ERIKA Enterprise Basic Manual

...multithreading on a thumb!

version: 1.1.2
July 18, 2008

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,
c/o Incubatore Pont-Tech
Viale Rinaldo Piaggio, 32
56025 Pontedera (PI), Italy
Tel: +39 0587 274 823
Fax: +39 0587 291 904

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.

This document is Copyright 2005-2008 Evidence S.r.l.
Information and images contained within this document are copyright and the property of Evidence S.r.l. All trademarks are hereby acknowledged to be the properties of their respective owners. The information, text and graphics contained in this document are provided for information purposes only by Evidence S.r.l. Evidence S.r.l. does not warrant the accuracy, or completeness of the information, text, and other items contained in this document. Matlab, Simulink, Mathworks are registered trademarks of Matworks Inc. Microsoft, Windows are registered trademarks of Microsoft Inc. Java is a registered trademark of Sun Microsystems. OSEK is a registered trademark of Siemens AG. The Microchip Name and Logo, and Microchip In Control are registered trademarks or trademarks of Microchip Technology Inc. in the USA. and other countries, and are used under license. All other trademarks used are properties of their respective owners. This document has been written using LaTeX and LyX.

Contents

1 Introduction **5**
1.1 Erika Enterprise . . . 5

2 API reference **7**
2.1 Introduction . . . 7
2.1.1 Conformance Classes . . . 7
2.1.2 Available primitives . . . 7
2.2 Constants . . . 9
2.2.1 INVALID_TASK . . . 9
2.2.2 EE_MAX_NACT . . . 9
2.2.3 RES_SCHEDULER . . . 9
2.2.4 Task States . . . 9
2.3 Types . . . 11
2.3.1 AlarmType . . . 11
2.3.2 CounterType . . . 11
2.3.3 ResourceType . . . 11
2.3.4 SemType . . . 11
2.3.5 SemRefType . . . 12
2.3.6 TaskType . . . 12
2.3.7 TickType . . . 12
2.3.8 TickRefType . . . 12
2.3.9 TimeAbsType . . . 12
2.3.10 TimeRelType . . . 13
2.4 Object Definitions . . . 14
2.4.1 TASK . . . 14
2.5 Task Primitives . . . 15
2.5.1 ActivateTask . . . 17
2.5.2 Schedule . . . 18
2.6 Resource primitives . . . 19
2.6.1 GetResource . . . 20
2.6.2 ReleaseResource . . . 21
2.7 Interrupt primitives . . . 22
2.8 Counter and Alarms primitives . . . 22
2.8.1 CounterTick . . . 23
2.8.2 GetAlarm . . . 24
2.8.3 SetRelAlarm . . . 25
2.8.4 SetAbsAlarm . . . 26

2.8.5 CancelAlarm . . . 27
2.9 Counting Semaphores . . . 28
2.9.1 STATICSEM . . . 29
2.9.2 InitSem . . . 30
2.9.3 WaitSem . . . 31
2.9.4 TryWaitSem . . . 32
2.9.5 PostSem . . . 33
2.9.6 GetValueSem . . . 34
2.10 Time handling . . . 35
2.10.1 GetTime . . . 36
2.11 System Startup . . . 37

3 History **38**

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 both Fixed Priority (FP) and Earliest Deadline First (EDF) 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].

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).

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 two 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.

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.

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.

2.2 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.

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.

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.

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.

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

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

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

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

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

2.3.8 TickRefType

Description

This is a pointer to `TickType`.

Conformance

FP, EDF

2.3.9 TimeAbsType

Description

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

Conformance

EDF

2.3.10 TimeRelType

Description

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

Conformance

EDF

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

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 is also available if the kernel is configured as multistack, for those tasks which has assigned a private stack. 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.

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 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 EDF deadlines for a pending activation are 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

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

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 kernel). A resource is locked using the primitive `GetResource`, and unlocked using `ReleaseResource`.

A special resource named `RES_SCHEDULER` is also supported. 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.

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

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

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

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

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

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

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

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).

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 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. Unfortunattely, 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 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.

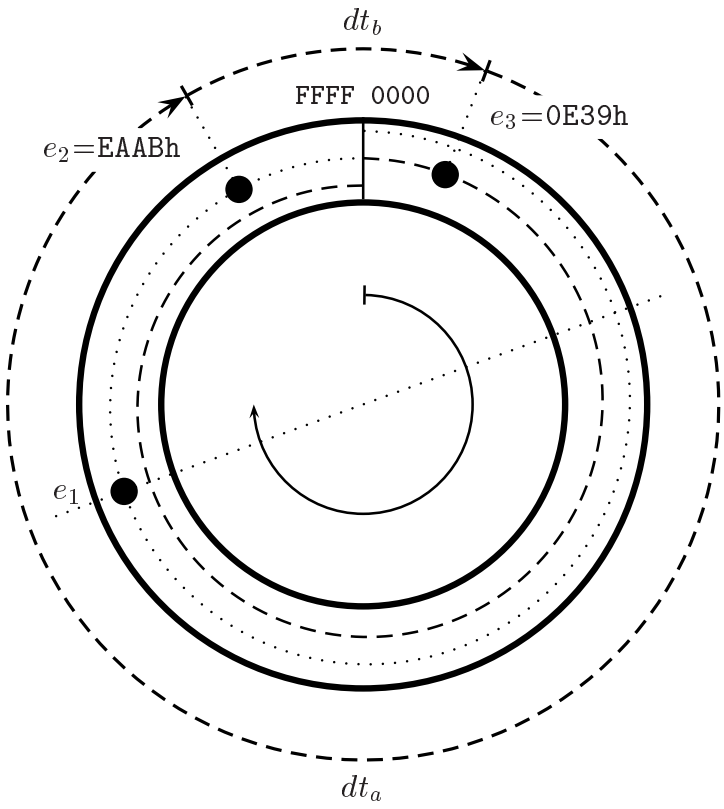

Figure 2.1: The relative timer representation. In the figure, e_2 comes before e_3

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

2.11 System Startup

When using the minimal API, there is no need a specific startup procedure. In particular, the kernel is already active after the first instruction of the `main` function.

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 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.

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.

Index

ActivateTask, 17
AlarmType, 11

CancelAlarm, 27
CounterTick, 23
CounterType, 11

EE_MAX_NACT, 9

GetAlarm, 24
GetResource, 20
GetTime, 36
GetValueSem, 34

InitSem, 30
INVALID_TASK, 9

PostSem, 33

ReleaseResource, 21
RES_SCHEDULER, 9
ResourceType, 11

Schedule, 18
SemRefType, 12
SemType, 11
SetAbsAlarm, 26
SetRelAlarm, 25
STATICSEM, 29
System counters, 22

TASK, 14
Task Instance, 15
Task States, 9
TaskType, 12
TickRefType, 12
TickType, 12
TimeAbsType, 12
TimeRelType, 13
TryWaitSem, 32

WaitSem, 31