OS timing hooks

Generic trace interface

Specification – Version 1.4

www.gliwa.com

GLIWA GmbH embedded systems
Pollingerstr. 1
82362 Weilheim i.OB.
GERMANY

fon +49 - 881 - 13 85 22 - 0
fax +49 - 881 - 13 85 22 - 99
info@gliwa.com
www.gliwa.com

Document ID: 6-1017-30-20-10-10

To support the fast and reliable integration of 3rd party timing solutions with an OS scheduler, we propose a standard interface of "hooks" (hook routines) in the OS. If the OS is supplied as source, these could be implemented with macros. If the OS is supplied as library code, these must be implemented as callouts. We justify our choice of measurement points in the context of the accurate timing measurement required by real automotive projects.

Release	*Date*	*Author*	*Comment*
1.0	2012-09-07	Peter Gliwa	First released version
1.1	2013-03-05	Peter Gliwa	Minor corrections, appendix added
1.2	2016-03-18	Alexandre Baufumé	Document template update, renaming and example updated, comments updated in ostimhooks.h
1.3	2017-02-15	Alexandre Baufumé, Peter Gliwa, Nick Merriam	1. Example reworked, comments updated in ostimhooks.h 2. update to latest layout 3. Introduced "Conformance Options", see 2.1 4. Added macro FAILACT to log failed task activations 5. Improved section 3.3 6. Added section 2.4 7. Added figures 8. Add missing RELEASE event 9. Renamed event RESUME to CONTINUE
1.4beta4	2017-05-02	Peter Gliwa	1. added figures and explanations 2. removed failed activation from task state diagrams 3. introduced NST (net slack time) 4. renamed event ACTIVATION to ACT 5. introduced events PSTART, ACT_START, STOP_ACT_START, RNEXT, RSTART, RSTOP

1.4beta5	2017-06-02	Peter Gliwa	1. removed ACT_START and STOP_ACT_START 2. added hook parameters 3. added header template at the end of the doc
1.4	2018-01-22	Peter Gliwa, Alexandre Baufumé, Andreas Knickeberg	1. added alternative to ambiguous ECC task, see listing 3 2. bug-fix: parameter `classId_` was missing for `OSTH_STOP_PSTART_NOSUSP` in listing 5 3. bug-fix: replaced missing definitions of macros `OSTH_STOP_(P)START_USER` in listing 5 4. added section 3.4 5. added instrumentation for non-terminating ECC tasks in case an event is already set before `WaitEvent` is called

Table 1: Document History

Contents

1 Introduction

To support the fast and reliable integration of 3rd party timing solutions with an OS scheduler, we propose a standard interface of "hooks" (hook routines) in the OS. The AUTOSAR/OSEK OS standard defines pre and post task hooks and it has been suggested that these hooks are suitable for timing debugging and measurement. Whilst the OSEK OS task hooks are ideal for other purposes, they have a number of characteristics that make them unsuitable for timing instrumentation:

1. The post task hook does not distinguish between a task being preempted and a task exiting. Similarly, the pre task hook does not distinguish between a task being resumed after preemption and a task starting.

2. Each transition from one task to another requires two hook routines. Firstly, this is inefficient at a potentially time-critical part of the schedule. Secondly, this tends to lead to the time between the two hook routines being unaccounted to any task, which means there is CPU load that cannot be properly budgeted for timing protection, scheduling or CPU load prediction. This makes it impossible to accurately predict real-time behaviour from the timing measurements.
    - Worse still, pre and post task hooks are not called for the idle task, preventing consistent measurement of timing in the idle task.

3. Rather than receiving the identity of the task being entered of left, user code in the hook routines has to use the OS service GetTaskId. To give an indication of how inefficient this is, we have observed a real project with a powerful 150MHz embedded CPU where 0.5% of the entire CPU load was consumed just calling GetTaskId in the task hooks.

4. The OS task hooks are explicitly *not* intended for use in a production system. Timing protection at the task level is provided for by the AUTOSAR OS timing protection mechanism. If the system safety concept requires that timing is controlled at a finer granularity, for example at software component boundaries, then such timing control cannot be implemented.

As a result, we propose hooks specifically intended for timing debugging and measurement that avoid the problems listed above.

If the OS is supplied as source, these hooks could be implemented with macros. If the OS is supplied as library code, these must be implemented as call-outs. We justify our choice of measurement points in the context of the accurate timing measurement required by real automotive projects.

Key new aspects of our approach include explicit treatment of the following:

- Measurement limitations and the measurement errors that arise
    - In contrast, other approaches have ignored measurement errors, with the result that they remain uncorrected in the final timing data. At best this leads to wasted capacity, at worst it leads to incorrect timing designs.

- Use of timing measurement results for scheduling analysis, including the consequence of errors

- Supervisor and user mode contexts
- Contexts (user mode) that can and cannot disable interrupts
- Contexts (user mode) that can and cannot write to shared memory
- Timing on multiple, parallel cores

Section 1.1 gives a brief summary of the OSEK/AUTOSAR OS task states and based on this summary, section 1.2 defines timing parameters. We present the hooks themselves in Section 2. The justification, in terms of measurement advantages, is explained in Section 3.

1.1 OSEK/AUTOSAR OS task states

AUTOSAR OS uses the scheduling concept as defined by OSEK. OSEK defines task-states for two different conformance classes, BCC (Basic Conformance Class) and ECC (Extended Conformance Class). The corresponding task-state diagrams are shown in figures 1 and 2.

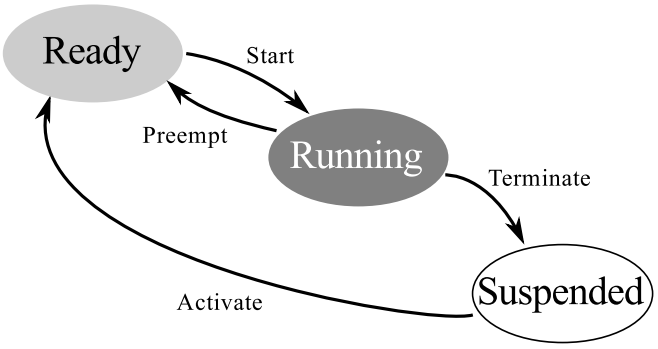

Figure 1: Task states and transitions as defined by AUTOSAR OS BCC

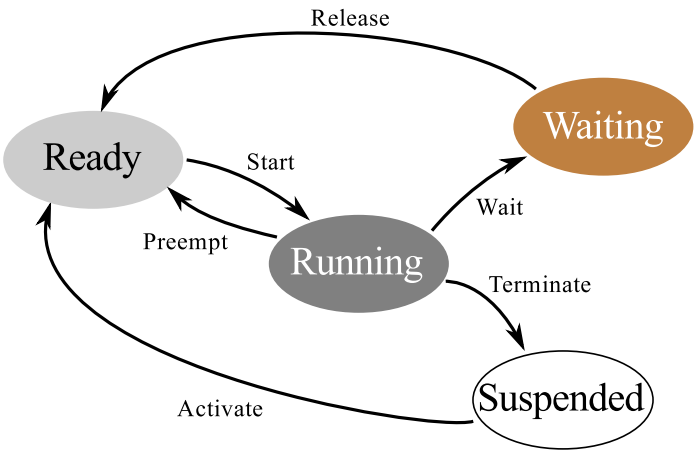

Figure 2: Task states and transitions as defined by AUTOSAR OS ECC

1.2 Timing parameters

Figure 3 shows the principle timing parameters of a task that determine its real-time behavior within a system and table 2 defines the symbols used. Note that the color used to indicate a task's current state at a given point in time corresponds to the color used for this state in figure 2.

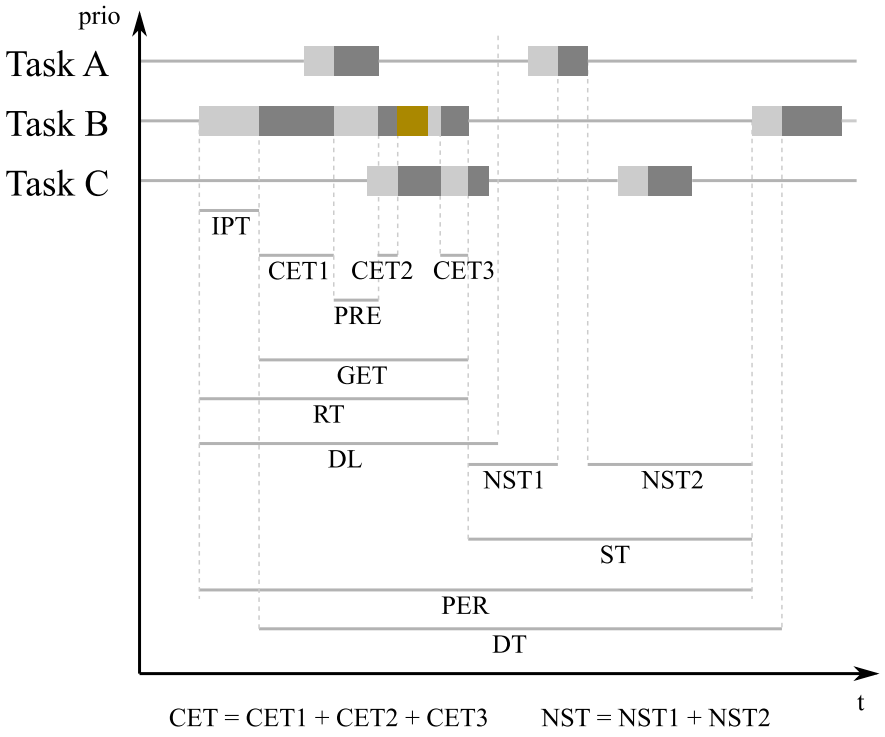

Figure 3: Timing parameters visualised in a trace (all related to TASK B)

1.3 Comments on AUTOSAR OS ECC

Typically, AUTOSAR OS tasks get started and then terminate at some point in time. This is absolutely mandatory for tasks of the AUTOSAR OS basic conformance class (BCC) and should also be the case for AUTOSAR OS extended conformance class (ECC) tasks.

However, there *are* set-ups with tasks that do *not* terminate but rather loop, using WaitEvent for scheduling. This is often true for RTE tasks being generated by the RTE configuration environment. See listing 1 for an example. Rather than having two periodical BCC tasks – e.g. `Main_Task_5ms` calling `CanTp_MainFunction` and `CanXcp_MainFunction` as well as `Main_Task_10ms` calling `CanNm_MainFunction` and `CanSM_MainFunction` – the RTE configurator generates a non terminating ECC task and adds a second level of scheduling being controlled by `WaitEvent` and `SetEvent`.

Listing 1: Non terminating ECC task using events for scheduling

```
TASK(Main_Task)
{
  EventMaskType ev;
```

ID	Abr.	Name EN	Description
1	IPT	initial pending time	from activation to start
2	CET	core execution time (computation time)	execution time not including any preemptions or “waiting” time
3	GET	gross execution time	execution time including all preemptions and “waiting” time
4	RT	response time	from activation to termination
5	DL	dead line	max. allowed response time
6	DT	delta time	from start to start (“measured period”)
7	PER	period	from activation to activation (period not as measured but as configured)
8	ST	slack time	“remaining” run-time: from termination to activation (tasks) or start (interrupts)
8	NST	net slack time	“potential additional” run-time: the ST minus all CET blocks of any TASKs or ISRs with higher priority during the ST
10	JIT	jitter	deviation of delta time from period

Table 2: Timing information

```
for(;;)
{
  (void)WaitEvent( Rte_Ev_Cyclic2_Main_Task_0_10ms |
                   Rte_Ev_Cyclic2_Main_Task_0_5ms );

  (void)GetEvent(Main_Task, &ev);

  (void)ClearEvent(ev & ( Rte_Ev_Cyclic2_Main_Task_0_10ms |
                      Rte_Ev_Cyclic2_Main_Task_0_5ms ));

  if ((ev & Rte_Ev_Cyclic2_Main_Task_0_10ms) != (EventMaskType)0)
  {
    CanNm_MainFunction();
    CanSM_MainFunction();
  }

  if ((ev & Rte_Ev_Cyclic2_Main_Task_0_5ms) != (EventMaskType)0)
  {
    CanTp_MainFunction();
    CanXcp_MainFunction();
  }
}
```

We will not elaborate on all the disadvantages of this approach at this point but we have to address non-terminating ECC tasks and allow timing analysis also for this case. The previous definition of the CET e.g. fails. For terminating tasks (BCC as well as ECC), the CET was defined as the sum of all “running” states between the start and the termination of the task. Obviously, the CET becomes infinite if the task does not terminate.

Figure 4 resembles figure 3 but now Task B is a non-terminating ECC task. Whoever implemented the task would expect the timing properties to be computed for one “round” of the endless-loop. The gap between two subsequent rounds reflects a pseudo suspended state for Task B and thus is visualized with transparency added to the wait-

ing state in figure 4.

Since the loop might include the usage of “regular” events, we now have to distinguish such “regular” events and their corresponding `WaitEvent` call from the events used for scheduling and *their* corresponding `WaitEvent` call. Listing 2 is derived

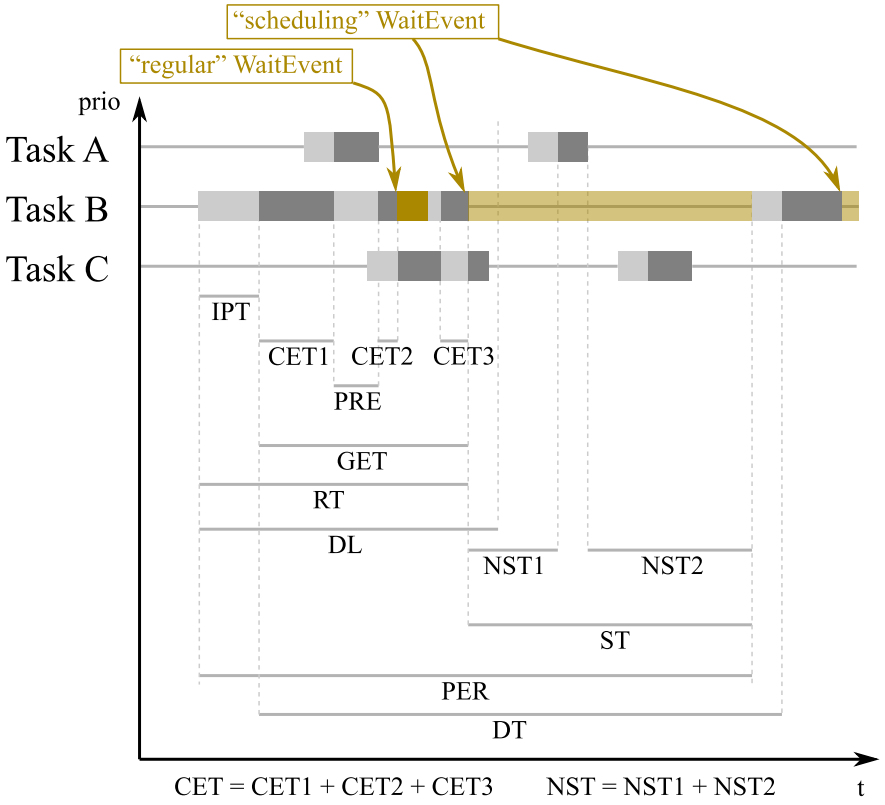

Figure 4: Timing parameters related to TASK B (here a non-terminating ECC task)

from listing 1. Comments have been added for explanation and to indicate when the task changes its state. Additionally, the task now also has a “regular” event `Can_Ev_TriggerSM_Main_Task`. The scheduling situation shown in figure 4 corresponds to listing 2.

Listing 2: Non terminating ECC task using events for scheduling

```
TASK(Main_Task)
{
  // Task starts here
  EventMaskType ev;

  for(;;) // non-terminating ECC task
  {
    // Task "ends" here (in fact it will switch to waiting)
    // the following WaitEvent call is a "scheduling" WaitEvent
    (void)WaitEvent( Rte_Ev_Cyclic2_Main_Task_0_10ms |
                     Rte_Ev_Cyclic2_Main_Task_0_5ms );
    // Task "starts" here again (in fact it returned from waiting)

    (void)GetEvent(Main_Task, &ev);

    (void)ClearEvent(ev & ( Rte_Ev_Cyclic2_Main_Task_0_10ms |
```

```
                        Rte_Ev_Cyclic2_Main_Task_0_5ms |
                        Can_Ev_TriggerSM_Main_Task ));

    if ((ev & Rte_Ev_Cyclic2_Main_Task_0_10ms) != (EventMaskType)0)
    {
      CanNm_MainFunction();
    // the following WaitEvent call is a "regular" WaitEvent
      (void)WaitEvent( Can_Ev_TriggerSM_Main_Task );
      CanSM_MainFunction();
    }

    if ((ev & Rte_Ev_Cyclic2_Main_Task_0_5ms) != (EventMaskType)0)
    {
      CanTp_MainFunction();
      CanXcp_MainFunction();
    }
}
```

The recommended task configuration for the same set-up is shown in listing 3. For each period – here 5ms and 10ms – it uses a dedicated task. Whenever possible, the task should be a BCC1 task. All tasks terminate.

Listing 3: Recommended configuration using a separate task per period

```
TASK(Main_Task_10ms) // ECC
{
    CanNm_MainFunction();
    // the following WaitEvent call is a "regular" WaitEvent
    (void)WaitEvent( Can_Ev_TriggerSM_Main_Task );
    CanSM_MainFunction();
    TerminateTask();
}

TASK(Main_Task_5ms) // BCC1
{
    CanTp_MainFunction();
    CanXcp_MainFunction();
    TerminateTask();
}
```

2 Timing hooks

The timing hooks described in this section support the fast and reliable integration of 3rd party instrumentation based timing solutions with an OS scheduler.

2.1 OStimHooks Conformance Options (OCO)

Instrumentation-based timing measurement or tracing has an impact on the software. The instrumentation is always a trade-off between many details with a high impact/overhead or fewer details with a smaller impact/overhead. In order to offer a scalable interface, *Conformance Options* define certain sets of macros and are identified by their ID (OCO1, OCO2, etc.).

Except for OCO2 the following general rule applies: the more options are chosen, the more events are logged, the more details are gained and the higher the impact/overhead. OCO2 includes a set of macros which combine more than one transition in one single event reducing the impact/overhead and – at the same time – increasing precision.

Table 3 shows all Conformance Options and a brief description for each. The colors found in the table are also used in the figures mapping the events onto the task-state schemes.

Conformance option ID	*Description*
OCO1	Macros related to task activations
OCO2	Macros for logging combined events for higher precision and more efficiency compared to separate events
OCO3	AUTOSAR events realated macros logging entry into and exit from the AUTOSAR OS ECC task state “waiting”
OCO4	Macros for logging events related to interrupt suspension or resources
OCO5	Macros for logging error related events
OCO6	Macros for logging runnables

Table 3: OStimHooks conformance options (OCO)

2.2 Definition of the timing hooks

Table 4 shows the OS events to be instrumented and the required hooks for tracing or measuring the timing properties in table 2. To get a complete hook name, the prefix string `OSTH_` needs to be put at the beginning and the the correct suffix from table 5 according to the context from which the hook is called needs to be appended.

`OSTH_ACT_NOSUSP` is one example for a complete hook name, `OSTH_LOCK_STOP_SPRVSR` another. In the following we will use only the relevant part of a hook name.

Where multiple events may be logged, the component parts are joined with underscore “_” to make this “first part”, for example `START_STOP`.

`STOP_START` should always be used in preference to separate stop and start hooks if at all possible. Separate hooks leave a gap that cannot be correctly attributed to any task, whereas `STOP_START` ensures a continuous trace with no unattributed gap, see section 3.4 “Instrumentation gaps”.

The macros and events are defined in a way that blocking can be considered correctly. Blocking occurs when a task or ISR of higher priority than the running task is inhibited from preempting by a resource lock.

2.2.1 Parameters passed to the hoooks

The parameters passed to a hook depend on its type and the context.

schedId identifies a schedulable. A schedulable is either a TASK or an ISR.

lockId Implementation specific. Might be e.g. an ID of a spinlock or the ID of a resource or it might be a priority.

runnableId identifies a runnable.

coreId For single core applications, this parameter is ignored. for multi core applications the coreId identifies the core on which the event related to the hook takes place.

classId This may be ignored but it can be used to split the instrumentation into classes such that the instrumentation in a class cannot be preempted by instrumentation in the same class.

No.	*Event description*	*Task state transition*	*Para-meter*	*First part of hook name*	*Conformance option*
1	*Prompt Start* of a task or interrupt that appears to switch from Suspended to Running "promptly" as no ready state was observed.	Suspended → Running	schedId	PSTART	mandatory
2	*Termination* of a task or *end* of an interrupt	Running → Suspended	schedId	STOP	mandatory
3	Successful task *activation*	Suspended → Activated (Ready)	schedId	ACT	OCO1
4	*Start* of a task or interrupt	Activated (Ready) → Running	schedId	START	OCO1
6	Very short ISR where only one hook is possible	Suspended → Running → Suspended	schedId	PSTART_STOP	OCO2
7	End of one task/ISR and the start of the next without return to a preempted context. The parameter schedulableId indicates the schedulable which is started. The schedulable which is stopped is identified by implication.	Running → Suspended for one task and Ready → Running for the next or Running → Running for the same task (reset CET calculation)	schedId	STOP_START	OCO2
9	*Continuation* of a terminating task which previously was in the Waiting state	Released (Ready) → Running	schedId	CONTINUE	OCO3
10	*Suspension* of a terminating task	Running → Waiting	schedId	SUSPEND	OCO3
11	*Release* of a terminating task	Waiting → Released (Ready)	schedId	RELEASE	OCO3
12	Commence lock of resource/interrupt	none	lockId	LOCKING	OCO4
13	Complete lock, especially spinlock	none	lockId	LOCKED	OCO4
14	Unlock resource/interrupt	none	lockId	UNLOCK	OCO4

15	*Failed activation*, specifically task over-activation (error E_OS_LIMIT as defined in AUTOSAR OS)	none	schedId	FAILACT	OCO5
16	*Killing* of a task or interrupt	undefined	schedId	KILL	OCO5
17	*End* of one runnable (if any) and *Start* of the next runnable (if any). Works only for a fixed, unconditional runnable to task mapping. No parameter required: the corresponding schedulable will be derived from other events.	none	none	RNEXT	OCO6
18	*Start* of a runnable	none	runnableId	RSTART	OCO6
19	*End* of a runnable	none	runnableId	RSTOP	OCO6

Table 4: Hooks (first part of the hook name indicating the event)

Context description	*Second part of hook name*
Interrupts are disabled when hook is called	_NOSUSP
The called hook may disable interrupts	_SPRVSR
The called hook *cannot* disable interrupts	_USER

Table 5: Hooks (second part of the hook name indicating the context)

2.3 Task states

Figures 5, 6 and 7 show task state diagrams which correspond to some of the Conformance Options. Note that the arrows between the states now reflect events rather than transitions. Exception: the transitions “Preemption” and “Resumption”. These transitions can be deducted from a trace and thus there is no need for a corresponding event.

The numbers in circles correspond to the column “ID” in table 4 and the color of the event arrows reflect the corresponding Conformance Option as defined in table 3.

Side remark: the first occurrence of Task B in figure 3 shows three blocks where the task is in the AUTOSAR OS *Ready* state. With the state scheme presented in figure 7, we can describe the run-time situation with a greater level of detail. The first *Ready* block of Task B reflects the *Activated (Ready)* state, the second the *Ready (Ready)* state

– as a result of a preemption by Task A – and the third block indicates the *Released (Ready)* state reflecting that the event Task B had been waiting for was set.

Figure 7 shows also the `STOP_START` event which is used to reset the calculation of the CET for a non-terminating ECC task when a "scheduling" `WaitEvent` is called and one of the events that task shall wait for is already set. In this case normally no rescheduling takes place and therefore the task remains in the "Running" state. Therefore it is mandatory that the OS provides a hook for this scenario.

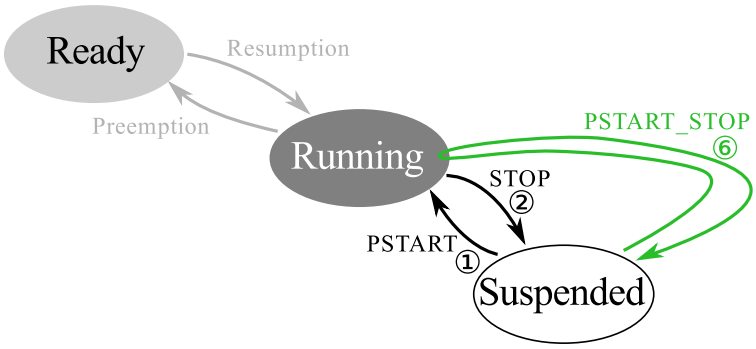

Figure 5: Minimalistic task state scheme with the mandatory start and stop events

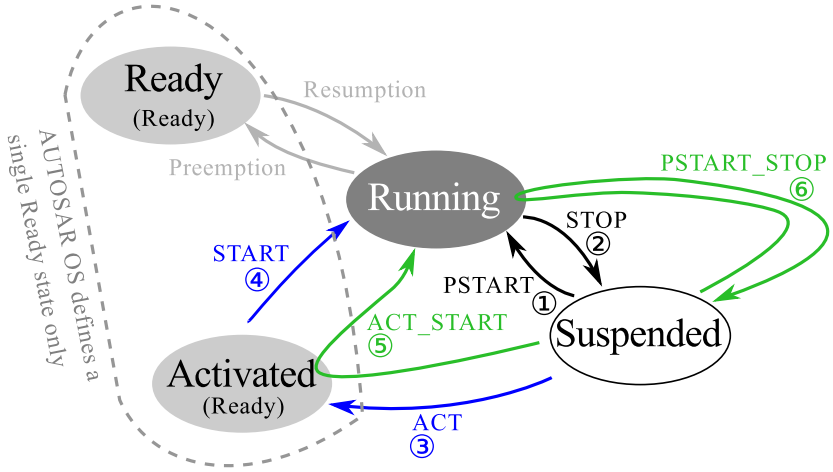

Figure 6: Task state scheme representing a BCC set-up

2.4 Run-time situation example

Figure 8 illustrates a run-time situation with three tasks and three interrupts. The numbers in the figure correspond to the numbers in the first column of table 4.

2.5 Instrumentation of non-terminating ECC tasks

Following the thoughts of section 1.3 "Comments on AUTOSAR OS ECC" on page 9, this section describes how to instrument non-terminating ECC tasks.

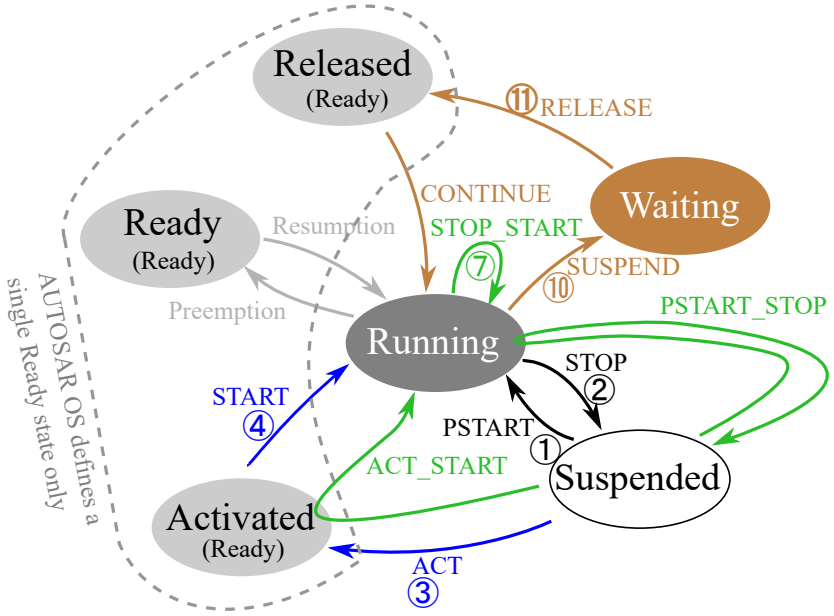

Figure 7: Task state scheme representing an ECC set-up

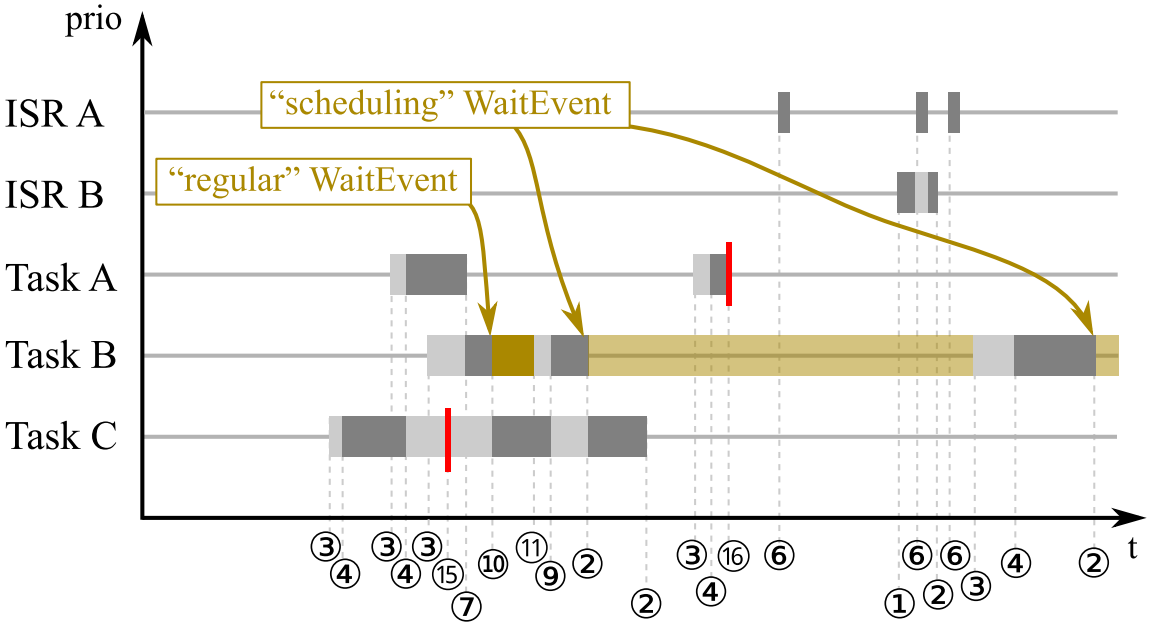

Figure 8: Run-time situation with most of the events (missing: 12-14; 17-19)

Non-terminating ECC tasks should be instrumented using stop and start events. This is because each return from WaitEvent is to be treated as the start of a new task instance, where the CET should be reset to zero. The suspend and resume events are only to be used for tasks where the return from a call to WaitEvent should *not* be considered the start of a new task instance and CET should continue to accumulate. See also figure 8

The hooks are located in places that both allow the OS easy access to the arguments to the hook macros or functions and that satisfy the constraints described in Section 3.

2.6 Instrumentation of runnables

Runnables are not directly related to scheduling, they are part of the code that gets executed when a task gets executed. However, since runnables play a fundamental role in the AUTOSAR concept, it makes sense to also standardize the instrumentation of runnables for two reasons:

1. Runnables which get traced can be visualized.
2. Some of the timing parameters can be calculated for runnables, namely CET, GET and DT.

Table 4 defines three macros related to runnables. The functionality of RSTART and RSTOP is obvious, they mark the beginning and the end of a runnable. However, having two events per runnables means a significant tracing overhead when tracing all runnables at the same time. Today's engine management systems typically have more than 2000 runnables so the overhead for tracing all of them using RSTART and RSTOP would be significant.

Although AUTOSAR does not exactly specify how runnables shall be called, the typical implementation places the corresponding function calls sequentially into a task. There might be `Schedule()` OS service calls in order to allow task switches for a NON-PREEMPTABLE task but this is not regarded as a runnable. Additionally, there is the final function call to `TerminateTask()` but apart from that all other function calls are typically runnables. See listing 4 for an example.

Listing 4: Typical task implementation: task calls runnables

```
TASK(TaskB)
{
   MyRunnable1();
   MyRunnable2();
   Schedule();
   MyRunnable3();
   Schedule();
   MyRunnable4();
   MyRunnable5();
   MyRunnable6();
   TerminateTask();
}
```

Figures 9 and 10 show an identical run-time situation where Task B with its 6 runnables gets preempted by Task A. Figure 9 shows the instrumentation with one RSTART and one RSTOP event for each runnable. Figure 10 shows the instrumentation with one RNEXT event between the runnables. The instrumentation with one RNEXT event between the runnables requires a strictly static and non-conditional mapping of runnables to tasks. In other words: Each task always calls the same set of runnables and always in the same order.

Note that the start-time of the task is also implicitly used as the start-time of the first runnable and the termination-time of the task is implicitly used as the stop-time of the last runnable. Also note that there are no longer gaps between the runnables. In figure 9 there *were* gaps and they were even bigger where there were calls to `Schedule()` in the task.

This level of detail is sacrificed for the higher efficiency (in this case 5 events for tracing the runnables rather than 12 events).

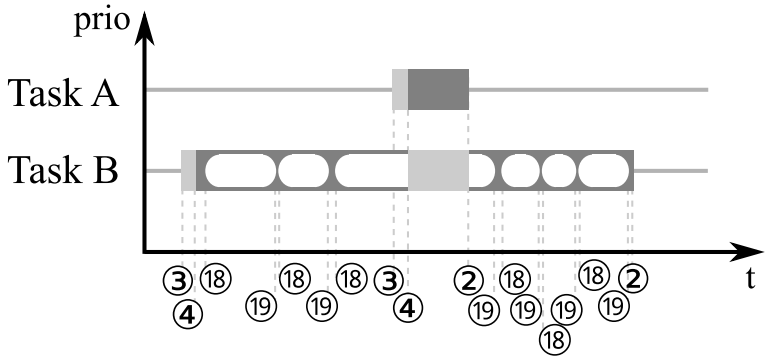

Figure 9: Run-time situation with runnables instrumented (RSTART/RSTOP)

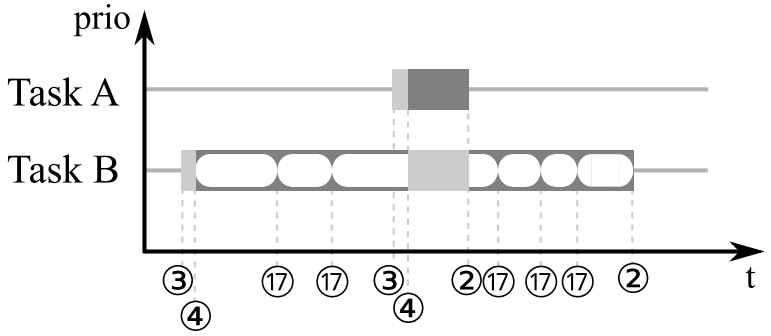

Figure 10: Run-time situation with runnables instrumented (RNEXT)

3 Timing measurement

Having established which events should be logged, we now have to consider when these events should be logged. Even if the ideal cannot be achieved, we should determine the ideal times for instrumentation.

3.1 Task start and stop

We start from the concept of computation time (CET) and three principles:

- CETs are themselves predictable, not depending on remote parts of the configuration
  - for example, CET should not depend on the priority of the task in which the code eventually runs, although GET may well vary greatly with priority
- CETs can be used to predict worst-case GET and RT values with static analysis
- CETs can be used to determine CPU load, or equivalently, CPU load can be partitioned in to CETs

Previous works have sometimes failed to attribute all CPU load to CETs. This unattributable CPU load has been dubbed “OS overhead” without defining exactly

what that means or how it should be handled with regard to timing protection, for example. In contrast, we aim to attribute all CPU load to CETs, that is, every part of the CPU load will be attributed to the task that causes that CPU load. This means that the CET for a task is significantly longer than the time between the start of user code and the entry to TerminateTask. It may well be interesting and useful to measure the time between the start of user code and the entry to TerminateTask but that is not the task CET.

The first principle of CETs is predictability, that if a function F is called in the same way from two different contexts it should have the same execution time. This is required in order that, for example, a library supplier can tell you the timing of the library functions without knowing the context from which they are called. Consider a possible function F:

```
void F( void )
{
   ActivateTask( Task_A );
}
```

Suppose that F is called from a task Task_B with lower priority than Task_A and that resource Resource is shared by both tasks and that no interrupt handlers preempt during this code. We need F to have the same execution time in the following two contexts:

```
/* Context 1: task switch to Task_A happens within F */
F( );

(void)GetResource( Resource );
/* Context 2: no task switch happens within F */
F( );
/* Task switch to Task_A happens within ReleaseResource */
(void)ReleaseResource( Resource );
```

This allows us to define exactly what part of the real time is the execution of F and which is the execution of Task_A. Since we need F to have the same execution time in both contexts, exactly the difference between the GET for F in the two contexts must be attributed to the CET of Task_A.

Interestingly, as we trace a task starting and stopping, we have freedom about when exactly we trace the start and stop events since this observation only constrains the time spanned between the start and the stop events. This freedom allows the instrumentation hooks to be positioned for maximum efficiency.

3.2 Interrupt start and stop

In Section 3.1, we observed that the dispatch of a task or not should leave the underlying code CET unchanged. Similarly, the running of an interrupt should not change the CET of the interrupted code. However, it is impossible to include the entire CET of an interrupt using software measurement since the start instrumentation must occur after the true start of the interrupt and the stop instrumentation must occur before the true resumption of interrupted code. This means that a measured interrupt CET will always be smaller than the actual interrupt CET. The execution time that is not attributed to the interrupt handler will be wrongly attributed to the underlying code. Since almost any code can be interrupted, there is the possibility that the measurement of a very short

and frequently called function could result in a proportionately large overestimation of that function's CET.

Consider the example of a function that always runs for 200ns. If we measure this function without interrupts, we will measure 200ns. However, if the function is interrupted while being measured, and the start event is recorded 500ns after the true start and the stop event is recorded 400ns before the true resumption of the function then a measured CET of 1100ns would result. This will have minimal impact on the average CET but will, of course, set the maximum CET to 1100ns, more than 5 times greater than the real CET. If this frequently called function actually consumes 2% of the CPU load then a simplistic multiplication of the frequency by the maximum CET would appear to show that this function can consume 11% of the CPU load, giving a quite wrong impression.

As a result, it is desirable to record an interrupt start event as close as possible to the actual start of the handler and to record an interrupt stop event as close as possible to the actual end of the handler.

The issue with tasks is not so acute as with interrupts, since task switches result from the use of ActivateTask and ReleaseResource and cannot occur in arbitrary contexts.

3.3 Resource locks

When instrumenting resource locks, the interesting timing information is the maximum time for which a higher priority task can be blocked. To ensure that blocking is not underestimated, we can measure the entire time from the start of the call to `GetResource` to the end of the call to `ReleaseResource`. Note that, since interrupts are typically disabled during OS services, the blocking time includes the execution of `GetResource` and `ReleaseResource`. With the priority ceiling protocol, blocking can occur at most once, so any slight overestimation of the the blocking time has a very minor impact.

However, if the instrumentation is added before the call the GetResource then it is possible that we could trace an interrupt apparently occurring within a resource lock that should inhibit that interrupt. Consider the example in Figure 11.

```
OSTH_LOCK_STOP_SPRVSR( ResourceTraceID,
                    0 /* single core */,
                    OS_TIMER( ) /* internal timestamp */ );
/*
 * Interrupt occurs here, which will be inhibited
 * by locking resource 'Resource'
 */
(void)GetResource( Resource );
...
(void)ReleaseResource( Resource );
OSTH_UNLOCK_SPRVSR( ResourceTraceID,
                  0 /* single core */,
                  OS_TIMER( ) /* internal timestamp */ );
```

Figure 11: Unsuitable instrumentation of resource lock

Note that a naive instrumentation of OS code *inside* the services GetResource and ReleaseResource could lead to an unsafe, underestimation of the blocking time.

Thus we have to instrument the obtaining of a resource as close as possible to the start of GetResource and we have to instrument the releasing of a resource as close as possible to the end of ReleaseResource but strictly before allowing pre-emption (that arises from releasing the resource).

For an ordinary resource, GetResource is typically instrumented just using a lock stop event at the start of the GetResource. In the case of a spinlock, the time required to obtain the lock is variable and may itself be an interesting subject of measurement. To facilitate this, locking can be instrumented using two events, lock start and lock stop:

```
OSTH_LOCK_START_SPRVSR( SpinlockTraceID,
                    0 /* single core */,
                    OS_TIMER( ) /* internal timestamp */ );
GetSpinLock( Spinlock );
OSTH_LOCK_STOP_SPRVSR( SpinlockTraceID,
                   0 /* single core */,
                   OS_TIMER( ) /* internal timestamp */ );
...
ReleaseSpinLock( Spinlock );
OSTH_UNLOCK_SPRVSR( SpinlockTraceID,
                 0 /* single core */,
                 OS_TIMER( ) /* internal timestamp */ );
```

Note that this example is only here to illustrate the use of lock start and lock stop events. It does not, of course, take into account the issues of potentially underestimating the blocking time or tracing the unlock event too late after the actual unlocking.

3.4 Instrumentation gaps

One issue arises when using the simple instrumentation with only START and STOP. Consider the following two examples, in both of which tasks A and B preempt task C.

In figure 12, two hooks are used for the transition A→B resulting in two different timestamps. The gap created represents small portion of time that is, incorrectly, allocated to the low priority task C. In figure 13, a single event STOP_START is traced, saving one hook and creating a single event with a single timestamp. The result is that all time is correctly attributed to tasks A and B and no time is allocated to task C at this task switch.

This optimization improves resource consumption by saving one call, timing accuracy by tracing a single timestamp, and visualization as there is no ambiguity about the pre-empted, low priority task.

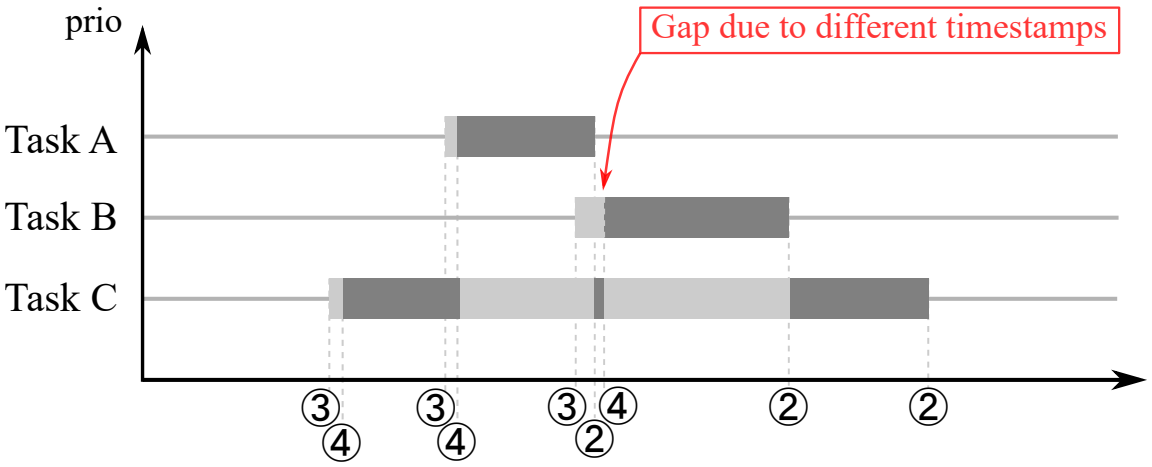

Figure 12: Task chaining instrumented with two events (STOP/START)

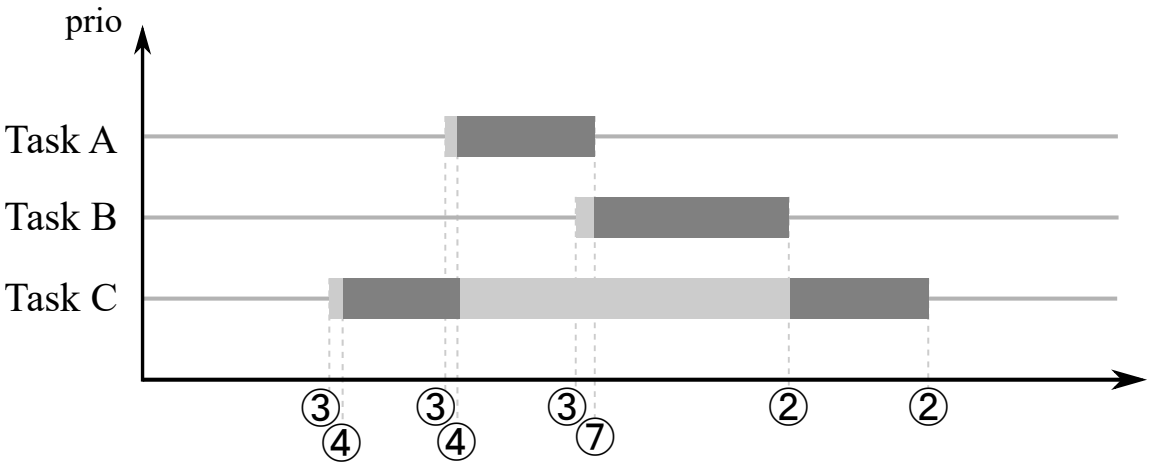

Figure 13: Task chaining instrumented with one event (STOP_START)

4 Appendix

Listing 5: Template header for mapping OS related events on trace functions

```
/*****************************************************************************************
* FILE: ostimhooks.h
*
* DESCRIPTION: Hook macros for use in an OS supporting timing measurement.
*
* $Author: alexandrebau $
*
* $Revision: 39861 $
*
* $URL: https://gliwa.com/svn/repos/1017_OStimHooks/trunk/50_src/ostimhooks.h $
*
* Copyright: GLIWA GmbH embedded systems
* Weilheim i.OB
* All rights reserved
*****************************************************************************************/
```

```
#ifndef OSTIMHOOKS_H_
#define OSTIMHOOKS_H_ (1)

/*
 * For single core applications, the argument coreId_ is ignored.
 *
 * The _NOSUSP variants have a classId_ parameter. This may be ignored
 * but it can be used to split the instrumentation into classes such
 * that the instrumentation in a class cannot be preempted by
 * instrumentation in the same class.
 *
 * schedId_ identifies a schedulable. A schedulable is either a TASK or an ISR.
 *
 * All these hooks are meant to be used inside the OS.
 * Instrumentation of e.g. 'naked' ISRs should be done by other means.
 */



/*
 * Hooks for use in supervisor mode where interrupts
 * can be and may need to be disabled.
 * Inside the OS code this is typically the case, so
 * these hooks are the most common choice.
 */
/* Activation of a task */
#define OSTH_ACTIVATE_SPRVSR( schedId_, coreId_ )
/* Start of a task or ISR */
/* To be used when a new instance of a task or Cat-2 ISR is started (dispatched) */
#define OSTH_START_SPRVSR( schedId_, coreId_ )
/*
 * Optionally to be used when a new instance of a task or Cat-2 ISR is started
 * (dispatched) when it is known that no corresponding activation event was logged.
 */
#define OSTH_PSTART_SPRVSR( schedId_, coreId_ )
/* End of a task or Cat-2 ISR */
/* ChainTask or TerminateTask in AUTOSAR terms when we return to the preempted context */
#define OSTH_STOP_SPRVSR( schedId_, coreId_ )
/* Start and end of a short ISR where only one hook is possible */
#define OSTH_START_STOP_SPRVSR( schedId_, coreId_ )
/*
 * End of one task or Cat-2 ISR and the start of the next without return to a
 * preempted context.
 * ChainTask or TerminateTask in AUTOSAR terms when we do not return to the preempted
 * context. The stopping schedulable is inferred from earlier events.
 */
#define OSTH_STOP_START_SPRVSR( startSchedId_, coreId_ )
/* As above but when no corresponding activation event was logged. */
#define OSTH_STOP_PSTART_SPRVSR( startSchedId_, coreId_ )
/*
 * Release of a waiting task
 * (used within the SetEvent implementation where a waiting task is being released).
 */
#define OSTH_RELEASE_SPRVSR( schedId_, coreId_ )
/* Resumption of a task (return from WaitEvent) */
#define OSTH_RESUME_SPRVSR( schedId_, coreId_ )
/* Suspension of a task (entry to WaitEvent) */
#define OSTH_SUSPEND_SPRVSR( schedId_, coreId_ )
/*
 * The following instrumentation hooks are optional for purely minimal tracing,
 * however, for a full understanding of the events that drive the actual shedule
 * it is highly recommended to make these available.
 */
/*
 * Commence lock of resource or interrupts (to raise the running priority)
 * In AUTOSAR context, a call to LockResource (see priority ceiling protocol),
 * a call to DisableAllInterrupts, SuspendAllInterrupts or SuspendOSInterrupts,
```

```
 * or spinlocks (GetSpinlock always, TryToGetSpinlock when it succeeds).
 * Tracing this event is necessary regardless of an actual change in effective
 * priority.
 */
#define OSTH_LOCK_START_SPRVSR( lockId_, coreId_ )
/* Complete acquisition of a lock, especially important for spinlock. */
/* In AUTOSAR context to be used inside GetSpinlock to measure spinning time. */
#define OSTH_LOCK_STOP_SPRVSR( lockId_, coreId_ )
/* Unlock of resource or interrupts (to lower the running priority) */
#define OSTH_UNLOCK_SPRVSR( lockId_, coreId_ )


/*
 * Hooks for use where instrumented code cannot preempt, therefore instrumented code
 * cannot be preempted by instrumented code. This is always the case when all interrrupts
 * are disabled or in a purely cooperative multitasking environment,
 * there may be other cases in which this condition holds.
 */
/* Activation of a task */
#define OSTH_ACTIVATE_NOSUSP( schedId_, coreId_, classId_ )
/* Start of a task or ISR */
/* To be used when a new instance of a task or Cat-2 ISR is started (dispatched) */
#define OSTH_START_NOSUSP( schedId_, coreId_, classId_ )
/*
 * Optionally to be used when a new instance of a task or Cat-2 ISR is started
 * (dispatched) when it is known that no corresponding activation event was logged.
 */
#define OSTH_PSTART_NOSUSP( schedId_, coreId_, classId_ )
/* End of a task or Cat-2 ISR */
/* ChainTask or TerminateTask in AUTOSAR terms when we return to the preempted context */
#define OSTH_STOP_NOSUSP( schedId_, coreId_, classId_ )
/* Start and end of a short ISR where only one hook is possible */
#define OSTH_START_STOP_NOSUSP( schedId_, coreId_, classId_ )
/*
 * End of one task or Cat-2 ISR and the start of the next without return to a
 * preempted context.
 * ChainTask or TerminateTask in AUTOSAR terms when we do not return to the preempted
 * context. The stopping schedulable is inferred from earlier events.
 */
#define OSTH_STOP_START_NOSUSP( startSchedId_, coreId_, classId_ )
/* As above but when no corresponding activation event was logged. */
#define OSTH_STOP_PSTART_NOSUSP( startSchedId_, coreId_, classId_ )
/*
 * Release of a waiting task
 * (used within the SetEvent implementation where a waiting task is being released).
 */
#define OSTH_RELEASE_NOSUSP( schedId_, coreId_, classId_ )
/* Resumption of a task (return from WaitEvent) */
#define OSTH_RESUME_NOSUSP( schedId_, coreId_, classId_ )
/* Suspension of a task (entry to WaitEvent) */
#define OSTH_SUSPEND_NOSUSP( schedId_, coreId_, classId_ )
/*
 * The following instrumentation hooks are optional for purely minimal tracing,
 * however, for a full understanding of the events that drive the actual shedule
 * it is highly recommended to make these available.
 */
/*
 * Commence lock of resource or interrupts (to raise the running priority)
 * In AUTOSAR context, a call to LockResource (see priority ceiling protocol),
 * a call to DisableAllInterrupts, SuspendAllInterrupts or SuspendOSInterrupts,
 * or spinlocks (GetSpinlock always, TryToGetSpinlock when it succeeds).
 * Tracing this event is necessary regardless of an actual change in effective
 * priority.
 */
#define OSTH_LOCK_START_NOSUSP( lockId_, coreId_, classId_ )
/* Complete acquisition of a lock, especially important for spinlock. */
```

```
/* In AUTOSAR context to be used inside GetSpinlock to measure spinning time. */
#define OSTH_LOCK_STOP_NOSUSP( lockId_, coreId_, classId_ )
/* Unlock of resource or interrupts (to lower the running priority) */
#define OSTH_UNLOCK_NOSUSP( lockId_, coreId_, classId_ )


/*
 * Hooks for use in user mode where interrupts may need to be disabled
 * but cannot be directly disabled. If these hooks are used, the integrator
 * has to provide a means of disabling interrupts from user mode,
 * e.g. by usingCallTrustedFunction in AUTOSAR. Therefore the other
 * hooks should be used in preference.
 */
/* Activation of a task */
#define OSTH_ACTIVATE_USER( schedId_, coreId_ )
/* Start of a task or ISR */
/* To be used when a new instance of a task or Cat-2 ISR is started (dispatched) */
#define OSTH_START_USER( schedId_, coreId_ )
/*
 * Optionally to be used when a new instance of a task or Cat-2 ISR is started
 * (dispatched) when it is known that no corresponding activation event was logged.
 */
#define OSTH_PSTART_USER( schedId_, coreId_ )
/* End of a task or Cat-2 ISR */
/* ChainTask or TerminateTask in AUTOSAR terms when we return to the preempted context */
#define OSTH_STOP_USER( schedId_, coreId_ )
/* Start and end of a short ISR where only one hook is possible */
#define OSTH_START_STOP_USER( schedId_, coreId_ )
/*
 * End of one task or Cat-2 ISR and the start of the next without return to a
 * preempted context.
 * ChainTask or TerminateTask in AUTOSAR terms when we do not return to the preempted
 * context. The stopping schedulable is inferred from earlier events.
 */
#define OSTH_STOP_START_USER( startSchedId_, coreId_ )
/* As above but when no corresponding activation event was logged. */
#define OSTH_STOP_PSTART_USER( startSchedId_, coreId_ )
/*
 * Release of a waiting task
 * (used within the SetEvent implementation where a waiting task is being released).
 */
#define OSTH_RELEASE_USER( schedId_, coreId_ )
/* Resumption of a task (return from WaitEvent) */
#define OSTH_RESUME_USER( schedId_, coreId_ )
/* Suspension of a task (entry to WaitEvent) */
#define OSTH_SUSPEND_USER( schedId_, coreId_ )
/*
 * The following instrumentation hooks are optional for purely minimal tracing,
 * however, for a full understanding of the events that drive the actual shedule
 * it is highly recommended to make these available.
 */
/*
 * Commence lock of resource or interrupts (to raise the running priority)
 * In AUTOSAR context, a call to LockResource (see priority ceiling protocol),
 * a call to DisableAllInterrupts, SuspendAllInterrupts or SuspendOSInterrupts,
 * or spinlocks (GetSpinlock always, TryToGetSpinlock when it succeeds).
 * Tracing this event is necessary regardless of an actual change in effective
 * priority.
 */
#define OSTH_LOCK_START_USER( lockId_, coreId_ )
/* Complete acquisition of a lock, especially important for spinlock. */
/* In AUTOSAR context to be used inside GetSpinlock to measure spinning time. */
#define OSTH_LOCK_STOP_USER( lockId_, coreId_ )
/* Unlock of resource or interrupts (to lower the running priority) */
#define OSTH_UNLOCK_USER( lockId_, coreId_ )
```

```
#endif /* OSTIMHOOKS_H_ */
```

GLIWA
embedded systems

www.gliwa.com

GLIWA GmbH embedded systems
Pollinger Str. 1
82362 Weilheim i.OB.
Germany

fon +49 - 881 - 13 85 22 - 0
fax +49 - 881 - 13 85 22 - 99
mail info@gliwa.com

Geschäftsführer (CEO) Peter Gliwa
Amtsgericht München | HRB 167925
USt-IdNr. DE814169157