ERIKA Enterprise Minimal API Manual

...multithreading on a thumb!

version: 1.1.3
December 11, 2012

About Evidence S.r.l.

Evidence is a spin-off company of the ReTiS Lab of the Scuola Superiore S. Anna, Pisa, Italy. We are experts in the domain of embedded and real-time systems with a deep knowledge of the design and specification of embedded SW. We keep providing significant advances in the state of the art of real-time analysis and multiprocessor scheduling. Our methodologies and tools aim at bringing innovative solutions for next-generation embedded systems architectures and designs, such as multiprocessor-on-a-chip, reconfigurable hardware, dynamic scheduling and much more!

Contact Info

Address:
Evidence Srl,
Via Carducci 56
Località Ghezzano
56010 S.Giuliano Terme
Pisa - Italy
Tel: +39 050 991 1122, +39 050 991 1224
Fax: +39 050 991 0812, +39 050 991 0855

For more information on Evidence Products, please send an e-mail to the following address: info@evidence.eu.com. Other informations about the Evidence product line can be found at the Evidence web site at: http://www.evidence.eu.com.

Contents

1 Introduction

1.1 Erika Enterprise

Evidence presents the Erika Enterprise RTOS, a minimal RTOS for single chip microcontrollers, which provides a simple and tiny multithreading environment with support for advanced real-time scheduling algorithms and which supports stack sharing.

The Erika Enterprise kernel has been developed with the idea of providing the minimal set of primitives which can be used to implement a multithreading environment. The Erika Enterprise API is available as a reduced OSEK/VDX API, providing support for thread activation, mutual exclusion, alarms, and counting semaphores.

Moreover, the Erika Enterprise kernel offers support for Fixed Priority (FP), Earliest Deadline First (EDF), and Contract-based scheduling (FRSH) [3] scheduling algorithms, to offer a choice between the tradiction and innovative efficient ways of scheduling concurrent threads.

The OSEK/VDX consortium provides the OIL language (OSEK Implementation Language) as a standard configuration language, which is used for the static definition of the RTOS objects which are instantiated and used by the application. Erika Enterprise fully supports the OIL language for the configuration of real-time applications.

To face the complexity of dealing with the OIL language and configuration files, Evidence ships the RT-Druid configuration and profiling environment, which allows to configure all the application parameters through a easy-to-use visual interface that automatically generates the application configuration file using the OIL language.

The typical application design flow include the definition of an OIL configuration file which defines the RTOS objects used by the application; after that, RT-Druid is run for generating appropriate makefiles and source code to configure the Erika Enterprise. Finally, the application is compiled to produce an executable file which can be run on the target.

The features provided by Erika Enterprise to developers are the following:

- Traditional RTOS features:
  - Support for preemptive and non-preemptive multitasking;
  - Support for fixed priority scheduling;
  - Support for shared resources;
  - Support for periodic activations using Alarms;
- Innovative features

- Support for stack sharing techniques, and one-shot task model to reduce the overall stack usage;
- Support for EDF scheduling by using a circular timer approach [1].
- Support for contract-based resource reservations using the IRIS scheduling algorithm [4]

The purpose of this document is to describe in detail the minimal API implemented by Erika Enterprise. Please check the Evidence web site for other documents describing the details of Erika Enterprise portings for the different supported embedded targets.

2 API reference

2.1 Introduction

The Erika Enterprise Operating System provides a basic interface for the execution of concurrent applications on a single processor systems.

The interface proposed is suited for small 8 to 32 bit microcontrollers, and proposes an architecture where tasks can execute concurrently exchanging data with a shared memory paradigm. Support for synchronization primitives is also provided.

Tasks in Erika Enterprise are scheduled according to fixed priorities, and share resources using the Immediate Priority Ceiling protocol (in case of the FP kernel) or the SRP protocol (in case of the EDF kernel). In the case of the FRSH kernel, the scheduling algorithm used is the IRIS scheduler [4], which implements resource reservation with reclaiming based on the IST FP6 FRESCOR project API [3].

On top of task execution there are interrupts, that always preempt the running task to execute urgent operations required by peripherals. Interrupts can be of two kind, names *ISR Type 1* and *ISR Type 2* (see Section 2.7).

2.1.1 Conformance Classes

Erika Enterprise implements the minimal API using three conformance classes:

FP The Fixed priority (FP) conformance class includes a set of functionalities similar to the Erika Enterprise conformance classes BCC2 or ECC2 (depending if the kernel is configured as monostack or multistack).

The FP conformance class basically supports fixed priority multithreading, with more than one task for each priority, with more than one pending activation for each task.

EDF The Earliest Deadline First (EDF) conformance class includes the support for an EDF scheduler. Each task has a relative deadline which is computed when the task activation is processed (which is at the time of the previous instance if the task has pending activations). The deadline is coded using the circular timer implementation (see Section 2.10).

FRSH The FRescor SCheduler (FRSH) conformance class includes the support for an EDF scheduler and, on top of it, an implementation of the IRIS scheduler [4] together with the implementation of binding/unbinding specific functions.

Service	Background Task	Task	ISR1	ISR2	Alarm Callback
ActivateTask	√	√		√	
Schedule		√			
GetResource		√		√	
ReleaseResource		√		√	
CounterTick		√		√	
GetAlarm	√	√		√	
SetRelAlarm	√	√		√	
SetAbsAlarm	√	√		√	
CancelAlarm	√	√		√	
InitSem	√	√		√	
WaitSem		√			
TryWaitSem	√	√		√	
PostSem	√	√		√	
GetValueSem	√	√		√	
GetTime	√	√		√	

Table 2.1: This table lists the environments where primitives can be called.

2.1.2 Available primitives

Erika Enterpriseprovides a set of primitives that can be called in different situations. The complete list of primitives is listed in Table 2.1, together with the locations where it is legal to call these functions.

In addition, Table 2.2 shows the same information for the aditional primitives introduced with the `FRSH` Kernel.

The FRSH implementation is different from FP and EDF in many different aspects. For this reason, most of the details about the `FRSH` implementation will be described separately in Chapter 3. To avoid repetitions of informations, all the primitives which have a similar behavior (for example, `ActivateTask`) will be reported only once below.

Service	Background Task	Task	ISR1	ISR2	Alarm Callback
`frsh_init`	√				
`frsh_strerror`	√	√		√	√
`frsh_contract_get_basic_params`	√	√		√	√
`frsh_contract_get_timing_reqs`	√	√		√	√
`frsh_thread_bind`		√			
`frsh_thread_unbind`		√			
`frsh_vres_get_vres_id`		√		√	
`frsh_vres_get_contract`	√	√		√	√
`frsh_synchobj_signal`		√			
`frsh_synchobj_wait`		√			
`frsh_synchobj_wait_with_timeout`		√			
`frsh_timed_wait`		√			
`frsh_config_is_admission_test_enabled`	√	√		√	√
`frsh_vres_get_remaining_budget`	√	√		√	√
`frsh_vres_get_usage`	√	√		√	√
`frsh_vres_get_budget_and_period`	√	√		√	√

Table 2.2: This table lists the environments where FRSH-specific primitives can be called.

2.2 Constants

This is a list of the Erika Enterprise constants that can be used by the developer for writing applications.

2.2.1 INVALID_TASK

Description

This constant represent an invalid task ID.

Conformance

FP, EDF, FRSH

2.2.2 EE_MAX_NACT

Description

This constant represent the maximum number of pending activations which can be stored for a given task. Its typical value is the maximum value for an unsigned integer on the particular architecture.

Conformance

FP, EDF

2.2.3 RES_SCHEDULER

Description

This is the ID of the `RES_SCHEDULER` resource.

That resource exists only when `USE_RESSCHEDULER` is set to `TRUE` inside the OIL configuration file. The `RES_SCHEDULER` ceiling depends on the tasks that exists in the system, and it is computed when RT-Druid generates the Erika Enterprise configuration code.

Conformance

FP, EDF

2.2.4 Task States

Description

This is the list of the task states a task can have during its life:

```
#define EE_READY      1
#define EE_STACKED    2
#define EE_BLOCKED    4
#define EE_WASSTACKED 8
```

Task States in Erika Enterprise are typically not visible to the application, because they are highly dependent on the particular Erika Enterprise configuration. In particular, when using a monostack configuration, task statuses are removed from the system to save RAM. The `EE_READY` status is used when a task is ready to execute but it has not been allocated in its stack yet. The `EE_STACKED` status refers to a task which is either the running task or it has been preempted on the stack. The status `EE_BLOCKED` considers a task which has executed and which is currently blocked on a synchronization primitive (e.g., a `WaitSem` primitive). An additional flag named `EE_WASSTACKED` is also defined for internal reasons to map a ready task which has been woken up from a synchronization but which is still in the ready queue waiting to execute.

Conformance

FP, EDF

2.3 Types

This Section contains a description of the data types used by the OS interface of Erika Enterprise. When the size of a given type is indicated to be of the size of a machine register, it is intended that such type has the same size of the CPU general purpose register.

2.3.1 AlarmType

Description

This (signed) type is used to store Alarm IDs, and it has the size of a register.

Conformance

FP, EDF, FRSH

2.3.2 CounterType

Description

This (signed) type is used to store Counter IDs, and it has the size of a register.

Conformance

FP, EDF, FRSH

2.3.3 ResourceType

Description

This (unsigned) type is used to store Resource ID values, and it has the size of a register.

Conformance

FP, EDF, FRSH

2.3.4 SemType

Description

This type is a structure storing the information related to a counting semaphore.

Conformance

FP, EDF

2.3.5 SemRefType

Description

This is a pointer to `SemType`.

Conformance

FP, EDF

2.3.6 TaskType

Description

This (signed) type is used to store Task ID, and it has the size of a register.

Conformance

FP, EDF, FRSH

2.3.7 TickType

Description

This (unsigned) type is used to store Counter Ticks, and it has the size of a register.

Conformance

FP, EDF, FRSH

2.3.8 TickRefType

Description

This is a pointer to `TickType`.

Conformance

FP, EDF, FRSH

2.3.9 TimeAbsType

Description

This is an absolute timer reference, coded using the circular timer method (see Section 2.10.

Conformance

EDF, FRSH

2.3.10 TimeRelType

Description

This is a relative timer reference, coded using the circular timer method (see Section 2.10.

Conformance

EDF, FRSH

2.4 Object Definitions

The following macro have to be used when defining a Task.

2.4.1 TASK

Synopsis

```
TASK(Funcname) {...}
```

Description

The TASK keyword must be used when declaring a TASK function.

Conformance

FP, EDF, FRSH

2.5 Task Primitives

Erika Enterprise minimal API supports the definition of tasks which are similar to the Basic Tasks of the OSEK/VDX Standard.

Erika Enterprise Tasks are typically implemented as normal C functions, that executes their code and then ends. One of these executions is called also *Task Instance*. After the end of a task, its stack is freed. Erika Enterprise tasks typically never block, allowing the developer to implement stack sharing between different tasks. Sharing the stack helps the developer to reduce the overall RAM used for the stack.

Support for blocking primitives like counting semaphores or FRSH synchronization objects is also available. In these cases, the kernel has to be configured as multistack, and the tasks using these primitives will need a private stack assigned to them (note that the multistack configuration is mandatory for the FRSH kernel implementation). Tasks using blocking primitives are typically implemented as a never ending task in which each instance ends with a synchronization implemented for example as a semaphore wait.

In the conformance class FP, the scheduling policy is a Fixed Priority Scheduling with Immediate Priority Ceiling and Preemption Thresholds. As a result, the following case of tasks may be implemented:

Full Preemptive Task A Full Preemptive task is a task that can be preempted in each instant by higher priority tasks.

Non Preemptive Task A Non Preemptive task is like a Full Preemptive task that executes all the time locking a resource with its ceiling equal to the maximum priority in the system. As a result, a non preemptive task cannot be preempted by other tasks: only interrupts can preempt it.

Mixed Preemptive Task A Mixed Preemptive task is a task which executes at a higher priority than the priority used to queue it in the ready queue (This technique is called *Preemption Thresholds*). As a result, preemption between tasks is reduced allowing consistent savings in the RAM space used for stacks.

In the conformance class EDF, the scheduling policy is an Earliest Deadline First implementation with Stack Resource Policy (SRP), and Preemption Thresholds. Task parameters include the specification of a relative deadline (specified in the `RELDLINE` OIL attribute) as well as a preemption level (specified in the `PRIORITY` atrribute. As a result, the following case of tasks may be implemented:

Full Preemptive Task A Full Preemptive task is a task that can be preempted in each instant by higher priority tasks.

Non Preemptive Task A Non Preemptive task is like a Full Preemptive task that executes all the time locking a resource with its ceiling equal to the maximum preemption level in the system. As a result, a non preemptive task cannot be preempted by other tasks: only interrupts can preempt it.

Mixed Preemptive Task A Mixed Preemptive task is a task which executes at a higher priority than the priority used to queue it in the ready queue (This technique is called *Preemption Thresholds*). As a result, preemption between tasks is reduced allowing consistent savings in the RAM space used for stacks.

In the conformance class FRSH, the scheduling policy is an Earliest Deadline First implementation with Stack Resource Policy (SRP), and with deadlines implemented using the IRIS scheduling algorithm [4]. In this case, each task is assigned a “Virtual Resource”, which is used as a means to control the task execution time. The VRES parameters which can be specified are a “budget”, which is an amount of time that the task can spend every “period”. The scheduling algorithm basically guarantees that at least the budget time will be guaranteed to the task every period, providing a so called “resource reservation”.

Tasks are activated using the primitive `ActivateTask`. Activating a task means that the activated task may be selected for scheduling and may execute one Task Instance. A task activation while a task is already waiting its execution or while being the running task is saved as a pending activation (up to a maximum number which is implementation defined). Note that EDF deadlines for a pending activation are computed when the previous instance ends.

Tasks scheduled with the minimal API are slightly different if compared with tasks scheduled with the OSEK conformance classes. These are the main differences:

- The minimal API does not support the primitives `TerminateTask` or `ChainTask`.
- The number of pending activations does not need to be specified inside the OIL file (as it happens in the BCC2 and ECC2 conformance classes of Erika Enterprise).

2.5.1 ActivateTask

Synopsis

```
void ActivateTask(TaskType TaskID);
```

Description

This primitive activates a task `TaskID`. Upon activation, the task may become the running task if it is the highest priority ready task (if using the FP kernel) or if it is the task with the earliest deadline and with preemption level greater than the system ceiling (using the EDF and FRSH kernel)..

Once activated, the task will run for an instance, starting from its first instruction. If the task is activated while being the running task, or being ready to execute, the activation is stored as a pending activation, which will be handled afterwards. There is a maximum number of pending activations. If the maximum number of pending activations is exceeded, the activation is ignored. Note that for the EDF kernel, the deadline for a pending activation is computed when the previous instance ends.

The function can be called from the Background task (typically, the `main()` function).

Parameters

- `TaskID` Task reference.

Return Values

- `void` The function never returns an error.

Conformance

FP, EDF, FRSH

2.5.2 Schedule

Synopsis

```
void Schedule(void)
```

Description

This primitive can be used as a rescheduling point for tasks that uses Preemption Thresholds and for non preemptive tasks.

When this primitive is called, a task using preemption thresholds sets its priority to the (lower) one used when queuing on the ready queue. Then, the system checks if there are higher priority tasks that have to preempt (in that case, a preemption is implemented). When the primitive returns, tasks using preemption thresholds will reacquire its threshold priority.

The primitive has no effect if the calling task is neither a non-preemptive task, neither a task using preemption thresholds.

Return Values

- `void` The function never returns an error.

Conformance

FP, EDF, FRSH

2.6 Resource primitives

Resources refer to binary semaphores used to implement shared critical sections.

Resources are implemented using the Immediate Priority Ceiling protocol (FP kernel), or using the Stack Resource Policy (EDF and FRSH kernel). A resource is locked using the primitive `GetResource`, and unlocked using `ReleaseResource`.

A special resource named `RES_SCHEDULER` is also supported for kernels FP and EDF. The `RES_SCHEDULER` resource has a ceiling equal to the highest priority (FP or highest preemption level in the case of EDF) in the system. As a result, a task locking `RES_SCHEDULER` becomes non-preemptive. If needed, the `RES_SCHEDULER` resource have to be configured in the OIL configuration file.

In the FRSH Kernel implementation, preemption due to VRES budget exaustion during critical sections is disabled. This approach has also been analyzed in the literature as "overrun and payback" mechanism by **??** and **??** w.r.t. a task continuing to execute when its budget runs out but it is holding a resource. In that case, the task budget becomes negative and the time consumed in excess will be borrowed from the next task instances (in pathological cases a task may be forced not to schedule for a few instances due to that, as shown in [2]). The rationale behind this choice is that it is too costly on a microcontroller to implement some control over the time a task owns a resource. Being the application environment somehow controlled, there is hope that the designer will not abuse of the timing spent in a critical resource. Please note that also binding and unbinding is disabled when a task owns a resource

2.6.1 GetResource

Synopsis

```
void GetResource (ResourceType ResID)
```

Description

This primitive can be used to implement a critical section guarded by Resource `ResID`. The critical section will end with the call to `ReleaseResource`.

Nesting between critical sections guarded by different resources is allowed.

Calls to `Schedule` are not allowed inside the critical section.

The service may be called from task level only.

Parameters

- `ResID` Reference to resource

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF, FRSH

2.6.2 ReleaseResource

Synopsis

```
void ReleaseResource (ResourceType ResID)
```

Description

`ReleaseResource` is used to release a resource locked using `GetResource`, closing a critical section.

For information on nested critical sections, see `GetResource`.

The service may be called from task level only.

Parameters

- `ResID` Resource identifier

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF, FRSH

2.7 Interrupt primitives

The minimal API gives support for interrupts. Interrupts are modeled considering typical microcontroller designs featuring interrupt controllers with a prioritized view of the interrupt sources.

To map the requirements of fast OS-independent requests, Erika Enterprise supports the definition of fast interrupts handlers, called *ISR Type 1*, that on one side can handle interrupts in the fastest possible way, but on the other side lack the possibility to call OS services.

On the other hand, lower priority interrupts, called *ISR Type 2* and used (for example) for hardware timers, can call selected OS primitives but are slower than ISR Type 1 due to the OS bookkeeping needed to implement preemption.

Most of implementation details related to IRQ handling highly depends on the particular microcontroller on which Erika Enterprise is used. Please refer to the documents related to the porting of Erika Enterprise to the specific architecture for further details.

2.8 Counter and Alarms primitives

Erika Enterprise supports a notification mechanism based on *Counter*s and *Alarm*s.

A Counter is basically an integer value that can be incremented by 1 "Tick" using the primitive `CounterTick`.

An Alarm is a notification that is attached to a specific Counter. The link between a Counter and an Alarm is specified at compile time in the OIL Configuration file.

An Alarm can be set to fire at a specified tick value using the primitives `SetRelAlarm` and `SetAbsAlarm`. Alarms can be set to be cyclically reactivated. Alarms can be canceled using the primitive `CancelAlarm`.

When an Alarm fires, a notification takes place. A notification is set to be one of the following actions:

Task activation. In this case, a task is activated when the Alarm fires.

Alarm callback. In this case, an alarm callback (defined as `void f(void)`) is called.

The notifications are executed inside the `CounterTick` function. It is up to the developer placing the counter in meaningful places (e.g., a timer interrupt).

Counters, Alarms, and their notifications are specified inside the OIL configuration file.

Warning: Currently there is no support for automatically generated system counters (e.g., counters that are automatically linked to hardware timers). All the counters have to be defined within the OIL Configuration file, and the programmer have to call `CounterTick` to increment them.

2.8.1 CounterTick

Synopsis

```
void CounterTick(CounterType c)
```

Description

This function receives a counter identifier as parameter, and it increments it by 1. This function is typically called inside an ISR type 2 or inside a task to notify that the event monitored by a counter has happened.

The function also implements the notification of expired alarms, that is implemented, depending on the alarm configuration, as:

- an alarm callback function;
- a task activation.

The function is atomic, and no rescheduling will take place after the execution of this function. When called from the task level, to implement the rescheduling the application should call `Schedule` after the call to this function. When called from the ISR type 2 level, the rescheduling will automatically take place at the end of the interrupt routines.

Parameters

- `c` The counter that needs to be incremented.

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF, FRSH

2.8.2 GetAlarm

Synopsis

```
void GetAlarm (AlarmType AlarmID, TickRefType Tick)
```

Description

The system service GetAlarm returns the relative value in ticks before the alarm `AlarmID` expires. `AlarmID` *must* be in use. Allowed on task level, ISR, and in several hook routines.

Parameters

- `AlarmID` Alarm identifier
- `Tick` (out) Relative value in ticks before the alarm expires

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF, FRSH

2.8.3 SetRelAlarm

Synopsis

```
void SetRelAlarm (AlarmType AlarmID, TickType increment, TickType cycle)
```

Description

After `increment` ticks have elapsed, the `AlarmID` notification is executed.

If the relative value `increment` is very small, the alarm may expire, and the notification can be executed before the system service returns to the user. If `cycle` is unequal zero, the alarm element is logged on again immediately after expiry with the relative value `cycle`.

The alarm `AlarmID` must not already be in use: before changing the value of an alarm already in use, the alarm must be canceled. Allowed on task level and in ISR.

Parameters

- `AlarmID` Reference to alarm
- `increment` Relative value in ticks representing the offset with respect to the current time of the first alarm expiration.
- `cycle` Cycle value in case of cyclic alarm. In case of single alarms, this parameter must be set to 0.

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF, FRSH

2.8.4 SetAbsAlarm

Synopsis

```
void SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)
```

Description

The primitive occupies the alarm `AlarmID` element. When `start` ticks are reached, the `AlarmID` notification is executed.

If the absolute value `start` is very close to the current counter value, the alarm may expire, and the task may become ready or the alarm-callback may be called before the system service returns to the user.

If the absolute value `start` was already reached before the system call, the alarm shall only expire when the absolute value `start` is reached again, i.e. after the next overrun of the counter.

If `cycle` is unequal zero, the alarm element is logged on again immediately after expiry with the relative value `cycle`.

The alarm `AlarmID` shall not already be in use: before changing the value of an alarm already in use, the alarm must be canceled. Allowed on task level and in ISR.

Parameters

- `AlarmID` reference to alarm.
- `start` Absolute value in ticks representing the time of the first expiration of the alarm.
- `cycle` cycle value in case of cyclic alarm. In case of single alarms, this parameter must be set to 0.

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF, FRSH

2.8.5 CancelAlarm

Synopsis

```
void CancelAlarm (AlarmType AlarmID)
```

Description

The primitive cancels the alarm `AlarmID`. Allowed on task level and in ISR.

Parameters

- `AlarmID` reference to alarm

Return Values

- `void` the function does not return an error.

Conformance

FP, EDF, FRSH

2.9 Counting Semaphores

This section describes in detail the primitives provided by the minimal API of Erika Enterprise to support counting semaphores as a way to implement mutual exclusion and synchronization between tasks.

A counting semaphore is a RTOS abstraction of an integer counter coupled with a blocking queue. Basically two main operations are possible on a semaphore, which are *waiting* on a semaphore, which results in decrementing the counter if the counter has a value greater than 0, or blocking the running task if the counter is 0, and *posting* on a semaphore, which results in a counter increment if there are no task blocked, or in the unblock of a blocked task otherwise.

Erika Enterprise counting semaphores exports a simple interface which covers the basic functionalities of a semaphore, like:

- Initializing a semaphore (`InitSem`);
- Waiting on a semaphore in a blocking (`WaitSem`) or non-blocking way (`TryWaitSem`;
- Posting on a semaphore (`PostSem`);
- Getting the value of a semaphore (`GetValueSem`).

Since waiting on a semaphore may result in blocking the running task, the `WaitSem` primitive should be called only if the calling task has a separate stack allocated to it (which means that Erika Enterprise has been configured as multistack).

Semaphores can also be allocated statically as a global variable, which allow to bypass the call to `InitSem`.

Semaphores definition are not listed in the OIL file; semaphore primitives receive as a parameter a pointer to the semaphore descriptor.

Warning: Counting semaphores *do not* avoid Priority Inversion problems. Please use Resources instead (see Section 2.6).

Warning: Counting semaphores are not implementef in the FRSH kernel.

2.9.1 STATICSEM

Synopsis

```
SemType s = STATICSEM(value);
```

Description

This macro can be used to statically initialize a semaphore. It must be used inside the definition of a global semaphore variable to initialize a semaphore to a given value.

Parameters

- `value` The counter value for the semaphore being initialized.

Return Values

- `none` The function is a macro used at variable definition time.

Conformance

FP, EDF

2.9.2 InitSem

Synopsis

```
void InitSem(SemType s, int value);
```

Description

This macro can be used to initialize a semaphore at runtime. It receives as a parameter the init value of the semaphore counter.

Parameters

- `s` The semaphore being initialized.
- `value` The counter value for the semaphore being initialized.

Return Values

- `void` The function is a macro and it does not return an error.

Conformance

FP, EDF

2.9.3 WaitSem

Synopsis

```
void WaitSem(SemRefType s);
```

Description

If the semaphore counter is greater than 0, then the counter is decremented by one. If the counter has a value of 0, then the calling (running) task blocks. A separate stack must be allocated to all the tasks which will call this primitive, because its execution may block the task.

Parameters

- `s` The semaphore used by the primitive.

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF

2.9.4 TryWaitSem

Synopsis

```
int TryWaitSem(SemRefType s);
```

Description

This is a non-blocking version of `SemWait`. If the semaphore counter is greater than 0, then the counter is decremented by one, and the primitive returns 0. If the counter has a value of 0, then the counter is not decremented, and the primitive returns 1.

Parameters

- `s` The semaphore used by the primitive.

Return Values

- `int` 0 if the semaphore counter has been decremented, 1 otherwise.

Conformance

FP, EDF

2.9.5 PostSem

Synopsis

```
void PostSem(SemRefType s);
```

Description

This primitive unblocks a task eventually blocked on the semaphore. If there are no tasks blocked on the semaphore, then the semaphore counter is incremented by one.

Parameters

- `s` The semaphore used by the primitive.

Return Values

- `void` The function does not return an error.

Conformance

FP, EDF

2.9.6 GetValueSem

Synopsis

```
int GetValueSem(SemRefType s);
```

Description

If there are tasks blocked on the semaphore, the function returns `-1`; otherwise, this primitive returns the value of the semaphore counter.

Parameters

- `s` The semaphore used by the primitive.

Return Values

- `int -1` if there are tasks blocked on the semaphore, or the semaphore counter value otherwise.

Conformance

FP, EDF

2.10 Time handling

The implementation of the EDF and FRSH scheduler done in the minimal API is based on a timing reference which is made to be efficiently implemented in small microcontrollers.

The traditional way of implementing a timing representation which can be used to compute and store timing references used as example for deadlines is based on the POSIX `struct timespec` data structure. Unfortunately, the `struct timespec` data structure is not suited to be implemented on small mirocontrollers. The structure in fact is composed by two 32-bit integer representing seconds and nanoseconds, which require a substantial code amount to implement the most common operations.

For that reason, the EDF and FRSH implementation proposed by Erika Enterprise uses a relative notion of time. That is, the system proposes a timing reference which has the same size of a hardware timer register. All the timings are then considered relative to the current timing, and the timings are ordered by using the sign of their difference.

The timing reference is often implemented using a hardware timer or using a software incremented timer (e.g., like a software counter incremented by a periodic interrupt).

Using this method it is possible to represent a set of deadlines which has a maximum distance of half the wraparound time of the hardware or software timer linked to them (see Figure 2.1).

The approximation in general is quite good, because it allows to handle the common cases of periodic tasks with deadline spanning from a few milliseconds to hundreds of microseconds. with a relatively good precision.

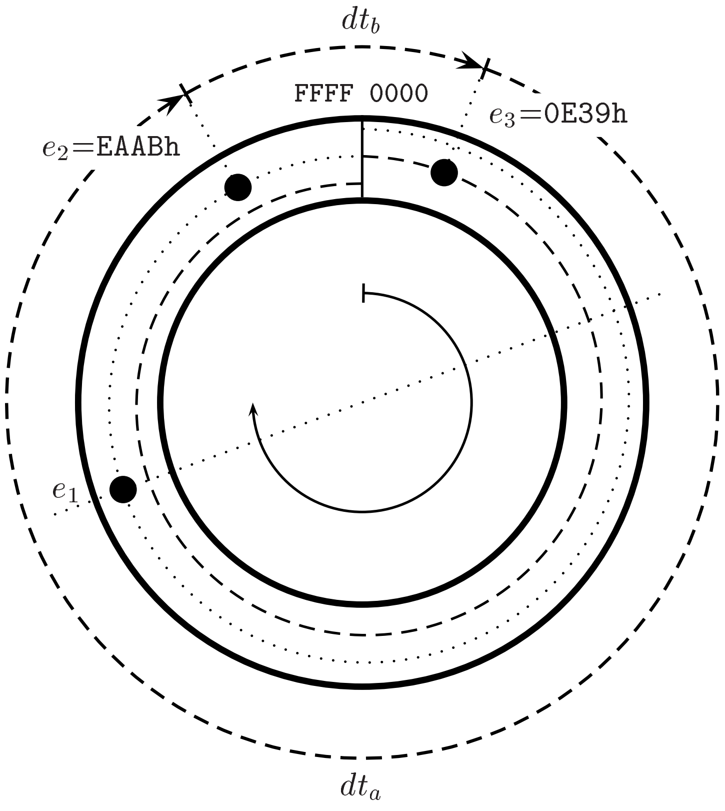

Figure 2.1: The relative timer representation. In the figure, e_2 comes before e_3

2.10.1 GetTime

Synopsis

`TimeAbsType GetTime(void);`

Description

This function is used to return the current system time. This function is typically called inside a task, inside the main task or inside a ISR type 2.

Return Values

- `TimeAbsType` The current timer value.

Conformance

EDF, FRSH

2.11 System Startup

When using the FP or EDF kernel, there is no need a specific startup procedure. In particular, the kernel is already active after the first instruction of the `main` function. When using the FRSH kernel, the function `frsh_init` must be used to initialize the resource reservation and deadline checking features.

A typical application will be structured with application dependent initialization routines inside the `main` function. Then, tasks will be activated with calls to `ActivateTask`, and finally the `main` task will end with a forever loop, implementing in this way the background task.

3 FRSH Kernel details

This brief Section is dedicated to the implementation of the FRESCOR IST FP6 Project [3] inside Erika Enterprise. The Section start with an introduction of the FRESCOR Project, continuing with the description of the implemented FRSH API.

3.1 Introduction to FRESCOR

The FRESCOR project [3] is aimed at developing a framework that integrates advanced flexible scheduling techniques directly into an embedded systems design methodology, covering all the levels involved in the implementation, from the OS primitives, through the middleware, up to the application level. This will be centred on a new contract model that will specify which are the application requirements with respect to the flexible use of the schedulable resources in the system, and also what are the resources that must be guaranteed if an application component is to be installed into the system, and how the system can distribute any spare capacity that it has, to achieve the highest usage of the available resources or to optimise their usage while addressing energy constraints. The project will build on the results of two previous EU projects: FIRST (Flexible Integrated Real-Time Systems Technology) and OCERA (Open Components for Embedded Real-time Applications).

The main objective of the project is to develop the enabling technology and infrastructure required to effectively use the most advanced techniques developed for real-time flexible scheduling in embedded systems design methodologies and tools, providing the necessary elements to target reconfigurable processing modules and reconfigurable distributed architectures.

In the area of development methods for embedded applications with complex timing requirements we have identified a large gap between the research state of the art and the applicability. The project will be addressed at closing this gap, and providing embedded systems developers the engineering solutions to manage timing requirements at a high level of abstraction, thus lowering the design and development costs, and speeding up the time to market. The project will address the OS primitives needed to support the contract-based scheduling, the development of middleware to support the contracts themselves and adaptively manage the quality of service, the integration of this infrastructure into a container framework for component-based development methodologies, the development of simulation and analysis tools, and the evaluation and exploitation of the results.

FRESCOR produced a framework capable of giving a system's view of the timing requirements and the schedulable resources. The framework enables the developer of an application component to focus on its own timing requirements without having to focus on system-level issues, which is essential in the dynamic scenario of reconfigurable architectures. In addition, the framework will provide for a transparent management of the quality of service at the system's level, by providing an end-to-end distributed transaction manager, adaptive system-level quality of service management middleware, and system-level design and optimization methods. The technologies developed in the project will increase competitiveness of EU industry by providing the proper abstractions that make it possible to handle complex timing requirements in embedded applications, explicitly supporting the integrated scheduling of reconfigurable resources, including processors, reconfigurable hardware modules, memory, energy, and networks.

The implementation presented in this Section is the result of the effort of Evidence in porting the FRESCOR scheduling API (named FRSH, pronounced as "fresh") to Erika Enterprise. The FRSH kernel basically adapt the FRSH API as defined in the FRESCOR Project to a minimal environment such as the one available on a microcontroller. The result is a kernel configuration without support for dynamic reconfiguration with a ROM footprint of around 10k for the complete system (more details on [2]).

Please note that the API implemented in the FRSH kernel is the merge of two different APIs, one similar to the OSEK/VDX API, and one as a subset of the FRSH API considering only the parts related to binding, unbinding, and synchronization. This merge is still visible in the primitive names, since the FRSH API derivative functions still have the `frsh_` prefix[1].

It is out of the scope of this document to explain in detail all the caveats of the FRSH API. Instead, we will concentarte on the functions implemented into Erika Enterprise. The interested reader can find more information on the following references: [3], [2], [4].

3.2 What is currently implemented of the FRSH API

The implementation we presented on Erika Enterprise is not a complete implementation of the FRSH API. In particular, the implementation is a tradeoff between functionality and performance, made in a way to obtain a reasonable implementation on small microcontrollers.

The features which have been implemented are the following:

- Task activation, resources and alarms with API similar to the `FP` and `EDF` kernel;
- Task binding and unbinding following the FRSH API specification;
- Synchronization objects following the FRSH API specification.
- IRIS [4] scheduling algorithm for resource reservations.
- Support for the configuration using OIL and RT-Druid.
- FRSH types implemented in a similar ways to the FRSH API.

[1] the naming of these functions will be probably uniformed in the near future.

For more information on the FRSH extensions on the OIL language implemented by RT-Druid, you can refer to the RT-Druid reference manual.

3.3 FRSH API - Task and VRES Statuses

The following paragraphs describe Task and VRES statuses. Please note that the status of a task is orthogonal to the status of a VRES. In particular, the Task status controls the possibility for a task to execute given its activations and suspensions on synchronizations. The VRES status control the possibility for a task to be scheduled, which is controlled by the budget and period assigned to the VRES in the CONTRACT section of the OIL file.

3.3.1 Task States

Description

This is the list of the task states a task can have during its life:

```
#define EE_TASK_SUSPENDED  0
#define EE_TASK_READY      1
#define EE_TASK_STACKED    2
#define EE_TASK_BLOCKED    4
#define EE_TASK_EXEC       8
#define EE_TASK_WASSTACKED 128
```

Task States in Erika Enterprise are typically not visible to the application, because they are highly dependent on the particular Erika Enterprise kernel implementation. In the `FRSH` kernel, the states have the following meaning: `EE_TASK_SUSPENDED` is used when a task is not requiring execution. `EE_TASK_READY` when a task is ready to execute and it is waiting that its VRES is active with the earliest deadline in the system to pass to the `EE_TASK_EXEC` status. `EE_TASK_STACKED` is used to track preempted tasks which are owning a Resource. `EE_TASK_BLOCKED` are used to track tasks which are blocked on synchronization objects. `EE_TASK_WASSTACKED` is used to track the fact that a task has been preempted or has been blocked still retaining its status on the stack.

Conformance

FRSH

3.3.2 VRES States

Description

This is the list of the VRES states a task can have during its life:

```
#define EE_VRES_FREEZED    0
#define EE_VRES_INACTIVE   1
```

```
#define EE_VRES_ACTIVE     2
#define EE_VRES_RECHARGING 4
```

VRES states are not visible to the application, although they control the possibility for a task to be scheduled. The possible states of a VRES are the following: `EE_VRES_INACTIVE` means that the VRES is linked to a task which has not been activated or which is blocked. When the task is activated, or when it wokes up from a synchronization, then the VRES is put in the `EE_VRES_ACTIVE` state, as specified by the IRIS [4] scheduling algorithm. When the task uses all its budget, the VRES is put in the state `EE_VRES_RECHARGING`. In this state, the task linked to the VRES cannot be executed, until the VRES recharging time happens. The VRES recharging time depends on the VRES budget, as well as on the idle times which may have the effect to anticipate the recharging instant. Due to the implementation of the circular timer, similar to what is done in the EDF kernel (see Section 2.10), an inactive VRES may be put in the `EE_VRES_FREEZED` status when the deadline of an inactive VRES arrives in the past.

Conformance

FRSH

3.3.3 FRSH API - Init and general functions

3.3.4 frsh_init

Synopsis

`int frsh_init(void);`

Description

We cannot call any frsh functions before frsh_init. After calling frsh_init, the FRSH Kernel will be initialized. The second time this function is called it fails.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_SYSTEM_ALREADY_INITIALIZED` FRSH Kernel already initialized.

Conformance

FRSH

3.3.5 frsh_strerror

Synopsis

```
int frsh_strerror(int error, char *message, size_t size);
```

Description

Converts a FRSH error code to a string.

Parameters

- `error` The error code for which we want the description.
- `message` The error message.
- `size` The size of the buffer.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` The error value is wrong.

Conformance

FRSH

3.3.6 frsh_vres_get_contract

Synopsis

```
int frsh_vres_get_contract (const frsh_vres_id_t vres, frsh_contract_t *contract)
```

Description

This operation stores the contract parameters currently associated with the specified vres in the variable pointed to by contract. It returns an error if the vres_id is not recognised.

Parameters

- `vres` the VRES
- `contract` the pointer to the contract object

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` If the contract argument is NULL or the value of the vres argument is not in range.

Conformance

FRSH

3.3.7 frsh_contract_get_basic_params

Synopsis

```
int frsh_contract_get_basic_params (const frsh_contract_t *contract,
                                    frsh_rel_time_t *budget_min,
                                    frsh_rel_time_t *period_max,
                                    frsh_workload_t *workload,
                                    frsh_contract_type_t *contract_type);
```

Description

This operation obtains from the specified contract object its budget, period, and workload, and copies them to the places pointed to by the corresponding output parameters.

Parameters

- `contract` the pointer to the contract object
- `budget_min` pointer to preallocated space
- `period_max` pointer to preallocated space
- `workload` pointer to preallocated space
- `contract_type` pointer to preallocated space

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` Some of the parameters are NULL.

Conformance

FRSH

3.3.8 frsh_contract_get_timing_reqs

Synopsis

```
int frsh_contract_get_timing_reqs(const frsh_contract_t *contract,
                                   int *d_equals_t,
                                   frsh_rel_time_t *deadline,
                                   frsh_signal_t *budget_overrun_signal,
                                   frsh_signal_info_t *budget_overrun_siginfo,
                                   frsh_signal_t *deadline_miss_signal,
                                   frsh_signal_info_t *deadline_miss_siginfo);
```

Description

The operation obtains the corresponding input parameters from the specified contract object. This function has been implemented only for compatibility with the FRSH API. It returns that Deadline is equal to the period, and there are no signals defined.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` If the contract is NULL.

Conformance

FRSH

3.3.9 frsh_thread_get_vres_id

Synopsis

```
int frsh_thread_get_vres_id(const frsh_thread_id_t thread, frsh_vres_id_t *vres_id);
```

Description

This operation stores the Id of the vres associated with the specified thread in the variable pointed to by vres. It returns an error if the thread does not exist, it is not under the control of the scheduling framework, or is not bound.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_NOT_BOUND` If the given thread does not have a valid vres bound to it.
- `FRSH_ERR_BAD_ARGUMENT` If the given thread does not exist or the vres argument is NULL.

Conformance

FRSH

3.3.10 frsh_contract_get_basic_params

Synopsis

```
int frsh_contract_get_basic_params (const frsh_contract_t *contract,
                                    frsh_rel_time_t *budget_min,
                                    frsh_rel_time_t *period_max,
                                    frsh_workload_t *workload,
                                    frsh_contract_type_t *contract_type);
```

Description

This operation obtains from the specified contract object its budget, period, and workload, and copies them to the places pointed to by the corresponding output parameters

Parameters

- `contract` the pointer to the contract object
- `budget_min` pointer to preallocated space
- `period_max` pointer to preallocated space
- `workload` pointer to preallocated space
- `contract_type` pointer to preallocated space

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if one of the contract or pointers is NULL.

Conformance

FRSH

3.3.11 frsh_config_is_admission_test_enabled

Synopsis

```
bool frsh_config_is_admission_test_enabled(void)
```

Description

Always returns 0.

Conformance

FRSH

3.3.12 frsh_vres_get_remaining_budget

Synopsis

```
int frsh_vres_get_remaining_budget (const frsh_vres_id_t vres, frsh_rel_time_t *budget);
```

Description

This function stores in the variable pointed to by budget the remaining execution-time budget associated with the specified vres.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` If the value of the vres argument is not in range or budget is NULL.

Conformance

FRSH

3.3.13 frsh_vres_get_usage

Synopsis

```
int frsh_vres_get_usage (const frsh_vres_id_t vres, frsh_rel_time_t *spent)
```

Description

This function stores the current execution time spent by the threads bound to the specified vres in the variable pointed to by cpu_time.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if the value of the vres argument is not in range or spent is NULL.

Conformance

FRSH

3.3.14 frsh_vres_get_budget_and_period

Synopsis

```
int frsh_vres_get_budget_and_period (const frsh_vres_id_t vres,
                                     frsh_rel_time_t *budget,
                                     frsh_rel_time_t *period);
```

Description

This function stores in the variables pointed to by budget and period, the execution-time budget and the period respectively associated with the specified vres. If any of these pointers is NULL, the corresponding information is not stored.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if the value of the vres argument is not in range, or budget and period are both NULL.

Conformance

FRSH

3.4 FRSH API - Binding and unbinding

3.4.1 frsh_thread_bind

Synopsis

```
int frsh_thread_bind(const frsh_vres_id_t vres, const frsh_thread_id_t thread)
```

Description

This operation associates a thread with a vres, which means that it starts consuming the vres's budget and is executed according to the contract established for that vres. If the thread is already bound to another vres, it is effectively unbound from it and bound to the specified one.

It fails if there is already a thread bound to this vres.

Parameters

- `vres` the vres to bind to the task
- `thread` the thread to which we have to bind the vres

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if the vres value does not complain with the expected format or valid range or the given thread does not exist.
- `FRSH_ERR_NOT_CONTRACTED_VRES` if the referenced vres is not valid
- `FRSH_ERR_ALREADY_BOUND` if the given vres has a thread already bound

Conformance

FRSH

3.4.2 frsh_thread_unbind

Synopsis

```
int frsh_thread_unbind(const frsh_thread_id_t thread);
```

Description

This operation unbinds a thread from a vres. Since threads with no vres associated are not allowed to execute, they remain in a dormant state until they are either eliminated or bound again.

If the thread is inside a critical section the effects of this call are deferred until the critical section is ended

Parameters

- `thread` The thread to be unbinded

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if the given thread does not exist.
- `FRSH_ERR_NOT_BOUND` if the given thread does not have a valid vres bound to it

Conformance

FRSH

3.5 FRSH API - Synchronization objects

3.5.1 frsh_synchobj_signal

Synopsis

```
int frsh_synchobj_signal(const frsh_synchobj_handle_t synch_handle);
```

Description

This function sends a notification event to the synchronization object specified as parameter. If there is at least one vres waiting on the synchronization object, it is awaken. If more than one vres are waiting, just one of them is awaken. If no vres is waiting on the synchronization object, the notification event is queued.

Parameters

- `synch_handle` The synchronisation object.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if synch_handle is 0.

Conformance

FRSH

3.5.2 frsh_synchobj_wait

Synopsis

```
int frsh_synchobj_wait (const frsh_synchobj_handle_t synch_handle,
                        frsh_rel_time_t *next_budget,
                        frsh_rel_time_t *next_period,
                        bool *was_deadline_missed,
                        bool *was_budget_overran);
```

Description

This operation is invoked by threads to indicate that a job has been completed (and that the scheduler may reassign the unused capacity of the current job to other vres). This implementation de facto does not void the budget, but simply blocks the the task. In fact, the IRIS [4] scheduler automatically reclaims unused bandwidth.

As a difference with `frsh_timed_wait`, here the vres specifies to be awakened by the arrival of a signal operation instead of at a precise point of time.

The vres' budget will be made zero for the remainder of the vres' period, and FRSH will not replenish it until an event has been notified to the synchronisation object by another vres. It can happen that the synchronisation object has notification events queued from the past, in this case one of the events is dequeued immediately and the vres won't have to wait for another one.

At the time of reception of a notification event (wether in the future or in the past), all pending budget replenishments (if any) are made effective. Once the vres has a positive budget and the scheduler schedules the calling thread again, the call returns and the vres continues executing. Except for those parameters equal to NULL pointers, the system reports the current period and budget for the current job, it informs if the deadline of the previous job was missed or not, and whether the budget of the previous job was overrun or not. Note: this implementation using the IRIS scheduler returns the VRES period and budget (which may not be in sync with budget and periods of the task). For the same reason, since there is not a direct link of the budget/period of a vres with the task deadline, the deadline miss and budget overrun information are not provided (in other words, they are meaningless in this implementation).

Parameters

- `synch_handle` Synchronisation object upon which the vres will be waiting.
- `next_budget` Upon return of this function, the variable pointed by this function will be equal to the current vres budget. If this parameter is set to NULL, no action is taken
- `next_period` The vres period upon return (ignored if NULL).

- `was_deadline_missed` NOT IMPLEMENTED
- `was_budget_overran` NOT IMPLEMENTED

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if synch_handle is 0.
- `FRSH_ERR_INTERNAL_ERROR` if the task still uses a resource.

Conformance

FRSH

3.5.3 frsh_synchobj_wait_with_timeout

Synopsis

```
int frsh_synchobj_wait_with_timeout (const frsh_synchobj_handle_t synch_handle,
                                     const frsh_abs_time_t *abs_timeout,
                                     bool *timed_out,
                                     frsh_rel_time_t *next_budget,
                                     frsh_rel_time_t *next_period,
                                     bool *was_deadline_missed,
                                     bool *was_budget_overran);
```

Description

This call is the same as `frsh_synchobj_wait` but with an extra absolute timeout. The timed_out argument, indicates whether the function returned because of the expiration of the timeout or not.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if synch_handle is 0 or the abs_timeout argument is NULL or its value is in the past.
- `FRSH_ERR_INTERNAL_ERROR` if the task still uses a resource.

Conformance

FRSH

3.5.4 frsh_timed_wait

Synopsis

```
int frsh_timed_wait (const frsh_abs_time_t *abs_time,
                      frsh_rel_time_t *next_budget,
                      frsh_rel_time_t *next_period,
                      bool *was_deadline_missed,
                      bool *was_budget_overran);
```

Description

This operation is invoked by threads associated with bounded workload vres to indicate that a job has been completed (and that the scheduler may reassign the unused capacity of the current job to other vres). It is also invoked when the first job of such threads has to be scheduled.

As an effect, the system will make the current vres's budget zero for the remainder of the vres's period, and will not replenish the budget until the specified absolute time. At that time, all pending budget replenishments (if any) are made effective. Once the vres has a positive budget and the scheduler schedules the calling thread again, the call returns and at that time, except for those parameters equal to NULL pointers, the system reports the current period and budget for the current job, whether the deadline of the previous job was missed or not, and whether the budget of the previous job was overrun or not.

Note about this implementation: The same exceptions to this description made for the `frsh_synchobj_wait` applies.

Parameters

- `abs_time` absolute time at which the budget will be replenished
- `next_budget` upon return of this function, the variable pointed by this function will be equal to the current vres budget. If this parameter is set to NULL, no action is taken.
- `next_period` upon return of this function, the variable pointed by this function will be equal to the current vres period. If this parameter is set to NULL, no action is taken.
- `was_deadline_missed` upon return of this function, the variable pointed by this function will be equal to true if the previous vres deadline was missed, to false otherwise. If this parameter is set to NULL, no action is taken.
- `was_budget_overrun` upon return of this function, the variable pointed by this function will be equal to true if the previous vres budget was overrun, to false otherwise. If this parameter is set to NULL, no action is taken.

Return Values

- `FRSH_NO_ERROR` No errors
- `FRSH_ERR_BAD_ARGUMENT` if abs_time is NULL.

Conformance

FRSH

4 History

Version	Comment
1.0.0	First version of the document.
1.0.1	Added few content; new versioning mechanism.
1.1.0	Added description for the EDF kernel. Typos.
1.1.1	Typos.
1.1.2	Typos.Erika Enterprise Basic renamed to Erika Enterprise.
1.1.3	Added FRSH.

Bibliography

[1] Alessio Carlini and Giorgio Buttazzo. An efficient time representation for real-time embedded systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2003), track on Embedded Systems: Applications, Solutions, and Techniques*, Melbourne, Florida, USA, March 2003.

[2] FRESCOR Consortium. Deliverable EP7v2 - multiprocessor execution platforms. http://www.frescor.org, 2005.

[3] FRESCOR Consortium. Ist FRESCOR FP6/2005/IST/5-034026. http://www.frescor.org, 2005.

[4] Luca Marzario, Giuseppe Lipari, Patricia Balbastre, and Alfons Crespo. Iris: a new reclaiming algorithm for server-based real-time systems. In *Proceedings of the Real-Time Application Symposium (RTAS 04)*, Toronto (Canada), May 2004.

Index