LUND UNIVERSITY

# A Matlab Toolbox for Real-Time and Control Systems Co-Design

Cervin, Anton; Eker, Johan

1999

Link to publication

*Citation for published version (APA):*
Cervin, A., & Eker, J. (1999). *A Matlab Toolbox for Real-Time and Control Systems Co-Design*. Paper presented at 6th International Conference on Real-Time Computing Systems and Applications, China.

*Total number of authors:*
2

**General rights**
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

**Take down policy**
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

6th International Conference on Real-Time Computing Systems and Applications, Hong Kong, P.R. China, December 1999

# A Matlab Toolbox for Real-Time and Control Systems Co-Design

**Johan Eker and Anton Cervin**

Department of Automatic Control
Lund Institute of Technology
Box 118, SE-221 00 Lund, Sweden
{johane,anton}@control.lth.se

## Abstract

The paper presents a Matlab toolbox for simulation of real-time control systems. The basic idea is to simulate a real-time kernel in parallel with continuous plant dynamics. The toolbox allows the user to explore the timely behavior of control algorithms, and to study the interaction between the control tasks and the scheduler. From a research perspective, it also becomes possible to experiment with more flexible approaches to real-time control systems, such as feedback scheduling. The importance of a more unified approach for the design of real-time control systems is discussed. The implementation is described in some detail and a number of examples are given.

## 1. Introduction

Real-time control systems are traditionally designed jointly by two different types of engineers. The control engineer develops a model for the plant to be controlled, designs a control law and tests it in simulation. The real-time systems engineer is given a control algorithm to implement, and configures the real-time system by assigning priorities, deadlines, etc.

The real-time systems engineer usually regards control systems as hard real-time systems, i.e. deadlines should never be missed. The control engineer on the other hand expects the computing platform to be predictive and support equidistant sampling. In reality none of the assumptions are necessarily true. This is even more obvious in the case where several control loops are running on the same hardware unit. The controllers will interact with each other since they are sharing resources such as CPU, network, analog/digital converters, etc. see Figure 1.

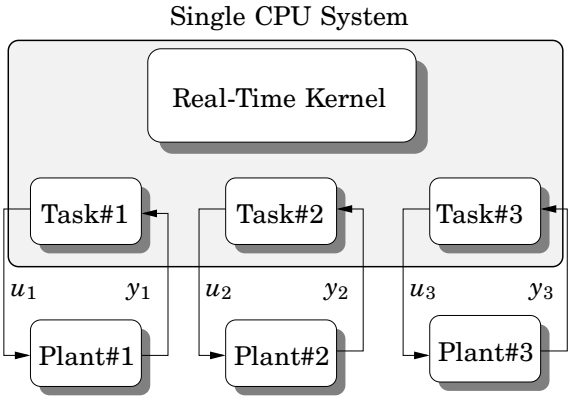



**Figure 1** Several control loops execute concurrently on one CPU. The interaction between the control tasks will affect the control performance.

A new interdisciplinary approach is currently emerging where control and real-time issues are discussed at all design levels. One of the first papers that dealt with co-design of control and real-time systems was [Seto *et al.*, 1996], where the sampling rates for a set of controllers sharing the same CPU are calculated using standard control performance metrics. Control and scheduling co-design is also found in [Ryu *et al.*, 1997], where the control performance is specified in terms of steady state error, overshoot, rise time, and settling time. These performance parameters are expressed as functions of the sampling period and the input-output latency. A heuristic iterative algorithm is proposed for the optimization of these parameters subject to schedulability constraints.

Good interaction between control theory and real-time systems theory opens up for a unified approach and more integrated algorithms. Scheduling parameters could for example be adjusted automatically online by a kernel supervisor. Such a setup would allow much more flexible real-time control systems than those available today. Ideas on adaption of scheduling parameters are for example found in [Abdelzaher

*et al.*, 1997] and [Stankovic *et al.*, 1999].

The development of algorithms for co-design of control and real-time systems requires new theory and new tools. This paper presents a novel simulation environment for co-design of control systems and real-time systems within the Matlab/Simulink environment. The advantages of using Matlab for this purpose are many. Matlab/Simulink is commonly used by control engineers to model physical plants, to design control systems, and to evaluate their performance by simulations. A missing piece in the simulations, however, has been the actual execution of the controllers when implemented as tasks in a real-time system. On the other hand, most of the existing tools for task simulations, for instance STRESS [Audsley *et al.*, 1994], DRTSS [Storch and Liu, 1996], and the simulator in [Ancilotti *et al.*, 1998], give no support for the simulation of continuous dynamics. Not much work has previously been done on mixed simulations of both process dynamics, control tasks, and the underlying real-time kernel. An exception is [Liu, 1998], where a single control task and a continuous plant was simulated within the Ptolemy II framework.

The simulator proposed in this paper is designed for simultaneous simulation of continuous plant dynamics, real-time tasks, and network traffic, in order to study the effects of the task interaction on the control performance.

## 2. The Basic Idea

The interaction between control tasks executing on the same CPU is usually neglected by the control engineer. It is however the case that having a set of control tasks competing for the computing resources will lead to various amounts of delay and jitter for different tasks. Figure 2 shows an example where three control tasks with the same execution times but different periods are scheduled using rate-monotonic priorities. In this case the schedule does not tell the whole story. In the example, the actual control delay (the delay from reading the input signal until writing a new output signal) for the low priority task varies from one to three times the execution time. Intuitively, this delay will affect the control performance, but how much, and how can we investigate this?

To study how the execution of tasks affects the control performance we must simulate the whole system, i.e. both the continuous dynamics of the controlled plant and the execution of the controllers in the CPU. We need not simulate the execution of the controller code on instruction level. In fact, it is enough to model the timely aspects of the code that are of relevance to other tasks and to the controlled

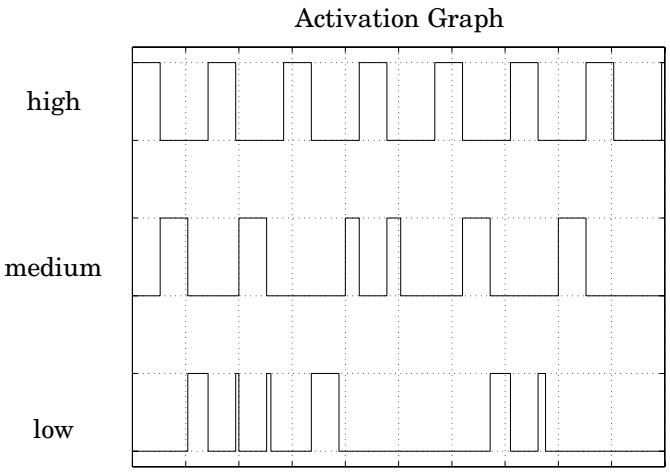



**Figure 2** The activation graph for three control tasks, with fixed priorities (high, medium, low), running in a pre-emptive kernel. The execution times are the same for all three processes.

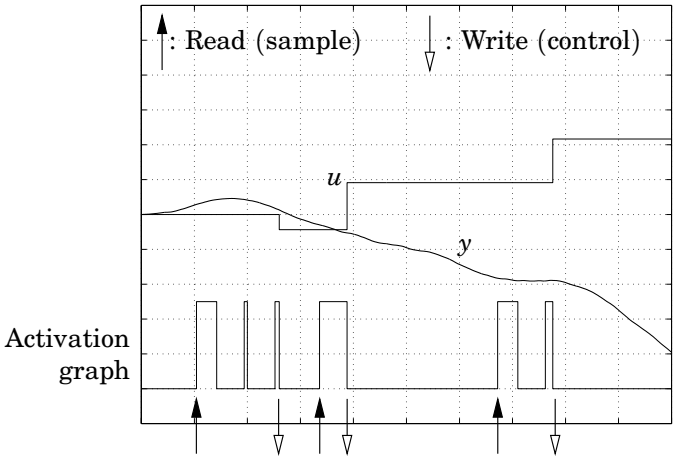



**Figure 3** This is how the low priority task from Figure 2 interacts with its plant. ($u$ is the control signal, $y$ is the measurement signal.)

plant. This includes computational phases, input and output actions, and blocking of common resources (other than the CPU).

Figure 3 shows the activation graph for the low priority task from Figure 2 and how it interacts with the continuous plant. The controller samples the continuous measurement signal from the plant ($y$) and writes new control outputs ($u$).

Figure 4 provides a schematic view of how we simulate the system. A model of a real-time kernel handles the scheduling of the control tasks and is

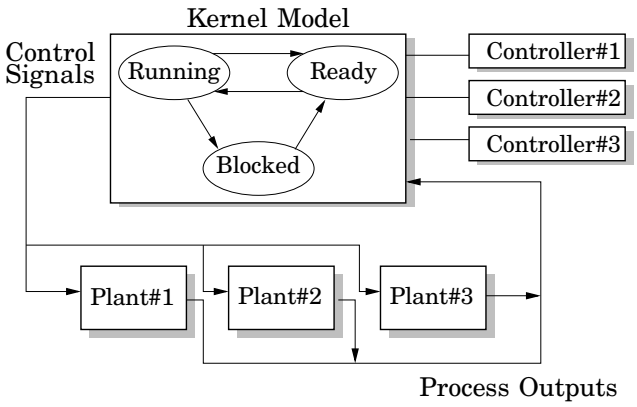



**Figure 4** Schematic view of the simulation setup. The controllers are tasks executing in a simulated pre-emptive kernel. The controllers and the control signals are discrete while the plant dynamics and the plant output are continuous. The continuous signals from the plants are sampled by the control tasks.

also responsible for properly interfacing the tasks with the physical environment. The outputs from the kernel model, i.e. the control signals, are piecewise constant. The plant dynamics and the plant outputs, i.e. the measurement signals, are continuous.

## 3. The Simulation Model

The heart of the toolbox is a Simulink block (an S-function) that simulates a tick-driven preemptive real-time kernel. The kernel maintains a number of data structures that are commonly found in a real-time kernel: a set of task records, a ready queue, a time queue, etc. At each clock tick, the kernel is responsible for letting the highest-priority ready task, i.e. the running task, execute in a virtual CPU. The scheduling policy used is determined by a priority function, which is a function of the attributes of a task. For instance, a priority function that returns the period of a task implements rate-monotonic scheduling, while a function that returns the absolute deadline of a task implements earliest-deadline-first scheduling. There currently exist predefined priority functions for rate-monotonic (RM), deadline-monotonic (DM), arbitrary fixed-priority (FP), and earliest-deadline-first (EDF) scheduling. The user may also write his own priority function that implements an arbitrary scheduling policy.

The execution model used is similar to the *live task model* described in [Storch and Liu, 1996]. During a simulation, the kernel executes user-defined code, i.e. Matlab functions, that have been associated with the different tasks. A code function returns an execution time estimate, and the task is not allowed to resume execution until the same amount of time has been consumed by the task in the virtual CPU.

### 3.1 The Task

Each task in the kernel has a set of basic attributes: A name, a list of code segments to execute, a period, a release time, a relative deadline, and the remaining execution time to be consumed in the virtual CPU. Some of the attributes, such as the release time and the remaining execution time, are constantly updated by the kernel during a simulation. The other attributes, such as the period and the relative deadline, remain constant unless they are explicitly changed by kernel function calls from the user code.

### 3.2 The Code

The local memory of a task is represented by two local, user-defined data structures `states` and `parameters`. The states may be changed by the user code, while the parameters remain constant throughout the execution.

To capture the timely behavior of a task, the associated code is divided into one or several code segments, see Figure 5. The execution time of a segment is determined dynamically at its invocation. Normally, the segments are executed in order, but this may be changed by kernel function calls from the user code.

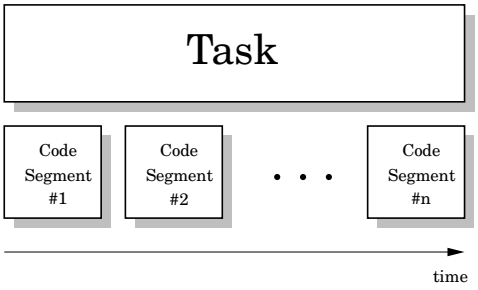



**Figure 5** The execution structure of a task. The flexible structure supports data dependent execution times and advanced scheduling techniques.

On a finer level, actual execution of statements in a code segment can only occur at two points: at the very beginning of the code segment (in the `enterCode` part) or at the very end of the code segment (in the `exitCode` part), see Figure 6. Typically, reading of input signals, locking of resources, and calculations are performed in the `enterCode` part. Writing of output signals, unlocking of resources, and other kernel function calls are typically performed in the `exitCode` part. The following examples illustrate how code segments can model real-time tasks.

EXAMPLE 1
A task implementing a control loop can often be divided into two parts: one that calculates a new control signal and one that updates the controller

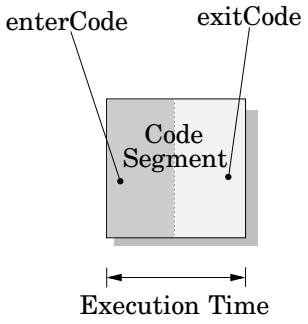
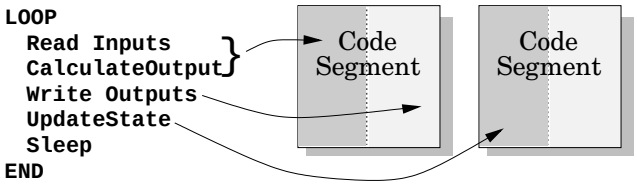



**Figure 6** A code segment is divided in two parts: the `enterCode` part and the `exitCode` part.

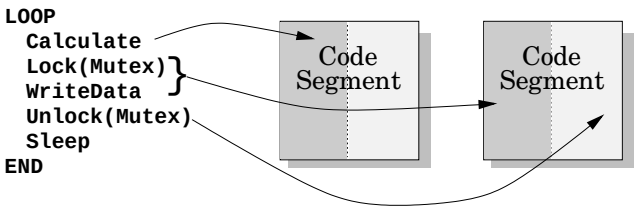
states. The first part, called Calculate Output, has a hard timing constraint and should finish as fast as possible. The timing requirement for the second part, Update State, is that it must finish before the next invocation of the task. Two code segments are appropriate to model the task:




The `enterCode` of the first segment contains the reading of the measurement signals, and the calculation of a new control signal. In the same segment, in `exitCode`, the control signal is sent to the actuator. The control delay of the controller is thus equal to the execution time of the first segment. The `enterCode` of the second code segment contains Update State. When the last segment has completed, the task is suspended until the next period by the kernel. □

EXAMPLE 2

The structure of a periodic task that first calculates some data and then writes to a common resource could look like this:




Again, two code segments can capture the timely behavior. The first code segment contains the `Calculate` statement, located in the `enterCode` part. The `enterCode` part of the second code segment contains the `Lock(Mutex)` and `WriteData` statements, while `exitCode` contains the `Unlock(Mutex)` statement. When the last segment has completed, the task is suspended until the next period by the kernel. □

## 4. Using the Simulator

From the user's perspective, the toolbox offers a Simulink block that models a computer with a real-time kernel. Connecting the Computer block's inputs and outputs (representing for instance A-D and D-A converters) to the plant, a complete computer-controlled system is formed, see Figure 7.

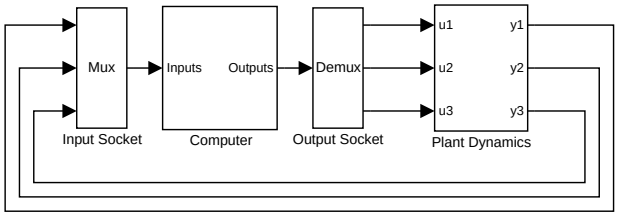



**Figure 7** The simulation environment offers a Simulink Computer block that can be connected to the model of the plant dynamics.

The plant dynamics may have to be controlled by several digital controllers, each implemented as a periodic control task in the computer. Besides the controllers, other tasks could be executing in the computer, for instance planning tasks, supervision tasks, and user communication tasks.

Opening up the Computer block, the user may study detailed information about the execution of the different tasks, see Figure 8. It is for instance

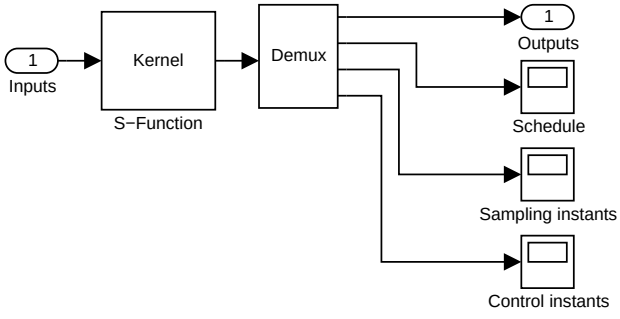



**Figure 8** Inside the Computer block, it is possible to study details about the execution of different tasks.

possible to study the schedule, i.e. a plot that shows when different tasks are executing, at run-time. Further statistics about the execution is stored in the workspace and may be analyzed when the simulation has stopped.

### 4.1 Controller Implementation

It is highly desirable that the design of the kernel is flexible and allows components to be reused and replaced. Much effort has been put into writing control algorithms in Matlab, and these algorithms should be straightforward to reuse. In the toolbox, a control algorithm can be implemented as a code segment with the following format:

```
function [exectime,states] = ...
      myController(flag,states,params)
 switch flag,
  case 1, % enterCode
    y = analogIn(params.inChan);
    states.u = <place control law here>
    exectime = 0.002;
   case 2, % exitCode
    analogOut(params.outChan,states.u)
 end
```

The input variables to `myController` are the state variables `states`, and the controller parameters `params`. The `flag` is used to indicate whether the `enterCode` or the `exitCode` part should be executed. If the `enterCode` part is executed, the function returns the execution time estimate `exectime` and the new state variables. The control signal is sent to the plant in the `exitCode` part.

***Remark*** The output signal $u$ is not normally regarded as a state variable in a controller. In this example, however, we need to store the value of $u$ between two invocations of the `myController` function.

### 4.2 Configuration

Before a simulation can start, the user must define what tasks that should exist in the system, what scheduling policy should be used, whether any common resources exist, etc. The initialization is performed in a Matlab script.

EXAMPLE 3
Three dummy tasks are initialized in the script below. The tick-size of the kernel is set to 0.001 s and the scheduling type is set to rate monotonic. The dummy code segment `empty` models a task that computes nothing for a certain amount a time. Each task is assigned a period, and a deadline which is equal to the period.

```
function rtsys = rtsys_init

% 1 = RM, 2 = DM, 3 = Arbitrary FP, 4 = EDF
rtsys.st = 1;
rtsys.tick_size = 0.001;

T = [0.10 0.08 0.06];  % Task Periods
D = [0.10 0.08 0.06];  % Deadlines
C = [0.02 0.02 0.02];  % Computation times

rtsys.Tasks = {}
code1 = code('empty',[],C(1))
code2 = code('empty',[],C(2))
code3 = code('empty',[],C(3))

rtsys.Tasks{1}=task('Task1',code1,T(1),D(1));
rtsys.Tasks{2}=task('Task2',code2,T(2),D(2));
rtsys.Tasks{3}=task('Task3',code3,T(3),D(3));
```

The initialization script is given as a parameter to a Computer block in a Simulink model. Simulating the model for one second produces, among other things, the schedule plot shown in Figure 9. □

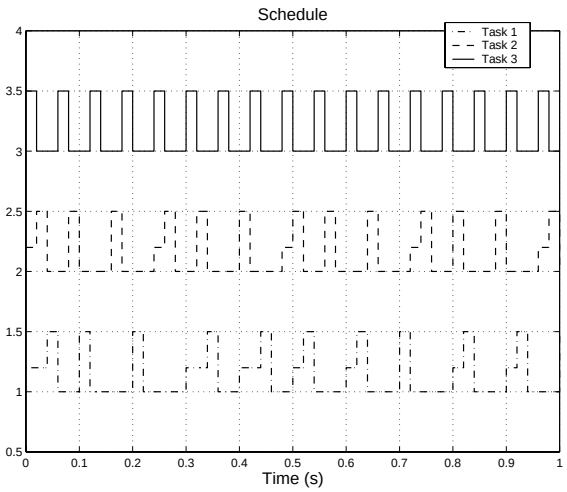



**Figure 9** The schedule resulting from the simulation in Example 3. The bottom graph shows when Task 1 is running (high), ready (medium) or blocked (low). The other two graphs represent Task 2 and Task 3.

### 4.3 Connecting a Continuous Plant

Figure 10 shows a Simulink diagram where a Computer block is connected to three pendulum models. The continuous plant models are described by other Simulink blocks.

A real-time system with three control loops are created in Example 4. One code segment named `myController` is associated with each task.

EXAMPLE 4

```
function rtsys = rtsys_init
% Scheduling type, 1=RM, 2=DM, 3=FP, 4=EDF
rtsys.st=1;
rtsys.tick_size=0.001;

% Desired bandwidths
omega=[3 5 7];
% Sampling periods
```

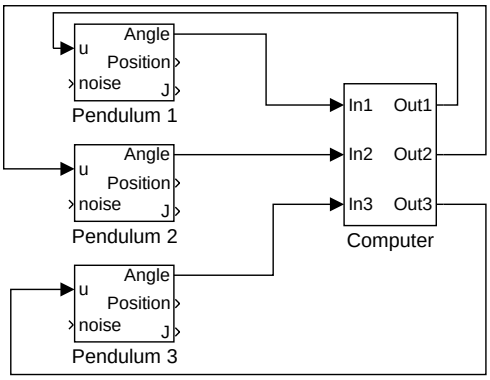



**Figure 10** A Simulink diagram where three continuous pendulum models are connected with the real-time kernel. The simulation result from this system is both the activation graph and the output from the continuous plants.

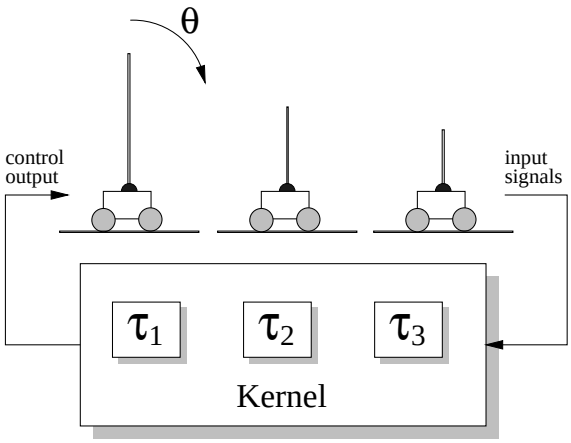



**Figure 11** The setup from described in Section 5. Three inverted pendulums with different lengths are controlled by three control tasks running on the same CPU.

```
T=[0.167 0.100 0.071];
for i=1:3
  % Design controller
  params=ctrl_design(omega(i),T(i));
  % Initialize control code
  states.xhat=[0 0]';
  % The controller reads from input i
  params.inChan=i;
  % The controller writes to output i
  params.outChan=i;
  sfbcode=code('myController',states,params);
  % Create task
  tasks{i}=task(['Task 'num2str(i)],...
                sfbcode, T(i), T(i));
end
rtsys.tasks=tasks;
```

□

The outputs from a simulation of this system are a set of continuous signals from the plants together with an activation graph. It is hence possible to evaluate the performance of the real-time systems both from a control design point of view and from a scheduling point of view.

## 5. A Co-Design Example

Using the simulator, it is possible to evaluate different scheduling policies and their effect on the control performance. Again consider the problem of controlling three inverted pendulums using only one CPU, see Figure 11. The inverted pendulum may be approximated by the following linear differential equation

$$\ddot{\theta} = \omega_0^2 \theta + \omega_0^2 u/g,$$

where $\omega_0 = \sqrt{g/l}$ is the natural frequency for a pendulum with length $l$. The goal is to minimize the angles, so for each pendulum we want to minimize the accumulated quadratic loss function

$$J_i(t) = \int_0^t \theta_i^2(s) ds. \tag{1}$$

Three discrete-time controllers with state feedback and observers are designed. Sampling periods for the controllers are chosen according to the desired bandwidths (3, 5 and 7 rad/s respectively) and the CPU resources available. The execution times of the control tasks, $\tau_i$, are all 28 ms, and the periods are $T_1 = 167$ ms, $T_2 = 100$ ms, and $T_3 = 71$ ms.

Task objects are created according to Example 4. Also, similar to Example 1, the control algorithm is divided into two code segments, Calculate Output and Update State, with execution times of 10 and 18 ms respectively.

In a first simulation, the control tasks are assigned constant priorities according to the rate-monotonic schema, and the two code segments execute at the same priority. In a second simulation, the Calculate Output code segments are assigned higher priorities than the Update State segments, according to iterative priority/deadline assignment algorithm suggested in [Cervin, 1999]. The accumulated loss function for the slow pendulum ($T_1 = 167$ ms) is easily recorded in the Simulink model, and the results from both simulations are shown in Figure 12.

A close-up inspection of the schedule produced in the second simulation is shown in Figure 13. It can be seen that the faster tasks sometimes allow the slower tasks to execute, and in this way the control delays in the slower controllers are minimized. The result is a smaller accumulated loss, and thus, better control performance.

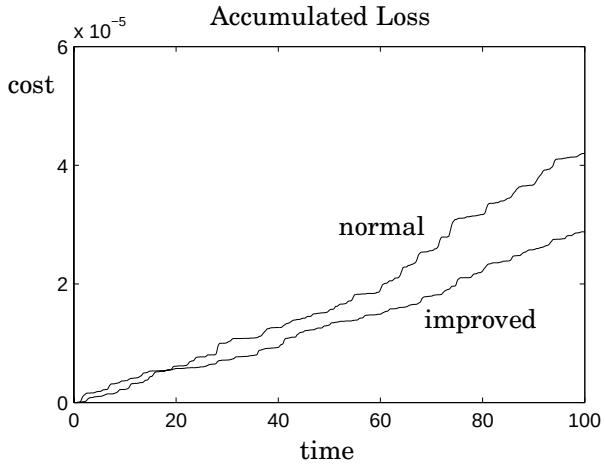



**Figure 12** The accumulated loss, see Equation (1), for the low priority controller using normal and improved scheduling. The cost is substantially reduced under the improved scheduling.

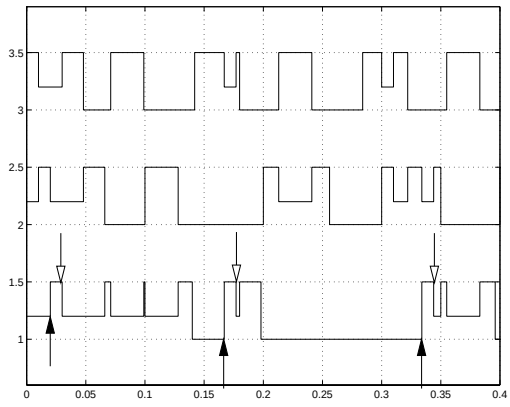



**Figure 13** The activation graph when the improved scheduling strategy is used. Note that the control delay for the low priority task is approximately the same as for the other tasks.

## 6. Simulation Features

Further features of the toolbox are the support for common real-time primitives like mutual exclusion locks, events (also known as condition variables), and network communication blocks.

### 6.1 Locks and Events

The control tasks do not only interact with each through the use of same CPU, but also due to sharing other user-defined resources. The kernel allows the user to define monitors and events, for implementing complex dependencies between the task. The syntax and semantics of the `mutex` and `event` primitives are demonstrated by a small example. Two tasks `Regul` and `OpCom` are sharing a variable called `data`. To ensure mutual exclusion the variable is protected by the mutex variable `M1`. Associated with `M1` is a monitor event called `E1`. The `Regul`-task consists of two code segments called `rseg1` and `rseg2`, that are shown in Example 5. Each time the `Regul`-task is released it tries to lock the monitor variable `M1`. Once the monitor is locked it may access the shared data. If the value of the `data`-variable is less than two, it waits for the event `E1` to occur.

EXAMPLE 5

```
function [exectime, states] = ...
            rseg1(flag,states,params)
 switch flag,
  case 1,   % enterCode
    if lock('M1')
      data = readData('M1');
      if data < 2
        await('E1');
        exectime = 0;
      else
        exectime = 0.003;
      end
    else
      exectime = 0;
    end
  case 2,   % exitCode
    unlock('M1');
 end

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

□

The locks and the events are designed similarly to how monitors and events are implemented in a standard real-time kernel, i.e. using queues associated with the monitor for storing tasks blocking on locks or events. The execution time used for trying, but failing to lock, is in the example above zero.

### 6.2 Network Blocks

It is possible to include more than one Computer block in a Simulink model, and this opens up the possibility to simulate much more complex systems than the ones previously discussed. Distributed control systems may be investigated. Furthermore, fault-tolerant systems, where, for redundancy, several computers are used for control, could also be simulated. In order to simulate different communication protocols in such systems, communication blocks for

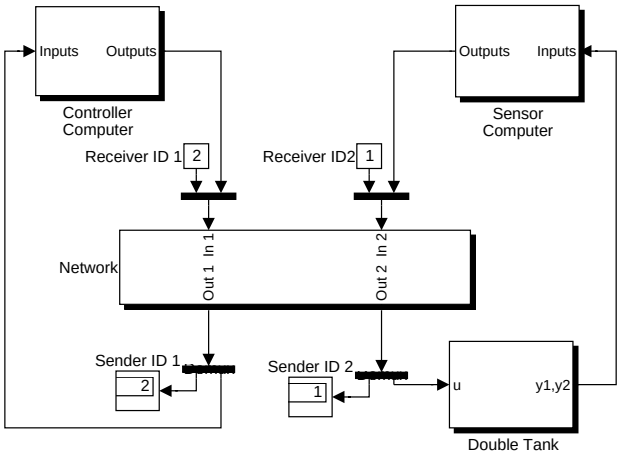



**Figure 14** A distributed control system where the sensor and the CPU are dislocated. The controller and the sensor are implemented as periodic tasks running on separate CPUs.

sending data between the different Computer blocks are needed. Figure 14 shows a simulation setup for a simple distributed system where the controller, and the actuator and sensor, are located at different places. Besides the kernel blocks there is a network block for communication. The network block is event driven, and each time any of the input signals change, the network is notified. The user needs to implement the network protocol, since the blocks simply provides the mechanisms for sending data between kernels.

### 6.3 High Level Task Communication

One of the main reasons for designing the kernel and the network blocks was to facilitate the simulation of flexible embedded control system, i.e. systems where the task set is allowed to change dynamically and the underlying real-time system must compensate for this. From a control theory perspective we might say that we want to design a feedback connection between the control tasks and the scheduler, see Figure 15. To support the simulation of feedback

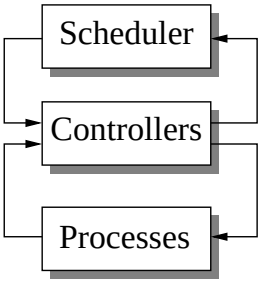



**Figure 15** The control tasks and the task scheduler are connected in a feedback loop.

scheduling, there must be ways for the tasks and the task scheduler to communicate. Therefore the kernel also supports system-level message passing between tasks.

## 7. Conclusions

This paper presented a novel simulator for the co-design of real-time systems and control systems. The main objective is to investigate the consequences on control performance of task interaction on kernel level. This way, scheduling algorithms may be evaluated from a control design perspective. We believe that this is an issue of increasing importance. There are many more things to be implemented and improved before this block set will become a truly useful tool. Currently the kernel is tick-based, and has little support for external interrupts. The next version of the kernel block will probably be event-based in order to better support interrupts and event-based sampling. To make the simulations more realistic, the scheduler itself could also be modeled as a task that consumes CPU time. This would also enhance the possibilities for the user to implement new scheduling strategies for control tasks.

## Acknowledgments

This work was sponsored by the Swedish national board of technical development (NUTEK) and by the Swedish network for real-time research and education (ARTES).

## References

Abdelzaher, T., E. Atkins, and K. Shin (1997): "QoS negotiation in real-time systems, and its application to flight control." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Ancilotti, P., G. Buttazzo, M. D. Natale, and M. Spuri (1998): "Design and programming tools for time critical applications." *Real-Time Systems*, **14:3**, pp. 251–269.

Audsley, N., A. Burns, M. Richardson, and A. Wellings (1994): "STRESS—A simulator for hard real-time systems." *Software—Practice and Experience*, **24:6**, pp. 543–564.

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

Liu, J. (1998): "Continuous time and mixed-signal simulation in Ptolemy II." Technical Report UCB/ERL Memorandum M98/74. Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Ryu, M., S. Hong, and M. Saksena (1997): "Streamlining real-time controller design: From performance specifications to end-to-end timing constraints." In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Seto, D., J. Lehoczky, L. Sha, and K. Shin (1996): "On task schedulability in real-time control systems." In *Proceedings of the IEEE Real-Time Systems Symposium*.

Stankovic, J. A., C. Lu, S. H. Son, and G. Tao (1999): "The case for feedback control real-time scheduling." In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pp. 11–20.

Storch, M. F. and J. W.-S. Liu (1996): "DRTSS: A simulation framework for complex real-time systems." In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 160–169.