IST-2001-37652

Hard Real-time CORBA

Title

TrueTime and Jitterbug

Authors

Anton Cervin(LTH)
Bo Lincoln (LTH)
Dan Henriksson(LTH)
Karl-Erik Årzén (LTH)

www.hardrealtimecorba.org

Summary Sheet

IST Project 2001-37652
HRTC
Hard Real-time CORBA

TrueTime and Jitterbug

Abstract:

The present document is an additional deliverable describing the analysis
and simulation tools TrueTime and Jitterbug that will be applied to
CORBA-based networked control loops within the HRTC project.

HRTC Partners:

Universidad Politécnica de Madrid
Lunds Tekniska Högskola
Technische Universität Wien
SCILabs Ingenieros.

Release Sheet (1)

Release:	**1.0 Final**
Date:	2003/02/28
Scope	Initial version
Sheets	All

Table of Contents

www.hardrealtimecorba.org

1 Introduction

Jitterbug is a new Matlab-based toolbox that makes it possible to compute a quadratic performance criterion for a linear control system under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers.

The new Matlab/Simulink-based tool TrueTime facilitates simulation of the temporal behavior of a networked control loops consisting of nodes with multitasking real-time kernel executing controller tasks, and communication networks. The tasks are controlling processes that are modeled as ordinary Simulink blocks. Different task scheduling policies may be used (e.g., priority-based preemptive scheduling, static cyclic scheduling, and earliest-deadline-first (EDF) scheduling) and different communication protocols can be used (e.g., Ethernet, CAN, and TDMA).

TrueTime makes it possible to study more general and detailed timing models of computer-controlled systems. The toolbox offers two Simulink blocks: a Real-Time Kernel block and a Real-Time Network block. The delays in the control loop are captured by simulation of the execution of tasks in the kernel and the transmission of messages over the network.

Within HRTC Jitterbug and Truetime will be used to analyze and simulate CORBA-based networked control loops. The scenarios that will be investigated are:
- CORBA over IIOP
- R-T CORBA over IIOP
- HRT CORBA over switched Ethernet
- HRT CORBA over TTP

This deliverable consists of three parts:
- An IEEE Control Systems Magazine article describing the two toolboxes.
- The reference manual for Jitterbug.
- The reference manual for Truetime.

The three parts are provided as separate appendices. The two toolboxes can be downloaded from http://www.control.lth.se/~lincoln/jitterbug/ and from http://www.control.lth.se/~dan/truetime/

2 Appendices

I. Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, Karl-Erik Årzén (2003): *Analysis and Simulation of Controller Timing,* IEEE Control Systems Magazine, June 2003. To appear.

II. Anton Cervin and Bo Lincoln (2003): Jitterbug Reference Manual, Dept of Automatic Control, Lund Institute of Technology, ISSN 0280-5316, ISRN LUTFD2/TFRT—7604—SE

III. Dan Henriksson and Anton Cervin (2002): TrueTime v1.04 – Reference Manual, Dept of Automatic Control, Lund Institute of Technology

Analysis and Simulation of Controller Timing

Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, Karl-Erik Årzén

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

Abstract

The article presents two MATLAB-based tools for analysis and simulation of real-time control systems: JITTERBUG and TRUETIME. JITTERBUG allows the user to compute a quadratic performance criterion for a linear control system under various timing conditions. The control system is described using a number of continuous- and discrete-time linear systems. A stochastic timing model with random delays is used to describe the execution of the system. The tool can also be used to investigate aperiodic controllers, multi-rate controllers, and jitter-compensating controllers. TRUETIME facilitates event-based co-simulation of a multitasking real-time kernel containing controller tasks and the continuous dynamics of controlled plants. The simulations capture the true, timely behavior of real-time controller tasks and communication networks, and dynamic control and scheduling strategies can be evaluated from a control performance perspective. The controllers can be implemented as Matlab functions, C functions, or ordinary discrete-time Simulink blocks. A number of examples that illustrate the use of the tools are given.

Introduction

Control systems are becoming increasingly complex from the perspectives of both control and computer science. Today, even seemingly simple embedded control systems often contain a multitasking real-time kernel and support networking. At the same time, the market demands that the cost of the system be kept at a minimum. For optimal use of computing resources, the control algorithm and the control software designs need to be considered at the same time. For this reason, new, computer-based tools for real-time and control co-design are needed.

Many computer-controlled systems are distributed systems consisting of computer nodes and a communication network connecting the various systems. It is not uncommon for the sensor, the actuator, and the control calculations to reside on different nodes, as in vehicle systems, for example. This gives rise to networked control loops (see [1]). Within the individual nodes, the controllers are often implemented as one or several tasks on a microprocessor with a real-time operating system. Often the microprocessor also contains tasks for other functions (e.g., communication and user interfaces). The operating system typically uses multiprogramming to multiplex the execution of the various tasks. The CPU time and the communication bandwidth can hence be viewed as shared resources for which the tasks compete.

Digital control theory normally assumes equidistant sampling intervals and a negligible or constant control delay from sampling to actuation. However, this can seldom be achieved in practice. Within a node, tasks interfere with each other through preemption and blocking when waiting for common resources. The execution times of the tasks themselves may be data-dependent or may vary due to hardware features such as caches. On the distributed level, the communication gives rise to delays that can be more or less deterministic depending on the communication protocol. Another source of temporal nondeterminism is the increasing use of commercial off-the-shelf (COTS) hardware and software components in

real-time control (e.g., general-purpose operating systems such as Windows and Linux and general-purpose network protocols such as Ethernet). These components are designed to optimize average-case rather than worst-case performance.

The temporal nondeterminism can be reduced by the proper choice of implementation techniques and platforms. For example, time-driven static scheduling improves the determinism, but at the same time it reduces the flexibility and limits the possibilities for dynamic modifications. Other techniques of similar nature are time-driven architectures such as TTA [2] and synchronous programming languages such as Esterel, Lustre, and Signal [3]. Even with these techniques, however, some level of temporal nondeterminism is unavoidable.

The delay and jitter introduced by the computer system can lead to significant performance degradation. To achieve good performance in systems with limited computer resources, the constraints of the implementation platform must be taken into account at design time. To facilitate this, software tools are needed to analyze and simulate how the timing affects the control performance. This article describes two such tools: JITTERBUG[1] and TRUETIME[2].

JITTERBUG is a MATLAB-based toolbox that makes it possible to compute a quadratic performance criterion for a linear control system under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The main contribution of the toolbox, which is built on well-known theory (LQG theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

The use of JITTERBUG assumes knowledge of sampling period and latency distributions. This information can be difficult to obtain without access to measurements from the true target system under implementation. Also, the analysis cannot capture all the details and nonlinearities (especially in the real-time scheduling) of the computer system. A natural approach is to use simulation instead. However, today's simulation tools make it difficult to simulate the true temporal behavior of control loops. What is normally done is to introduce time delays in the control loop representing average-case or worst-case delays. Taking a different approach, the MATLAB/Simulink-based tool TRUETIME facilitates simulation of the temporal behavior of a multitasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary Simulink blocks. TRUETIME also makes it possible to simulate simple models of communication networks and their influence on networked control loops. Different scheduling policies may be used (e.g., priority-based preemptive scheduling and earliest-deadline-first (EDF) scheduling). (For more on real-time scheduling, see [4].)

TRUETIME can also be used as an experimental platform for research on dynamic real-time control systems. For instance, it is possible to study compensation schemes that adjust the control algorithm based on measurements of actual timing variations (i.e., to treat the temporal uncertainty as a disturbance and manage it with feedforward or gain scheduling). It is also easy to experiment with more flexible approaches to real-time scheduling of controllers, such as feedback scheduling [5]. There the available CPU or network resources are dynamically distributed according the current situation (CPU load, the performance of the different loops, etc.) in the system.

Comparison of the Tools

JITTERBUG offers a collection of MATLAB routines that allow the user to build and analyze simple timing models of computer-controlled systems. A control system is built by connecting a number of continuous-time and discrete-time systems. For each subsystem, optional noise and cost specifications may be given. In the simplest case, the discrete-time systems are assumed to be updated in order during the control period. For each discrete system, a

---

[1]Available at http://www.control.lth.se/~dan/truetime
[2]Available at http://www.control.lth.se/~lincoln/jitterbug

random delay (described by a discrete probability density function) can be specified that must elapse before the next system is updated. The total cost of the system (summed over all subsystems) is computed algebraically if the timing model system is periodic or iteratively if the timing model is aperiodic.

To make the performance analysis feasible, JITTERBUG can only handle a certain class of system. The control system is built from linear systems driven by white noise, and the performance criterion to be evaluated is specified as a quadratic, stationary cost function. The timing delays in one period are assumed to be independent from the delays in the previous period. Also, the delay probability density functions are discretized using a time-grain that is common to the whole model.

Even though a quadratic cost function can hardly capture all aspects of a control loop, it can still be useful when one wants to quickly judge several possible controller implementations against each other. A higher value of the cost function typically indicates that the closed-loop system is less stable (i.e., more oscillatory), and an infinite cost means that the control loop is unstable. The cost function can easily be evaluated for a large set of design parameters and can be used as a basis in the control and real-time design.

The MATLAB/Simulink-based tool TRUETIME makes it possible to study more general and detailed timing models of computer-controlled systems. The toolbox offers two Simulink blocks: a Real-Time Kernel block and a Real-Time Network block. The delays in the control loop are captured by simulation of the execution of tasks in the kernel and the transmission of messages over the network.

Being a simulation tool, TRUETIME is not restricted to evaluation of a quadratic performance criterion, but can of course be used to evaluate any time-domain behavior of the control loop. A drawback is that if there are many random variables, very long simulations may be needed to draw conclusions about the system.

The Simulink blocks are event-driven, so there is no need to specify a time-grain for the model. The execution of a task can be simulated on an arbitrarily fine timescale by dividing the code into segments. Typically, it is enough to divide a control task into a few segments (for instance, Calculate and Update) to capture its timely behavior. The code segments can be likened to the discrete-time subsystems in JITTERBUG. A difference is that they can contain any user-written code (including calls to real-time primitives) and not just linear update equations.

Finally, although JITTERBUG can only analyze the stationary behavior of a control loop, TRUETIME can be used to investigate transient responses in conjunction with, for example, temporary CPU overloads. It can also be used to study systems where the controller and scheduling parameters are adapted to the current situation in the real-time control system.

Networked Control System

As a recurring example in this article (among other examples), we will study a control loop that is closed over a communications network. Closing control loops over networks is becoming increasingly popular in embedded applications because of its flexibility, but it also introduces many new problems. From a control perspective, the computer system will introduce (possibly random) delays in the control loop. There is also the potential problem of lost measurement signals or control signals. From a real-time perspective, the first problem is figuring out the temporal constraints (deadlines, etc.) of the different tasks in the system, and then scheduling the CPUs and the network such that all constraints are met during runtime.

In the example, we will study the setup shown in Fig. 1. In our control loop, the sensor, the actuator, and the controller are distributed among different nodes in a network. The sensor node is assumed to be time-driven, whereas the controller and actuator nodes are assumed to be event-driven. At a fixed period h, the sensor samples the process and sends the measurement sample over the network to the controller node. There the controller computes a control signal and sends it over the network to the actuator node, where it is subsequently actuated. This kind of setup was studied in [6], where an optimal, delay-compensating LQG controller was derived. Here we are more interested in the interplay between control and real-time design and choose to study a simple process and controller.
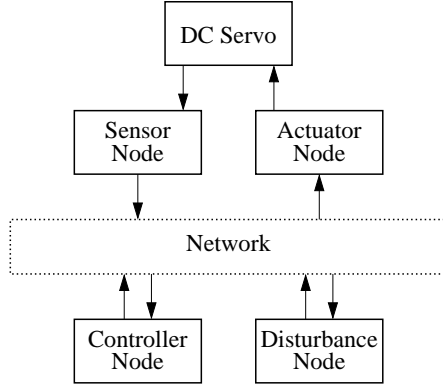
Figure 1 The networked control system is used as a recurring example in the article.

We will assume that the process to be controlled is a DC servo and that the controller is a simple PD controller. In the JITTERBUG section, we will study the impact of sampling period, delay, and jitter on the control loop performance. A simple jitter-compensating controller is introduced where the parameters of the PD controller are adjusted according to the actual measured delay from the sensor node to the controller node. The delay model at this point is very simple: the delay from one node to another is described by a uniformly distributed random variable. In the TRUETIME section, a more detailed delay model is obtained by simulating the execution of tasks in the nodes and scheduling of messages in the network. Long random delays are caused by interfering traffic generated by a disturbance node in the network. It will be seen that the behavior in the simulations agrees with the results obtained by the more simplistic analysis.

Analysis Using Jitterbug

In JITTERBUG, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example of a JITTERBUG model is shown in Fig. 2, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

Signal Model
A *continuous-time system* is described by

$$\dot{x}_c(t) = Ax_c(t) + Bu(t) + v_c(t)$$
$$y(t) = Cx_c(t),$$

where A, B, and C are constant matrices, and v_c is a continuous-time white noise process with covariance R_{1c}. (In the toolbox, it is also possible to specify discrete-time measurement noise. This will be interpreted as input noise at any connected discrete-time system.) The cost of the system is specified as

$$J_c = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x_c(t) \\ u(t) \end{pmatrix}^T Q_c \begin{pmatrix} x_c(t) \\ u(t) \end{pmatrix} dt,$$
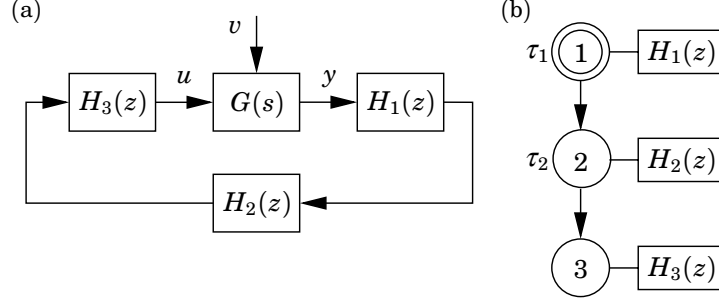
Figure 2 A simple JITTERBUG model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$ and the controller is described by the three discrete-time systems $H_1(z)$, $H_2(z)$, and $H_3(z)$, representing the sampler, the control algorithm, and the actuator. The discrete systems are executed according to the periodic timing model.

where Q_c is a positive semidefinite matrix.

A *discrete-time system* is described by

$$x_d(t_{k+1}) = \Phi x_d(t_k) + \Gamma u(t_k) + v_d(t_k)$$
$$y(t_k) = C x_d(t_k) + D u(t_k) + e_d(t_k),$$

where Φ, Γ, C, and D are possibly time-varying matrices (see below). The covariance of the discrete-time white noise processes v_d and e_d is given by

$$R_d = \mathbf{E} \begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix} \begin{pmatrix} v_d(t_k) \\ e_d(t_k) \end{pmatrix}^T.$$

The input signal u is sampled when the system is updated, and the state x_d and the output signal y are held between updates. The cost of the system is specified as

$$J_d = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x_d(t) \\ u(t) \end{pmatrix}^T Q_d \begin{pmatrix} x_d(t) \\ u(t) \end{pmatrix} dt,$$

where Q_d is a positive semidefinite matrix. Note that the update instants t_k need not be equidistant in time, and that the cost is defined in continuous time.

The *total system* is formed by appropriately connecting the inputs and outputs of a number of continuous-time and discrete-time systems. Throughout, MIMO formulations are allowed, and a system may collect its inputs from a number of other systems. The total cost to be evaluated is summed over all continuous- and discrete-time systems:

$$J = \sum J_c + \sum J_d.$$

Timing Model

The timing model consists of a number of timing nodes. Each node can be associated with zero or more discrete-time systems in the signal model, which should be updated when the node becomes active. At time zero, the first node is activated. The first node can also be declared to be *periodic* (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every h seconds. This is useful for modeling periodic controllers and also greatly simplifies the cost calculations.

Each node is associated with a time delay τ, which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a discrete-time probability density function

$$P_\tau = [\,P_\tau(0) \quad P_\tau(1) \quad P_\tau(2) \quad \dots\,],$$
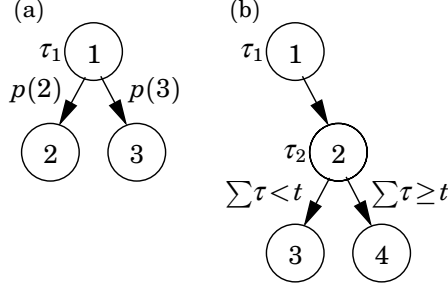
5

Figure 3 Alternative execution paths in a JITTERBUG execution model: (a) random choice of path and (b) choice of path depending on the total delay from the first node.

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time grain δ is a constant that is specified for the whole model.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period h. Any remaining timing nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model a controller that samples "as fast as possible" instead of waiting for the next period.

Node- and Time-Dependent Execution   The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations (i.e., different Φ, Γ, C and D matrices) in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. An example of this type is given later.

It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers for example.

Alternative Execution Paths   For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In JITTERBUG, two such cases can be modeled (see Fig. 3):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model a sample being lost with some probability.

(b) A vector n of next nodes can be specified with a timevector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

Computation of Cost and Spectral Densities

The computation of the total cost is performed in three steps: First, the cost functions, the continuous-time noise, and the continuous-time systems are sampled using the time-grain of the model. Second, the closed-loop system is formulated as a jump linear system, where Markov nodes are used to represent the time-steps in and between the execution nodes. Third, the stationary variance of all states in the system is calculated.

For periodic systems, the Markov state always returns to the periodic execution node every h/δ time steps. The stationary variance in the periodic execution node can then be obtained by solving a linear system of equations. The cost is then calculated over the time steps in one period. In this case, the cost calculation is fast and exact. It is also straightforward to compute the spectral densities of all outputs as observed in the periodic timing node. For systems without a periodic node, the variance must be computed iteratively. In both cases, the toolbox will return an infinite cost if the total system is not stable (in the mean-square sense). More details about the internal workings of JITTERBUG can be found in [7].

6

```
G = 1000/(s*(s+1));                        Define the process
H1 = 1;                                    Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);                  Define the controller
H3 = 1;                                    Define the actuator

Ptau1 = [ ... ];             Define delay probability distribution 1
Ptau2 = [ ... ];             Define delay probability distribution 2

N = initjitterbug(delta,h);            Set time-grain and period
N = addtimingnode(N,1,Ptau1,2);            Define timing node 1
N = addtimingnode(N,2,Ptau2,3);            Define timing node 2
N = addtimingnode(N,3);                    Define timing node 3

N = addcontsys(N,1,G,4,Q,R1,R2);   Add plant, specify cost and noise
N = adddiscsys(N,2,H1,1,1);            Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);            Add controller to node 2
N = adddiscsys(N,4,H3,3,3);            Add actuator to node 3

N = calcdynamics(N);               Calculate internal dynamics
J = calccost(N);                    Calculate the total cost
```

Figure 4 This MATLAB script shows the commands needed to compute the performance index of the networked control system using JITTERBUG.

The Networked Control System

The first example we will look at is the networked control system introduced earlier. We will begin by investigating how sensitive the control loop is to slow sampling and delays, and then we will look at delay and jitter compensation.

The JITTERBUG model of the system was shown in Fig. 2. The DC servo process is given by the continuous-time system

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

The process is driven by white continuous-time input noise. There is assumed to be no measurement noise.

The process is sampled periodically with the interval h. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1,$$

and the discrete-time PD controller is implemented as

$$H_2(z) = -K \left(1 + \frac{T_d}{h} \frac{z-1}{z} \right),$$

where the controller parameters are chosen as $K = 1.5$ and $T_d = 0.035$. (A real implementation would include a low-pass filter in the derivative part, but that is ignored here.)

The delays in the computer system are modeled by the two (possibly random) variables τ_1 and τ_2. The total delay from sampling to actuation is thus given by $\tau_{tot} = \tau_1 + \tau_2$. It is assumed that the total delay never exceeds the sampling period (otherwise JITTERBUG would skip the remaining updates).

Finally, we need to specify the control performance criterion to be evaluated. As a cost function, we choose the sum of the squared process input and the squared process output:

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \left(y^2(t) + u^2(t) \right) dt. \tag{1}$$

An outline of the MATLAB commands needed to specify the model and compute the value of the cost function are given in Fig. 4.
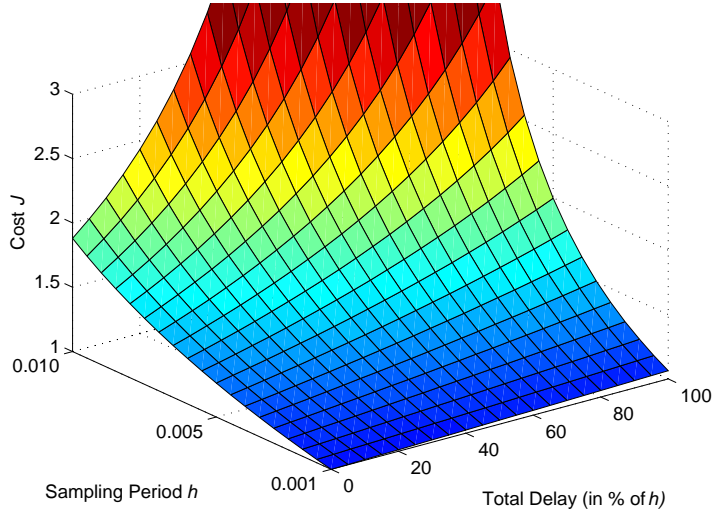
Figure 5 Example of a cost function computed using JITTERBUG. The plot shows the cost as a function of sampling period and delay in the networked control system example.

Sampling Period and Constant Delay A control system can typically give satisfactory performance over a range of sampling periods. In textbooks on digital control, rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval h should be chosen such that

$$0.2 < \omega_b h < 0.6,$$

where ω_b is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 ms. The effect of computational delay is typically not considered in such rules of thumb, however. Using JITTERBUG, the combined effect of sampling period and computational delay can be easily investigated. In Fig. 5, the cost function (1) for the networked control system has been evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for constant total delay ranging from 0 to 100% of the sampling interval. As can be seen, a one-sample delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost J goes to infinity).

Random Delays and Jitter Compensation If system resources are very limited (as they often are in embedded control applications), the control engineer may have to live with long sampling intervals. Delay in the control loop then becomes a serious issue. Ideally, the delay should be accounted for in the control design. In many practical cases, however, even the mean value of the delay will be unknown at design time. The actual delay at run-time will vary from sample to sample due to real-time scheduling, the load of the system, etc. A simple approach is to use gain scheduling—the actual delay is measured in each sample and the controller parameters are adjusted according to precalculated values that have been stored in a table. Since JITTERBUG allows time-dependent controller parameters, such delay compensation schemes can also be analyzed using the tool.

In the JITTERBUG model of the networked control system, we now assume that the delays τ_1 and τ_2 are uniformly distributed random variables between 0 and $\tau_{max}/2$, where τ_{max} denotes the maximum round-trip delay in the loop. A range of PD controller parameters (ranging from $K = 1.5$ and $T_d = 0.035$ for zero delay to $K = 0.78$ and $T_d = 0.052$ for 7.5 ms delay) are derived and stored in a table. When a sample arrives at the controller node, only the delay τ_1 from sensor to controller is known, however, so the remaining delay is predicted by its expected value of $\tau_{max}/4$.
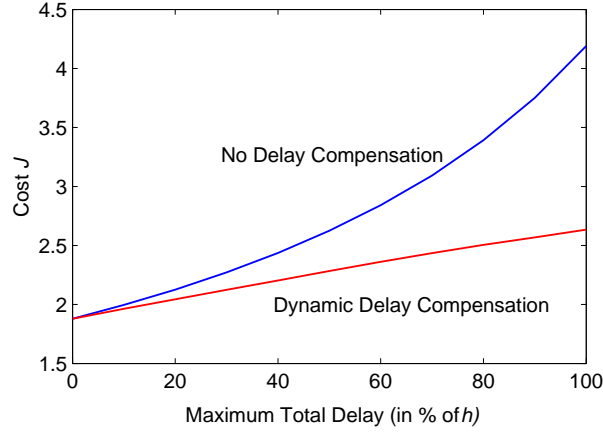
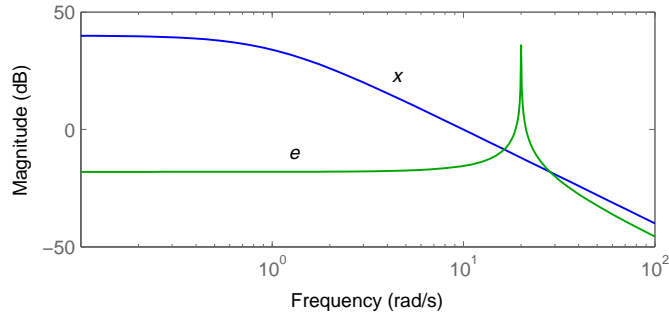**Figure 6** Cost as a function of maximum delay in the networked control system example with random delays.

Figure 7 Bode diagrams of the systems generating the good signal x and the disturbance signal e.

The sampling interval is set to $h = 10$ ms to make the effects of delay and jitter clearly visible. In Fig. 6, the cost function (1) has been evaluated with and without delay compensation for values of the maximum delay ranging from 0 to 100% of the sampling interval. The cost increases much more rapidly for the uncompensated system. The same example will be studied in more detail later using the TRUETIME simulator.

Signal Processing Application

As a second example, we will look at a signal processing application. Cleaning signals from disturbances using notch filters is important in many control systems. In some cases, the filters are very sensitive to lost samples due to their narrow-band frequency characteristics, and in real-time systems lost samples are sometimes inevitable. In this example, JITTERBUG is used to evaluate the effects of lost samples in different filters and possible compensation techniques.

The setup is as follows. A good signal x (modeled as low-pass-filtered white noise) is to be cleaned from an additive disturbance e (modeled as band-pass-filtered white noise). The Bode diagrams of the signal-generating systems are shown in Figure 7. An estimate $\hat{x}$ of the good signal should be found by applying a digital filter with the sampling interval $h = 0.1$ to the measured signal $x + e$. Unfortunately, a fraction p of the measurement samples are lost.

A JITTERBUG model of the system is shown in Fig. 8. The signals x and e are generated by filtered continuous-time white noise through the two continuous-time systems G_1 and G_2. The digital filter is represented as two discrete-time systems: *Samp* and *Filter*. The good signal is buffered in the system *Delay* and is then compared to the filtered estimate
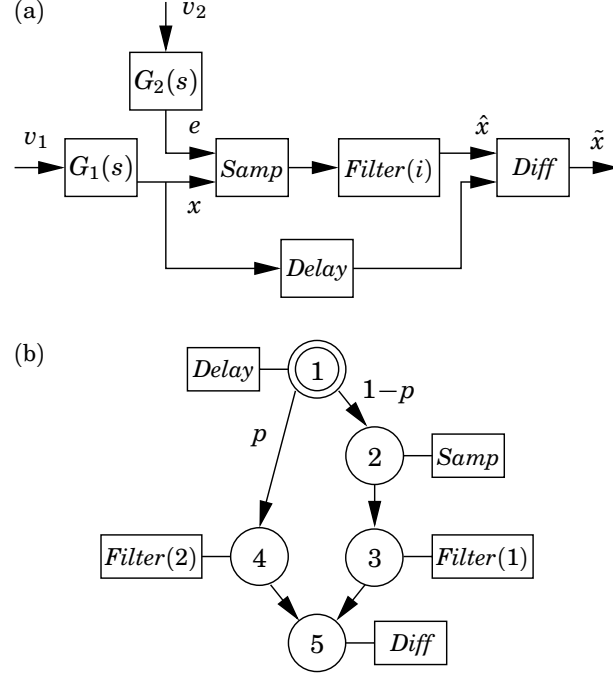
9

Figure 8 JITTERBUG model of the signal processing application: (a) signal model and (b) timing model.

in the system *Diff*.

In the execution model, there is a probability p that the *Samp* system will not be updated. In that case, an alternate version, *Filter*(2), of the filter dynamics can be executed and used to compensate for the lost sample.

Two different filters are compared. The first filter is an ordinary second-order notch filter with two zeros on the unit circle. It is updated with the same equations even if no sample is available. The second filter is a second-order, nonoptimal Kalman filter, which is based on a simplified model of the signal dynamics. In the case of a lost sample, only prediction is performed in the Kalman filter.

The performance of the filters is evaluated using the cost function

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \tilde{x}^2(t)\, dt,$$

which measures the variance of the estimation error. In Fig. 9, the cost has been plotted for different probabilities of lost samples. The figure shows that the ordinary notch filter performs better in the case of no lost samples, but the Kalman filter performs better as the probability of lost samples increases. This is because the Kalman filter can perform prediction when no sample is available.

Simulation Using TrueTime

Analysis using JITTERBUG can be used to quickly determine how sensitive a control system is to slow sampling, delay, jitter, and so on. For more detailed analysis as well as systemwide real-time design, the more general simulation tool TRUETIME can be used.

In TRUETIME, which is based on MATLAB/Simulink, computer and network blocks are introduced. The computer blocks are event-driven and execute user-defined tasks and interrupt handlers representing, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual computer blocks is arbitrary and decided by the user. Likewise, in the network, messages are sent and received according to a chosen network model. A comparison between a TRUETIME simulation model and a traditional simulation model of a distributed control system is shown in Fig. 10.
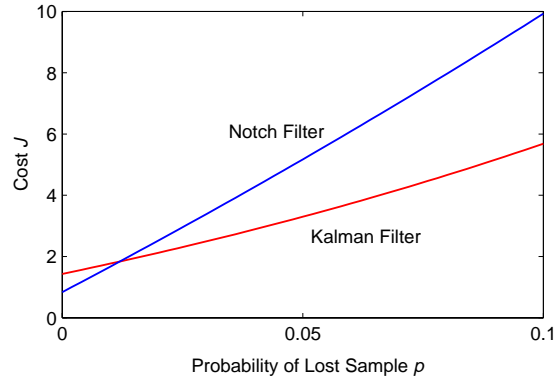
Figure 9 The variance of the estimation error in the different filters as a function of the probability of lost samples.
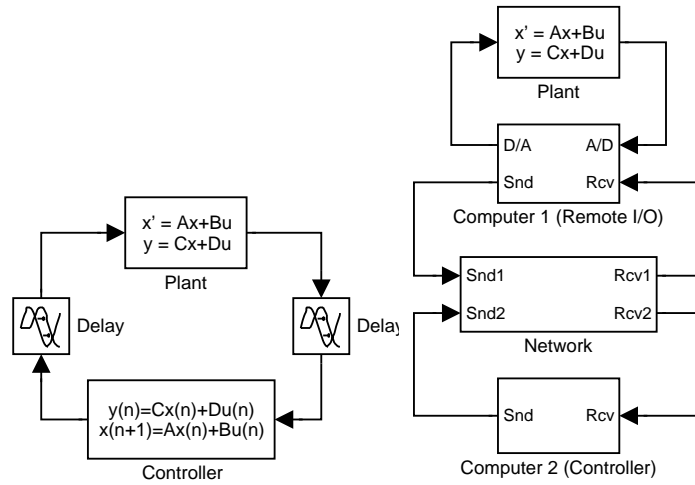

Figure 10 Left: Traditional simulation model of a distributed control system. Computers and network are modeled as simple delays. Right: TrueTime model where the execution of tasks and the transmission of messages are simulated in parallel with the plant dynamics.

The level of simulation detail is also chosen by the user—it is often neither necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TRUETIME allows the execution time of tasks and the transmission times of messages to be modeled as constant, random, or data-dependent. Furthermore, TRUETIME allows simulation of context switching and task synchronization using events or monitors.

TRUETIME can be used in several ways:

- to investigate the effects of timing nondeterminism, caused, for example., by preemption or transmission delays, on control performance

- to develop compensation schemes that adjust the controller dynamically based on measurements of actual timing variations

- to experiment with new, more flexible approaches to dynamic scheduling, such as feedback scheduling of CPU time and communication bandwidth and quality-of-service (QoS)-based scheduling approaches

- to simulate event-driven control systems (e.g., engine controllers and distributed controllers).

Simulation Environment

The interfaces to the computer and network Simulink blocks are shown in Fig. 11. Both blocks are event-driven, with the execution determined both by internal and external events.
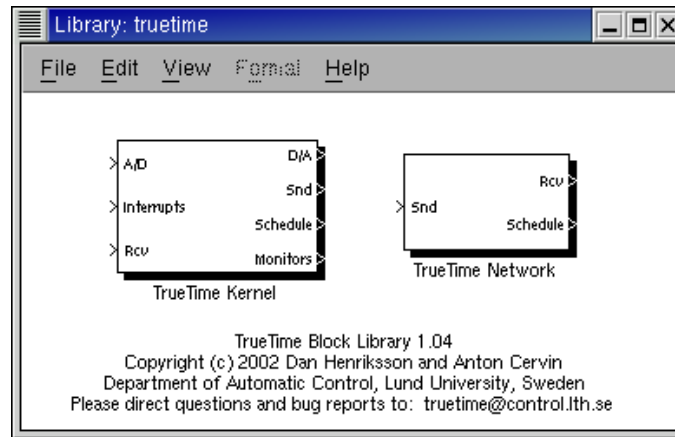
11

Figure 11 The TRUETIME block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

Internal events are timely and correspond to events such as "a timer has expired," "a task has finished its execution," or "a message has completed its transmission." External events correspond to external interrupts, such as "a message arrived on the network" or "the crank angle passed zero degrees."

The block inputs are assumed to be discrete-time signals, except the signals connected to the A/D converters of the computer block, which may be continuous-time signals. All outputs are discrete-time signals. The Schedule and Monitors outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

The blocks are variable-step, discrete, MATLAB S-functions written in C++, the Simulink engine being used only for timing and interfacing with the rest of the model (the continuous dynamics). It should thus be easy to port the blocks to other simulation environments, provided these environments support event detection (zero-crossing detection).

The Computer Block

The computer block S-function simulates a computer with a simple but flexible real-time kernel, A/D and D/A converters, a network interface, and external interrupt channels.

Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors and timers that have been created for the simulation.

The execution of tasks and interrupt handlers is defined by user-written code functions. These functions can be written either in C++ (for speed) or as MATLAB m-files (for ease of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams.

Tasks   The task is the main construct in the TRUETIME simulation environment. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and aperiodic activities, such as communication tasks and event-driven controllers.

An arbitrary number of tasks can be created to run in the TRUETIME kernel. Each task is defined by a set of attributes and a code function. The attributes include a name, a release time, a worst-case execution time, an execution time budget, relative and absolute deadlines, a priority (if fixed-priority scheduling is used), and a period (if the task is periodic). Some of the attributes, such as the release time and the absolute deadline, are constantly updated by the kernel during simulation. Other attributes, such as period and priority, are normally kept constant but can be changed by calls to kernel primitives when the task is executing.

Furthermore, it is possible to associate three different interrupt handlers with each task. A task termination handler will be triggered when the code function of the task has executed its last segment, see below. A default termination handler is provided by the kernel for periodic tasks. This simply updates the release and absolute deadline and puts the task to sleep until next period. In accordance with [8] two overrun handlers may also be attached

12

to each task: a deadline overrun handler (triggered if the task misses its deadline) and an execution time overrun handler (triggered if the task executes longer than its worst-case execution time).

Interrupts and Interrupt Handlers Interrupts may be generated in two ways: externally or internally. An external interrupt is associated with one of the external interrupt channels of the computer block. The interrupt is triggered when the signal of the corresponding channel changes value. This type of interrupt may be used to simulate engine controllers that are sampled against the rotation of the motor or distributed controllers that execute when measurements arrive on the network.

Internal interrupts are associated with timers. Both periodic timers and one-shot timers can be created. The corresponding interrupt is triggered when the timer expires. Timers are also used internally by the kernel to implement the overrun handlers described in the previous section.

When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt. An interrupt handler works much the same way as a task, but is scheduled on a higher priority level. Interrupt handlers will normally perform small, less time-consuming tasks, such as generating an event or triggering the execution of a task. An interrupt handler is defined by a name, a priority, and a code function. External interrupts also have a latency during which they are insensitive to new invocations.

Priorities and Scheduling Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the task level (lowest priority). The execution may be preemptive or nonpreemptive; this can be specified individually for each task and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the task level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a task is given by a user-defined priority function, which is a function of the task attributes. This makes it easy to simulate different scheduling policies. For instance, a priority function that returns a priority number implies fixed-priority scheduling, whereas a priority function that returns a deadline implies deadline-driven scheduling. Predefined priority functions exist for most of the commonly used scheduling schemes.

Code The code associated with tasks and interrupt handlers is scheduled and executed by the kernel as the simulation progresses. The code is normally divided into several segments, as shown in Fig. 12. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output delays, blocking when accessing shared resources, etc. The simulated execution time of each segment is returned by the code function, and can be modeled as constant, random, or even data-dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in the next segment when the task has been running for the time associated with the previous segment. This means that preemption from higher-priority activities and interrupts may cause the actual delay between the segments to be longer than the execution time.

Fig. 13 shows an example of a code function corresponding to the time line in Fig. 12. The function implements a simple controller. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution, which will trigger execution of the termination handler of the task.

The functions `calculateOutput` and `updateState` are assumed to represent the implementation of an arbitrary controller. The data structure `data` represents the local memory of the task and is used to store the control signal and measured variable between calls to the different segments. A/D and D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Besides A/D and D/A conversion, many other kernel primitives exist that can be called from the code functions. These include functions to send and receive messages over the
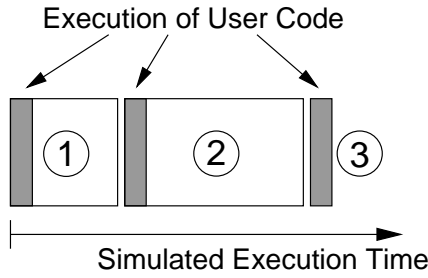
Execution of User Code

Simulated Execution Time

Figure 12 The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

```
function [exectime, data] = myController(seg, data)
switch seg,
   case 1,
      data.y = ttAnalogIn(1);
      data.u = calculateOutput(data.y);
      exectime = 0.002;
   case 2,
      ttAnalogOut(1, data.u);
      updateState(data.y);
      exectime = 0.003;
   case 3,
      exectime = -1; % finished
end
```

Figure 13 Example of a simple code function.

network, create and remove timers, perform monitor operations, and change task attributes. Some of the kernel primitives are listed in Table 1.

Graphical Controller Representation As an alternative to textual implementation of the controller algorithms, TRUETIME also allows for graphical representation of the controllers. Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions, using the primitive `ttCallBlockSystem`. A block diagram of a PI controller is shown in Fig. 14. The block system has two inputs, the reference signal and the process output, and two outputs, the control signal and the execution time.

Table 1 Examples of kernel primitives (pseudo syntax) that can be called from code functions associated with tasks and interrupt handlers.

|	
ttAnalogIn(ch)	Get the value of an input channel
ttAnalogOut(ch, val)	Set the value of an output channel
ttSendMsg(rec,data,len)	Send message over network
ttGetMsg()	Get message from network input queue
ttSleepUntil(time)	Wait until a specific time
ttCurrentTime()	Current time in simulation
ttCreateTimer(time,ih)	Trigger interrupt handler at a specific time
ttEnterMonitor(mon)	Enter a monitor
ttWait(ev)	Await an event
ttNotifyAll(ev)	Activate all tasks waiting for an event
ttSetPriority(val)	Change the priority of a task
ttSetPeriod(val)	Change the period of a task
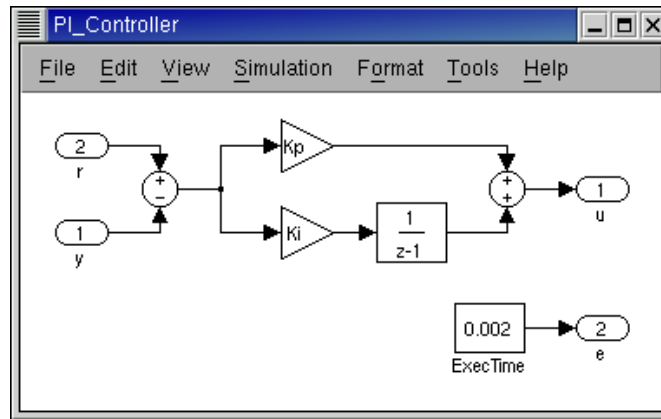
Figure 14 Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The example above shows a PI controller.

```
function [exectime, data] = eventController(seg, data)
switch (segment),
  case 1,
    ttWait('input_event');
    exectime = 0.0;
  case 2,
    data.y = ttAnalogIn(1);
    data.u = calculateOutput(data.y);
    exectime = 0.002;
  case 3,
    ttAnalogOut(1, data.u);
    updateState(data.y);
    exectime = 0.003;
  case 4,
    ttSetNextSegment(1); % loop
end
```

Figure 15 Example of a code function implementing an event-based controller.

Synchronization Synchronization between tasks is supported by monitors and events. Monitors are used to guarantee mutual exclusion when accessing common data. Events can be associated with monitors to represent condition variables. Events may also be free (i.e., not associated with a monitor). This feature can be used to obtain synchronization between tasks where no conditions on shared data are involved. The example in Fig. 15 shows the use of a free event input_event to simulate an event-driven controller task. The corresponding ttNotifyAll-call of the event is typically performed in an interrupt handler associated with an external interrupt port.

Output Graphs Depending on the simulation, several different output graphs are generated by the TRUETIME blocks. Each computer block will produce two graphs, a computer schedule and a monitor graph, and the network block will produce a network schedule. The computer schedule will display the execution trace of each task and interrupt handler during the course of the simulation. If context switching is simulated, the graph will also display the execution of the kernel. An example of such an execution trace is shown in Fig. 16. If the signal is high it means that the task is running. A medium signal indicates that the task is ready but not running (preempted), whereas a low signal means that the task is idle. In an analogous way, the network schedule shows the transmission of messages over the network, with the states representing sending (high), waiting (medium), and idle (low). The monitor graph shows which tasks are holding and waiting on the different monitors during the simulation. Generation of these execution traces is optional and can be specified
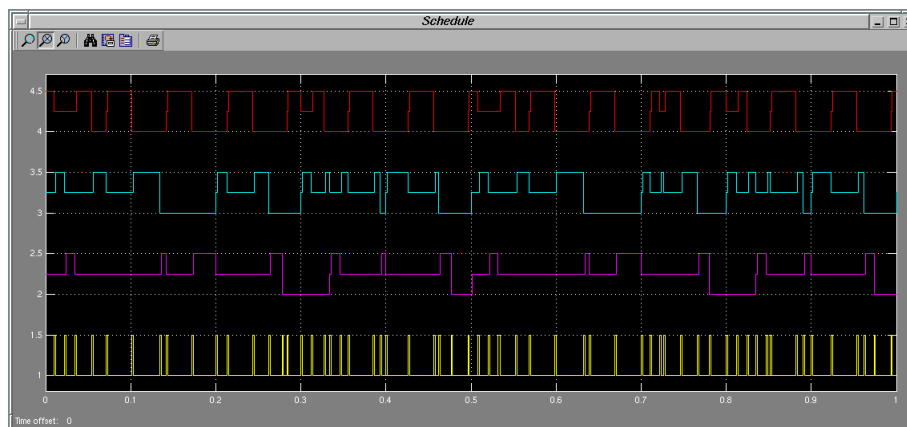
15

Figure 16 Example of an execution trace generated by a computer block during a simulation. The example involves three tasks. The lower graph shows the execution of the kernel simulating context switches.
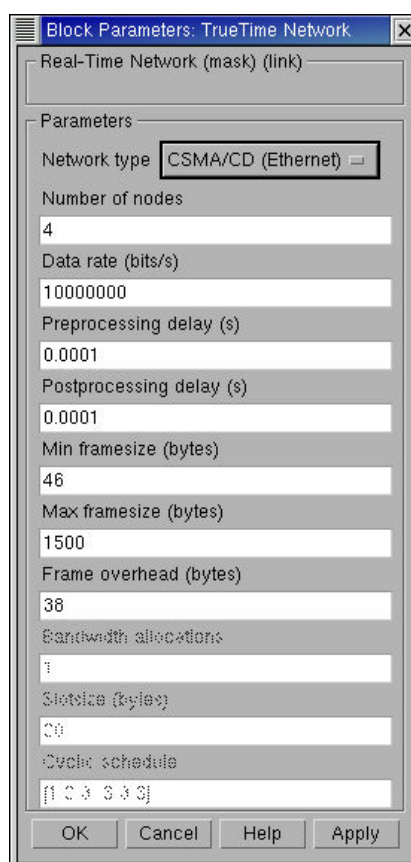

Figure 17 The dialogue of the TRUETIME Network block.

individually for each task, interrupt handler, and monitor.

The Network Block

The network model is similar to the real-time kernel model, albeit simpler. The network block is event-driven and executes when messages enter or leave the network. A message contains information about the sending and the receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

In the network block, it is possible to specify the transmission rate, the medium access control protocol (CSMA/CD, CSMA/CA, round robin, FDMA, or TDMA), and a number of

16

other parameters, see Fig. 17. A long message can be split into frames that are transmitted in sequence, each with an additional overhead. When the simulated transmission of a message has completed, it is put in a buffer at the receiving computer node, which is notified by a hardware interrupt.

Networked Control System

As a first example of simulation in TRUETIME, we again turn our attention to the networked control system. Using TRUETIME, general simulation of the distributed control system is possible wherein the effects of scheduling in the CPUs and simultaneous transmission of messages over the network can be studied in detail. TRUETIME allows simulation of different scheduling policies of CPU and network and experimentation with different compensation schemes to cope with delays.

The TRUETIME simulation model of the system contains one computer block for each node and a network block (see Fig. 18). The time-driven sensor node contains a periodic task, which at each invocation samples the process and sends the sample to the controller node over the network. The controller node contains an event-driven task that is triggered each time a sample arrives over the network from the sensor node. Upon receiving a sample, the controller computes a control signal, which is then sent to the event-driven actuator node, where it is actuated. Finally, the interference node contains a periodic task that generates random interfering traffic over the network.

Initialization of the Actuator Node   Fig. 19 shows the complete code needed to initialize the actuator node in this particular example. The computer block contains one task and one interrupt handler, and their execution is defined by the code functions `actcode` and `msgRcvHandler`, respectively. The task and interrupt handler are created in the `actuator_init` function together with an event (`packet`) used to trigger the execution of the task. The node is "connected" to the network in the function `ttInitNetwork` by supplying a node identification number and the interrupt handler to be executed when a message arrives to the node. In the `ttInitKernel` function the kernel is initialized by specifying the number of A/D and D/A channels and the scheduling policy. The built-in priority function `prioFP` specifies fixed-priority scheduling. Other predefined scheduling policies include rate monotonic (`prioRM`), earliest deadline first (`prioEDF`), and deadline monotonic (`prioDM`) scheduling.

Experiments   In the following simulations, we will assume a CAN-type network where transmission of simultaneous messages is decided based on priorities of the packages. The PD controller executing in the controller node is designed a 10-ms sampling interval. The same sampling interval is used in the sensor node.

In a first simulation, all execution times and transmission times are set equal to zero. The control performance resulting from this ideal situation is shown in Fig. 20.

Next we consider a more realistic simulation where execution times in the nodes and transmission times over the network are taken into account. The execution time of the controller is 0.5 ms and the ideal transmission time from one node to another is 1.5 ms. The ideal round-trip delay is thus 3.5 ms. The packages generated by the disturbance node have high priority and occupy 50% of the network bandwidth. We further assume that an interfering, high-priority task with a 7-ms period and a 3-ms execution time is executing in the controller node. Colliding transmissions and preemption in the controller node will thus cause the round-trip delay to be even longer on average and timevarying. The resulting degraded control performance can be seen in the simulated step response in Fig. 21. The execution of the tasks in the controller node and the transmission of messages over the network can be studied in detail (see Fig. 22).

Finally, a simple compensation is introduced to cope with the delays. The packages sent from the sensor node are now time-stamped, which makes it possible for the controller to determine the actual delay from sensor to controller. The total delay is estimated by adding the expected value of the delay from controller to actuator. The control signal is then calculated based on linear interpolation among a set of controller parameters precalculated for different delays. Using this compensation, better control performance is obtained, as seen in Fig. 23.
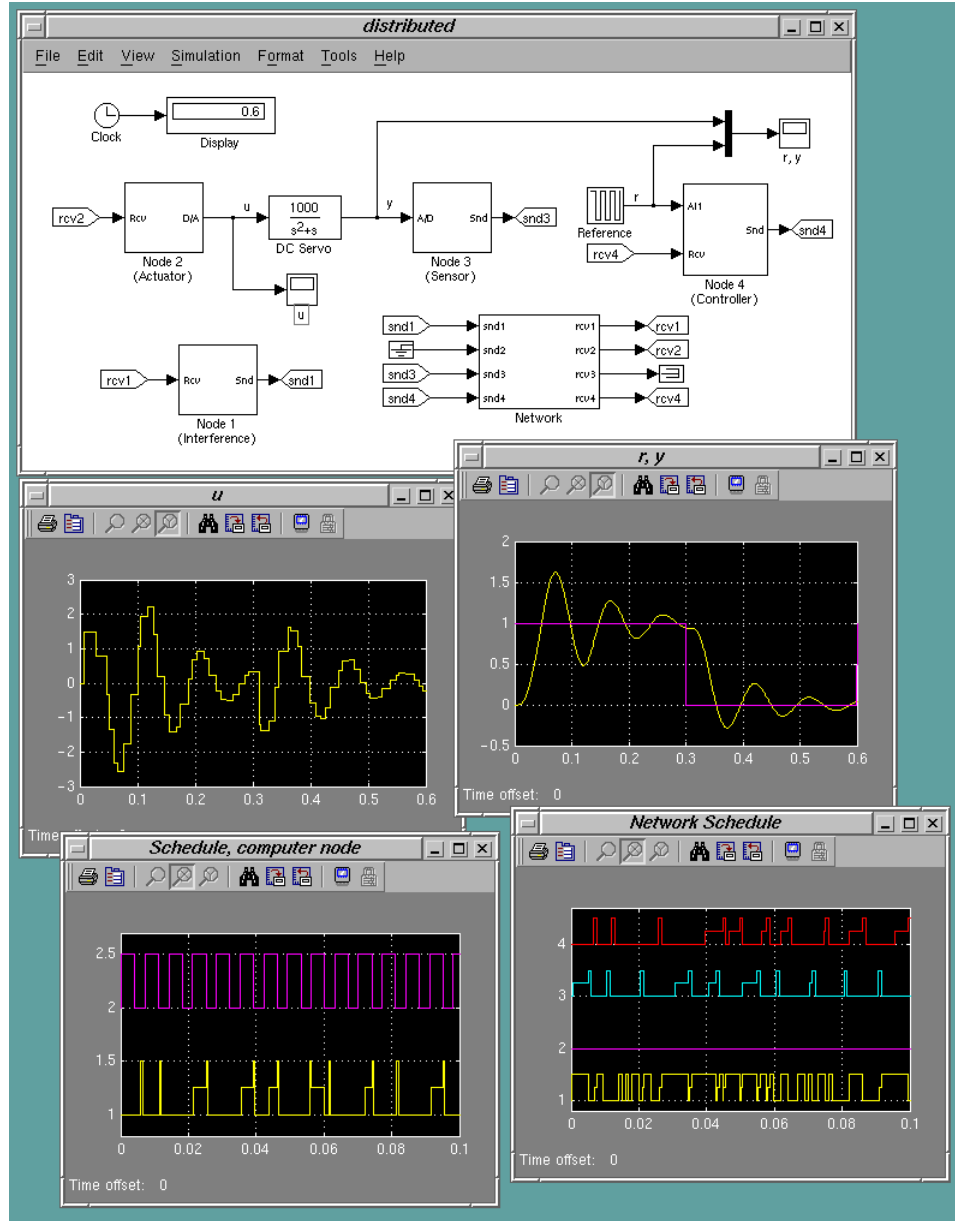
Figure 18 TRUETIME simulation of the networked control system. The poor control performance is a result of delays caused by colliding network transmissions and preemption in the controller node.

Feedback Scheduling

An a second example, we will look at a feedback scheduling application. Some controllers, including hybrid controllers that switch between different modes, can have highly varying execution-time demands. This makes the real-time scheduling design for this type of controller difficult. Basing the real-time design on worst-case execution time (WCET) estimates may lead to low utilization, slow sampling, and poor control performance. On the other hand, basing the real-time design on average-case assumptions may lead to temporary CPU overloads and, again, poor control performance.

One way to solve the problem is to introduce feedback in the real-time system. The CPU overload problem can be resolved by online adjustment of the sampling frequencies of the hybrid controllers based on feedback from execution-time measurements. The scheduler may also use feedforward information from control tasks that are about to switch mode. The scheme was originally presented in [9] and is illustrated in Fig. 24.

In this example, we consider feedback scheduling of a set of double-tank controllers. The

```
%% Code function for the actuator task
function [exectime, data] = actcode(seg, data)

  switch seg,
    case 1,
       ttWait('packet');
       exectime = 0.0;
    case 2,
       data.u = ttGetMsg;
       exectime = 0.0005;
    case 3,
       ttAnalogOut(1, data.u);
       ttSetNextSegment(1); % wait for new msg
       exectime = 0.0;
  end

%% Code function for the network interrupt handler
function [exectime, data] = msgRcvHandler(seg, data)

  ttNotifyAll('packet');
  exectime = -1;

%% Initialization function
%% creating the task, interrupt handler and event
function actuator_init

  nbrOfInputs = 0;
  nbrOfOutputs = 1;
  ttInitKernel(nbrOfInputs, nbrOfOutputs, 'prioFP');

  priority = 5;
  deadline = 0.010;
  release = 0.0;
  ttCreateTask('act_task', deadline, priority, 'actcode');
  ttCreateJob('act_task', release);

  ttCreateInterruptHandler('msgRcv', 1, 'msgRcvHandler');
  ttInitNetwork(2, 'msgRcv'); % I am node 2
  ttCreateEvent('packet');
```

Figure 19   Complete initialization of the actuator node in the networked control system simulation.
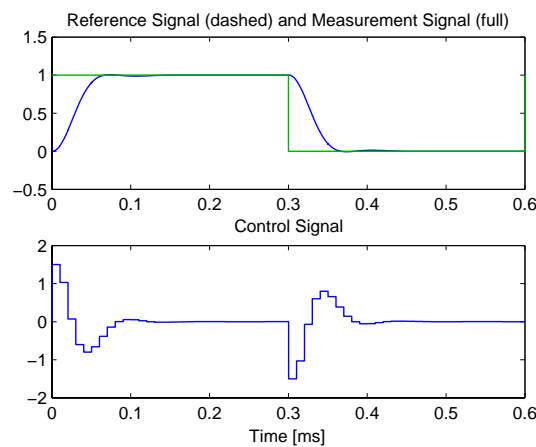

Figure 20   Control performance without time delay.

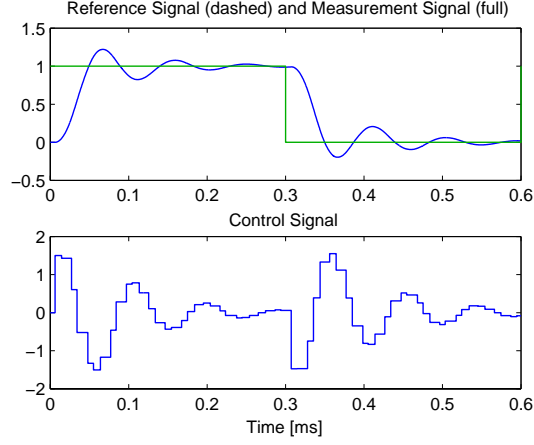**Figure 21** Control performance with interfering network messages and interfering task in the controller node.
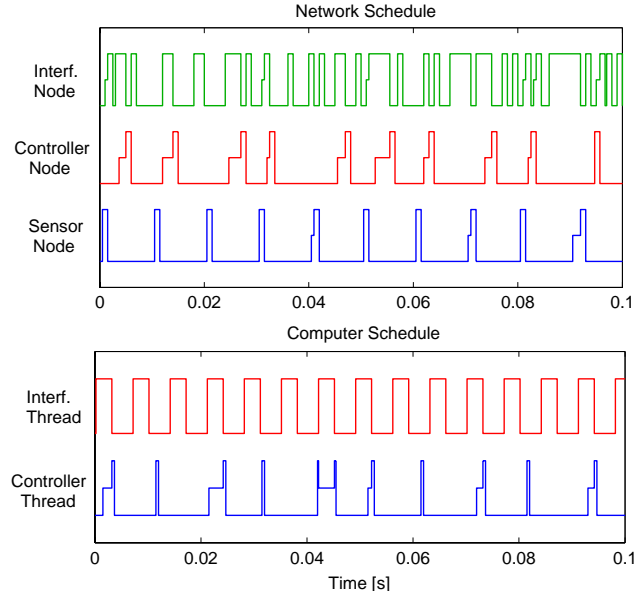

Figure 22 Close-up of schedules showing the allocation of common resources: network (top) and controller node (bottom). A high signal means sending or executing, a medium signal means waiting, and a low signal means idle.
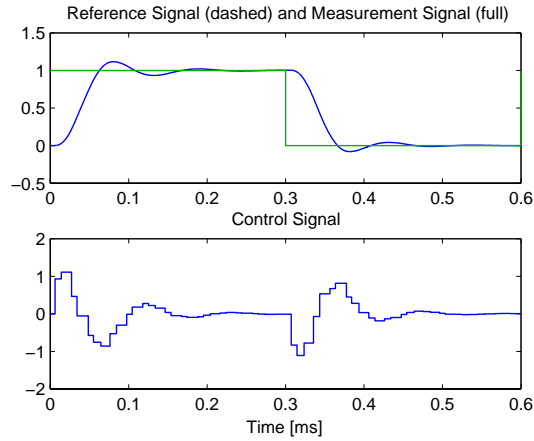

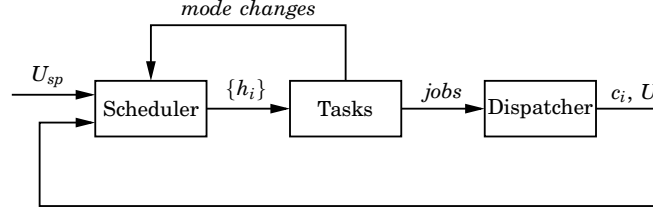**Figure 23** Control performance with delay-compensation.

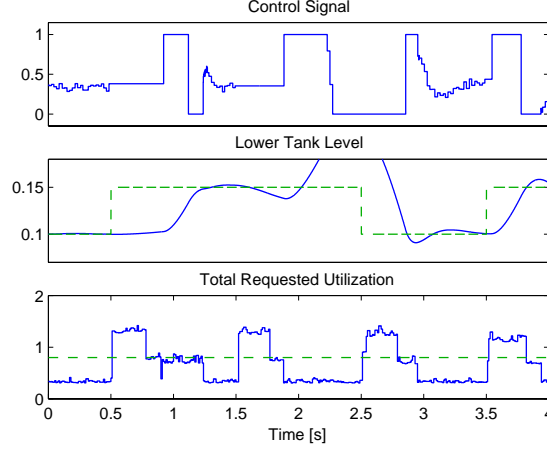**Figure 24** The feedback scheduling structure.

Figure 25 Performance of Controller 1 under ordinary rate-monotonic scheduling. The CPU becomes overloaded and the controller is blocked, which deteriorates the performance.

double-tank process is described by the nonlinear state-space equations of the form

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} -\alpha\sqrt{x_1} + \beta u \\ \alpha\sqrt{x_1} - \alpha\sqrt{x_2} \end{pmatrix}$$

The objective is to control the level of the lower tank, x_2, using the pump, u. A hybrid controller for the double-tank process was presented in [10]. The controller consisted of two subcontrollers: a time-optimal controller for set-point changes and a PID controller for steady-state regulation.

Measurements on the controller showed that in optimal control mode, the execution time was about three times longer than in PID control mode. The problem becomes pronounced when several hybrid controllers share a common computational unit. In the worst case, all controllers will be in optimal control mode at the same time, and the CPU load can become very high.

Experiments   It is assumed that three hybrid double-tank controllers should be scheduled on the same computer. The tanks have different time constants, $(T_1, T_2, T_3) = (210, 180, 150)$, and the corresponding controllers are therefore assigned different nominal sampling periods $(h_{nom1}, h_{nom2}, h_{nom3}) = (21, 18, 15)$ ms. Each controller is implemented as a separate TRUETIME task. The simulated execution time of a controller in PID mode is $C_{PID} = 2$ ms and the simulated execution time of a controller in optimal control mode is $C_{Opt} = 10$ ms.

First, ordinary rate-monotonic scheduling is attempted. According to this scheduling principle, the task with the longest period gets the lowest priority. In the worst case, when all controllers are in optimal control mode, the utilization will be $U = \sum \frac{C_i}{h_i} = 1.7$ and the lowest-priority task (Controller 1) will be blocked. Simulation results are shown in Figs. 25 and 26 displaying the control performance of the low-priority controller task and a close-up of the computer schedule. The performance of Controller 1 is very poor due to preemption from the higher-priority tasks.

Next a feedback scheduler is introduced. The feedback scheduler is implemented as a task executing at the highest priority with a period of $h_{FBS} = 100$ ms and an execution
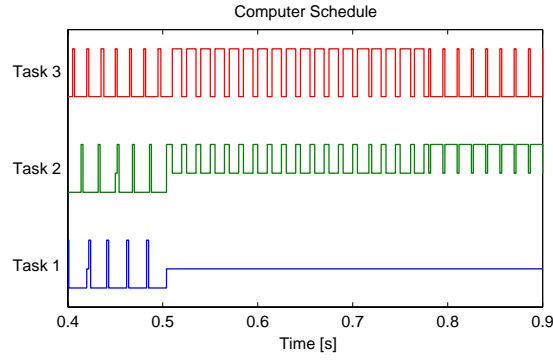
21

Figure 26 Close-up of the computer schedule during ordinary rate-monotonic scheduling. When the system becomes overloaded, the low-priority controller is preempted during a significant amount of time.
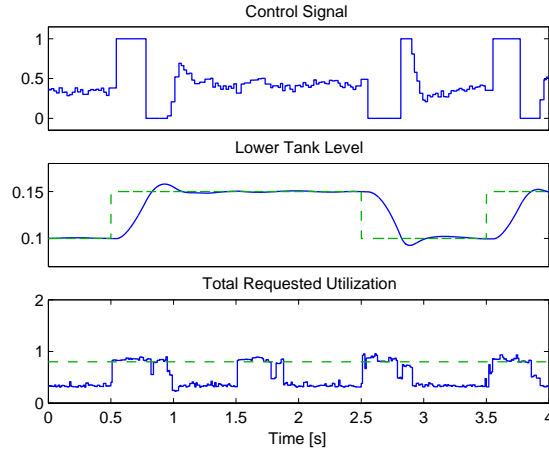

Figure 27 Performance of Controller 1 under feedback scheduling. The CPU utilization is controlled never to exceed 0.8, and the control performance is good throughout.

time of $C_{FBS} = 2$ ms. It also executes an extra time whenever a task switches from PID to optimal mode. The feedback scheduler estimates the workload of the controllers and rescales the task periods, if necessary, to achieve a utilization level of at most $U_{sp} = 0.8$. Results from a simulation are shown in Figs. 27 and 28. The performance of Controller 1 is much better, even though it cannot always execute at its nominal period.

Conclusion

Designing a real-time control system is essentially a co-design problem. Choices made in the real-time design will affect the control design and vice versa. For instance, deciding on a particular network protocol will give rise to certain delay distributions that must be taken into account in the controller design. On the other hand, bandwidth requirements in the control loops will influence the choice of CPU and network speed. Using an analysis tool such as JITTERBUG, one can quickly assert how sensitive the control loop is to slow sampling rates, delay, jitter, and other timing problems. Aided by this information, the user can proceed with more detailed, systemwide real-time and control design using a simulation tool such as TRUETIME.

JITTERBUG allows the user to compute a quadratic performance criterion for a linear control system under various timing conditions. The control system is described using a number of continuous-time and discrete-time linear systems. A stochastic timing model with random delays is used to describe the execution of the system. The tool can also be used to investigate aperiodic controllers, multirate controllers, and jitter-compensating
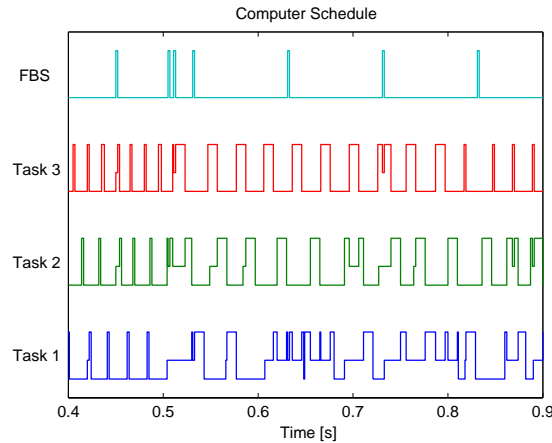
22

Figure 28 Close-up of the computer schedule during feedback scheduling. The sampling intervals of the tasks are rescaled to avoid overload.

controllers.

TRUETIME facilitates event-based co-simulation of a multitasking real-time kernel containing controller tasks and the continuous dynamics of controlled plants. The simulations capture the true, timely behavior of real-time controller tasks and communication networks, and dynamic control and scheduling strategies can be evaluated from a control performance perspective. The controllers can be implemented as M-functions, C-functions, or ordinary, discrete-time Simulink blocks.

Acknowledgments

References

[1] *IEEE Control Systems Magazine*, Special Section on Networks and Control, vol. 21, no. 1, Feb. 2001.

[2] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Boston, MA: Kluwer Academic Pub., 1997.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Boston, MA: Kluwer Academic Pub., 1993.

[4] J.W.S. Liu, *Real-Time Systems*. Upper Saddle River, NJ: Prentice Hall, 2000.

[5] J. Eker, P. Hagander, and K.-E. Årzén, "A feedback scheduler for real-time control tasks," *Control Engineering Practice*, vol. 8, no. 12, 2000, pp. 1369–1378.

[6] J. Nilsson, *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT–1049–SE, Department of Automatic Control, Lund Institute of Technology, Sweden, January 1998.

[7] B. Lincoln and A. Cervin, "Jitterbug: A tool for analysis of real-time control performance," in *Proc. 41st IEEE Conf. on Decision and Control*, Las Vegas, NV, 2002.

[8] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull, *The Real-Time Specification for Java*. Reading, MA: Addison-Wesley, 2000.

[9] A. Cervin and J. Eker, "Feedback scheduling of contol tasks," in *Proc. 39th IEEE Conf. on Decision and Control*, Sydney, Australia, 2000, pp. 4871–4876.

[10] J. Eker and J. Malmborg, "Design and implementation of a hybrid control strategy," *IEEE Control Systems Magazine*, vol. 19, no. 4, Aug. 1999, pp. 12–21.

Jitterbug Reference Manual

Anton Cervin
Bo Lincoln

Department of Automatic Control
Lund Institute of Technology
January 2003

Author(s)
Anton Cervin, Bo Lincoln

Supervisor

Sponsoring organisation

Title and subtitle
Jitterbug Reference Manual

Abstract

The manual describes the use of Jitterbug, a Matlab toolbox for real-time control performance analysis. The tool facilitates the computation of a quadratic performance index for a linear control system under various timing conditions.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

Contents

1. Introduction

JITTERBUG [Lincoln and Cervin, 2002] is a MATLAB-based toolbox that allows the computation of a quadratic performance criterion for a linear control system under various timing conditions. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. As an additional feature, it is also possible to compute the spectral density of the signals in the control system. The main contribution of the toolbox, which is built on well-known theory (LQG theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

2. System Description

In JITTERBUG, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example of a JITTERBUG model is shown in Figure 1, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

2.1 Signal Model
The signal model consists of a number of inter-connected continuous-time and discrete-time linear systems driven by white noise. The cost is specified as a stationary, continuous-time quadratic cost function.
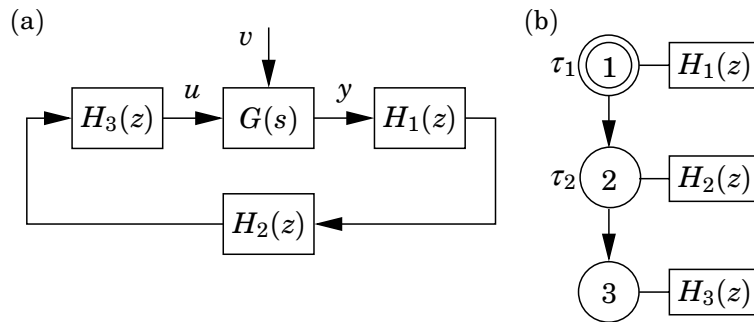

Figure 1 A simple JITTERBUG model of a computer-controlled system: (a) signal model and (b) timing model.

Continuous-Time Systems   A continuous-time system is described by

$$\dot{x}_c(t) = Ax_c(t) + Bu(t) + v_c(t)$$
$$y^0(t) = Cx_c(t) \qquad \text{(continuous output)}$$
$$y(t_k) = y^0(t_k) + e_d(t_k) \quad \text{(measured discrete output)}$$

where A, B, and C are constant matrices, and v_c is a continuous-time white-noise process with zero mean and covariance matrix R_1 (strictly speaking, v_c has the spectral density $\phi(\omega) = \frac{1}{2\pi}R_1$), and e_d is a discrete-time white-noise process with zero mean and covariance matrix R_2. Note that direct terms are not allowed (i.e., the system must be strictly proper). Also note that there is no *continuous-time* output noise. The ability to specify discrete-time measurement noise in connection with the plant is only offered as a convenience. The discrete-time output noise will be translated to input noise at any connected discrete-time system(s).

The cost of the system is specified as

$$J_c = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x_c(t) \\ u(t) \end{pmatrix}^T Q_c \begin{pmatrix} x_c(t) \\ u(t) \end{pmatrix} dt$$

where Q_c is a positive semi-definite matrix.

The system may also be specified in transfer-function form (see the description of `addcontsys` on page 27).

Discrete-Time Systems   A discrete-time system is described by

$$x_d(t_{k+1}) = \Phi x_d(t_k) + \Gamma u(t_k) + v_d(t_k)$$
$$y^0(t_k) = Cx_d(t_k) + Du(t_k) \quad \text{(discrete output)}$$
$$y(t_k) = y^0(t_k) + e_d(t_k) \qquad \text{(measured discrete output)}$$

where Φ, Γ, C, and D are possibly time-varying matrices (see below). The covariance of the discrete-time white noise processes v_d and e_d is given by

$$R = \mathbf{E} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix}^T.$$

The update instants t_k are determined by the timing model and are not necessarily equidistant in time. The input signal u is sampled when the system is updated, and the state x_d and the output signal y^0 are held between updates.

The cost of the system is specified as

$$J_d = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x_d(t) \\ y^0(t) \\ u(t) \end{pmatrix}^T Q_d \begin{pmatrix} x_d(t) \\ y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q_d is a positive semi-definite matrix. Note that $x_d(t)$ and $y^0(t)$ are piecewise constant signals, while $u(t)$ may be a continuous signal.

The system may also be specified in transfer-function form (see the description of `adddiscsys` on page 29).

Connecting Systems The total system is formed by appropriately connecting the inputs and outputs of a number of continuous-time and discrete-time systems. Throughout, MIMO formulations are allowed, and a system may collect its inputs from a number of other systems. The total cost to be evaluated is summed over all continuous-time and discrete-time systems:

$$J = \sum J_c + \sum J_d \tag{1}$$

It's important to understand how cost and noise are handled when systems are interconnected. Three principal cases can be distinguished (see Figure 2):

(a) The interconnection of two continuous-time systems. Note that any discrete-time output noise e_d will be ignored.

(b) The interconnection of two discrete-time systems. No surprises here.

(c) The interconnection of a continuous-time and a discrete-time system. Note that the discrete-time output noise e_d will not be included in the input cost of the discrete-time system.

2.2 Timing Model

The timing model consists of a number of timing nodes. Each node can be associated with zero or more discrete-time systems in the signal model, which should be updated when the node becomes active. At time zero, the first node is activated. The first node can also be declared to be *periodic* (indicated by an extra circle in the illustrations), which means that the execution will restart at this node every h seconds. This is useful for modeling periodic controllers and also greatly simplifies the cost calculations.

Each node is associated with a time delay τ, which must elapse before the next node can become active. (If unspecified, the delay is assumed to be zero.) The delay can be used to model computational delay, transmission delay in a network, etc. A delay is described by a discrete-time probability density function

$$P_\tau = [\, P_\tau(0) \quad P_\tau(1) \quad P_\tau(2) \quad \ldots \,],$$

where $P_\tau(k)$ represents the probability of a delay of $k\delta$ seconds. The time grain δ is a constant that is specified for the whole model.

In periodic systems, the execution is preempted if the total delay $\sum \tau$ in the system exceeds the period h. Any remaining timing nodes will be skipped. This models a real-time system where hard deadlines (equal to the period) are enforced and the control task is aborted at the deadline.

An aperiodic system can be used to model a real-time system where the task periods are allowed to drift if there are overruns. It could also be used to model a controller that samples "as fast as possible" instead of waiting for the next period.

Figure 2 Possible interconnections of continuous-time and discrete-time systems.

Node- and Time-Dependent Execution   The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations (i.e., different Φ, Γ, C and D matrices) in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. An example of this type is given later.

It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers for example.

Alternative Execution Paths   For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In JIT-TERBUG, two such cases can be modeled (see Figure 3):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with proba-
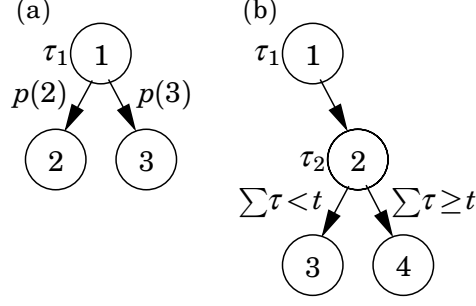
Figure 3 Alternative execution paths in a JITTERBUG execution model: (a) random choice of path and (b) choice of path depending on the total delay from the first node.

bility $p(i)$. This can be used to model a sample being lost with some probability.

(b) A vector n of next nodes can be specified with a timevector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be activated next. This can be used to model time-outs and various compensation schemes.

3. Internal Workings

Inside JITTERBUG, the states and the cost are considered in continuous time. The inherently discrete-time states, e.g. in discrete-time controllers or filters, are treated as continuous-time states with zero dynamics. This means that the total system can be written as

$$\dot{x}(t) = Ax(t) + w(t) \tag{2}$$

where x collects all the states in the system, and w is continuous-time white noise process with covariance $\widetilde{R}$. To model the discrete-time changes of some states as a timing node n is activated, the state is instantaneously transformed by

$$x(t^+) = E_n x(t) + e_n(t)$$

where e_n is a discrete-time white noise process with covariance W_n.

The total cost (1) for the system can be written as

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T x^T(t) \widetilde{Q} x(t)\, dt \tag{3}$$

where $\widetilde{Q}$ is a positive semidefinite matrix.

3.1 Sampling the System

JITTERBUG relies on discretized time to calculate the variance of the states and the cost. No approximations are involved, however. Sampling the system (2) with a period of δ (the time-grain in the delay distributions) gives

$$x(k\delta + \delta) = \Phi x(k\delta) + v(k\delta) \tag{4}$$

6

where the covariance of v is R, and the cost (3) becomes

$$J = \lim_{N \to \infty} \frac{1}{N\delta} \sum_{k=0}^{N-1} \left(x^T(k\delta)Qx(k\delta) + q \right)$$

The matrices Φ, R, Q, and q are calculated as

$$\Phi = e^{A\delta}$$

$$R = \int_0^\delta e^{A(\delta-\tau)} \widetilde{R} e^{A^T(\delta-\tau)} \, d\tau$$

$$Q = \int_0^\delta e^{A^T t} \widetilde{Q} e^{At} \, dt$$

$$q = \text{tr}\left(\widetilde{Q} \int_0^\delta \int_0^\delta e^{A(t-\tau)} \widetilde{R} e^{A^T(t-\tau)} \, d\tau \, dt \right)$$

or, equivalently, from

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} = \exp\left(\begin{pmatrix} -A^T & \widetilde{Q} \\ 0 & A \end{pmatrix} \delta \right)$$

and

$$\begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{pmatrix} = \exp\left(\begin{pmatrix} -A & I & 0 \\ 0 & -A & \widetilde{R}^T \\ 0 & 0 & A^T \end{pmatrix} \delta \right)$$

so that

$$\Phi = P_{22}$$
$$Q = P_{22}^T P_{12}$$
$$R = M_{33}^T M_{23}$$
$$q = \text{tr}\left(Q M_{33}^T M_{13} \right)$$

3.2 Timing Representation

As time is discretized, we can transform the system description into a jump linear system, where the Markov state represents the current timing state of the system. Each timing node is represented by one Markov node. In between timing nodes additional Markov nodes representing the delay are inserted as illustrated in Figure 4.

Consider following one path in the Markov chain. For each node which is not a timing node, only the continuous states of the system change. In each time-step, they evolve as in (4), and thus the state covariance $P(k\delta) = \mathbf{E}\left\{ x(k\delta)x^T(k\delta) \right\}$ evolves as

$$P(k\delta + \delta) = \Phi P(k\delta)\Phi^T + R$$

At each timing node n, the system is additionally transformed as in (3),

$$P(k\delta^+) = E_n P(k\delta) E_n^T + W_n$$
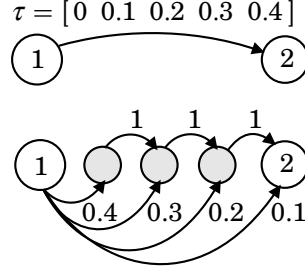
Figure 4 A random delay (above) modeled as a jump linear system (below), where the delay is represented by additional Markov nodes in between the timing nodes.
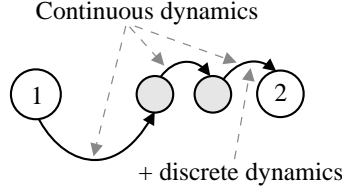
Figure 5 The continuous-time dynamics is active between all Markov nodes, whereas the discrete-time dynamics is activated only before a timing node.

where W_n is the covariance of the discrete-time noise $e_n(k\delta)$ in node n. See Figure 5 for an illustration. Combining the above, we define Φ_n as

$$\Phi_n = \begin{cases} \Phi & \text{if } n \text{ is not a timing node} \\ E_n\Phi & \text{if } n \text{ is a timing node} \end{cases}$$

and similarly R_n as

$$R_n = \begin{cases} R & \text{if } n \text{ is not a timing node} \\ E_n R E_n^T + W_n & \text{if } n \text{ is a timing node} \end{cases}$$

3.3 Calculating Variance and Cost

Now consider all possible Markov states simultaneously. Let $\pi_n(k\delta)$ be the probability of being in Markov state n at time $k\delta$, and let $P_n(k\delta)$ be the covariance of the state if the system is in Markov state n at time $k\delta$. Furthermore, let the transition matrix of the Markov chain be σ, such that

$$\pi(k\delta + \delta) = \sigma\pi(k\delta)$$

The state covariance then evolves as

$$P_n(k\delta + \delta) = \sum_i \sigma_{ni}\pi_i(k\delta)\Big(\Phi_n P_i(k\delta)\Phi_n^T + R_n\Big) \tag{5}$$

and the immediate cost at time $k\delta$ is calculated as

$$\frac{1}{\delta}\sum_n \pi_n(k\delta)\Big(\mathrm{tr}\big(P_n(k\delta)Q\big) + q\Big)$$

For systems without a periodic node, equation (5) must be iterated until the cost and variance converge. For periodic systems, the Markov state always

returns to the periodic timing node every h/δ time steps. As equation (5) is affine in P, we can find the stationary covariance $P_1(\infty)$ in the periodic node by solving a linear system of equations. The total cost is then calculated over the timesteps in one period. The toolbox returns the cost $J = \infty$ if the system is not mean-square stable.

3.4 Calculating Spectral Densities

For periodic systems, the toolbox also computes the discrete-time spectral densities of all outputs as observed in the periodic timing node. The spectral density of an output y is defined as

$$\phi_y(\omega) = \frac{1}{2\pi} \sum_{k=-\infty}^{\infty} r_y(k) e^{-ik\omega}$$

The covariance function $r_y(k)$ is given by

$$r_y(k) = \mathbf{E} \left\{ y(t) y^T(t + kh) \right\} = \mathbf{E} \left\{ Cx(t) x^T(t + kh) C^T \right\}$$
$$= \mathbf{E} \left\{ C\bar{\Phi}^{|k|} x(t) x^T(t) C^T \right\} = C\bar{\Phi}^{|k|} P_1(\infty) C^T$$

where $\bar{\Phi}$ is the average transition matrix over a period, and $P_1(\infty)$ is the stationary covariance in the periodic node. The spectral density is returned as a discrete-time linear system $F(z)$ such that $\phi_y(\omega) = F(e^{i\omega})$.

4. Examples

In this section, various examples that illustrate the use of JITTERBUG are given.

4.1 Distributed Control System

In the example, we will study the distributed control system shown in Figure 6. The setup is taken from [Nilsson, 1998]. In the control loop, the sen-
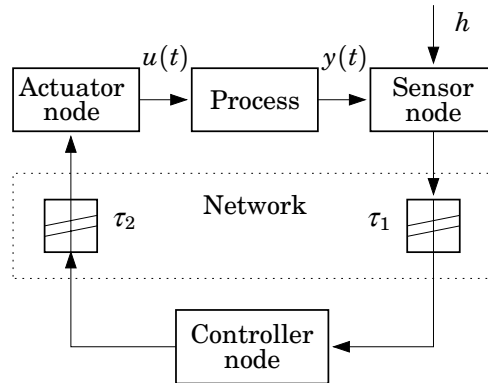
Figure 6 Distributed control system with communication delays τ_1 and τ_2.

sor, the actuator, and the controller are distributed among different nodes in a network. The sensor node is assumed to be time-driven, whereas the controller and actuator nodes are assumed to be event-driven. At a fixed

period h, the sensor samples the process and sends the measurement sample over the network to the controller node. There the controller computes a control signal and sends it over the network to the actuator node, where it is subsequently actuated.

The JITTERBUG model of the system was shown in Figure 1 on page 2. The DC servo process is given by the continuous-time system

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

The process is driven by white continuous-time input noise. There is assumed to be no measurement noise.

The process is sampled periodically with the interval h. The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1,$$

and the discrete-time PD controller is implemented as

$$H_2(z) = -K \left(1 + \frac{T_d}{h} \frac{z-1}{z} \right),$$

where the controller parameters are chosen as $K = 1.5$ and $T_d = 0.035$. (A real implementation would include a low-pass filter in the derivative part, but that is ignored here.)

The delays in the computer system are modeled by the two random variables τ_1 and τ_2. The total delay from sampling to actuation is given by $\tau_{tot} = \tau_1 + \tau_2$. It is assumed that the total delay never exceeds the sampling period (otherwise JITTERBUG would skip the remaining updates).

As a cost function, we choose the sum of the squared process input and the squared process output:

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \left(y^2(t) + u^2(t) \right) dt. \tag{6}$$

Sampling Period and Constant Delay   A control system can typically give satisfactory performance over a range of sampling periods. In textbooks on digital control, rules of thumb for sampling period selection are often given. One such rule suggests that the sampling interval h should be chosen such that

$$0.2 < \omega_b h < 0.6,$$

where ω_b is the bandwidth of the closed-loop system. In our case, a continuous-time PD controller with the given parameters would give a bandwidth of about $\omega_b = 80$ rad/s. This would imply a sampling period of between 2.5 and 7.5 ms. The effect of computational delay is typically not considered in such rules of thumb, however. Using JITTERBUG, the combined effect of sampling period and computational delay can be easily investigated. In Figure 7, the cost function (6) for the networked control system has been
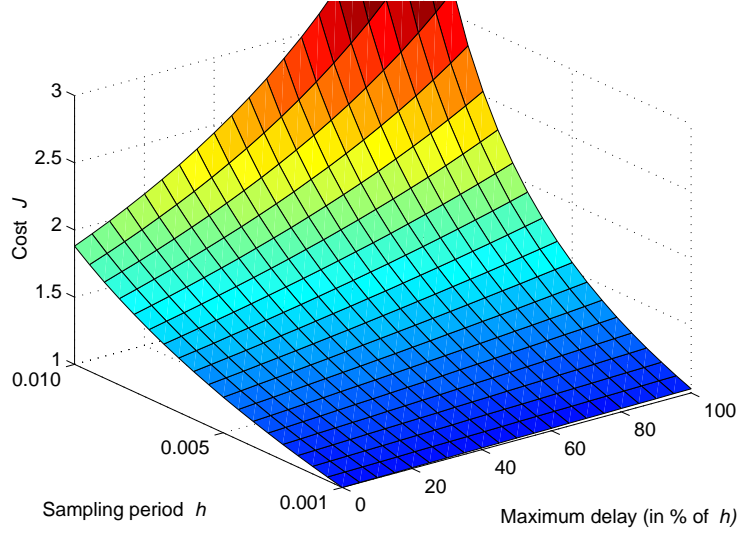
Figure 7 The cost as a function of sampling period and constant delay in the distributed control system example.

evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for constant total delay ranging from 0 to 100% of the sampling interval. As can be seen, a one-sample delay gives negligible performance degradation when $h = 1$ ms. When $h = 10$ ms, a one-sample delay makes the system unstable (i.e., the cost J goes to infinity).

Random Delays and Jitter Compensation If system resources are very limited (as they often are in embedded control applications), the control engineer may have to live with long sampling intervals. Delay in the control loop then becomes a serious issue. Ideally, the delay should be accounted for in the control design. In many practical cases, however, even the mean value of the delay will be unknown at design time. The actual delay at run-time will vary from sample to sample due to real-time scheduling, the load of the system, etc. A simple approach is to use gain scheduling—the actual delay is measured in each sample and the controller parameters are adjusted according to precalculated values that have been stored in a table. Since JITTERBUG allows time-dependent controller parameters, such delay compensation schemes can also be analyzed using the tool.

In the JITTERBUG model of the networked control system, we now assume that the delays τ_1 and τ_2 are uniformly distributed random variables between 0 and $\tau_{max}/2$, where τ_{max} denotes the maximum round-trip delay in the loop. A range of PD controller parameters (ranging from $K = 1.5$ and $T_d = 0.035$ for zero delay to $K = 0.78$ and $T_d = 0.052$ for 7.5 ms delay) are derived and stored in a table. When a sample arrives at the controller node, only the delay τ_1 from sensor to controller is known, however, so the remaining delay is predicted by its expected value of $\tau_{max}/4$.

In Figure 8, the cost function (6) for the networked control system has been evaluated for different sampling periods in the interval 1 to 10 milliseconds, and for maximum total delay ranging from 0 to 100% of the sampling interval. Compared to Fig 7, the cost is considerably lower.

11
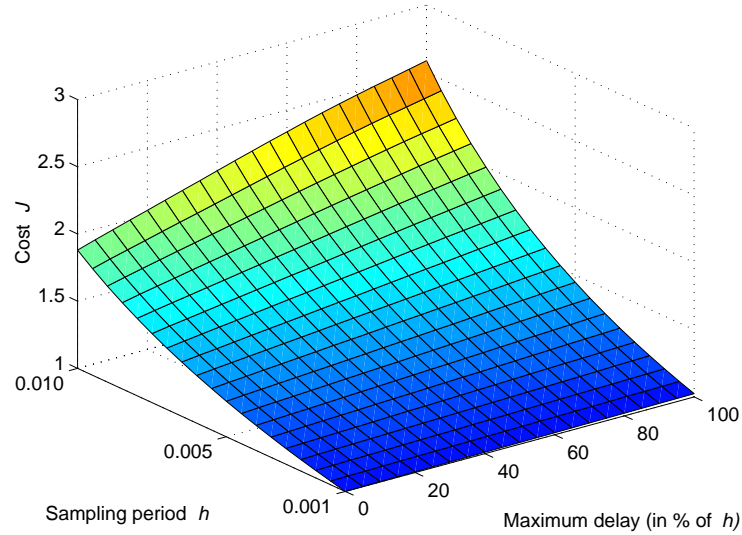
Figure 8 The cost as a function of sampling period and maximum delay with jitter compensation in the distributed control system example.

The Matlab script for the computations is given below:

```
% Jitterbug example: distributed.m
% ================================
% Calculate the performance of a distributed control system with
% delays/jitter

scenario = 1;  % 1 = constant delay, 2 = random delay,
               % 3 = random delay + jitter compensation
s = tf('s');
G = 1000/(s^2+s);  % The process
R1 = 1;            % Input noise
R2 = 0;            % Output noise
Q = diag([1 1]);   % J = E(y^2 + u^2)

% Default PD parameters
K = 1.5;
Td = 0.035;

% Gain(delay)-scheduled PD parameters
tauv = [0 0.0035 0.0045 0.0055 0.0065 0.0075];
Kv = [1.5 1.2 1.1 0.98 0.86 0.78];
Tdv = [0.035 0.04 0.042 0.046 0.049 0.052];

hvec = 0.001:0.0005:0.010;
Jmat = [];
for h = hvec
  dt = h/40;
  taumaxvec = 0:2*dt:h;
  for taumax=taumaxvec
    Ptau = zeros(1,round(h/dt)+1);
    if scenario == 1
      Ptau(round(taumax/2/dt)+1) = 1;   % constant delay
    else
      Ptau(1:round(taumax/2/dt)+1) = 1; % random delay
```

```
        end
        Ptau = Ptau/sum(Ptau);

        H1 = 1;                           % Sampler
        H2 = ss(0,1,K*Td/h,-K*(Td/h+1),-1); % Controller
        H3 = 1;                           % Actuator

        N = initjitterbug(dt,h);          % Initialize Jitterbug

        N = addtimingnode(N,1,Ptau,2);    % Add node 1
        N = addtimingnode(N,2,Ptau,3);    % Add node 2
        N = addtimingnode(N,3);           % Add node 3

        N = addcontsys(N,1,G,4,Q,R1,R2);  % Add sys 1 (G)
        N = adddiscsys(N,2,H1,1,1);       % Add sys 2 (H1) to node 1
        N = adddiscsys(N,3,H2,2,2);       % Add sys 3 (H2) to node 2
        N = adddiscsys(N,4,H3,3,3);       % Add sys 4 (H3) to node 3

        if scenario == 3  % jitter compensation
          for k=1:round(taumax/2/dt)
            tau1 = dt*k;       % known delay
            tau2 = taumax/4;   % predicted remaining delay
            t = tau1 + tau2;
            Kt = interp1(tauv,Kv,t,'linear','extrap');
            Tdt = interp1(tauv,Tdv,t,'linear','extrap');
            H2 = ss(0,1,Kt*Tdt/h,-Kt*(Tdt/h+1),-1);
            N = adddisctimedep(N,3,H2,k);  % Make sys 3 (H2) time-dependent
          end
        end

        N = calcdynamics(N);       % Calculate the internal dynamics
        J = calccost(N)            % Calculate the cost
        Jmat(find(h==hvec),find(taumax==taumaxvec)) = J;
  end
end

Jmat=Jmat/Jmat(1,1);  % scale plot to 1 in (0,0)
figure
surf(0:5:100,hvec,Jmat)
axis([0 100 hvec(1) hvec(end) 1 3])
caxis([0.7 3])
xlabel('Maximum Delay (in % of h)')
ylabel('Sampling Period h')
zlabel('Cost J')
```

4.2 Notch Filter

Cleaning signals from disturbances using e.g. notch filters is important in many applications. In some cases these filters are very sensitive to lost samples due to the very narrow-band characteristics, and in real-time systems lost samples are sometimes inevitable. In this example JITTERBUG is used to evaluate the effects of this problem on different filters.

The setup is as follows. A good signal x (modeled as low-pass filtered noise) is to be cleaned from an additive disturbance e (modeled as band-pass filtered noise). An estimate $\hat{x}$ of the good signal should be found by applying
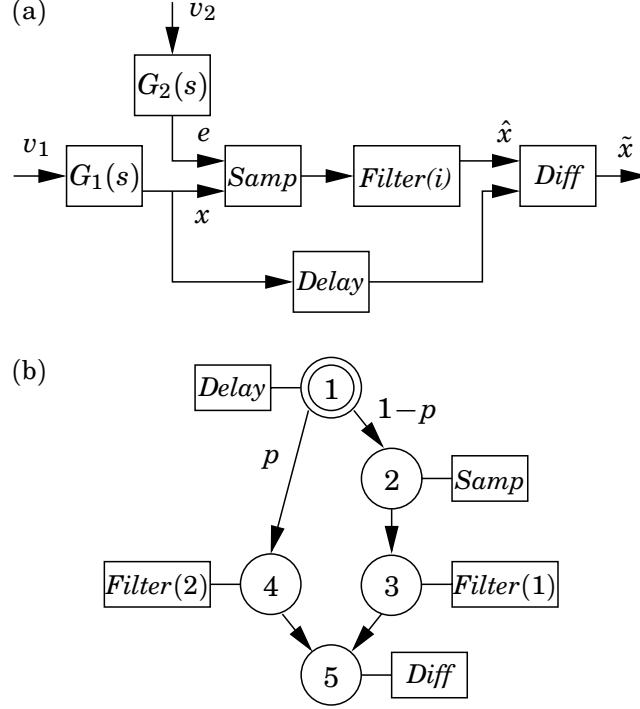
Figure 9 JITTERBUG model of the notch filter: (a) signal model and (b) timing model.

a digital notch filter with the sampling interval $h = 0.1$ to the measured signal $x + e$. Unfortunately, a fraction p of the measurements are lost.

A JITTERBUG model of the system is shown in Figure 9. The signals x and e are generated by white noise being filtered through the continuous-time systems G_1 and G_2. The digital filter is represented as two discrete-time systems: *Samp* and *Filter*. The good signal is buffered in the system *Delay* and is compared to the filtered estimate in the system *Diff*. In the timing model, there is a probability p that the *Samp* system will not be updated. In that case, it is possible to execute an alternate version, *Filter*(2), of the filter dynamics.

Two different filters are compared. The first filter is an ordinary second-order notch filter with two zeros on the unit circle. The same update equations are used regardless if a sample is available or not. The second filter is a second-order Kalman filter based on a simple model of the signal dynamics. In the case of lost samples, only prediction is performed in the filter.

The spectral density of the estimation error $\tilde{x} = x - \hat{x}$ in the two filter cases is shown in Figure 10. It has been assumed that $p = 10\%$ of the samples are lost. It is seen that the ordinary notch filter performs well around the disturbance frequency while the lost samples introduce a large error at lower frequencies. The time-varying Kalman filter is less sensitive towards lost samples and has a more even error spectrum. Overall, the variance of the estimation error is about 40% lower in the Kalman filter case.

14
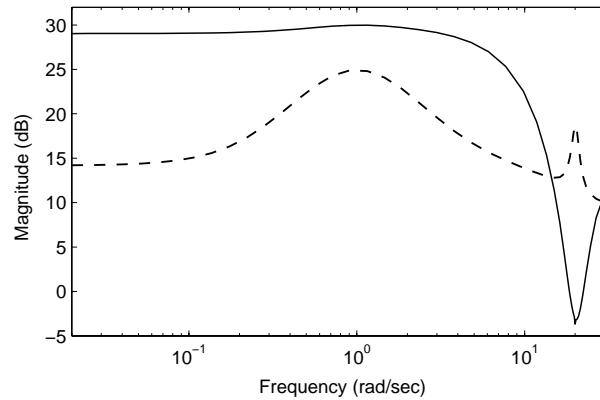
Figure 10 The spectral density of the error output $\tilde{x}$ when 10% of the samples are lost, using a notch filter (full) or a time-varying Kalman filter (dashed).

The Matlab script for the computations is given below:

```
% Jitterbug example: notch.m
% ===========================
% Calculate the performance of a notch filter with lost samples.

scenario = 1; % 1=no filter, 2=notch filter, 3=Kalman filter
p = 0.1;      % Probability of lost sample

s = tf('s');
z = tf('z');

h = 0.1;         % Sampling period

% System generating the good signal
G1 = 100/(s+1)^2;
R1 = 2*pi;      % Input noise variance

% System generating the disturbance
omega = 20;     % Resonance frequency
zeta = 0.001;   % Damping
G2 = 50/(s^2+2*zeta*omega*s+omega^2);
R2 = 2*pi;      % Input noise variance

Samp = [1 1];  % Discrete-time system that samples x + e
Diff = [1 -1]; % Discrete-time system that computes x - xhat

switch scenario,
 case 1,
  % No filter
  Filter1 = 1;
  Filter2 = []; % same dynamics (i.e., none)
  Delay = 1;

 case 2,
  % Zero-phase notch filter
  a = -0.5/cos(omega*h);
  Filter1 = ss(tf([a 1 a],[1 0 0],h));
  Filter1 = Filter1/dcgain(Filter1);
```

15

```
  Filter2 = []; % same dynamics
  Delay = 1/z;  % The notch filter has a delay of one sample

 case 3,
  % Kalman filter based on simple model of G1 (integrator)
  [a1,g1,c1] = ssdata(ss(-0.00001,15,1,0));
  [a2,g2,c2] = ssdata(G2);
  a = blkdiag(a1,a2);
  g = eye(size(a,1));
  c = [c1 c2];
  r1 = blkdiag(g1*g1',g2*g2');
  r2 = 0;
  phi = ssdata(c2d(ss(a,g,c,0),h));
  kf = lqed(a,g,c,r1/h,r2,h);
  k = phi*kf;
  phio = (phi-k*c);
  gammao = k;
  co = [c1 0*c2]*(eye(size(a,1))-kf*c);
  do = [c1 0*c2]*kf;
  Filter1 = ss(phio,gammao,co,do,h);           % Prediction and correction
  Filter2 = ss(phi,zeros(size(a,1),1),[c1 0*c2],0,-1); % Prediction only
  Delay = 1;
end

delta = h;          % Time-grain = sampling interval
Ptau = [1];         % Zero delay between timing nodes
Q = diag([1 0 0]);  % J = xtilde^2

N = initjitterbug(delta,h);          % Initialize Jitterbug

N = addtimingnode(N,1,Ptau,[2 4],[1-p p]);  % Add node 1
N = addtimingnode(N,2,Ptau,3);              % Add node 2
N = addtimingnode(N,3,Ptau,5);              % Add node 3
N = addtimingnode(N,4,Ptau,5);              % Add node 4
N = addtimingnode(N,5);                     % Add node 5

N = addcontsys(N,1,G1,0,[],R1);     % Add sys 1 (G1)
N = addcontsys(N,2,G2,0,[],R2);     % Add sys 2 (G2)
N = adddiscsys(N,3,Samp,[1 2],2);   % Add sys 3 (Samp) to node 2
N = adddiscsys(N,4,Filter1,3,3);    % Add sys 4 (Filter) to node 3
N = adddiscexec(N,4,Filter2,3,4);   % Add execution of sys 4 to node 4
N = adddiscsys(N,5,Delay,1,1);      % Add sys 5 (Delay) to node 1
N = adddiscsys(N,6,Diff,[5 4],5,Q); % Add sys 6 (Diff) to node 5

N = calcdynamics(N);    % Calculate internal dynamics
[J,P,F] = calccost(N);  % Calculate cost and spectral densities
J

figure
bodemag(F{1},F{2},F{4},F{6})  % Plot spectra of outputs 1,2,4,6
axis([0.1 pi/h -20 100]);
legend('Good Signal','Disturbance','Filter Output','Error');
title('Spectral Densities')
```

4.3 Multirate Controller

In this example we will show how to compute the performance of a multi-
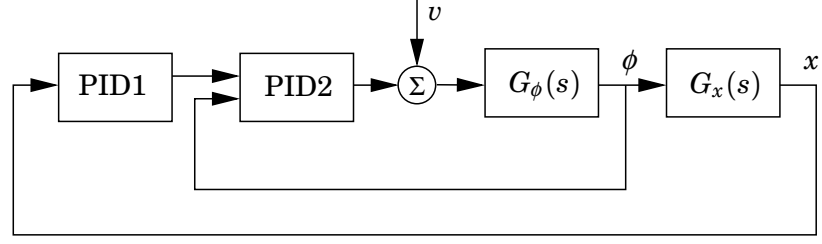
Figure 11   The ball & beam cascaded controller.

rate controller. This is illustrated on a cascade controller for the ball and beam process, see Figure 11. In this control structure, the inner controller, PID2, is responsible for controlling the beam dynamics,

$$G_\phi(s) = \frac{4.4}{s},$$

while the outer controller, PID1, controls the ball on beam dynamics,

$$G_x(s) = -\frac{9.0}{s^2}.$$

Since the inner loop is typically designed to be much faster than the outer loop, it can be a good idea to execute the inner loop at a higher frequency, especially if CPU resources are scarce. We will compare the performance of an ordinary cascade controller with a multirate cascade controller where the inner controller executes at twice the frequency of the outer controller.

The JITTERBUG timing model in the multirate case is shown in Figure 12. The sampling interval of the outer controller is denoted h. The sampling interval of the inner controller is thus $h/2$. The execution time of the control algorithm is ignored in this simple model. At the beginning of each period, PID1 is executed, immediately followed by PID2, which uses the control signal produced by PID1 as a reference value. Then, half a period later, the PID2 is executed again, using the same reference value as in the first invocation but a new measurement value.

Assume that the process is disturbed by white input noise v with unit variance, and that the performance is measured by the cost function

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \left(\phi^2(t) + x^2(t) \right) dt$$
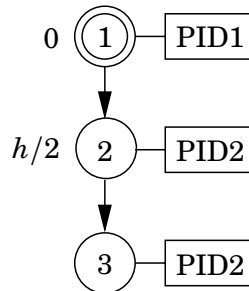
Figure 12   Timing model for the multirate ball & beam controller.

17

Assuming certain PID parameters, the performance in the different cases becomes

$$J_{ordinary} = 3.40, \qquad J_{multirate} = 1.99.$$

(Running both controllers at the fast rate gives $J = 1.93$, i.e. only a small further improvement.)

The Matlab script comparing the two cases is shown below:

```
% Jitterbug example: multirate.m
% ==============================
% Calculate the performance of ordinary/multirate ball & beam controller

s = tf('s');

Gphi = 4.4/s;
Gx = -9.0/s^2;

Q = diag([1 0]);
R1 = 1;
R2 = 0;

h = 0.1;
delta = h/2;

K1 = -0.2;
Ti = 10;
Td = 1;
N = 10;
PID1c = -K1*(1+1/Ti/s+s*Td/(1+s*Td/N));  % PID controller
PID1 = c2d(PID1c,h,'matched');

K2 = 4;
PID2 = K2*[1 -1];                         % P controller

%% Case 1: ordinary cascade controller
N = initjitterbug(delta,h);
N = addtimingnode(N,1,[1],2);        % Add node 1
N = addtimingnode(N,2);              % Add node 2
N = addcontsys(N,1,Gphi,4,Q,R1);    % Add sys 1 (Gphi)
N = addcontsys(N,2,Gx,1,Q);         % Add sys 2 (Gx)
N = adddiscsys(N,3,PID1,2,1);       % Add sys 3 (PID1) to node 1
N = adddiscsys(N,4,PID2,[3 1],2);   % Add sys 4 (PID2) to node 2
N = calcdynamics(N);                % Calculate internal dynamics
J = calccost(N)                     % Calculate cost

%% Case 2: multirate cascade controller
N = initjitterbug(delta,h);
N = addtimingnode(N,1,[1],2);        % Add node 1
N = addtimingnode(N,2,[0 1],3);      % Add node 2
N = addtimingnode(N,3);              % Add node 3
N = addcontsys(N,1,Gphi,4,Q,R1);    % Add sys 1 (Gphi)
N = addcontsys(N,2,Gx,1,Q);         % Add sys 2 (Gx)
N = adddiscsys(N,3,PID1,2,1);       % Add sys 3 (PID1) to node 1
N = adddiscsys(N,4,PID2,[3 1],2);   % Add sys 4 (PID2) to node 2
N = adddiscexec(N,4,[],[3 1],3);    % Add exec of sys 4 (PID2) to node 3
```
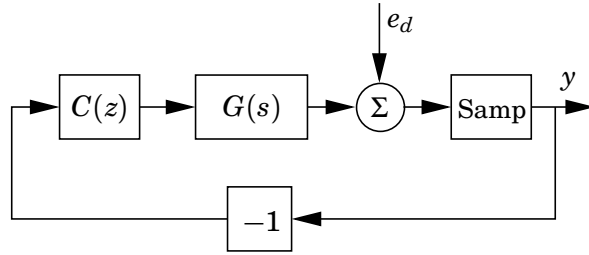
Figure 13 The signal model to calculate the sensitivity spectral density (i.e., the spectral density of y when e_d is white noise).
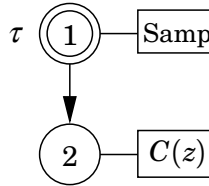
Figure 14 The timing model of the spectral density example.

```
N = calcdynamics(N);             % Calculate internal dynamics
J = calccost(N)                  % Calculate cost
```

4.4 Spectral Density Calculation

The following example computes the influence of jitter on the sensitivity function for a control system. The sensitivity function for a control system with a plant G and a controller C is defined as $S = \frac{1}{1+CG}$. For a randomly time-varying system, though, this definition cannot be used.

The idea in this example is to form a system which is driven by white noise e_d at the output of the process G (see Figure 13). The spectral density of the output y may then be interpreted as a kind of sensitivity function for the stochastic system.

The example system is a continuous $G(s) = \frac{1}{s^2}$ which is controlled by a LQG-designed controller $C(z)$. The process is sampled periodically, but there is a random delay τ between the process and the controller (see Figure 14). The delay is uniformly distributed between zero and $\tau_{\max}$. The sensitivity spectral density for $\tau_{\max}$ between zero and h (for different amounts of jitter) is plotted in Figure 15.

The Matlab script for the computations is given below:

```
% Jitterbug example: spectdens.m
% ==============================
% Compute the sensitivity power spectral density with jitter

s = tf('s');
G = 1/s^2;                % The process is a double integrator

h = 0.25;
delta = h/10;
```

19
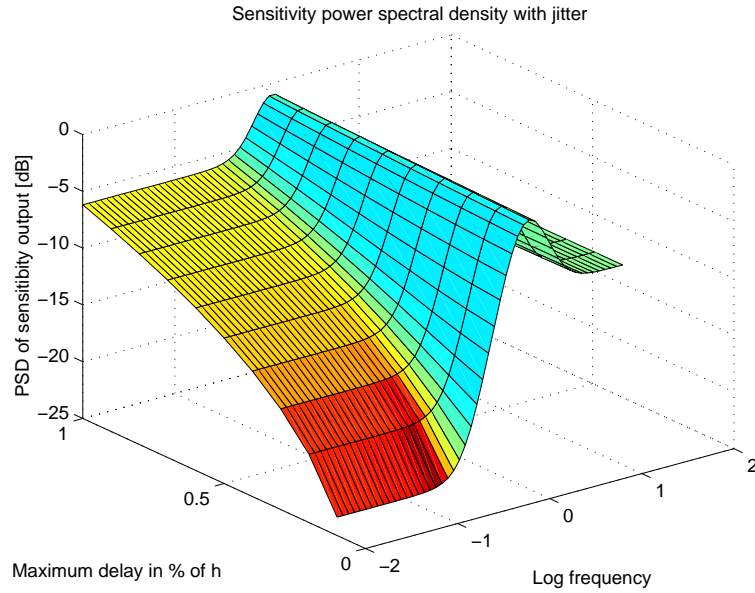
Figure 15 The sensitivity spectral density from the example.

```
Mvec = [];
delays = (1:round(h/delta))/round(h/delta);
for delay = delays
  Ptau1 = ones(1,delay*round(h/delta)+1); % Uniform delay
  Ptau1 = Ptau1/sum(Ptau1);

  Q = diag([1 1]);
  R1 = 1;
  R2 = 1;

  Samp = 1;                               % Sampler system
  C = lqgdesign(G,Q,R1,R2,h,h*delay/2);   % Design an LQG controller

  N = initjitterbug(delta,h);             % Initialize Jitterbug
  N = addtimingnode(N,1,Ptau1,2);         % Add node 1
  N = addtimingnode(N,2);                 % Add node 2
  N = addcontsys(N,1,G,3,Q,[],1);         % Add sys 1 (G) with output noise
  N = adddiscsys(N,2,Samp,1,1);           % Add sys 2 (Samp) to node 1
  N = adddiscsys(N,3,C,2,2);              % Add sys 3 (C) to node 2

  N = calcdynamics(N);                    % Calculate internal dynamics
  [J,P,F] = calccost(N);                  % Calculate spectral densities
  H = F{2};                               % y is the second output (sys 2)
  w = logspace(-2,log10(pi/h),50);
  M = bode(H,w);
  M = squeeze(M);
  Mvec = [Mvec M];
end
figure
surfl(log10(w),delays,10*log10(Mvec)')
title('Sensitivity power spectral density with jitter');
xlabel('Log frequency');
ylabel('Maximum delay in % of \ith');
zlabel('PSD of sensitibity output [dB]')
```

20
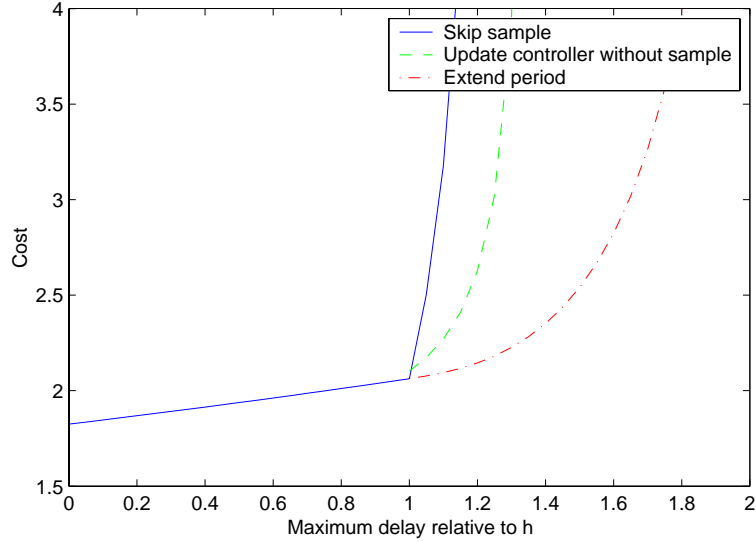
Figure 16 The costs in the overrun example.

4.5 Overrun Handling Methods

This example presentes (a rather long) script that compares three ways of handling long delays in a control system. The problem is what to do if the controller does not get a process sample in time. Three approaches are tested:

a) Do not update the controller, and use the old control signal.

b) Update the controller state and control signal based on no input. This is not done by feeding zeros to the observer, but rather by disconnecting the input part of the observer.

c) Extend the sample period until the sample does arrive. This creates an aperiodic system, and an iterative solver has to be used.

This kind of problem can also be interpreted as a computation time problem, where the computation of some control signal may take long enough time to miss a deadline.

The set-up is as follows. A plant $G(s) = \frac{1}{s(s^2+2\zeta\omega s+\omega^2)}$ with $\zeta = 0.2$ and $\omega = 1$ is to be controlled by an LQG regulator. The controller is calculated for the mean time delay using the function `lqgdesign()`. As for the delay, it is uniformly distributed between 0 and τ_{max}, where τ_{max} is swept from 0 to 2h (i.e. two sample periods).

The three cases are compared in Figure 16. As expected, using the old control signal gives the worst performance, while extending the control period until the control signal is produced gives the best results.

The Matlab script for the computations is given below. Note that the iterative solver is very much slower than the algebraic solver, and is only used when the system is aperiodic.

```
% Jitterbug example: overrun.m
% ==============================
% Compare three overrun handling methods for a control system with
% delayed samples. The plant to be controlled is an integrator with a
% resonance (a third order system). The controller is an LQG
% controller, designed for the mean time delay. The delay for the
% sample from the plant is uniformly distributed between 0 and
% tau_max, which varies between 0 and 2h.
%
% When a sample is delayed more than one period,
% the controller will:
% Case 1) Not be updated at all
% Case 2) Let its observer run without input
% Case 3) Extend the period until the sample arrives (aperiodic system).
%
% The last case is very computationally intensive as it requires an
% iterative solver.

s = tf('s');

zeta = 0.2;
omega = 1;
G = 1/s/(s^2+2*zeta*omega*s+omega^2); % The process
Samp = 1;

h = 0.25;
delta = h/20;

Q = diag([1 1]);
R1 = 1;
R2 = 0.001;

clf;
hold on;
for mode = 1:3
  slots = round(h/delta);
  if mode == 1
    delays = (0:2*slots)/slots;
  else
    delays = (slots:2*slots)/slots;
  end
  Jvec = [];

  for delay = delays
    % All three modes do the same thing for delay < 1.
    if (mode < 2 | delay >= 1)
      Ptau = ones(1,round(delay*slots)+1); % Uniform delay
      Ptau = Ptau/sum(Ptau);
      if (mode == 2)
        if (size(Ptau,2) > slots+1)
          Ptau = [Ptau(1:slots) sum(Ptau(1,slots+1:end))];
        end
      end
      Pwait = zeros(round(slots*delay)+1,slots+1);
      for d = 1:(slots*delay+1)
        if (d > slots+1)
```

```
        Pwait(d,1) = 1;
      else
        Pwait(d,slots-d+2) = 1;
      end
    end

    % Create optimal controller based on mean delay
    [C,L,Obs,K,Kbar,Gd] = lqgdesign(G,Q,R1,R2,h,delay*h/2);
    % Create optimal controller based on observer with no input
    Cnodata = ss(Gd.A-Gd.B*L,Gd.B*0,-L,Gd.D*0,h);

    % Add different timing nodes depending on mode.
    if (mode == 3)
      N = initjitterbug(delta,0);       % Aperiodic system
    else
      N = initjitterbug(delta,h);       % Periodic system
    end
    if (mode == 2)
      N = addtimingnode(N,1,Ptau,[2*ones(1,round(h/delta)) 3]);
    else
      N = addtimingnode(N,1,Ptau,2);
    end
    if (mode == 3)
      N = addtimingnode(N,2,Pwait,1);
    else
      N = addtimingnode(N,2);
    end
    if (mode == 2)
      N = addtimingnode(N,3);
    end

    N = addcontsys(N,1,G,3,Q,R1,R2);    % Add sys 1 (G)
    N = adddiscsys(N,2,Samp,1,1);       % Add sys 2 (Samp) to node 1
    N = adddiscsys(N,3,C,2,2);          % Add sys 3 (C) to node 2
    if (mode == 2)
      N = adddiscexec(N,3,Cnodata,2,3); % Add exec of sys 3 (C) to node 3
    end
    N = calcdynamics(N);                % Calculate internal dynamics
    J = calccost(N)                     % Calculate cost
    Jvec = [Jvec J];
    if J == Inf
      delays = delays(1:find(delays==delay));
      break; % Skip remaining delays
    end
  end
end
if (mode == 1)
  plot(delays,Jvec,'b');
  Jvec1 = Jvec;
elseif (mode == 2)
  plot(delays(find(delays >= 1)),Jvec,'g');
  Jvec2 = Jvec;
else
  plot(delays(find(delays >= 1)),Jvec,'r');
  Jvec3 = Jvec;
end
```

```
   Jvec = [];
   pause(0);
end
hold off;
legend('Skip sample', 'Update controller without sample', 'Extend period');
xlabel('Maximum delay relative to h');
ylabel('Cost');
axis([0 2 1.5 4])
```

5. Command Reference

A summary of the available JITTERBUG commands are given in Table 1.

Command	Description
initjitterbug	Initialize a new JITTERBUG system.
addtimingnode	Add a timing node.
addcontsys	Add a continuous-time system.
adddiscsys	Add a discrete-time system to a timing node.
adddiscexec	Add an execution of a previously defined discrete-time system.
adddisctimedep	Add time-dependence to a previously defined discrete-time system.
calcdynamics	Calculate the internal dynamics of a JITTERBUG system.
calccost	Calculate the total cost of a JITTERBUG system and, for periodic systems, calculate the spectral densities of the outputs.
lqgdesign	Design a discrete-time LQG controller for a continuous-time plant with a constant time delay and a continuous-time cost function.

Table 1   Summary of the JITTERBUG commands.

initjitterbug

Purpose

Initialize a new JITTERBUG system.

Syntax

```
N = initjitterbug(delta,h)
```

Description

Initialize a new JITTERBUG system with a given time-grain and period.

Arguments

delta   The time grain (in seconds). The computations in JITTERBUG
are completely based on this discretization. Computation times
and memory scale inversely proportionally to `delta`.

h   The period of the system (in seconds). Specify 0 if the system
should be aperiodic.

Return Values

N   The JITTERBUG system which must be passed to all other functions.

addtimingnode

Purpose

Add a timing node to a JITTERBUG system.

Syntax

```
N = addtimingnode(N,nodeid)
N = addtimingnode(N,nodeid,Ptau,nextnode)
N = addtimingnode(N,nodeid,Ptau,nextnodes,nextprobs)
N = addtimingnode(N,nodeid,Ptau,timedepnextnodes)
```

Description

Add a timing node to the JITTERBUG system `N`. The delay in the node is given by the discrete probability distribution `Ptau`. The next node to be visited can be either deterministic, random, or dependent on the total delay since the first node.

Note 1: The JITTERBUG system must have a node with ID 1. If the system is periodic, this will be the periodic node.

Note 2: If the total delay exceeds the period, the execution will restart in the periodic node (if the system is periodic).

Arguments

N	The JITTERBUG system to add this timing node to.
nodeid	The ID of this timing node (a positive integer).
Ptau	The delay probability vector. `Ptau(1)` is the probability of a delay of `0*delta` seconds, `Ptau(2)` is the probability of a delay of `1*delta` seconds, etc. If omitted, the system will stay in this node until the next period.
nextnode	The next node to be visisited, after the delay in this node has elapsed.
nextnodes	A vector of possible next nodes to be visited.
nextprobs	A vector specifying the probabilities for each of the nodes in `nextnodes` to be visited next.
timedepnextnodes	A vector of next nodes to be visited depending on the total delay since the first node (including the delay in this node).

addcontsys

Purpose

Add a continuous-time system to a JITTERBUG system.

Syntax

```
N = addcontsys(N,sysid,sys,inputid)
N = addcontsys(N,sysid,sys,inputid,Q,R1,R2)
```

Description

The continuous-time system can be given in state-space form or in transfer-function (or zero-pole-gain) form.

In *state-space form*, the system is described by

$$\dot{x}(t) = Ax(t) + Bu(t) + v(t)$$
$$y^0(t) = Cx(t) \qquad \text{(continuous output)}$$
$$y(t_k) = y^0(t_k) + e(t_k) \quad \text{(measured discrete output)}$$

where v is a continuous-time white-noise process with zero mean and covariance[1] R_1, and e is a discrete-time white-noise process with zero mean and covariance R_2. The cost of the system is specified as

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

In *transfer-function form*, the system is described by

$$y^0(t) = G(p)\big(u(t) + v(t)\big) \quad \text{(continuous output)}$$
$$y(t_k) = y^0(t_k) + e(t_k) \qquad \text{(measured discrete output)}$$

where $G(p)$ is a strictly proper transfer function, v is a continuous-time white-noise process with zero mean and covariance R_1, and e is a discrete-time white-noise process with zero mean and covariance R_2. The cost of the system is specified as

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

Note that the measured discrete output is only used when the system is connected to a discrete-time system.

---

[1]By this we mean that v has the spectral density $\phi(\omega) = \frac{1}{2\pi}R_1$.

Arguments

`N`	The JITTERBUG system to add this continuous-time system to.
`sysid`	A unique ID number for this system (pick any). Used when referred to from other systems.
`sys`	A strictly proper continuous-time LTI system in state-space, transfer function, or pole-zero-gain form. Internally, the system will be converted to state-space form.
`inputid`	A vector of system IDs. The outputs of the corresponding systems will be used as inputs to this system. The number of inputs in this system must equal the total number of outputs in the input systems. A negative input ID specifies that the corresponding system's *state* should be used instead of its output. An input ID of zero specifies that the input should be taken from the *null* system (which has a scalar output equal to zero).

Optional Arguments

`Q`	The cost matrix (default is zero).
`R1`	The state or input noise covariance matrix (default is zero).
`R2`	The measurement noise covariance matrix (default is zero). Note that measurement noise will only be added when the system is sampled by a discrete-time system. The measurement noise *will not* be included in the input cost of the connected discrete-time system. Also, the measurement nosie *will not* affect any connected *continuous-time* systems (see Figure 2).

Any optional arguments can be left as `[]` for default values.

Limitations

To avoid problems with algebraic loops and infinite variances, continuous-time systems with direct terms are not supported. Also, continuous-time output noise is not supported.

adddiscsys

Purpose

Add a discrete-time system to a JITTERBUG system.

Syntax

```
N = adddiscsys(N,sysid,sys,inputid,nodeid)
N = adddiscsys(N,sysid,sys,inputid,nodeid,Q,R)
```

Description

The discrete-time system can be given in state-space form or in transfer-function form.

In *state-space form*, the system is described by

$$
\begin{aligned}
x(t_{k+1}) &= Ax(t_k) + Bu(t_k) + v(t_k) \\
y^0(t_k) &= Cx(t_k) + Du(t_k) \quad \text{(discrete output)} \\
y(t_k) &= y^0(t_k) + e(t_k) \qquad \text{(measured discrete output)}
\end{aligned}
$$

where v and e are discrete-time white-noise processs with zero mean and covariance

$$
R = \mathbf{E} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix}^T.
$$

The cost of the system is specified as

$$
J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ y^0(t) \\ u(t) \end{pmatrix} dt
$$

where Q is a positive semi-definite matrix. Note that $x(t)$ and $y^0(t)$ are piecewise constant signals, while $u(t)$ may be a continuous signal.

In *transfer-function form*, the system is described by

$$
\begin{aligned}
y^0(t_k) &= H(q)\big(u(t_k) + v(t_k)\big) \quad \text{(discrete output)} \\
y(t_k) &= y^0(t_k) + e(t_k) \qquad \text{(measured discrete output)}
\end{aligned}
$$

where $H(q)$ is a proper transfer function, and v and e are discrete-time white-noise processs with zero mean and covariance

$$
R = \mathbf{E} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix} \begin{pmatrix} v(t_k) \\ e(t_k) \end{pmatrix}^T.
$$

The cost of the system is specified as

$$
J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt
$$

where Q is a positive semi-definite matrix. Again, note that $y^0(t)$ is a piecewise constant signal, while $u(t)$ may be a continuous signal.

Arguments

    N   The JITTERBUG system to add this discrete-time system to.

    sysid   A unique ID number of this system (pick any). Used when referred to from other systems.

    sys   A discrete-time LTI system in state-space or transfer function form, or a double/matrix (interpreted as a static gain transfer function). Internally, the system will be converted to state-space form.

    inputid A vector of system IDs. The outputs of the corresponding systems will be used as inputs to this system. The number of inputs in this system must equal the total number of outputs in the input systems. A negative input ID specifies that the corresponding system's *state* should be used instead of its output. An input ID of zero specifies that the input should be taken from the *null* system (which has a scalar output equal to zero).

    nodeid The timing node where this discrete-time system should be executed. If you want the same system to be executed in further nodes, use adddiscexec.

Optional Arguments

    Q   The cost matrix (default is zero).

    R   The noise covariance matrix (default is zero). Added each time the system is updated. Note that noise may also enter the system from the output nose of another system.

Any optional arguments can be left as [] for default values.

Remark

The input cost is really defined on whatever signal is used as input. If the input signal is continuous, the continuous cost (*not* sampled) will be calculated. If you really want the sampled cost, insert a sampling discrete-time system in between.

See Also

adddiscexec, adddisctimedep

adddiscexec

Purpose

Add an execution of a previously defined discrete-time system.

Syntax

```
N = adddiscexec(N,sysid,sys,inputid,nodeid)
```

Arguments

N	The JITTERBUG system.
sysid	The ID of the discrete-time system.
sys	A discrete-time LTI system or [] for the same dynamics as before. To ensure that the same state vector is used internally, both this and the original system should be given in state-space form.
inputid	A vector of system IDs. The outputs of the corresponding systems will be used as inputs to this system. The number of inputs in this system must equal the total number of outputs in the input systems. A negative input ID specifies that the corresponding system's *state* should be used instead of its output. An input ID of zero specifies that the input should be taken from the *null* system (which has a scalar output equal to zero).
nodeid	The ID of the timing node where this discrete-time system should be executed again.

Remark

It is not possible to change the cost or the noise of the system.

See Also

adddiscsys, adddisctimedep

adddisctimedep

Purpose

Add time-dependence to a previously defined discrete-time system.

Syntax

```
N = adddisctimedep(N,sysid,sys,timestep)
```

Description

Makes the dynamics of the discrete-time system with ID `sysid` time-dependent. The system model `sys` will be used for all delays greater than or equal to `timestep*delta` seconds since the first timing node (unless another definition overrides for longer delays).

Arguments

`N`	The JITTERBUG system.
`sysid`	The ID of the discrete-time system.
`sys`	A discrete-time LTI system describing the new dynamics. To ensure that the same state vector is used internally, both this and the original system should be given in state-space form.
`timestep`	The system model `sys` will be used for all delays greater than or equal to `timestep*delta` seconds since the first timing node.

Remark

It is not possible to change the cost or the noise of the system.

See Also

`adddiscexec, adddiscsys`

calcdynamics

Purpose

Calculate the internal dynamics of a JITTERBUG system.

Syntax

```
N = calcdynamics(N)
```

Description

Calculate the total system dynamics for the JITTERBUG system N. The continuous-time noise, the continuous-time cost functions, and the continuous-time systems are sampled with the time grain `delta`. The resulting system description is stored in `N.nodes`.

This function must be called before `calccost`.

Arguments

N   The JITTERBUG system.

See Also

calccost

calccost

Purpose

Calculate stationary variance, cost, and output spectral densities of a JIT-TERBUG system.

Syntax

```
[J,P,F] = calccost(N)
[J,P,F] = calccost(N,options)
```

Description

Calculate the stationary variance and cost of the JITTERBUG system N. For periodic systems, also compute the (discrete-time) spectral densities of all outputs in the periodic node.

If the system is periodic, the solution is calculated algebraically, by solving a linear system of equations. If the system is aperiodic, an iterative solver is used.

This function must be called after `calcdynamics`.

Arguments

N	The JITTERBUG system.
options	For aperiodic systems, `options` is a struct with any of the following fields:

	accuracy	The iterative solver will quit whenever the relative cost change for one time step is less than this. Default is `1e-7`.
	horizon	The horizon over which the cost is averaged. May be Inf. Default is the maximum system period.
	print	Enable printouts. Default is `1` (on).

Return Values

- J The cost (`Inf` if unstable).
- P The stationary variance in the periodic node (`Inf` if unstable).
- F The spectral densities of the outputs (in the order they were defined). The spectral density of each output is returned as a discrete-time system $F(z)$ such that $\phi(\omega) = F(e^{i\omega h})$. Use e.g. `bodemag(F{1})` to plot the spectral density of the output of the first defined system.

See Also

`calcdynamics`

lqgdesign

Purpose

Design a discrete-time LQG controller for a continuous-time plant with a constant time delay and a continuous-time cost function.

Syntax

```
[ctrl,L,obs,K,Kf,sysd] = lqgdesign(sys,Q,R1,R2,h,tau)
```

Description

Design a discrete-time LQG controller with direct term for the continuous-time system sys assuming a constant sampling interval h and a constant time delay tau. The system can be given in state-space form or in transfer-function (or zero-pole-gain) form.

In *state-space form*, the system is described by

$$\dot{x}(t) = Ax(t) + Bu(t - \tau) + v(t)$$
$$y(t_k) = Cx(t_k) + e(t_k)$$

where τ is a constant time delay, v is a continuous-time Gaussian white-noise process with zero mean and covariance R_1, and e is a discrete-time Gaussian white-noise process with zero mean and covariance R_2. The noise processes v and e are assumed to be independent. The sampling instants are given by $t_k = kh$. The cost function to be minimized by the controller is specified as

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} x(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} x(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

In *transfer-function form*, the system is described by

$$y^0(t) = G(p)\big(u(t - \tau) + v(t)\big)$$
$$y(t_k) = y^0(t_k) + e(t_k)$$

where τ is a constant time delay, $G(p)$ is a strictly proper transfer function, v is a continuous-time white-noise process with zero mean and covariance R_1, and e is a discrete-time white-noise process with zero mean and covariance R_2. The cost of the system is specified as

$$J = \lim_{T \to \infty} \frac{1}{T} \int_0^T \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y^0(t) \\ u(t) \end{pmatrix} dt$$

where Q is a positive semi-definite matrix.

The resulting controller has the form

$$u(k) = -L\hat{x}_e(k \mid k)$$
$$\hat{x}_e(k \mid k) = \hat{x}_e(k \mid k - 1) + K_f\big(y(k) - C_e\hat{x}_e(k \mid k - 1)\big)$$
$$\hat{x}_e(k + 1 \mid k) = \Phi_e\hat{x}_e(k \mid k - 1) + \Gamma_e u(k) + K\big(y(k) - C_e\hat{x}_e(k \mid k - 1)\big)$$

where $\hat{x}_e(k) = \begin{pmatrix} \hat{x}(k) \\ u(k-1) \end{pmatrix}$.

Arguments

sys   A strictly proper continuous-time LTI system in state-space, transfer function, or pole-zero-gain form. Any delay specified in this system will be ignored. Use the `tau` argument instead.

Q   The cost matrix.

R1   The state or input noise covariance matrix.

R2   The measurement noise covariance matrix.

h   The sampling period of the controller (in seconds).

tau   The time delay (in seconds), $0 \leq$ `tau` $\leq$ `h`.

Return Values

ctrl   The complete LQG controller as a discrete-time LTI system.

L   The state feedback gain vector.

obs   The observer as a discrete-time LTI system.

K, Kf   The observer gains.

sysd   The sampled delayed plant, `sysd = ss(Phie,Gammae,Ce,0,h)`.

6. References

Lincoln, B. and A. Cervin (2002): "Jitterbug: A tool for analysis of real-time control performance." In *Proceedings of the 41st IEEE Conference on Decision and Control*.

Nilsson, J. (1998): *Real-Time Control Systems with Delays*. PhD thesis ISRN LUTFD2/TFRT--1049--SE, Department of Automatic Control, Lund Institute of Technology, Sweden.

TrueTime v 1.04 – Reference Manual

November 11, 2002

Dan Henriksson, Anton Cervin

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

1. Introduction

This manual describes the use of the MATLAB/Simulink-based simulator TRUETIME, which facilitates co-simulation of controller task execution in real-time kernels, network transmissions, and continuous plant dynamics. The simulator was presented in [Henriksson *et al.*, 2002; Cervin *et al.*, 2002], but several differences from these papers exist.

The manual describes the fundamental steps in the creation of a TRUETIME simulation together with a reference section for all the functions provided by the simulator. The final version of this manual will also contain a number of tutorial examples and a description of the internal workings of the simulator, including the data structures and program flow.

Please direct questions and bug reports to

truetime@control.lth.se

2. Getting started

If not done already, download the compressed files (truetime-1.04.zip) available at:

http://www.control.lth.se/~dan/truetime/

Extract the files to any suitable directory $DIR and start MATLAB. In MATLAB, run the Matlab-script `init_truetime.m` located in the directory $DIR/truetime/kernel:

```
>> cd $DIR/truetime/kernel;
>> init_truetime;
```

This will set up the necessary paths needed to run the simulator.

Issuing the command

```
>> truetime
```

from the MATLAB prompt will then open the TRUETIME block library, see Figure 1.
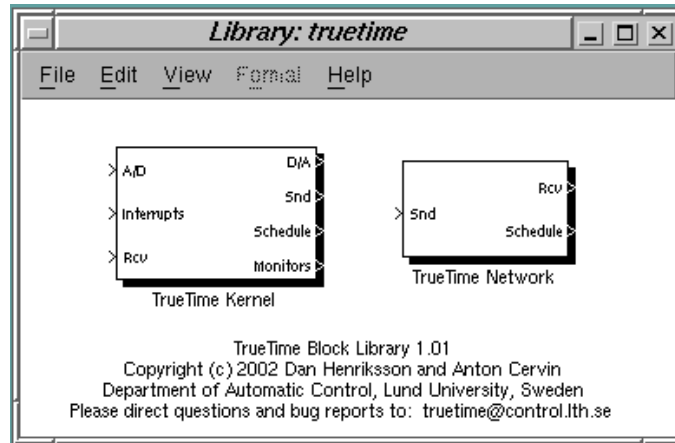
Figure 1 The TRUETIME block library.

3. Using the simulator

The TRUETIME blocks are connected with ordinary Simulink blocks to form a real-time control system, see Figure 2. Before a simulation can be run, however, it is necessary to initialize computer blocks and the network block, and to create tasks, interrupt handlers, timers, events, monitors, etc for the simulation.
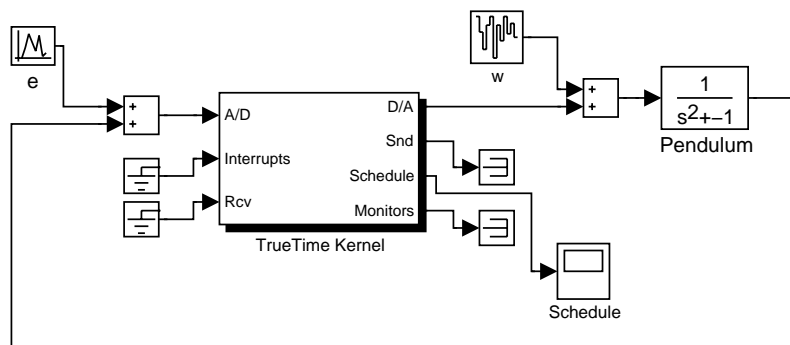
Figure 2 A TRUETIME computer block connected to a continuous pendulum process.

The initialization code as well as the code that is executed during simulation may be written either as C++ code or as MATLAB M-files. The former is faster but the latter is probably more convenient. How the code functions are defined and what must be provided during initialization will be described below. It will also be described how the code is compiled to executable MATLAB code.

4. Writing Code Functions

The execution of tasks and interrupt handlers is defined by code functions. A code function is further divided into code segments according to the execution model in Figure 3. All execution of user code is done in the beginning of each code segment. The execution time of each segment should be returned by the code function.
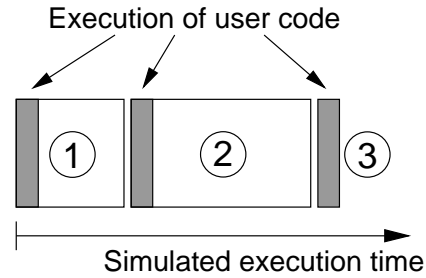
Figure 3 The execution of user-code is modeled by a sequence of segments executed in order by the kernel.

4.1 Writing a Matlab code function

The syntax of a MATLAB code function is given by the following code implementing a simple P-controller:

```
function [exectime, data] = Pcontroller(segment, data)

switch (segment),
  case 1,
    r = ttAnalogIn(1);
    y = ttAnalogIn(2);
    data.u = data.K*(r-y);
    exectime = 0.002;
  case 2,
    ttAnalogOut(1, data.u);
    exectime = -1; % finished
end
```

The variable `segment` indicates which segment that should be executed, while `data` is a user-defined data structure that has been associated with the task during initialization. The data is updated and returned by the code function.

The execution time of the first segment is 2 ms. This means that the delay from input to output for this task will be at least 2 ms. However, preemption from higher priority tasks may cause the delay to be longer. The second segment returns a negative execution time. This is used to indicate end of execution, i.e. that there are no more segments to execute.

`ttAnalogIn` and `ttAnalogOut` are real-time primitives used to read and write signals. A complete reference for all functions provided by the simulator is given at the end of this manual.

Note: The directory `$DIR/truetime/kernel/matlab` contains the MEX-interfaces for all the functions provided by the simulator. These functions must be compiled in order to be called from MATLAB functions, e.g.

```
>> mex ttAnalogIn.cpp
```

Since the compiled MEX-files become rather large, it is recommended to only compile the functions that are used in the simulation.

4.2 Writing a C++ code function

The C++ syntax for the code function in the previous section is given below. We here assume the definition of a structure `Task_Data` that contains the control signal u and the controller gain, K.

```
double Pcontroller(int segment, void* data) {

  Task_Data* d = (Task_Data*) data;

  switch (segment) {
  case 1:
    double r = ttAnalogIn(1);
    double y = ttAnalogIn(2);
    d->u = d->K*(r-y);
    return 0.002;
  case 2:
    ttAnalogOut(1, d->u);
    return FINISHED;
  }
}
```

4.3 Calling Simulink block diagrams

Whether implemented as C++ code or as M-files, it is possible to call Simulink block diagrams from within the code functions. This is a convenient way to implement controllers. Below follows an example where the discrete PID-controller in Figure 4 is used in a code function:

```
function [exectime, data] = PIcontroller(segment, data)

switch (segment),
  case 1,
    inp(1) = ttAnalogIn(1);
    inp(2) = ttAnalogIn(2);
    outp = ttCallBlockSystem(2, inp, 'controller');
    data.u = outp(1);
    exectime = outp(2);
  case 2,
    ttAnalogOut(1, data.u);
    exectime = -1; % finished
end
```

See the function reference at the end of this manual for further explanation of the command `ttCallBlockSystem`.
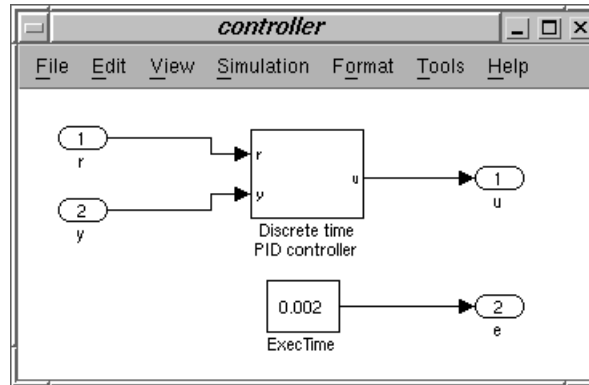
Figure 4 Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The only requirement is that the blocks are discrete with the sample time set to one.

5. Initialization

Initialization involves specifying the number of inputs and outputs of the computer block, defining the scheduling policy, and creating tasks, interrupt handlers, events, monitors, etc for the simulation. This is done in a initialization function for each computer block.

5.1 Writing a Matlab initialization function

The initialization code below shows the minimum of initialization needed to run a simulation. The kernel is initialized by providing the number of inputs and outputs and the scheduling policy using the function `ttInitKernel`. A periodic task is created by the function `ttCreatePeriodicTask`. This task uses the code function `Pcontroller` defined in Section 4.1. See the function reference for further explanation of the functions.

```
function example_init

ttInitKernel(2, 1, 'prioFP');

data.u = 0;
data.K = 2;
ttCreatePeriodicTask('example_task', 0.0, 0.005, 2, 'Pcontroller',
                     data);
```

5.2 Writing a C++ initialization script

An initialization script in C++ must follow a certain format given by the template below:

```
#define S_FUNCTION_NAME filename

#include "$DIR/truetime/kernel/ttkernel.cpp"

// include code functions

void init() {
```

5

```
// perform the initialization
}

void cleanup() {
// free dynamic memory allocated in the script
}
```

filename is the name of the file, e.g. example_init.cpp. The actual initialization is performed in the init()-function, which is called by the kernel at the start of simulation. Any dynamic memory allocated in the script can be deallocated in the cleanup()-function, which is called by the kernel at the end of simulation.

The C++ version of the initialization from the previous section is given below

```
#define S_FUNCTION_NAME example_init

#include "$DIR/truetime/kernel/ttkernel.cpp"

#include "Pcontroller.cpp"

class Task_Data {
public:
  double u;
  double K;
};

Task_Data* data;

void init() {

  ttInitKernel(2, 1, FP);

  data = new Task_Data;
  data->u = 0.0;
  data->K = 2.0;

  ttCreatePeriodicTask("example_task", 0.0, 0.005, 2, Pcontroller,
                       data);
}

void cleanup() {

  delete data;
}
```

6. Compilation

In the MATLAB case, you will have to compile the file `ttkernelMATLAB.cpp` in the directory `$DIR/truetime/kernel`:

```
>> mex ttkernelMATLAB.cpp
```

You will also need to compile the kernel primitives that you use in your code functions, e.g. ttInitKernel and ttAnalogIn. These files are located in `$DIR/truetime/kernel/matlab`. This compilation only has to be done once, and no further compilation is required if code functions or initialization scripts are changed.

However, you may experience that nothing changes in the simulations, although changes are being made to the code functions or the initialization script. If that is the case, type the following at the MATLAB prompt

```
>> 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')
```

In the C++ case the initialization script (`example_init.cpp` in the example above) must be compiled before running a simulation

```
>> mex example_init.cpp
```

This file also needs to be recompiled each time changes are made to the code functions or in the initialization.

In both the MATLAB and the C++ case, the name of the initialization script without extension (e.g. `example_init`) must be provided in the parameter field of the TRUETIME computer block.

See the examples located in the directory `$DIR/truetime/examples` for further reference. Read the README-files in the examples directories for instructions on how to compile the respective simulations.

7. The Network Block

The network block is compiled once and for all (`>> mex ttnetwork.cpp`) and is configured through the block mask dialogue, see Figure 5. Five simple models of network protocols are supported: CSMA/CD (e.g. Ethernet), CSMA/CA (e.g. CAN), Round Robin (e.g. Token Bus), FDMA, and TDMA (e.g. TTP). The propagation delay is ignored, since it is typically very small in a local area network. In all protocols, a long message may be split into a number of *frames* which are transmitted in sequence. Minimum and maximum frame sizes can be specified, and each frame can carry an overhead.

The following network parameters are common to all models:
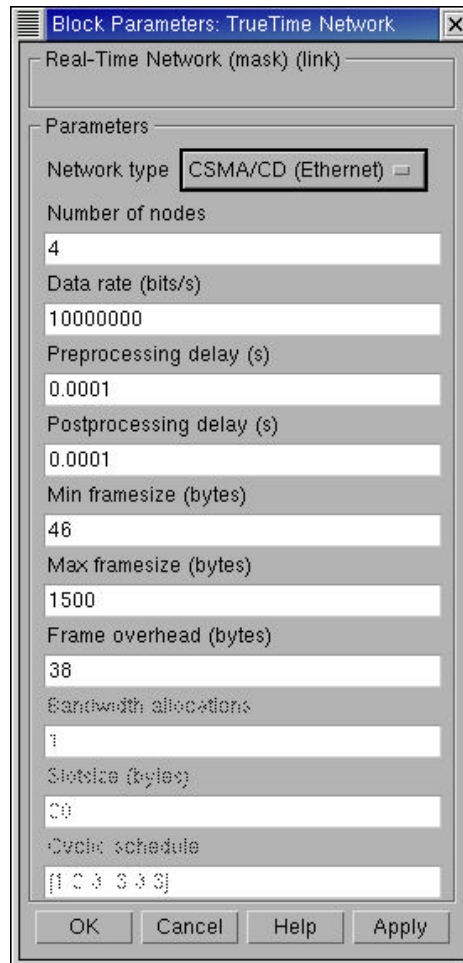
Figure 5 The dialogue of the TrueTime Network block.

Number of Nodes The number of nodes that are connected to the network. This number will determine the size of the `Snd`, `Rcv` and `Schedule` input and outputs of the block.

Data rate (bits/s) The speed of the network.

Proprocessing delay (s) The time a message is delayed by the network interface on the sending end. This can be used to model e.g. a slow serial connection between the computer and the network interface.

Postprocessing delay (s) The time a message is delayed by the network interface on the receiving end.

Minimum frame size (bytes) A message or frame shorter than this will be padded to give the minimum length. Denotes the minimum payload, before the overhead is added. E.g., the minimum Ethernet payload is 46 bytes.

Maximum frame size (bytes) A message longer than this size will be split into several frames. Again, denotes the maximum payload, before the overhead is added. E.g., the maximum Ethernet payload is 1500 bytes. To avoid splitting into frames, specify a very large maximum frame size.

Frame overhead (bytes) Each frame will be extended with this number of bytes before transmission. This number should include any inter-frame gap. E.g.,

the Ethernet frame overhead is 38 bytes.

7.1 CSMA/CD

CSMA/CD stands for Carrier Sense Multiple Access with Collision Detection. If the network is busy, the sender will wait until it occurs to be free. A collision will occur if a message is transmitted within 1 microsecond of another (this corresponds to the propagation delay in a 200 m cable; the actual number is not very important since collisions are only likely to occur when two or more nodes are waiting for the cable to be idle). When a collision occurs, the sender will back off for a time defined by

(minimum frame size + frame overhead) / data rate $\times R$

where $R = \text{rand}(0, 2^K - 1)$ (discrete uniform distribution) and K is the number of collisions in a row (but maximum 10—there is no upper limit on the number of retransmissions, however). Note that for CSMA/CD, both minimum frame size and frame overhead cannot be 0.

After waiting, the node will attempt to retransmit. In an example where two nodes are waiting for a third node to finish its transmission, they will first collide with probability 1, then with probability $1/2$ ($K = 1$), then $1/4$ ($K = 2$), and so on.

7.2 CSMA/CA

CSMA/CD stands for Carrier Sense Multiple Access with Collision Avoidance. If the network is busy, the sender will wait until it occurs to be free. If a collision occurs (again, if two transmissions are being started within 1 microsecond), the message with the highest priority (the lowest priority number) will continue to be transmitted. If two messages with the same priority seek transmission simultaneously, an arbitrary choice is made as to which is transmitted first. (In real CAN applications, all sending nodes have a unique identifier, which serves as the message priority.)

7.3 Round Robin

The nodes in the network take turn (from lowest to highest node number) to transmit one frame each. Between turns, the network is idle for (minimum frame size + frame overhead)/(date rate) seconds, representing the time to pass a token to the next node.

7.4 FDMA

FDMA stands for Frequency Division Multiple Access. The transmissions of the different nodes are completely independent and no collisions can occur. In this mode, there is an extra attribute

Bandwidth allocations a vector of shares for the sender nodes which must sum to at most one.

The actual bit rate of a sender is computed as (allocated bandwidth $\times$ data rate).

7.5 TDMA

TDMA stands for time division multiple access. Works similar to FDMA, except each node has 100 % of the bandwidth but only in its scheduled slots. If a full frame cannot be transmitted in a slot, the transmission will continue in the next scheduled slot, without any extra penalty. Note that overhead is added to each frame just as in the other protocols. The extra attributes are

Slot size (bytes) the slot time is hence given by (slot size)/(data rate).

Schedule A vector of sender node ID's (1 ... nbrOfNodes) specifying a cyclic send schedule.

8. Timing Implementation Details

To support external interrupts and arbitrary time steps, the blocks have a continuous sample time. Discrete (i.e., piecewise constant) outputs are obtained by specifying `FIXED_IN_MINOR_STEP_OFFSET`:

```
static void mdlInitializeSampleTimes(SimStruct *S)
{
  ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, FIXED_IN_MINOR_STEP_OFFSET);
}
```

The timing of the block is implemented using a zero-crossing function. The next time the kernel should wake up is denoted `nextHit`. If there is no known wake-up time, this variable is set to infinity. The basic structure of the zero-crossing function is

```
static void mdlZeroCrossings(SimStruct *S)
{
  store all inputs;
  if (any interrupt input has changed value) {
    nextHit = ssGetT(S);
  }
  ssGetNonsampledZCs(S)[0] = nextHit - ssGetT(S);
}
```

This will ensure that `mdlOutputs` executes every time an internal or external event occur.

Since several kernel and network blocks may be connected in a circular fashion, *direct feedthrough* is not allowed. We exploit the fact that, when an input changes as a step, `mdlOutputs` is called, followed by `mdlZeroCrossings`. Since direct feedthrough is not allowed, the inputs may only be checked for changes in `mdlZeroCrossings`. There, the zero-crossing function is changed so that the next major step occurs at the current time. This scheme will introduce a small timing error ($< 10^{-10}$).

The kernel function is only called from `mdlOutputs` since this is where the outputs (D/A, schedule, network) can be changed.

User Implications For the user, the timing implementation has the following implications:

- *Zero-crossing detection* must be turned on (default, can be changed under *Simulation Parameters/Advanced*).

- The model must contain at least one block with a continuous state. This is usually not a problem, since a model typically contains some continuous process dynamics. Otherwise, inserting for instance a *Clock* block solves the problem.

9. References

Cervin, A., D. Henriksson, B. Lincoln, and K.-E. Årzén (2002): "Jitterbug and Truetime: Analysis tools for real-time control systems." In *Proceedings of the 2nd Workshop on Real-time Tools*. Copenhagen, Denmark.

Henriksson, D., A. Cervin, and K.-E. Årzén (2002): "Truetime: Simulation of control loops under shared computer resources." In *Proceedings of the 15th IFAC World Congress on Automatic Control*. Barcelona, Spain.

ttInitKernel

Purpose

Initialize the TRUETIME kernel.

Matlab syntax

```
ttInitKernel(nbrInp, nbrOutp, prioFcn)
ttInitKernel(nbrInp, nbrOutp, prioFcn, csoh)
```

C++ syntax

```
void ttInitKernel(int nbrInp, int nbrOutp, int prioFcn)
void ttInitKernel(int nbrInp, int nbrOutp, int prioFcn, double csoh)
```

Arguments

nbrInp	Number of input channels, i.e. the size of the A/D port of the computer block.
nbrOutp	Number of output channels, i.e. the size of the D/A port of the computer block.
prioFcn	The scheduling policy used by the kernel.
csoh	Time for a full context switch. Unless specified, zero overhead will be associated with context switches.

Description

This function performs necessary initializations of the computer block and *must* be called first of all in the initialization script. The priority function should be any of the following in the MATLAB case; 'prioFP', 'prioRM', 'prioDM', or 'prioEDF'. The corresponding identifiers in the C++ case are; FP, RM, DM, and EDF. To define an arbitrary priority function, see ttAttachPrioFcn.

See Also

ttAttachPrioFcn

ttCreateTask

Purpose

Create a TRUETIME task.

Matlab syntax

```
ttCreateTask(name, deadline, priority, codeFcn)
ttCreateTask(name, deadline, priority, codeFcn, data)
```

C++ syntax

```
bool ttCreateTask(char* name, double deadline, double priority,
                  double (*codeFcn)(int, void*))
bool ttCreateTask(char *name, double deadline, double priority,
                  double (*codeFcn)(int, void*), void* data)
```

Arguments

name	Name of the task. Must be a unique, non-empty string.
deadline	Deadline of the task.
priority	Priority of the task.
codeFcn	The code function of the task, where codeFcn is a string (name of an m-file) in the MATLAB version and a function pointer in the C++ case.
data	An arbitrary data structure representing the local memory of the task.

Description

This function is used to create a task to run in the TRUETIME kernel. The function returns true if successful and false otherwise. Note that no task instance (job) is created by this function. This is done by the primitive ttCreateJob.

See Also

ttCreatePeriodicTask, ttCreateJob, ttSetX

ttCreatePeriodicTask

Purpose

Create a periodic TRUETIME task.

Matlab syntax

```
ttCreatePeriodicTask(name, release, period, priority, codeFcn)
ttCreatePeriodicTask(name, release, period, priority, codeFcn, data)
```

C++ syntax

```
bool ttCreatePeriodicTask(char* name, double release, double period,
              double priority, double (*codeFcn)(int, void*))
bool ttCreatePeriodicTask(char *name, double release, double period,
        double priority, double (*codeFcn)(int, void*), void* data)
```

Arguments

name	Name of the task. Must be a unique, non-empty string.
release	Release of the first instance of the periodic task.
period	Period of the task.
priority	Priority of the task.
codeFcn	The code function of the task, where codeFcn is a string (name of an m-file) in the MATLAB version and a function pointer in the C++ case.
data	An arbitrary data structure representing the local memory of the task.

Description

This function is used to create a periodic task to run in the TRUETIME kernel. The function returns true if successful and false otherwise. The periodicity is implemented by a periodic timer, generating task instances. The deadline and worst-case execution time of the task are by default put equal to the task period. This may be changed by a suitable set-function.

See Also

ttCreateTask, ttSetX

ttCreateInterruptHandler

Purpose

Create a TRUETIME interrupt handler.

Matlab syntax

```
ttCreateInterruptHandler(name, priority, codeFcn)
ttCreateInterruptHandler(name, priority, codeFcn, data)
```

C++ syntax

```
bool ttCreateInterruptHandler(char *name, double priority,
                              double (*codeFcn)(int, void*))
bool ttCreateInterruptHandler(char *name, double priority,
                    double (*codeFcn)(int, void*), void* data)
```

Arguments

name
: Name of the handler. Must be a unique, non-empty string.

priority
: Priority of the handler.

codeFcn
: The code function of the handler, where codeFcn is a string (name of an m-file) in the MATLAB version and a function pointer in the C++ case.

data
: An arbitrary data structure representing the local memory of the handler.

Description

This function is used to create an interrupt handler to run in the TRUETIME kernel. The function returns `true` if successful and `false` otherwise. Interrupt handlers may be associated with external interrupts, timers, or attached to tasks as overrun handlers.

See Also

```
ttCreateTimer, ttCreateExternalTrigger, ttAttachDLHandler,
ttAttachWCETHandler
```

ttCreateMonitor

Purpose

Create a TRUETIME monitor.

Matlab syntax

```
ttCreateMonitor(name, display)
```

C++ syntax

```
bool ttCreateMonitor(char *name, bool display)
```

Arguments

name   Name of the monitor. Must be a unique, non-empty string.

display   To indicate if the monitor should be included in the monitor graph generated by the simulation.

Description

This function is used to create a monitor in the TRUETIME kernel. The function returns true if successful and false otherwise.

See Also

ttEnterMonitor, ttExitMonitor

ttCreateEvent

Purpose

Create a TRUETIME event.

Matlab syntax

```
ttCreateEvent(eventname, monitorname)
```

C++ syntax

```
bool ttCreateEvent(char *eventname, char *monitorname)
```

Arguments

eventname   Name of the event. Must be a unique, non-empty string.

monitorname Name of an already created monitor to which the event is to be associated. monitor.

Description

This function is used to create an event in the TRUETIME kernel. The function returns `true` if successful and `false` otherwise. To create a free event, i.e. an event not associated with a monitor, put the monitorname to `FREE_EVENT` in C++ mode or an empty string in MATLAB mode.

See Also

ttWait, ttNotifyAll

ttCreateExternalTrigger

Purpose

Associate a TRUETIME interrupt handler with an external interrupt channel.

Matlab syntax

```
ttCreateExternalTrigger(handlername, latency)
```

C++ syntax

```
bool ttCreateExternalTrigger(char *nameOfHandler, double latency)
```

Arguments

handlername	Name of the handler to be associated with the external interrupt.
latency	The time interval during which the interrupt is insensitive to new invocations.

Description

This function is used to associate an interrupt handler with an external interrupt channel. The function returns `true` if successful and `false` otherwise. The size of the external interrupt port will be decided depending on the number of created triggers.

See Also

```
ttCreateInterruptHandler
```

ttNoSchedule

Purpose

Switch off the schedule generation for a specific task or interrupt handler.

Matlab syntax

```
ttNoSchedule(name)
```

C++ syntax

```
void ttNoSchedule(char* name)
```

Arguments

name   Name of a task or interrupt handler.

Description

This function is used to switch off the schedule generation for a specific task or interrupt handler. The schedule is generated by default and this function must be called to turn it off.

ttNonPreemptive

Purpose

To make a task non-preemptive.

Matlab syntax

```
ttNonPreemptive(taskname)
```

C++ syntax

```
void ttNonPreemptive(char* taskname)
```

Arguments

taskname   Name of a task.

Description

Tasks are by default preemptive. Use this function to make certain tasks non-preemptive.

ttAttachDLHandler

Purpose

Attach a deadline overrun handler to a task.

Matlab syntax

```
ttAttachDLHandler(taskname, handlername)
```

C++ syntax

```
void ttAttachDLHandler(char* taskname, char* handlername)
```

Arguments

taskname	Name of a task.
handlername	Name of an interrupt handler.

Description

This function is used to attach a deadline overrun handler to a task. The interrupt handler is activated if the task executes past its deadline.

See Also

`ttAttachWCETHandler, ttSetDeadline`

ttAttachWCETHandler

Purpose

Attach a worst-case execution time overrun handler to a task.

Matlab syntax

```
ttAttachWCETHandler(taskname, handlername)
```

C++ syntax

```
void ttAttachWCETHandler(char* taskname, char* handlername)
```

Arguments

taskname	Name of a task.
handlername	Name of an interrupt handler.

Description

This function is used to attach a worst-case execution time overrun handler to a task. The interrupt handler is activated if the task executes longer than its associated worst-case execution time.

See Also

ttAttachDLHandler, ttSetWCET

ttAttachPrioFcn (C++ only)

Purpose

Attach an arbitrary priority function to be used by the TRUETIME kernel.

C++ syntax

```
void ttAttachPrioFcn(double (*prioFcn)(Task*))
```

Arguments

prioFcn   The priority function to be attached.

Description

This function is used to attach an arbitrary priority function to the TRUETIME kernel. The input to the priority function is a pointer to a Task structure, see $DIR/truetime/kernel/task.h for the definition. The output from the priority function should be a number that gives the (possibly dynamic) priority of the task. As an example, the simple priority function implementing fixed-priority scheduling is given below:

```
double prioFP(Task* task) {

  return task->priority;
}
```

ttAttachHook (C++ only)

Purpose

Attach a run-time hook to a task.

C++ syntax

```
void ttAttachHook(char* taskname, int ID, void (*hook)(Task*))
```

Arguments

taskname	Name of a task.
ID	An identifier telling when the hook should be called during simulation. Possible values are RELEASE, START, SUSPEND, RESUME, and FINISH.
hook	The hook to be attached.

Description

This function is used to attach a run-time hook to a specific task. When the hook will be called is determined by the identifier ID. It is possible to attach hooks that are called when the task is released, when the task starts to execute, when the task is suspended, when the task resumes after being suspended, and when the task finishes execution.

The input to the hook is a pointer to the Task structure of the specific task, see $DIR/truetime/kernel/task.h for the definition.

ttAnalogIn

Purpose

Read a value from an analog input channel.

Matlab syntax

```
ttAnalogIn(inpChan)
```

C++ syntax

```
double ttAnalogIn(int inpChan)
```

Arguments

inpChan   The input channel to read from.

Description

This function is used to read an analog input from the environment. The input channel must be between 1 and the number of input channels specified in `ttInitKernel`.

See Also

ttAnalogOut

ttAnalogOut

Purpose

Write a value to an analog output channel.

Matlab syntax

```
ttAnalogOut(outpChan, value)
```

C++ syntax

```
ttAnalogOut(int outpChan, double value)
```

Arguments

outpChan   The output channel to write to.

value   The value to write.

Description

This function is used to write an analog output to the environment. The output channel must be between 1 and the number of output channels specified in `ttInitKernel`.

See Also

`ttAnalogIn`

ttCreateJob

Purpose

Create a job of a specific task.

Matlab syntax

```
ttCreateJob(taskname, release)
```

C++ syntax

```
void ttCreateJob(char *taskname, double release)
```

Arguments

taskname   Name of a task.

release   Release time of the job.

Description

This function is used to create job instances of tasks. If there already exist pending jobs for the task, the job is queued and served as soon as possible. This function must be called to activate non-periodic tasks, i.e. tasks created using ttCreateTask.

See Also

ttCreateTask, ttKillJob

ttKillJob

Purpose

Kill the running job of a specific task.

Matlab syntax

```
ttKillJob(taskname)
```

C++ syntax

```
void ttKillJob(char *taskname)
```

Arguments

taskname   Name of a task.

Description

This function is used to kill the running job instance of a task. If there exist queued jobs for the task, the first job in the queue will be scheduled for execution.

See Also

```
ttCreateJob
```

ttCreateTimer

Purpose

Create a one-shot timer and associate an interrupt handler with the timer.

Matlab syntax

```
ttCreateTimer(time, timername, handlername)
```

C++ syntax

```
bool ttCreateTimer(double time, char *timername, char *handlername)
```

Arguments

time	The time when the timer is set to expire.
timername	Name of the timer. Must be unique, non-empty string.
handlername	Name of interrupt handler associated with the timer.

Description

This function is used to create a one-shot timer. When the timer expires the associated interrupt handler is activated and scheduled for execution. The function returns `true` if successful and `false` otherwise.

See Also

`ttCreateInterruptHandler`, `ttCreatePeriodicTimer`, `ttRemoveTimer`

ttCreatePeriodicTimer

Purpose

Create a periodic timer and associate an interrupt handler with the timer.

Matlab syntax

```
ttCreatePeriodicTimer(start, period, timername, handlername)
```

C++ syntax

```
bool ttCreatePeriodicTimer(double start, double period,
                           char *timername, char *handlername)
```

Arguments

start	The time for the first expiry of the timer.
period	The period of the timer.
timername	Name of the timer. Must be unique, non-empty string.
handlername	Name of interrupt handler associated with the timer.

Description

This function is used to create a periodic timer. Each time the timer expires the associated interrupt handler is activated and scheduled for execution. The function returns `true` if successful and `false` otherwise.

See Also

`ttCreateInterruptHandler, ttCreateTimer, ttRemoveTimer`

ttRemoveTimer

Purpose

Remove a specific timer.

Matlab syntax

```
ttRemoveTimer(timername)
```

C++ syntax

```
bool ttRemoveTimer(char *timername)
```

Arguments

timername   Name of the timer to be removed.

Description

This function is used to remove timers. Both one-shot and periodic timers can be removed by this function. The function returns `true` if successful and `false` otherwise.

See Also

ttCreateTimer, ttCreatePeriodicTimer

ttCurrentTime

Purpose

Get the current time.

Matlab syntax

```
ttCurrentTime
```

C++ syntax

```
double ttCurrentTime(void)
```

Description

This function returns the current time in the simulation, in seconds.

ttSleepUntil

Purpose

Put a task to sleep until a certain point in time.

Matlab syntax

```
ttSleepUntil(time)
ttSleepUntil(time, taskname)
```

C++ syntax

```
void ttSleepUntil(double time)
void ttSleepUntil(double time, char *taskname)
```

Arguments

time   The time when the task should wake up.

taskname   Name of a task.

Description

This function is used to make a task sleep until a specified point in time. If the argument taskname is not specified, the call will affect the currently running task.

See Also

ttSleep

ttSleep

Purpose

Put a task to sleep for a certain time.

Matlab syntax

```
ttSleep(duration)
ttSleep(duration, taskname)
```

C++ syntax

```
void ttSleep(double duration)
void ttSleep(double duration, char *taskname)
```

Arguments

duration   The time that the task should sleep.

taskname   Name of a task.

Description

This function is used to make a task sleep for a specified amount of time. If the argument `taskname` is not specified, the call will affect the currently running task. This function is equivalent to `ttSleepUntil(duration + ttCurrentTime())`.

See Also

`ttSleepUntil`

ttSetNextSegment

Purpose

Set the next segment to be executed.

Matlab syntax

```
ttSetNextSegment(segment)
```

C++ syntax

```
void ttSetNextSegment(int segment)
```

Arguments

segment   Number of the segment.

Description

This function is used to set the next segment to be executed, overriding the normal execution order. This can be used to implement conditional branching and loops (see `ttWait` below). The segment number should be between 1 and the number of segments defined in the code function.

ttInvokingTask

Purpose

Get the name of the task that invoked an interrupt handler.

Matlab syntax

```
ttInvokingTask
```

C++ syntax

```
char *ttInvokingTask(void)
```

Description

This function returns the name of the task that has invoked an interrupt handler. Used in interrupt handlers associated with task overruns (deadline, WCET) to determine which task that caused the interrupt. In the cases when the interrupt was generated externally or by the expiry of a timer, this function returns an empty string.

See Also

```
ttAttachDLHandler, ttAttachWCETHandler
```

ttEnterMonitor

Purpose

Attempt to enter a monitor.

Matlab syntax

```
ttEnterMonitor(monitorname)
```

C++ syntax

```
bool ttEnterMonitor(char *monitorname)
```

Arguments

monitorname Name of a monitor.

Description

This function is used to attempt to enter a monitor. If the attempt fails, the task will be removed from the ready queue and be inserted in the waiting queue of the monitor on a FIFO basis. When the task currently holding the monitor exits, the first task in the waiting queue will be moved to the ready queue now holding the monitor. Execution will then resume in the segment after the call to `ttEnterMonitor`. The function returns `true` if successful, `false` otherwise.

Example:

```
function [exectime, data] = ctrl(seg, data)

switch seg,

 case 1,
  ttEnterMonitor('mutex');
  exectime = 0.0;
 case 2,
  criticalOperation();
  exectime = 0.001;
 case 3,
  ttExitMonitor('mutex');
  exectime = -1;
end
```

See Also

ttCreateMonitor, ttExitMonitor

ttExitMonitor

Purpose

Exit a monitor.

Matlab syntax

```
ttExitMonitor(monitorname)
```

C++ syntax

```
void ttExitMonitor(char *monitorname)
```

Arguments

monitorname   Name of a monitor.

Description

This function is used to exit a monitor. The function can only be called by the task currently holding the monitor. The call will cause the first task in the waiting queue of the monitor to be moved to the ready queue.

See Also

ttCreateMonitor, ttEnterMonitor

ttWait

Purpose

Wait for an event.

Matlab syntax

```
ttWait(eventname)
```

C++ syntax

```
void ttWait(char *eventname)
```

Arguments

eventname   Name of an event.

Description

This function is used to wait for an event. If the event is associated with a monitor, the call must be performed inside a `ttEnterMonitor-ttExitMonitor` construct. The call will cause the task to be moved from the ready queue to the waiting queue of the event. When the task is later notified, it will be moved to the waiting queue of the associated monitor, or to the ready queue if it is a free event.

Example of an event-driven code function:

```
function [exectime, data] = ctrl(seg, data)

switch seg,

 case 1,
  ttWait('Event1');
  exectime = 0.0;
 case 2,
  performCalculations;
  exectime = 0.001;
 case 3,
  ttSetNextSegment(1); % loop
  exectime = 0.0;
end
```

See Also

ttCreateEvent, ttNotifyAll

ttNotifyAll

Purpose

Notify all tasks waiting for an event.

Matlab syntax

```
ttNotifyAll(eventname)
```

C++ syntax

```
void ttNotifyAll(char *eventname)
```

Arguments

eventname   Name of an event.

Description

This function is used to notify all tasks waiting for an event. If the event is associated with a monitor, the call must be performed inside a `ttEnterMonitor-ttExitMonitor` construct. The call will cause all tasks waiting for the event to be moved to the waiting queue of the associated monitor, or to the ready queue if it is a free event.

See Also

`ttCreateEvent, ttWait`

ttCallBlockSystem

Purpose

Call a Simulink block diagram from within a code function.

Matlab syntax

```
outp = ttCallBlockSystem(nbroutp, inp, blockname)
```

C++ syntax

```
bool ttCallBlockSystem(int nbroutp, double *outp, int nbrinp,
                       double *inp, char *blockname)
```
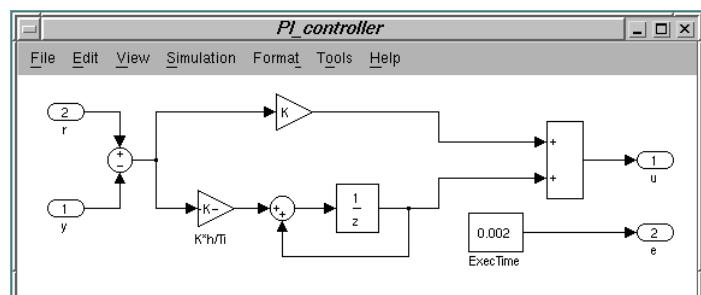
Arguments

nbrinp	Number of inputs to the block diagram.
nbroutp	Number of outputs from the block diagram.
inp	Vector of input values.
outp	Vector of output values.
blockname	The name of the Simulink block diagram.

Description

This function is used to call a Simulink block diagram from within a code function. The states of the block diagram are stored in the kernel between calls. *The block diagrams may only contain discrete blocks and the sampling times should be set to one*. The C++ function returns `true` if successful, and `false` otherwise. The MATLAB function returns zero if unsuccessful. The inputs and outputs are defined by Simulink inports and outports, see the figure below. Here follows an example using this Simulink diagram:

```
function [exectime, data] = PIcontroller(segment, data)

switch (segment),
  case 1,
    inp(1) = ttAnalogIn(1);
    inp(2) = ttAnalogIn(2);
    outp = ttCallBlockSystem(2, inp, 'PI_controller');
    data.u = outp(1);
    exectime = outp(2);
  case 2,
    ttAnalogOut(1, data.u);
    exectime = -1;
end
```

ttSetX

Purpose

Set a task attribute.

Matlab syntax

```
ttSetX(value)
ttSetX(value, taskname)
```

C++ syntax

```
void ttSetX(double value)
void ttSetX(double value, char *taskname)
```

Arguments

`value`	Value to be set.
`taskname`	Name of a task.

Description

These functions are used to manipulate task attributes. There exist functions for the following attributes (with the true function name in parenthesis):

- release (`ttSetRelease`)
- deadline (`ttSetDeadline`)
- absolute deadline (`ttSetAbsDeadline`)
- priority (`ttSetPriority`)
- period (`ttSetPeriod`)
- execution time budget (`ttSetBudget`)
- worst-case execution time (`ttSetWCET`)

All these functions except `ttSetRelease` exist in overloaded versions as shown by the syntax above. If the argument `taskname` is not specified, the call will affect the currently running task. `ttSetRelease` can only be called for tasks in the time queue, i.e. not by the currently running task.

Depending on the scheduling policy used, a call to `ttSetX` may lead to a context switch or a re-ordering of the ready queue. For example, if rate-monotonic scheduling ('prioRM') is used, a call to `ttSetPeriod` will change the priority of the affected task.

A call to `ttSetBudget` will only have effect if there is a worst-case execution time overrun handler attached to the task.

Use `ttSetX` functions to set other attributes than the default attributes set by `ttCreateTask` and `ttCreatePeriodicTask`.

See Also

`ttCreateTask, ttCreatePeriodicTask, ttGetX`

ttGetX

Purpose

Get a task attribute.

Matlab syntax

```
value = ttGetX
value = ttGetX(taskname)
```

C++ syntax

```
double ttGetX(void)
double ttGetX(char *taskname)
```

Arguments

taskname   Name of a task.

Description

These functions are used to retrieve values of task attributes. There exist functions for the following attributes (with the true function name in parenthesis):

- release (`ttGetRelease`)
- deadline (`ttGetDeadline`)
- absolute deadline (`ttGetAbsDeadline`)
- priority (`ttGetPriority`)
- period (`ttGetPeriod`)
- execution time budget (`ttGetBudget`)
- worst-case execution time (`ttGetWCET`)

All these functions exist in overloaded versions as shown by the syntax above. If the argument taskname is not specified, the call will affect the currently running task.

To be able to retrieve a dynamic attribute (release, absolute deadline, and budget) it is required that there exist a job instance of the task.

See Also

ttSetX

ttInitNetwork

Purpose

Initialize the network interface.

Matlab syntax

```
ttInitNetwork(nodenumber, handlername)
```

C++ syntax

```
void ttInitNetwork(int nodenumber, char *handlername)
```

Arguments

nodenumber   The address of the node in the network. Must be a number between 1 and the number of nodes as specified in dialogue in the TrueTime Network block.

handlername   The name of an interrupt handler that should be called when a message arrives over the network.

Description

The network interface must be initialized using this command before any messages can be sent or received. The initialization will fail if there is not exactly one TrueTime Network block in the Simulink model.

See Also

```
ttSendMsg, ttGetMsg
```

ttSendMsg

Purpose

Send a message over the network.

Matlab syntax

```
ttSendMsg(receiver, data, length)
ttSendMsg(receiver, data, length, priority)
```

C++ syntax

```
void ttSendMsg(int receiver, void *data, int length)
void ttSendMsg(int receiver, void *data, int length, int priority)
```

Arguments

receiver	The number of the receiving node (a number between 1 and the number of nodes). It is allowed to send messages to oneself.
data	An arbitrary data structure representing the contents of the message
length	The length of the message, in bytes. Determines the time it will take to transmit the message.
priority	The priority of the message (relevant only for CSMA/CA networks). If not specified, the priority will be given by the number of the sending node, i.e., messages sent from node 1 will have the highest priority by default.

Description

The network interface must have been initialized using `ttInitNetwork` before any messages can be sent.

See Also

`ttInitNetwork, ttGetMsg`

ttGetMsg

Purpose

Get a message that has been received over the network.

Matlab syntax

```
ttGetMsg
```

C++ syntax

```
void *ttGetMsg()
```

Description

This function is used to retrieve a message that has been received over the network. Typically, you have been notified that a message exists in the network interface input queue by an interrupt, but it is also possible to poll for new messages. If no message exists, the function will return NULL (C++) or an empty struct (Matlab).

The network interface must have been initialized using `ttInitNetwork` before any messages can be received.

C++ example of an event-driven receiver:

```cpp
/* Thread that waits for and reads messages */
double receiver_thread(int seg, void *data) {
  MyMsgType *msg;
  switch (seg) {
  case 1:
    ttWait("message");
    return 0.0;
  case 2:
    // Get all messages (may be more than one!)
    while ((msg = (MyMsgType *)ttGetMsg()) != NULL) {
      printf("I got a message!\n");
      delete msg; // don't forget to free memory
    }
    ttSetNextSegment(1); // loop
    return 0.0;
  }
}

/* Interrupt handler that is called by the network interface */
double msgRcvhandler(int seg, void *data)
{
  ttNotifyAll("message");
  return FINISHED;
}
```

See Also

```
ttInitNetwork, ttSendMsg
```