Tasking Framework

- Open-source ✔
- Event-driven ✔
- Multithreading ✔
- C++ ✔
- Development library ✔
- Execution platform ✔

- Embedded systems with static memory requirements ✔
- Non-embedded software ✔
- Compatible with POSIX-like OS: Linux, RTEMS ✔

- Operating system ✗
- Testing platform ✗
Open-source

https://github.com/DLR-SC/tasking-framework

Copyright 2012 **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.
Solution strategy

The solution strategy involves using a Tasking Framework that integrates application functionalities with a platform-specific code. The diagram shows:

- **Application** with functionalities: Functionality 1, Functionality 2, ..., Functionality n.
- **API** with override options for classes and methods.
- **Entry Point** that uses platform-specific code:
  - Classes, Abstract classes, few virtual methods.
  - Linux OUTPOST (RTEMS).
- **Platform** and **Tasking Framework**

The diagram illustrates how the application functionalities interact with the platform and the tasking framework, emphasizing the use of overrides for customization and integration.
Application Programming Interface (API)
Concept

Separation between data and functionality

Channel₁ → Task₁ → Channel₂ → Task₂

Data container

Interface

Reusability

Data handling

Distributed systems
Sequence

Channel

Sensor A

Msg. A

push()

1

Input

notifyInput()

1

Sensor B

Msg. B

1

Sensor C

Msg. C

E = F(A, B, C)

0, final

1

Task

activate()

perform()
Sequence cont’d

Channel: push
Input: notifyInput, synchronizeStart, synchronizeEnd, reset
Task: activate, isActivated*, perform, synchronizeStart, synchronizeEnd, reset
Scheduler: signal, synchronizeStart, execute, synchronizeEnd, reset
Executor
Example
ioChannel:
1. read from the standard input
2. If input = “end”:
   1. Exit
3. else: append the new input with the last printed line
4. print the new line on the standard output
5. if there is no new input within 5 sec after the last print:
   1. remove the latest input from the line
   2. print the new line
   3. if there is no new input within 3 sec after the last print:
      1. goto 5.1.
   4. else: goto 1.
6. else: goto 1.
Example

InputChannel → \( \text{in}_1 \) → \( \tau_1 \) → OutputChannel

\( \tau_2 \) → \( \text{in}_2 \) → Terminal

\( \text{FIFO} \)
Activation model
Activation model

1. \( \text{in}_0 \) \( \rightarrow \) Task
   \( \text{in}_1 \) \( \rightarrow \) Task
   \( \text{in}_2 \) \( \rightarrow \) Task
   \( \text{and} \)

2. \( \text{in}_0, \text{final} \) \( \rightarrow \) Task
   \( \text{in}_1, \text{final} \) \( \rightarrow \) Task
   \( \text{in}_2, \text{final} \) \( \rightarrow \) Task
   \( \text{or} \)

3. \( \text{in}_0 \) \( \rightarrow \) Task
   \( \text{in}_1 \) \( \rightarrow \) Task
   \( \text{in}_2, \text{final} \) \( \rightarrow \) Task

bool final;
If true, activate the task immediately
Activation model – Time driven

Relative-time

SpaceWire

Period

Offset

Relative-time

Task $\tau_1$

Task $\tau_2$

in

Task

class Event : public Channel
Tasking::Event

```cpp
class Event : public Channel

if (parent.shallFire())
{
    parent.onFire();
    static_cast<UnprotectedChannelAccess&>(parent).push();
}
```
Call semantics

1. Asynchronous (Default)

2. Synchronous (Tasking::Group)
Task model

Directed Acyclic Graph (DAG)

i₀ → τ₀ → ch₂ → i₂ → τ₂ → ch₄ → i₄ → τ₅ → ch₆ → i₆ → τ₆

i₁ → τ₁ → ch₃ → i₃ → τ₃ → ch₅ → i₅ → τ₅

i₈ → ch₈ → i₈ → τ₇

iMU

Optical navigation system (ATON)
Execution model
Execution model

- Work-conserving scheduling
- The load is balanced on the available executors

Thread’s life cycle:
- Dormant
  - create()
- Running
  - wait (&cond_var)
  - signal()
- Waiting
  - join()
Execution model – Scheduling

- Work-conserving scheduling
- The load is balanced on the available executors
- Global scheduling in tasking framework
- Hierarchy scheduling
Scheduling and priority handling

Application

Scheduler1
- Task1,0
- Task1,1
- Task1,n1

Scheduling policy

Global scheduling

ex1,1  ex1,m1

Scheduler2
- Task2,0
- Task2,1
- Task2,n2

Scheduling policy

Global scheduling

ex2,1  ex2,m2

Scheduler3
- Task3,0
- Task3,1
- Task3,n3

Scheduling policy

Global scheduling

ex3,1  ex3,m3

Scheduler4
- Task4,0
- Task4,1
- Task4,n4

Scheduling policy

Global scheduling

ex4,1  ex4,m4
Handling Events

```c
if (data->schedulerImpl->clock.isPending())
{
    data->schedulerImpl->handleEvents();
}
clock.startAt(*this, (nextActivation_ms + period_ms));
```

**Event Queue**

- Interface to the timer
- In Linux, it is a thread

**Periodic Schedule**

- Optimize sorting
- Event queue
Execution Platform

Platform Specific

SchedulerExecutionModel

Mutex

Signalizer

ClockExecutionModel
Unit Test
void
Tasking::SchedulerUnitTest::schedule(Tasking::Time timeSpan)
{
    // Step forward in time
    unitTestClock.step(timeSpan);
    // Doing execution
    SchedulerImpl& implementation = getImpl();
    do
    {
        implementation.handleEvents();
        for (TaskImpl* task = implementation.policy.nextTask(); (task != nullptr);
             task = implementation.policy.nextTask())
        {
            implementation.execute(*task);
        }
    } while (implementation.clock.isPending());
}
Unit tests

- main.cpp
- testBarrier.cpp
- testClock.cpp
- testSchedulePolicyLifo.cpp
- testSchedulePolicyPriority.cpp
- testTaskEvent.cpp
- testTaskGroup.cpp
- testSchedulerUnitTest.cpp
- testPeriodicSchedule.cpp
- testSchedulePolicyFifo.cpp
- testTask.cpp
- testTaskChannel.cpp
- testTaskInputArray.cpp
- testTaskInput.cpp
Unit tests

```cpp
#include <gtest/gtest.h>

int main(int argc, char **argv)
{
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}
```
Documentation
In-line comments

Documentation

doc/Tasking_framework.docx

Doxygen
Live Q&A session

Sunday 20.09.2020 at 11:00 – 12:00 am EDT at 05:00-06:00 pm Berlin

See you later!

Tutorial 4: Tasking Framework: An open-source software development library for on-board software systems

11:00am - 12:00pm, Sep 20

Tutorial

This tutorial is a live Q&A session with pre-recorded videos made available before the event.

Tasking Framework is a C++ software development library and an event-driven multithreading execution platform. It is developed by the Institute for Software Technology, German Aerospace Center (DLR). Tasking Framework is dedicated to improve the reusability in developing embedded software systems and to reconcile the embedded software with model-driven software development. It can be used to develop, but not dedicated for, critical as well as non-critical embedded software on single-core as well as parallel architectures. Tasking Framework gives software developers the ability to implement their applications as task graphs with arbitrary activation patterns (periodic, aperiodic and sporadic) using a set of abstract classes with virtual methods. It is compatible with the POSIX-based operating systems, mainly Linux and RTEMS. The Tasking Framework was successfully used in, for instance, the attitude orbit

Read more