OS timing hooks

Generic trace interface

Specification – Version 1.3

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-beta	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 2.3 7. Added figures 2, 3, 4 and 5 8. Add missing RELEASE event 9. Renamed event RESUME to CONTINUE

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

Figure 1 shows the principle timing properties of a task that determine its real-time behaviour within a system and Table 2 defines the symbols used.

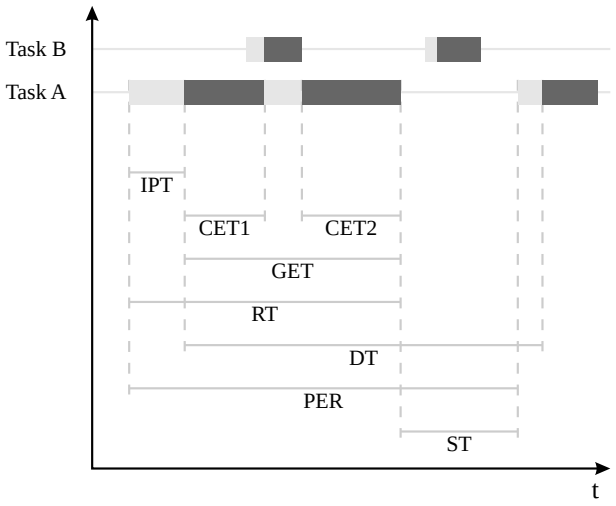

Figure 1: Timing information visualised in a trace (all related to TASK A)

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
3	GET	gross execution time	execution time including all preemptions
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)
9	JIT	jitter	deviation of delta time from period

Table 2: Timing information

We present the hooks themselves in Section 2. The justification, in terms of measurement advantages, is explained in Section 3.

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 Conformance options

Instrumentation based timing measurement 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. The more options are chosen, the more events are logged, the more details are gained and the higher the impact/overhead. Table 3 shows all Conformance options and a brief description for each.

Conformance option ID	*Description*
OCO1	Task activations related macros (logging successful as well as failed activations)
OCO2	Macros for logging combined start/stop and stop/start events for higher precision and more efficiency compared to separate start events and stop 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

Table 3: OStimHooks conformance options (OCO)

Table 4 shows the OS events to be instrumented and the required hooks for measuring the timing properties in Figure 1. To get a complete macro name, the right suffix from table 5 according to the context from which the macro is called needs to be appended. `START_NOSUSP` is one example for a complete macro name, `LOCK_STOP_SPRVSR` another.

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 Task states

Figures 2, 3 and 4 show task state diagrams which correspond to some of the conformance options. The numbers in circles correspond to the column “ID” in table 4.

2.3 Run-time situation example

Figure 5 illustrates a run-time situation with three tasks and two interrupts. It includes the mandatory events START and STOP plus the events of Conformance Option 1 (OCO1), namely successful task activation and task over-activation (E_OS_LIMIT),

No.	*Event description*	*Task state transition*	*First part of hook name*	*Conformance option*
1	*Start* of a task or interrupt	Activated (Ready) → Running or Suspended → Running	START	mandatory
2	*Termination* of a task or *end* of an interrupt	Running → Suspended	STOP	mandatory
3	Successful task *activation*	Suspended → Activated (Ready)	ACTIVATE	OCO1
4	*Failed activation*, i.e. task over-activation (error E_OS_LIMIT as defined in AUTOSAR OS)	none	FAILACT	OCO1
5	Start and end of a short ISR where only one hook is possible	Supended → Running → Suspended	START_STOP	OCO2
6	End of one task and the start of the next without return to a preempted context	Running → Suspended for one task and Ready → Running for the next	STOP_START	OCO2
7	*Continuation* of a terminating task which previously was in the Waiting state	Released (Ready) → Running	CONTINUE	OCO3
8	*Suspension* of a terminating task	Runninng → Waiting	SUSPEND	OCO3
9	*Release* of a terminating task	Waiting → Released (Ready)	RELEASE	OCO3
10	Commence lock of resource or interrupts	none	LOCKING	OCO4
11	Complete lock, especially spinlock	none	LOCKED	OCO4
12	Unlock resource or interrupts	none	UNLOCK	OCO4

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)

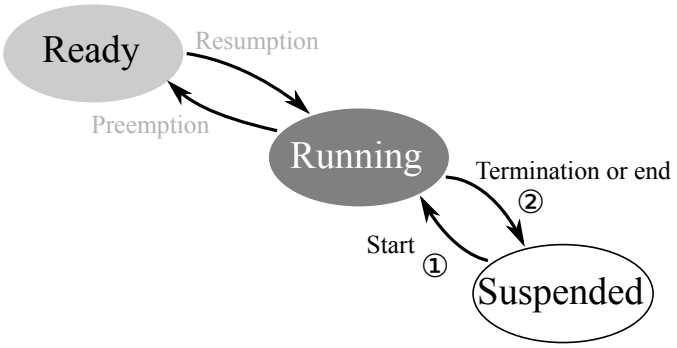

Figure 2: Minimalistic task state scheme with the mandatory start and stop events

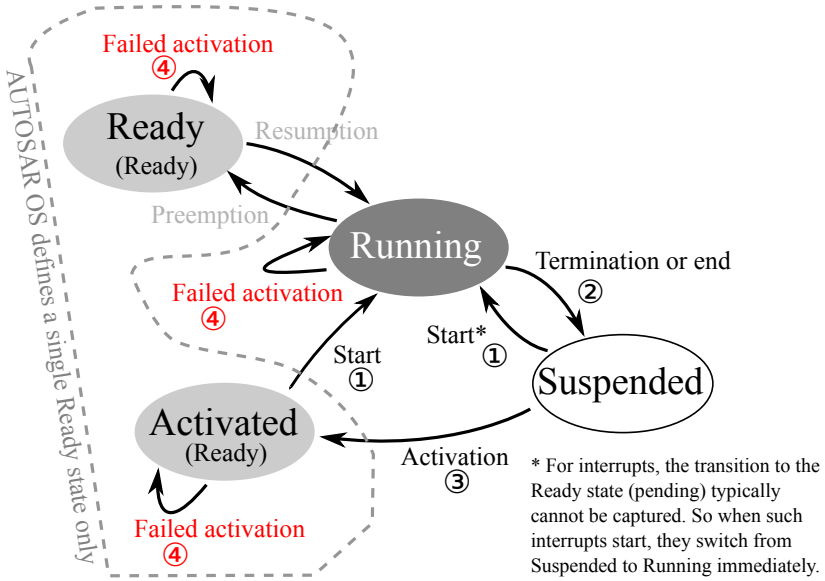

Figure 3: Task state scheme representing an AUTOSAR BCC set-up

ACTIVATE and FAILACT respectively. The numbers in the figure correspond to the numbers in the first column of table 4.

Imagine TASK B being a periodical task. The failed task activation of TASK B could have occurred because of the many preemptions/interruptions of TASK B which could not complete execution before the next activation.

2.4 Comments on AUTOSAR OS ECC

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

However, there *are* set-ups with tasks that do *not* terminate but rather loop, using WaitEvent for scheduling. Such 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

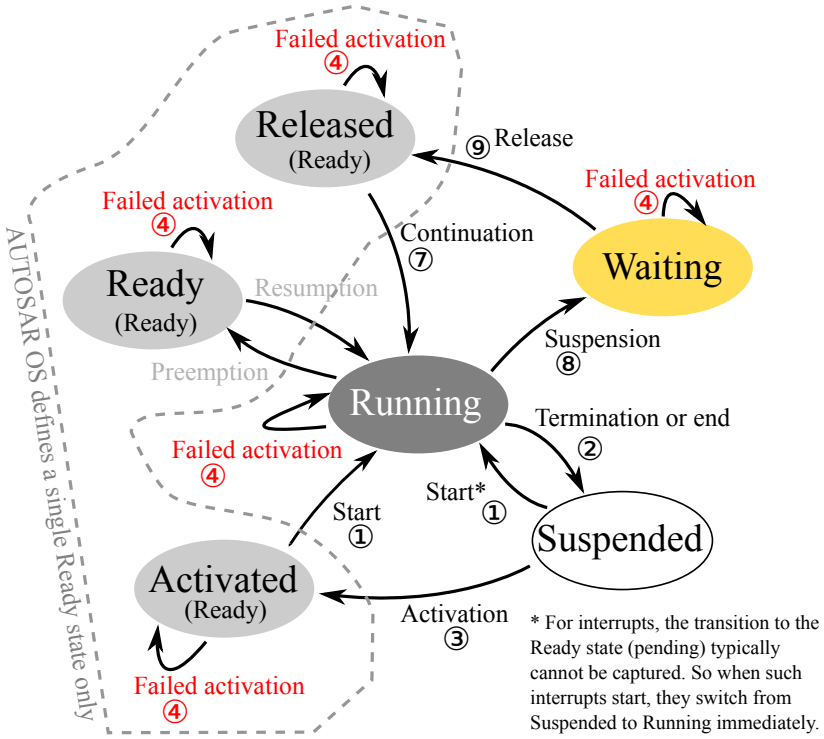

Figure 4: Task state scheme representing an AUTOSAR ECC set-up

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.

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.

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

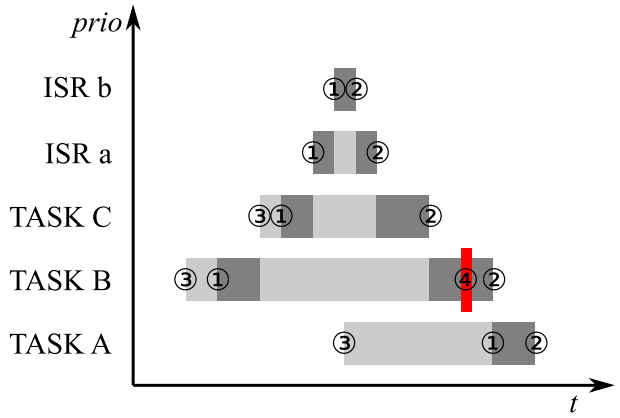

Figure 5: Run-time situation)

- 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 6.

```
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 6: 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.

4 Appendix

Listing 1: 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: christianwen $
*
* $Revision: 25172 $
*
* $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 core_ is ignored.
 * If timestamps are read internally within hook routimes,
 * the argument timeStamp_ is ignored.
 * The _NOSUSP variants have a class 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.
 * 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( taskId_, core_, timeStamp_ )
/* 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( taskId_, core_, timeStamp_ )
/* End of a task or Cat-2 ISR */
/* ChainTask or TerminateTask in AUTOASR terms when we return to the preempted context */
#define OSTH_STOP_SPRVSR( taskId_, core_, timeStamp_ )
/* Start and end of a short ISR where only one hook is possible */
#define OSTH_START_STOP_SPRVSR( taskId_, core_, timeStamp_ )
/*
 * End of one task 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.
 */
#define OSTH_STOP_START_SPRVSR( task1Id_, task2Id_, core_, timeStamp_ )
/*
 * Release of a waiting task
 * (used within the SetEvent implementation where a waiting task is being released).
 */
#define OSTH_RELEASE_SPRVSR( taskId_, core_, timeStamp_ )
/* Resumption of a task (return from WaitEvent) */
#define OSTH_RESUME_SPRVSR( taskId_, core_, timeStamp_ )
/* Suspension of a task (entry to WaitEvent) */
```

```
#define OSTH_SUSPEND_SPRVSR( taskId_, core_, timeStamp_ )
/*
 * 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( resId_, core_, timeStamp_ )
/* 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( resId_, core_, timeStamp_ )
/* Unlock of resource or interrupts (to lower the running priority) */
#define OSTH_UNLOCK_SPRVSR( resId_, core_, timeStamp_ )


/*
 * 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( taskId_, core_, timeStamp_, class_ )
/* 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( taskId_, core_, timeStamp_, class_ )
/* End of a task or Cat-2 ISR */
/* ChainTask or TerminateTask in AUTOASR terms when we return to the preempted context */
#define OSTH_STOP_NOSUSP( taskId_, core_, timeStamp_, class_ )
/* Start and end of a short ISR where only one hook is possible */
#define OSTH_START_STOP_NOSUSP( taskId_, core_, timeStamp_, class_ )
/*
 * End of one task 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.
 */
#define OSTH_STOP_START_NOSUSP( task1Id_, task2Id_, core_, timeStamp_, class_ )
/*
 * Release of a waiting task
 * (used within the SetEvent implementation where a waiting task is being released).
 */
#define OSTH_RELEASE_NOSUSP( taskId_, core_, timeStamp_, class_ )
/* Resumption of a task (return from WaitEvent) */
#define OSTH_RESUME_NOSUSP( taskId_, core_, timeStamp_, class_ )
/* Suspension of a task (entry to WaitEvent) */
#define OSTH_SUSPEND_NOSUSP( taskId_, core_, timeStamp_, class_ )
/*
 * 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( resId_, core_, timeStamp_, class_ )
/* 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( resId_, core_, timeStamp_, class_ )
/* Unlock of resource or interrupts (to lower the running priority) */
#define OSTH_UNLOCK_NOSUSP( resId_, core_, timeStamp_, class_ )


/*
 * 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( taskId_, core_, timeStamp_ )
/* 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( taskId_, core_, timeStamp_ )
/* End of a task or Cat-2 ISR */
/* ChainTask or TerminateTask in AUTOASR terms when we return to the preempted context */
#define OSTH_STOP_USER( taskId_, core_, timeStamp_ )
/* Start and end of a short ISR where only one hook is possible */
#define OSTH_START_STOP_USER( taskId_, core_, timeStamp_ )
/*
 * End of one task 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.
 */
#define OSTH_STOP_START_USER( task1Id_, task2Id_, core_, timeStamp_ )
/*
 * Release of a waiting task
 * (used within the SetEvent implementation where a waiting task is being released).
 */
#define OSTH_RELEASE_USER( taskId_, core_, timeStamp_ )
/* Resumption of a task (return from WaitEvent) */
#define OSTH_RESUME_USER( taskId_, core_, timeStamp_ )
/* Suspension of a task (entry to WaitEvent) */
#define OSTH_SUSPEND_USER( taskId_, core_, timeStamp_ )
/*
 * 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( resId_, core_, timeStamp_ )
/* 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( resId_, core_, timeStamp_ )
/* Unlock of resource or interrupts (to lower the running priority) */
#define OSTH_UNLOCK_USER( resId_, core_, timeStamp_ )

#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