FUSION FOR ENERGY

MARTe2 Users Meeting
Threading and services

Andre Neto, Filippo Sartori

May, 2019

Threads

- **Threads:** launch named threads bounded by to a specific CPU affinity (see ProcessorType) and with a defined amount of stack memory
- **Atomic:** portable functions that are guaranteed to be uninterruptible in a multi-threaded application environment
- **High Resolution Timer**: portable concept of ticks and time
- **Sleep:** voluntarily return the control back to the scheduler.
  - With the exception of **Sleep::Busy**

Warning

The change of some of the thread parameters (e.g. priorities) might fail if the user does not have the appropriate operating system permissions.

Semaphores

- **MutexSem**: mutex semaphore that offers exclusive access to a critical area;
- **EventSem:** barrier semaphore that, after being released, allows shared access to a critical area;
- **FastPollingMutexSem and FastPollingEventSem:** mutex and event semaphores that do not require an operating system scheduler. Both work by polling, with atomic operations, a given memory location.

Warning

The **Event::Reset** operation is not atomic. As consequence, depending on the specific use-case, it may require a mutex semaphore to protect access to a given shared resource while this operating is being performed.

Services

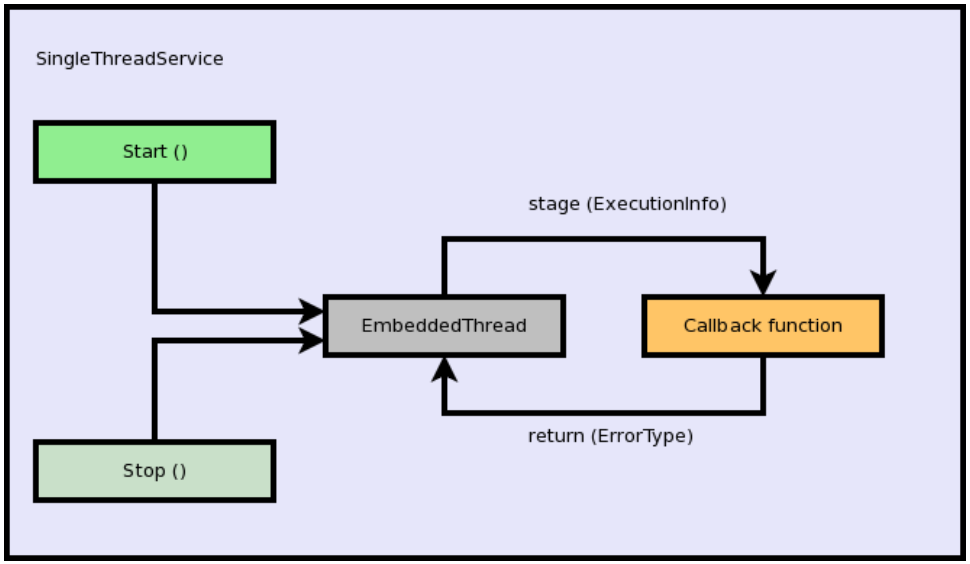
Promote uniformity in the management of threads, and in particular of the thread initialisation and destruction processes

- **Single-thread**
- **Multi-thread**
  - Each concurring for the same resource
- **Client oriented connection (e.g. web server)**
  - Automatically spawn a new thread to handle the connection
- All offers a Start and a Stop method.
  - Guarantees that the thread is killed if it not gracefully terminated by the application with-in a given timeout period.
  - All the thread parameters (affinity, stack, number of threads, …) can be changed using the specific service API or using the standard Initialise method.

Note
The callback function shall avoid blocking the execution and return as soon as possible.

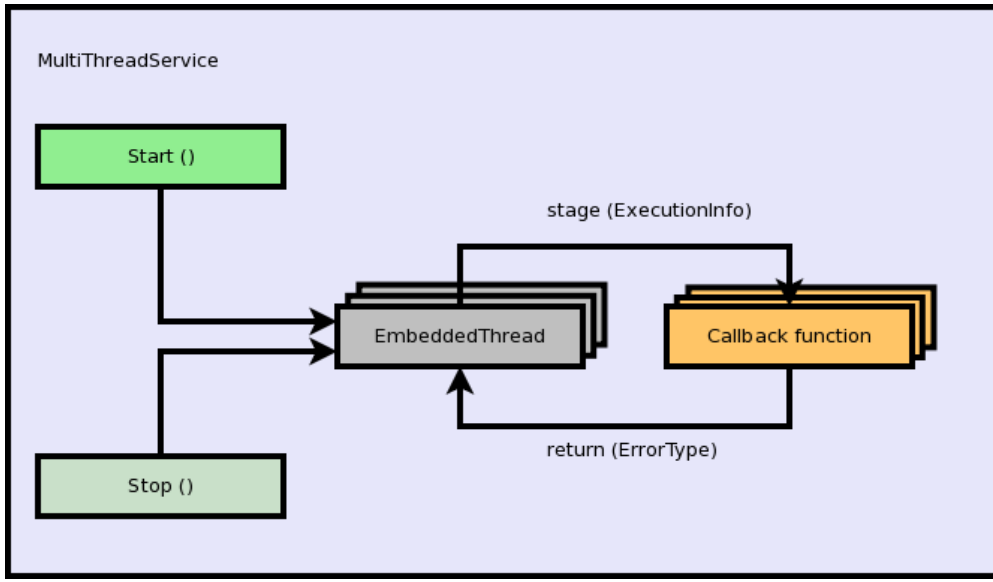
SingleThreadService

Stage	Condition(s)
StartupStage	After SingleThreadService::Start().
	If the callback returns an ErrorType which is not NoError and SingleThreadService::Stop() was not called.
MainStage	Until SingleThreadService::Stop() is called or if the callback returns an ErrorType which is not NoError.
TerminationStage	If the callback returns an ErrorType which is ErrorManagement::Completed.
BadTerminationStage	If the callback returns an ErrorType which is not ErrorManagement::Completedor if SingleThreadService::Stop() was called.
AsyncTerminationStage	If the thread was killed after trying to gracefully terminate with a SingleThreadService::Stop().



MultiThreadService

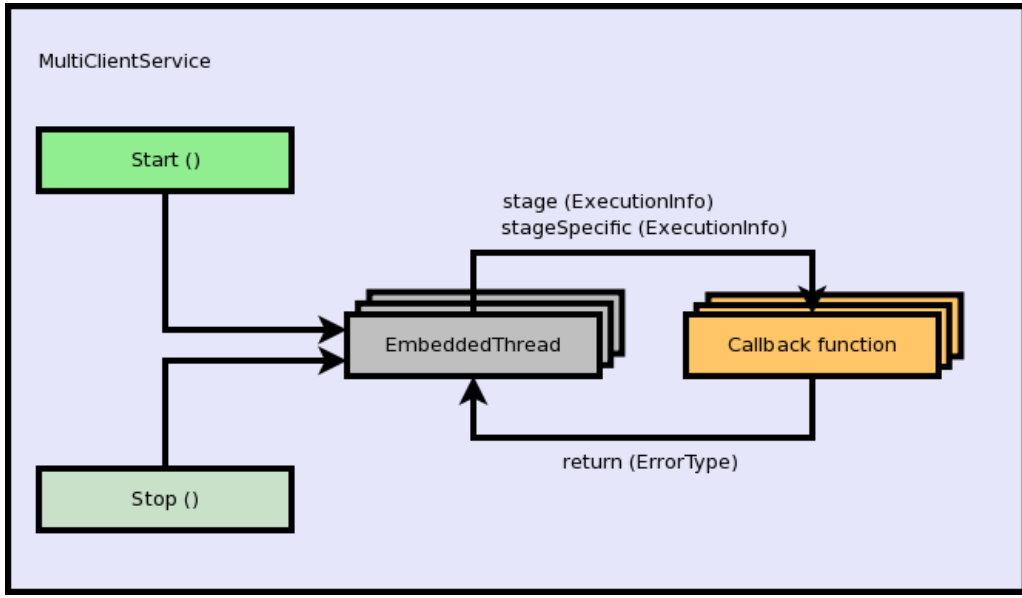
Stage	Condition(s)
StartupStage	After SingleThreadService::Start().
	If the callback returns an ErrorType which is not NoError and SingleThreadService::Stop() was not called.
MainStage	Until SingleThreadService::Stop() is called or if the callback returns an ErrorType which is not NoError.
TerminationStage	If the callback returns an ErrorType which is ErrorManagement::Completed.
BadTerminationStage	If the callback returns an ErrorType which is not ErrorManagement::Completedor if SingleThreadService::Stop() was called.
AsyncTerminationStage	If the thread was killed after trying to gracefully terminate with a SingleThreadService::Stop().



MultiClientService

- The number of threads is allowed to be increased/decreased by the service between the values defined by **GetMinimumNumberOfPoolThreads** () and **GetMaximumNumberOfPoolThreads** ().

StageSpecific	Condition(s)
WaitRequestStageSpecific	Wait for a connection request.
ServiceRequestStageSpecific	Serve a specific request.



Note

The callback should not block and should return ErrorManagement::Timeout while awaiting for a connection to be established. After a connection is established (ServiceRequestStageSpecific) the callback shall return ErrorManagement::Completed when the service has been completed.

Exercises

- Single thread Service & semaphores
- Further reading and examples:
  - https://vcis.f4e.europa.eu/marte2-docs/master/html/core/objects/objectsintro.html

SingleThreadService

Objective: learn how a single thread service can be used to decouple an activity

- Modify Other/Examples/SingleThreadServiceExample1.cpp
- Use a **FastPollingMutexSem** to guarantee that the value of the sharedResource always prints with a zero
- Compile

```
cd ~/Projects/MARTe2-demos-padova/
export MARTe2_DIR=~/Projects/MARTe2-dev
export MARTe2_Components_DIR=~/Projects/MARTe2-components/
make -f Makefile.x86-linux
```

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/
export LD_LIBRARY_PATH=.:$MARTe2_DIR/Build/x86-linux/Core/
Build/x86-linux/Components/Other/Examples/SingleThreadServiceExample1.ex
```

Success: application always prints the value of sharedResource with a zero

SingleThreadService II

Objective: learn how a single thread service can be used to decouple an activity

- Modify Other/Examples/SingleThreadServiceExample1.cpp
- Use a **MutexSem** to guarantee that the value of the sharedResource always prints with a zero
- Compile

```
cd ~/Projects/MARTe2-demos-padova/
export MARTe2_DIR=~/Projects/MARTe2-dev
export MARTe2_Components_DIR=~/Projects/MARTe2-components/
make -f Makefile.x86-linux
```

- Run the application

```
cd ~/Projects/MARTe2-demos-padova/
export LD_LIBRARY_PATH=.:$MARTe2_DIR/Build/x86-linux/Core/
Build/x86-linux/Components/Other/Examples/SingleThreadServiceExample1.ex
```

Success: application always prints the value of sharedResource with a zero

FUSION FOR ENERGY

Thank you for your attention

Follow us on:

www.f4e.europa.eu

www.twitter.com/fusionforenergy

www.youtube.com/fusionforenergy

www.linkedin.com/company/fusion-for-energy

www.flickr.com/photos/fusionforenergy