LUND UNIVERSITY

TrueTime 1.13 - Reference Manual

Henriksson, Dan; Cervin, Anton

2003

Link to publication

Citation for published version (APA):
Henriksson, D., & Cervin, A. (2003). TrueTime 1.13 - Reference Manual.

Total number of authors:
2

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

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

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

LUND UNIVERSITY

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--7605--SE

TRUETIME 1.13—Reference Manual

Dan Henriksson
Anton Cervin

Department of Automatic Control
Lund Institute of Technology
October 2003

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
INTERNAL REPORT

Date of issue
October 2003

Document Number
ISRN LUTFD2/TFRT--7605--SE

Author(s)
Dan Henriksson, Anton Cervin

Supervisor

Sponsoring organisation

Title and subtitle
TrueTime 1.13—Reference Manual

Abstract

The manual describes the use of TrueTime, a Matlab/Simulink-based tool for simulation of distributed real-time control systems. The tool facilitates detailed co-simulation of plant dynamics, controller task execution, and network transmissions. The TrueTime Kernel and TrueTime Network blocks are described, and the real-time kernel primitives are detailed.

Key words
Real-time control systems, Event-based simulation, Shared resources, Real-time kernel, Feedback scheduling, Networked control systems.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
73

Security classification

Recipient's notes

The report may be ordered from the Department of Automatic Control or borrowed through:
University Library 2, Box 3, SE-221 00 Lund, Sweden
Fax +46 46 222 44 22 E-mail ub2@ub2.lu.se

Contents

1. Introduction . . . 7

2. Getting Started . . . 7

3. Using the Simulator . . . 8

4. Writing Code Functions . . . 9

4.1 Writing a MATLAB Code Function . . . 9

4.2 Writing a C++ Code Function . . . 10

4.3 Calling Simulink Block Diagrams . . . 10

5. Initialization . . . 11

5.1 Writing a MATLAB Initialization Script . . . 11

5.2 Writing a C++ Initialization Script . . . 11

6. Compilation . . . 12

6.1 The MATLAB Case . . . 13

6.2 The C++ Case . . . 13

6.3 The Kernel Block Parameter . . . 13

7. The TrueTime Network . . . 14

7.1 CSMA/CD (Ethernet) . . . 14

7.2 CSMA/AMP (CAN) . . . 15

7.3 Round Robin (Token Bus) . . . 15

7.4 FDMA . . . 16

7.5 TDMA (TTP) . . . 16

7.6 Switched Ethernet . . . 16

7.7 Compiling the Network Block . . . 17

8. Examples . . . 17

8.1 Process and Controller . . . 17

8.2 Real-time Control of the DC-servo . . . 17

8.3 Distributed Control of the DC-servo . . . 19

9. Implementing Higher Level Network Protocols . . . 21

9.1 Opening a TCP Connection . . . 21

9.2 Sending a TCP Data Packet . . . 21

9.3 Receiving a TCP Data Segment . . . 22

9.4 Communicating with the Application Layer . . . 23

10. Implementation Details 25
10.1 Task Model 25
10.2 The Kernel Function 26
10.3 Timing 28
11. TrueTime Command Reference 29
ttInitKernel 32
ttCreatePeriodicTask 33
ttCreateTask 34
ttCreateJob 35
ttKillJob 36
ttCreateInterruptHandler 37
ttCreateExternalTrigger 38
ttNoSchedule 39
ttNonPreemptable 40
ttAttachDLHandler 41
ttAttachWCETHandler 42
ttAttachPrioFcn (C++ only) 43
ttAttachHook (C++ only) 44
ttCreateMonitor 45
ttEnterMonitor 46
ttExitMonitor 47
ttCreateEvent 48
ttWait 49
ttNotifyAll 50
ttCreateMailbox 51
ttTryFetch 52
ttTryPost 53
ttCreateTimer 54
ttCreatePeriodicTimer 55
ttRemoveTimer 56
ttCurrentTime 57
ttSleepUntil 58
ttSleep 59

ttAnalogIn 60
ttAnalogOut 61
ttSetNextSegment 62
ttInvokingTask 63
ttCallBlockSystem 64
ttSetX 65
ttGetX 67
ttInitNetwork 69
ttSendMsg 70
ttGetMsg 71
12. References 73

1. Introduction

This manual describes the use of the MATLAB/Simulink-based [The Mathworks, 2000] simulator TRUETIME, which facilitates co-simulation of controller task execution in real-time kernels, network transmissions, and continuous plant dynamics. The simulator is presented in [Cervin *et al.*, 2003; Henriksson *et al.*, 2002], but several differences from these papers exist.

The manual describes the fundamental steps in the creation of a TRUETIME simulation. That include how to write the code that is executed during simulation, how to configure the kernel and network blocks, and what compilation that must be done to get an executable simulation. The code functions for the tasks and the initialization commands may be written either as C++ functions or as M-files, and both cases are described.

Two tutorial examples are provided, both treating PID-control of a DC-servo. In the first example the DC-servo is controlled by a controller task implemented in a TRUETIME kernel block. This example is also extended to the case of three PID-tasks running concurrently on the same CPU controlling three different servo systems. The second example simulates distributed control of the DC-servo, with the sensor, controller, and actuator represented as three different nodes communicating over a network.

The manual also includes a section describing some of the internal workings of TRUETIME, including the task model, implementation details, and timing details. A TRUETIME command reference with detailed explanations of all functionality provided by the simulator is given at the end of the manual.

For questions and bug reports, please direct these issues to

truetime@control.lth.se

2. Getting Started

Download the compressed files available at:

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

Note that TRUETIME currently supports both MATLAB 6.1 (R12.1) with Simulink 4.1 and MATLAB 6.5 (R13) with Simulink 5.0. Later releases, however, may not support MATLAB 6.1.

Extract the files to any suitable directory `$DIR`. Before starting MATLAB, you must set the environment variable `TTKERNEL` to point to the directory with the TRUETIME kernel files. This is typically done in the following manner:

- Unix/Linux: `export TTKERNEL=$DIR/kernel`
- Windows: use Control Panel / System / Advanced / Environment Variables

After starting MATLAB, you must also add the directories `$DIR/kernel` and `$DIR/kernel/matlab` to your MATLAB path. You could for instance do this in your MATLAB startup script as follows:

```
addpath(getenv('TTKERNEL'))
addpath([getenv('TTKERNEL') '/matlab'])
```

Issuing the command

```
>> truetime
```

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

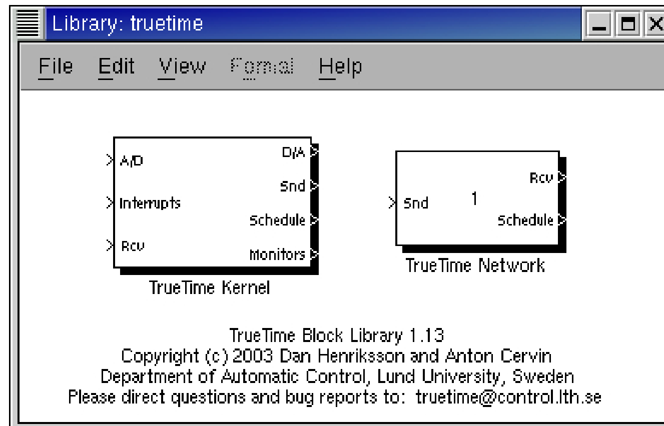


Figure 1 The TRUETIME block library.

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.

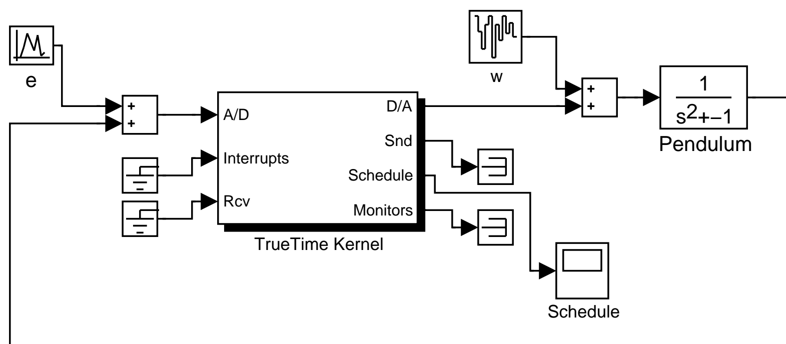


Figure 2 A TRUETIME computer block connected to a continuous pendulum process.

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.

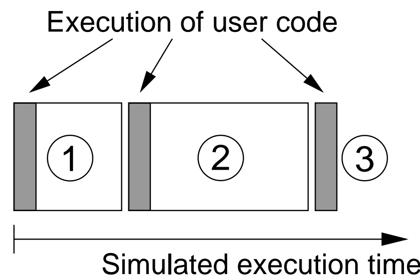


Figure 3 The execution of user-code is modeled by a sequence of segments executed in order by the kernel.

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` determines which segment that should be executed, and `data` is a user-defined data structure that has been associated with the task when it was created, see `ttCreateTask` and `ttCreatePeriodicTask` in the command reference. The data is updated and returned by the code function. The code function also returns the execution time of the executed segment.

In this example, 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 to the environment. Detailed descriptions of these functions can be found in the command reference at the end of this manual.

Note: The directory `$DIR/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

Writing a code function in C++ follows a similar pattern as the code function described above. The C++ syntax for the simple P-controller code function in the previous section is given below. We here assume 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; // end of execution
  }
}
```

4.3 Calling Simulink Block Diagrams

Whether implemented in 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 PI-controller in Figure 4 is used in a code function:

```
function [exectime, data] = PIcode(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; % finished
end
```

See the command reference at the end of this manual for further explanation of the command `ttCallBlockSystem`.

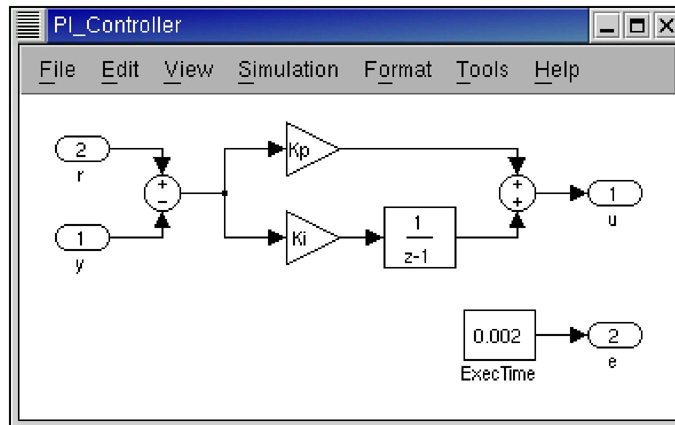


Figure 4 Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The only requirement is that the blocks are discrete with the sample time set to one.

5. Initialization

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

5.1 Writing a Matlab Initialization Script

The initialization code below shows the minimum of initialization needed for a TRUETIME 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 command reference for further explanation of the functions.

```
function example_init
ttInitKernel(2, 1, 'prioFP');
data.u = 0;
data.K = 2;
ttCreatePeriodicTask('ctrl', 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 "ttkernel.cpp"

// insert your code functions here

void init() {
// perform the initialization
}
```

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

The file `ttkernel.cpp` contains the Simulink call-back functions meaning that the initialization script is actually a complete MATLAB S-function. `filename` should be the name of the source file, e.g. if the source file is called `example_init.cpp`, `S_FUNCTION_NAME` should be defined to `example_init`.

The `init()`-function is called at the start of simulation (from the Simulink call-back function `mdlInitializeSizes`), and it is here all initialization should be performed. Any dynamic memory allocated from the `init()`-function can be deallocated from the `cleanup()`-function, which is called at the end of simulation (from `mdlTerminate`).

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

```
#define S_FUNCTION_NAME example_init
#include "ttkernel.cpp"

#include "Pcontroller.cpp"  // P controller code funtion

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

Task_Data* data; // pointer to local memory for the task

void init() {
  ttInitKernel(2, 1, FP);
  data = new Task_Data;
  data->u = 0.0;
  data->K = 2.0;
  ttCreatePeriodicTask("ctrl", 0.0, 0.005, 2, Pcontroller, data);
}

void cleanup() {
  delete data;
}
```

6. Compilation

Depending on whether the code functions and the initialization script are written in C++ code or as M-files, different amounts of compilation must be performed before running a simulation.

In the C++ case, the initialization script itself is compiled, producing a MATLAB MEX-file for the simulation. In the MATLAB case, a kernel function (`ttkernelMATLAB-`

.cpp) is compiled once and for all to a MATLAB MEX-file. This S-function then calls the initialization script (M-file) at the start of simulation.

The following compilers are supported (it may, however, also work using other compilers):

- Visual Studio C++ 6.0 under Windows
- gcc, g++ - GNU project C and C++ Compiler (gcc-2.96) for LINUX and UNIX

6.1 The Matlab Case

Compile the file `ttkernelMATLAB.cpp` in the directory `$DIR/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 the directory `$DIR/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')
```

6.2 The C++ Case

In the C++ case the initialization script (`example_init.cpp` in the example from the previous section) itself should be compiled

```
>> ttmex example_init.cpp
```

Note: The `ttmex` command is the same as the ordinary `mex` command but includes the path to the kernel files automatically.

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

6.3 The Kernel Block Parameter

The TRUETIME kernel block takes one parameter. This parameter is the name of the initialization script without extension. I.e., in the example in the previous section, this parameter should be `example_init`, both in the MATLAB and C++ case.

7. The TrueTime Network

The TrueTime Network block simulates medium access and packet transmission in a local area network. Six simple models of networks are supported: CSMA/CD (e.g. Ethernet), CSMA/AMP (e.g. CAN), Round Robin (e.g. Token Bus), FDMA, TDMA (e.g. TTP), and Switched Ethernet. The propagation delay is ignored, since it is typically very small in a local area network. Only packet-level simulation is supported—it is assumed that higher protocol levels in the kernel nodes have divided long messages into packets, etc.

The network block is configured through the block mask dialog, see Figure 5. The following network parameters are common to all models:

Network Number The number of the network block. The networks must be numbered from 1 and upwards.

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.

Pre-processing 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.

Post-processing 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 frame size, including any overhead introduced by the protocol. E.g., the minimum Ethernet frame size, including a 14-byte header and a 4-byte CRC, is 64 bytes.

Loss Probability (0–1) The probability that a network message is lost during transmission. Lost messages will consume network bandwidth, but will never arrive at the destination.

7.1 CSMA/CD (Ethernet)

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

$$t_{backoff} = \text{minimum frame size} \ / \ \text{data rate} \times R$$

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, minimum frame size 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

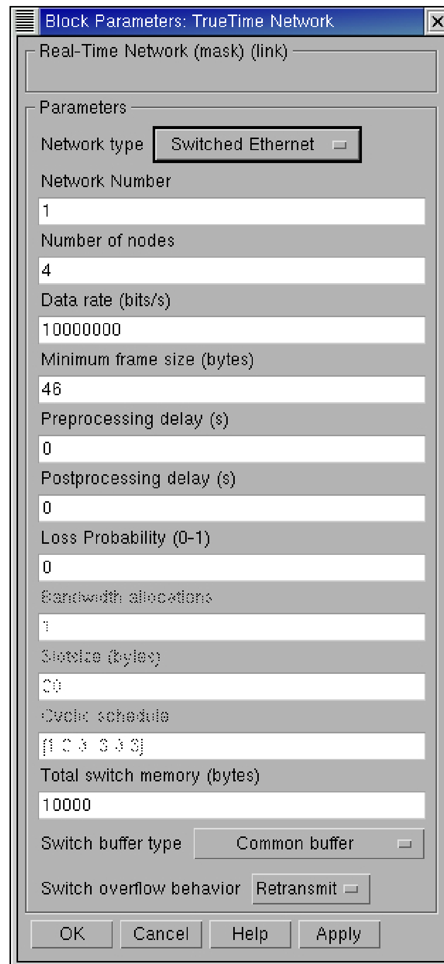


Figure 5 The dialog of the TrueTime Network block.

with probability 1, then with probability 1/2 ($K = 1$), then 1/4 ($K = 2$), and so on.

7.2 CSMA/AMP (CAN)

CSMA/AMP stands for Carrier Sense Multiple Access with Arbitration on Message Priority. 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 (Token Bus)

The nodes in the network take turns (from lowest to highest node number) to transmit one frame each. Between turns, the network is idle for a time

$$t_{idle} = \text{minimum frame size / date rate},$$

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 (TTP)

TDMA stands for Time Division Multiple Access. Works similar to FDMA, except that 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 size of a sending slot. The slot time is hence given by

$$t_{slot} = \text{slot size} \ / \ \text{data rate}.$$

Schedule A vector of sender node ID's (1 ... nbrOfNodes) specifying a cyclic send schedule. A zero is also an allowed node ID, meaning that no-one is allowed to transmit in that time slot.

7.6 Switched Ethernet

In Switched Ethernet, each node in the network has its own, full-duplex connection to a central switch. Compared to an ordinary Ethernet, there will never be any collisions on the network segments in a Switched Ethernet. The switch stores the received messages in a buffer and then forwards them to the correct destination nodes. This common scheme is known as *store and forward*.

If many messages in the switch are destined for the same node, they are transmitted in FIFO order. There can be either one queue that holds all the messages in the switch, or one queue for each output segment. In case of heavy traffic and long message queues, the switch may run out of memory. The following options are associated with the Switched Ethernet:

Total switch memory (bytes) This is the total amount of memory available for storing messages in the switch. An amount of memory equal to the length of the message is allocated when the message has been fully received in the switch. The same memory is deallocated when the complete message has reached its final destination node.

Switch buffer type This setting describes how the memory is allocated in the switch. *Common buffer* means that all messages are stored in a single FIFO queue and share the same memory area. *Symmetric output buffers* means that the memory is divided into n equal parts, one for each output segment connected to the switch. When one output queue runs out of memory, no more messages can be stored in that particular queue.

Switch overflow behavior This options describes what happens when the switch has run out of memory. When the complete message has been received in the switch, it is deleted. *Retransmit* means that the switch then informs the sending node that it should try to retransmit the message. *Drop* means that no notification is given—the message is simply deleted.

7.7 Compiling the Network Block

The S-function implementing the network block is located in the directory `$DIR/truetime/kernel`. This file is compiled once and for all with the command

```
>> mex ttnetwork.cpp
```

8. Examples

The directory `$DIR/examples` contains two examples of PID-control of a DC-servo, with the second example treating the distributed case. The descriptions below will only treat the MATLAB case. For detailed instructions on how to compile the examples in the C++ case, see the README-files in the two example directories.

8.1 Process and Controller

The DC-servo is described by the continuous-time transfer function

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

The PID-controller is implemented according to the following equations

$$\begin{aligned} P(k) &= K \cdot (r(k) - y(k)) \\ I(k+1) &= I(k) + \frac{Kh}{T_i}(r(k) - y(k)) \\ D(k) &= a_d D(k-1) + b_d(y(k-1) - y(k)) \\ u(k) &= P(k) + I(k) + D(k) \end{aligned} \qquad (1)$$

where $a_d = \frac{T_d}{Nh+T_d}$ and $b_d = \frac{NKT_d}{Nh+T_d}$. The controller parameters were chosen to give the system a closed-loop bandwidth, $\omega_c = 20$ rad/s, and a relative damping, $\zeta = 0.7$.

8.2 Real-time Control of the DC-servo

The first example considers simple PID control of the DC-servo process, and is intended to give a basic introduction to the TRUETIME simulation environment. The process is controlled by a controller task implemented in a TRUETIME kernel block. Two versions of the code function are provided, one standard PID implementation and one that calls a Simulink block diagram to calculate the control signal in each sample. The example is also extended to the case of three PID-tasks running concurrently on the same CPU controlling three different servo systems. The files are found in the directory `$DIR/examples/simple_pid/matlab`.

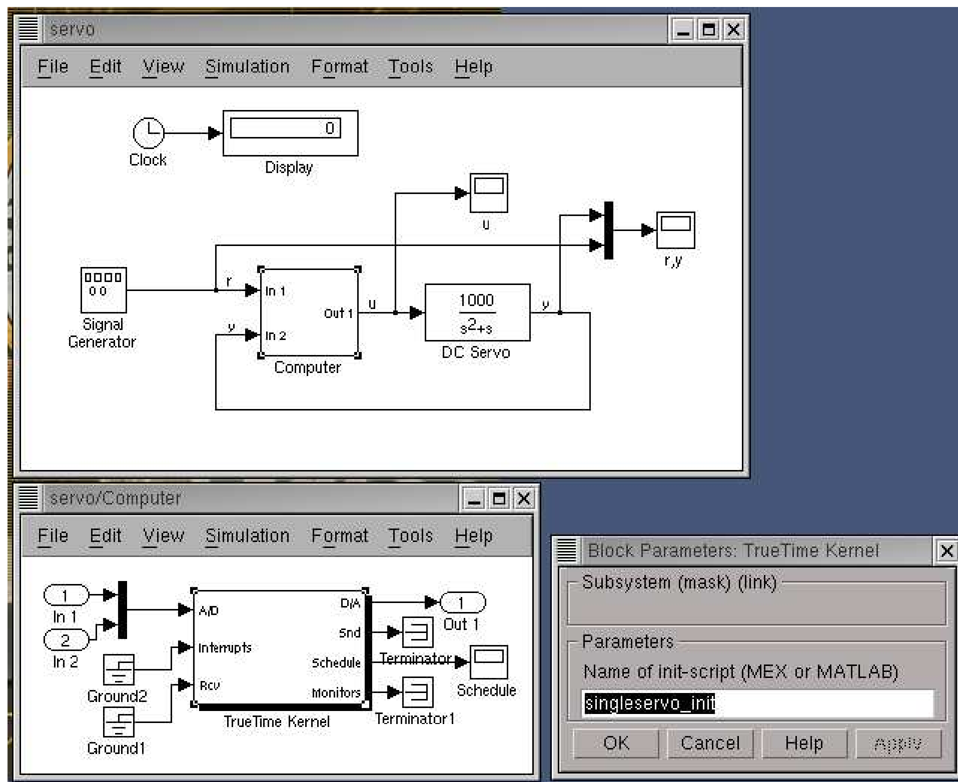


Figure 6 The TRUETIME model of the DC-servo system.

Code Function The MATLAB code function (`pidcode.m`) for the controller task is given below

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

switch seg,
 case 1,
  r = ttAnalogIn(data.rChan);
  y = ttAnalogIn(data.yChan);
  data = pidcalc(data, r, y);
  exectime = 0.002;
 case 2,
  ttAnalogOut(data.uChan, data.u);
  exectime = -1;
end
```

where the function `pidcalc.m` implements the controller (1).

Initialization Script The simulation model (`servo.mdl`) is given in Figure 6, and the corresponding initialization script (`singleservo_init.m`) looks like this:

```
function singleservo_init

ttInitKernel(2, 1, 'prioFP'); % nbrOfInputs, nbrOfOutputs, FP
```

```
data.K = 0.96;
data.Ti = 0.12;
data.Td = 0.049;
data.N = 10;
data.h = 0.006;
data.u = 0;
data.Iold = 0;
data.Dold = 0;
data.yold = 0;
data.rChan = 1;
data.yChan = 2;
data.uChan = 1;

ttCreatePeriodicTask('pid_task', 0.0, 0.006, 2, 'pidcode', data);
```

Experiments with a Single PID Task Run the M-file `makepid.m` to compile the files necessary for the simulation. Then open the model `servo.mdl` to run the single PID task simulation. Try the following

- Run a simulation and verify that the controller behaves as expected. Notice the computational delay of 2 ms in the control signal. Compare with the code function. Study the schedule plot (high=running, medium=ready, low=idle).
- Try changing the execution time of the first segment of the code function, to simulate the effect of different input-output delays.
- Try changing the sampling period and study the resulting control performance.
- A PID-controller is implemented in the Simulink block `controller.mdl`. Study the code function `blockpid.m` and the initialization script `block_init.m`. Change the name of the init-script in the parameter field of the kernel block to `block_init`. Now the code function will use the PID-controller block to compute the control signal in each sample. Run a simulation.

Experiments with Three PID Tasks Open the model `threeservos.mdl` to run the simulation of three concurrent PID tasks. Try the following

- Make sure that rate-monotonic scheduling is specified by the function `ttInitKernel` in the initialization script (`threeservos_init.m`) and simulate the system. Study the computer schedule and the control performance. Task 1 will miss all its deadlines and the corresponding control loop is unstable.
- Change the scheduling policy to earliest-deadline-first and run a new simulation. Again study the computer schedule and the control performance. After an initial transient all tasks will miss their deadlines, but the overall control performance, however, is satisfactory.

8.3 Distributed Control of the DC-servo

This example simulates distributed control of the DC-servo. The example contains four computer nodes, each represented by a TrueTime kernel block. A

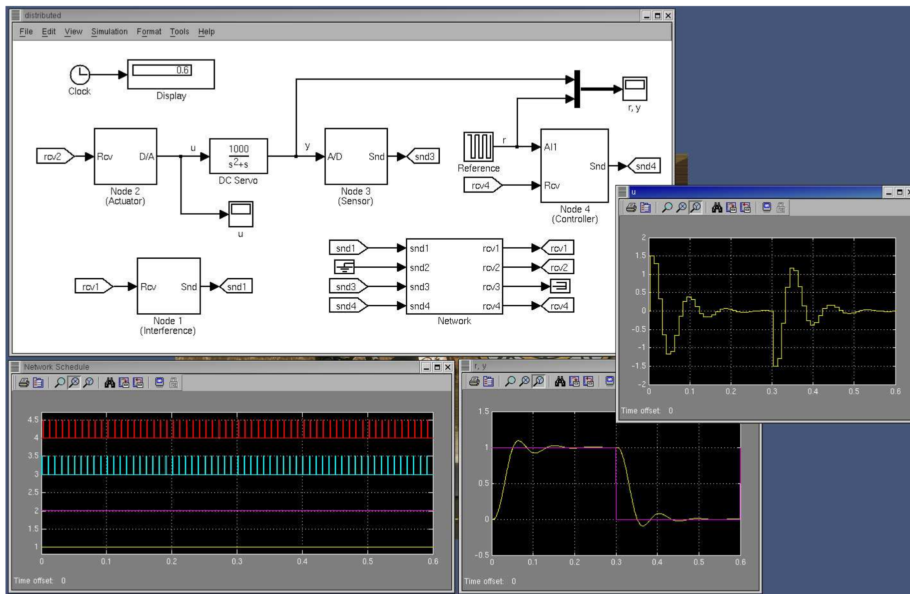


Figure 7 The TRUETIME model of the distributed control system.

time-driven sensor node samples the process periodically and sends the samples over the network to the controller node. The control task in this node calculates the control signal and sends the result to the actuator node, where it is subsequently actuated. The simulation also involves an interfering node sending disturbing traffic over the network, and a disturbing high-priority task executing in the controller node. The simulation model is shown in Figure 7. The files are found in the directory `$DIR/examples/distributed/matlab`.

Experiments Compile the MATLAB simulation by running the M-file `makedist.m`. Then open the model `distributed.mdl` to run the simulation. Try the following

- Study the initialization scripts and code functions for the different nodes. The event-driven nodes contain interrupt handlers, which are activated as messages arrive over the network. The handler then notifies the corresponding task that a message has arrived.
- Run a first simulation without disturbing traffic and without interference in the controller node. This is obtained by setting the variable `BWshare` in the code function of the interfering node (`interfcode.m`) to zero, and by commenting out the creation of the task `'dummy'` in `controller_init`. In this case we will get a constant round-trip delay and satisfactory control performance. Study the network schedule (high=sending, medium=waiting, low=idle) and the resulting control performance.
- Switch on the disturbing node and the interfering task in the controller node. Set the variable `BWshare` to the percentage of the network bandwidth to be used by the disturbing node. Again study the network schedule and the resulting control performance. Experiment with different network protocols and different scheduling policies in the controller node.

9. Implementing Higher Level Network Protocols

The TrueTime network block simulates the basic properties of standard MAC (media access control) layer protocols. These protocols constitute the link layer in the Internet protocol stack, and are typically implemented in a network interface card, see [Kurose and Ross, 2001].

It is, however, straight-forward to also implement higher level protocols using TrueTime. Transport layer protocols, such as TCP and UDP, are usually implemented in software in the end systems, and may be emulated directly in the various TrueTime computer nodes using dedicated tasks and interrupt handlers.

A simple TCP implementation will be outlined below. In the simulation it is possible to specify TCP specific parameters such as sizes of the buffers at the receiving and sending ends, corresponding sending and receiving windows, maximum segment size (MSS), and acknowledgment time-outs. Flow control is supported by the use of receive windows. The window gives an indication of the free buffer space at the receiving side, and dictates how much data that can be transmitted on that specific connection. The window size is constantly updated by the receiving node, as messages are being read from the application layer. This information is sent back to the sender with each acknowledgment. No congestion control is implemented.

9.1 Opening a TCP Connection

Since TCP is connection-oriented, a socket connection must be established before two nodes can start sending and receiving messages. When a connection is set up, sending and receiving buffers are created at each end of the connection. Special TrueTime sending and receiving tasks are also associated with each connection. Using tasks for the processing of incoming and outgoing TCP packets, it is possible to simulate overhead in the TCP layer. The functionality performed by these tasks will be described below.

9.2 Sending a TCP Data Packet

When sending a message over TCP it is divided in segments of size MSS which are sent in sequence to the receiving end, where the message is recreated. In addition to the data, each TCP data segment includes a header containing fields for source and destination identifiers, sequence number, acknowledgment number, and window size. When a segment is transmitted, a timer is created. If no acknowledgment has been received at the expiry of the timer, the segment is resent. The sending of a message is summarized in the following pseudo-code

```
double TCPSend_code(int seg, void *data)
{
  i = 1;
  ready = false;
  // Send packets and set up timers

  while (!ready && sendBuffer->currentSize() > 0 ) {
    // Take next segment from send buffer
    segment = (TCP_Segment*) sendBuffer->getElemByNbr(i);
    // Send if window allows
    if (segment->seqNbr <= sendWindow) {
```

```
      ttSendMsg (segment->destination, segment, segment->size);
      time = ttCurrentTime() + TIMEOUT;
      Create timer for resending at t = time;
    } else {
      // Send window full, can not send
      ready = true;
    }
    // Increase buffer index
    i++;
    if (i > sendBuffer->currentSize()) {
      // No more segments in send buffer
      ready = true;
    }
  }
  return snd_overhead_time; // task execution time
}
```

9.3 Receiving a TCP Data Segment

When a TCP segment arrives at a node, it is handled by a receiving task. An incoming TCP segment may be either a data segment or an acknowledgment of a previously transmitted segment. In the first case, it is checked if all preceding segments have been received. In this case the data is put in the receive buffer, otherwise the segment is discarded. An acknowledgment, with the latest received sequence number, is then sent back to the source node. When all segments of a message have been received, the application layer is notified.

In the case that the incoming segment is a first-time acknowledgment, it works as a cumulative acknowledgment of all previous data, and the corresponding timers are removed. If we get a duplicate acknowledgment, however, this indicates that segments in between have been lost. In this case a fast re-transmit is performed, before the actual expiry of the timer of the segment. The implementation is summarized in the following pseudo-code

```
double TCPReceive_code(int seg, void *data)
{
  // Get segment from data link layer
  TCP_Segment* segment = (TCP_Segment*) ttGetMsg();

  // Received data may be data packets or acknowledgements

  if (segment->ackNbr == -1) {
    // We received a data segment
    if (segment->seqNbr == lastRcv) {
      // have got all previous segments, put in buffer
      rcvBuffer->put(segment);
      lastRcv = segment->seqNbr + segment->size;
      Increase size of receive window;
    } else {
      // Out-of-order segment, ignore
    }
    // Send Ack
    TCP_Segment* ack = new TCP_Segment;
```

```
    ack->seqNbr = -1;
    ack->ackNbr = lastRcv;
    ack->window = rcvWindow;
    ack->source = segment->destination;
    ack->destination = segment->source;
    ttSendMsg(ack->destination, ack, ACKSIZE);

  } else {
    // We received an acknowledgement segment
    sendWindow = segment->window;
    if (segment->ackNbr > lastAck) {
       // new Ack
       lastAck = segment->ackNbr;
       Remove timeout timers;
       Delete segments from send buffer;
    } else {
      // same Ack as previously received
      Packets was lost, fast re-transmit;
    }
  }
  return rcv_overhead_time; // task execution time
}
```

9.4 Communicating with the Application Layer

The sending task is triggered from the application layer when a user wants to send a message on the specific connection. Then the message is divided in segments and stored in the send buffer for subsequent transmission to the receiver. When the message is later resembled at the receiving end the application layer is notified and the message can be read from the receive buffer.

The following example shows the code function and initialization code for a controller node communicating with a sensor and actuator node over TCP.

```
void init() {

  ttInitKernel(1, 0, FP);

  data = new Taks_Data;
  data->K = 1.5; // Controller gain
  data->u = 0.0; // To store control signal

  // Controller task
  ttCreateTask("ctrl_task", 0.006, 2.0, ctrl_code, data);
  ttCreateJob("ctrl_task", 0.0);

  // open a TCP connection with node 2 (actuator) on port #1
  actConn = ttTCPOpen(1, 2, 1);

  // open a TCP connection with node 3 (sensor) on port #2
  sensConn = ttTCPOpen(1, 3, 2);

}
```

```
void cleanup() {

  delete data;
  ttTCPClose(actConn);
  ttTCPClose(sensConn);
}

double ctrl_code(int seg, void *data)
{
  double *m;
  Task_Data* d = (Task_Data*) data;

  switch(seg) {
  case 1:
    ttTCPReceive(sensConn); // Receive TCP message from sensor node
                            // blocking call, notified by the receiving
                            // TCP task when there is a message to read
    return 0.0;

  case 2:
    m = (double*) ttTCPGet(sensConn); // Get TCP message data
                                      // from receive buffer
    d->y = *m;
    delete m;

    r = ttAnalogIn(1);

    // Compute control signal
    d->u = d->K*(r - d->y);

    return 0.0005;

  case 3:
    // Try to send a 10-byte message
    if (!ttTCPSend(10)) {
      // Send buffer full, can not send
      ttSetNextSegment(1); // Loop and wait for new message on connection
    }
    return 0.0;

  case 4:
    m = new double;
    *m = d->u;
    ttTCPPut(actConn, m, 10); // Send 10-byte message with TCP to actuator
                              // triggers the sending TCP task of the conn

    ttSetNextSegment(1); // Loop and wait for new message on connection
    return 0.0;
  }
}
```

10. Implementation Details

10.1 Task Model

TRUETIME tasks may be periodic or aperiodic. Aperiodic tasks are executed by the creation of task instances (jobs), using the command `ttCreateJob`. All pending jobs are inserted in a job queue of the task sorted by release time. For periodic task (created by the command `ttCreatePeriodicTask`), an internal timer is set up to periodically create jobs for the task.

Apart from its code function, each task is characterized by a number of attributes. The static attributes of a task include

- a relative deadline
- a priority
- a worst-case execution time
- a period (if the task is periodic)

These attributes are kept constant throughout the simulation, unless explicitly changed by the user (see `ttSetX` in the command reference).

In addition to these attributes, each task instance has dynamic attributes associated with it. These attributes are updated by the kernel as the simulation progresses, and include

- an absolute deadline
- a release time
- an execution time budget (by default equal to the worst-case execution time at the start of each task instance)
- the remaining execution time

These attributes (except the remaining execution time) may also be changed by the user during simulation. Depending on the scheduling policy, the change of an attribute may lead to a context switch. E.g., if the absolute deadline is changed and earliest-deadline-first scheduling is simulated.

In accordance with [Bollella *et al.*, 2000] it is possible to associate two interrupt handlers with 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). These handlers can be used to experiment with dynamic compensation schemes, handling missed deadlines or prolonged computations. Overrun handlers are attached to tasks with the commands `ttAttachDLHandler` and `ttAttachWCETHandler`.

Furthermore, to facilitate arbitrary dynamic scheduling mechanisms, it is possible to attach small pieces of code (*hooks*) to each task. These hooks are executed at different stages during the simulation, as shown in Figure 8. E.g.„ the overrun handling mentioned above is conveniently implemented using hooks. The following actions are taken in the various hooks

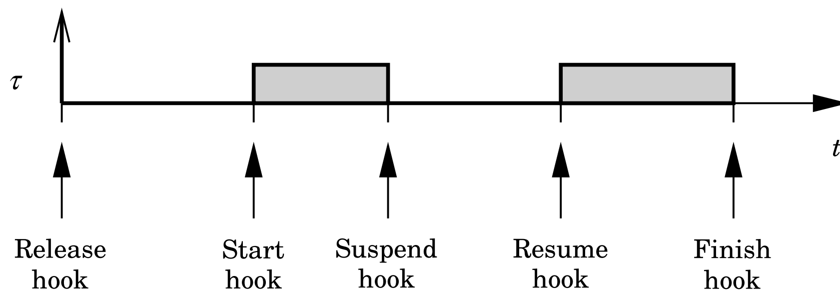


Figure 8 Scheduling hooks.

Release hook: When a task is released and it has an attached deadline overrun handler, a timer is created. The expiry of this timer is set to the absolute deadline of the task, and the deadline overrun handler is triggered upon expiry. It may be the case that, because of previous overruns, the absolute deadline of the task has already expired when the task instance is released. In this case the overrun handler is activated immediately.

Start hook: When a task is started and it has an attached worst-case execution time overrun handler, a corresponding timer is created. If the timer expires, the worst-case execution time handler is triggered.

Suspend hook: When a task is suspended, the execution time budget of the task is decreased with the time elapsed since it last began execution. The worst-case execution time timer is temporarily removed.

Resume hook: When a task is resumed, and it has remaining execution time budget, the worst-case execution time timer is again created.

Finish hook: When the task finishes execution, both overrun timers are removed.

See the file `$DIR/kernel/defaulthooks.cpp` for the actual implementation of these hooks.

10.2 The Kernel Function

The functionality of the TRUETIME kernel is implemented by the function `runKernel` in `$DIR/kernel/ttkernel.cpp`. This function manipulates the basic data structures of the kernel, such as the ready queue and the time queue, and is called by the Simulink call-back functions at appropriate times during the simulation. See Section 10.3 for timing implementation details. It is also from this function the code functions for tasks and interrupt handlers are called. The kernel keeps track of the current segment and updates it when the time associated with the previous segment has elapsed. The hooks mentioned above are also called from this function.

A simple model for how the kernel works is given by the following pseudo code. Note that interrupt handlers are not treated in the code below. However, they are treated essentially in the same way as the tasks.

```
double runKernel() {

    // Compute time elapsed since last invocation
    timeElapsed = currentTime - prevHit;
    prevHit = currentTime;
    nextHit = 0;

    while (nextHit == 0) {

        // Count down execution time for current task instance
        // and check if it has finished its execution

        if (there exists a running task) {
            Decrease remaining execution time with timeElapsed
            if (remaining execution time == 0) {
                Execute next segment of the code function
                Update remaining execution time
                Update execution time budget
                if (remaining execution time < 0.0) {
                    // Negative execution time = Job finished
                    Remove the task from the ready queue
                    Execute finish-hook
                    Simulate saving context
                    if (there are pending jobs) {
                        Move the next job to the time queue
                    }
                }
            }
        }

        // Go through the time queue (ordered after release time)

        for (each task) {
            if (release time - currentTime < 0.0) {
                Remove the task from the time queue
                Move the task to the ready queue
                Execute release-hook
            }
        }

        // Go through the timer queue (ordered after expiry)

        for (each timer) {
            if (expiry - currentTime < 0.0) {
                Activate handler associated with timer
                Remove timer from timer queue
                if (timer is periodic) {
                    Increase the expiry with the period
                    Insert the timer in the timer queue
                }
            }
        }
```

```
        // Dispatching

        Make the first task in the ready queue running task
        if (the task is being started) {
            Execute the start-hook for the task
            Simulate restoring context
        } else if (the task is being resumed) {
            Execute the resume-hook for the task
            Simulate restoring context
        }
        if (another task is suspended) {
            Execute suspend-hook of the previous task
            Simulate saving context
        }

        // Determine nextHit, next invocation of the kernel function

        time1 = remaining execution time of the current task
        time2 = next release of a task from the time queue
        time3 = next expiry of a timer
        nextHit = min(time1, time2, time3);

    } // loop while nextHit = 0.0
    return nextHit;
}
```

10.3 Timing

The TRUETIME blocks are event-driven and support external interrupt handling. Therefore, 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. As we saw above, the next time the kernel should wake up (e.g., because a task is to be released from the time queue or a task has finished its execution) 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 has occurred.

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

The timing implementation implies that *zero-crossing detection* must be turned on (this is default, and can be changed under *Simulation Parameters/Advanced*).

11. TrueTime Command Reference

The available TRUETIME commands can be divided into three categories; commands used to create and initialize TRUETIME objects, commands used to set and get task attributes, and real-time primitives. The commands are summarized in the tables below, and the rest of the manual contains detailed descriptions of their functionality.

Command	Description
ttInitKernel	Initialize the TRUETIME kernel.
ttInitNetwork	Initialize the TRUETIME network interface.
ttCreatePeriodicTask	Create a periodic TRUETIME task.
ttCreateTask	Create a TRUETIME task.
ttCreateInterruptHandler	Create a TRUETIME interrupt handler.
ttCreateExternalTrigger	Associate a TRUETIME interrupt handler with an external interrupt channel.
ttCreateMonitor	Create a TRUETIME monitor.
ttCreateEvent	Create a TRUETIME event.
ttCreateMailbox	Create a TRUETIME mailbox for inter-task communication.
ttNoSchedule	Switch off the schedule generation for a specific task or interrupt handler.
ttNonPreemptable	Make a task non-preemptable.
ttAttachDLHandler	Attach a deadline overrun handler to a task.
ttAttachWCETHandler	Attach a worst-case execution time overrun handler to a task.
ttAttachPrioFcn (C++ only)	Attach an arbitrary priority function to be used by the kernel.
ttAttachHook (C++ only)	Attach a run-time hook to a task.

Table 1 Commands used to create and initialize TRUETIME objects.

Command	Description
ttSetDeadline	Set the relative deadline of a task.
ttSetAbsDeadline	Set the absolute deadline of a task instance.
ttSetPriority	Set the priority of a task.
ttSetPeriod	Set the period of a periodic task.
ttSetBudget	Set the execution time budget of a task instance.
ttSetWCET	Set the worst-case execution time of a task.
ttGetRelease	Get the release time of a task instance.
ttGetDeadline	Get the relative deadline of a task.
ttGetAbsDeadline	Get the absolute deadline of a task instance.
ttGetPriority	Get the priority of a task.
ttGetPeriod	Get the period of a periodic task.
ttGetBudget	Get the execution time budget of a task instance.
ttGetWCET	Get the worst-case execution time of a task.

Table 2 Commands used to set and get task attributes.

Command	Description
ttCreateJob	Create a job (task instance) of a TrueTime task.
ttKillJob	Kill the running job of a task.
ttEnterMonitor	Attempt to enter a monitor.
ttExitMonitor	Exit a monitor.
ttWait	Wait for an event.
ttNotifyAll	Notify all tasks waiting for an event.
ttTryFetch	Fetch a message from a mailbox.
ttTryPost	Post a message to a mailbox.
ttCreateTimer	Create a one-shot timer and associate an interrupt handler with the timer.
ttCreatePeriodicTimer	Create a periodic timer and associate an interrupt handler with the timer.
ttRemoveTimer	Remove a specific timer.
ttCurrentTime	Get the current time in the simulation.
ttSleepUntil	Put a task to sleep until a certain point in time.
ttSleep	Put a task to sleep for a certain time.
ttAnalogIn	Read a value from an analog input channel.
ttAnalogOut	Write a value to an analog output channel.
ttSetNextSegment	Set the next segment to be executed in the code function.
ttInvokingTask	Get the name of the task that invoked an interrupt handler.
ttCallBlockSystem	Call a Simulink block diagram from within a code function.
ttSendMsg	Send a message over the network.
ttGetMsg	Get a message that has been received over the network.

Table 3 Real-time primitives.

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`	The overhead 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`

ttCreatePeriodicTask

Purpose

Create a periodic TRUETIME task.

Matlab syntax

```
ok = ttCreatePeriodicTask(name, release, period, priority, codeFcn)
ok = 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 time of the first instance of the periodic task.
`period`	Period of the task.
`priority`	Priority of the task. This should be a value greater than zero, where a small number represents a high priority.
`codeFcn`	The code function of the task, where `codeFcn` is a string (name of an M-file) in the MATLAB case 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 set equal to the task period. This may be changed by a suitable set-function.

See Also

`ttCreateTask`, `ttSetX`

ttCreateTask

Purpose

Create a TRUETIME task.

Matlab syntax

```
ok = ttCreateTask(name, deadline, priority, codeFcn)
ok = 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`	Relative deadline of the task.
`priority`	Priority of the task. This should be a value greater than zero, where a small number represents a high priority.
`codeFcn`	The code function of the task, where `codeFcn` is a string (name of an M-file) in the MATLAB case 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 an a-periodic 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`

ttCreateJob

Purpose

Create a job of a task.

Matlab syntax

```
ok = ttCreateJob(release)
ok = ttCreateJob(release, taskname)
```

C++ syntax

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

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. Jobs are queued and served after release time. This function must be called to activate a-periodic tasks, i.e., tasks created using `ttCreateTask`. The function returns `true` if successful and `false` otherwise. If the task name is not specified the call will affect the currently running task.

See Also

`ttCreateTask`, `ttKillJob`

ttKillJob

Purpose

Kill the running job of a 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 pending jobs for the task that should be released, the first job in the queue will be scheduled for execution.

See Also

`ttCreateJob`

ttCreateInterruptHandler

Purpose

Create a TRUETIME interrupt handler.

Matlab syntax

```
ok = ttCreateInterruptHandler(name, priority, codeFcn)
ok = 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. This should be a value greater than zero, where a small number represents a high priority.
`codeFcn`	The code function of the handler, where `codeFcn` is a string (name of an M-file) in the MATLAB case 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`

ttCreateExternalTrigger

Purpose

Associate a TRUETIME interrupt handler with an external interrupt channel.

Matlab syntax

```
ok = ttCreateExternalTrigger(handlername, latency)
```

C++ syntax

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

Arguments

`handlername`	Name of a created handler to be associated with the external interrupt.
`latency`	The time interval during which the interrupt channel 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. The interrupt handler is activated when the signal connected to the external interrupt port changes value. If the external signal changes again within the interrupt latency, this interrupt is ignored.

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. This function can only be called from the initialization script.

ttNonPreemptable

Purpose

Make a task non-preemptable.

Matlab syntax

```
ttNonPreemptable(taskname)
```

C++ syntax

```
void ttNonPreemptable(char* taskname)
```

Arguments

`taskname`	Name of a task.

Description

Tasks are by default preemptable. Use this function to specify that a task can not be preempted by other tasks. Non-preemptable tasks may, however, still be preempted by interrupts.

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 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/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/kernel/task.h` for the definition.

ttCreateMonitor

Purpose

Create a TRUETIME monitor.

Matlab syntax

```
ok = 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`

ttEnterMonitor

Purpose

Attempt to enter a monitor.

Matlab syntax

`ttEnterMonitor(monitorname)`

C++ syntax

`void 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 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`. *Priority inheritance* is used if a task tries to enter a monitor currently held by a lower priority task. If the attempt to enter the monitor fails, the suspend-hook of the task will be executed. When the task enters the monitor, the resumed-hook is executed.

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`

ttCreateEvent

Purpose

Create a TRUETIME event.

Matlab syntax

```
ok = ttCreateEvent(eventname)
ok = ttCreateEvent(eventname, monitorname)
```

C++ syntax

```
bool ttCreateEvent(char *eventname)
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.

Description

This function is used to create an event in the TRUETIME kernel. The function returns `true` if successful and `false` otherwise. Events may be free, or associated with a monitor.

See Also

`ttWait`, `ttNotifyAll`

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. A call to this function will trigger execution of the suspend-hook of the task. When the task is notified of the event, the resume-hook will be executed.

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 and wait for new event
  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`

ttCreateMailbox

Purpose

Create a TRUETIME mailbox for inter-task communication.

Matlab syntax

```
ok = ttCreateMailbox(mailboxname, maxsize)
```

C++ syntax

```
bool ttCreateMailbox(char *mailboxname, int maxsize)
```

Arguments

`mailboxname`	Name of the mailbox. Must be a unique, non-empty string.
`maxsize`	The size of the buffer associated with the mailbox.

Description

This function is used to create a mailbox for communication between tasks. The function returns `true` if successful and `false` otherwise. The TRUETIME mailbox implements asynchronous message passing with indirect naming. A buffer is used to store incoming messages, and the size of this buffer is specified by `maxsize`.

See Also

`ttTryFetch`, `ttTryPost`

ttTryFetch

Purpose

Fetch a message from a mailbox.

Matlab syntax

```
msg = ttTryFetch(mailboxname)
```

C++ syntax

```
void* ttTryFetch(char* mailboxname)
```

Arguments

`mailboxname`	Name of a mailbox.

Description

This function is used to fetch messages from a mailbox. If successful, the function returns the oldest message in the buffer of the mailbox. Otherwise, it returns NULL (C++) or an empty struct (MATLAB).

See Also

`ttCreateMailbox`, `ttTryPost`

ttTryPost

Purpose

Post a message to a mailbox.

Matlab syntax

```
ok = ttTryPost(mailboxname, msg)
```

C++ syntax

```
bool ttTryPost(char* mailboxname, void* msg)
```

Arguments

`mailboxname`	Name of a mailbox.
`msg`	An arbitrary data structure representing the contents of the message to be posted.

Description

This function is used to post messages to a mailbox. If successful, the message is put in the buffer of the mailbox, and the function returns `true`. Otherwise, the function returns `false`.

See Also

`ttCreateMailbox`, `ttTryFetch`

ttCreateTimer

Purpose

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

Matlab syntax

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

C++ syntax

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

Arguments

`timername`	Name of the timer. Must be unique, non-empty string.
`time`	The time when the timer is set to expire.
`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

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

C++ syntax

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

Arguments

`timername`	Name of the timer. Must be unique, non-empty string.
`start`	The time for the first expiry of the timer.
`period`	The period of the timer.
`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

```
void 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. Using this function on a periodic timer will remove the timer completely, and not only the current instance.

See Also

`ttCreateTimer`, `ttCreatePeriodicTimer`

ttCurrentTime

Purpose

Get the current time in the simulation.

Matlab syntax

```
time = 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. A call to this function will trigger execution of the suspend-hook of the task. When the task wakes up, the resume-hook will be executed.

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())`. A call to this function will trigger execution of the suspend-hook of the task. When the task wakes up, the resume-hook will be executed.

See Also

`ttSleepUntil`

ttAnalogIn

Purpose

Read a value from an analog input channel.

Matlab syntax

```
value = 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 of the computer block specified in `ttInitKernel`.

See Also

`ttAnalogOut`

ttAnalogOut

Purpose

Write a value to an analog output channel.

Matlab syntax

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

C++ syntax

```
void 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`

ttSetNextSegment

Purpose

Set the next segment to be executed in the code function.

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, e.g., the description of `ttWait`). 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

```
task = ttInvokingTask
```

C++ syntax

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

Description

This function returns the name of the task that has invoked an interrupt handler. Used, e.g., in generic 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 NULL (C++) or an empty struct (MATLAB).

See Also

`ttAttachDLHandler`, `ttAttachWCETHandler`

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

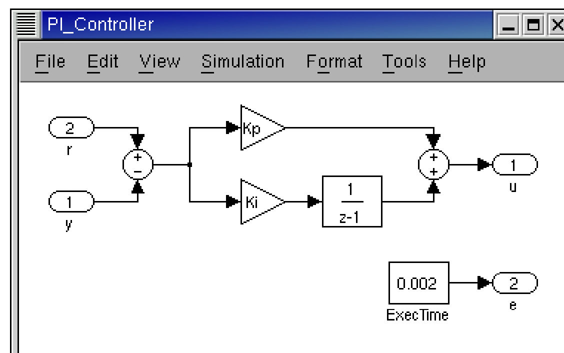
`nbrinp`	Number of inputs to the block diagram.
`nbroutp`	Number of outputs from the block diagram.
`inp`	Vector of input values.
`outp`	Vector of output values.
`blockname`	The name of the Simulink block diagram.

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 a vector of zeros 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] = PIDcontroller(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;
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):

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

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

Following are some special notes on the individual functions:

ttSetDeadline: Changing the relative deadline of a task will only affect subsequent task instances and not the absolute deadline of the currently running task instance. If deadline-monotonic scheduling is used, a call to this function may lead to a context switch, or a re-ordering of the ready queue.

ttSetAbsDeadline: A call to this function will only affect the absolute deadline for the current task instance. If a deadline overrun handler is attached to the task, this will be triggered based on the new absolute deadline. Using earliest-deadline-first scheduling, a call to this function may cause a context switch, or a re-ordering of the ready queue.

ttSetPriority: Priority values for tasks should be positive. If the task is holding a monitor, and is currently inheriting the priority of a higher priority task, the

new priority will not be assigned until the task exits the monitor. In the case of fixed-priority scheduling a call to this function may lead to a context switch, or a re-ordering of the ready queue.

ttSetPeriod: This function is only applicable to periodic tasks. Assuming a period h_1 before the call, task instances are created at times h_1, $2h_1$, $3h_1$, etc. If the call is executed at time $h_1 + \tau$, new task instances will be created at the times $h_1 + h_2$, $h_1 + 2h_2$, $h_1 + 3h_2$, etc., where h_2 is the new period of the task. Using rate-monotonic scheduling, a call to this function may cause a context switch, or a re-ordering of the ready queue.

ttSetWCET: Changes the worst-case execution time of the task. Each new task instance will get an execution time budget equal to the worst-case execution time associated with task. A call to this function will not influence the execution time budget of the currently running task instance.

ttSetBudget: This call is used to dynamically change the execution time budget of a running task instance. When a task instance is created, the execution time budget is set to the worst-case execution time of the task. A call to this function will only have effect if there is a worst-case execution time overrun handler attached to the task. This handler is activated when the budget is exhausted, and will be triggered based on the new execution time budget.

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

Use the `ttGetX` functions to retrieve the current attributes of a task. All the 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. The functions will return a value of zero if there is no task running or if the specified task does not exist.

Following are some special notes on the individual functions:

ttGetRelease: Returns the time when the current task instance was released. If there is no running task instance the function will return zero.

ttGetDeadline: Returns the relative deadline of the task.

ttGetAbsDeadline: Returns the absolute deadline of the current task instance. If there is no running task instance the function will return zero.

ttGetPriority: Returns the priority of the task. The function will return the current priority of the task, i.e., if the priority has been raised because of priority inheritance the higher priority will be returned.

ttGetPeriod: Returns the period of a periodic task.

ttGetWCET: Returns the worst-case execution time of a task.

ttGetBudget: Returns the remaining execution time budget of the current task instance. The execution time budget is decreased each time a new segment of the code function is executed, as well as when the task is suspended by another task. If there is no running task instance the function will return zero.

See Also

`ttSetX`

ttInitNetwork

Purpose

Initialize the TRUETIME network interface. If the kernel should be connected to several networks, this function must be called several times.

Matlab syntax

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

C++ syntax

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

Arguments

`network`	The number of the TrueTime network block. The default network number is 1.
`nodenumber`	The address of the node in the network. Must be a number between 1 and the number of nodes as specified in the dialog of the TRUETIME Network block.
`handlername`	The name of an interrupt handler that should be invoked 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 are no TRUETIME Network blocks in the Simulink model.

See Also

`ttSendMsg`, `ttGetMsg`

ttSendMsg

Purpose

Send a message over a network.

Matlab syntax

```
ttSendMsg(receiver, data, length)
ttSendMsg(receiver, data, length, priority)
ttSendMsg([network receiver], data, length)
ttSendMsg([network receiver], data, length, priority)
```

C++ syntax

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

Arguments

`network`	The network interface on which the message should be sent. The default network number is 1.
`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/AMP 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(s) 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 a network.

Matlab syntax

```
ttGetMsg
ttGetMsg(network)
```

C++ syntax

```
void *ttGetMsg()
void *ttGetMsg(int network)
```

Arguments

`network`	The network interface from which the message should be received. The default network number is 1.

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:

```
// Task that waits for and reads messages
double receiver_task(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

12. References

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

Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003): “How does control timing affect performance?” *IEEE Control Systems Magazine*, **23:3**, pp. 16–30.

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.

Kurose, J. F. and K. W. Ross (2001): *Computer Networking – A Top-Down Approach Featuring the Internet*. Addison-Wesley.

The Mathworks (2000): *Simulink: Dynamic System Simulation for* MATLAB. The MathWorks Inc., Natick, MA.