Timer - Periodicity and Synchronization

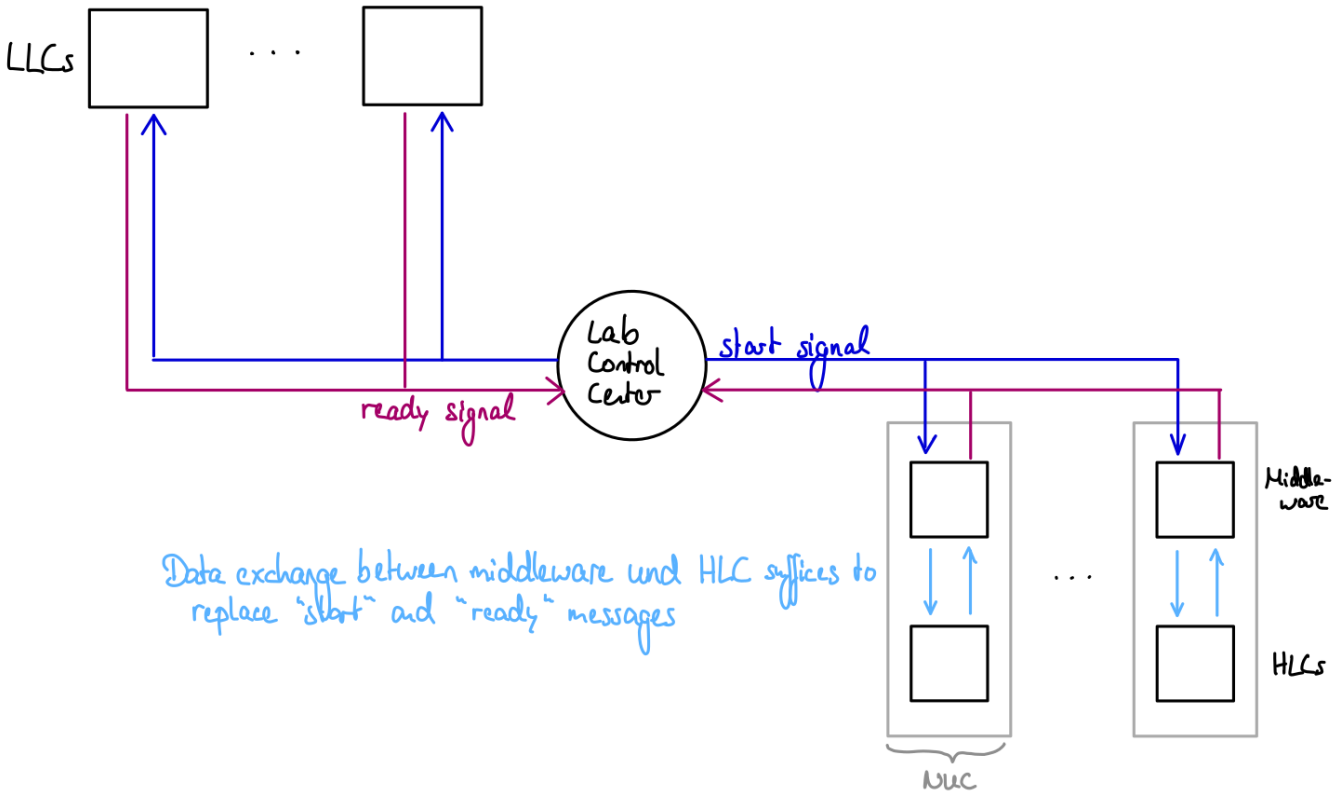
> **Important**
>
> The timer can also be used to implement periodical behavior in your program, as its callback function is called every X nanoseconds after the timer has been started (X is set by the user, and the timer must not be started externally if you set it not to listen for start signals).

The CPM timer is used to achieve a **synchronized behavior across different devices** that cooperate in the network. For example, all vehicles are supposed to publish their current state (like battery voltage) **periodically** at the same time. Such a behavior can be achieved with the CPM timer. Timing is either performed using the current real time (system clock, synchronized using NTP) or a fictitious time, which is distributed by a central timing instance.

Synchronization across all participants in the RTI DDS network is very important for all tasks that

- are performed on a regular basis and
- require up-to-date information from other participants in the network to perform these tasks properly

Synchronization allows to coordinate different tasks across the network and gives some guarantees about the shared information. Some of these tasks are caused e.g. by the mutual dependency between LLCs and HLCs. LLCs require regularly updated commands from the HLCs to maneuver crash-free and as desired according to e.g. their planned trajectory. HLCs need up-to-date vehicle states from the LLCs to calculate commands that accommodate the current state of the overall system.



CPM Timer (Timer.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/Timer.hpp

The timer class can be used to call tasks periodically, based either on the **system clock** (**TimerFD**.hpp) **or the simulated time** (**TimerSimulated**.hpp, timing signals are given by the LCC). It **can wait for a central start signal** and can be **stopped when a stop signal is sent** (both by the LCC). Thus, timers on multiple systems can be started at the same time, and then work simultaneously if their period is set accordingly as well.

Several parameters must be set to use to cpm timer:

- node_id (String): The unique ID identifying the timer in the network, so that the LCC can tell different timer signals apart
- period_nanoseconds (uint64_t): The callback function of the timer is called every period_nanoseconds nanoseconds

- offset_nanoseconds (uint64_t): Initial offset - 1 in real-time case (offset from unix timestamp 0), first callback time in simulated time case
- wait_for_start (bool): If false, real-time timing is started immediately after creation without waiting for the LCC - this parameter is ignored by the timer that uses simulated time, as there it is required to listen to a central timing instance
- simulated_time_allowed (bool): Specify whether the task can be performed using simulated instead of real-time
- simulated_time (bool): Indicates if simulated or real time should be used

The callback function is registered within the `start(...)` function. The timer can be started synchronously, blocking / using the current thread until it was stopped, or asynchronously in a new thread, using `start_async(...)`. It can either be stopped manually using `stop()`, which is especially useful if the LCC is not used for real-time timing purposes, or it can be stopped with an LCC command.

The **callback function** registered in start **is called periodically** until the timer is stopped, according to period, offset and the usage of simulated time. The following figures illustrate the role of the timer in combination with the LCC as central timing instance.

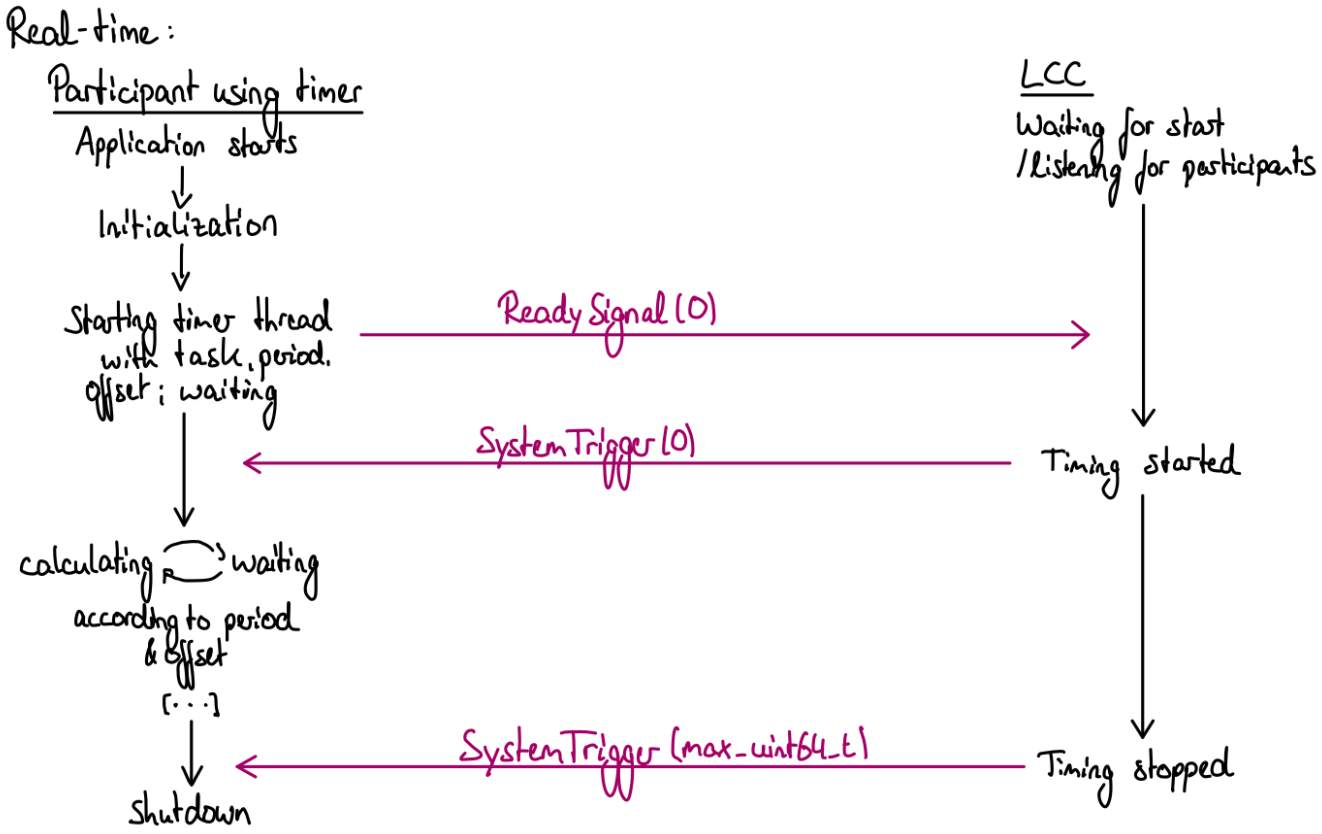
There is also an option to set a **callback for the stop signal**: If it is set, the timer is not stopped when a stop signal is received, but instead calls this function. You can set an empty callback to ignore stop signals, or define your own stopping behavior. You can call stop() at the end of the callback on the timer object to actually stop the timer itself afterwards.

Real-time (TimerFD.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/TimerFD.hpp

Real-time refers to the **actual current time of the system**, given by the **system's current timestamp**. To make sure that timestamps across different devices are comparable, **NTP is used for clock synchronization** on these devices. In a real-time scenario, all participants begin their computation (after being started either manually or by the LCC) at unix time `1 + period * n`, where n is the smallest n such that the overall term is greater than or equal to the point in time when the timer was started. This assures synchronized periodic behavior for the registered components (using callbacks) across all devices. Real-time is useful whenever the system is used for real driving scenarios, e.g. when the actual physical vehicles are used.

The following drawing shows which role the LCC plays in the real-time timing case.



This timer's constructor also takes an **optional parameter to re-define the stop-signal** to another value than max. uint64_t. *Do not change this value unless you know what you are doing* and only if you want your timer not to react to a global stop signal within the system, but a custom one. (If you just want **to ignore the stop signal, define your own**, potentially empty, **stop callback function** and pass that when starting the timer **instead**).

The timer also allows to get its start time. Most other details were already explained in the **CPM Timer** section.

> **Important**
>
> Only use TimerFD specifically if you do not need your timer to be able to switch to simulated time!

"Real" Time (SimpleTimer.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/SimpleTimer.hpp

This timer is **not intended to be used for actual real-time** purposes. It can be used instead **for GUI tools etc. to achieve periodic threaded callback** of a function in **periods** which are **multiples of 50 milliseconds**. Apart from that, it works like the timer above (TimerFD).

This timer can be killed easier - while a **TimerFD** timer might take **up to period_nanoseconds** nanoseconds to be killed, the **SimpleTimer is killed** or destructed **within ~50 milliseconds** and thus does not block e.g. the UI for too long.

The support for a custom stop signal was mainly implemented to be used with this timer - if the SimpleTimer is used for other purposes than simulation, then it might be undesired that the simulation's stop signal can stop this timer. To stop multiple running SimpleTimers at once, it might still be in the user's interest to use a stop signal (differing from the one used in the simulation).

If you just need threaded periodic behaviour in your code, without any real-time requirements, then use this timer instead of an own implementation whenever possible.

Simulated time (TimerSimulated.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/src/TimerSimulated.hpp

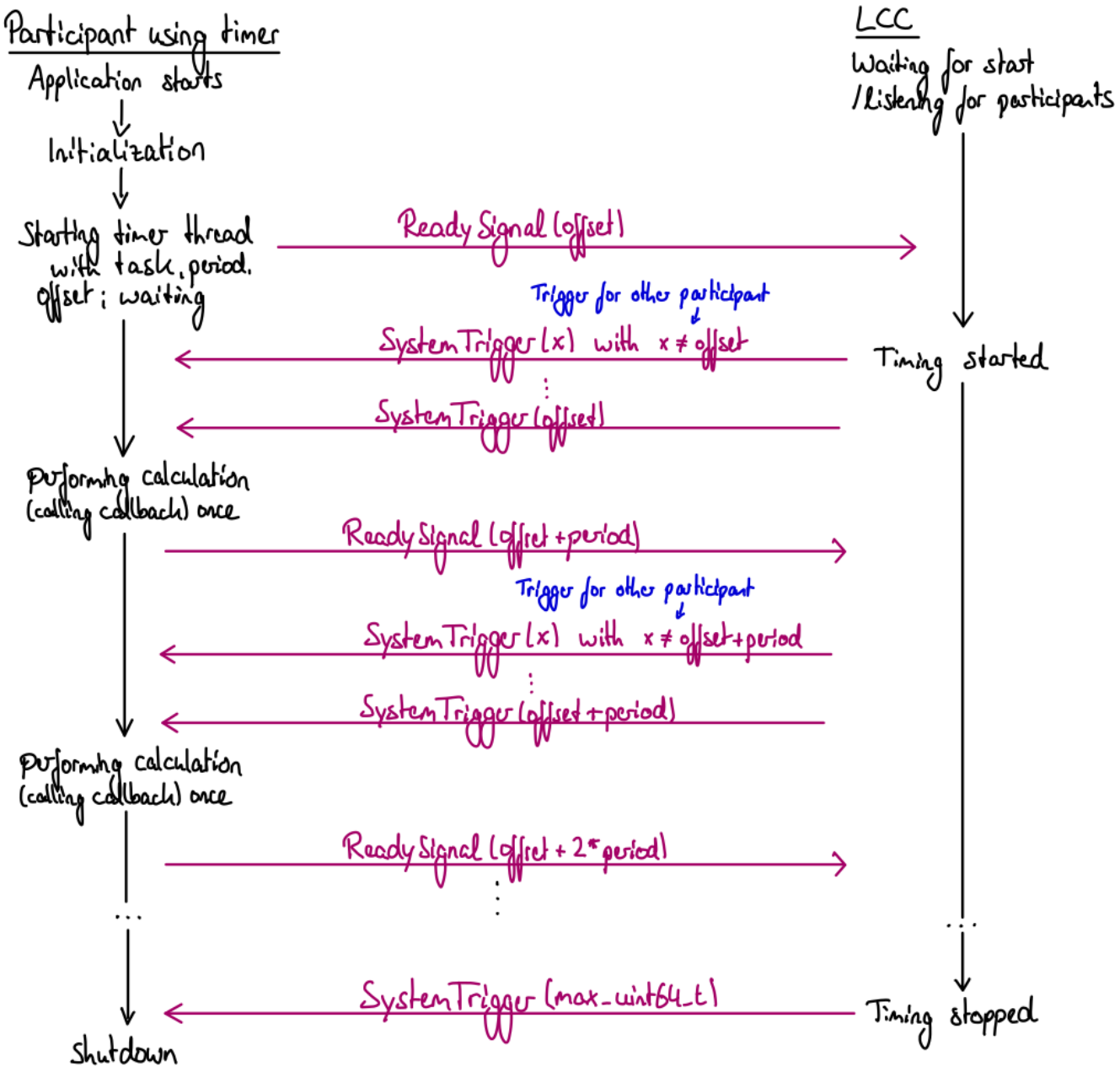
Simulated time refers to a fictional measure of time, that is not related to the unix time of the physical devices on which the different tasks operate. Instead, the current time is determined by a **central timing instance**, in this case by the LCC. Simulated time can be **used to speed up or slow down the simulation** of a scenario, which might help to detect errors, to improve the timing of the individual components or to test faster how the system operates on the long run.

In a simulated time case, it is **required that all participants are uniquely identifiable** (here by setting **`node_id`** differently for all participating tasks). After being invoked on their currently set time step, the participants register the next time when their callback function should be called according to the fictional time measure, which is based on the offset and period settings. Initially, the LCC collects all these ready signals for the first timer callbacks until timing is started manually by the user (or until ready signals from all pre-registered timers have been received). It then performs the following tasks in a loop:

- Select the smallest next fictional time stamp of all received timestamps (that is greater than the current simulated time) and send a system trigger to all participants containing this time stamp
- Wait for all participants that registered this timestamp to register a new, greater timestamp

The simulated time progresses this way until the timing is stopped in the LCC. The time can of course only progress if all participants send new ready signals after they have been invoked (and after the work of their callback function has been done).

The following drawing illustrates which role the LCC plays in a simulated time use case.



Important

Try not to use TimerSimulated specifically - use the CPM Timer instead, which can switch between real and simulated time based on the settings taken in the LCC.

LCC Timer

As mentioned before, the usage of **simulated time and start and stop signals** requires a central timing instance. This instance is **located in the LCC**. Timing can be controlled by the user using the LCC's **UI**.

The central class for controlling the timing in the system is `TimerTrigger`. In a real-time scenario, the class is merely used to send start and stop signals, and to keep track of which participants initially registered with a ready signal. These participants are shown in the UI, so registering them allows the user to see if they perform as expected up to the creation of the timer - if they do not show up, something must have gone wrong in the application.

The class is also responsible for handling simulated time. Here, further tasks are required. The current simulated time as well as the newest ready signals of the participants need to be stored. From the current time and the participants that rely on it, the system can also infer whether a participant is currently working or waiting for another time step, and if it is out of sync (if only old ready messages are received by it). The class receives and processes all ready signals so that the progression to the next smallest time step works properly. Of course, this also includes waiting for participants that are currently working and have not yet registered a new time step in which they want to be woken up again.

get_time_ns.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/get_time_ns.hpp

Whenever you want to **obtain the current system real-time** (as timestamp, in nanoseconds, using clock_gettime internally), use this function. This can be useful to e.g. obtain timestamps for your messages.

stamp_message.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/stamp_message.hpp

The function name is self-explanatory. If your message types includes and uses Header (https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/dds_idl/Header.idl), you need to **set a create and valid_after stamp**. Both can be set **in one line** using this function, where you pass your message object as reference so that the fields can be set on the original object. It is thus merely a **convenience function** that you are not required to use.

The create stamp usually should be the time of message creation.

The valid_after stamp tells the receiver when the message should become valid, e.g. right now (same as create stamp), in the near future (i.e. create stamp + 1000000 nanoseconds) etc.

References

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/lab_control_center/src/TimerTrigger.hpp

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/lab_control_center/ui/timer