Contents

1. Introduction .................................................. 7
2. Getting Started .................................................... 7
   2.1 Software Requirements .................................... 7
   2.2 Installation ............................................... 8
   2.3 Compilation ............................................... 8
3. Using the Simulator ............................................. 9
4. Writing Code Functions ......................................... 9
   4.1 Writing a Matlab Code Function .......................... 9
   4.2 Writing a C++ Code Function .............................. 10
   4.3 Calling Simulink Block Diagrams ......................... 10
5. Initialization .................................................... 12
   5.1 Writing a Matlab Initialization Script .................. 12
   5.2 Writing a C++ Initialization Script ..................... 13
6. Compilation ..................................................... 15
7. The TrueTime Kernel ............................................ 15
   7.1 Dynamic Voltage Scaling ................................ 15
8. The TrueTime Network ........................................... 16
   8.1 CSMA/CD (Ethernet) ........................................ 17
   8.2 CSMA/AMP (CAN) ......................................... 18
   8.3 Round Robin (Token Bus) ................................ 18
   8.4 FDMA .................................................. 18
   8.5 TDMA (TTP) ................................................ 18
   8.6 Switched Ethernet ....................................... 19
9. The TrueTime Wireless Network ................................. 20
   9.1 802.11b/g (WLAN) ....................................... 21
   9.2 802.15.4 (ZigBee) ..................................... 22
   9.3 Calculation of Error Probabilities ....................... 23
   9.4 User-Defined Path-Loss Function ......................... 24
10. The TrueTime Battery ........................................... 26
11. The TrueTime Standalone Network Blocks ....................... 26
12. Examples ....................................................... 26
   12.1 PID-control of a DC-servo ............................... 27
12.2 Task Scheduling and Control ........................................ 30
12.3 Distributed Control System ........................................ 31
12.4 Deadline Overrun Handling ....................................... 32
12.5 Task Synchronization Using Monitors ............................. 32
12.6 Distributed Wireless Control System ............................ 34
12.7 Control and Coordination of Mobile Motes ....................... 36
12.8 Wireless Ad-hoc Routing Using AODV ........................... 36
12.9 Mote Soccer ............................................................ 39
12.10 ttSendMsg/ttGetMsg stand-alone network blocks ................ 39

13. Kernel Implementation Details ..................................... 40
13.1 Kernel Data Structures ............................................. 40
13.2 Task Model .............................................................. 43
13.3 The Kernel Function .................................................. 44
13.4 Timing ................................................................. 46

14. TrueTime Command Reference .................................... 46
   ttAbortSimulation (TH) .................................................. 49
   ttAnalogIn (TH) .......................................................... 50
   ttAnalogOut (TH) ....................................................... 51
   ttAttachDLHandler (I) .................................................. 52
   ttAttachHook (C++ only) (I) ........................................... 53
   ttAttachWCETHandler (I) ............................................. 54
   ttCallBlockSystem (TH) ................................................ 55
   ttCreateEvent (I) ....................................................... 56
   ttCreateExternalTrigger (I) ......................................... 57
   ttCreateInterruptHandler (I) ....................................... 58
   ttCreateJob (ITH) ...................................................... 59
   ttCreateLog (I) .......................................................... 60
   ttCreateMailbox (I) .................................................... 62
   ttCreateMonitor (I) ..................................................... 63
   ttCreatePeriodicTask (I) .............................................. 64
   ttCreatePeriodicTimer (ITH) ......................................... 65
   ttCreateSemaphore (I) ................................................ 66
   ttCreateTask (I) ......................................................... 67
ttCreateTimer (ITH) ......................................... 68
 ttCurrentTime (ITH) ....................................... 69
 ttDiscardUnsentMessages (IT) .............................. 70
 ttEnterMonitor (T) .......................................... 71
 ttExitMonitor (T) ........................................... 73
 ttFetch (T) .................................................. 74
 ttGetData (TH) .............................................. 75
 ttGetMsg (TH) ............................................... 76
 ttGetInitArg (C++ only) (I) ................................. 77
 ttGetX (ITH) ................................................ 78
 ttGive (TH) .................................................. 80
 ttInitKernel (I) ............................................. 81
 ttInitNetwork (I) ............................................ 82
 ttInvokingTask (H) ......................................... 83
 ttKillJob (TH) ............................................... 84
 ttLogNow (T) ............................................... 85
 ttLogStart (T) .............................................. 86
 ttLogStop (T) ............................................... 87
 ttNonPreemptable (I) .................................... 88
 ttNoSchedule (I) .......................................... 89
 ttNotify (TH) .............................................. 90
 ttNotifyAll (TH) ......................................... 91
 ttPost (T) ................................................... 92
 ttRemoveTimer (TH) ....................................... 93
 ttRetrieve (T) ............................................. 94
 ttSendMsg (TH) ......................................... 95
 ttSetData (TH) ............................................. 96
 ttSetKernelParameter (ITH) ............................ 97
 ttSetNetworkParameter (ITH) .......................... 98
 ttSetNextSegment (TH) ................................... 99
 ttSetX (ITH) ............................................... 100
 ttSleep (T) ................................................. 102
 ttSleepUntil (T) .......................................... 103
15. References
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; Andersson et al., 2005], but be aware that 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.

Ten tutorial examples are provided, treating standard and distributed PID-control, scheduling, overrun handling, synchronization, control over wireless networks, mote coordination, wireless ad-hoc routing using AODV, mobile robot soccer, and the new ttSendMsg/ttGetMsg blocks.

The manual also describes some of the internal workings of TRUETIME, including the task model, implementation details, and timing details. The network blocks and the radio model used for the wireless simulations are also presented in some detail. 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

You can also visit the TRUETIME discussion forum at

http://www.control.lth.se/truetime/forum

2. Getting Started

2.1 Software Requirements

TRUETIME currently supports Matlab 7.x (R14 and later) with Simulink 6.x and Matlab 6.5.1 (R13.1) with Simulink 5.1. Support for Matlab 6.1 (R12.1) with Simulink 4.1 was dropped in TRUETIME 1.3.

A C++ compiler is required to run TRUETIME in the C++ version. For the Matlab version, pre-compiled files are provided in the archive that is 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
2.2 Installation
Download and extract the compressed archive (trtrue-time-1.5.zip), available at http://www.control.lth.se/truetime

Extracting the file creates a truetime-1.5 directory, which will be referred to as $DIR in the sequel.

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;

Starting Matlab and issuing the command

>> truetime

from the Matlab prompt will now open the TRUETime block library, see Figure 1.

2.3 Compilation
Since the TRUETime archive contains pre-compiled files, no compilation is required to run TRUETime with the M-file API.

However, TRUETime also supports simulations written in C++ code, which then must be compiled. In this case, you first need to configure your C++ compiler in Matlab. This can be done by issuing the command
In the setup, make sure that you change from the Matlab default compiler to a proper C++ compiler. For more detailed instructions on how to compile individual simulations, see Section 6 in this manual.

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

As mentioned above, the initialization code and the code that is executed during simulation may be written either as Matlab M-files or as C++ code (for increased simulation speed). 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](image_url)  
**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 shown 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.

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

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.

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

In both the C++ and m-file cases, 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 1

Example of a P-controller code function written in Matlab code.

```matlab
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
    }
    return FINISHED; // to supress compiler warnings
}

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.

Listing 3  Simulink block diagrams are called from within code function using the TRUE-TIME 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. The period of the task is given by the init argument of the TrueTime Kernel block dialogue. (Note that the init argument may be an arbitrary Matlab struct.) This task uses the code function Pcontroller that was defined in Listing 1. See the command reference for further explanation of the functions.

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 mymodel)

    >> set_param('mymodel', 'InitFcn', 'clear 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.

```matlab
function example_init(initarg)
    ttInitKernel(2, 1, 'prioFP');
    data.u = 0;
    data.K = 2;
    offset = 0;
    period = initarg;
    prio = 5;
    ttCreatePeriodicTask('ctrl', offset, period, prio, 'Pcontroller', data);
```
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.

```c
#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
}
```

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. The initial argument supplied in the TRUETIME Kernel block dialogue can be retrieved using the function ttGetInitArg. The C++ version of the Matlab initialization script of Listing 4 is given in Listing 6.

6. Compilation
Compilation requires that a proper C++ compiler has been configured in Matlab as described in Section 2.3. If needed, the TRUETIME kernel may be re-compiled by issuing the command

```matlab
>> make_truetime
```

from the Matlab prompt. This script compiles the kernel and network S-functions and the MEX-files for the TRUETIME primitives.

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

```matlab
>> ttmex example_init.cpp
```
Listing 6  Example of a TrueTime initialization script in the C++ version. Corresponds to the Matlab version from Listing 4.

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

#include "Pcontroller.cpp"  // P-controller code function

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

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

void init() {
    mxArray *initarg = ttGetInitArg();
    ttInitKernel(2, 1, prioFP);
    data = new Task_Data;
    data->u = 0.0;
    data->K = 2.0;
    double offset = 0.0;
    double period = mxGetPr(initarg)[0];
    double prio = 5.0;
    ttCreatePeriodicTask("ctrl", offset, period, prio, Pcontroller, data);
}

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

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

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

7. The TrueTime Kernel

The kernel block is configured through the block mask dialog, see Figure 5, with the following parameters (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, 0.01 if the local time should run 1% faster than the nominal time (the actual simulation time).

Clock offset  A constant time offset from the nominal 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, see Section 11.

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. Using the command ttSetNetworkParameter, see Section 14, it is also possible to change some parameters on a per node basis. 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.

**Minimum frame size (bits)** 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 512 bits.

**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 backoff for a time defined by

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

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

\[ t_{\text{idle}} = \text{minimum frame size} / \text{data rate}, \]

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 \((\text{allocated bandwidth} \times \text{data rate})\).
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 (bits)** The size of a sending slot. The slot time is hence given by

\[ t_{\text{slot}} = \frac{\text{slot size}}{\text{data rate}}. \]

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

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 (bits)** 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.
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). Some more technical details about the wireless network can be found in [Ohlin, 2006]. The radio model used includes support for:

- Ad-hoc wireless networks.
- Isotropic antenna.
- Inability to send and receive messages at the same time.
- Path loss of radio signals modeled as $\frac{1}{d^a}$ where $d$ is the distance in meters and $a$ is a parameter chosen 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 type** Determines the MAC protocol to be used. Can be either 802.11b/g (WLAN) or 802.15.4 (ZigBee).
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 Snr, Rcv and Schedule input and outputs of the block.

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

Minimum frame size (bits) 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 bits, 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.

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 in 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 the percentage of
bit errors 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 $\mu$s. 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. Initialize:
   \[ NB = 0 \]
   \[ BE = macMinBE \]

2. Delay for a random number of backoff periods in the interval \([0, 2^{BE} - 1]\)

3. Is the medium idle?
   if yes: send
   else: goto 4

4. Update the backoff counters:
   \[ NB = NB + 1 \]
   \[ BE = \min(BE+1, aMaxBE) \]

5. Is \( NB > \text{macMaxCSMABackoffs} \)?
   if yes: drop the packet
   else: goto 2

The variable names are taken from the standard to make comparisons easier. A small explanation of their names is provided below.

- **NB** Number of backoffs.
- **BE** Backoff exponent.
- **macMinBE** The minimum value of the backoff exponent in the CSMA/CA algorithm. The default value is 3.
**aMaxBE** The maximum value of the backoff exponent in the CSMA/CA algorithm. The default value is 5.

**macMaxCSMABackoffs** The maximum number of backoffs the CSMA/CA algorithm will attempt before declaring a channel access failure. The default value is 4.

### 9.3 Calculation of Error Probabilities

During the calculation of error probabilities, it is for simplicity assumed that BPSK\(^1\) is always used in the transmissions. This is of course an approximation, but it relates well to reality.

Assume that a symbol is sent, in our case this is a bit, i.e., a 0 or a 1. Additive white Gaussian noise gives a probability density function for the received symbol, that for some signal-to-noise ratio may look like Figure 8. A threshold is then used to decide if the received symbol is a 0 or a 1. The decision threshold is marked as a line in the middle of the figure. The darker area to the left of the threshold gives the probability of a symbol error. A higher signal to noise ratio translates the curve to the right, making the probability of error smaller.

The above standard procedure should ideally be performed for every bit in the message. The total number of calculated bit errors should then be compared to the error coding threshold. This is, however, not done, because it would be very computationally expensive. Instead, the maximum noise level during the transmission is saved, and used to calculate the worst case SNR. By assuming that bit errors in a message are uncorrelated, it is deduced that the number of bit errors, \(X\), belongs to a binomial distribution \(X \in Bin(n, p)\), where \(n\) is the number of bits in the message, and \(p\) is the probability that a certain bit is erroneous. If the value of \(n\) is large, the binomial distribution can be approximated with a normal distribution, using the central limit theorem. This gives that \(X \in N(np, \sqrt{npq})\) where \(q = 1 - p\). What we are really interested in is the probability that \(bn\), where \(b\) is the error coding threshold, is larger than the total number of bit errors in a message. This probability is calculated using

\[
P(X \leq bn) = \begin{cases} 
\Phi\left(\frac{bn - np}{\sqrt{npq}}\right) & \text{if } bn - np > 0 \\
1 - \Phi\left(\frac{|bn - np|}{\sqrt{npq}}\right) & \text{if } bn - np \leq 0 
\end{cases}
\]

where \(\Phi\) is the standard normal cumulative distribution function.

**Example** Assume that a message consists of 100 bits, i.e., \(n = 100\). The probability that a certain bit is erroneous has been calculated to 0.1 using the above method, i.e., \(p = 0.1\) and \(q = 1 - p = 0.9\). The error coding threshold has been set to 5%, i.e., \(b = 0.05\). Then the probability that we can decode the complete message is

\[
P(X \leq bn) = 1 - \Phi\left(\frac{|bn - np|}{\sqrt{npq}}\right) = 1 - \Phi\left(\frac{5}{\sqrt{9}}\right) \approx 0.0478
\]

\(^1\)Binary Phase Shift Keying (BPSK) is a means of transmitting symbols by altering the phase of a reference signal. It uses two phases separated by 180° and is hence binary.
9.4 User-Defined Path-Loss Function

The default path-loss function (or propagation model) used in the TrueTime wireless simulations is

\[ P_{receiver} = \frac{1}{d^\alpha} P_{sender} \]

where \( P \) is the power, \( d \) is the distance in meters, and \( \alpha \) is a parameter that can be chosen to model different environments. This model is often used in simulations, but in some cases it can be advantageous to use other models. Therefore, TrueTime has the possibility to register a user-defined path-loss function. The function is written as a Matlab m-file and can therefore take advantage of all the built-in functions available in Matlab or Simulink. In particular, this includes the possibility to use persistent variables, i.e., variables which are retained in memory between calls to the function. This function can, for example, be used to model a Rayleigh\(^2\) fading or blocking of radio signals to and from certain points in the environment. At the moment, nodes in the TrueTime framework only have \( x \) and \( y \) coordinates, but if a direction was to be introduced this function could also be used to model directive effects in the antenna behaviour.

The Matlab function takes the following arguments

- Transmission power
- Name of the sending node
- \( x \) and \( y \) coordinates of the sending node

\(^2\)In a Rayleigh fading, the relative speed of two nodes and the number of multiple paths that the signal takes from the sender to the receiver is taken into account. See Figure 9 for an example.
Figure 9  Example of a Rayleigh fading using a radio frequency of 2.4 MHz. Nodes moving with a relative speed of 6 km/h, 1 second sampling interval and 10 random phasers.

- Name of the receiving node
- $x$ and $y$ coordinates of the receiving node
- Current simulation time

and returns the signal power in the receiving node.

A small example showing the structure of how a Rayleigh fading could be implemented can be seen in Listing 7.

Listing 7  Example of path-loss function modeling Rayleigh fading.

```matlab
function [power]=rayleigh(transmitPower, node1, x1, y1, node2, x2, y2, time)

% Calculate the exponential pathloss
distance = sqrt((x1 - x2)^2 + (y1 - y2)^2);
power = transmitPower/(distance+1)^3.5;

% Kalman filter to get the relative velocity of the two nodes
velocity = kalman_velocity(node1, x1, y1, node2, x2, y2, time);

% Calculate the rayleigh fading
factor = calculate_rayleigh(node1, node2, velocity, time);

% Add the rayleigh fading to the exponential path loss
power = power * factor;
```

24
10. The TrueTime Battery

The battery block 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 the example in Section 12.6 below for more details on the use of TrueTime batteries.

11. The TrueTime Standalone Network Blocks

The standalone network blocks, named $ttSendMsg$ and $ttGetMsg$, as seen in Figure 1, can be used to send messages using the network blocks without using kernel blocks. This makes it possible to create TrueTime network simulations without having to initialize kernels, create and install interrupt handlers, etc. It is in other words possible to create a whole network simulation in Simulink without any M-files at all. It is also possible to mix these blocks with kernel blocks, so that some stations use the standalone network blocks, while others use the standard $ttSend$ and $ttGetMsg$ primitives from within a code function executing in a kernel block.

The standalone network blocks are configured through the block mask dialogs, seen in Figure 10. The parameters are the same that are used in the $ttSendMsg$ and $ttGetMsg$ primitives. The $ttSendMsg$ block has a Simulink trigger input port, which can be configured to trigger on raising, falling or either flanks. The $ttGetMsg$ block has an optional trigger output port. The value of the trigger output switches back and forth between 0 and 1 as messages are received. This port can be used to trigger something that should be executed after a new message has been received.

12. Examples

The directory $DIR/examples$ contains ten examples:

- **simple_pid**: Four different ways of implementing a simple periodic controller are demonstrated. Both M-file and C++ implementations are provided.
- **threeservos**: Illustrates task scheduling and control, where three periodic control tasks execute on the same CPU. Both M-file and C++ implementations are provided.
- **distributed**: Demonstrates distributed (networked) control involving four computer nodes and the wired network block. Both M-file and C++ implementations are provided.
- **overrun**: Demonstrates the use of deadline overrun handlers for flexible scheduling. M-file implementation only.
Illustrates how to use monitors and events to simulate mutual exclusive access to shared variables. M-file implementation only.

Demonstrates simulation and animation of collaborating mobile robots communicating over a wireless network. M-file implementation only.

Demonstrates wireless networked control with transmission power control to minimize the energy consumption. M-file implementation only.

Illustrates how higher-layer communication protocols such as AODV routing can be implemented in TRUETime. M-file implementation only.

A larger example showing simulation and animation of ten mobile robots playing soccer. M-file implementation only.

Demonstrates how the new ttSendMsg/ttGetMsg blocks can be used to simulate time-triggered or event-triggered networked control loops without the need for kernel blocks.

The first three 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.

12.1 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

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

(1)

The PID-controller is implemented according to the following equations

\[ P(k) = K \cdot (\beta r(k) - y(k)) \]
\[ I(k + 1) = I(k) + \frac{K_h}{T_i}(r(k) - y(k)) \]
\[ D(k) = a_d D(k - 1) + b_d (y(k - 1) - y(k)) \]
\[ u(k) = P(k) + I(k) + D(k) \]

(2)

where \( a_d = \frac{T_d}{N_h + T_d} \) and \( b_d = \frac{NKT_d}{N_h + T_d} \), see [Åström and Hägglund, 1995]. The controller parameters were chosen to give the system a closed-loop bandwidth of \( \omega_c = 20 \text{ rad/s} \) and a relative damping of \( \zeta = 0.7 \).
Figure 10  The dialog of the ttGetMsg block to the left, and the dialog of the ttSendMsg block to the right.

**Simulation Files** The initialization script (servo_init.m) is given in an abbreviated version in Listing 8. 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 pidcode2.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 pidcode3.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 sampler code.m and pidcode4.m, respectively.
Simulations The Simulink model is called servo.mdl and is given in Figure 11. 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.

Listing 8 The initialization script for the PID-control example.

```matlab
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 1: using the built-in support for periodic tasks
        %
        ttCreatePeriodicTask('pid_task', offset, period, prio, 'pidcode1', data);
    case 2,
        % IMPLEMENTATION 2: calling Simulink block within code function
        %
        data2.u = 0;
        ttCreatePeriodicTask('pid_task', offset, period, prio, 'pidcode2', data2);
    case 3,
        % IMPLEMENTATION 3: sleepUntil and loop back
        data.t = 0;
        ttCreateTask('pid_task', deadline, prio, 'pidcode3', data);
        ttCreateJob('pid_task');
    case 4,
        % IMPLEMENTATION 4: 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, 'pidcode4', data);
    end
```
• 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, pidcode2.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, pidcode3.m.

• Change to implementation 4 and run a simulation. Study the code functions, samplercode.m and pidcode4.m. Notice the inclusion of the handler in the schedule plot.

12.2 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
• 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.

12.3 Distributed Control System

Introduction This example simulates distributed (networked) 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`.

Simulations The Simulink model is called `distributed.mdl` and is given in Figure 12. 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.

### 12.4 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 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.

### 12.5 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
Process and Controller  The ball and beam laboratory process is shown in Figure 13. 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 $\phi$, and the ball position on the beam, denoted by $x$. A linearized model of the system is given by

$$G(s) = G_\phi(s)G_x(s)$$  \hspace{1cm} (3)

where

$$G_\phi(s) = \frac{k_\phi}{s}$$  \hspace{1cm} (4)

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

$$G_x(s) = -\frac{k_x}{s^2}$$  \hspace{1cm} (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 14. 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 15. 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 $\phi_r$ in Figure 14). 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.

12.6 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.

- 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.
12.7 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 16. 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

12.8 Wireless Ad-hoc Routing Using AODV

Introduction The TrueTime wireless network simulates communication in an ad-hoc network, i.e., no centralized access point or infrastructure exists to coordinate the traffic across the network. In such networks it is necessary to implement decentralized functionality to be able to route the traffic over the network. This example describes a TrueTime implementation of one such ad-hoc wireless routing protocol.

AODV [Perkins and Royer, 1999] stands for Ad-hoc On-Demand Distance Vector routing and contrary to most routing mechanisms, it does not rely on periodic
transmission of routing messages between the nodes. Instead, routes are created on-demand, i.e., only when actually needed to send traffic between a source and a destination node. This leads to a substantial decrease in the amount of network bandwidth consumed to establish routes. Below follows a brief description of the functionality of AODV. For a complete definition of the AODV protocol, see [Perkins and Royer, 2003].

AODV uses three basic types of control messages in order to build and invalidate routes: route request (RREQ), route reply (RREP), and route error (RERR) messages. These control messages contain source and destination sequence numbers, which are used to ensure fresh and loop-free routes.

A node that requires a route to a destination node initiates route discovery by broadcasting an RREQ message to its neighbors. A node receiving an RREQ starts by updating its routing information backwards towards the source. If the same RREQ has not been received before, the node then checks its routing table for a route to the destination. If a route exists with a sequence number greater than or equal to that contained in the RREQ, an RREP message is sent back towards the source. Otherwise, the node rebroadcasts the RREQ. When an RREP has propagated back to the original source node, the established route may be used to send data. Periodic hello messages are used to maintain local connectivity information between neighboring nodes. A node that detects a link break will check its routing table to find all routes which use the broken link as the next hop. In order to propagate the information about the broken link, an RERR message is then sent to each node that constitute a previous hop on any of these routes.

The files for this tutorial example are found in the directory $DIR/examples/AODV. All nodes are initialized using the same initialization script, node_init.m. Two TRUETime tasks are created in each node to handle AODV send and receive actions, respectively. The AODV send task is activated from the application code as a data message should be sent to another node in the network. The AODV receive task handles incoming AODV control messages and forwarding of data messages. Communication between the application layer and the AODV layer is handled using TRUETime mailboxes.

The AODV send task (AODVsendcode.m) operates according to the following code:

```
IF (data message received from application)
  check the routing table for a route to the destination;
  IF (a valid route exists)
    forward data message to next hop on route;
    update expiry time of route entry;
  ELSE
    initiate route discovery by broadcasting RREQ message;
    buffer data message until route has been established;
  END
ELSE IF (notified of established new route)
  send all buffered data messages to destination
END
```

The AODV receive task (AODVrcvcode.m) performs the following:
IF (receiving data message)
    update expiry timer for reverse route entry to source;
IF (this node is the destination)
    Pass data message to application;
ELSE
    forward data message to next hop on route;
    update expiry timer of route entry;
END
ELSE
    SWITCH message_type
    CASE RREQ:
        IF (first time this RREQ is received)
            enter RREQ in cache;
            create or update route entry to source;
            check the routing table for a route to the destination;
            IF (a route exists)
                send RREP message back towards source;
            ELSE
                update and rebroadcast the RREQ;
            END
        END
    CASE RREP:
        check the routing table for a route to the destination;
        IF (no route exists)
            create route entry to destination;
        ELSE IF (route entry exists but should be updated)
            update route entry to destination;
        END
        IF (this node is the original source)
            notify the AODV send task about the new route;
        ELSE IF (route to destination was created or updated)
            update reverse route entry towards source;
            propagate RREP to next hop towards source;
        END
    CASE RERR:
        find and invalidate all affected route entries;
        propagate the RERR to all previous hops on the routes;
    END
END

Each node also contains a periodic task (hellocode.m), responsible for broadcasting hello messages and determine local connectivity based on hello messages received from neighboring nodes. Finally, each node has a task to handle timer expiry of route entries (expcode.m).

**Simulations** Open the model AODV.mdl to run the simulation.

- The simulation example consists of seven nodes. Choose the option Update Diagram in the Edit menu to bring up an animation window of the simulation. This will show the original positions of the seven nodes and their respective signal reach.
• Run a simulation. In the simulation scenario, the left-most node (node 1) sends data periodically to node 7 with period 0.5. The initial route that is established is 1 → 3 → 5 → 7. At time t = 3, node 5 starts to move which eventually leads to the route breaking. At time t = 10, node 6 repairs the route by moving in between node 4 and 7. The printouts in the Matlab command window describe the actions in the AODV layer in more detail. Also study the global variable routing_table.

• Open and examine the file initsim.m. This file initializes the global variables used in the simulation, e.g., the routing table and the node positions. By changing the variable verbose from 0 to 1 even more detailed AODV information will be displayed when the simulation is run.

• The global variables sent and received show the data that is sent (by node 1) and received (by node 7) in the simulation. Examine the lengths of these vectors to determine how many messages that where lost due to the delay in detecting and propagating the information about the broken link back to the source node. (Answer: The messages sent at times 8.0002, 8.5002, and 9.0002 are lost.)

• The hello interval determines who fast the network will respond to broken links (and also the bandwidth overhead). Try changing the AODV parameter HELLO_INTERVAL (both in initsim.m and node_init.m) to decrease the number of lost data messages. In this case only two messages are lost.

12.9 Mote Soccer
This largely undocumented example features a slightly more complex simulation set-up where ten mobile robots are playing soccer. The robots and the ball are animated during the simulation, see Figure 17. The simulation files are found in the directory $DIR/examples/soccer.

Each mobile robot (mote) is modeled using a kernel block and two integrators representing the x and y coordinates of the robot. The ball is modeled as a forth-order system (an integrator plus damping in each direction) implemented in an S-function (ballmotion.m). This S-function also handles the player interaction with the ball. The robots within each team communicate over the wireless network. The goalkeeper is each team acts as the “master” and coordinates the players.

The mask of the network block is programmed to dynamically change the contents of the underlying subsystem (study the initialization commands). This is a very convenient feature when working with a large number of network nodes.

12.10 ttSendMsg/ttGetMsg stand-alone network blocks
The two examples timetriggered.mdl and eventtriggered.mdl in the directory $DIR/examples/ttsendmsg demonstrate how the new stand-alone network blocks can be used to simulate networked control loops, see Figure 18.

In the time-triggered example, the send blocks are triggered by pulse generators with a period of 0.1 s. The pulse generator of the second node has an offset (phase delay) of 0.05 s. A simple P-controller is used to stabilize the integrator which is disturbed by white noise.

In the event-triggered example, the send block at the process is triggered whenever the magnitude of the process output exceeds 0.25. The controller at the
receiver is also event-triggered, and in turn triggers the second send block with a delay of 0.001 s. The control signal at the second receiver (at the process) is held between updates. It is seen that this set-up gives almost as good performance as the time-triggered scheme, while the network utilization is much lower.

13. 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.

13.1 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:

```cpp
class RTsys {
public:
    double time; // Current time in simulation
    double* inputs; // Vector of input port values
```
Figure 18  Event-triggered networked control loop using the new \ttSendMsg/\ttGetMsg stand-alone network blocks.

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 lists. 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 \texttt{prioFcn}, which is a function that returns a (possibly dynamic) priority number from a Task instance, see the description of \texttt{ttInitKernel} in the command reference.

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

```cpp
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
};
```
The execution time of the running task is updated each time the kernel executes, see Listing 9. 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.

13.2 Task Model

TRUE_TIME 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

- 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 the tutorial example in Section 12.4 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 19. 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.

### 13.3 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 13.4 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 9. This code focuses on user tasks. See $DIR/kernel/ttkernel.cpp for the complete implementation.
Listing 9  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;
}
13.4 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:

```c
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

```c
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 (< $10^{-15}$ s).

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).

14. TrueTime Command Reference

The available TrueTime commands are summarized in Tables 2–4, 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.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttInitKernel</td>
<td>Initialize the kernel.</td>
</tr>
<tr>
<td>ttInitNetwork</td>
<td>Initialize the network interface.</td>
</tr>
<tr>
<td>ttGetInitArg (C++ only)</td>
<td>Retrieve the initialization argument.</td>
</tr>
<tr>
<td>ttCreateTask</td>
<td>Create a task.</td>
</tr>
<tr>
<td>ttCreatePeriodicTask</td>
<td>Create a periodic task.</td>
</tr>
<tr>
<td>ttCreateInterruptHandler</td>
<td>Create an interrupt handler.</td>
</tr>
<tr>
<td>ttCreateExternalTrigger</td>
<td>Associate a interrupt handler with an external interrupt channel.</td>
</tr>
<tr>
<td>ttCreateMonitor</td>
<td>Create a monitor.</td>
</tr>
<tr>
<td>ttCreateEvent</td>
<td>Create an event.</td>
</tr>
<tr>
<td>ttCreateLog</td>
<td>Create a log structure and specify data to log.</td>
</tr>
<tr>
<td>ttCreateMailbox</td>
<td>Create a mailbox for inter-task communication.</td>
</tr>
<tr>
<td>ttCreateSemaphore</td>
<td>Create a counting semaphore.</td>
</tr>
<tr>
<td>ttNoSchedule</td>
<td>Switch off the schedule generation for a specific task or interrupt handler.</td>
</tr>
<tr>
<td>ttNonPreemptable</td>
<td>Make a task non-preemptable.</td>
</tr>
<tr>
<td>ttAttachDLHandler</td>
<td>Attach a deadline overrun handler to a task.</td>
</tr>
<tr>
<td>ttAttachWCETHandler</td>
<td>Attach a worst-case execution time overrun handler to a task.</td>
</tr>
<tr>
<td>ttAttachHook (C++ only)</td>
<td>Attach a run-time hook to a task.</td>
</tr>
<tr>
<td>ttAbortSimulation</td>
<td>Abort the simulation.</td>
</tr>
</tbody>
</table>

**Table 2**  Commands used to create and initialize TrueTime objects, and to control the simulation.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttSetDeadline</td>
<td>Set the relative deadline of a task.</td>
</tr>
<tr>
<td>ttSetAbsDeadline</td>
<td>Set the absolute deadline of a task job.</td>
</tr>
<tr>
<td>ttSetPriority</td>
<td>Set the priority of a task.</td>
</tr>
<tr>
<td>ttSetPeriod</td>
<td>Set the period of a periodic task.</td>
</tr>
<tr>
<td>ttSetBudget</td>
<td>Set the execution time budget of a task job.</td>
</tr>
<tr>
<td>ttSetWCET</td>
<td>Set the worst-case execution time of a task.</td>
</tr>
<tr>
<td>ttSetData</td>
<td>Update the local memory data structure of a task.</td>
</tr>
<tr>
<td>ttGetRelease</td>
<td>Get the release time of a task job.</td>
</tr>
<tr>
<td>ttGetDeadline</td>
<td>Get the relative deadline of a task.</td>
</tr>
<tr>
<td>ttGetAbsDeadline</td>
<td>Get the absolute deadline of a task job.</td>
</tr>
<tr>
<td>ttGetPriority</td>
<td>Get the priority of a task.</td>
</tr>
<tr>
<td>ttGetPeriod</td>
<td>Get the period of a periodic task.</td>
</tr>
<tr>
<td>ttGetBudget</td>
<td>Get the execution time budget of a task job.</td>
</tr>
<tr>
<td>ttGetWCET</td>
<td>Get the worst-case execution time of a task.</td>
</tr>
<tr>
<td>ttGetData</td>
<td>Retrieve the local memory data structure of a task.</td>
</tr>
</tbody>
</table>

**Table 3**  Commands used to set and get task attributes.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ttCreateJob</td>
<td>Create a job of a task.</td>
</tr>
<tr>
<td>ttKillJob</td>
<td>Kill the running job of a task.</td>
</tr>
<tr>
<td>ttEnterMonitor</td>
<td>Attempt to enter a monitor.</td>
</tr>
<tr>
<td>ttExitMonitor</td>
<td>Exit a monitor.</td>
</tr>
<tr>
<td>ttWait</td>
<td>Wait for an event.</td>
</tr>
<tr>
<td>ttNotify</td>
<td>Notify the highest-priority task waiting for an event.</td>
</tr>
<tr>
<td>ttNotifyAll</td>
<td>Notify all tasks waiting for an event.</td>
</tr>
<tr>
<td>ttLogNow</td>
<td>Log the current time.</td>
</tr>
<tr>
<td>ttLogStart</td>
<td>Start a timing measurement for a log.</td>
</tr>
<tr>
<td>ttLogStop</td>
<td>Stop a timing measurement and save in the log.</td>
</tr>
<tr>
<td>ttTryPost</td>
<td>Post a message to a mailbox (non-blocking).</td>
</tr>
<tr>
<td>ttTryFetch</td>
<td>Fetch a message from a mailbox (non-blocking).</td>
</tr>
<tr>
<td>ttPost</td>
<td>Post a message to a mailbox (blocking).</td>
</tr>
<tr>
<td>ttFetch</td>
<td>Fetch a message from a mailbox (blocking).</td>
</tr>
<tr>
<td>ttRetrieve</td>
<td>Read the actual message fetched from a mailbox.</td>
</tr>
<tr>
<td>ttTake</td>
<td>Take a semaphore.</td>
</tr>
<tr>
<td>ttGive</td>
<td>Give a semaphore.</td>
</tr>
<tr>
<td>ttCreateTimer</td>
<td>Create a one-shot timer and associate an interrupt handler with the timer.</td>
</tr>
<tr>
<td>ttCreatePeriodicTimer</td>
<td>Create a periodic timer and associate an interrupt handler with the timer.</td>
</tr>
<tr>
<td>ttRemoveTimer</td>
<td>Remove a specific timer.</td>
</tr>
<tr>
<td>ttCurrentTime</td>
<td>Get and/or set the current time in the simulation on a per node basis.</td>
</tr>
<tr>
<td>ttSleepUntil</td>
<td>Sleep until a certain point in time.</td>
</tr>
<tr>
<td>ttSleep</td>
<td>Sleep for a certain amount of time.</td>
</tr>
<tr>
<td>ttAnalogIn</td>
<td>Read a value from an analog input channel.</td>
</tr>
<tr>
<td>ttAnalogOut</td>
<td>Write a value to an analog output channel.</td>
</tr>
<tr>
<td>ttSetNextSegment</td>
<td>Set the next segment to be executed in the code function (to implement loops and branches).</td>
</tr>
<tr>
<td>ttInvokingTask</td>
<td>Get the name of the task that invoked a task overrun handler.</td>
</tr>
<tr>
<td>ttCallBlockSystem</td>
<td>Call a Simulink block diagram from within a code function.</td>
</tr>
<tr>
<td>ttSendMsg</td>
<td>Send a message over a TRUE TIME network.</td>
</tr>
<tr>
<td>ttGetMsg</td>
<td>Get a message that has been received over a TRUE TIME network.</td>
</tr>
<tr>
<td>ttDiscardUnsentMessages</td>
<td>Delete any unsent messages.</td>
</tr>
<tr>
<td>ttSetNetworkParameter</td>
<td>Set a specific network parameter on a per node basis.</td>
</tr>
<tr>
<td>ttSetKernelParameter</td>
<td>Set a specific kernel parameter on a per node basis.</td>
</tr>
</tbody>
</table>

**Table 4**  Real-time primitives.
ttAbortSimulation (TH)

Purpose
Stop the current simulation, causing an error.

Matlab syntax
```
   ttAbortSimulation
```

C++ syntax
```
   ttAbortSimulation()
```

Description
This function is used to abort a simulation before the simulation stop-time has been reached. For instance, after a certain condition has occurred, it may be pointless to continue the simulation.

If you run repeated simulations in a script, since this primitive generates an error, it may be useful to start the simulation from within a try statement:

```
for i=1:10
    try
        sim('mymodel')
    catch
        end
end
```
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**
```matlab
ttAttachDLHandler(taskname, handlername)
```

**C++ syntax**
```cpp
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:

```c
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);
}
```
**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.

Note: Avoid using the “Discrete-Time Integrator” and “Discrete Derivative” Simulink blocks, which seem to implement something else than the transfer functions that are displayed on the blocks. For the “Discrete Derivative” block it is also not possible to specify the sample rate, which makes in incompatible with TRUE_TIME.

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

```matlab
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
```

54
**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 (so called condition variables).

**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)

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

\[
\text{ttCreateLog(taskname, logtype, variable, size)}
\]

C++ syntax

\[
\text{void ttCreateLog(char* taskname, int logtype, char* variable, int size)}
\]

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 \texttt{ttLogStart}, \texttt{ttLogStop}, and \texttt{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

\[
% \text{In initialization script}
\]

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

% User log #1 (type 6) for logging of I/O latency  
\texttt{ttCreateLog('ctrl_task', 6, 'IOLatency', 100)};

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

\[
\text{switch seg, case 1,} \text{ ttLogStart(1); % start I/O logging in user log #1} \text{ y = ttAnalogIn(1); % Input} \text{ data.u = calculateOutput(y);} \text{ exectime = 0.003;}
\]
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)`
- `ttCreateMailbox(mailboxname, maxsize)`

**C++ syntax**
- `void ttCreateMailbox(char *mailboxname)`
- `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 (default is `INT_MAX`)

**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 maximum size of this buffer may be specified by `maxsize`.

**See Also**
- `ttTryPost`, `ttTryFetch`, `ttPost`, `ttFetch`, `ttRetrieve`
**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`
- `ttWait`
- `ttNotify`
- `ttNotifyAll`
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`
**Purpose**

Create a simple, counting TrueTime semaphore.

**Matlab syntax**

```
 ttCreateSemaphore(name, initval)
 ttCreateSemaphore(name, initval, maxval)
```

**C++ syntax**

```
 void ttCreateSemaphore(char *name, int initval)
 void ttCreateSemaphore(char *name, int initval, int maxval)
```

**Arguments**

- `name`: Name of the semaphore. Must be a unique, non-empty string.
- `initval`: The initial value of the semaphore. Must be an integer.
- `maxval`: The maximum value of the semaphore. Must be an integer. The default maximum value is INT_MAX.

**Description**

This function is used to create a simple, counting semaphore. Semaphores can be used to synchronize tasks, for instance, to synchronize a producer and a consumer task. Note that a semaphore does not have the concept of a current owner task and hence should not be used to achieve mutual exclusion.

The `ttGive` command has no effect if the current value of the semaphore is equal to `maxval`. To create a binary semaphore, set `maxval` to 1.

Compared to TrueTime monitors, TrueTime semaphores do not produce any output graphs. Also, they do not support any priority inheritance protocol, but rather implement a simple FIFO waiting queue.

**DO NOT** use semaphores to protect critical regions! For that purpose you should use monitors!

**See Also**

`ttTake, ttGive`
ttCreateTask (I)

Purpose
Create a TrueTime task.

Matlab syntax

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

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

C++ syntax

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

```cpp
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(timernamex, time, handlername)
```

**C++ syntax**
```
void ttCreateTimer(char *timernamex, double time, char *handlername)
```

**Arguments**
- **timernamex**: 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.
ttDiscardUnsentMessages (IT)

Purpose
Delete unsent messages in the network queue.

Matlab syntax
nbr = ttDiscardUnsentMessages
nbr = ttDiscardUnsentMessages(network)

C++ syntax
int ttDiscardUnsentMessages()
int ttDiscardUnsentMessages(int network)

Arguments
network The network interface from which messages should be deleted
(default is 1).

Description
This function can be used to delete any unsent messages waiting in the network
interface output queue. A message that has already started to be sent is not
affected. The function returns the total number of deleted messages.
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,
        if some_condition_not_fulfilled
            ttWait('condvar');
            ttSetNextSegment(2);
            exectime = 0;
        else
            criticalOperation;
            exectime = 0.005;
        end
    case 3,
        ttExitMonitor('mutex');
        exectime = -1;
end
See Also

ttCreateMonitor, ttExitMonitor, ttCreateEvent, ttWait, ttNotify, ttNotifyAll
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
**ttFetch (T)**

**Purpose**
Fetch a message from a mailbox, blocking if the mailbox is empty.

**Matlab syntax**

ttFetch(mailboxname)

**C++ syntax**

```cpp
void ttFetch(char *mailboxname)
```

**Arguments**

- `mailboxname` Name of a mailbox.

**Description**
This function is used to wait for a message at a mailbox. If the mailbox is empty, the task will be blocked until a message has arrived. Tasks waiting to fetch from a mailbox are sorted in FIFO order.

Note that this function does not read the actual message – that is done using `ttRetrieve`. (This somewhat awkward solution is due to the segment structure of `TRUETime`.)

**Example**

```matlab
function [exec_time, data] = ctrl(seg, data)
switch seg,
    case 1,
        ttFetch('mailbox'); % wait for a message
        exec_time = 0;
    case 2,
        msg = ttRetrieve('mailbox'); % read the actual message
        doStuff;
        exec_time = 0.005;
    case 3,
        exec_time = -1;
end
```

**See Also**

- `ttCreateMailbox`, `ttTryPost`, `ttTryFetch`, `ttPost`, `ttRetrieve`
ttGetData (TH)

Purpose
Retrieve the local memory data structure associated with a specific task.

Matlab syntax
\[ \text{data} = \text{ttGetData}(	ext{taskname}) \]

C++ syntax
\[ \text{void *ttGetData(char *taskname)} \]

Arguments
\[ \text{taskname} \quad \text{Name of a task.} \]

Description
This function is used to retrieve the data structure of a specific task. The data structure represents local memory of the task and is normally updated in the code function of the task. Using this function in combination with ttSetData it is possible to modify the data also outside the code function.

See Also
ttSetData, ttCreateTask, ttCreatePeriodicTask
**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**

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

**C++ syntax**

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

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>network</td>
<td>The network interface from which the message should be received. The default network number is 1.</td>
</tr>
<tr>
<td>signalPower</td>
<td>The value of the received signal power corresponding to this message. Only used in the wireless network.</td>
</tr>
</tbody>
</table>

**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**

```matlab
% 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`
ttGetInitArg (C++ only) (I)

Purpose
Retrieve a pointer to the init argument of the TrueTime block.

C++ syntax
mxArray *ttGetInitArg()

Description
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
**ttGive (TH)**

**Purpose**

Give (signal) a semaphore.

**Matlab syntax**

```matlab
ttGive(semname)
```

**C++ syntax**

```c++
void ttGive(char *semname)
```

**Arguments**

- `semname` The name of the semaphore.

**Description**

This function is used to signal a semaphore. Its semantics can be described by the following pseudo-code:

```plaintext
if (value < maxval) {
    value++;
    if (value <= 0) {
        release the first task from the semaphore queue;
    }
}
```

**See Also**

`ttCreateSemaphore`, `ttTake`
ttInitKernel (I)

**Purpose**
Initialize the TRUETIME kernel.

**Matlab syntax**

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

**C++ syntax**

```
void ttInitKernel(int nbrInp, int nbrOutp, double (*prioFcn)(UserTask*))
void ttInitKernel(int nbrInp, int nbrOutp, double (*prioFcn)(UserTask*),
                  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'. In the C++ case, an arbitrary priority function can be supplied. There are predefined priority functions called prioFP, prioRM, prioDM, and prioEDF.

**Example**
As two examples, the priority functions implementing fixed-priority scheduling and earliest-deadline-first scheduling are given below. Note that a small return value corresponds to a high priority.

```
double prioFP(UserTask* task) {
    return task->priority;
}

double prioEDF(UserTask* task) {
    return task->absDeadline;
}
```
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
\[
\text{ttInitNetwork(nodenum, handlername)} \\
\text{ttInitNetwork(network, nodenum, handlername)}
\]

C++ syntax
\[
\text{void ttInitNetwork(int nodenum, char *handlername)} \\
\text{void ttInitNetwork(int network, int nodenum, char *handlername)}
\]

Arguments
- \text{network} The number of the TrueTime network block. The default network number is 1.
- \text{nodenum} 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.
- \text{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
- \text{ttSendMsg, ttGetMsg}
**ttInvokingTask (H)**

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

**Matlab syntax**

```matlab
 task = ttInvokingTask
```

**C++ syntax**

```cpp
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**

```cpp
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 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**
```matlab
ttLogStart(logID)
```

**C++ syntax**
```cpp
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**

```matlab
ttLogStop(logID)
```

**C++ syntax**

```cpp
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`
ttPost (T)

Purpose
Post a message to a mailbox, blocking if the mailbox is full.

Matlab syntax
ttPost(mailboxname, msg)

C++ syntax
void ttPost(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 the mailbox is full, the task will be blocked. Tasks waiting to post to a mailbox are sorted in FIFO order.

See Also
ttCreateMailbox, ttTryPost, ttTryFetch, ttFetch, ttRetrieve
**ttRemoveTimer (TH)**

**Purpose**
Remove a specific timer.

**Matlab syntax**
```matlab
ttRemoveTimer(timername)
```

**C++ syntax**
```cpp
void ttRemoveTimer(char *timename)
```

**Arguments**
- `timename` 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`
**ttRetrieve (T)**

**Purpose**
Read a message that was fetched using `ttFetch`.

**Matlab syntax**

```matlab
msg = ttRetrieve(mailboxname)
```

**C++ syntax**

```c
void* ttRetrieve(char *mailboxname)
```

**Arguments**

- `mailboxname` Name of a mailbox.

**Description**
This function may only be called if `ttFetch` was called in the previous segment. It is used to read the fetched message. (This somewhat awkward solution is due to the fact that blocking primitives in TrueTime cannot return any data.)

**Example**

```matlab
function [exectime, data] = ctrl(seg, data)
switch seg,
    case 1,
        ttFetch('mailbox');    % wait for a message
        exectime = 0;
    case 2,
        msg = ttRetrieve('mailbox');  % read the actual message
        doStuff;
        exectime = 0.005;
    case 3,
        exectime = -1;
end
```

**See Also**

- `ttTryPost`, `ttTryFetch`, `ttPost`, `ttFetch`
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 bits. Determines the time it will take to transmit the message. **NOTE:** Changed from bytes to bits in TrueTime 1.4
- **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`
ttSetData (TH)

Purpose
Update the local memory data structure associated with a specific task.

Matlab syntax
ttSetData(taskname, data)

Arguments
- taskname: Name of a task.
- data: Updated data structure.

Description
This function is used to update the data structure of a specific task. The data structure represents local memory of the task and is normally updated in the code function of the task. Using this function in combination with ttGetData it is possible to modify the data also outside the code function.

The ttSetData function only exists in Matlab syntax. No C++ version is needed, since ttGetData gives a pointer to the task data structure.

See Also
ttGetData, ttCreateTask, ttCreatePeriodicTask
**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'**: The transmitpower parameter is only valid when using the wireless network and defaults to whatever is set in the block mask.
- **'predelay'**: 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. The default value is zero.
- **'postdelay'**: The time a message is delayed by the network interface on the receiving end. The default value is zero.
**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 (T)

**Purpose**
Put the running task to sleep for a certain time.

**Matlab syntax**
ttSleep(duration)

**C++ syntax**
void ttSleep(double duration)

**Arguments**
duration The time that the task should sleep.

**Description**
This function is used to make the running task sleep for a specified amount of time. 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 (T)

Purpose
Put the running task to sleep until a certain point in time.

Matlab syntax
    ttSleepUntil(time)

C++ syntax
    void ttSleepUntil(double time)

Arguments
    time  The time when the task should wake up.

Description
This function is used to make the running task sleep until a specified point in
time. A call to this function will trigger execution of the suspend-hook of the
task.

See Also
    ttSleep
ttTake (T)

**Purpose**
Take (wait for) a semaphore.

**Matlab syntax**
```matlab
ttTake(semname)
```

**C++ syntax**
```cpp
void ttTake(char *semname)
```

**Arguments**
- `semname` The name of the semaphore.

**Description**
This function is used to take a semaphore. Its semantics can be described by the following pseudo-code:

```plaintext
value--;  
if (value < 0) {
    suspend the task and put it in the semaphore queue;
}
```

**Example**
```matlab
function [exectime, data] = producer_code(seg, data)
    switch seg,
    case 1,
        produce_data;
        exectime = 0.020;
    case 2,
        ttTake('sem'); % wait until the consumer task is ready
        exectime = 0;
    case 3,
        send_data_to_consumer;
        exectime = 0.005;
    case 4,
        exectime = -1;
    end
```

**See Also**
ttCreateSemaphore, ttGive
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, ttPost, ttFetch, ttRetrieve
**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, ttPost, ttFetch, ttRetrieve
**ttWait (T)**

**Purpose**
Wait for an event.

**Matlab syntax**
```matlab
ttWait(eventname)
```

**C++ syntax**
```c
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:

```matlab
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**
- `ttEnterMonitor`
- `ttCreateEvent`
- `ttNotify`
- `ttNotifyAll`
15. References


