectime,states] = allInOne(flag,states,param)
switch currentSegment,
  case 1, % Calculate Output
    switch flag,
      case 1, % enterCode
      ...
  case 2, % Update State
    switch flag,
      case 1, % enterCode
      ...
```

9.10 setNextSegment

Purpose Set the next segment to be executed.

Syntax `setNextSegment(segNbr)`

Description `setNextSegment` sets the number of the next segment to be executed, overriding the normal order. This could be used to implement data-dependent execution of code such as conditional loops. `segNbr` must be a number between zero and `nbrOfSegs`. A segment number of zero means that no more segments should be executed, and that the task should be suspended until the next period.

Example `setNextSegment(4);`

9.11 currentTime

Purpose Get the current time.

Syntax `now = currentTime`

Description `currentTime` gets the current time in the simulation, in seconds.

9.12 delayUntil

Purpose	Delay the task until a point in time.
Syntax	`delayUntil(time)`
Description	This function must be called from the `exitCode`.

`delayUntil` delays the task until some point in time. It should only be used to delay a task between code segments, since when the last code segment has finished, the task is automatically delayed until the next period by the kernel.

9.13 delay

Purpose	Delay the task for some time.
Syntax	`delay(duration)`
Description	This function must be called from the `exitCode`.

`delay` delays the task for some time. It is equivalent to `delayUntil(currentTime+duration)`.

9.14 setPeriod

Purpose	Set the period of the task.
Syntax	`setPeriod(period)`
Description	This function must be called from the `exitCode`.

`setPeriod` changes the period attribute of the task. If rate-monotonic scheduling (`prioRM`) is used, the task will have a different priority at the next clock-tick. The next release of the task will be calculated using the new priority.

9.15 setDeadline

Purpose	Set the relative deadline of the task.
Syntax	`setDeadline(deadline)`
Description	This function must be called from the `exitCode`.

`setDeadline` changes the relative deadline attribute of the task. If deadline-monotonic scheduling (`prioDM`) or earliest-deadline-first scheduling (`prioEDF`) is used, the task will have a different priority at the next clock-tick.

9.16 setPriority

Purpose	Set the priority of the task.
Syntax	`setPriority(priority)`
Description	This function must be called from the `exitCode`. `setPriority` changes the priority attribute of the task. This attribute is only used if arbitrary fixed-priority scheduling (`prioFP`) is used. Otherwise, this function has no effect.

9.17 sendMsg

Purpose	Send a message to a task.

Syntax	`sendMsg(taskNbr,data)`
	`sendMsg(taskName,data)`

Description	`sendMsg` sends user-defined data to a task. The message is put in the message queue of the receiving task.

Example

```
data.h = 0.020;
sendMsg(2,data)
```

9.18 receiveMsg

Purpose	Attempt to receive a message.
Syntax	`receiveMsg`
Description	This function must be called from the `enterCode`. `receiveMsg` gets the first message from the task's message queue. If there are no messages, the function returns an empty vector.

Example

```
data = receiveMsg;
if ~isempty(data)
   ...
```

10. References

Åström, K. J. and B. Wittenmark (1997): *Computer-Controlled Systems*, third edition. Prentice Hall.

Cervin, A. (1999): "Improved scheduling of control tasks." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 4–10. York, England.

Eker, J. and A. Cervin (1999): "A Matlab toolbox for real-time and control systems co-design." In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pp. 320–327. Hong Kong, P.R. China.