TRUETIME 1.3—Reference Manual

Martin Andersson
Dan Henriksson
Anton Cervin

Department of Automatic Control
Lund Institute of Technology
June 2005

Contents

1. Introduction

This manual describes the use of the MATLAB/Simulink-based [The Mathworks, 2001] 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 [Henriksson *et al.*, 2003; 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. This 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 performed to get an executable simulation. The code functions for the tasks and the initialization commands may be written either as C++ functions or as MATLAB m-files, and both cases are described.

Seven tutorial examples are provided, treating standard and distributed PID-control, scheduling, overrun handling, synchronization, control over wireless networks, and mote coordination, respectively. In the first example a DC-servo is controlled by a controller task implemented in a TRUETIME kernel block and four different implementations of the periodic controller task are demonstrated. This example is extended in the second example to the case of three PID-tasks running concurrently on the same CPU controlling three different servo systems. The third example treats networked control. The fourth and fifth examples deal with deadline overrun handling and task synchronization using TRUETIME overrun handlers and monitors, respectively. The last two examples show the use of the wireless network block and the battery block, and how to animate the movements of mobile motes.

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

2.1 Software Requirements

TRUETIME currently supports MATLAB 7.0 (R14) with Simulink 6.0 and MATLAB 6.5 (R13) with Simulink 5.0. Earlier versions also supported MATLAB 6.1 (R12.1) with Simulink 4.1, but that support has been dropped in this version.

A C++ compiler is required to run TRUETIME in the C++ version. For the MATLAB version, pre-compiled files can be downloaded from the TRUETIME web site. The following compilers are currently supported (it may, of course, also work using other compilers):

- Visual Studio C++ 7.0 (for all supported MATLAB versions) for Windows
- gcc, g++ - GNU project C and C++ Compiler for LINUX and UNIX

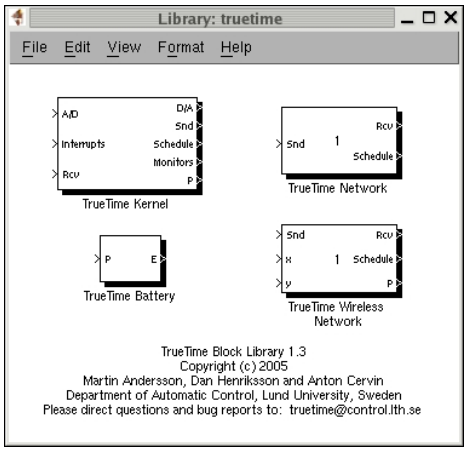


Figure 1 The TRUETIME block library.

2.2 Installation

Download and extract the compressed file (`truetime-1.3.zip`), available at:

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

Extracting the file creates a `truetime-1.3` directory, referred to as `$DIR` from now on.

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

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

Then add the following lines to your MATLAB startup script. This will set up all necessary paths to the TRUETIME kernel files.

```
addpath(getenv('TTKERNEL'))
init_truetime;
```

After starting MATLAB and before running TRUETIME *for the first time*, you must compile the TRUETIME blocks and the MEX-functions for the TRUETIME commands (unless you have downloaded the archive with pre-compiled files). This is done by issuing the command

```
>> make_truetime
```

from the MATLAB prompt. This performs all necessary compilation needed to run the MATLAB version of TRUETIME. For instructions on how to compile individual simulations in the C++ case, see Section 6.

Issuing the command

```
>> truetime
```

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

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 kernel blocks and the network block, and to create tasks, interrupt handlers, timers, events, monitors, etc for the simulation.

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.

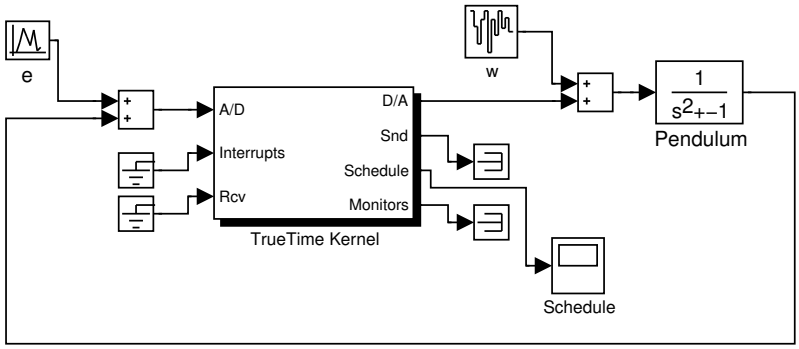


Figure 2 A TRUETIME kernel block connected to a continuous pendulum process.

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.

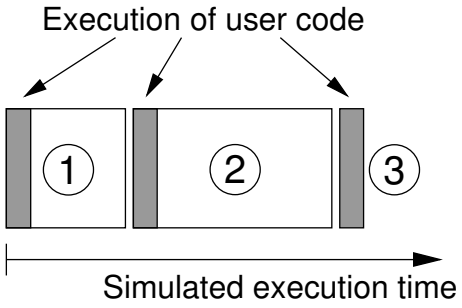


Figure 3 The execution of user code is modeled by a sequence of segments executed in order by the kernel.

4.1 Writing a MATLAB Code Function

The syntax of a MATLAB code function implementing a simple P-controller is given by Listing 1.

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.

Listing 1 Example of a P-controller code function written in MATLAB code.

```
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
```

Listing 2 The C++ version of the code function in Listing 1.

```
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
  }
}
```

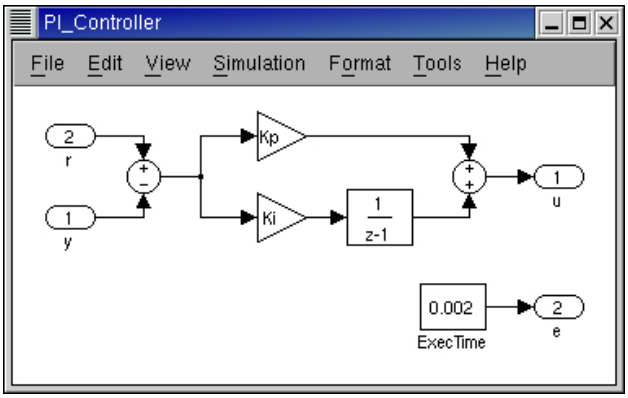


Figure 4 Controllers represented using ordinary discrete Simulink blocks may be called from within the code functions. The only requirement is that the blocks are discrete with the sample time set to one.

4.2 Writing a C++ Code Function

Writing a code function in C++ follows a similar pattern as the code function described in Listing 1. The corresponding C++ syntax for the P-controller code function is given in Listing 2. We here assume the existence of a data structure `Task_Data` that contains the control signal u and the controller gain, K.

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. Listing 3 shows an example where the discrete PI-controller in Figure 4 is used in a code function.

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

Listing 3 Simulink block diagrams are called from within code function using the TRUETIME function `ttCallBlockSystem`.

```
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
```

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. The initialization script can (in the MATLAB case) also take an optional parameter to limit the number of similar code functions. The other TRUETIME kernel block parameters are described in Section 7.

In the examples given below, the initialization script is called `example_init`, both in the MATLAB and C++ cases. The optional parameter is called `argument` when it is used.

5.1 Writing a MATLAB Initialization Script

The initialization code in Listing 4 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` that was defined in Listing 1. See the command reference for further explanation of the functions.

Listing 4 Example of a TRUETIME initialization script in the MATLAB version. The kernel is initialized using the function `ttInitKernel`, and a periodic task is created that uses the P-controller code function from Listing 1. The period of the controller is passed to the initialization script as a parameter.

```
function example_init(argument)

ttInitKernel(2, 1, 'prioFP');

data.u = 0;
data.K = 2;
offset = 0;
period = argument;
prio = 5;
ttCreatePeriodicTask('ctrl', offset, period, prio, 'Pcontroller', data);
```

5.2 Writing a C++ Initialization Script

An initialization script in C++ must follow a certain format given by the template in Listing 5.

The included file `ttkernel.cpp` contains the Simulink callback functions that implement the TRUETIME kernel, 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 callback 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 MATLAB initialization script of Listing 4 is given in Listing 6.

6. Compilation

Having run the script `make_truetime.m` as described in Section 2, no further compilation is required in the MATLAB case. This script compiles the kernel and network S-functions and the MEX-files for the TRUETIME primitives once and for all.

In the C++ case, the initialization script (`example_init.cpp` in the example from the previous section) must be compiled to produce a MATLAB MEX-file for the simulation. This is done by the command:

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

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

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

In the MATLAB case, 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', 'InitFcn', 'clear functions')
```

Listing 5 Template for writing initialization scripts in C++. The final script is actually a complete MATLAB S-function, since the included file, `ttkernel.cpp`, contains the Simulink callback functions that implement the kernel.

```
#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
}
```

7. The TrueTime Kernel

The kernel block is configured through the block mask dialog, see Figure 5. Some parameters can also be set at run time with the command `ttSetKernelParameter`.

Init function The name of the initialization script, see Section 5.

Init function argument an optional argument to the initialization script.

Battery Enable this check box if the kernel should depend on a power source.

Clock drift The time drift, 1.01 if the local time should run 1% faster than the nominal (simulated) time.

Clock offset A constant time offset from the nominal (simulated) time.

7.1 Dynamic Voltage Scaling

With the use of `ttSetKernelParameter`, the current execution speed of the kernel can be set and also the current power consumption. This makes it possible to simulate Dynamic Voltage Scaling. This functionality is very useful together with the battery block.

Listing 6 Example of a TRUETIME initialization script in the C++ version. Corresponds to the MATLAB version from Listing 4.

```
#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;
  double offset = 0.0;
  double period = 0.005;
  double prio = 5.0;
  ttCreatePeriodicTask("ctrl", offset, period, prio, Pcontroller, data);
}

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

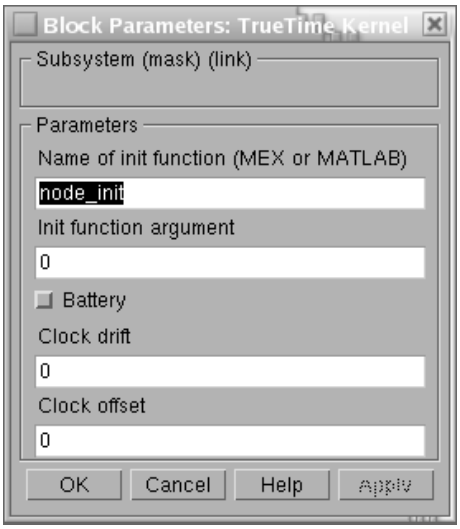


Figure 5 The dialog of the TRUETIME kernel block.

8. The TrueTime Network

The TRUETIME network block simulates medium access and packet transmission in a local area network. When a node tries to transmit a message (using the primitive `ttSendMsg`, a triggering signal is sent to the network block on the corresponding input channel. When the simulated transmission of the message is finished, the network block sends a new triggering signal on the output channel corresponding to the receiving node. The transmitted message is put in a buffer at the receiving computer node. 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.

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 6. Some parameters can also be set on a per node basis with the command `ttSetNetworkParameter`, see Section 13. 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. Wired and wireless networks are not allowed to use the same number.

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.

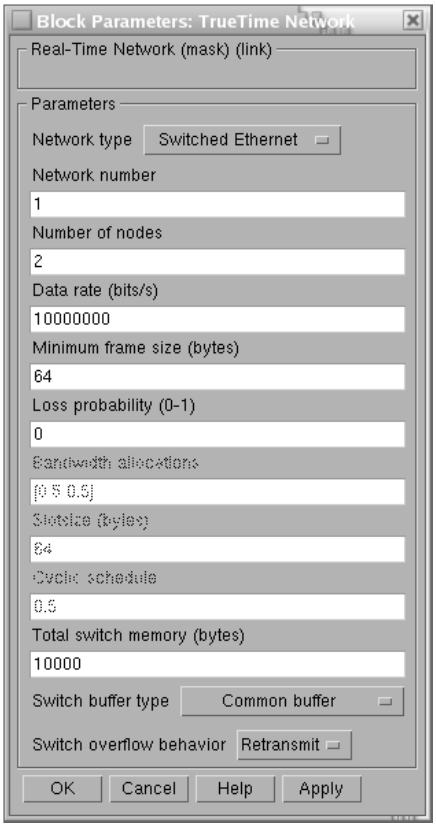


Figure 6 The dialog of the TRUETIME network block.

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.

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.

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.

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

8.2 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.)

8.3 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.

8.4 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).

8.5 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

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

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.

8.6 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 option 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.

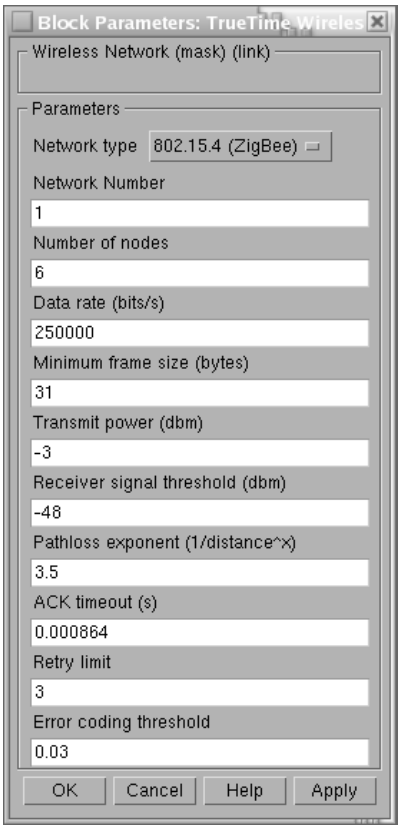


Figure 7 The dialog of the TRUETIME wireless network block.

9. The TrueTime Wireless Network

The usage of the wireless network block is similar to and works in the same way as the wired one. To also take the path-loss of the radio signals into account, it has `x` and `y` inputs to specify the true location of the nodes. Two network protocols are supported at this moment: IEEE 802.11b/g (WLAN) and IEEE 802.15.4 (ZigBee). The radio model used includes support for:

- Ad-hoc wireless networks.
- Path loss of radio signals modeled as $\frac{1}{d^a}$ where d is the distance in meters and a is a suitably chosen parameter to model the environment.
- Interference from other terminals.

The wireless network block is configured through the block mask dialog, see Figure 7. Some parameters can also be set on a per node basis with the command `ttSetNetworkParameter`. The following parameters are common to all models:

Network number The number of the network block. The networks must be numbered from 1 and upwards. Wired and wireless networks are not allowed to use the same number.

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.

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., most network protocols have a fixed number of header and tail bytes, so the frame must be at least $sizeof(header) + sizeof(tail)$ long.

Transmit power Determines how strong the radio signal will be, and thereby how long it will reach.

Receiver signal threshold If the received energy is above this threshold, then the medium is accounted as busy.

Path-loss exponent The path loss of the radio signal is modeled as $\frac{1}{d^a}$ where d is the distance in meters and a is a suitably chosen parameter to model the environment. Typically chosen in the interval 2-4.

ACK timeout The time a sending node will wait for an ACK message before concluding that the message was lost and retransmit it.

Retry limit The maximum number of times a node will try to retransmit a message before giving up.

Error coding threshold A number in the interval $[0, 1]$ which defines the percentage of block errors in a message that the coding can handle. For example, certain coding schemes can fully reconstruct a message if it has less than 3% block errors. The number of block errors are calculated using the signal-to-noise ratio, where the noise is all other ongoing transmissions.

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.

9.1 802.11b/g (WLAN)

IEEE 802.11b/g is used in many laptops and mobile devices of today. The protocol is based on CSMA/CA with some modifications.

In the simulation, a package transmission is modeled like this: The node that wants to transmit a packet checks to see if the medium is idle. The transmission may proceed, if the medium is found to be idle, and has stayed so for 50 μs. If, on the other hand, the medium is found to be busy, a random back-off time is chosen and decremented in the same way as when colliding (described later this section). When a node starts to transmit, its relative position to all other nodes in the same network is calculated, and the signal level in all those nodes are calculated according to the path-loss formula $\frac{1}{d^a}$.

The signal is assumed to be possible to detect, if the signal level in the receiving node is larger than the **receiver signal threshold**. If this is the case, then the signal-to-noise ratio (SNR) is calculated and used to find the block error rate (BLER). Note that all other transmissions add to the background noise when calculating the SNR. The BLER, together with the size of the message, is used to calculate the number of bit errors in the message and if this number is lower than the **error coding threshold**, then it is assumed that the channel coding

scheme is able to fully reconstruct the message. If there are (already) ongoing transmissions from other nodes to the receiving node and their respective SNRs are lower than the new one, then all those messages are marked as collided. Also, if there are other ongoing transmissions which the currently sending node reaches with its transmission, then those messages may be marked as collided as well.

Note that a sending node does not know if its message is colliding, therefore ACK messages are sent on the MAC protocol layer. From the perspective of the sending node, lost messages and message collisions are the same, i.e., no ACK is received. If no ACK is received during **ACK timeout**, the message is retransmitted after waiting a random back-off time within a contention window. The contention window size is doubled for every retransmission of a certain message. The back-off timer is stopped if the medium is busy, or if it has not been idle for at least 50 μs. There are only **Retry limit** number of retransmissions before the sender gives up on the message and it is not retransmitted anymore.

9.2 802.15.4 (ZigBee)

ZigBee is a protocol designed with sensor and simple control networks in mind. It has a rather low bandwidth, but also a really low power consumption. Although it is based on CSMA/CA as 802.11b/g, it is much simpler and the protocols are not the same.

The packet transmission model in ZigBee is similar to WLAN, but the MAC procedure differ and is modeled as:

1. $NB = 0$
   $BE = macMinBE$
2. Delay for $random(2^{BE} - 1)$ unit backoff periods
3. Check if the medium is idle:
   if yes, send
   else, continue
4. $NB = NB + 1$
   $BE = min(BE + 1, aMaxBE)$
5. Check if $NB > macMaxCSMABackoff$:
   if yes, drop the packet
   else, goto 2

10. The TrueTime Battery

The battery has one parameter, the initial power, which can be set using the configuration mask. To use the battery, enable the check box in the kernel configuration mask and connect the output of the battery to the `E` input of the kernel block. Connect every power drain such as the `P` output of the kernel block, ordinary Simulink models, and the wireless network block to the `P` input of the battery. The battery uses a simple integrator model, so it can be both charged and recharged.

Note that the kernel will not execute any code if it is configured to use batteries and the input `P` to the kernel block is zero. See Example 11.6 for more details of how Batteries can be used.

11. Examples

The directory `$DIR/examples` contains seven examples. In the first example a DC-servo is controlled by a controller task implemented in a TRUETIME kernel block and four different implementations of the controller task are demonstrated. This example is extended in the second example to the case of three PID-tasks running concurrently on the same CPU controlling three different servo systems. The third example treats networked control. The fourth and fifth examples deal with deadline overrun handling and task synchronization using TRUETIME overrun handlers and monitors, respectively. The last two examples show the use of the wireless network block and the battery block, and how to animate the movements of mobile motes.

Three of the examples are provided in both MATLAB and C++ versions. However, the descriptions below will only treat the MATLAB case. For detailed instructions on how to compile the examples, see the README-files in the corresponding example directories.

11.1 PID-control of a DC-servo

Introduction The first example considers simple PID control of a 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. Four different implementations of the controller task are provided to show different ways to implement periodic activities. The files are found in the directory `$DIR/examples/simple_pid/matlab`.

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

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

The PID-controller is implemented according to the following equations

$$\begin{aligned} P(k) &= K \cdot (r(k) - y(k)) \\ I(k+1) &= I(k) + \frac{Kh}{T_i}(r(k) - y(k)) \\ D(k) &= a_d D(k-1) + b_d(y(k-1) - y(k)) \\ u(k) &= P(k) + I(k) + D(k) \end{aligned} \tag{2}$$

where $a_d = \frac{T_d}{Nh+T_d}$ and $b_d = \frac{NKT_d}{Nh+T_d}$, see [Åström and Hägglund, 1995]. The controller parameters were chosen to give the system a closed-loop bandwidth, $\omega_c = 20$ rad/s, and a relative damping, $\zeta = 0.7$.

Simulation Files The initialization script (`servo_init.m`) is given in an abbreviated version in Listing 7. As seen in the initialization script, it is possible to choose between four different implementations of the periodic control task. They are specified by the init function parameter in the kernel block dialog.

- *Implementation 1*: Uses the TRUETIME built-in support for periodic tasks, and the code function is given in the file `pidcode1.m`.
- *Implementation 2*: Also uses the TRUETIME built-in support for periodic tasks, but the computation of the control signal in each sample is done by calling a Simulink block diagram. The code function is given in the file `blockcode.m`. Since all the controller parameters and states are contained in the Simulink block, the task data (`data2`) only consist of the control signal, u.
- *Implementation 3*: Implements the periodic task by using the TRUETIME primitive `ttSleepUntil`. The code function is given in the file `pidcode2.m`.
- *Implementation 4*: Implements the periodic task by using a periodic timer. The associated interrupt handler samples the process and triggers task jobs. The handler and controller task communicate using a mailbox. The code functions for the handler and controller are given in the files `samplercode.m` and `pidcode3.m`, respectively.

Simulations The Simulink model is called `servo.mdl` and is given in Figure 8. Open the Simulink model and 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, `pidcode1.m`. 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.
- Change the sampling period and study the resulting control performance.
- A PID-controller is implemented in the Simulink block `controller.mdl`. Change the init function parameter of the kernel block from 1 to 2, so that implementation 2 is used instead of 1. Study the corresponding code function, `blockcode.m`. This code function is using the Simulink block to compute the control signal in each sample.
- Change to implementation 3 and run a simulation. Study the code function, `pidcode2.m`.
- Change to implementation 4 and run a simulation. Study the code functions, `samplercode.m` and `pidcode3.m`. Notice the inclusion of the handler in the schedule plot.

11.2 Task Scheduling and Control

Introduction This example extends the simple PID control example from the previous section to the case of three PID-tasks running concurrently on the same CPU controlling three different DC-servo systems. The effect of the scheduling policy on the global control performance is demonstrated. The files are found in the directory `$DIR/examples/threeservos/matlab`.

Simulations Open the Simulink model `threeservos.mdl` and try the following

Listing 7 The initialization script for the PID-control example.

```
  function servo_init(mode)

  ttInitKernel(2, 1, 'prioFP'); % nbrOfInputs, nbrOfOutputs, fixed priority

  period = 0.006;
  deadline = period;
  offset = 0.0;
  prio = 1;

  data.K = 0.96;
  ... % more task data

switch mode,
 case 1,
  % IMPLEMENTATION 1a: using the built-in support for periodic tasks
  %
  ttCreatePeriodicTask('pid_task',offset,period,prio,'pidcode1',data);

 case 2,
  % IMPLEMENTATION 1b: calling Simulink block within code function
  %
  data2.u = 0;
  ttCreatePeriodicTask('pid_task',offset,period,prio,'blockcode',data2);

 case 3,
  % IMPLEMENTATION 2: sleepUntil and loop back

  data.t = 0;
  ttCreateTask('pid_task',deadline,prio,'pidcode2',data);
  ttCreateJob('pid_task');

 case 4,
  % IMPLEMENTATION 3: sampling in timer handler, triggers task job

  hdl_data.yChan = 2;
  ttCreateInterruptHandler('timer_handler',prio,'samplercode',hdl_data);
  ttCreatePeriodicTimer('timer',offset,period,'timer_handler');
  ttCreateMailbox('Samples',10);
  ttCreateTask('pid_task',deadline,prio,'pidcode3',data);
end
```

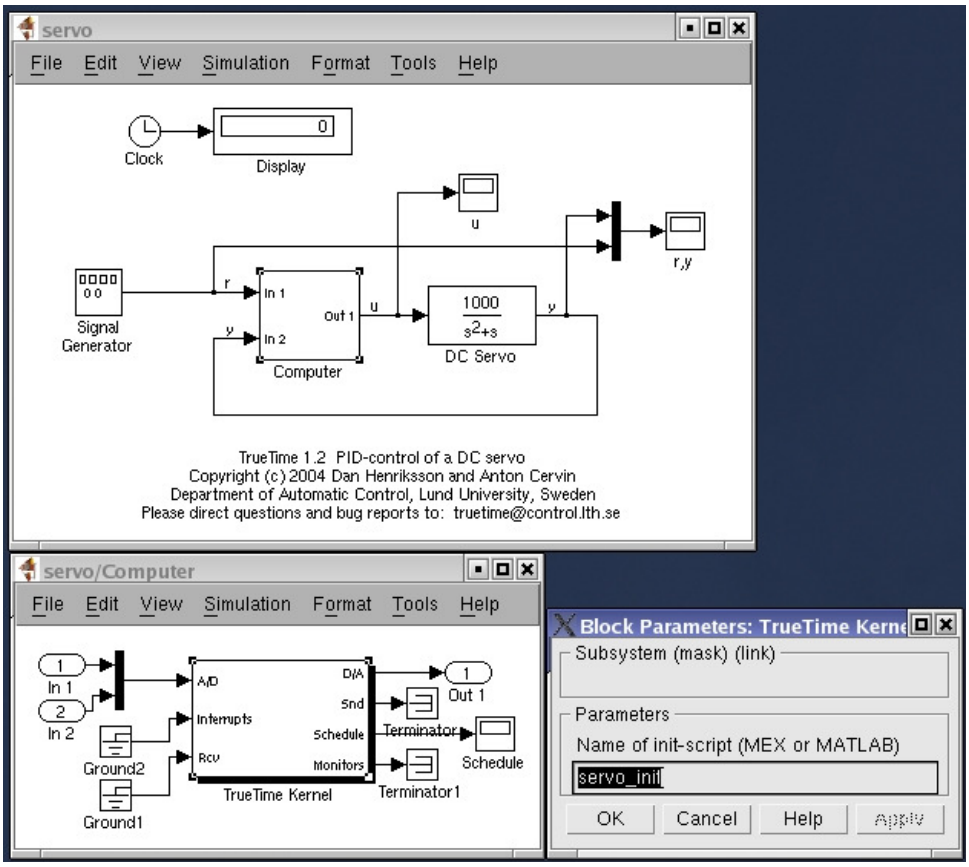


Figure 8 The TRUETIME model of the DC-servo system.

- 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 (change `'prioRM'` to `'prioEDF'`) 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 still the overall control performance is satisfactory.

11.3 Distributed Control System

Introduction This example simulates distributed control of the DC-servo of Equation (1). The example contains four computer nodes, each represented by a TRUETIME kernel block. A 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 files are found in the directory `$DIR/examples/distributed/matlab`.

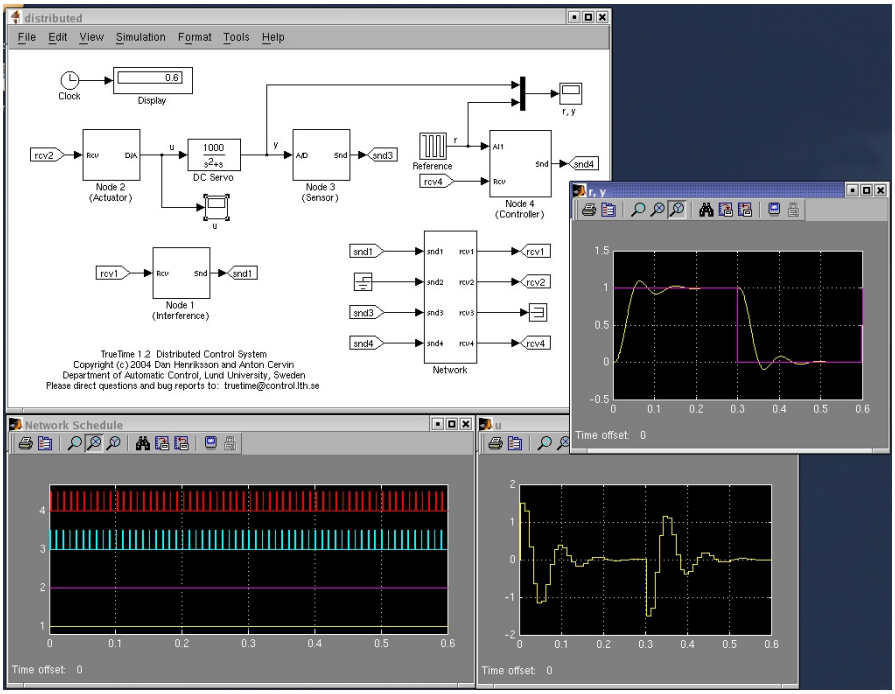


Figure 9 The TRUETIME model of the distributed control system.

Simulations The Simulink model is called `distributed.mdl` and is given in Figure 9. Open the Simulink model and 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 triggers the task that will read and process the message.
- 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.

11.4 Deadline Overrun Handling

Introduction This example will show how to use the TRUETIME overrun handling functionality. TRUETIME provides two types of overrun handlers; deadline and worst-case execution time overrun handlers. The example again considers PID-control of the DC-servo described by Equations (1) and (2). However, now

Figure 10 The ball and beam laboratory process.

the controller task is having a stochastically varying execution time that occasionally will exceed the sampling interval. Two approaches to deal with the period overruns are evaluated in the simulation. The first allows the task to continue into next sample (no overrun handler is attached), whereas the second uses an overrun handler that terminates the current job if the deadline is exceeded. The files are found in the directory `$DIR/examples/overrun`.

Simulations Open the Simulink model `overrun.mdl` and try the following

- Study the code function `pidcode.m`, then run a simulation. The period of the controller task is 6 ms, and the execution time is modeled as $C = 5 + U(0, 2)$ ms. Consequently, the task will experience overruns. The bad control performance is due to the long delays and the sampling period jitter induced by the overruns.
- Uncomment the last two lines of the initialization file (`overrun_init.m`). This will create an interrupt handler and attach it to the controller task as a deadline handler. Study the code executed by the overrun handler (`hdlcode.m`).
- Run a simulation to evaluate the performance obtained by terminating jobs at the deadline (is this a good approach?). Studying the control signal, one can notice that it often remains constant over several samples.

11.5 Task Synchronization Using Monitors

Introduction This example shows how to use monitors to obtain mutual exclusion in TRUETIME. A cascaded controller for a ball-and-beam process is implemented using separate tasks for the two loops in the cascade. The output from the outer controller is used as input to the inner controller and is communicated using a global variable. This variable is a shared resource, and mutual exclusion is achieved by a TRUETIME monitor. The files are found in the directory `$DIR/examples/synch`.

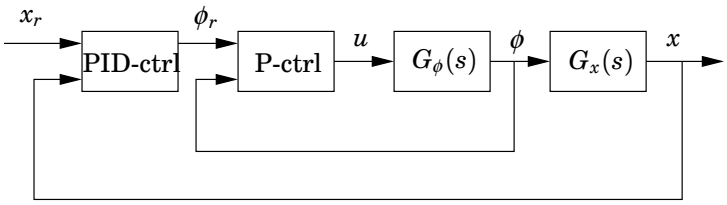


Figure 11 The cascaded controller structure for the ball and beam process.

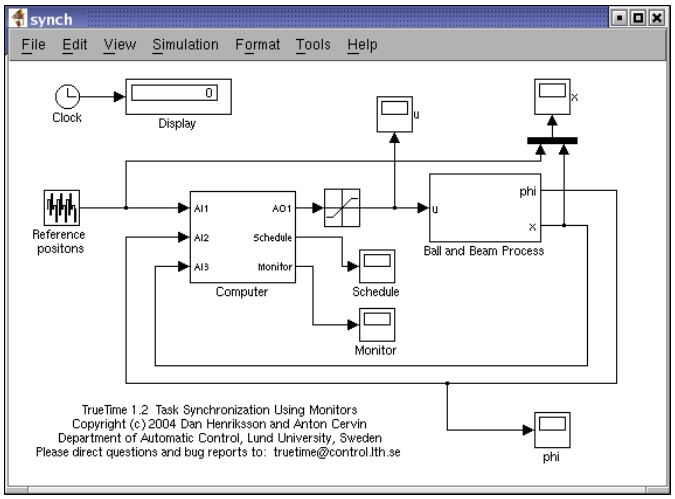


Figure 12 The TRUETIME model of the ball and beam system.

Process and Controller The ball and beam laboratory process is shown in Figure 10. The horizontal beam is controlled by a motor, and the objective is to balance the ball along the beam. The measurement signals from the system are the beam angle, denoted by ϕ, and the ball position on the beam, denoted by x. A linearized model of the system is given by

$$G(s) = G_\phi(s)G_x(s) \tag{3}$$

where

$$G_\phi(s) = \frac{k_\phi}{s} \tag{4}$$

is the transfer function between the motor input and the beam angle, and

$$G_x(s) = -\frac{k_x}{s^2} \tag{5}$$

is the transfer function between the beam angle and the ball position. The gains of the systems are given by $k_\phi \approx 4.4$ and $k_x \approx 9$.

The cascaded controller is shown in Figure 11. The outer controller is a PID-controller (implemented according to Equation (2)) and the inner controller is a simple P-controller.

Simulations The Simulink model is called `synch.mdl` and is given in Figure 12. Open the Simulink model and try the following

- Study the initialization function (`synch_init.m`). This creates two tasks for the outer and inner loop, respectively. A global variable, `outerU`, is used for task communication. This variable is the output from the outer controller (thus its name) and is used as reference for the inner controller (it is denoted ϕ_r in Figure 11). Finally, a TRUETIME monitor is created.
- Study the code functions for the controller tasks (`outercode.m` and `innercode.m`). To ensure that no further instructions are executed in the case that `ttEnterMonitor` fails, this primitive needs to be called from its own segment (since all code of a TRUETIME segment is executed at once before scheduling decisions are made). The same holds for `ttExitMonitor` to make sure that no further code is executed in the case a context switch will occur when the monitor is released.
- Run a simulation and study the monitor graph. This graph displays when the various tasks have been holding the monitor during the simulation.
- Try modifying the periods of the tasks to change the phasing and to see which loop that is most sensitive to slower sampling.

11.6 Distributed Wireless Control System

Introduction This example shows distributed control of a DC-servo described by Equation (1) using communication over a wireless network. The example also shows how to simulate power consumption and how to use the battery block. The model contains two computer nodes located 20 m apart, each represented by a TRUETIME kernel and battery block. A time-driven sensor/actuator 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 back to the sensor/actuator node, where it is actuated. The wireless communication link is at the same time subject to a simple power control scheme. Power control tasks running in both the sensor/actuator node and in the controller node periodically send out ping messages to the other node to test the channel transmission. If a reply is received, the channel is assumed to be good and the transmission power is lowered. If on the other hand no reply is received, then the transmission power is considerably increased until it saturates or a reply is received again. The files are found in the directory `$DIR/examples/wireless`.

Simulations Open the model `wireless.mdl` to run the simulation.

- Run a first simulation without modifying anything. Look at the plots showing the battery levels in the two nodes. Note that the power control scheme is not activated until 2 seconds have elapsed. Also note how the measured values at some times deviate more than usual from the reference values. This deviation is caused by the fact that it is possible to lose several consecutive sensor value readings when using the simple power control that is implemented in the nodes.
- Switch off the power control scheme in the controller node. This is done by commenting out the creation of the task `power_controller_task` in `controller_init`. Run the simulation again and now note that the power drain is constant in the controller node. This causes the battery to run out of energy and the control is lost.

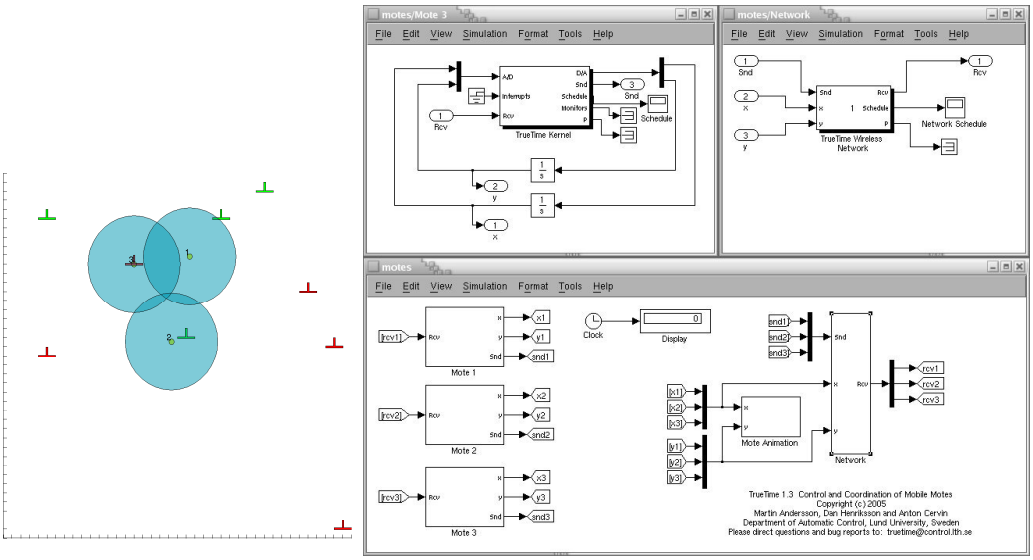


Figure 13 Visualisation of motes in Example 11.7.

- Experiment with different network parameters and protocols and see how it affects the control behaviour. In this example, the kernel block is set to consume 10 mW. This can easily be changed by using the command `ttSetKernelParameter`. This command can also be used to set the CPU scaling factor to enable Dynamic Voltage Scaling.

11.7 Control and Coordination of Mobile Motes

Introduction This example shows how to visualise the movements of dynamically moving motes using the built in functionality of matlab. The example consists of three motes, with dynamics in the `x` and `y` directions modeled using simple integrators. The motes are sent out on a mission which consists of visiting a number of checkpoints seen as red marks in the animation window in Figure 13. In the window, the transmission range of the motes can be seen as large partly transparent coloured circles around the smaller coloured motes. The checkpoints should be visited at least once by some node in the group. When the motes are able to communicate, they tell each other where they are heading and share information on which nodes that have been visited by the group. Some of this information is visible as printouts during the execution. When a checkpoint has been visited it changes colour from green to red. The files are found in the directory `$DIR/examples/motes`.

Simulations Open the model `motes.mdl` to run the simulation.

- Run `init` to set up the data structures before you run the simulation.
- Run the simulation to see the animation. If you are interested in building your own custom animation, look at the file `moteanimation.m` where the `patch` command is used to paint the initial picture of every mote. These pictures are then moved with the `set` command using the `XData` and `YData` parameters.

The motes operate according to the following algorithm:

1. Read new network messages containing information of:
   visited nodes
   target of the sending node
2. If (someone has the same target as we do && we have the lowest priority),
   then change target
3. If (heading to a place that has already been visited),
   then change target
4. If (arrived at target)
   then paint the target green && change target
5. Send new network messages to other nodes

12. Kernel Implementation Details

This section will give a brief description of the implementation of the TRUETIME kernel. The main data structures will be described as well as the kernel implementation. It will also be described how the event-based kernel simulation is achieved in Simulink, using the zero-crossing detection mechanism.

12.1 Kernel Data Structures

The main data structure of the TRUETIME kernel is a C++ class called `RTsys`, see `$DIR/kernel/ttkernel.h`. An instance (`rtsys`) of this class is created in the initialization step of the kernel S-function. The `rtsys` object is stored in the `UserData` field of the kernel block between simulation steps. Among others, the `RTsys` class contains the following attributes:

```
class RTsys {
 public:
  double time;              // Current time in simulation

  double* inputs;           // Vector of input port values
  double* outputs;          // Vector of output port values

  Task* running;            // Currently running task

  List* readyQ;   // usertasks and handlers ready for execution, prio-sorted
  List* timeQ;    // usertasks and timers waiting for release, time-sorted

  List* taskList;    // A list containing all created tasks
  List* handlerList;
  List* monitorList;
  List* eventList;

  double (*prioFcn)(Task*); // Priority function
};
```

The ready queue and time queue are sorted linked list. The elements in the time queue (tasks and timers) are sorted according to release times and expiry times.

A timer in the time queue is actually represented by its corresponding handler. The tasks in the ready queue are sorted according to the priority function `prioFcn`, which is a function that returns a (possibly dynamic) priority number from a `Task` instance, see the description of `ttAttachPrioFcn` in the command reference.

The `Task` class (`$DIR/kernel/task.h`) inherits from the `node` class of the linked list (`$DIR/kernel/linkedlist.h`) and contains the following basic attributes:

```
class Task : public Node {
 public:
  char* name;
  int segment;      // the current segment of the code function
  double execTime;  // the remaining execution time of the current segment

  void *data;       // task data (C++ case)
  char* dataMATLAB; // name of global variable for task data (MATLAB case)

  double (*codeFcn)(int, void*); // Code function (C++ case)
  char* codeFcnMATLAB;  // Name of m-file code function (MATLAB case)
};
```

The `exectime` of the running task is updated each time the kernel executed, see Listing 8. When it has reached zero, the next segment of the code function is executed. The task data in the MATLAB case is represented as a name of a unique global variable. The code function of a task is represented either as a function pointer in the C++ case or the name of a MATLAB m-file.

User tasks and interrupt handlers are both subclasses to `Task` and contain the attributes given below, among others. See `$DIR/kernel/usertask.h` and `$DIR/kernel/handler.h` for complete descriptions.

```
class UserTask : public Task {
public:
  double priority;
  double wcExecTime;
  double deadline;
  double absDeadline;
  double release;   // task release time if in timeQ
  double budget;

  int state;  // Task state (IDLE; WAITING; SLEEPING; READY; RUNNING)

  double tempPrio;  // temporarily raised prio value

  List *pending; // list of pending jobs

  InterruptHandler* deadlineORhandler; // deadline overrun handler
  InterruptHandler* exectimeORhandler; // execution-time overrun handler

  int nbrOfUserLogs;  // Number of user-created log entries
  Log* logs[NBRLOGS];

  void (*arrival_hook)(UserTask*);  // hooks
  void (*release_hook)(UserTask*);
```

```
  void (*start_hook)(UserTask*);
  void (*suspend_hook)(UserTask*);
  void (*resume_hook)(UserTask*);
  void (*finish_hook)(UserTask*);
};
```

The kernel implements priority inheritance to avoid priority inversion. Therefore each task has a dynamic priority value that may be raised while executing inside a monitor. Pending jobs are stored in the job queue of the task sorted by release time. See `$DIR/kernel/log.h` for the contents of the `Log` class.

```
class InterruptHandler : public Task {
 public:
  double priority;

  int type;  // {UNUSED, OVERRUN, TIMER, NETWORK, EXTERNAL}

  UserTask *usertask; // if overrun handler to task

  Timer* timer;       // if associated with timer interrupt

  Network* network;   // if associated with network receive interrupt

  Trigger* trigger;   // if associated with external interrupt
  int pending;        // list of pending invocations, if new external
                      // interrupt occurs before the old is served
};
```

See the corresponding header files in `$DIR/kernel` for the specifications of the classes `Timer`, `Network`, and `Trigger`.

12.2 Task Model

TRUETIME user tasks may be periodic or aperiodic. Aperiodic tasks are triggered by the creation of task 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 job has dynamic attributes associated with it. These attributes are updated by the kernel as the simulation progresses, and include

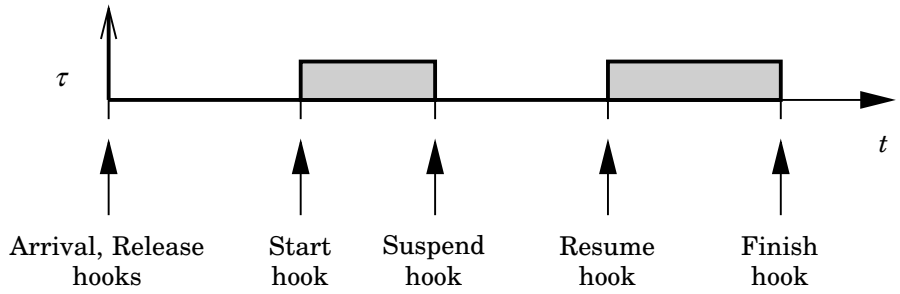


Figure 14 Scheduling hooks.

- 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 job)
- 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`. See Section 11.4 for an example on how to use overrun handlers.

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 14. Usually the arrival and release of a task job coincide. The exception is when a job is created while previous jobs have yet to finish. In that case, the arrival hook is executed immediately (at the call of `ttCreateJob`) and the release hook is called when the job is subsequently released from the job queue.

The hooks can, e.g., be used to monitor different scheduling schemes and keep track of context switches and deadline overruns. By default, the hooks implement logging, simulation of context switching, and contain code to trigger the worst-case execution time and deadline overrun handlers possibly associated with the different tasks. For the default hook implementation, see `$DIR/kernel/default_hooks.cpp`.

12.3 The Kernel Function

The functionality of the TRUETIME kernel is implemented by the function `runKernel` located 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 12.4 for timing implementation details.

It is also from this function that 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 pseudo code in Listing 8. This code focuses on user tasks. See `$DIR/kernel/ttkernel.cpp` for the complete implementation.

Listing 8 Pseudo code for the TRUETIME kernel function.

```
double runKernel(void) {

  timeElapsed = rtsys->time - rtsys->prevHit; // time since last invocation
  rtsys->prevHit = rtsys->time;  // update previous invocation time
  nextHit = 0.0;

  while (nextHit == 0.0) {
    // Count down execution time for current task
    // and check if it has finished its execution
    if (there exists a running task) {
      task->execTime -= timeElapsed;
      if (task->execTime == 0.0) {
        task->segment++;
        task->execTime = task->codeFcn(task->segment, task->data);
        if (task->execTime < 0.0) {
          // Negative execution time = task finished
          task->execTime = 0.0;
          task->segment = 0;
          Remove task from readyQ;
          task->finish_hook(task);
          if (job queue is non-empty)
            Release next job and execute release-hook ;
        }
      }
    } // end: counting down execution time of running task

    // Check time queue for possible releases (user tasks or timers)
    for (each task) {
      if ((release time - rtsys->time) == 0.0) {
        Move task to ready queue
      }
    } // end: checking timeQ for releases

    // Determine task with highest priority and make it running task
    newrunning = rtsys->readyQ->getFirst();
    oldrunning = rtsys->running;
    if (oldrunning is being suspended) {
      oldrunning->suspend_hook(oldrunning);
    }
    if (newrunning is being resumed or started) {
      if (newrunning->segment == 0) {
        newrunning->start_hook(newrunning);
      } else {
        newrunning->resume_hook(newrunning);
      }
    } // end: task dispatching

    // Determine next invocation of kernel function
    time1 = remaining execution time of current task;
    time2 = next release time of a task from the time queue
    nextHit = min(time1, time2);
  } // end: loop while nextHit == 0.0
  return nextHit;
}
```

12.4 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*).

13. TrueTime Command Reference

The available TRUETIME commands are summarized in Tables 1–3, and the rest of the manual contains detailed descriptions of their functionality. The commands are categorized according to their intended use (**I**; initialization script, **T**; task code function, and **H**; interrupt handler code function). Note that the set and get primitives are collected under the headings `ttSetX` and `ttGetX`, respectively.

By typing `help command`, where `command` is the name of a TRUETIME function, in the MATLAB command window, the syntax of the various TRUETIME functions will be displayed.

Command	Description
`ttInitKernel`	Initialize the kernel.
`ttInitNetwork`	Initialize the network interface.
`ttCreateTask`	Create a task.
`ttCreatePeriodicTask`	Create a periodic task.
`ttCreateInterruptHandler`	Create an interrupt handler.
`ttCreateExternalTrigger`	Associate a interrupt handler with an external interrupt channel.
`ttCreateMonitor`	Create a monitor.
`ttCreateEvent`	Create an event.
`ttCreateLog`	Create a log structure and specify data to log.
`ttCreateMailbox`	Create a 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 job.
`ttSetPriority`	Set the priority of a task.
`ttSetPeriod`	Set the period of a periodic task.
`ttSetBudget`	Set the execution time budget of a task job.
`ttSetWCET`	Set the worst-case execution time of a task.
`ttGetRelease`	Get the release time of a task job.
`ttGetDeadline`	Get the relative deadline of a task.
`ttGetAbsDeadline`	Get the absolute deadline of a task job.
`ttGetPriority`	Get the priority of a task.
`ttGetPeriod`	Get the period of a periodic task.
`ttGetBudget`	Get the execution time budget of a task job.
`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 of a task.
ttKillJob	Kill the running job of a task.
ttEnterMonitor	Attempt to enter a monitor.
ttExitMonitor	Exit a monitor.
ttWait	Wait for an event.
ttNotify	Notify the highest-priority task waiting for an event.
ttNotifyAll	Notify all tasks waiting for an event.
ttLogNow	Log the current time.
ttLogStart	Start a timing measurement for a log.
ttLogStop	Stop a timing measurement and save in the log.
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 and/or set the current time in the simulation on a per node basis.
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 (to implement loops and branches).
ttInvokingTask	Get the name of the task that invoked a task overrun handler.
ttCallBlockSystem	Call a Simulink block diagram from within a code function.
ttSendMsg	Send a message over a TRUETIME network.
ttGetMsg	Get a message that has been received over a TRUETIME network.
ttSetNetworkParameter	Set a specific network parameter on a per node basis.
ttSetKernelParameter	Set a specific kernel parameter on a per node basis.

Table 3 Real-time primitives.

ttAnalogIn (TH)

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 kernel block specified by `ttInitKernel`.

See Also

`ttInitKernel`, `ttAnalogOut`

ttAnalogOut (TH)

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 by `ttInitKernel`.

See Also

`ttInitKernel`, `ttAnalogIn`

ttAttachDLHandler (I)

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`

ttAttachHook (C++ only) (I)

Purpose

Attach a run-time hook to a task.

C++ syntax

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

Arguments

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

Description

This function is used to attach a run-time hook to a specific task. The hook identifier determines at which times during the simulation the hook will be called. It is possible to attach hooks that are called when a task job arrives, 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. Usually the arrival and release of a task job coincide. The exception is when a job is created while previous jobs have yet to finish. In that case, the arrival hook is executed immediately (at the call of `ttCreateJob`) but the job is queued. The release hook is called when the job is subsequently released from the job queue.

The input to the hook function is a pointer to a `UserTask` object. `UserTask` inherits from the superclass `Task`. See `$DIR/kernel/usertask.h` and `$DIR/kernel/task.h` for the definitions. The kernel uses hooks internally to implement logging, triggering of task overrun handlers, and simulation of context switching. These hooks are contained in the file `$DIR/kernel/defaulthooks.cpp` and should be included in the user-defined hooks (see the example below).

Example

The example below shows a custom finish hook that estimates the execution time of the task using a first-order filter:

```
void myFinishHook(UserTask* task) {

  // Compute execution time (the initial budget of a task job is the WCET)
  double exectime = task->wcExecTime - task->budget;

  // Update estimate
  double lambda = 0.5;
  task->data->Chat = lambda*task->data->Chat + (1.0-lambda)*exectime;

  // Execute default finish hook
  default_finish(task);
}
```

ttAttachPrioFcn (C++ only) (I)

Purpose

Attach an arbitrary priority function to be used for task scheduling.

C++ syntax

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

Arguments

`prioFcn` The priority function to be attached.

Description

This function is used to attach an arbitrary priority function to the kernel. The priority function returns a number that will be used by the kernel for task scheduling. The lower the number, the higher the priority of the task. The input to the priority function is a pointer to a `UserTask` object. `UserTask` inherits from the superclass `Task`. See `$DIR/kernel/usertask.h` and `$DIR/kernel/task.h` for the definitions.

Example

As two examples, the priority functions implementing fixed-priority scheduling and earliest-deadline-first scheduling are given below:

```
double prioFP(UserTask* task) {

  return task->priority;
}


double prioEDF(UserTask* task) {

  return task->absDeadline;
}
```

ttAttachWCETHandler (I)

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`

ttCallBlockSystem (TH)

Purpose

Call a Simulink block diagram from within a code function.

Matlab syntax

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

C++ syntax

```
void 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

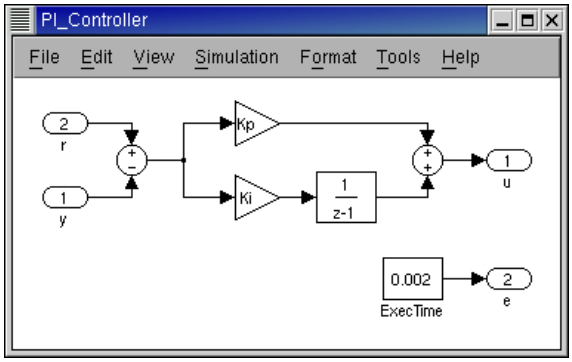
This function is used to call a Simulink block diagram from within a code function. At each call, a one-second simulation of the block is performed, using the old states as initial values. The states of the block diagram are stored internally by the kernel between calls. Consequently, *the block diagrams may only contain discrete blocks and the sampling times should be set to one*. The inputs and outputs are defined by Simulink inports and outports, see the figure below.

Example

Here follows an example using the Simulink diagram in the figure:

```
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
```



ttCreateEvent (I)

Purpose

Create a TRUETIME event.

Matlab syntax

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

C++ syntax

```
void ttCreateEvent(char *eventname)
void 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. Events may be free, or associated with a monitor.

See Also

`ttWait`, `ttNotify`, `ttNotifyAll`

ttCreateExternalTrigger (I)

Purpose

Associate an interrupt handler with an external interrupt channel.

Matlab syntax

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

C++ syntax

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

Arguments

`handlername`	Name of the interrupt 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 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`

ttCreateInterruptHandler (I)

Purpose

Create a TRUETIME interrupt handler.

Matlab syntax

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

C++ syntax

```
void ttCreateInterruptHandler(char *name, double priority,
                              double (*codeFcn)(int, void*))
void 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 a handler that will be executed in response to interrupts. Interrupt handlers may be associated with timers, the network receive channel, external interrupt channels, or attached to tasks as overrun handlers. Each handler may only be associated with one interrupt source.

See Also

`ttCreateTimer`, `ttCreatePeriodicTimer`, `ttCreateExternalTrigger`, `ttInitNetwork`, `ttAttachDLHandler`, `ttAttachWCETHandler`

ttCreateJob (ITH)

NOTE: The syntax has been changed since TrueTime 1.13.

Purpose

Create a job of a task.

Matlab syntax

```
ttCreateJob(taskname)
```

C++ syntax

```
void ttCreateJob(char *taskname)
```

Arguments

`taskname`	Name of a task.

Description

This function is used to create job jobs of tasks. If there already is a job active for the task, the job is queued and served as soon as possible. `ttCreateJob` must be called to activate aperiodic tasks, i.e., tasks created using `ttCreateTask`. A call to `ttCreateJob` will trigger the arrival hook of the task. If there are no active jobs the release hook will be called as well. Otherwise, the release hook will be called when the job is later activated from the job queue.

See Also

`ttCreateTask`, `ttKillJob`

ttCreateLog (I)

Purpose

Create a log structure and specify attribute to log.

Matlab syntax

```
ttCreateLog(taskname, logtype, variable, size)
```

C++ syntax

```
void ttCreateLog(char* taskname, int logtype, char* variable, int size)
```

Arguments

`taskname`	Name of a task.
`logtype`	The log type (see description below).
`variable`	The name of the variable in MATLAB workspace to which the log will be written after the simulation.
`size`	The maximum number of elements in the log.

Description

This function is used to create logs for individual tasks. Five pre-defined log types exist to log response time, release latency, start latency, execution time, and context switch instances. These are obtained by setting the variable `logtype` to any of the constants one to five, respectively. It is also possible to create five additional log structures for each task (by specifying log type number six). These user-controlled logs are written to from the code functions using the primitives `ttLogStart`, `ttLogStop`, and `ttLogNow`. After the simulation the logged attributes can be found in MATLAB workspace, having the name specified by `variable`.

Example

Logging of response time and input-output latency

```
% In initialization script

% Automatic log of response time (type 1)
ttCreateLog('ctrl_task', 1, 'Responsetime', 100);

% User log #1 (type 6) for logging of I/O latency
ttCreateLog('ctrl_task', 6, 'IOlatency', 100);


% Code function
function [exectime, data] = ctrl(seg, data)

switch seg,
  case 1,
    ttLogStart(1);  % start I/O logging in user log #1
    y = ttAnalogIn(1);   % Input
    data.u = calculateOutput(y);
    exectime = 0.003;
  case 2,
```

```
    ttLogStop(1);  % stop and write log entry in user log #1
    ttAnalogOut(1, data.u);  % Output
    exectime = -1;
end
```

See Also

ttLogNow, ttLogStart, ttLogStop

ttCreateMailbox (I)

Purpose

Create a TRUETIME mailbox for inter-task communication.

Matlab syntax

```
ttCreateMailbox(mailboxname, maxsize)
```

C++ syntax

```
void 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 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`

ttCreateMonitor (I)

Purpose

Create a TRUETIME monitor.

Matlab syntax

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

C++ syntax

```
void 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 for task synchronization. Condition variables for the monitor can be created using `ttCreateEvent`. The kernel block has a monitor output that will display a graph showing when the various tasks have access to the monitors.

See Also

`ttEnterMonitor`, `ttExitMonitor`, `ttCreateEvent`

ttCreatePeriodicTask (I)

Purpose

Create a periodic TRUETIME task.

Matlab syntax

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

C++ syntax

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

Arguments

`name`	Name of the task. Must be a unique, non-empty string.
`offset`	Release time for the first job 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. Only relevant for fixed-priority scheduling.
`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 periodicity is implemented internally by the kernel using a periodic timer. See the simple PID-control example in `$DIR/examples/simple_pid` for other ways to implement periodic activities. *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`

ttCreatePeriodicTimer (ITH)

Purpose

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

Matlab syntax

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

C++ syntax

```
void 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.

See Also

`ttCreateInterruptHandler`, `ttCreateTimer`, `ttRemoveTimer`

ttCreateTask (I)

Purpose

Create a TRUETIME task.

Matlab syntax

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

C++ syntax

```
void ttCreateTask(char* name, double deadline, double priority,
                  double (*codeFcn)(int, void*))
void 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. Only relevant for fixed-priority scheduling.
`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 task to run in the TRUETIME kernel. Note that no task job is created by this function. This is done by the primitive `ttCreateJob`. *The worst-case execution time of the task is by default set equal to the task deadline*. This may be changed by a suitable set-function.

See Also

`ttCreatePeriodicTask`, `ttCreateJob`, `ttSetX`

ttCreateTimer (ITH)

Purpose

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

Matlab syntax

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

C++ syntax

```
void 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.

See Also

`ttCreateInterruptHandler`, `ttCreatePeriodicTimer`, `ttRemoveTimer`

ttCurrentTime (ITH)

Purpose

Get and/or set the current time in the simulation on a per node basis.

Matlab syntax

```
time = ttCurrentTime
time = ttCurrentTime(newTime)
```

C++ syntax

```
double ttCurrentTime(void)
double ttCurrentTime(double newTime)
```

Arguments

`newTime`	Sets the current time to this time.

Description

This function returns the current time in the simulation, in seconds. When the function is used to set the current time, the old current time is returned.

ttEnterMonitor (T)

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 (the waiting queue is sorted using the priority function in the same way as the ready queue). This will also trigger the suspend hook of the task.

When the task currently holding the monitor exits, the first task in the waiting queue will be moved to the ready queue and is now holding the monitor. Execution will then resume in the segment after the call to `ttEnterMonitor`. To ensure that no further instructions are executed in the case that `ttEnterMonitor` fails, it needs to be called from its own segment (since all code of a TRUETIME segment is executed at once before scheduling decisions are made). See the example below.

To avoid *priority inversion*, standard *priority inheritance* is used if a task tries to enter a monitor currently held by a lower priority task.

Example

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

switch seg,

  case 1,
    ttEnterMonitor('mutex');
    exectime = 0;
  case 2,
    criticalOperation;
    exectime = 0.001;
  case 3,
    ttExitMonitor('mutex');
    exectime = -1;
end
```

See Also

`ttCreateMonitor`, `ttExitMonitor`

ttExitMonitor (T)

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. To ensure that no further instructions are executed in the case that a context switch should occur when the monitor is released, `ttExitMonitor` needs to be called from its own segment (since all code of a TRUETIME segment is executed at once before scheduling decisions are made).

Example

See `ttEnterMonitor`.

See Also

`ttCreateMonitor`, `ttEnterMonitor`

ttGetMsg (TH)

Purpose

Get a message that has been received over a network. This function is used both for wired and wireless networks.

Matlab syntax

```
[msg, signalPower] = ttGetMsg
[msg, signalPower] = ttGetMsg(network)
```

C++ syntax

```
void *ttGetMsg(void)
void *ttGetMsg(int network)
void *ttGetMsg(int network, double *signalPower)
```

Arguments

`network`	The network interface from which the message should be received. The default network number is 1.
`signalPower`	The value of the received signal power corresponding to this message. Only used in the wireless network.

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

Example

```
% Task that waits for and reads messages
function [exectime, data] = receiver(seg, data)
switch seg,
  case 1,
    ttWait('message');
    exectime = 0;
  case 2,
    msg = ttGetMsg;
    disp('I got a message!');
    exectime = 0.001;
  case 3,
    ttSetNextSegment(1); % loop back and wait for new message
    exectime = 0;
end
% Interrupt handler that is called by the network interface
function [exectime, data] = msgRcvhandler(seg, data)
ttNotifyAll('message');
exectime = -1;
```

See Also

`ttInitNetwork`, `ttSendMsg`

ttGetX (ITH)

Purpose

Get a specific 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. Below follow some special notes on the individual functions:

ttGetRelease: Returns the time when the current task job was released. An error will occur if the task has no running job.

ttGetDeadline: Returns the relative deadline of the task.

ttGetAbsDeadline: Returns the absolute deadline of the current task job. An error will occur if the task has no running job.

ttGetPriority: Returns the assigned base priority of the task.

ttGetPeriod: Returns the period of a periodic task. An error will occur if the task is not periodic.

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

ttGetBudget: Returns the remaining execution time budget of the current task job. 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. The execution time budget is reset to the worst-case execution time at the start of each task job.

See Also

`ttSetX`

ttInitKernel (I)

Purpose

Initialize the TRUETIME kernel.

Matlab syntax

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

C++ syntax

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

Arguments

`nbrInp`	Number of input channels, i.e. the size of the A/D port of the kernel block.
`nbrOutp`	Number of output channels, i.e. the size of the D/A port of the kernel block.
`prioFcn`	The scheduling policy used by the kernel.
`cs_oh`	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 kernel 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 constants in the C++ case are; `FP`, `RM`, `DM`, and `EDF`. To define an arbitrary priority function and attach it to the kernel, see `ttAttachPrioFcn`.

See Also

`ttAttachPrioFcn`

ttInitNetwork (I)

Purpose

Initialize the TRUETIME network interface. If the kernel should be connected to several networks, this function must be called several times. This function is used both for wired and wireless networks.

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 corresponding 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`

ttInvokingTask (H)

Purpose

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

Matlab syntax

```
task = ttInvokingTask
```

C++ syntax

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

Description

This function returns the name of the task that has invoked an overrun handler. This facilitates the use of generic code functions for interrupt handlers associated with task overruns (deadline, WCET). In the cases when the interrupt was generated externally or by the expiry of a timer, this function returns NULL (C++) or an empty matrix (MATLAB). See the overrun example in `$DIR/examples/overrun`.

See Also

`ttAttachDLHandler`, `ttAttachWCETHandler`

ttKillJob (TH)

Purpose

Kill the current 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 current active job of a task. The finish hook of the task will be called as the job is killed. If there exist pending jobs for the task that should be released, the first job in the queue will be scheduled for execution and the release hook will be called.

See Also

`ttCreateJob`

ttLogNow (T)

Purpose

Log the current time in a user-controlled log.

Matlab syntax

```
ttLogNow(logID)
```

C++ syntax

```
void ttLogNow(int logID)
```

Arguments

`logID`	The identifier of the user-controlled log.

Description

This function is used to write the current time in user-controlled logs, i.e., logs that have been created using `ttCreateLog` and `logtype = 6`. `logID` should be a number between and one and five that identifies which log to write to. The logs are numbered in order of creation.

See Also

`ttCreateLog`, `ttLogStart`, `ttLogStop`

ttLogStart (T)

Purpose

Start a timing measurement in a user-controlled log.

Matlab syntax

```
ttLogStart(logID)
```

C++ syntax

```
void ttLogStart(int logID)
```

Arguments

`logID`	The identifier of the user-controlled log.

Description

This function is used to start timing measurements in user-controlled logs, i.e., logs that have been created using `ttCreateLog` and `logtype = 6`. `logID` should be a number between and one and five that identifies which log to write to. The logs are numbered in order of creation. Note that nothing is written in the log until a subsequent call to `ttLogStop`.

Example

See the example in the description of `ttCreateLog` that shows how to use `ttLogStart` and `ttLogStop` to log input-output latency in a code function.

See Also

`ttCreateLog`, `ttLogStop`, `ttLogNow`

ttLogStop (T)

Purpose

Stop a timing measurement in a user-controlled log.

Matlab syntax

```
ttLogStop(logID)
```

C++ syntax

```
void ttLogStop(int logID)
```

Arguments

`logID`	The identifier of the user-controlled log.

Description

This function is used to stop timing measurements in user-controlled logs, i.e., logs that have been created using `ttCreateLog` and `logtype = 6`. `logID` should be a number between and one and five that identifies which log to write to. The logs are numbered in order of creation. When this function is called, the difference between the current time and the time of the associated `ttLogStart` will be written in the log.

Example

See the example in the description of `ttCreateLog` that shows how to use `ttLogStart` and `ttLogStop` to log input-output latency in a code function.

See Also

`ttCreateLog`, `ttLogStart`, `ttLogNow`

ttNonPreemptable (I)

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.

ttNoSchedule (I)

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.

ttNotify (TH)

Purpose

Notify the highest-priority task waiting for an event.

Matlab syntax

```
ttNotify(eventname)
```

C++ syntax

```
void ttNotify(char *eventname)
```

Arguments

`eventname`	Name of an event.

Description

This function is used to notify the first task in the waiting queue associated with an event. The waiting queue is sorted according to the priority function of the kernel (in the same way as the ready queue). If the event is associated with a monitor, `ttNotify` must be performed inside a `ttEnterMonitor-ttExitMonitor` construct. The highest-priority waiting task will be moved to the waiting queue of the associated monitor, or directly to the ready queue if it is a free event.

See Also

`ttCreateEvent`, `ttWait`, `ttNotifyAll`

ttNotifyAll (TH)

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, `ttNotifyAll` 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 directly to the ready queue if it is a free event.

See Also

`ttCreateEvent`, `ttWait`, `ttNotify`

ttRemoveTimer (TH)

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 job.

See Also

`ttCreateTimer`, `ttCreatePeriodicTimer`

ttSendMsg (TH)

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. Specify receiver number 0 to broadcast a message to all nodes in the network.
`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`

ttSetKernelParameter (ITH)

Purpose

Set a specific kernel parameter.

Matlab syntax

```
ttSetKernelParameter(parameter, value)
```

C++ syntax

```
void ttSetKernelParameter(char* parameter, double value)
```

Arguments

`parameter`	The name of the parameter to be changed.
`value`	The new value of the parameter.

Description

This function makes it possible to change kernel parameters on the fly. At the moment the following parameters are supported:

- `cpuscaling`
- `energyconsumption`

The default parameter value is 1 for `cpuscaling`, and 0 for `energyconsumption`. Setting the cpuscaling to 2 will make the kernel block execute everything twice as fast as with 1. This makes it possible to experiment with different Dynamic Voltage Scaling methods. At the same time the energy consumption can be set to a realistic value and connected to a battery block.

ttSetNetworkParameter (ITH)

Purpose

Set a specific network parameter on a per node basis.

Matlab syntax

```
ttSetNetworkParameter(parameter, value)
ttSetNetworkParameter(networkNbr, parameter, value)
```

C++ syntax

```
void ttSetNetworkParameter(char* parameter, double value)
void ttSetNetworkParameter(int networkNbr, char* parameter, double value)
```

Arguments

`parameter`	The name of the parameter to be changed.
`value`	The new value of the parameter.
`networkNbr`	The network interface on which the parameter should be changed. The default network number is 1.

Description

This function makes it possible to change network parameters on a per node basis. At the moment the following parameters are supported:

- `transmitpower`
- `predelay`
- `postdelay`

The default parameter value is 0 for the predelay and the postdelay. The transmitpower parameter is only valid when using the wireless network and defaults to whatever is set in the block mask.

ttSetNextSegment (TH)

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.

ttSetX (ITH)

Purpose

Set a specific 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 defined 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. Below follow some special notes on the individual functions:

ttSetDeadline: Changing the relative deadline of a task will only affect subsequent task jobs and not the absolute deadline of the currently running task job. 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 job. 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. An error will occur if the task has no running job.

ttSetPriority: Priority values for tasks should be positive. 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 jobs are created at times h_1, $2h_1$, $3h_1$, etc. If the call is executed at time $h_1 + \tau$, new task jobs 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. An error will occur if the task is not periodic.

ttSetWCET: Changes the worst-case execution time of the task. Each new task job 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 job.

ttSetBudget: This call is used to dynamically change the execution time budget of a running task job. When a task job 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`

ttSleep (TH)

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`

ttSleepUntil (TH)

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.

See Also

`ttSleep`

ttTryFetch (TH)

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 matrix (MATLAB).

See Also

`ttCreateMailbox`, `ttTryPost`

ttTryPost (TH)

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`

ttWait (TH)

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 (the waiting queue is sorted using the priority function in the same way as the ready queue). When the task is later notified, it will be moved to the waiting queue of the associated monitor, or directly to the ready queue if it is a free event. A call to this function will cause the suspend-hook of the task to be executed.

Example

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
```

The event above may, e.g., be notified from an interrupt handler associated with an external interrupt channel or the network receive channel of the kernel block.

See Also

`ttCreateEvent`, `ttNotify`, `ttNotifyAll`

14. References

Åström, K. J. and T. Hägglund (1995): *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, Research Triangle Park, North Carolina.

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.

Henriksson, D., A. Cervin, and K.-E. Årzén (2003): "TrueTime: Real-time control system simulation with MATLAB/Simulink." In *Proceedings of the Nordic MATLAB Conference*. Copenhagen, Denmark.

The Mathworks (2001): *Simulink: A Program for Simulating Dynamic Systems – User's Guide*. The MathWorks Inc., Natick, MA.