ERIKA Enterprise Manual

Real-time made easy

version: 1.4.4
July 23, 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 and RT-Druid

Erika Enterprise is a free of charge, open-source RTOS implementation of the OSEK/VDX API, available for various microcontrollers on the web page http://erika.tuxfamily.org.

Erika Enterprise offers the availability of a real-time scheduler and resource managers allowing the full exploitation of the power of new generation micro-controllers and multicore platforms while guaranteeing predictable real-time performance and retaining the programming model of conventional single processor architectures.

The advanced features provided by Erika Enterprise are:

- Support for four conformance classes to match different application requirements;
- Support for preemptive and non-preemptive multitasking;
- Support for fixed priority scheduling;
- Support for stack sharing techniques, and one-shot task model to reduce the overall stack usage;
- Support for shared resources;
- Support for periodic activations using Alarms;
- Support for centralized Error Handling;
- Support for hook functions before and after each context switch;

The Erika Enterprise kernel is a complete OSEK/VDX environment, which can be used to implement multithreading applications. The Erika Enterprise API providing support for thread activation, mutual exclusion, alarms, events and counting semaphores.

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.

Erika Enterprise is natively supported by RT-Druid, a tool suite for the automatic configuration and deployment of embedded applications which enables to easily exploit multi-processor architectures and achieve the desired performance without modifying the application source code. More details about RT-Druid are available in a dedicated reference manual.

The purpose of this document is to describe in detail the Erika Enterprise API. Details about specific architecture can be found in dedicated manuals, available for download on the Erika web site.

2 API reference

2.1 Introduction

The Erika Enterprise Operating System provides a interface according to OSEK/VDX specification, version 2.2.3. In addition, extended that specification allowing the development of multicore applications.

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

Tasks in Erika Enterprise are scheduled according to fixed priorities, and share resources using the Immediate Priority Ceiling protocol.

On top of task execution there are interrupts, that always preempt the running task to execute urgent operations required by peripherals, or, in case of a multicore system, by other CPUs.

Erika Enterprise can be configured to allow a normal (named Standard Status) or extended (named Extended Status) error recognition.

Extended status is mainly used to check as many errors as possible during the Debug phase. Once the application has been debugged, some error checking can be disabled, saving execution time and code footprint. Standard and Extended status are enabled in the OIL configuration file. When describing Erika Enterprise primitive return values, an "(Extended)" token near the error description means that the error is raised only when the system runs with extended status.

2.1.1 Conformance Classes

To further reduce the overall footprint of the kernel, Erika Enterprise provides subsets of the OS API. These subsets are called *Conformance Classes*. There are four Conformance Classes, named BCC1, BCC2, ECC1, ECC2.

Conformance classes starting with the letter **B** (that is, BCC1 and BCC2) only support Basic Tasks. Conformance classes starting with the letter **E** (that is, ECC1 and ECC2) support both Basic Tasks and Extended Tasks.

BCC1 and BCC2 conformance classes are designed to be as small as possible, and in particular these conformance classes are the most suited to implement small concurrent systems with little RAM footprint, thanks to the stack sharing that can be obtained between basic tasks.

ECC1 and ECC2 conformance classes are designed to be higher end conformance classes supporting synchronization primitives that implies the usage of separate stacks, and of course higher OS overheads due to the stack change mechanisms.

Conformance classes ending with the number 1 (that is, BCC1 and ECC1) does not store pending activations. The ready queue implementation is done using a linear queue with $O(n)$ access time where n is the number of tasks in the ready queue (using a linear queue allow the minimization of the overall OS RAM footprint). Priorities are stored as bit fields, implying that the number of different priorities in the system is limited by the register width (for example, a 32 bit CPU can have up to 32 different priorities).

Conformance classes ending with the number 2 (that is, BCC2 and ECC2) allow a task to store one or more pending activations (the maximum number of pending activation is specified in the `ACTIVATION` attribute of a Task in the OIL Specification). The ready queue implementation is done using a bit field that exposes an $O(1)$ complexity that is independent on the number of tasks in the system. The BCC2 Conformance class have only up to 8 different task priorities, whereas ECC2 has up to 16 different task priorities.

More than one task with the same priority can coexist at the same time for all the four conformance classes.

Resources, Alarms, and Application modes are codified as integers, so there can be up to $2^n - 1$ different entities, where n is the number of bits of the CPU register width (e.g., 32 for a 32 bit CPU).

Figure 2.1 shows the current limits with respect to the number of OS objects allowed in the system.

2.1.2 Available primitives

Erika Enterprise provides a set of primitives that can be called in different situations. The complete list of primitives is listed in Table 2.2, together with the locations where it is legal to call these functions.

The Background Task is the context where the application `main()` function is executed, after the call to `StartOS`. Please note that by default the current implementation of the `StartOS` primitive never returns. To make it return, tou need to add the following option to the OIL file:

```
EE_OPT = __OO_STARTOS_OLD__
```

Feature	BCC1	BCC2	ECC1	ECC2
Multiple requesting of task activation	no	yes	no	BT: yes; ET: no
Number of tasks which are not in the *suspended* state	at least 255		at least 255 (any combination of BT/ET)	
More than one task per priority	yes	yes	yes	yes
Number of events per task	-		*nbits*	
Number of task priorities	*nbits*	16	*nbits*	16
Resources	$2^{nbits} - 1$ (including RES_SCHEDULER)			
Internal Resources	no limit (they are automatically computed by the OIL Compiler)			
Alarms	$2^{nbits} - 1$			
Application modes	$2^{nbits} - 1$			

Table 2.1: This table lists the current limits with respect to the number of OS objects allowed in the system. *nbits* means the number of bits stored in a microcontroller register (16 on a 16 bit machine, 32 on a 32 bit machine, with a minimum value of 16).

Service	Background Task	Task	ISR1	ISR2	ErrorHook	PreTaskHook	PostTaskHook	StartupHook	ShutdownHook	Alarm Callback
ActivateTask	√*	√		√						
TerminateTask		√								
ChainTask		√								
Schedule		√								
ForceSchedule	√*	√*								
GetTaskID		√		√	√	√	√			
GetTaskState		√		√	√	√	√			
DisableAllInterrupts	√*	√	√	√						
EnableAllInterrupts	√*	√	√	√						
SuspendAllInterrupts	√*	√	√	√	√	√	√			√
ResumeAllInterrupts	√*	√	√	√	√	√	√			√
SuspendOSInterrupts	√*	√	√	√						
ResumeOSInterrupts	√*	√	√	√						
GetResource		√		√						
ReleaseResource		√		√						
SetEvent	√*	√		√						
ClearEvent		√								
GetEvent		√		√	√	√	√			
WaitEvent		√								
IncrementCounter	√*	√*		√*						
GetAlarmBase		√		√	√	√	√			
GetAlarm		√		√	√	√	√			
SetRelAlarm		√		√						
SetAbsAlarm		√		√						
CancelAlarm		√		√						
GetActiveApplicationMode		√		√	√	√	√	√	√	
StartOS	√	√								
ShutdownOS		√		√	√			√		

Table 2.2: This table lists the environments where primitives can be called. √* means that the feature is an additional feature of Erika Enterprise that is not part of the OSEK Standard.

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 Error List

Description

This is the list of the error values returned by the kernel primitives:

```
#define E_OK          0
#define E_OS_ACCESS   1
#define E_OS_CALLEVEL 2
#define E_OS_ID       3
#define E_OS_LIMIT    4
#define E_OS_NOFUNC   5
#define E_OS_RESOURCE 6
#define E_OS_STATE    7
#define E_OS_VALUE    8
#define E_OS_SYS_INIT 9
```

2.2.2 INVALID_TASK

Description

This constant represent an invalid task ID, and is returned by `GetTaskID` when the function is called and no task is running.

2.2.3 OSService IDs

Description

This is the list of Service IDs values that can be returned by `OSErrorGetServiceId`:

```
#define OSServiceId_ActivateTask      1U
#define OSServiceId_TerminateTask     2U
#define OSServiceId_ChainTask         3U
#define OSServiceId_Schedule          4U
#define OSServiceId_GetTaskID         5U
#define OSServiceId_GetTaskState      6U
#define OSServiceId_GetResource       7U
#define OSServiceId_ReleaseResource   8U
#define OSServiceId_SetEvent          9U
#define OSServiceId_ClearEvent        10U
#define OSServiceId_GetEvent          11U
#define OSServiceId_WaitEvent         12U
#define OSServiceId_GetAlarmBase      13U
```

```
#define OSServiceId_GetAlarm           14U
#define OSServiceId_SetRelAlarm        15U
#define OSServiceId_SetAbsAlarm        16U
#define OSServiceId_CancelAlarm        17U
#define OSServiceId_IncrementCounter   18U
#define OSServiceId_GetCounterValue    19U
#define OSServiceId_GetElapsedValue    20U
#define OSServiceId_StartOS            21U
#define OSServiceId_ForceSchedule      22U
```

Please note that the primitives:

- DisableAllInterrupts
- EnableAllInterrupts
- SuspendAllInterrupts
- ResumeAllInterrupts
- SuspendOSInterrupts
- ResumeOSInterrupts
- GetActiveApplicationMode
- ShutdownOS

never return an error, and for that reason they are not listed here.

2.2.4 RES_SCHEDULER

Description

This is the ID of the RES_SCHEDULER resource.

That resource exists only when USERESSCHEDULER is set to TRUE within 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.

2.2.5 Task States

Description

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

```
#define RUNNING   0
#define WAITING   1
#define READY     2
#define SUSPENDED 3
```

Please note that the WAITING state is only available in the conformance classes ECC1 and ECC2.

2.2.6 Counters Constants

Description

For all configurated counters RT-Druid generate the return values of `GetAlarmBase` as constants:

OSMAXALLOWEDVALUE_x Maximum possible allowed value of counter x in ticks

OSTICKSPERBASE_x Number of ticks required to reach a specific unit of counter x

OSMINCYCLE_x Minimum allowed number of ticks for a cyclic alarm of counter x

Thus, if the counter name is known, it is not necessary to call `GetAlarmBase`. When system counter is configured, the constants of this counter are additionally accessible via the following constants:

OSMAXALLOWEDVALUE Maximum possible allowed value of the system counter in ticks.

OSTICKSPERBASE Number of ticks required to reach a specific unit of the system counter.

OSMINCYCLE Minimum allowed number of ticks for a cyclic alarm of the system counter.

Additionally the following constant is supplied:

OSTICKDURATION Duration of a tick of the system counter in nanoseconds.

2.2.7 OSDEFAULTAPPMODE

Description

This is the default Application Mode. This value is always a valid Application Mode that can be passed to `StartOS`.

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 type is specified to be of the size of a machine register, it is intended that the type has the same size of the CPU general purpose register.

2.3.1 AlarmBaseType

Description

This structure is used to store the basic information about Counters. It has the following fields:

TickType maxallowedvalue Is the maximum allowed count value in ticks for a counter.

TickType ticksperbase It is the number of ticks required to reach a counter-specific significant unit.

TickType mincycle It is the smallest allowed value for the `cycle` parameter of the primitives `SetRelAlarm`/`SetAbsAlarm`. This field is only present when Extended status is selected.

2.3.2 AlarmBaseRefType

Description

This is a pointer to `AlarmBaseType`.

2.3.3 AlarmType

Description

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

2.3.4 AppModeType

Description

This (unsigned) type is used to store Application Mode IDs, and it has the size of a register.

2.3.5 CounterType

Description

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

2.3.6 EventMaskType

Description

This (unsigned) type is used to store Event masks as bit fields, and it has the size of a register.

2.3.7 EventMaskRefType

Description

This is a pointer to `EventMaskType`.

2.3.8 OSServiceIdType

Description

This unsigned 8-bit integer type is used to store Service IDs, and it is used within the `OSErrorGetServiceId`.

2.3.9 ResourceType

Description

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

2.3.10 SemType

Description

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

2.3.11 SemRefType

Description

This is a pointer to `SemType`.

2.3.12 StatusType

Description

This type is an `unsigned char` used to store function error return values.

2.3.13 TaskType

Description

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

2.3.14 TaskRefType

Description

This is a pointer to `TaskType`.

2.3.15 TaskStateType

Description

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

2.3.16 TaskStateRefType

Description

This is a pointer to `TaskStateType`.

2.3.17 TickType

Description

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

2.3.18 TickRefType

Description

This is a pointer to `TickType`.

2.4 Object Declarations

The following declarations have to be used to declare Tasks, Resources, Alarms, and Events within the application code.

2.4.1 DeclareAlarm

Synopsis

```
DeclareAlarm (AlarmIdentifier)
```

Description

Declares an alarm.

This declaration is currently not mandatory because alarm identifiers are all declared within the code generated by RT-Druid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.4.2 DeclareEvent

Synopsis

```
DeclareEvent(EventID)
```

Description

`DeclareEvent` serves as an external declaration of an event. The function and use of this service are similar to that of the external declaration of variables.

This declaration is currently not mandatory because event identifiers are all declared within the code generated by RT-Druid.

Conformance

ECC1, ECC2

2.4.3 DeclareResource

Synopsis

```
DeclareResource(ResourceID)
```

Description

`DeclareResource` serves as an external declaration of a resource. The function and use of this service are similar to that of the external declaration of variables.

This declaration is currently not mandatory because Resource identifiers are all declared within the code generated by RT-Druid.

Parameters

- `ResourceID` Resource Identifier

Conformance

BCC1, BCC2, ECC1, ECC2

2.4.4 DeclareTask

Synopsis

```
DeclareTask(TaskType TaskID);
```

Description

DeclareTask serves as an external declaration of a task. The function and use of this service are similar to that of the external declaration of variables.

This declaration is currently not mandatory because task identifiers are all declared within the code generated by RT-Druid.

Parameters

- `TaskID` Task reference.

Conformance

BCC1, BCC2, ECC1, ECC2

2.5 Object Definitions

The following macro have to be used when defining Tasks, ISRs and Alarm Callbacks.

2.5.1 ALARMCALLBACK

Synopsis

```
ALARMCALLBACK(t)
```

Description

This macro is used to declare and to define an alarm callback.

Parameters

- `t` Name of the alarm callback.

Conformance

BCC1, BCC2, ECC1, ECC2

2.5.2 ISR

Synopsis

```
ISR(Funcname) {...}
```

Description

The ISR keyword must be used when declaring an ISR function, to distinguish it from other function types and from tasks.

Conformance

BCC1, BCC2, ECC1, ECC2

2.5.3 TASK

Synopsis

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

Description

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

Conformance

BCC1, BCC2, ECC1, ECC2

2.6 Task Primitives

Erika Enterprise supports two flavors of tasks:

Basic Tasks A basic task is the simplest task in Erika Enterprise, providing concurrency together with a one-shot task model. Basic Tasks can share their stack to reduce the overall RAM usage.

Extended Tasks An extended task is a task that can block on the synchronization primitive `WaitEvent`.

Basic 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 basic task, its stack is freed. Basic Tasks never block, and they are the ideal kind of tasks for implementing stack sharing techniques.

Extended Tasks, on the converse, are typically implemented as a never ending task in which each instance ends with a synchronization implemented with a call to the `WaitEvent` primitive. Extended tasks always have a private stack.

RT-Druididentifies a task as Extended when its OIL definition contains the specification of `Events`. A task without any `Event` assigned is a Basic Task.

The scheduling policy of Erika Enterprise is a Fixed Priority Scheduling with Immediate Priority Ceiling. 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 like a task that executes all the time locking a pseudo-resource (also called *Internal Resource*. As a result, only tasks with higher priority than the ceiling of the Internal Resource, and interrupts, can preempt it.

Independently of the task type all kernel primitives that may cause rescheduling may be called in any Tasks' sub-functions.

Tasks are activated using the primitives `ActivateTask` or `ChainTask`. 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 has an effect that depends on how many pending activations the particular Conformance Class can store. BCC1 and ECC1 does not store pending activations, whereas BCC2 and ECC2 can store pending activations if the task has been properly configured in the OIL Configuration file.

Tasks **must end** with a call either to `TerminateTask` or `ChainTask`. Terminating a task without one of these two primitives leads to indefinite results.

On multiprocessor systems, Tasks are statically assigned to CPUs at compile time. The CPU a task is assigned to is specified within the OIL configuration file.

2.6.1 ActivateTask

Synopsis

```
StatusType ActivateTask(TaskType TaskID);
```

Description

This primitive activates a task `TaskID`, putting it in the `READY` state, or in the `RUNNING` state if the scheduler finds that the activated task should become the running task.

Once activated, the task will run for an instance, starting from its first instruction. For the BCC2 and ECC2 Conformance classes, pending activations can be stored if the task has been configured with a number of activations greater than 1 within the OIL configuration file.

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

Parameters

- `TaskID` Task reference.

Return Values

- `E_OK` No error.
- `E_OS_LIMIT` Too many pending activations of `TaskID`.
- `E_OS_ID` (Extended) `TaskID` is invalid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.6.2 TerminateTask

Synopsis

```
StatusType TerminateTask(void)
```

Description

This primitive terminates the calling task. The function can be called from any function nesting: the stack space used by the task is also freed. The calling task should not have any Resource locked when this primitive is called (apart an Internal Resource that is automatically released with this call).

After the call, the calling task is set in the `SUSPENDED` state, and it can be reactivated again using `ActivateTask`, `ChainTask`, or using Alarm notifications.

All the tasks must terminate with a call to `TerminateTask` or `ChainTask`. Otherwise, the behavior after task end is undefined.

With Standard Status, the primitive never returns. With Extended Status, the primitive may return in case of errors.

Return Values

- `no return` (Standard) – in this case, the function is declared as returning `void`.
- `E_OS_RESOURCE` (Extended) The task still occupies resources.
- `E_OS_CALLEVEL` (Extended) The function was called at interrupt level.

Conformance

BCC1, BCC2, ECC1, ECC2

2.6.3 ChainTask

Synopsis

```
StatusType ChainTask(TaskType TaskID);
```

Description

This primitive is similar to `TerminateTask`, with the differences listed below.

After the calling task is terminated, `TaskID` is activated again.

If `TaskID` is the calling task ID, then the calling task is terminated, the Internal Resource is unlocked, and then the calling task is put again in the ready queue to be scheduled. The Internal Resource will be locked again when the task will be again selected for scheduling.

When called successfully, `ChainTask` does not return to the caller. In case of error the primitive returns, and the returned error value can be evaluated by the application.

When an extended task is transferred from suspended state into ready state all its events are cleared.

Parameters

- `TaskID` Task reference

Return Values

- `No return` If the call is successful.
- `E_OS_LIMIT` Too many task activations of `TaskID`. The Task activation in this case is ignored.
- `E_OS_ID` (Extended) Task `TaskID` is invalid.
- `E_OS_RESOURCE` (Extended) Calling task still occupies resources.
- `E_OS_CALLEVEL` (Extended) Call at interrupt level.

Conformance

BCC1, BCC2, ECC1, ECC2

2.6.4 Schedule

Synopsis

```
StatusType Schedule(void)
```

Description

This primitive can be used as a rescheduling point for tasks that have Internal Resources and for non preemptive tasks[1].

When this primitive is called, the task releases its Internal Resource, and checks if there are higher priority tasks that have to preempt (In that case, a preemption is implemented). When the primitive returns, the task will reacquire its internal resource.

The primitive does nothing if the calling task has no internal resource assigned.

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` No error.
- `E_OS_CALLEVEL` (Extended) The primitive was called at interrupt level.
- `E_OS_RESOURCE` (Extended) The calling task occupies resources.

Conformance

BCC1, BCC2, ECC1, ECC2

[1] Non preemptive tasks are tasks with an Internal Resource with the highest priority ceiling available assigned.

2.6.5 ForceSchedule

Synopsis

```
StatusType ForceSchedule(void)
```

Description

This function implements a preemption check. If a higher-priority task is ready, the current task is put into the ready state, its context is saved and the higher-priority task is executed. Otherwise the calling task is continued.

The difference of this primitive with respect to `Schedule` is that Internal Resources are not released.

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` No error.
- `E_OS_CALLEVEL` (Extended) Call at interrupt level.

Conformance

BCC1, BCC2, ECC1, ECC2

2.6.6 GetTaskID

Synopsis

```
void GetTaskID ( TaskRefType TaskID )
```

Description

`GetTaskID` returns the TaskID of the running task.
If no task is running, `INVALID_TASK` is returned.

Conformance

BCC1, BCC2, ECC1, ECC2

2.6.7 GetTaskState

Synopsis

```
StatusType GetTaskState(TaskType TaskID, TaskStateRefType State)
```

Description

This primitive returns the state of a given task. possible states are listed in Section 2.2.5.

If the task `TaskID` supports pending activation, and the task has been activated more than once, the results refer to the state of its oldest activation.

Parameters

- `TaskID` (in) Task reference.
- `State` (out) Reference to the state of task `TaskID`.

Return Values

- `void` (Standard) The function returns void if Standard Mode is used.
- `E_OK` (Extended) No error.
- `E_OS_ID` (Extended) Task `TaskID` is invalid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.7 Interrupt primitives

Erika Enterprise 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 way possible, but on the other side lack the possibility to call OS services.

On the other end, 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.

Erika Enterprise also offers a set of primitives to directly control interrupt disabling and enabling, with also a nested version of these primitives.

2.7.1 DisableAllInterrupts

Synopsis

```
void DisableAllInterrupts(void)
```

Description

`DisableAllInterrupts` and `EnableAllInterrupts` are used to implement critical sections with interrupt disabled.

This primitive disables all the interrupts sources in the system, and saves the interrupt state that will be restored by a call to `EnableAllInterrupts`.

The primitive may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines. No primitives can be called within critical sections surrounded by `DisableAllInterrupts` and `EnableAllInterrupts`.

Critical sections using `DisableAllInterrupts` `EnableAllInterrupts` cannot be nested. If you need nested critical sections, please use `SuspendOSInterrupts` / `ResumeOSInterrupts` or `SuspendAllInterrupts` / `ResumeAllInterrupts`.

Conformance

BCC1, BCC2, ECC1, ECC2

2.7.2 EnableAllInterrupts

Synopsis

```
void EnableAllInterrupts(void)
```

Description

DisableAllInterrupts and EnableAllInterrupts are used to implement critical sections with interrupt disabled.

This primitive restores the state saved by DisableAllInterrupts, enabling the recognition of interrupts.

The primitive may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines. No primitives can be called inside critical sections surrounded by DisableAllInterrupts and EnableAllInterrupts.

Conformance

BCC1, BCC2, ECC1, ECC2

2.7.3 SuspendAllInterrupts

Synopsis

```
void SuspendAllInterrupts(void)
```

Description

SuspendAllInterrupts and ResumeAllInterrupts are used to implement critical sections with interrupt disabled, with nesting support.

This primitive disables all the interrupts sources in the system, and saves the interrupt state that will be restored by a call to ResumeAllInterrupts.

The service may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from all hook routines.

No primitive calls beside SuspendAllInterrupts / ResumeAllInterrupts pairs and SuspendOSInterrupts / ResumeOSInterrupts pairs are allowed within this critical section.

Conformance

BCC1, BCC2, ECC1, ECC2

2.7.4 ResumeAllInterrupts

Synopsis

```
void ResumeAllInterrupts(void)
```

Description

SuspendAllInterrupts and ResumeAllInterrupts are used to implement critical sections with interrupt disabled, with nesting support.

This primitive restores the state saved by SuspendAllInterrupts, enabling the recognition of interrupts if it is the last call in a series of nested calls of SuspendAllInterrupts / ResumeAllInterrupts and SuspendOSInterrupts / ResumeOSInterrupts pairs.

This primitive may be called from an ISR category 1 and category 2, from alarm-callbacks and from the task level, but not from all hook routines.

Conformance

BCC1, BCC2, ECC1, ECC2

2.7.5 SuspendOSInterrupts

Synopsis

```
void SuspendOSInterrupts(void)
```

Description

SuspendOSInterrupts and ResumeOSInterrupts are used to implement critical sections with interrupt category 2 disabled, with nesting support.

This primitive disables all the interrupts sources of category 2 in the system, and saves the interrupt state that will be restored by a call to ResumeOSInterrupts.

The service may be called from an ISR category 1 and category 2, and from the task level.

No primitive calls beside SuspendAllInterrupts / ResumeAllInterrupts pairs and SuspendOSInterrupts / ResumeOSInterrupts pairs are allowed within this critical section.

Conformance

BCC1, BCC2, ECC1, ECC2

2.7.6 ResumeOSInterrupts

Synopsis

```
void ResumeOSInterrupts(void)
```

Description

SuspendOSInterrupts and ResumeOSInterrupts are used to implement critical sections with interrupt category 2 disabled, with nesting support.

This primitive restores the state saved by SuspendOSInterrupts, enabling the recognition of interrupts if it is the last call in a series of nested calls of SuspendAllInterrupts / ResumeAllInterrupts and SuspendOSInterrupts / ResumeOSInterrupts pairs.

The primitive may be called from an ISR category 1 and category 2 and from the task level, but not from hook routines.

Conformance

BCC1, BCC2, ECC1, ECC2

2.8 Resource primitives

Resources are the term used by Erika Enterprise to refer to binary semaphores used to implement shared critical sections.

Resources are implemented using the Immediate Priority Ceiling protocol. 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 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.

On multiprocessor systems, Resources are divided in:

Local resources A Resource is local when all the tasks that uses it are assigned to the same processor.

Global resources A Resource is global when the tasks that uses it are assigned to different processors.

A special kind of resources, called *Internal Resources*, are also supported by Erika Enterprise. Internal Resources are locked when the tasks enter the `RUNNING` state, and it is released when the task ends. Internal resources are used by optimization algorithms to limit the maximum stack space used by application tasks. Please note that `Schedule` explicitly release any Internal Resource locked by the running task, thus limiting the possibility to reduce the overall stack in the system. Also `WaitEvent` always release the internal resource of the task; however, this fact does not impact on stack usage because tasks using `WaitEvent` must run on a private stack since `WaitEvent` is a blocking primitive. Please also note that on Multicore systems, Internal Resources can only be *local*. Global Internal Resources are not supported.

The primitives `GetResource` and `ReleaseResource` automatically internally uses a spinlock mechanism when called on a Global Resource.

Erika Enterprise support resource management at ISR level for a selected number of architectures. Please check on the Erika Enterprise wiki whether the architecture you are currently using supports this feature. The implementation of this feature is implemented extending the OIL configuration to accept ISR priority as extra field in ISR Object definition. The ISR Priority field contains architecture independent values which are then mapped by RT-Druid to real interrupt priority register values.

2.8.1 GetResource

Synopsis

```
StatusType 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 `TerminateTask`, `ChainTask`, `Schedule`, and `WaitEvent` 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 is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error.
- `E_OS_ID` (Extended) Resource `ResID` is invalid.
- `E_OS_ACCESS` (Extended) Attempt to get a resource which is already occupied by any task or ISR, or the statically assigned priority of the calling task or interrupt routine is higher than the calculated ceiling priority.

Conformance

BCC1, BCC2, ECC1, ECC2

2.8.2 ReleaseResource

Synopsis

```
StatusType 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 is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error
- `E_OS_ID` (Extended) `ResID` is an invalid identifier
- `E_OS_NOFUNC` (Extended) Attempt to release a resource which is not occupied by any task or ISR, or another resource shall be released before.
- `E_OS_ACCESS` (Extended) Attempt to release a resource which has a lower ceiling priority than the statically assigned priority of the calling task or interrupt routine.

Conformance

BCC1, BCC2, ECC1, ECC2

2.9 Event related primitives

Events represents a technique used by Erika Enterprise to implement synchronization primitives. Events are assigned to tasks. Tasks with events assigned to are called *Extended tasks*. Tasks without events assigned to are called *Basic Tasks*.

Extended tasks are supported only in the ECC1 and ECC2 conformance classes. To assign an event to a task, the event have to be listed inside the task declaration in the OIL configuration file.

Events are implemented as bits in a bit mask. Each task in the system is associated to a bit mask, which is typically as large as a CPU data register. The bit mask is initialized to 0 at system startup.

The status of an extended task event mask can be read by tasks and ISRs using the `GetEvent` primitive. Events can also be set from tasks or ISRs using the `SetEvent` primitive (more than one event can be set with a single call to `SetEvent`).

An extended task can wait for one ore more events from an event mask to be set using the `WaitEvent` primitive. An extended task needs then to explicitly clear an event calling the `ClearEvent` primitive.

Calls to WaitEvent may provoke the task to block. For that reason, extended tasks must have a private stack assigned to them, implying also the fact that the multistack kernel have to be used.

2.9.1 SetEvent

Synopsis

```
StatusType SetEvent (TaskType TaskID, EventMaskType Mask)
```

Description

The events of task `TaskID` are set according to the event mask `Mask`. The call to `SetEvent` may cause `TaskID` to wakeup from a `WaitEvent` primitive. Any events not set in the event mask remain unchanged. The service may be called from an interrupt service routine and from the task level, but not from hook routines.

The function can be called from the Background task. Typically, it is called within the `main()` function.

Parameters

- `TaskID` Task identifier
- `Mask` Mask of the events to be set

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error
- `E_OS_ID` (Extended) Reference `TaskID` is invalid.
- `E_OS_ACCESS` (Extended) Task `TaskID` is not an extended task.
- `E_OS_STATE` (Extended) Events can not be set as the referenced task is in the suspended state.

Conformance

ECC1, ECC2

2.9.2 ClearEvent

Synopsis

```
StatusType ClearEvent (EventMaskType Mask)
```

Description

`ClearEvent` clears the events `Mask` of the calling task.

This system call is restricted to extended tasks which own the event.

Parameters

- `Mask` Mask of the event to be cleared

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error
- `E_OS_ACCESS` (Extended) The service has been invoked by a non-extended task.
- `E_OS_CALLEVEL` (Extended) The service has been invoked at the interrupt level.

Conformance

ECC1, ECC2

2.9.3 GetEvent

Synopsis

```
StatusType GetEvent(TaskType TaskID, EventMaskRefType Event)
```

Description

This primitive returns the current state of all event bits of the task `TaskID`. The service may be called from interrupt service routines, task level and some hook routines. The current status of the event mask of task `TaskID` is copied to `Event`. The referenced task shall be an extended task.

Parameters

- `TaskID` task whose mask is to be returned.
- `Event` (out) returned mask.

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error
- `E_OS_ID` (Extended) `TaskID` is an invalid reference.
- `E_OS_ACCESS` (Extended) `TaskID` is not an extended task.
- `E_OS_STATE` (Extended) `TaskID` is in suspended state.

Conformance

ECC1, ECC2

2.9.4 WaitEvent

Synopsis

```
StatusType WaitEvent (EventMaskType Mask)
```

Description

The calling task blocks if none of the events specified in `Mask` are set.

If the calling task blocks, the system is reschedule, and the Internal resource of the task is released. This service shall only be called from the extended task owning the events.

Parameters

- `Mask` mask of the events waited for

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error
- `E_OS_ACCESS` (Extended) The calling task is not extended
- `E_OS_RESOURCE` (Extended) Calling task occupies resources.
- `E_OS_CALLEVEL` (Extended) Call at interrupt level.

Conformance

ECC1, ECC2

2.10 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 `IncrementCounter`.

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.

Event set. In this case, an event mask is set on a task when the Alarm fires.

Alarm callback. In this case, an alarm callback (defined using `ALARMCALLBACK`) is called.

The notifications are executed inside the `IncrementCounter` 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.

On multiprocessor systems, Counters are statically assigned to CPUs at compile time. Counters are local to (and only visible in) a CPU. Alarms are local to the CPU that hosts the counter they are linked to.

> **Warning:** The OSEK/VDX standard provides support for s (e.g., counters that are automatically linked to hardware timers). Please check on the Erika Enterprise wiki whether the architecture you are using supports this feature. If the architecture does not support the feature, then all the counters have to be defined inside the OIL Configuration file, and the user have to call `IncrementCounter` to increment them.

2.10.1 IncrementCounter

Synopsis

```
StatusType IncrementCounter(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 trigger the counter is counting 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;
- an event mask set on an extended task.

The function is atomic, and a reschedule will happen at the end if the primitive is called at task level. If called at ISR level, the reschedule will happen at the end of the outermost nested IRQ.

Parameters

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

Return Values

- `E_OK` No error.
- `E_OS_ID` (Extended) Reference `CounterID` is invalid or counter is implemented in hardware and can not be incremented by software.

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.2 GetCounterValue

Synopsis

```
StatusType GetCounterValue(CounterType CounterID, TickRefType Value)
```

Description

This service reads the current count value of a counter returning or the software ticks.

Parameters

- `CounterID` The Counter which tick value should be read.
- `Value` Will contains the current tick value of the counter.

Return Values

- `E_OK` No error.
- `E_OS_ID` (Extended) Reference `CounterID` is invalid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.3 GetElapsedValue

Synopsis

```
StatusType GetElapsedValue(CounterType CounterID, TickRefType Value, TickRefType ElapsedVal
```

Description

This service gets the number of ticks between the current tick value and a previously read tick value. If the timer already passed the ¡Value¿ value a second (or multiple) time, the result returned is wrong. The reason is that the service can not detect such a relative overflow.

Parameters

- `CounterID` The Counter which tick value should be read.
- `Value` Contains the previously read tick value the counter.
- `ElapsedValue` Will contains the difference to the previous read value.

Return Values

- `E_OK` No error.
- `E_OS_ID` (Extended) Reference `CounterID` is invalid.
- `E_OS_VALUE` (Extended) The given Value is invalid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.4 GetAlarmBase

Synopsis

```
StatusType GetAlarmBase(AlarmType AlarmID, AlarmBaseRefType Info)
```

Description

Returns the alarm base characteristics. The return value `Info` is a structure in which the information of data type `AlarmBaseType` is stored.

Allowed on task level, ISR, and in several hook routines.

Parameters

- `AlarmID` Alarm identifier.
- `Info` Reference to the structure containing the constants that define the alarm base.

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` (Extended) No error.
- `E_OS_ID` (Extended) Reference `AlarmID` is invalid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.5 GetAlarm

Synopsis

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

Description

The system service GetAlarm returns the relative value in ticks before the alarm `AlarmID` expires. If `AlarmID` is not in use, `Tick` has an undefined value. 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

- `E_OK` No error
- `E_OS_NOFUNC AlarmID` is not used
- `E_OS_ID` (Extended) reference `AlarmID` is invalid

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.6 SetRelAlarm

Synopsis

```
StatusType 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. To change values of alarms already in use the alarm shall be canceled first.

If the alarm is already in use, this call will be ignored and the error `E_OS_STATE` is returned. Allowed on task level and in ISR, but not in hook routines.

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

- `E_OK` No error.
- `E_OS_STATE` Alarm is already in use.
- `E_OS_ID` (Extended) reference `AlarmID` is invalid
- `E_OS_VALUE` (Standard and Extended) Value of `increment` equal to 0.
- `E_OS_VALUE` (Extended) Value of `increment` outside of the admissible limits, or value of `cycle` unequal to 0 and outside of the admissible counter limits.

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.7 SetAbsAlarm

Synopsis

```
StatusType 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 equal to the actual counter value, the alarm not expire immediately but will expire the next time the counter will reach the `start` value..

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. To change values of alarms already in use the alarm shall be canceled first. If the alarm is already in use, this call will be ignored and the error `E_OS_STATE` is returned. Allowed on task level and in ISR, but not in hook routines.

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

- `E_OK` No error.
- `E_OS_STATE` Alarm is already in use.
- `E_OS_ID` (Extended) reference `AlarmID` is invalid
- `E_OS_VALUE` (Extended) Value of `start` outside of the admissible limits, or value of `cycle` unequal to 0 and outside of the admissible counter limits

Conformance

BCC1, BCC2, ECC1, ECC2

2.10.8 CancelAlarm

Synopsis

```
StatusType CancelAlarm (AlarmType AlarmID)
```

Description

The primitive cancels the alarm `AlarmID`. Allowed on task level and in ISR, but not in hook routines.

Parameters

- `AlarmID` reference to alarm

Return Values

- `E_OK` No error.
- `E_OS_STATE` Alarm is already in use.
- `E_OS_ID` (Extended) reference `AlarmID` is invalid.

Conformance

BCC1, BCC2, ECC1, ECC2

2.11 Internal Messages

Erika Enterprise supports internal messaging following the OSEK COM specifiations, conformace classes CCCA and CCCB. Please refer to the Erika Enterprise COM Manual for more information.

2.12 Counting Semaphores

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

Counting semaphores are an RTOS abstractions 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. For this reason, the `WaitSem` primitive can only be called by extended tasks in conformance classes ECC1 and ECC2. Semaphores are available as non blocking in conformance classes BCC1 and BCC2.

Semaphores can 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.

The current semaphore implementation does *not* support multiprocessor systems. That is, semaphores must be defined and used locally to the same CPU.

> **Warning:** Counting semaphores *do not* avoid Priority Inversion problems. Please use Resources instead (see Section 2.8).

2.12.1 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.

Conformance

BCC1, BCC2, ECC1, ECC2

2.12.2 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

BCC1, BCC2, ECC1, ECC2

2.12.3 WaitSem

Synopsis

```
StatusType 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.

This function can only be called by extended tasks in conformance classes ECC1 and ECC2.

Parameters

- `s` The semaphore used by the primitive.

Return Values

- `void` The function is redefined as returning `void` when Standard Error is used.
- `E_OK` No error.
- `E_OS_CALLEVEL` (Extended) The primitive was called at interrupt level.
- `E_OS_RESOURCE` (Extended) The calling task occupies resources.
- `E_OS_ACCESS` (Extended) The calling task is not an extended task.

Conformance

ECC1, ECC2

2.12.4 TryWaitSem

Synopsis

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

Description

This is a non-blocking version of `WaitSem`. 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 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

BCC1, BCC2, ECC1, ECC2

2.12.5 PostSem

Synopsis

```
StatusType 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

- `E_OK` No error.
- `E_OS_VALUE` The semaphore has not been incremented because its counter was equal to the semaphore maximum value `EE_MAX_SEM_COUNTER`.

Conformance

BCC1, BCC2, ECC1, ECC2

2.12.6 GetValueSem

Synopsis

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

Description

This primitive returns the value of the semaphore counter.

Parameters

- `s` The semaphore used by the primitive.

Return Values

- `int` The semaphore counter value.

Conformance

BCC1, BCC2, ECC1, ECC2

2.13 Application modes, Startup and Shutdown primitives

Erika Enterprise supports the specification of a set of *Application Modes*. Application modes are startup configurations that are used to configure the running application for a certain mode of operation. Examples of Application Modes are for example "Normal execution", "Flash Programming", "Debug Mode", and so on.

The main idea is that the CPU startup code somehow discovers the current Application mode[2]. Once done that, the application mode is passed to `StartOS`, that sets the application mode for this run. Once the system is started, the application mode value can be read using a call to `GetActiveApplicationMode`.

Application modes are listed inside the OIL configuration file. There always exist at least one application mode called `OSDEFAULTAPPMODE`. Once set at startup into `StartOS`, the Application mode cannot be changed.

Application modes are also useful to autostart tasks and alarms following a call to `StartOS`. Tasks and alarms autostart features must be specified inside the OIL configuration file.

Application modes should not be used to map application states that may vary during the application execution.

The primitive `ShutdownOS` is used to prepare the system for system shutdown. Currently the function simply calls `ShutdownHook` and then it starts a forever loop waiting for an hardware reset.

[2]Typical ways to discover Application Modes are for example the usage of dip switches on the device board.

2.13.1 GetActiveApplicationMode

Synopsis

```
AppModeType GetActiveApplicationMode(void)
```

Description

The function returns the current Application Mode, that has been set up by `StartOS` at system startup.

Return Values

- `Application mode` The function returns the current Application Mode.

Conformance

BCC1, BCC2, ECC1, ECC2

2.13.2 StartOS

Synopsis

```
StatusType StartOS (AppModeType Mode)
```

Description

The user can call this system service to start the operating system in a specific Application mode. Only allowed outside of the operating system, at startup. The function calls in order the StartupHook, then it activates the tasks and set the alarms AUTOSTART set as TRUE inside the OIL configuration file. After that, the system is rescheduled and the highest priority ready task is executed.

The StartOS primitive by default never returns to the caller.

Please note that old versions of Erika Enterprise implemented the StartOS primitive in a way that it was returning to the user, to enable the possibility to do some meaningful background activities. To re-enable this old behavior, please add the following line to the OIL file:

```
EE_OPT = "__OO_STARTOS_OLD__"
```

If the function is configured to return to the caller, then it will return after the first idle time is reached. It is up to the caller to implement a meaningful background activity. If unsure of a meaningful background activity, just use the `for(;;)` construct.

Parameters

- `Mode` application mode

Return Values

- `E_OK` (Extended) No error.
- `E_OS_SYS_INIT` (Extended) Error during initialization.
- `void` The function returns void when Extended Status is not selected.

Conformance

BCC1, BCC2, ECC1, ECC2

2.13.3 ShutdownOS

Synopsis

```
void ShutdownOS (StatusType Error)
```

Description

The user can call this system service to abort the overall system (e.g. emergency off).

If a `ShutdownHook` is configured the hook routine `ShutdownHook` is always called (with `Error` as argument) before shutting down the operating system.

The Operating system shutdown is currently implemented as a forever loop.

Parameters

- `Error` The identifier of the error that occurred.

Conformance

BCC1, BCC2, ECC1, ECC2

2.14 Hooks and Error handling primitives

Erika Enterprisesupports five application callbacks that are called when specific situations happens during application execution.

On multiprocessor systems, hooks are local callbacks to each processor.

The `ErrorHook` callback is called every time an error is detected inside an Erika Enterpriseprimitive. The callback can be used to implement centralized error handling. A set of macros are also available to better understand the source of the error (see Section 2.15).

The `StartupHook` callback is called inside the `StartOS` primitive to implement the application startup procedure.

The `ShutdownHook` callback is called inside the `ShutdownOS` primitive to implement the Application specific shutdown procedures.

The `PreTaskHook` and `PostTaskHook` callbacks are called respectively every time a task becomes the `RUNNING` Task, and every time a task stops to be the `RUNNING` Task.

The application can only call a limited set of primitives inside Hook functions.

2.14.1 Placement of the application startup code

In general, there are three ways where the application startup code can be placed:

1. before the call to `StartOS`. This case is typically used for non-OS related initializations, because calling Erika Enterprise primitives before `StartOS` may have in general unpredictable results.

2. inside `StartupHook`. This case is used for the initializations that require a call to Erika Enterprise primitives. Please note that `StartupHook` is called before the rescheduling in `StartOS` takes place.

3. after the call to `StartOS`. This case can be used also for Erika Enterprise related initializations: the startup code is called *after* the end of `StartOS`, and that means at the first idle time *after* the autostart tasks has been activated and executed Please note that if the autostart tasks never terminate, these initializations are never called.

2.14.2 ErrorHook

Synopsis

```
void ErrorHook (StatusType Error)
```

Description

When configured in the OIL File, the system calls this callback every time the return value of a function is different from `E_OK`. The application can then get additional informations using the `OSErrorGetServiceId` macro, that returns the function that is generating the Error. Once the function that generated the error is known, the application can also access the parameters that generated the error using the `OSError_XXX_YYY` macros, where `XXX` is the name of the function, and `YYY` is the name of the parameter passed to the function.

Parameters

- `Error` the identifier of the error that occurred

Conformance

BCC1, BCC2, ECC1, ECC2

2.14.3 PreTaskHook

Synopsis

```
void PreTaskHook (void)
```

Description

When configured in the OIL File, this hook function is called every time a task becomes the `RUNNING` task due to the call of other functions, or due to the preemption done by interrupts. The ID of the task which has just become the `RUNNING` task can be read using the `GetTaskID` function.

Conformance

BCC1, BCC2, ECC1, ECC2

2.14.4 PostTaskHook

Synopsis

```
void PreTaskHook (void)
```

Description

When configured in the OIL File, this hook function is called every time a task is no more the `RUNNING` task due to the call of other functions, or due to the preemption done by interrupts. The ID of the task which has just finished to be the `RUNNING` task can be read using the `GetTaskID` function.

Conformance

BCC1, BCC2, ECC1, ECC2

2.14.5 StartupHook

Synopsis

```
void StartupHook (void)
```

Description

When configured in the OIL File, this hook function is called inside `StartOS`. Inside this callback, the Application can call the Operating System-related functions `ActivateTask`, `SetRelAlarm`, and `SetAbsAlarm`. Please note that the simplest way to activate a task or set an alarm at startup is to specify their initial activation/setting inside the OIL File.

Conformance

BCC1, BCC2, ECC1, ECC2

2.14.6 ShutdownHook

Synopsis

```
void ShutdownHook (StatusType Error)
```

Description

When configured in the OIL File, this hook function is called inside `ShutdownOS`. Inside this callback, the Application can implement application dependent shutdown functions.

Parameters

- `Error` the identifier of the error that occurred, that is the same value that has been passed to the ShutdownOS primitive.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15 ErrorHook Macros

These macros are meaningful inside the ErrorHook Hook function, and are used to better understand the source of the error. In particular, ErrorHook receives as parameter the error that is raised by the primitive. Then, a call to OSErrorGetServiceId returns informations about which primitive caused the error. Finally, calls to the macros OSError_XXX_YYY returns the values of the YYY parameter of the primitive XXX.

2.15.1 OSErrorGetServiceId

Synopsis

```
OSServiceIdType OSErrorGetServiceId(void)
```

Description

The function may be used inside ErrorHook to return the Service ID that generated the error that caused the call to ErrorHook.

Return Values

- Service ID The service ID causing the error.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.2 OSError_ActivateTask_TaskID

Synopsis

```
TaskType OSError_ActivateTask_TaskID(void)
```

Description

The function returns the TaskID parameter passed to ActivateTask. The function must be used inside ErrorHook after having discovered using OSErrorGetServiceId that the error was caused by that function.

Return Values

- Task ID The value of the TaskID parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.3 OSError_ChainTask_TaskID

Synopsis

```
TaskType OSError_ChainTask_TaskID(void)
```

Description

The function returns the `TaskID` parameter passed to `ChainTask`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Task ID` The value of the `TaskID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.4 OSError_GetTaskState_TaskID

Synopsis

```
TaskType OSError_GetTaskState_TaskID(void)
```

Description

The function returns the `TaskID` parameter passed to `GetTaskState`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Task ID` The value of the `TaskID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.5 OSError_GetTaskState_State

Synopsis

```
TaskStateRefType OSError_GetTaskState_State(void)
```

Description

The function returns the `State` parameter passed to `GetTaskState`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `StateRef` The value of the `State` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.6 OSError_GetResource_ResID

Synopsis

```
ResourceType OSError_GetResource_ResID(void)
```

Description

The function returns the `ResID` parameter passed to `GetResource`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Resource ID` The value of the `ResID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.7 OSError_ReleaseResource_ResID

Synopsis

```
ResourceType OSError_ReleaseResource_ResID(void)
```

Description

The function returns the `ResID` parameter passed to `ReleaseResource`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Resource ID` The value of the `ResID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.8 OSError_SetEvent_TaskID

Synopsis

```
TaskType OSError_SetEvent_TaskID(void)
```

Description

The function returns the `TaskID` parameter passed to `SetEvent`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Task ID` The value of the `TaskID` parameter.

Conformance

ECC1, ECC2

2.15.9 OSError_SetEvent_Mask

Synopsis

```
EventMaskType OSError_SetEvent_Mask(void)
```

Description

The function returns the `Mask` parameter passed to `SetEvent`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Mask` The value of the `Mask` parameter.

Conformance

ECC1, ECC2

2.15.10 OSError_ClearEvent_Mask

Synopsis

```
EventMaskType OSError_ClearEvent_Mask(void)
```

Description

The function returns the `Mask` parameter passed to `ClearEvent`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Mask` The value of the `Mask` parameter.

Conformance

ECC1, ECC2

2.15.11 OSError_GetEvent_TaskID

Synopsis

```
TaskType OSError_GetEvent_TaskID(void)
```

Description

The function returns the `TaskID` parameter passed to `GetEvent`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Task ID` The value of the `TaskID` parameter.

Conformance

ECC1, ECC2

2.15.12 OSError_GetEvent_Event

Synopsis

```
EventMaskRefType OSError_GetEvent_Event(void)
```

Description

The function returns the `TaskID` parameter passed to `GetEvent`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Event` The value of the `Event` parameter.

Conformance

ECC1, ECC2

2.15.13 OSError_WaitEvent_Mask

Synopsis

```
EventMaskType OSError_WaitEvent_Mask(void)
```

Description

The function returns the `Mask` parameter passed to `WaitEvent`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Mask` The value of the `Mask` parameter.

Conformance

ECC1, ECC2

2.15.14 OSError_GetAlarmBase_AlarmID

Synopsis

```
AlarmType OSError_GetAlarmBase_AlarmID(void)
```

Description

The function returns the `AlarmID` parameter passed to `GetAlarmBase`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Alarm ID` The value of the `AlarmID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.15 OSError_GetAlarmBase_Info

Synopsis

```
AlarmBaseRefType OSError_GetAlarmBase_Info(void)
```

Description

The function returns the `Info` parameter passed to `GetAlarmBase`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Info` The value of the `Info` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.16 OSError_GetAlarm_AlarmID

Synopsis

```
AlarmType OSError_GetAlarm_AlarmID(void)
```

Description

The function returns the `AlarmID` parameter passed to `GetAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Alarm ID` The value of the `AlarmID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.17 OSError_GetAlarm_Tick

Synopsis

```
TickRefType OSError_GetAlarm_Tick(void)
```

Description

The function returns the `Tick` parameter passed to `GetAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Tick` The value of the `Tick` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.18 OSError_SetRelAlarm_AlarmID

Synopsis

```
AlarmType OSError_SetRelAlarm_AlarmID(void)
```

Description

The function returns the `AlarmID` parameter passed to `SetRelAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Alarm ID` The value of the `AlarmID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.19 OSError_SetRelAlarm_increment

Synopsis

```
TickType OSError_SetRelAlarm_increment(void)
```

Description

The function returns the `increment` parameter passed to `SetRelAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `increment` The value of the `increment` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.20 OSError_SetRelAlarm_cycle

Synopsis

```
TickType OSError_SetRelAlarm_cycle(void)
```

Description

The function returns the `cycle` parameter passed to `SetRelAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `cycle` The value of the `cycle` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.21 OSError_SetAbsAlarm_AlarmID

Synopsis

```
AlarmType OSError_SetAbsAlarm_AlarmID(void)
```

Description

The function returns the `AlarmID` parameter passed to `SetAbsAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Alarm ID` The value of the `AlarmID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.22 OSError_SetAbsAlarm_start

Synopsis

```
TickType OSError_SetAbsAlarm_start(void)
```

Description

The function returns the `start` parameter passed to `SetAbsAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `start` The value of the `start` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.23 OSError_SetAbsAlarm_cycle

Synopsis

```
TickType OSError_SetAbsAlarm_cycle(void)
```

Description

The function returns the `cycle` parameter passed to `SetAbsAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `cycle` The value of the `cycle` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.24 OSError_CancelAlarm_AlarmID

Synopsis

```
AlarmType OSError_CancelAlarm_AlarmID(void)
```

Description

The function returns the `AlarmID` parameter passed to `CancelAlarm`. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Alarm ID` The value of the `AlarmID` parameter.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.25 OSError_IncrementCounter_AlarmID

Synopsis

```
AlarmType OSError_IncrementCounter_AlarmID(void)
```

Description

The function returns the `AlarmID` of the Alarm notification triggered by `IncrementCounter` that raised the error. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Alarm ID` The value of the `AlarmID` of the Alarm notification that triggered the error.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.26 OSError_IncrementCounter_TaskID

Synopsis

```
TaskType OSError_IncrementCounter_TaskID(void)
```

Description

The function returns the `TaskID` related to the Alarm notification triggered by `IncrementCounter` that raised the error. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Task ID` The value of the `TaskID` of the Alarm notification that triggered the error.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.27 OSError_IncrementCounter_Mask

Synopsis

```
EventMaskType OSError_IncrementCounter_Mask(void)
```

Description

The function returns the Event `Mask` related to the Alarm notification triggered by `IncrementCounter` that raised the error. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Mask` The value of the Event Mask of the Alarm notification that triggered the error.

Conformance

ECC1, ECC2

2.15.28 OSError_IncrementCounter_action

Synopsis

```
EE_TYPENOTIFY OSError_IncrementCounter_action(void)
```

Description

The function returns the `action` related to the Alarm notification triggered by `IncrementCounter` that raised the error. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `action` The value of the action of the Alarm notification that triggered the error. Possible values of this parameter are: `EE_ALARM_ACTION_TASK` for task notifications, `EE_ALARM_ACTION_CALLBACK` if the notification is an alarm callback, and `EE_ALARM_ACTION_EVENT` (only available with ECC1 and ECC2), if the notification is an event set.

Conformance

BCC1, BCC2, ECC1, ECC2

2.15.29 OSError_StartOS_Model

Synopsis

```
AppModeType OSError_StartOS_Mode(void)
```

Description

The function returns the `Model` related to the StartOS Call. The function must be used inside `ErrorHook` after having discovered using `OSErrorGetServiceId` that the error was caused by that function.

Return Values

- `Mode` The Mode parameter that was passed to the `StartOS` primitive.

Conformance

BCC1, BCC2, ECC1, ECC2

2.16 Interrupt service routines

Erika Enterprise supports both ISR Category 1 (which are ISR not directly handled by the operating system) and ISR Category 2 (which are handled by the operating system, and that can call OS primitives inside their handlers).

The following restriction applies: all interrupts of Category 1 must have a higher or equal hardware priority compared with interrupts of Category 2. This limitation has been introduced to avoid various rescheduling problems appearing when a ISR2 interrupts a lower priority ISR1.

This is the only limitation common to all Erika Enterprise porting, so you should check specific architecture manuals and/or Erika Enterprise wiki pages if more limitations regarding ISR priority levels has been introduced.

3 ORTI and Lauterbach Trace32 support

Erika Enterprise provides direct support for application debugging using the Lauterbach Trace32 debugger [1].

The provided support includes:

- Automatic generation of Trace32 scripts to simplify the load and debugging of Erika Enterprise applications.
- Automatic generation of batch scripts for single and multicore Nios II designs. An instance of Trace32 will be started and configurated for each core in the design.
- Automatic generation of the ORTI files to enable kernel awareness with Lauterbach Trace32. In this way, all the object declared in the system (Tasks, Resources, Alarms, Stacks, Context switches, ...) can be evaluated from the Trace32 debugger.

This chapter describes the impact of the ORTI usage in Erika Enterprise applications with respect to the impact to the code footprint of the ORTI code, and with respect to the stack usage statistics performed by Lauterbach Trace32. More information on how to generate the Lauterbach Trace32 scripts, and the ORTI files can be found in the RT-Druid Reference Manual.

3.1 ORTI and Erika Enterprise footprint

The ORTI support provided by Lauterbach Trace32 is a nice feature that allow developers to track in a graphical way the kernel objects like tasks, resources, alarms, and stacks. When using ORTI, Erika Enterprise is configured appropriately by RT-Druid by adding a set of parameters inside the makefiles. As a result, the kernel records a set of additional information which are then referred inside the ORTI file. For that reason, for a maximum reduction of the Erika Enterprise footprint, the ORTI options should be disabled.

3.2 ORTI and stack usage statistics

Another interesting feature of the ORTI support for Lauterbach Trace32 is the possibility of tracking the stack usage in the system. To use the feature, the user should remind to fill up all the stack space with the value `0xa5a5a5a5`. That can be usually done by putting a call to `EE_trace32_stack_init` as the first instruction into `main` or into `alt_main`.

3.2.1 EE_trace32_stack_init

Synopsis

```
void EE_trace32_stack_init(void);
```

Description

This inline function can be used on the Altera Nios II platform to fill up the memory space from `__alt_stack_limit` to the current stack pointer with the pattern `0xa5a5a5a5`.

The memory region from `__alt_stack_limit` to `0xa5a5a5a5` is the memory region that is typically allocated as stack space by the Altera System Library.

This function should be called among the first functions in the system, before any call to `malloc`, because the heap region is typically allocated at the end of the available stack space. Failing to do that may result in overwriting the data structures allocated with malloc, with potential unexpected results and application crashes.

This function is only useful to get an estimation of the used stack space, and should be used in conjunction with the ORTI stack informations produced for Lauterbach Trace32. If this function is not called, it is likely that Lauterbach Trace32 will return a stack utilization of 100% for all the stacks defined in the system.

Conformance

BCC1, BCC2, ECC1, ECC2

4 History

Version	Comment
1.2.x	These are versions for Nios II 5.0 and 5.1.
1.3.0	Updated text, corrected typos. Removed Altera Nios II features and the related multicore informations which are now in a separate document.
1.3.1	Added counting semaphores. StartOS always returns.
1.4.0	Re-organized and extended content, corrected typos.
1.4.2	Added OIL section; fixed typos; added new versioning mechanism.
1.4.3	Fixed Typos.
1.4.4	Fixed Typos. Erika Enterprise Basic renamed to Erika Enterprise.

Bibliography

[1] Lauterbach GMBH. The Lauterbach Trace32 Debugger for Nios II. http://www.lauterbach.com, 2005.

Index