**C:\ProgramData\DLR\CD-Vorlagen\Stammdaten\dlrlogo\tif\DLR-Logo_Grau_US.tif**

**Tasking Framework – Documentation**

SC-SRV-OSS

Date: 27.08.2019

Author: Zain Alabedin Haj Hammadeh

Contents

[Overview 3](#_Toc17815790)

[Solution Strategy 3](#_Toc17815791)

[Application Programming Interface (API) 5](#_Toc17815792)

[Activation Model 6](#_Toc17815793)

[Time-triggered 7](#_Toc17815794)

[Periodic 7](#_Toc17815795)

[Relative-time 7](#_Toc17815796)

[Call Semantics 8](#_Toc17815797)

[Asynchronous 8](#_Toc17815798)

[Synchronous 8](#_Toc17815799)

[Execution Model 9](#_Toc17815800)

[Scheduling and priority handling 11](#_Toc17815801)

[Thread lifecycle 12](#_Toc17815802)

[Unit Test 12](#_Toc17815803)

[Copyrights 12](#_Toc17815804)

[Illustrative Example 12](#_Toc17815805)

[Final Remark 14](#_Toc17815806)

# Overview

Tasking Framework is an event-driven multithreading execution platform and software development library. It is dedicated to develop space applications, which perform on-board data processing and sophisticated control algorithms.

* Lightweight C++ API
* Implemented in C++
* Event-driven programming paradigm
* Inverse of control programming principle (OOP)
* Multithreading execution with thread polls
* Compatible with POSIX-like operating systems, e.g., RTEMS
* Tasking Framework is neither a testing platform nor an operation system
* Apache License 2.0
* Developed at German Aerospace Center (DLR)
* Main contact: Olaf Maibaum (olaf.maibaum@dlr.de)

# Solution Strategy

In embedded control systems, sensors data are collected and processed to update the system status, and to send accordingly control commands to actuators. That keeps the state trajectory of the system. Traditionally, computation tasks are activated periodically based on sensors delay, which is conservatively bounded. Therefore, computation tasks get a tight time budget to satisfy the real-time requirements.

Tasking Framework provides means to activate the computation tasks on a reactive way instead on a fixed timing. Reactive means, that a computation task is activated at once all input data are available. As soon as possible scheduling of ready tasks is therefore possible.

Real-time capabilities are necessary to analyze optical sensor data and to react on the system’s estimated position.

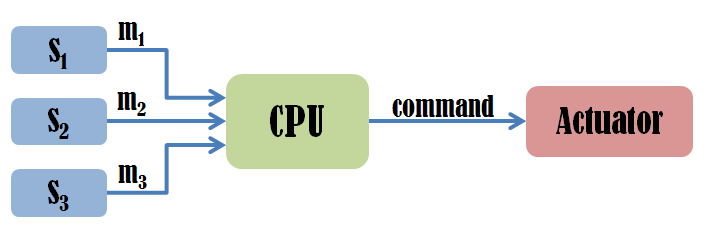


Figure 1: Typical embedded control system

Tasking Framework is implemented as a *namespace* Tasking, which comprises abstract classes with few virtual methods. It consists of the *execution platform* and the *application programming interface (API)*. Using the Tasking Framework, applications are implemented as a graph of *tasks* that are connected via *channels*, and each task has one or more *inputs*. Periodic tasks are connected to a source of *events* to trigger the task periodically. In practice:

* Each computation block of a software component is realized by the class Tasking::Task. The virtual method *Task::execute()* will be overridden by the code of the software component;
* Each input of a task is realized by the class *Tasking::Input*;
* Each input object is associated with an object of the class *Tasking::Channel*;
* Each task may have multiple inputs and multiple outputs;
* A set of tasks, inputs and channels are framed in a scheduler entity, which is realized by the class *Tasking::Scheduler*;
* Each scheduler entity is provided with a scheduling policy;
* Each scheduler entity has threads to execute the assigned tasks according to the specified scheduling policy. The number of threads is specified by the software developer.
* Tasks can be activated also periodically by the means of the class *Tasking::Event*

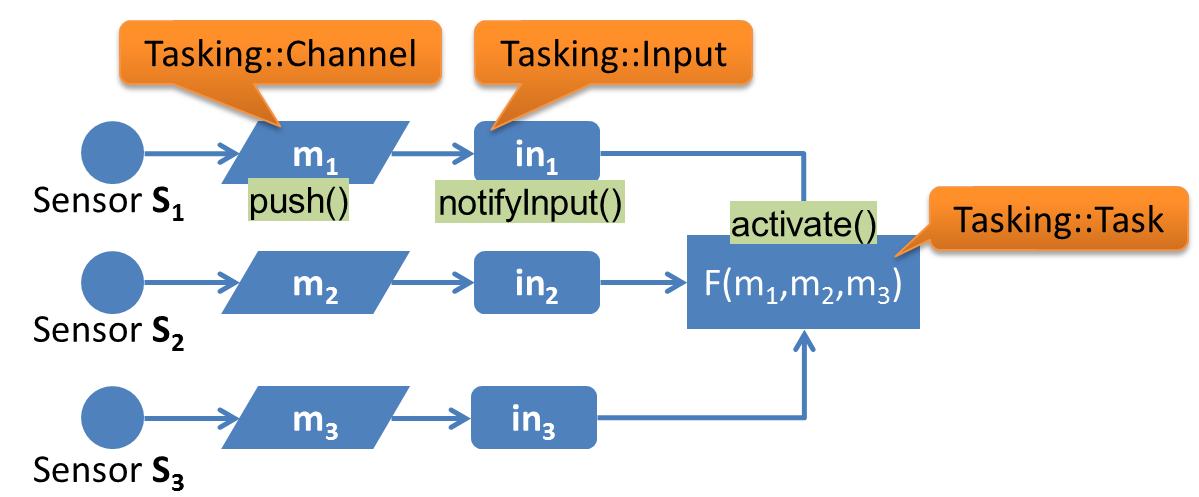


Figure 2: How Tasking Framework realizes the example in Figure 1

Although space software standards discourages virtual methods, the *execute()* method of tasks should be virtual to let the developer implement different tasks. A few other methods are intentionally virtual to add application code, e.g. synchronization of channel data. To simplify setting up an object w.r.t. static memory management, we designed templates for the main classes.

The implementation follows the way of the operating systems used in the BIRD and TET-1, which is also the concept in the Core Avionics activities in DLR. In these systems the run time system functionality is provided to the application by inheritance from run time system classes. These classes provide virtual methods as entry point which shall overloaded by the application to add the computations to the software.

To achieve the portability of an application the implementation is split in the API part of the implementation and an execution platform part with the platform specific code. The application programmer has no access to the execution platform part. So an application can cross compiled for each platform by selecting the corresponding platform. The important logic for the implemented functionality shall be part of the API.

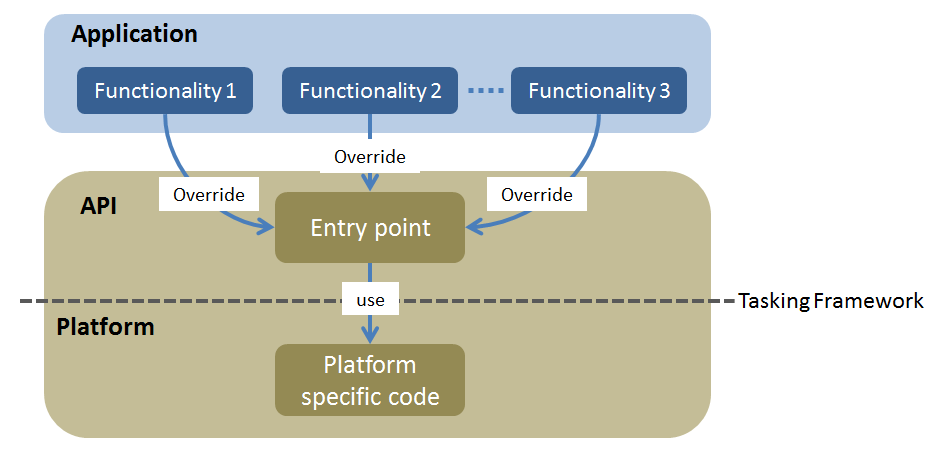


Figure 3: Solution strategy

# Application Programming Interface (API)

A lightweight C++ API was developed representing the main elements of Tasking Framework, namely, *Tasking::Scheduler*, *Tasking::Channel*, *Tasking::Input*, *Tasking::Task*.

To simplify setting up an object w.r.t. static memory management, we designed templates for the main classes.

In addition, Tasking Framework has two more classes in its API:

*Tasking::Group*: The default call semantics among tasks that is supported in Tasking Framework is asynchronous, in which a task activates the successor tasks, and then it can be executed again regardless of the status of the successor tasks, see ***Call Semantics***. However, in some applications, the graph of tasks or a subset of it has synchronous call semantics such that activates the successor tasks and it will not be executed again till all tasks in the synchronous subset finish their execution. To support the synchronous call semantics, the class *Tasking::Group* is provided.

*Tasking::Barrier*: The number of activations at an input is declared at compile time. In situations, where the number of data elements is only known at run time, the activation cannot be adapted. This can be the case when, for example, a data source have states where no data is sent. The class *Tasking::Barrier* is a mean to control the activation of tasks with an unknown number of data packets. By default the barrier can be instantiated with a minimum number of expected push operations on the barrier. After the minimum number of pushes happens, the barrier will activate all associated inputs, as long as data sources did not increase the number of expected push operations on the channel. If it is increased, more push operations are expected.

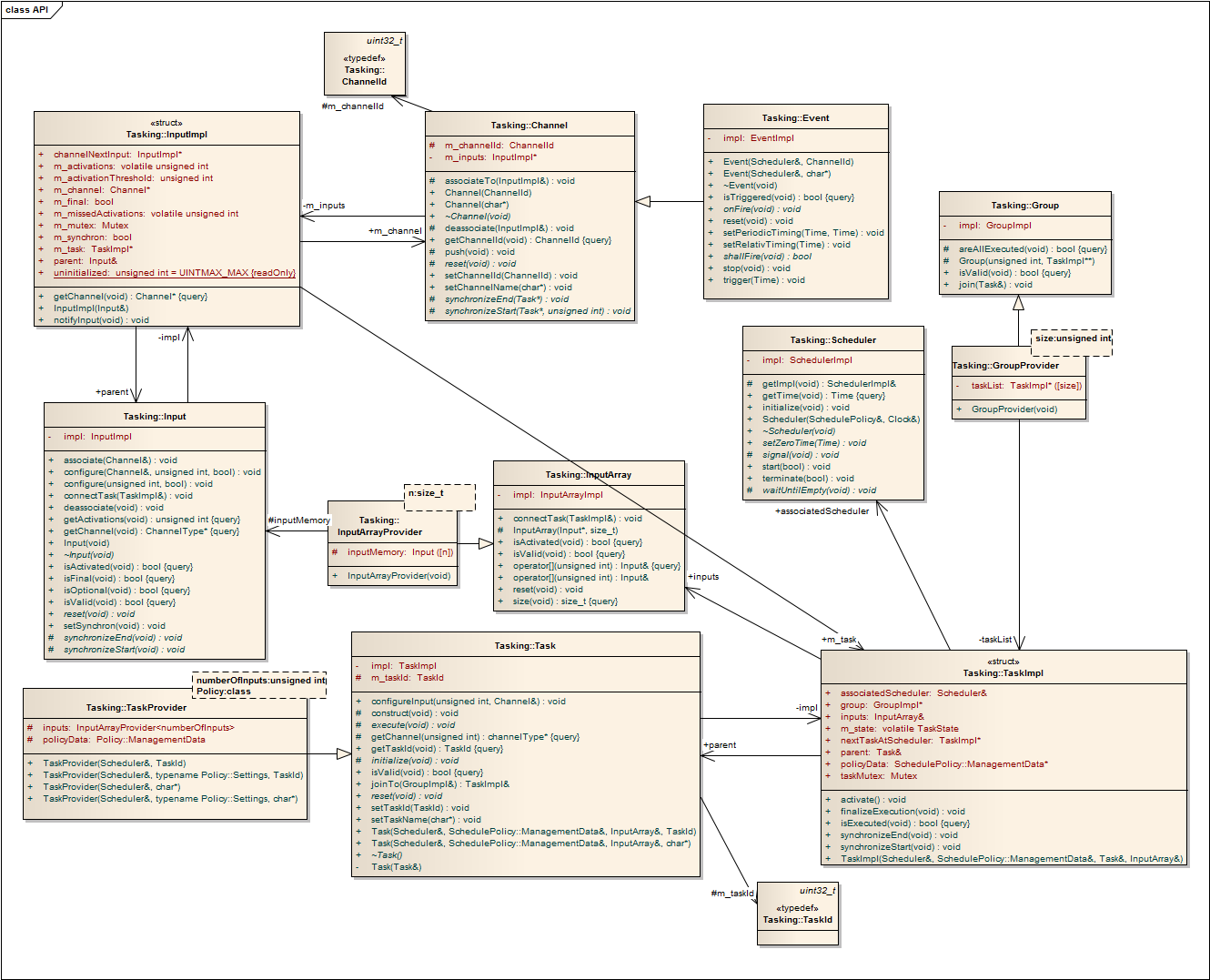


Figure 4: Class diagram

# **Activation Model**

A task is activated and an instance of it will be queued when all inputs are activated (*and* semantic). *Or* semantic is also supported by providing the final flag. When the *final* flag is set for an input, the task will be activated regardless of other inputs.

The j-th input of task is activated when a predecessor task or other sources, e.g. the main thread, calls *Channel::push()* on the associated channel with . In the context of *Channel::push()*, the input will be set to active and if the final flag is set then will be activated, otherwise, the other inputs will be checked and will be activated only when all inputs associated to it are set  
to active.

Note that, it is possible to set the final flag for one or more of inputs as a custom activation.

Figure 5, Figure 6, Figure 7 show and, or custom semantics respectively.

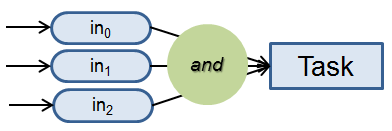


Figure 5: And semantic

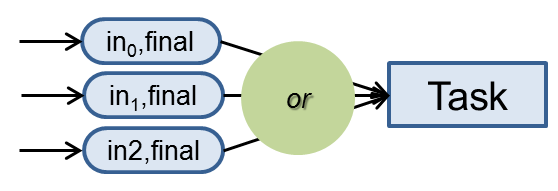


Figure 6: Or semantic; the final flag is set for all inputs

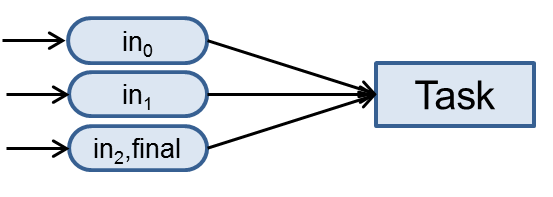


Figure 7: Custom activation; the final flag is set only for one input

## Time-triggered

Although we design our platform to be event-driven, time-triggered activation is supported by presenting the class *Tasking::Event*. Two time-triggered activation methods are supported: periodic and relative time.

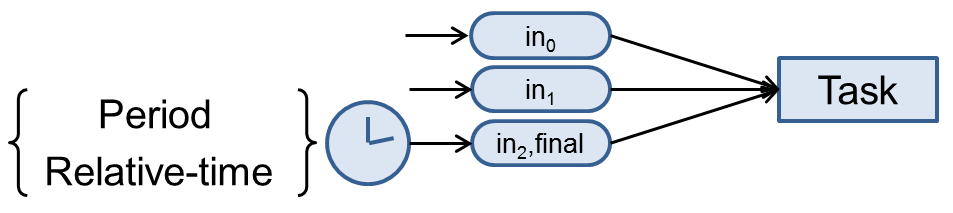


Figure 8

### Periodic

In the periodic method, the given time duration represents the distance between two successive events.

### Relative-time

Relative time method is used, for instance, when sensor data is needed. A task sends a request command to the sensor then it sets the timer to predefined time duration (relative time) and terminates. After the timeout occurrence, the following task reads the response sent by the sensor. Note that, this solution is similar to using self-suspending tasks. Using relative time (in general using self-suspending tasks) requires to tightly bound the timeout. However, using channels connected to Interrupt Service Routines (ISR) of IO drivers (event-driven programming), in which is activated only when the sensor data is available, can improve the utilization.

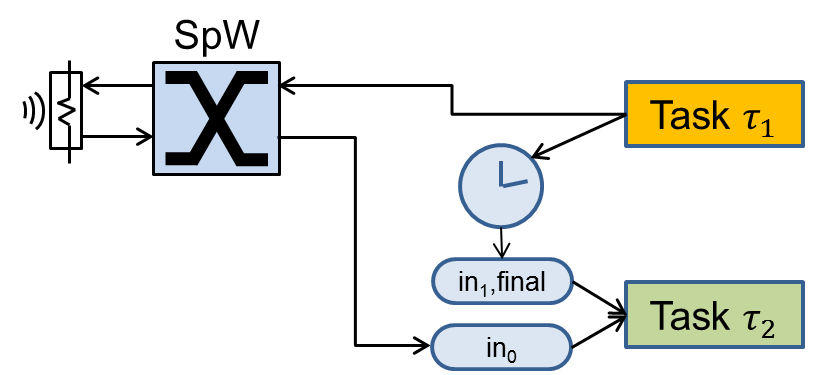
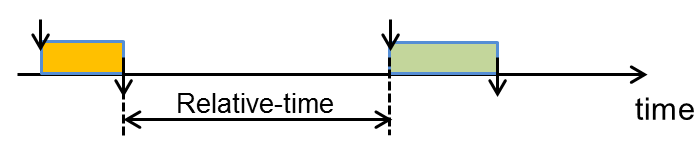


Figure 9: An example of using the relative-time

# Call Semantics

This section describes how to execute a task chain, i.e., asynchronous or synchronous.

## Asynchronous

The default call semantics among tasks that is supported in Tasking Framework is asynchronous, in which a task activates the successor tasks, and then it can be executed again regardless of the status of the successor tasks.

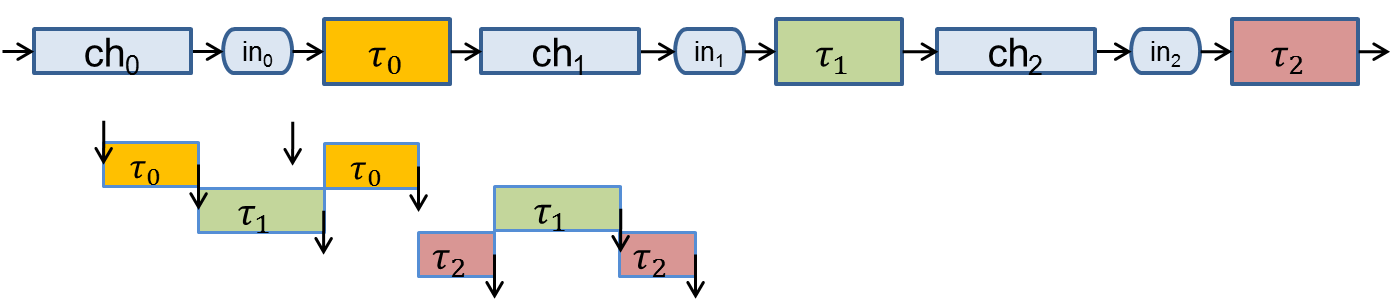


Figure 10: Asynchronous execution of the task chain ()

## Synchronous

In some applications, the graph of tasks or a subset of it has synchronous call semantics such that activates the successor tasks and it will not be executed again till all tasks in the synchronous subset finish their execution. To support the synchronous call semantics, the class *Tasking::Group* is provided.

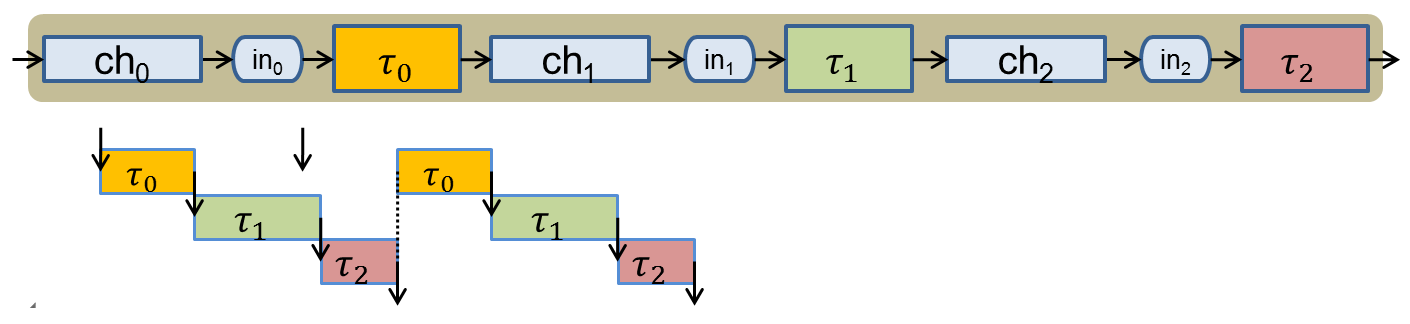


Figure 11: Synchronous execution of the task chain () using Tasking::Group

# Execution Model

Tasking Framework is a multithreading execution platform. The software developer should specify the number of threads, called executors. Therefore, there will be n + 1 threads: the main thread plus n executors. The implementation of the execution model is platform specific. We have three implementations of the execution model: the POSIX threading model (targeting Linux), C++11 threading and OUTPOST-core (targeting RTEMS and FreeRTOS).  
The execution model is represented by 4 classes:

* *Tasking::SchedulerExecutionModel*: Creating, scheduling, managing the executor threads and interfacing to the API.
* *Tasking::ClockExecutionModel*: Managing the time for time events. In embedded Linux, the clock is represented by a thread.
* *Tasking::Mutex*: An encapsulation of the mutex.
* *Tasking::Signaler*: An encapsulation of the conditional variables.

In the implementation for Linux, the clock is implemented as a thread with the real-time clock provided by POSIX. The clock thread goes to sleep for a timeout equal to, e.g., the period of a periodic task. Then it signals a free executor if there is any, and it computes the next timeout.

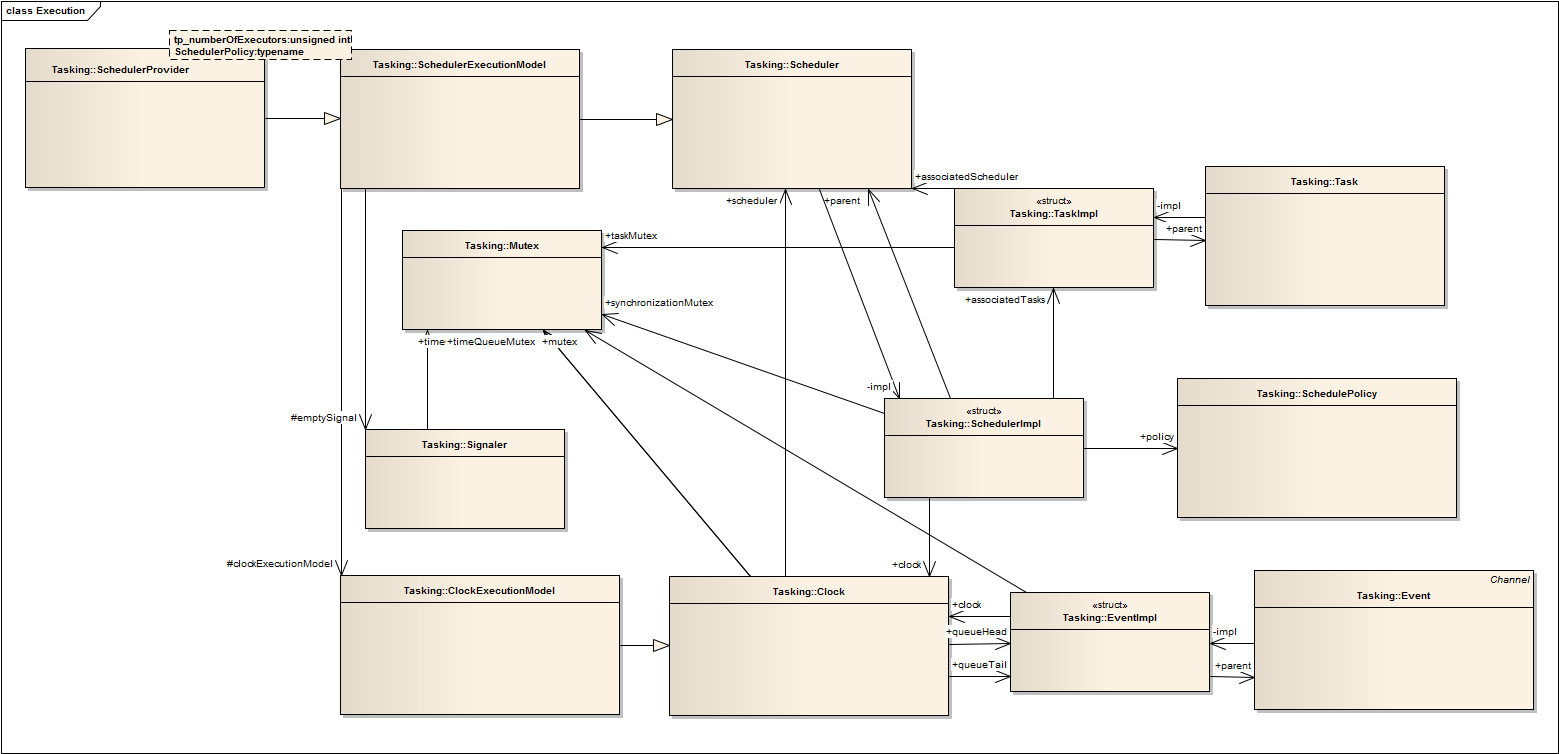


Figure 12: Class diagram illustrating the execution platform

Tasking Framework schedules the ready task instances to the available executors according to the following scheduling policies: First-In-First-Out (FIFO), Last-In-First-Out (LIFO), and Static Priority Non-Preemptive (SPNP). The software developer can assign a priority to each task to be used by the SPNP queue.

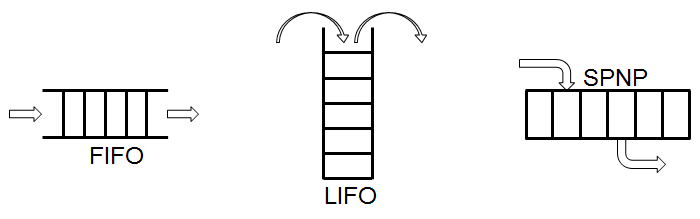


Figure 13: Scheduling policies in Tasking Framework

The executor that executes the task activates the successor tasks and queue them in the ready queue, and it will signal a free executor, which is in *WAITING* state, if there is any. That is to say, Tasking Framework balances the load on the available executors. Even in case of one executor, the executor returns first from the execute() method of before checking the ready queue and executing the successor tasks of τi. The sequence of method calls that are performed by Tasking Framework to execute a task by an executor thread is shown in Figure 14. Because we have multiple threads that may try to access the data stored in the channel, a protection mechanism is implemented to synchronize the access to this shared data by different threads. The protection mechanism is implemented by the means of two virtual methods: *Channel::synchronizeStart* and *Channel::synchronizeEnd*.

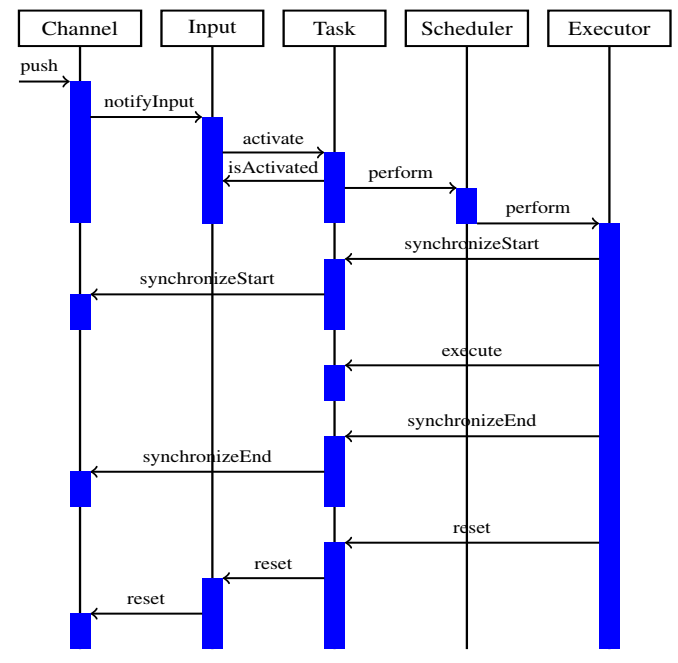
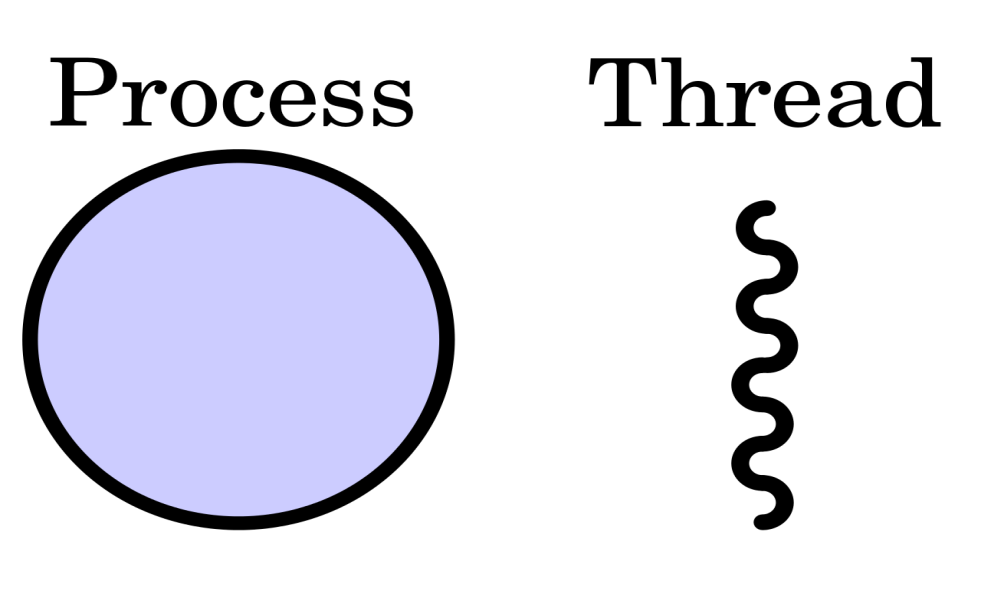
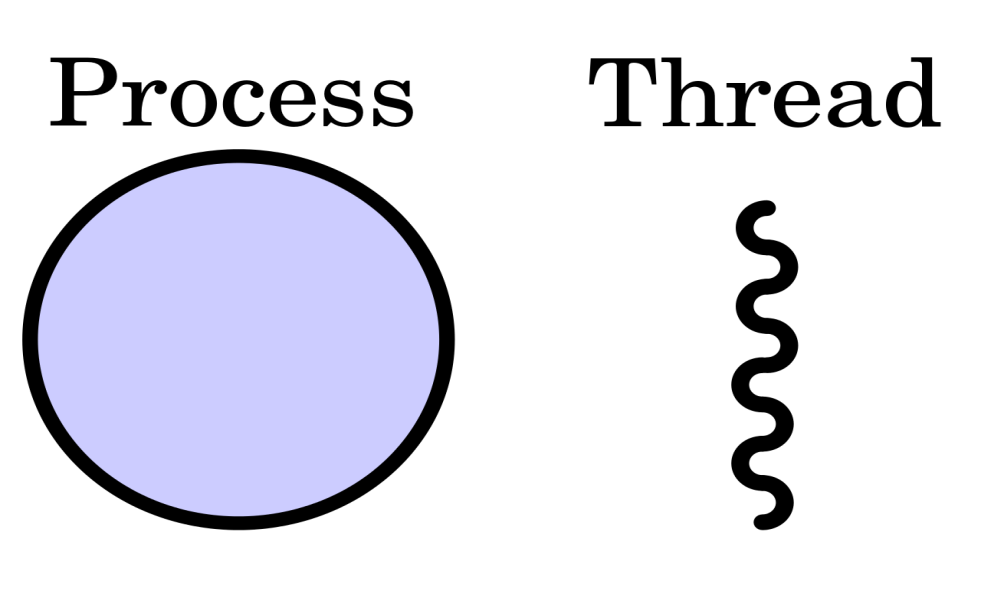


Figure 14: Sequence Diagram

## Scheduling and priority handling

Each Scheduler entity is assigned a number of tasks, scheduling policy and a pool of threads. In one scheduler entity, tasks are scheduled to the threads according to the global scheduling with work-conserving scheduling. Threads belong to the same scheduler entity (threads pool) have the same priority.



Scheduler

Task0

Task1

Taskn1

Scheduling policy

ex1

exm1

Queue

Figure 15: Tasking Framework realizes an application by a scheduler entity, number of tasks, scheduling policy, and a pool of threads

## Thread lifecycle

An executor thread goes to sleep, i.e. waits on a conditional variable, after being created till it gets a signal from the clock thread (or a timer) in case of time-triggered tasks, or from other sources, e.g. the main thread. The thread suspends when there is no load to execute or on a conditional variable to safely access a shared region, e.g., the channel’s memory element.

Figure 16: Thread lifecycle

**create()**

**wait(&cond\_var)**

**signal()**

**join()**

# Unit Test

We provide a special scheduler SchedulerUnitTest with step operation to support unit testing. Using Googletest (gtest), we provide twelve classes to test the API.

Note that, the execution model has to be tested separately by the developer using other means, e.g. stress test.

# Copyrights

Copyright 2012-2019 German Aerospace Center (DLR) SC

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

# Illustrative Example

We show here the *ioChannelExample.cpp* example in *example* folder. In this example input from the keyboard and the print out to the screen is done by special channels connected to standard IO streams. By default the new line is appended to the existing line, but after a time the first word would disappear. Figure 17 shows the example.

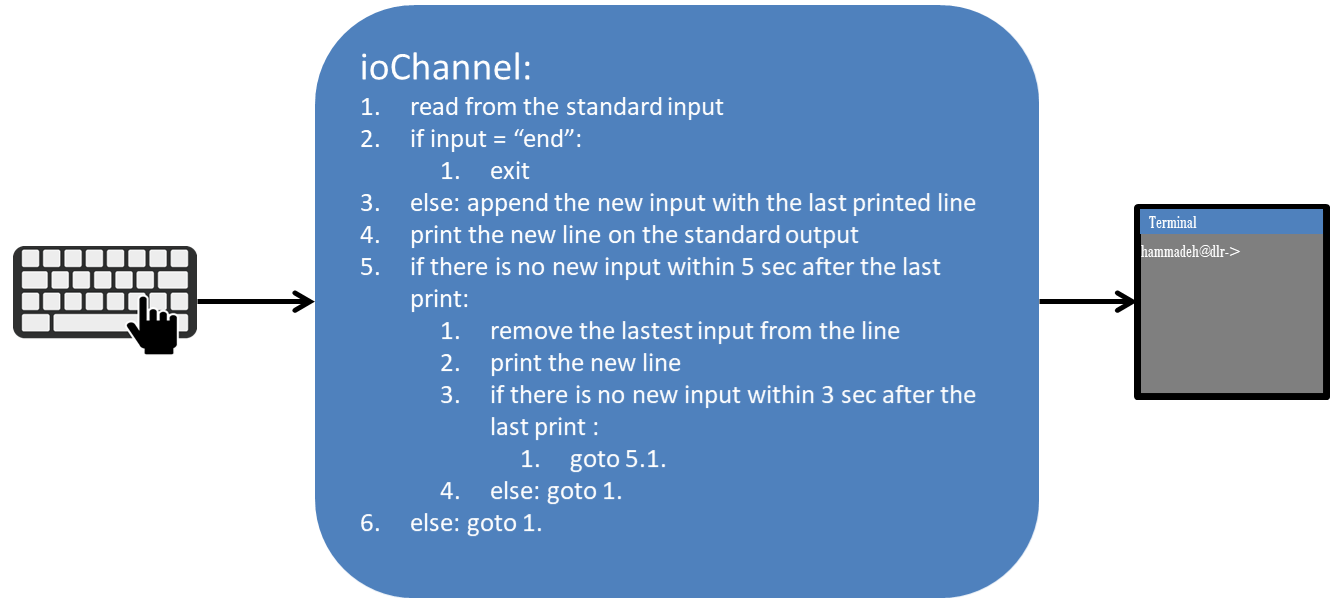


Figure 17: ioChannel example

The example was realized by 1 scheduler entity, 2 channels, 2 inputs, 2 tasks, 1 timer (trigger), FIFO scheduling policy, and 1 executor.

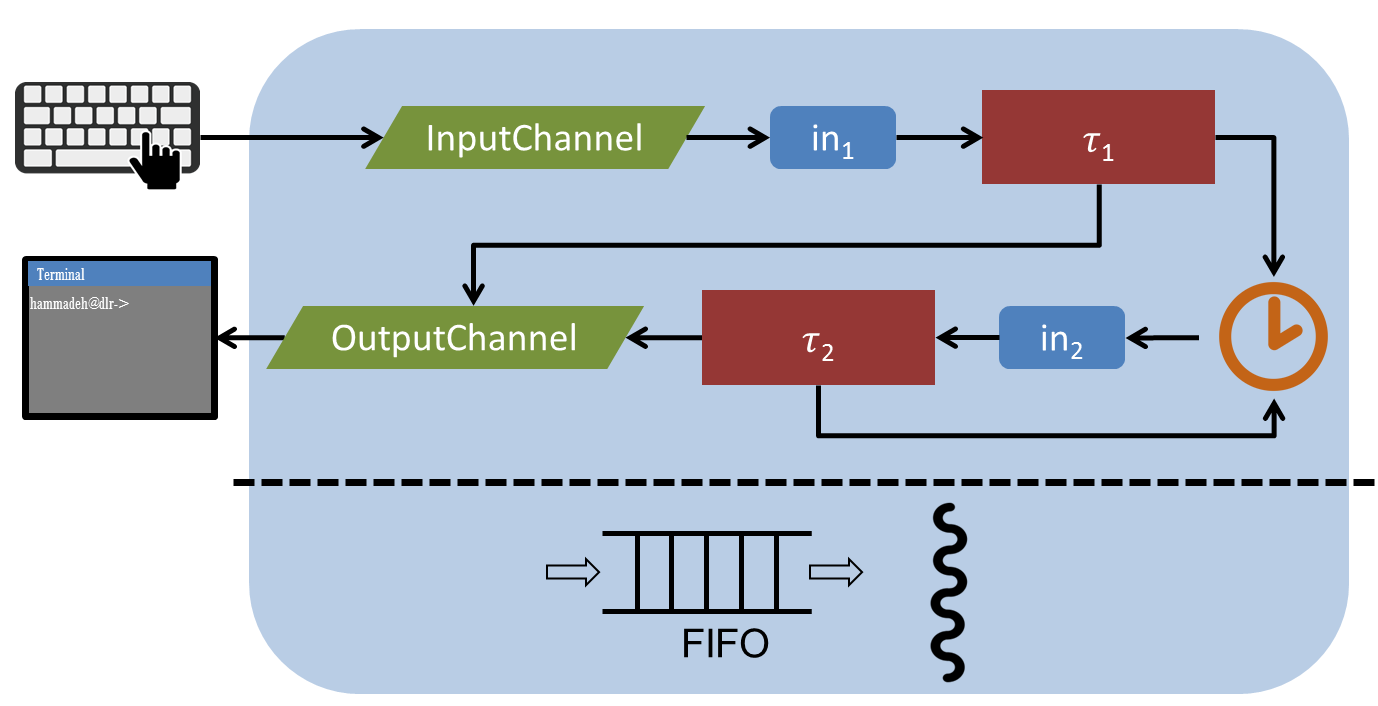


Figure 18:Tasking Framework implementation of ioChannel example

We define the channels as classes:

class KeyboardInputChannel : public Tasking::Channel

class OutputChannel : public Tasking::Channel

appends the last printed line with the new input, pushes to the output channel, and sets the timer (trigger) for 5 sec. Using the template *TaskProvider<numberOfInputs, policy>*, we define a new class:

class HandleKeyboardInput : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo>

Similarly, we define a new class for :

class ModifiyLastWrittenLine : public Tasking::TaskProvider<1u, Tasking::SchedulePolicyFifo>

is responsible for removing the latest input from the line, printing the new line, and set the timer for 3 sec.

After defining the classes, we create an object of the scheduler entity using the template *SchedulerProvider<tp\_numberOfExecutors, SchedulerPolicy>*. Also, we create an object of the class *Tasking::Event* to be the timer (trigger), and we assign it to the corresponding scheduler entity. We define objects of the new defined classes as well. Task objects are assigned to the scheduler entity and connected to the corresponding outputs.

Tasking::SchedulerProvider<1u, Tasking::SchedulePolicyFifo> scheduler;

KeyboardInputChannel inChannel;

OutputChannel outChannel;

Tasking::Event modifyTrigger(scheduler);

HandleKeyboardInput handleInput(scheduler, outChannel, modifyTrigger);

ModifiyLastWrittenLine modifyTask(scheduler, outChannel);

We need to configure the inputs of tasks. is connected to the input channel inChannel, while has the trigger as input:

handleInput.configureInput(0u, inChannel);

modifyTask.configureInput(0u, modifyTrigger);

The entry point is main(), in which we start the scheduler entity. That will create the executor and start the execution of the designed software. Note that, we have for each application at least 2 threads: the main and one executor. In this example, we use the main thread to read from the input (keyboard).

// Start Tasking scheduler

scheduler.start();

To safely finish the execution of Tasking Framework, the method *terminate()* joins the executor thread.

// Stop Tasking scheduler

scheduler.terminate(true);

# Final Remark

For further and more detailed comments, please check the comments on the source code. Note that, the comments follow the Doxygen style, therefore, you can generate an HTML or a PDF format of the comments.