

TASKING[®]

***TASKING POSIX
Implementation***

MA000-700 (v1.3) July 29, 2015

Copyright © 2015 Altium BV.

All rights reserved. You are permitted to print this document provided that (1) the use of such is for personal use only and will not be copied or posted on any network computer or broadcast in any media, and (2) no modifications of the document is made. Unauthorized duplication, in whole or part, of this document by any means, mechanical or electronic, including translation into another language, except for brief excerpts in published reviews, is prohibited without the express written permission of Altium BV. Unauthorized duplication of this work may also be prohibited by local statute. Violators may be subject to both criminal and civil penalties, including fines and/or imprisonment. Altium, TASKING, and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed.

Table of Contents

Manual Purpose and Structure	v
1. Introduction to POSIX	1
1.1. Real-time Systems	1
1.2. Real-time Operating System	1
1.2.1. Services of the RTOS	2
1.2.2. Real-time Response of the RTOS	3
1.2.3. Sharing Memory and Other Resources	3
1.3. What is POSIX	4
1.4. POSIX Real-time Profiles	4
1.5. TASKING's Real-time POSIX Implementation	5
2. System Description	7
2.1. System Overview	7
2.2. Function sysconf()	8
2.3. Units of Functionality	10
2.3.1. POSIX_THREADS_BASE	10
2.3.2. XSI_THREAD_EXT	11
2.3.3. XSI_THREAD_MUTEX_EXT	12
2.3.4. POSIX_SIGNALS	12
2.3.5. POSIX_SINGLE_PROCESS	13
2.3.6. POSIX_C_LANG_JUMP	14
2.3.7. POSIX_C_LANG_SUPPORT	14
2.3.8. POSIX_DEVICE_IO	15
2.3.9. POSIX_FILE_LOCKING	15
2.4. Option Requirements	16
2.4.1. _POSIX_CLOCK_SELECTION	16
2.4.2. _POSIX_FSYNC	16
2.4.3. _POSIX_MEMLOCK	16
2.4.4. _POSIX_MEMLOCK_RANGE	17
2.4.5. _POSIX_MONOTONIC_CLOCK	17
2.4.6. _POSIX_NO_TRUNC	17
2.4.7. _POSIX_REALTIME_SIGNALS	17
2.4.8. _POSIX_SEMAPHORES	18
2.4.9. _POSIX_SHARED_MEMORY_OBJECTS	19
2.4.10. _POSIX_SYNCHRONIZED_IO	19
2.4.11. _POSIX_THREAD_ATTR_STACKADDR	20
2.4.12. _POSIX_THREAD_ATTR_STACKSIZE	20
2.4.13. _POSIX_THREAD_CPUTIME	20
2.4.14. _POSIX_THREAD_PRIO_INHERIT	21
2.4.15. _POSIX_THREAD_PRIO_PROTECT	21
2.4.16. _POSIX_THREAD_PRIORITY_SCHEDULING	21
2.4.17. _POSIX_THREAD_SPORADIC_SERVER	22
2.4.18. _POSIX_TIMEOUTS	23
2.4.19. _POSIX_TIMERS	23
2.4.20. _POSIX_MESSAGE_PASSING	24
2.4.21. POSIX.26	24
3. Interrupt Control Extension	25
3.1. Introduction	25
3.2. Definitions	25

TASKING POSIX Implementation

3.2.1. Interrupt Lines and Interrupt Controllers	25
3.2.2. Exceptions	26
3.2.3. Interrupts	26
3.2.4. Interrupt Service Routines	27
3.3. Interrupt Framework	27
3.3.1. Interrupt Handlers	30
3.3.2. Interrupt Control Interfaces	32
4. Build and Configure	51
4.1. How to Build a Multi-threading Application	51
4.2. Multi-threading Applications	51
4.3. How to Configure a POSIX Application	52
4.4. How to Debug a POSIX Application	53

Manual Purpose and Structure

Manual Purpose

This manual aims to provide you with the necessary information to build real-time applications using the POSIX Real Time Operating System delivered with the toolset.

After reading the document, you should:

- know how POSIX is implemented by Altium,
- understand the benefits of using the POSIX,
- know how to build real-time POSIX applications,
- be able to customize POSIX settings in the Software Platform Builder to your project needs,
- be familiar with the most relevant POSIX concepts.

This manual assumes that the relevant POSIX standards are at your disposition.

Manual Structure

Chapter 1, *Introduction to POSIX*

This chapter provides an introduction to the TASKING POSIX real-time operating system. It explains what POSIX is and describes four real-time application environment profiles.

Chapter 2, *System Description*

This chapter describes the POSIX functionality supported by the implementation and includes the complete list of supported services and option requirements. Check the standard for a full description of each listed service.

Chapter 3, *Interrupt Control Extension*

This chapter describes the TASKING extension to the POSIX standards to include Interrupt Control services. In the first part of the chapter, the relation between threads, interrupt handlers, peripherals, interrupt controllers, and interrupts is discussed. The second part of this chapter describes the related interfaces in detail.

Chapter 4, *Build and Configure*

This chapter describes how you can use the Software Platform Builder to build and configure a POSIX application.

Chapter 1. Introduction to POSIX

This chapter provides an introduction to the Portable Operating System Interface POSIX[®]. It explains what POSIX is and describes its real-time application environment profiles. This POSIX implementation complies to the profile for minimal real-time systems.

1.1. Real-time Systems

A real-time system is used when there are rigid timing requirements on the operations of a processor to perform certain tasks. Real-time applications perform an action or give an answer to an external event in a timely and predictable manner. They cover a wide range of tasks with different time dependencies.

The timing requirements of actions usually differ between real-time applications; what may be fast for one application may be slow or late for another. In all cases, there should be well-defined time requirements.

The concept of *predictability* for real-time applications generally means that a task or set of tasks must always be completed within a predetermined amount of time. Depending on the situation, an unpredictable real-time application can result in loss of data or loss of deadlines.

There are two flavors of real-time systems:

- A *hard real-time* system must guarantee that critical tasks complete on time. Processing must be done within the defined constraints or the system will fail.
- A *soft real-time* system is less restrictive. In a soft real-time system, failure to produce the correct response at the correct time is also undesirable but not fatal.

Many real-time applications require high I/O throughput while still guaranteeing a fast response time to asynchronous external events. The ability to schedule tasks rapidly and implement secure communication mechanisms among multiple tasks becomes crucial.

Real-time applications are usually characterized by a blend of requirements. Some parts of the application may consist of hard, critical tasks which must meet their deadlines. In reality, most applications consist of tasks with both hard and soft real-time constraints. The key to a successful real-time application is your ability to accurately define application requirements at every point in the program.

1.2. Real-time Operating System

As explained, most applications consist of tasks with both hard and soft real-time constraints. If these tasks are single purposed, you could implement them as semi-independent program segments. Still you would need to embed the processor allocation logic inside the application tasks. Implementations of this kind typically take the form of a control loop that continually checks for tasks to execute. Such techniques suffer from numerous problems and do not represent a solution for regular real-time applications. Besides, they complicate the maintenance and reusability of the software.

A Real Time Operating System (RTOS) is a dedicated operating system fully designed to overcome the time constraints of a real-time system. An RTOS, like any other operating system, provides an environment

TASKING POSIX Implementation

in which you can execute programs in a convenient and structured manner, but without the risk of failing the real-time constraints.

In general, the benefits of using an RTOS are:

- An RTOS eliminates the need for processor allocation in the application software.
- Modifications, or additions of completely new tasks can be made in the application software without affecting critical system response requirements.
- Besides managing task execution, most real-time operating systems also provide facilities that include task communication, task synchronization, timers, memory management etc.
- An RTOS hides the underlying hardware specific concerns to the user offering a run-time environment that is completely independent of the target processor.
- Easy migration to other targets (provided that the RTOS vendor offers support for these other processor families).

1.2.1. Services of the RTOS

An RTOS provides a number of services which can be divided into several categories.

Task Management

The most basic category of RTOS services is task management. This set of services allows application software developers to design their software as a number of separate chunks of software; each handling a distinct topic, a distinct goal, and perhaps its own real-time deadline. Each separate chunk of software is called a 'task' ('thread' in POSIX). Interfaces in this category include the ability to launch tasks and assign priorities to them. The main RTOS service in this category is the scheduling of tasks as the embedded system is in operation. The Task Scheduler controls the execution of application software tasks and makes them run in a very timely and responsive manner.

Inter-task Communication and Synchronization

The second category of RTOS interfaces is inter-task communication and synchronization. These interfaces make it possible for tasks to pass information from one to another while eliminating the danger of damaging that information. These interfaces also let tasks coordinate, so they can cooperate with one another productively. Without the help of these RTOS interfaces, tasks might well communicate corrupted information or otherwise interfere with each other.

Timer Services

Since many embedded systems have stringent timing requirements, most RTOSes also provide some basic timer services, such as task delays and time-outs.

Interrupt Control and Processing

The fourth category of RTOS interfaces is interrupt control and processing. These are interfaces to associate interrupt service routines to hardware interrupts and/or to (un)mask specific interrupts among other things.

Dynamic Memory Allocation

Many (not all) RTOSes also provide dynamic memory allocation services. This category of interfaces allows tasks to 'borrow' chunks of RAM memory for temporary use in application software. Often these chunks of memory are then passed from task to task, as a means of quickly communicating large amounts of data between tasks. Some very small RTOSes that are intended for tightly memory limited environments, do not offer dynamic memory allocation services.

Device I/O

Many (but not all) RTOSes also provide a device I/O category of services. These services, if available, provide a uniform framework for organizing and accessing the many hardware device drivers that are typical in an embedded system.

1.2.2. Real-time Response of the RTOS

The responsiveness of the operating system to asynchronous events is a critical element of real-time systems. Real-time operating systems must be capable of meeting the demands of hard real-time tasks with tight deadlines. To do this, the operating system's reaction time must be short and the scheduling algorithm must be simple and efficient.

Real-time operating systems must be 'preemptive'. A preemptive RTOS guarantees that a higher priority task can interrupt a lower priority task.

Of course, there are states when this is simply not allowed to happen. For example, when handling interrupts while the RTOS is executing the scheduler itself or while the RTOS is accessing its internal critical sections. These states are called 'system locks'. In all other cases, preemption should be allowed. Obviously, whenever the system exits from one of these system locks, a test should determine whether the current task should be preempted by a task with a higher priority. Outside a system lock, preemption happens automatically as part of the interrupt servicing that causes a higher-priority task to become runnable (also called 'event-driven preemption').

As opposed to desktop operating systems where application code runs in user mode and kernel code runs in kernel (privileged) mode, hard real-time tasks always run in kernel mode. This way, hard real-time tasks can always intercept interrupts (event-driven preemption) and they can invoke system calls very fast because it is unnecessary to switch between kernel mode and user mode by using costly software interrupts or context switches. A system call for hard real-time tasks is no longer a software trap but a function call.

1.2.3. Sharing Memory and Other Resources

Hard real-time tasks have privileged access to the entire memory space with no address isolation or memory protection offered. In hard real-time systems, all the application code is supposed to be hard real-time, which effectively means that all tasks and the operating system share the same physical address space. Usual concepts, such as processes (UNIX) or memory management unit (MMU) mechanisms, simply do not exist in this context. Or, equivalently, we could say that in hard real-time systems, all tasks share the state information of a single process, and all tasks share memory and other resources directly. Context switching between tasks in a hard real-time system is much faster than context switching between processes.

TASKING POSIX Implementation

Thus, a real-time task could easily overwrite the address space of another task and/or the operating system. During the development phase, always be aware of the problem of data corruption (thread stack overflows or wrong pointer-based memory accesses). These types of programming errors are typically extremely difficult to detect and will normally result in critical failures once the full system has been deployed in the field.

1.3. What is POSIX

Although originated to refer to the original IEEE Std 1003.1-1988, the name POSIX more correctly refers to the family of related standards: IEEE Std 1003.*n* (where *n* is a number). POSIX is developed by the Portable Applications Standard Committee (PASC) of the IEEE Computer Society.

POSIX is based on UNIX, a well-established technology dating back to the early 1970s, and defines a standard way for an application to interface to the operating system.

Thus, POSIX is a set of books specifying APIs, it is neither a piece of code nor an operating system. The POSIX standards do not specify how RTOS services must be implemented, just their semantics. Implementers can choose their implementation as long as they follow the specification of the interface.

Some of these standards have already been approved, while others are currently being developed. The standards can be grouped in three categories:

- **Base standards.** These standards define system interfaces related to different aspects of the operating system. The standard specifies the syntax and semantics of system interfaces so that application programs can directly invoke the operating system services.

POSIX.1 is the System Interface, the basic reference standard. The latest version of POSIX.1 is the IEEE Std 1003.1, 2013 Edition.

- **Language bindings.** These standards provide the actual interfaces for different programming languages: C, Ada, Fortran 77, and Fortran 90, ...
- **Operating system environment.** These standards include a guide to the POSIX environment and application profiles. An application profile is a list of POSIX standards that are required for a certain application environment, along with the options and parameters of these standards whose support is required for that application environment. Application profiles are a very important means of achieving a small number of well-defined types of operating system implementations appropriate for particular application environments.

POSIX.13 is the standard that describes the Real-time Application Environment Profiles. The *TASKING POSIX Implementation* is based on POSIX.13 IEEE Std 1003.13-2003 edition.

1.4. POSIX Real-time Profiles

The complete set of POSIX services is useful for large applications, but the set is considered to be too large for most embedded systems. Embedded systems usually have tight memory requirements, may not have memory management capabilities, and may not have a secondary memory for implementing the UNIX file system.

For these reasons the POSIX standard recognizes the need for the creation of subsets of the operating system services. These should be standard subsets that allow portability of applications from one implementation to another.

The IEEE 1003.13 (POSIX.13, 2003) standard describes four real-time Application Environment Profiles (AEPs) and their minimum hardware requirements:

- Minimal Real-time System AEP (PSE51)
- Real-time Controller System AEP (PSE52)
- Dedicated Real-time System AEP (PSE53)
- Multi-Purpose Real-time System AEP (PSE54)

Each profile is an extension of the previous profile. For details see the *IEEE Std 1003.13™-2003 IEEE Standard for Information Technology - Standardized Application Environment Profile (AEP) - POSIX® Realtime and Embedded Application Support*.

1.5. TASKING's Real-time POSIX Implementation

The TASKING RTOS kernel implements most of the POSIX Minimal Realtime System AEP (PSE51) as described in the *IEEE Std 1003.13-2003* edition. This means that the implementation supports the required interfaces referenced in the appropriate standardized profile. These interfaces support the functional behavior and any additional constraints or options as described in the appropriate base standard.

The TASKING implementation supports almost all referenced interfaces. [Chapter 2, System Description](#) describes the specific extent of conformance (with rationale) and includes the complete list of interfaces supported by the implementation. An application is said to be strictly conforming to a given POSIX.13 profile if the application requires only the facilities required in that profile. Such an application must accept any behavior described in the profile as unspecified or implementation defined. And for symbolic constants, the application must accept any value in the range permitted by the profile. The RTOS implementation makes sure that strictly PSE51 conforming applications can be run in this implementation with predictable behavior (provided that they limit their use to supported interfaces).

The TASKING RTOS implementation also provides an extension to the standard, which is described in this manual. This extension deals with interrupt controls (see [Chapter 3, Interrupt Control Extension](#)). Obviously, applications that use this extension are strictly spoken not conforming to the standard.

Extended POSIX services always start with `posix_type_t`.

Extended POSIX types always start with an underscore character `_`.

Chapter 2. System Description

This chapter describes the POSIX functionality supported by the implementation and includes the complete list of supported "Units of Functionality" and "Option Requirements". Check the standard for a full description of each the listed functions.

2.1. System Overview

Units of Functionality

The system profile is defined in terms of component profiles that specify Units of Functionality that can be combined to realize the application platform. A Unit of Functionality is a defined set of services which can be implemented. If implemented, the standard prescribes that all services in the Unit must be implemented.

A Minimal Realtime System Profile implementation must support the following Units of Functionality as defined in IEEE Std. 1003.1 (also referred to as POSIX.1).

POSIX.1 Units of Functionality Requirements
POSIX_C_LANG_JUMP
POSIX_C_LANG_SUPPORT
POSIX_DEVICE_IO
POSIX_FILE_LOCKING
POSIX_SIGNALS
POSIX_SINGLE_PROCESS
POSIX_THREADS_BASE
XSI_THREAD_MUTEX_EXT
XSI_THREADS_EXT

Section 2.3, *Units of Functionality* describes the Units as implemented in the TASKING POSIX. This section is comprehensible without reading the standard, although it does not replace the standard. Therefore, it is recommended to read the standard as well.

Option Requirements

An implementation supporting the Minimal Realtime System Profile must support the POSIX.1 Option Requirements which are defined in the standard. Options Requirements are used for further sub-profiling within the units of functionality: they further define the functional behavior of the system service (normally adding extra functionality). Depending on the profile to which the POSIX implementation complies, parameters and/or the precise functionality of certain services may differ.

The following table shows the option requirements that are implemented in the TASKING POSIX.

TASKING POSIX Implementation

POSIX.1 Option Requirements
<code>_POSIX_CLOCK_SELECTION</code>
<code>_POSIX_FSYNC</code>
<code>_POSIX_MEMLOCK</code>
<code>_POSIX_MEMLOCK_RANGE</code>
<code>_POSIX_MONOTONIC_CLOCK</code>
<code>_POSIX_REALTIME_SIGNALS</code>
<code>_POSIX_SEMAPHORES</code>
<code>_POSIX_SHARED_MEMORY_OBJECTS</code>
<code>_POSIX_SYNCHRONIZED_IO</code>
<code>_POSIX_THREAD_ATTR_STACKADDR</code>
<code>_POSIX_THREAD_ATTR_STACKSIZE</code>
<code>_POSIX_THREAD_CPUTIME</code>
<code>_POSIX_THREAD_PRIO_INHERIT</code>
<code>_POSIX_THREAD_PRIO_PROTECT</code>
<code>_POSIX_THREAD_PRIORITY_SCHEDULING</code>
<code>_POSIX_THREAD_SPORADIC_SERVER</code>
<code>_POSIX_TIMEOUTS</code>
<code>_POSIX_TIMERS</code>
<code>_POSIX_NO_TRUNC</code> ¹

¹ Since a file system is not a part of the minimal realtime system profile, the `_POSIX_NO_TRUNC` requirement is applied to the names of devices and shared memory objects.

Section 2.4, *Option Requirements* describes the option requirements as implemented in the TASKING POSIX. This section is comprehensible without reading the standard, although it does not replace the standard. Therefore, it is recommended to read the standard as well. The labels in the table correspond to labels in the standard.

2.2. Function `sysconf()`

Before describing the Units of Functionality and the Option Requirements that are implemented in the TASKING POSIX system, we introduce the function `sysconf()`.

With the function `sysconf()` you can determine the current value of a configurable system limit or option requirement. This function is especially useful if you need to write portable code which possibly will run on operating systems that may not comply to the POSIX.1 standard. With this function you can query the system if it supports a function, or query the system about certain limits.

The function has the following prototype:

```
#include <unistd.h>
```

```
long sysconf( int name );
```

The *name* argument represents the system variable to be queried. `limits.h` and `unistd.h` contain a minimal set of system variables that can be returned by `sysconf()`. `unistd.h` also defines the symbolic constants (`_SC_label`) which are the corresponding values that you can use for *name*.

Examples

The following example shows how you can test implementation options, for instance, support for POSIX timers. The first test is only performed at compile time but may not work because the standard does not require the presence of these macros. (The TASKING POSIX does support them). Therefore, in case the first test fails, the second test is performed at run-time. This test always works because the `sysconf()` function is prescribed by the standard.

```
int have_timers (void)
{
    #if ( defined(_POSIX_TIMERS) && (_POSIX_TIMERS == 200112L) )
        return 1;
    #else
        if ( sysconf(_SC_TIMERS) == 200112L )
            return 1;
        else
            return 0;
    #endif
}
```

In the TASKING implementation the values returned by `sysconf()` for options are either -1 (functionality not present) or 200112L (functionality present, 200112L corresponds to its version).

The following example shows how you can test for numerical limits, for instance, the maximum number of POSIX timers. Again, if the macro is not supported (because the standard does not prescribe it), the `sysconf()` function is added to your code:

```
int get_timer_max()
{
    #ifdef TIMER_MAX
        return TIMER_MAX;
    #else
        return sysconf(_SC_TIMER_MAX);
    #endif
}
```

TASKING POSIX Implementation

The include file `limits.h` contains values as prescribed by the standard. However, if you know your application will run on the TASKING POSIX implementation, you can overrule these values to optimize your application.

In the Software Platform Builder, you can set your own values; these are stored in a separate file which is included in `limits.h`.

The values in `limits.h` and `unistd.h` should never change at run-time.

2.3. Units of Functionality

This section describes the Units of Functionality (fixed sets of interfaces) which are implemented in the TASKING POSIX. Please refer to the standard for a full description of each listed interface.

2.3.1. POSIX_THREADS_BASE

The basic assumption in this profile is that the system consists of a single (implicit) process with multiple threads. Therefore, the standard requires all basic thread services, except those related to multiple processes.

The POSIX.1 process model (PSE53 and PSE54) may not be adequate for systems that require high efficiency, because of the following reasons:

- processes have high context switch times
- substantial time is needed to create or destroy processes
- special hardware is required (memory management units) to provide each process with an independent address space
- the model is not adequate for shared memory multi-processors.

In most of the commercially available real-time operating systems for small embedded systems the concurrence model is based on tasks that share the same address space and have an associated state that is small, compared to POSIX processes.

The POSIX.4a standard (thread extensions) defines interfaces to support multiple concurrent activities, called threads, inside each POSIX process. It defines functions to:

- create a thread
- wait for thread termination
- terminate a thread in a normal way
- detach a thread; that is, indicate to the implementation that the storage associated with a thread may be reclaimed when the thread terminates
- create a particular thread only if it has not been created already

Other functions allow to handle thread identifiers. Also, functions are defined to manage thread creation attributes such as the size of the thread's stack, to define whether the thread storage is detachable from creation time, and more.

The scheduling policies defined for threads are the same as those defined for processes in the POSIX.4 standard (priority preemptive, with either FIFO or round robin treatment of equal priority threads). The implementation follows a Global Scheduling approach: all threads have global contention scope and every thread is scheduled against all other threads in the system. Services include setting scheduling policies and parameters, and yielding the processor to other equal-priority threads.

Two synchronization primitives are defined for threads: mutexes and condition variables. Mutexes are used to synchronize threads for mutually exclusive access to shared resources (for example, two threads are trying to write to the same device), while condition variables are used to signal and wait for events among threads. Waiting for a condition variable to be signaled can be specified with a time-out.

POSIX_THREADS_BASE	
pthread_atfork() ¹⁾	pthread_exit()
pthread_attr_destroy()	pthread_getspecific()
pthread_attr_getdetachstate()	pthread_join()
pthread_attr_getschedparam()	pthread_key_create()
pthread_attr_init()	pthread_key_delete()
pthread_attr_setdetachstate()	pthread_kill()
pthread_attr_setschedparam()	pthread_mutex_destroy()
pthread_cancel()	pthread_mutex_init()
pthread_cleanup_pop()	pthread_mutex_lock()
pthread_cleanup_push()	pthread_mutex_trylock()
pthread_cond_broadcast()	pthread_mutex_unlock()
pthread_cond_destroy()	pthread_mutexattr_destroy()
pthread_cond_init()	pthread_mutexattr_init()
pthread_cond_signal()	pthread_once()
pthread_cond_timedwait()	pthread_self()
pthread_cond_wait()	pthread_setcancelstate()
pthread_condattr_destroy()	pthread_setcanceltype()
pthread_condattr_init()	pthread_setspecific()
pthread_create()	pthread_sigmask()
pthread_detach()	pthread_testcancel()
pthread_equal()	

¹⁾ pthread_atfork() is not implemented

2.3.2. XSI_THREAD_EXT

The XSI_THREADS_EXT Unit of Functionality is required because it provides functions to control a thread's stack. This is considered useful for any real-time application.

The following services are supported:

TASKING POSIX Implementation

XSI_THREAD_EXT	
<code>pthread_attr_getguardsize()</code>	<code>pthread_attr_setstack()</code>
<code>pthread_attr_getstack()</code>	<code>pthread_getconcurrency()</code>
<code>pthread_attr_setguardsize()</code>	<code>pthread_setconcurrency()</code>

The `pthread_getconcurrency()/pthread_setconcurrency()` interfaces are present just for source compatibility. They have no effect whatsoever in the behavior of the system.

Also the `pthread_attr_getguardsize()/pthread_attr_setguardsize()` interfaces are present just for source compatibility. They will have no effect whatsoever in the behavior of the system. The reasoning behind this (especially in real-time systems) is:

- If the Stack Attribute (location or size) is set, the guard size attribute is effectively ignored in all cases. The rationale behind this is, that if an application is setting the stack location or size it should be easily able to allocate its own "guard buffer" area. This is what normally happens in real-time threads where application stack buffers are normally allocated at compile time to speed up performance.
- Use of guard spaces, while potentially making a threaded application more robust, is also a drain on system resources.
- Calculations for the guard buffer size should correlate to, at a minimum, the size of the largest data structure which will ever exist on the stack.

2.3.3. XSI_THREAD_MUTEX_EXT

The `XSI_THREAD_MUTEX_EXT` Unit of Functionality is required because it has options for controlling the behavior of mutexes under erroneous application use. This capability is interesting for any real-time application, including those targeted at small embedded systems.

The following services are supported:

XSI_THREAD_MUTEX_EXT	
<code>pthread_mutexattr_gettype()</code>	<code>pthread_mutexattr_settype()</code>

2.3.4. POSIX_SIGNALS

Signal services are a basic mechanism within POSIX-based systems and are required for error and event handling.

Real-time systems typically have several logically concurrent software elements executing. Each such entity must respond to several cyclic and/or a-cyclic stimuli, often in a time-critical manner. Although purely synchronous models can supply such functionality via the use of additional processes or threads, the current real-time practice for asynchronous notification for events such as, message arrival, and hardware interrupt can generally be expected to offer higher performance and lower latency.

This implementation understands that POSIX signals and real-time are not always the perfect choice. The usage of POSIX signals is somehow restricted here in comparison with desktop systems. In desktop UNIX systems:

- An exception, such as a divide-by-zero or segmentation violation, may be detected by hardware, causing the UNIX kernel to generate an appropriate signal (such as SIGFPE or SIGSEGV) and send it to the current process. However, in Minimal Realtime Systems everything (all threads) form the current process. The system cannot recover from a divide-by-zero operation (there are no other processes to run). Therefore you need to debug the exception until no divide-by-zero operations occur. The kernel simply offers an interface so that applications can install exception handlers to deal with these situations.
- A user may press certain terminal keys, such as Ctrl-C, to control the behavior of the running program. This causes the terminal driver program to send a signal (such as SIGINT) to the user-level process in which the program is running. In Minimal Realtime Systems where interrupt latency is important, this kind of behavior should be best avoided. The RTOS kernel implements faster and dedicated built-in synchronization mechanisms between I/O threads and device drivers with less overhead.
- A process may request a signal from the operating system when a timer expires or a message arrives at an empty message queue. The RTOS kernel prefers mutex and conditional variables over signals for its internal synchronization.

However, and for the same reason that user-level processes may send signals to another processes in UNIX, in Minimal Realtime Systems, threads may send signals to another threads.

The following services are supported:

POSIX_SIGNALS	
abort()	sigemptyset()
alarm()	sigfillset()
kill()	sigismember()
pause()	signal()
raise()	sigpending()
sigaction()	sigprocmask()
sigaddset()	sigsuspend()
sigdelset()	sigwait()

2.3.5. POSIX_SINGLE_PROCESS

The POSIX_SINGLE_PROCESS Unit of Functionality contains services for single process applications.

The following services are supported:

POSIX_SINGLE_PROCESS	
confstr()	setenv()
environ	sysconf()
errno	uname()
getenv()	unsetenv()

2.3.6. POSIX_C_LANG_JUMP

The POSIX_C_LANG_JUMP Unit of Functionality contains the C language jump functions.

The following services are supported:

POSIX_C_LANG_JUMP	
longjmp()	setjmp()

2.3.7. POSIX_C_LANG_SUPPORT

The POSIX_C_LANG_SUPPORT Unit of Functionality contains the general ISO C Library.

POSIX_C_LANG_SUPPORT			
abs()	imaxabs()	mktime()	strtod()
asctime()	imaxdiv()	qsort()	strtof()
asctime_r() ¹⁾	isalnum()	rand()	strtoimax()
atof()	isalpha()	rand_r() ¹⁾	strtok()
atoi()	isblank()	realloc()	strtok_r() ¹⁾
atol()	iscntrl()	setlocale()	strtol()
atoll()	isdigit()	snprintf()	strtold()
bsearch()	isgraph()	sprintf()	strtoll()
calloc()	islower()	strand()	strtoul()
ctime()	isprint()	sscanf()	strtoull()
ctime_r() ¹⁾	ispunct()	strcat()	strtoumax()
difftime()	isspace()	strchr()	strxfrm()
div()	isupper()	strcmp()	time()
feclearexcept() ³⁾	isxdigit()	strcoll()	tolower()
fegetenv() ³⁾	labs()	strcpy()	toupper()
fegetexceptflag() ³⁾	ldiv()	strcspn()	tzname() ²⁾
fegetround() ³⁾	llabs()	strerror()	tzset() ²⁾
fehldexcept() ³⁾	lldiv()	strerror_r() ¹⁾	va_arg()
feraiseexcept() ³⁾	localeconv()	strftime()	va_copy()
fesetenv() ³⁾	localtime()	strlen()	va_end()
fesetexceptflag() ³⁾	localtime_r() ¹⁾	strncat()	va_start()
fesetround() ³⁾	malloc()	strncmp()	vsnprintf()
fetestexcept() ³⁾	memchr()	strncpy()	vsprintf()
feupdateenv() ³⁾	memcmp()	strpbrk()	vsscanf()
free()	memcpy()	strrchr()	
gmtime()	memmove()	strspn()	
gmtime_r() ¹⁾	memset()	strstr()	

¹⁾ These thread safe versions are not implemented

²⁾ Time zone functionality is not implemented

³⁾ Functionality for `fenv.h` is not implemented

2.3.8. POSIX_DEVICE_IO

The following services are supported:

POSIX_DEVICE_IO	
clearerr()	gets()
close()	open()
fclose()	perror()
fdopen()	printf()
feof()	putc()
ferror()	putchar()
fflush()	puts()
fgetc()	read()
fgets()	scanf()
fileno()	setbuf()
fopen()	setvbuf()
fprintf()	stderr
fputc()	stdin
fputs()	stdout
fread()	ungetc()
freopen()	vfprintf()
fscanf()	vfscanf()
fwrite()	vprintf()
getc()	vscanf()
getchar()	write()

For an application that conforms to PSE51 the file descriptor is only used to specify the device and does not refer to any file system. In the TASKING POSIX implementation file system support is available.

2.3.9. POSIX_FILE_LOCKING

The File Locking option is required in the C language option to maintain a consistent and safe way of accessing `stdio` (`FILE*`) objects from threads, across the four real-time profiles.

POSIX_FILE_LOCKING ¹⁾	
ftrylockfile()	getchar_unlocked()
funlockfile()	putc_unlocked()
getc_unlocked()	putchar_unlocked()

¹⁾ This unit of functionality is not implemented.

2.4. Option Requirements

This section describes the Option Requirements (additional specification of how a service should work or what arguments it should take, depending on the chosen real-time profile) which are implemented in the TASKING POSIX. Please refer to the standard for a full description of each listed option requirement or service.

2.4.1. `_POSIX_CLOCK_SELECTION`

The Clock Selection option is required to enable choosing the clock on which sleep operations are performed and to have access to an absolute sleep operation, which is a common requirement in real-time applications with periodic timing requirements.

The following service allows sleep operations of absolute high-resolution.

<code>_POSIX_CLOCK_SELECTION</code>
<code>clock_nanosleep()</code> <code>pthread_condattr_getclock()</code> <code>pthread_condattr_setclock()</code>

2.4.2. `_POSIX_FSYNC`

The `_POSIX_FSYNC` option is not implemented. This option has one function:

<code>_POSIX_FSYNC</code>
<code>fsync()</code> ¹⁾

¹⁾ not implemented

2.4.3. `_POSIX_MEMLOCK`

For this implementation, system options have been chosen that allow an application to be upwards compatible without modifying application source code. For example, although the assumed hardware model implies a fixed address space without an MMU, the symbol `_POSIX_MEMLOCK` is still defined. This increases portability of the application code to higher level systems that do not necessarily have the same restrictions.

Process memory locking is inherent in systems following this profile because most PSE51 targets have no MMU and thus swapping is not supported; code and data stay in physical memory until explicitly removed. Nevertheless, memory locking APIs are required for upward portability to allow you to take code intended for a bare PSE51 target and test that code on a much larger and more capable platform, perhaps a PSE54, with minimal modification. In those targets without an MMU for virtual memory, the locking functions do nothing and always report success, while in the larger profiles there really is memory to be locked. In summary, by requiring this service in the PSE51 profile, it is possible to write portable application code that runs correctly in other profiles.

The following services are therefore supported for upwards compatibility:

_POSIX_MEMLOCK
<pre>mlockall() munlockall()</pre>

2.4.4. **_POSIX_MEMLOCK_RANGE**

The same applies as discussed in [Section 2.4.3, `_POSIX_MEMLOCK`](#)

The following services are supported for upwards compatibility:

_POSIX_MEMLOCK_RANGE
<pre>mlock() munlock()</pre>

2.4.5. **_POSIX_MONOTONIC_CLOCK**

The Monotonic Clock is required for real-time applications to ensure that deadlines and timing requirements are not affected by clock jumps.

The value of this clock cannot be changed explicitly; it only changes monotonically with the passage of time. Such a clock is interesting to prevent the effects that setting the system clock has on the application scheduling.

This POSIX.1 system interface option has no functions.

2.4.6. **_POSIX_NO_TRUNC**

Since a file system is not a part of the minimal realtime system profile, the `_POSIX_NO_TRUNC` requirement is applied to the names of devices and shared memory objects.

This POSIX.1 system interface option has no functions.

2.4.7. **_POSIX_REALTIME_SIGNALS**

The signal mechanism defined in POSIX.1 allows to notify events occurring in the system, but is not completely satisfactory for real-time applications. The signals are not queued, thus some events may be lost. Signals are not prioritized, and this implies longer response times for urgent events. Also, events of the same kind produce signals with the same number, which are indistinguishable.

Since many real-time systems are heavily based on the rapid exchange of events in the system, POSIX.4 has extended the signals interface to achieve the following features:

- Real-time signals are queued, so events are not lost.
- Pending real-time signals are dequeued in priority order, using the signal number as the priority. This allows designing applications with faster response times to urgent events.

TASKING POSIX Implementation

- Real-time signals contain an additional data field that may be used by the application to exchange data between the signal generator and the signal handler. For example, this data field may be used to identify the source of the signal.
- The range of signals available to the application is expanded.

The minimum number of real-time signals (RTSIG_MAX) that the implementation is required to support has been increased from 8 to 16. The rationale for this increase is that there are many applications that have more than 8 different kinds of events. Doubling the number of required real-time signals should have a minimum impact on the signal management overhead, while significantly increasing the number of event kinds that can be used by a strictly conforming application.

An application strictly conforming to PSE51 is considered erroneous if any signal results in abnormal termination of the process because this profile does not support multiple processes.

`pthread_kill()` allows applications to send a signal only to a specific thread and not to the whole process (not to be made available for other threads).

All functions of the `_POSIX_REALTIME_SIGNALS` option requirement are supported:

<code>_POSIX_REALTIME_SIGNALS</code>
<code>sigqueue()</code>
<code>sigtimedwait()</code>
<code>sigwaitinfo()</code>

`sigqueue()` is the real-time version of the traditionally used `kill()`.

2.4.8. `_POSIX_SEMAPHORES`

Semaphores are required in the PSE51 profile for synchronization between threads to maintain compatibility with past industry practice. However, the standard recommends not to use semaphores; POSIX semaphores do not have the mechanisms to avoid unbounded priority inversion when using them for mutually exclusive access to shared resources.

Priority inversion occurs when a high priority process has to wait for a lower priority process to complete an action. Using appropriate protocols, priority inversion can be bounded by only the duration of critical sections (that is, sections of code during which the process reserves a particular resource for exclusive use).

However, with conventional semaphores unbounded priority inversion may occur; this means that the delay experienced by high priority threads is not bounded by the duration of critical sections, but depends on the total execution time of lower priority threads. This situation can occur when a high priority thread is waiting for a low priority thread to release a semaphore that controls access to a shared resource, and the low priority thread is preempted by an intermediate priority thread. These long delays are usually unacceptable for threads with hard real-time requirements.

If appropriate protocols are used, the amount of priority inversion can be bounded by the duration of critical sections, which is usually very small. Mutexes with the appropriate priority inheritance or priority ceiling (also called priority protection) protocols can be used to avoid this unbounded priority inversion. That is why mutexes and condition variables are preferred in most current applications.

_POSIX_SEMAPHORES	
<code>sem_close()</code>	<code>sem_post()</code>
<code>sem_destroy()</code>	<code>sem_trywait()</code>
<code>sem_getvalue()</code>	<code>sem_wait()</code>
<code>sem_init()</code>	<code>sem_unlink()</code>
<code>sem_open()</code>	<code>sem_timedwait()</code> ¹⁾

¹⁾ also requires `_POSIX_TIMEOUTS`

2.4.9. **_POSIX_SHARED_MEMORY_OBJECTS**

Memory Mapped I/O may be implemented using the Shared Memory facility. An implementation is required to provide facilities for creating (shared) memory objects that represent ranges of physical memory that contain device control and status registers or buffers. These facilities encourage the development of portable applications.

Although the memory may be directly accessible, applications are advised to perform memory mapped I/O through shared memory objects, so that they can be ported to larger systems in which direct access to memory-mapped devices is forbidden to the application. Implementation of shared memory objects in a non-protected architecture where all memory is directly addressable, is straightforward.

_POSIX_SHARED_MEMORY_OBJECTS	
<code>mmap()</code> ^{1) 2)}	<code>shm_open()</code>
<code>munmap()</code> ²⁾	<code>shm_unlink()</code>
<code>posix_madvise()</code> ³⁾	

¹⁾ `mmap()` functionality is only available for file descriptors obtained using `shm_open()`

²⁾ available in combination with `_POSIX_MAPPED_FILES`

³⁾ not available due to no support for `_POSIX_ADVISORY`

2.4.10. **_POSIX_SYNCHRONIZED_IO**

Synchronized I/O is an interface to provide data integrity, so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices.

_POSIX_SYNCHRONIZED_IO
<code>fdatasync()</code> ¹⁾
<code>msync()</code> ^{1) 2)}

¹⁾ only available for file descriptors obtained using `shm_open()`

²⁾ available in combination with `_POSIX_MAPPED_FILES`

2.4.11. `_POSIX_THREAD_ATTR_STACKADDR`

Each new thread must have its own stack and all stacks of the different threads are in the same address space. The RTOS determines how the stacks are allocated. These services are provided so the application can interfere with the stack allocation. There are two fundamental reasons why an application would like to allocate the stacks of its threads (statically or in the heap): memory optimization and real-time performance.

In real-time applications, where the memory usage could easily become an issue, applications might want to allocate smaller stacks (than the RTOS's default) for some threads and larger stacks for others.

Besides, in most real-time applications, there is little room for surprise once the application has been debugged and has proven to work fine. The size of the stacks and their interrelation can be known even at compile time. Static allocation of stacks can improve the run-time performance significantly, this is done by locating stacks in memory with better access timing.

<code>_POSIX_THREAD_ATTR_STACKADDR</code>	
<code>pthread_attr_setstackaddr()</code>	<code>pthread_attr_getstack()</code> ¹⁾
<code>pthread_attr_getstackaddr()</code>	<code>pthread_attr_setstack()</code> ¹⁾

¹⁾ also requires `_POSIX_THREAD_ATTR_STACKSIZE`

2.4.12. `_POSIX_THREAD_ATTR_STACKSIZE`

For similar reasons as given in [Section 2.4.11, `_POSIX_THREAD_ATTR_STACKADDR`](#), the following interfaces are also supported:

<code>_POSIX_THREAD_ATTR_STACKSIZE</code>	
<code>pthread_attr_setstacksize()</code>	<code>pthread_attr_getstack()</code> ¹⁾
<code>pthread_attr_getstackSIZE()</code>	<code>pthread_attr_setstack()</code> ¹⁾

¹⁾ also requires `_POSIX_THREAD_ATTR_STACKADDR`

2.4.13. `_POSIX_THREAD_CPUTIME`

CPU Time clocks and timers are required as a means to detect and handle situations in which a thread overruns its assigned maximum execution time. Bounding the execution times of the different threads in the application increases predictability and reliability.

This is a very interesting feature for hard real-time systems, because the results of real-time schedulability analysis are only valid if the estimations of worst-case execution times are accurate. Execution-time timers enable us to validate this information on-line, and take corrective actions if an overrun occurs.

<code>_POSIX_THREAD_CPUTIME</code>
<code>pthread_getcpuclockid()</code>

To enable the `_POSIX_THREAD_CPUTIME` system interface option, enable option `CPUTIME` in the **Multi-threading Support** in the Software Platform Builder.

2.4.14. `_POSIX_THREAD_PRIO_INHERIT`

Mutex Priority Inheritance can lead to better handling of threads with different priorities. If a high-priority thread is waiting on a mutex which is held by a lower-priority thread, the latter continues to use its own priority although it blocks a thread with a higher priority. With the functionality of this option, you can define a mutex which automatically adjusts the priority of the thread holding the mutex based on the priority of the waiters.

<code>_POSIX_THREAD_PRIO_INHERIT</code>
<code>pthread_mutexattr_getprotocol()</code>
<code>pthread_mutexattr_setprotocol()</code>

2.4.15. `_POSIX_THREAD_PRIO_PROTECT`

Mutex Priority Ceiling can lead to better handling of threads with different priorities. If a high-priority thread is waiting on a mutex which is held by a lower-priority thread the latter continues to use its own priority although it blocks a thread with a higher priority. But if this option is defined the user can define a mutex which always increases the priority to a given level regardless of whether there are waiters or not.

<code>_POSIX_THREAD_PRIO_PROTECT</code>	
<code>pthread_mutex_getprioceiling()</code>	<code>pthread_mutexattr_getprotocol()</code>
<code>pthread_mutex_setprioceiling()</code>	<code>pthread_mutexattr_setprioceiling()</code>
<code>pthread_mutexattr_getprioceiling()</code>	<code>pthread_mutexattr_setprotocol()</code>

2.4.16. `_POSIX_THREAD_PRIORITY_SCHEDULING`

The base POSIX.1 standard defines a model of concurrent activities called processes, but does not specify any scheduling policy nor any concept of priority. For real-time applications it is necessary to specify some scheduling policy suitable for real-time in order to be portable.

POSIX.4 specifies three scheduling policies. Each process has a scheduling attribute that you can set to any of the three policies:

- `SCHED_FIFO`

This is a fixed-priority preemptive scheduling policy, in which processes with the same priority are treated in first-in-first-out (FIFO) order. At least 32 priority levels must be available for this policy.

- `SCHED_RR`

This policy is similar to `SCHED_FIFO`, but uses a time-sliced (round robin) method to schedule processes with the same priorities. It also has 32 priority levels.

TASKING POSIX Implementation

- SCHED_OTHER

This is an implementation-defined scheduling policy.

Fixed-priority scheduling is a popular scheduling policy for real-time systems. It is very simple, and high utilization levels can be achieved by using rate-monotonic or deadline monotonic priority assignments. With these scheduling policies and with the functions used to set and get the policies and priorities of each process, real-time applications can be scheduled in POSIX operating systems.

A common requirement of real-time systems is that they are able to run threads with real-time requirements together with threads with no real-time requirements. One common way of doing this is by running the real-time threads under the SCHED_FIFO scheduling policy, while the non-real-time threads run at a lower priority, under the round-robin policy (SCHED_RR) to fairly share the available portion of the processor among them.

POSIX.1 requires each policy to have a range of priorities of at least 32 distinct values, but does not impose any requirements on how these priority ranges relate to each other. It could happen that most or all of the SCHED_RR priorities were larger than the SCHED_FIFO priorities, thus making it impossible to mix real-time and non-real-time threads as required above.

To solve this problem in a portable way, this profile requires that there are at least 31 SCHED_RR priority levels below the maximum priority of SCHED_FIFO. In this way, a strictly conforming application can use the inclusive priority range [max_FIFO_prio, max_FIFO_prio-30] with SCHED_FIFO for real-time threads (with a total of 31 priority levels), and then use the priority value min(max_FIFO_prio-31, max_RR_prio) with the SCHED_RR policy, for the non real-time threads, with guarantee that the latter priority value is valid for the round-robin policy.

The implementation allows 64 priority levels [0,1, .. 63] where level 0 and level 63 are reserved for the kernel. Following the discussion above, you could populate levels within the range [1..31] only with SCHED_RR threads and levels within the range [32..62] only with SCHED_FIFO threads. This is however a system design decision: the kernel allows you to populate any level (apart from those kernel reserved) with both kind of threads.

_POSIX_THREAD_PRIORITY_SCHEDULING	
pthread_attr_getinheritsched()	pthread_getschedparam()
pthread_attr_getschedpolicy()	pthread_setschedparam()
pthread_attr_getscope()	pthread_setschedprio()
pthread_attr_setinheritsched()	sched_get_priority_max()
pthread_attr_setschedpolicy()	sched_get_priority_min()
pthread_attr_setscope()	

2.4.17. _POSIX_THREAD_SPORADIC_SERVER

The Sporadic Server Scheduling option enhances the support of applications with aperiodic timing requirements. This scheduling policy allows processing aperiodic activities with fast response times, while bounding their effect on lower priority threads, even in the presence of unbounded aperiodic requests. It is also useful for eliminating the negative effects that jitter has on the schedulability of lower priority threads. This interface option adds SCHED_SPORADIC scheduling policy.

To enable the `_POSIX_THREAD_SPORADIC_SERVER` system interface option, enable option `SPORADIC_SERVER` in the **Multi-threading Support** in the Software Platform Builder.

This option has no functions.

2.4.18. `_POSIX_TIMEOUTS`

Some operating system services defined in POSIX.1 and POSIX.4 can suspend the calling process for an indefinite period of time, until the necessary resources become available. In time-critical systems it is important to limit the maximum amount of time that a process can stay waiting for one of these services to complete. This allows to detect abnormal conditions, thus increasing the program robustness and allowing fault-tolerant implementations.

POSIX.4b defines new versions of some of the blocking services with built-in time-outs. These time-outs specify the maximum amount of time that the thread may be suspended while waiting for the service to complete.

The services chosen are those that are most likely going to be used in time-critical code, and did not already have the capability to:

1. wait for the arrival of a message to a message queue,
2. send a message to a queue,
3. wait for a mutex/condition to become unlocked.

<code>_POSIX_TIMEOUTS</code>	
<code>mq_timedreceive()</code>	<code>pthread_rwlock_timedrdlock()</code> ¹⁾
<code>mq_timedsend()</code>	<code>pthread_rwlock_timedwrlock()</code> ¹⁾
<code>sem_timedwait()</code>	<code>posix_trace_timedgetnext_event()</code> ¹⁾
<code>pthread_mutex_timedlock()</code>	

¹⁾ not supported

This concept has also been taken into account while defining the Altium extension for Interrupt Control. Here, the maximum amount of time that the thread may be suspended in a blocking service, may be also limited. For instance, while waiting for an event to happen in an interrupt or while waiting for an input/output request to complete.

2.4.19. `_POSIX_TIMERS`

Time services include the high resolution sleep operation plus clock and timer operations. Timers can be created by the user to measure time intervals and be notified by the system when such interval has elapsed, or when a given time has been reached. A signal is used as the notification mechanism.

All functions of the `_POSIX_TIMERS` interface option are supported.

TASKING POSIX Implementation

_POSIX_TIMERS	
clock_getres()	timer_delete()
clock_gettime()	timer_getoverrun()
clock_settime()	timer_gettime()
nanosleep()	timer_settime()
timer_create()	

2.4.20. _POSIX_MESSAGE_PASSING

In the PSE51 profile of IEEE Std 1003.13-1998, message queues were required because commercial real-time operating systems available at that time with similar functionality to the Minimal Realtime System Profile typically included some form of message queuing mechanism for communication between threads.

However, many embedded real-time applications for small systems do not require message queues, and this feature makes the implementation larger. Because message queues can be easily implemented by the application using mutexes and condition variables, this version of the standard, IEEE Std 1003.13-2003, has dropped the requirement to support message queues.

This option is included in the TASKING POSIX implementation, all functions are supported. To include this functionality to a project you must select **Message Queue Support** in the **Add Software Services and Device Stacks** dialog in the Software Platform Builder.

_POSIX_TIMERS	
mq_close()	mq_send()
mq_getattr()	mq_setattr()
mq_notify()	mq_unlink()
mq_open()	mq_timedreceive() ¹⁾
mq_receive()	mq_timedsend() ¹⁾

¹⁾ also requires `_POSIX_TIMEOUTS`

2.4.21. POSIX.26

An implementation conforming to PSE51 must also support POSIX.26: Device Control Application Program Interface (API) . This API consists of one function `posix_devctl()` this has been defined instead of `ioctl()`. The developers of POSIX.1 decided not to standardize `ioctl()` because it was not type safe, it had a variable number of parameters, and it had behaviors that could not be specified by the standard because they were driver-dependent.

POSIX.26 Functions
<code>posix_devctl()</code>

Chapter 3. Interrupt Control Extension

This chapter describes the Altium extension to the POSIX standards to include Interrupt Control services. The first part of this chapter describes the relation between threads, interrupt handlers, peripherals, and interrupts. The second part of this chapter describes the related services in detail.

3.1. Introduction

Real-time applications need to handle device interrupts directly in order to implement their application-level I/O drivers. Interrupt control is a necessary and crucial component in real-time development environments. The current official POSIX standards, however, do not yet define interrupt control.

Obviously, most operating systems designed specifically to support real-time applications provide interfaces for application interrupt handling and control. The TASKING POSIX implementation is no exception. Altium has implemented its own version of Interrupt Control. This is an Altium extension to the standard and therefore compatibility cannot be guaranteed when you use these interfaces with other real-time POSIX implementations.

The names of the interfaces are present in the include file `interrupts.h`.

With the interfaces defined in this chapter you can:

- block a thread while it waits for the arrival of an interrupt (while yielding the CPU).
- program interrupt handlers that you must associate with a given physical interrupt without having to write assembly code.
- dynamically add and remove the above mentioned handlers.
- protect critical sections that are shared by threads or by threads and interrupt handlers.
- port POSIX applications to new interrupt controllers.

3.2. Definitions

3.2.1. Interrupt Lines and Interrupt Controllers

In general a core has one or more exceptions, one or more internal interrupt lines, and one or more external interrupt lines. An internal line is "connected" to an internal interrupt source, for instance an internal timer unit. An external interrupt line is connected directly to an external interrupt source, or it may be connected to an interrupt controller. An interrupt controller connects multiple interrupt sources (IRQs) to the same external interrupt line while it organizes the correct priorities between these interrupts.

The following figure contains a schematic structure of interrupt lines and exceptions.

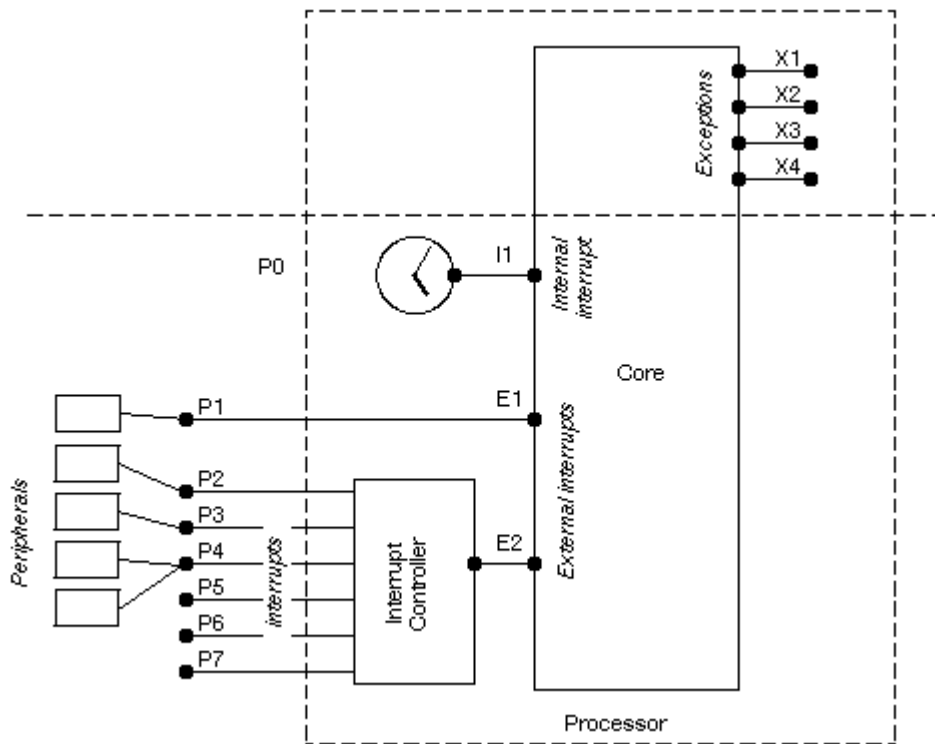


Figure 3.1. Interrupt lines and exceptions

3.2.2. Exceptions

Normally, exception lines are asserted after specific internal conditions occur in the core that often require immediate action by the system software. The amount and nature of these exceptions depend on the processor, but usually most exceptions are unexpected because they are the result of error conditions. A few exceptions can be programmed to occur through the use of exception-causing instructions. Still other exceptions can occur when pre-programmed conditions are recognized by the processor.

Exception interrupts can be maskable or non-maskable (most usual) and they are related only to the processor's architecture.

In the context of real-time POSIX systems, most of these exceptions should never occur since they normally indicate that a serious error occurred in the application. Therefore, threads are never allowed to block while waiting for the arrival of an exception. Neither can threads associate handlers with exceptions.

3.2.3. Interrupts

Interrupts that are generated by devices and communicated to the processor via external or internal interrupt lines, constitute the basic set of interrupts. An external interrupt line may be connected to an interrupt controller. An interrupt controller directs multiple interrupt requests (IRQs) to the same external

interrupt line while it organizes the correct priorities between these interrupts (see [Figure 3.1, “Interrupt lines and exceptions”](#)).

The number of interrupt priorities depends on the controller used. The scheduling of interrupts is hardware dependent (interrupts are scheduled by hardware while threads are scheduled by the kernel's scheduler). In any case, interrupt processing always has preference over thread processing, that is, an interrupt can always interrupt a thread. The kernel provides the interrupt framework, a run-time environment that dispatches execution to the right interrupt handler routine.

Interrupts are discussed in [Section 3.3, *Interrupt Framework*](#).

3.2.4. Interrupt Service Routines

An Interrupt Service Routine (ISR) is a handler that is associated with an interrupt. These routines respond to interrupt requests by saving the context of the preempted thread and then performing the processing required by the interrupt request to eventually unblock an associated thread.

Rescheduling does not take place inside the ISR handler routines. Rescheduling takes place on termination of the handler if no other interrupt is active and if the unblocked thread has a higher priority than the preempted thread.

3.3. Interrupt Framework

The purpose of the interrupt interface is to standardize the way in which non-standard interrupt generating hardware (external devices) can "connect" with application threads. It provides a framework to which all applications can refer.

Associating Interrupt Service Routines

A thread that is associated with an ISR provides four pieces of information:

- the address of the ISR code
- the address and size of the communication region of memory (used for data shared by the ISR and the registering thread)
- the interrupt ID
- the thread ID of the registering thread (implicitly).

As mentioned in the previous section, most hardware architectures have external peripherals hooked up to an external interrupt line (or to an IRQ line via an interrupt controller). You can associate each interrupt with zero or more interrupt service routines (ISRs).

The communication region is an area of memory that is visible to both the ISR (during an interrupt) and the thread (at all other times). It is your own responsibility to allocate a suitable region. The communication region is for example needed when an interrupt can be generated by more than one peripheral or when a thread receives notifications from several interrupt service routines. In these cases, information in the communication region can be used to determine which peripheral generated the interrupt or which handler sent an notification to the thread.

TASKING POSIX Implementation

There are several situations in which multiple ISRs may need to be associated with a single interrupt. Also, a single thread may associate multiple ISRs to multiple peripherals. The following figures show possible configurations.

Multiple threads are associated with multiple interrupts

The figure below shows a configuration of an application with multiple threads. Each thread associates an interrupt service routine (ISR) with an interrupt, and each interrupt is connected to a single peripheral.

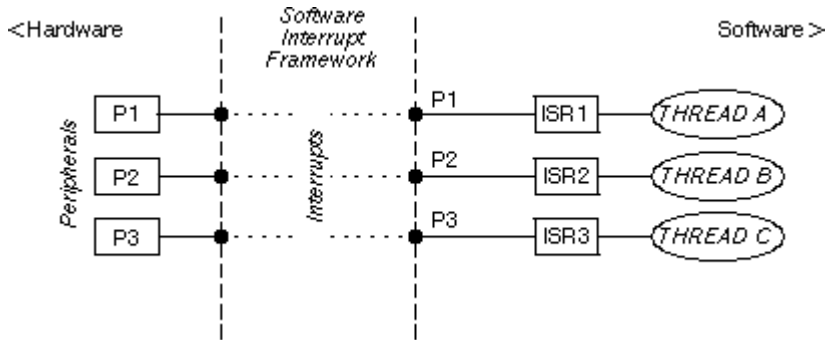


Figure 3.2. Multiple threads associated with multiple interrupts

A single thread is associated with multiple interrupts

It is possible that a single thread is associated with multiple peripherals. Each ISR may generate notification requests. The thread waits for such notifications at one place (using the `pthread_interrupt_timedwait()` function). The thread must be able to determine which interrupt triggered the ISR. Therefore, the ISRs must place all information in their respective communication regions. It is necessary to have exactly one wait point to prevent deadlocks due to interrupts arriving in an unexpected order.

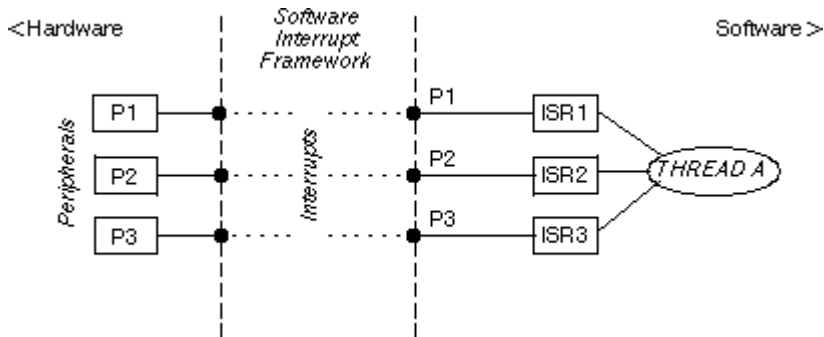


Figure 3.3. Single thread associated with multiple interrupts

Multiple peripherals are connected to the same interrupt

Many hardware architectures have less interrupts than peripheral devices so interrupts must be shared. In such cases, more than one interrupt service routine is invoked by a given interrupt (ISR1, ISR2 and ISR3 are all invoked by an interrupt on P2). In this situation, each ISR must detect which peripheral generated the interrupt, that is, whether the ISR needs to act or not.

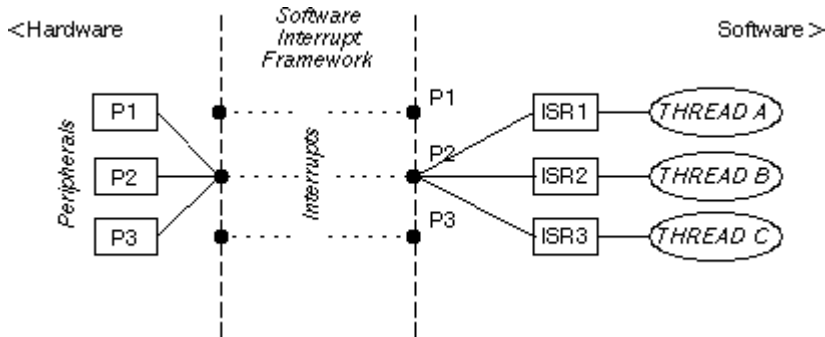


Figure 3.4. Multiple peripherals connected to the same interrupt

The same peripheral is needed by multiple threads

Several threads may need the same peripheral, in this example an internal clock. Therefore, each thread must associate its own interrupt service handler to the clock's interrupt line. In this case, it is even very likely that all ISRs run the same code; for example code to decrease a counter which is located at a given address in memory. This address would be given in the communication region of each ISR. When the counter reaches zero the associated thread is unblocked.

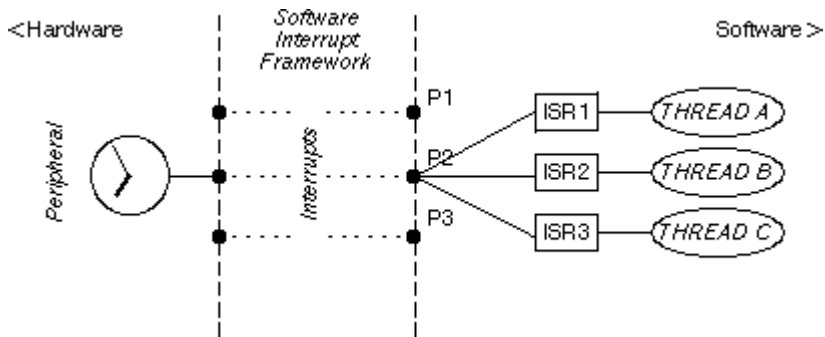


Figure 3.5. Multiple ISRs: multiple threads rely on the same peripheral

Shared interrupt service routines

In the last two configurations above, the interrupt service routines (ISR) are executed in reverse order of their registration: the last associated ISR is executed first.

TASKING POSIX Implementation

The interrupt service routine must first find out whether it should act on the incoming interrupt or not, by reading the information in the communication region. If not, the ISR immediately returns with a return value signifying that the next registered ISR handler must execute. If yes, the ISR does whatever is needed and returns with a return value stating whether the associated thread should be notified (awakened) or not and whether the next ISR handler should execute or not. Although the decision to notify (or not) is made by the user-written ISR code, the real notification is performed by the kernel. If no ISR in an interrupt chain claims the interrupt, the kernel simply ignores the unclaimed interrupt. In this case it is not specified how the software continues.

Multiple threads associated with a single interrupt service routine are not possible

The current TASKING POSIX implementation does not support the association of a single interrupt service routine with more than one thread.

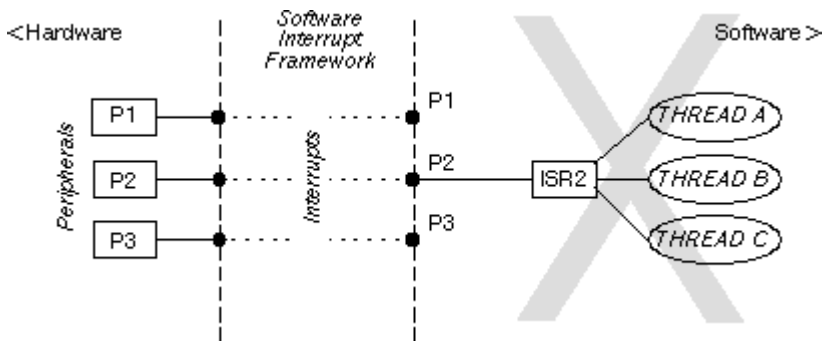


Figure 3.6. Not possible: multiple threads associated with a single interrupt handler

3.3.1. Interrupt Handlers

The interrupt framework supports two types of interrupt handlers: interrupt handlers handled by the Software Platform (further called normal interrupt handlers or interrupt handlers) and 'native' interrupt handlers handled by the application (further called native interrupt handlers). With normal interrupt handlers more than one interrupt handler per interrupt number is supported, with native interrupt handlers only one interrupt handler per interrupt number is supported.

Native interrupt handlers are installed directly into the interrupt controller's vector table. You cannot use native interrupt handlers to notify a thread. There are two types of native interrupt handlers, run-time native interrupt handlers (see [Section 3.3.1.2, Run-time Native Interrupt Handler](#)) and compile time native interrupt handlers (see [Section 3.3.1.3, Compile Time Native Interrupt Handler](#)).

Normal interrupt handlers can be installed at compile time. You can use normal interrupt handlers to notify a thread. To limit the memory usage of the normal interrupt handler administration, the maximum number of concurrently installed normal interrupt handlers is limited. You can change this limit in the Software Platform builder by changing the **Maximum Number of Interrupt Handlers** field of the **Interrupt Management** software service. If the **Interrupt Management** software service is not visible, you can add it via the **Add Software Services and Device Stacks** dialog. It may appear in the Software Platform Builder automatically when it is required by another software service or device stack.

3.3.1.1. Normal Interrupt Handler

A normal interrupt handler has the following function prototype:

```
uint32_t interrupt_handler(uint32_t number, void *context);
```

The argument `context` identifies a communication region. This is the same value as passed when registering the interrupt handler (see also `interrupt_register()`). When the ISR is called, it receives the address of the communication region as its second argument. The first argument is the interrupt number.

Normal interrupt handlers that are promoted to POSIX interrupt handlers, see `interrupt_set_posix()`, can notify the associated thread. This is done by ORing the notify bit in the return value of the interrupt handler. This causes the thread waiting in the corresponding `pthread_interrupt_timedwait()` to unblock. No other ISR-to-thread notification mechanism is available.

The possible return values (as defined in `interrupts.h`) are:

Return value	Description
<code>INTERRUPT_NOT_HANDLED</code>	This interrupt handler did not handle the interrupt and it calls the next interrupt handler for this interrupt number.
<code>INTERRUPT_HANDLED</code>	This interrupt handled the interrupt. No need to call other interrupt handlers for this interrupt number.
<code>INTERRUPT_NOTIFY INTERRUPT_HANDLED</code>	Notify the thread associated with this interrupt handler (interrupt handler, number and context).
<code>INTERRUPT_NOTIFY INTERRUPT_NOT_HANDLED</code>	Notify the thread associated with this interrupt handler (interrupt handler, number and context).

POSIX interface calls are not allowed to execute at ISR handler level. This would cause the system to behave in an undefined manner.

3.3.1.2. Run-time Native Interrupt Handler

A native interrupt handler has the following function prototype:

```
__NATIVE_HANDLER void native_interrupt_handler(void);
```

Native interrupt handlers are registered using the `interrupt_register_native()` function. This handler has no parameters, the interrupt number and context values can be obtained using the `interrupt_get_current()` and `interrupt_native_context()` routines.

The maximum number of concurrently active native interrupt handlers is not limited by the **Maximum Number of Interrupt Handlers** value of the **Interrupt Management** software service. This value only limits the maximum number of concurrently installed normal interrupt handlers.

3.3.1.3. Compile Time Native Interrupt Handler

A compile time native interrupt handler has the following function prototype:

```
__interrupt(n) void compiletime_interrupt_handler(void);
```

TASKING POSIX Implementation

where n is the interrupt number.

These interrupts are put directly at the right location in interrupt table or interrupt controller by the TASKING toolset. The Software Platform must be informed that the location of this interrupt in the interrupt table is not used by the Software Platform. For the Cortex-M architecture you can do this via the **INTERRUPT_MASK** option of the **Cortex-M Architecture** software service in the Software Platform Builder. For other architectures use the **Application ISR Mask** option of the **Interrupt Management** software service in the Software Platform Builder. Note that the **Application ISR Mask** is a 32-bit field and for each compile time native interrupt handler the corresponding bit in the ISR Mask must be set. For the Cortex-M architecture, **INTERRUPT_MASK** contains a list of interrupts that are reserved for/in use by the Software Platform, thus each compile time native interrupt used by the application must be removed from this list. For example, to exclude interrupt number 10 from the list, specify 0-9, 11-239.

3.3.2. Interrupt Control Interfaces

The following table gives an overview of the interrupt handler interfaces and usage with interrupt handler types:

Interface	Compile time native handler	Run-time native handler	Normal handler	POSIX handler
interrupts_enable	✓	✓	✓	✓
interrupts_disable	✓	✓	✓	✓
interrupts_disable_if_enabled	✓	✓	✓	✓
interrupts_enable_if	✓	✓	✓	✓
interrupt_enable	✓	✓	✓	✓
interrupt_disable	✓	✓	✓	✓
interrupt_register_native		✓		
interrupt_register			✓	✓
interrupt_deregister			✓	✓
interrupt_configure	✓	✓	✓	✓
interrupt_acknowledge	✓	✓	✓	✓
interrupt_native_context		✓		
interrupt_get_current		✓		
interrupt_set_posix			✓	
pthread_interrupt_associate				✓
pthread_interrupt_disassociate				✓
pthread_interrupt_timed_wait				✓

interrupts_enable

Syntax

```
void interrupts_enable(void);
```

Description

Enable interrupts on the processor. When running in a threaded environment interrupts are enabled at thread level, in a non-threaded environment interrupts need to be enabled in the application.

Returns

-

TASKING POSIX Implementation

interrupts_disable

Syntax

```
void interrupts_disable(void);
```

Description

Disable interrupts on the processor. When running in a threaded environment interrupts are enabled at thread level, in a non-threaded environment interrupts need to be enabled in the application.

Returns

-

interrupts_disable_if_enabled

Syntax

```
uint32_t interrupts_disable_if_enabled(void);
```

Description

Disable interrupts on the processor. This function returns a value not equal to 0 when interrupts were enabled at the time of this function call.

Returns

0 when interrupts were already disabled or a value not equal to 0 when interrupts were enabled.

TASKING POSIX Implementation

interrupts_enable_if

Syntax

```
void interrupts_enable_if(bool prev);
```

where *prev* can be:

true	Enable interrupts
false	Disable interrupts

Description

Enable interrupts depending on the value of *prev*. This function is for use with the `interrupts_disable_if_enabled()` function.

Returns

-

interrupt_enable

Syntax

```
void interrupt_enable(uint32_t number);
```

Description

Enable the interrupt specified by *number*.

Returns

-

TASKING POSIX Implementation

interrupt_disable

Syntax

```
void interrupt_disable(uint32_t number);
```

Description

Disable the interrupt specified by *number*.

Returns

-

interrupt_register_native

Syntax

```
bool interrupt_register_native(uint32_t number, void* context,
                              interrupt_native_handler_t handler);
```

where,

<i>number</i>	Interrupt number
<i>context</i>	Native handler context
<i>handler</i>	Native handler

Description

Declare an interrupt line as native and install its native handler. Native interrupt lines can hold only one handler which executes immediately upon interrupt arrival, bypassing all framework activities.

Native lines do not support sharing. In multi-threaded applications, threads cannot block on native handlers.

The *handler* function must be declared with the `__INTERRUPT_NATIVE` macro.

Returns

true if the native handler was installed, or false if the handler could not be installed.

interrupt_register

Syntax

```
bool interrupt_register(uint32_t number, void* context,  
                       interrupt_handler_t handler);
```

where,

<i>number</i>	Interrupt number
<i>context</i>	Normal handler context
<i>handler</i>	Normal handler

Description

Install a normal interrupt handler. More than one interrupt handler can be registered per interrupt number. The maximum number of simultaneously registered interrupt handlers is limited by the **Maximum Number of Interrupt Handlers** option of the **Interrupt Management** software service. The default for this option is 32 interrupt handlers. Handlers for shared interrupt lines are called in the reverse order than they were registered, until one of the handlers returns a code signifying that the interrupt has been handled.

In multi-threaded applications, threads can block on normal interrupt handlers. For this the application must promote the interrupt to a POSIX interrupt, using `interrupt_set_posix()`, and associate the interrupt to the interrupt handler, using `pthread_interrupt_associate()`, after which the thread can wait for interrupt notification using `pthread_interrupt_timedwait()`.

Returns

true to indicate success, or false otherwise.

interrupt_deregister

Syntax

```
bool interrupt_deregister(uint32_t number, void* context,  
                          interrupt_handler_t handler);
```

where,

<i>number</i>	Interrupt number
<i>context</i>	Normal handler context
<i>handler</i>	Normal handler

Description

De-register a normal interrupt handler from the specified interrupt line. De-registering an interrupt handler will not undo an `interrupt_set_posix()` on the interrupt number, but it will disassociate the thread that was associated with this interrupt handler.

Returns

true to indicate success, or false otherwise.

TASKING POSIX Implementation

interrupt_configure

Syntax

```
bool interrupt_configure(uint32_t number, interrupt_cfg_t cfg);
```

where,

<i>number</i>	Interrupt number
<i>cfg</i>	Interrupt configuration value

Description

Configure the given interrupt. This function should be used by drivers during initialization to configure the interrupt in the controller. Currently the `interrupt_cfg_t` type consists of the following values `EDGE_RISING`, `EDGE_FALLING`, `LEVEL_HIGH` and `LEVEL_LOW`.

Note: for interrupt controllers that do not support this functionality, false will be returned.

Returns

true to indicate success, or false otherwise.

interrupt_acknowledge

Syntax

```
void interrupt_acknowledge(uint32_t number);
```

where,

number Interrupt number

Description

Clear the interrupt pending flag in the interrupt controller to allow generation of a new interrupt. For example, you can use this function in interrupt handlers to acknowledge reception of the specified interrupt or before enabling the interrupt to clear a previous pending state of the interrupt.

Returns

-

TASKING POSIX Implementation

interrupt_native_context

Syntax

```
void * interrupt_native_context(uint32_t number);
```

where,

number Native interrupt number

Description

Return the native interrupt context that was provided with the `interrupt_register_native()` call.

Returns

The interrupt context.

interrupt_get_current

Syntax

```
uint32_t interrupt_get_current(void);
```

Description

Obtain the number of the currently executing interrupt. This function is for use in native interrupt handlers to find out which is the number of the interrupt causing this native interrupt handler call, as a replacement for the interrupt number parameter in normal interrupt handlers. You can use this number in a call to [interrupt_native_context\(\)](#) to get the accompanying interrupt context.

Returns

The number of the current interrupt.

TASKING POSIX Implementation

interrupt_set_posix

Syntax

```
bool interrupt_set_posix(uint32_t number, bool posix);
```

number Interrupt number

posix Treat this interrupt number as a POSIX interrupt (true) or normal interrupt (false)

Description

When an interrupt number is set to be a POSIX interrupt, a thread can associate itself with a registered normal interrupt handler, using `pthread_interrupt_associate()`, and wait for interrupt notification using `pthread_interrupt_timedwait()`. Threads can block waiting for interrupt synchronization using `pthread_interrupt_timedwait()`. The thread will wait until the interrupt handler returns a value that indicates the thread should be notified. You can limit the maximum wait time by specifying a time-out value.

Returns

true to indicate success, or false otherwise.

pthread_interrupt_associate

Syntax

```
int pthread_interrupt_associate(uint32_t number,  
                               interrupt_handler_t handler, void* context);
```

where,

<i>number</i>	Interrupt number
<i>handler</i>	Normal handler to associate
<i>context</i>	Normal handler context

Description

Associates an already installed normal interrupt handler with the current thread. The thread can block, waiting for notification, by calling `pthread_interrupt_timedwait()`. The handler can cause the thread to be notified by including the `INTERRUPT_NOTIFY` flag in its return value. Only one thread can be associated with a specific number/handler context combination at the same time.

Returns

0 if successful, or `EINVAL` error code when *number* is an invalid interrupt number, or when *number* is an interrupt number that has not been set to POSIX with `interrupt_set_posix()` or when the number/handler/context combination has not been used for `interrupt_register()`.

TASKING POSIX Implementation

pthread_interrupt_disassociate

Syntax

```
int pthread_interrupt_disassociate(uint32_t number,  
                                  interrupt_handler_t handler, void* context);
```

where,

<i>number</i>	Interrupt number
<i>handler</i>	Normal handler to disassociate
<i>context</i>	Normal handler context

Description

Cancel any existing association between the interrupt line and the specified interrupt handler. The handler, context and number parameters should match the ones given to the corresponding `interrupt_register()` and `pthread_interrupt_associate()` calls.

Returns

0 if successful, or `EINVAL` error code when *number* is an invalid interrupt number, or when *number* is an interrupt number that has not been set to POSIX with `interrupt_set_posix()` or when the number/handler/context combination has not been used for `interrupt_register()`.

pthread_interrupt_timedwait

Syntax

```
int pthread_interrupt_timedwait(const struct timespec* timeout);
```

where,

timeout Time-out interval, or NULL for no time-out

Description

Block the calling thread until notified by an interrupt handler. If an interrupt notification was delivered to the calling thread prior to the call to the `pthread_interrupt_timedwait()` function, and this notification has not previously caused a call to the `pthread_interrupt_timedwait()` function to be unblocked, then the calling thread is not blocked. Instead the `pthread_interrupt_timedwait()` function returns immediately.

If the value of the *timeout* input argument is not NULL, the wait for an interrupt to occur is terminated when the specified time-out period expires. If the *timeout* input argument is NULL, the wait is terminated only by the interrupt notification. The time-out expires after the interval specified by *timeout* has elapsed since the wait began. The time-out is based on the `CLOCK_REALTIME` clock. The resolution of the time-out is determined by the resolution of the clock that it uses.

Under no circumstance the function will fail with a time-out error if the interrupt notification occurred prior to the `pthread_interrupt_timedwait()` call. The validity of the *timeout* argument is not checked if the interrupt notification occurred prior to the `pthread_interrupt_timedwait()` call.

Returns

0 if successful, or otherwise one of the following error codes is returned:



<code>EINVAL</code>	The thread would have blocked, but the <i>timeout</i> argument specified an invalid value.
<code>EINTR</code>	A signal interrupted this function.
<code>ETIMEDOUT</code>	The interrupt notification was not received before the specified time-out expired.

Chapter 4. Build and Configure

This chapter describes how to use the TASKING toolset to build and configure a multi-threading application.

4.1. How to Build a Multi-threading Application

To build a multi-threaded application, take the following steps:

1. Add the POSIX application sources to your project.
2. Add a TASKING Software Platform Document to your project (**File » New » TASKING Software Platform Document**).
3. Use the Add Software Services and Device Stacks dialog () to add **POSIX Multi-threading**.
4. Change (or not) the default properties of the newly added **POSIX Multi-threading** software service.
5. Click the **Generate Code** button () , so the Software Platform Builder adds the required source code to the project and generates the required configuration files.
6. Build the application.

The **POSIX Multi-threading** software service does not include signals and message queues. You can add signal and message queue support to a project by selecting the **POSIX Signals** and/or **POSIX Message Queues** in the Add Software Services and Device Stacks dialog of the Software Platform builder.

4.2. Multi-threading Applications

As with non-POSIX applications, the simplest POSIX application is:

```
int main(void)
{
    return 1;
}
```

The difference here is that the code after `main()` runs as a thread embedded in an RTOS kernel. But if the application does not call POSIX services in `main()`, in essence it behaves similar to a non-POSIX application. The only difference is that interrupts are enabled at thread level. If you have an application that has not selected **POSIX Multi-threading** you need to enable interrupts in the application code (`interrupts_enable()`). When `main()` returns, the kernel realizes that there is nothing left to do and exits the program. The TASKING POSIX implementation does not support the `argc` and `argv` parameters of `main()`.

The following example is another application that uses very simple POSIX functionality.

```
#include <pthread.h>
```

TASKING POSIX Implementation

```
static volatile int forever;

void * threaded1( void * arg )
{
    /* thread 1 runs first and exits */
    return NULL;
}

void * threaded2( void * arg )
{
    /* thread 2 runs at last and forever */
    while(forever==0);
    return NULL;
}

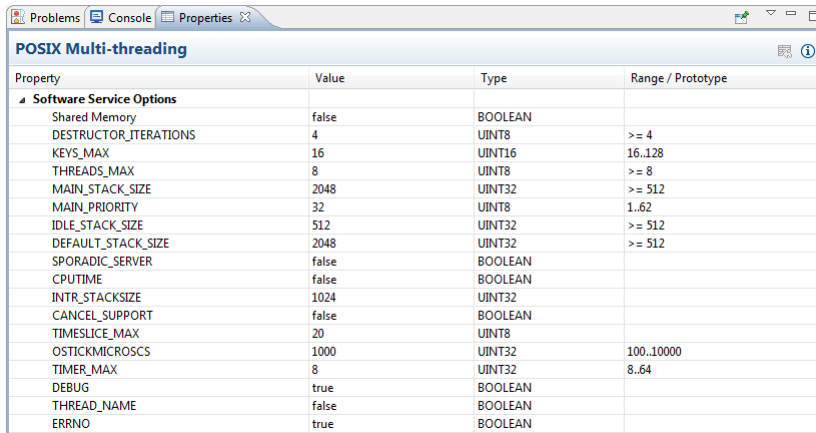
int main(void)
{
    pthread_t p1,p2;
    /* create thread 1 */
    pthread_create( &p1, NULL, threaded1, NULL );
    /* create thread 2 */
    pthread_create( &p2, NULL, threaded2, NULL );
    return (int)(p1+p2);
    /* the kernel has 2 other threads to run */
}
```

4.3. How to Configure a POSIX Application

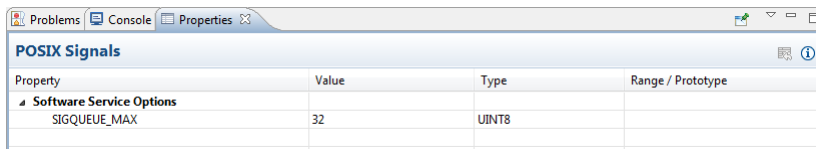
You can configure **POSIX Multi-threading** in the Properties view. If you want support for signals and message queues you must add it separately by selecting **POSIX Signals** and **POSIX Message Queues** respectively. The configuration parameters are displayed with a name, range, default value and a brief description.

After you have set the values of the POSIX Multi-threading, POSIX Signals and or POSIX Message Queues configuration parameters, and after you have used the Software Platform Builder to generate configuration files and add appropriate kernel source files to the project, the configuration files are visible in the `SoftwarePlatform/Configuration` directory and the source files are visible in the `SoftwarePlatform/services/posix` directory.

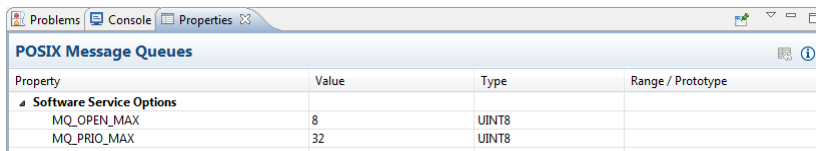
The following three figures show the configuration parameters.



Property	Value	Type	Range / Prototype
Software Service Options			
Shared Memory	false	BOOLEAN	
DESTRUCTOR_ITERATIONS	4	UINT8	>= 4
KEYS_MAX	16	UINT16	16..128
THREADS_MAX	8	UINT8	>= 8
MAIN_STACK_SIZE	2048	UINT32	>= 512
MAIN_PRIORITY	32	UINT8	1..62
IDLE_STACK_SIZE	512	UINT32	>= 512
DEFAULT_STACK_SIZE	2048	UINT32	>= 512
SPORADIC_SERVER	false	BOOLEAN	
CPUTIME	false	BOOLEAN	
INTR_STACKSIZE	1024	UINT32	
CANCEL_SUPPORT	false	BOOLEAN	
TIMESLICE_MAX	20	UINT8	
OSTICKMICROSCS	1000	UINT32	100..10000
TIMER_MAX	8	UINT32	8..64
DEBUG	true	BOOLEAN	
THREAD_NAME	false	BOOLEAN	
ERRNO	true	BOOLEAN	



Property	Value	Type	Range / Prototype
Software Service Options			
SIGQUEUE_MAX	32	UINT8	



Property	Value	Type	Range / Prototype
Software Service Options			
MQ_OPEN_MAX	8	UINT8	
MQ_PRIO_MAX	32	UINT8	

By selecting or deselecting the configuration parameters you can scale the system to suit your needs. For instance, by deselecting some of the Option Requirements, the system will perform less real-time checks, thus becoming faster and smaller (`_POSIX_THREAD_CPUTIME`, `_POSIX_THREAD_SPORADIC_SERVER` etc ..). The rule for best practice is to deselect all the Option Requirements that you do not need. This concept applies also to system limits. For example, you can use `THREADS_MAX`, `TIMER_MAX` and `MQ_OPEN_MAX` to limit the maximum number of these items that can be available at a time. Please note that some of the functionality that is available in the Software Platform also uses a thread. Keep this in mind when limiting these MAX values. By optimizing these system limits, the RTOS data size becomes smaller and with some checks disabled the code will run faster (less overhead due to checks).

4.4. How to Debug a POSIX Application

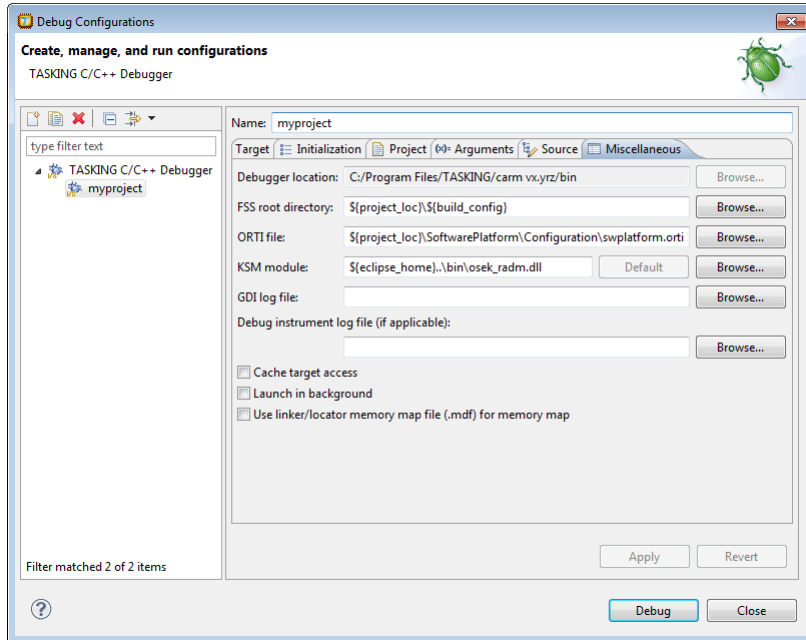
The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a *kernel support module* (KSM) or *RTOS-aware debugging module* (RADM). Specifically, the TASKING VX-toolset ships with a KSM supporting the OSEK standard (`osek_radm.dll`). The Software Platform Builder creates an OSEK Run Time Interface (ORTI) file (`swplatform.orti`) in the `SoftwarePlatform/Configuration` folder. You need to specify this file on the **Miscellaneous** tab while configuring a customized debug configuration:

TASKING POSIX Implementation

1. From the **Debug** menu, select **Debug Configurations...**

The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject**.
3. Open the **Miscellaneous** tab.



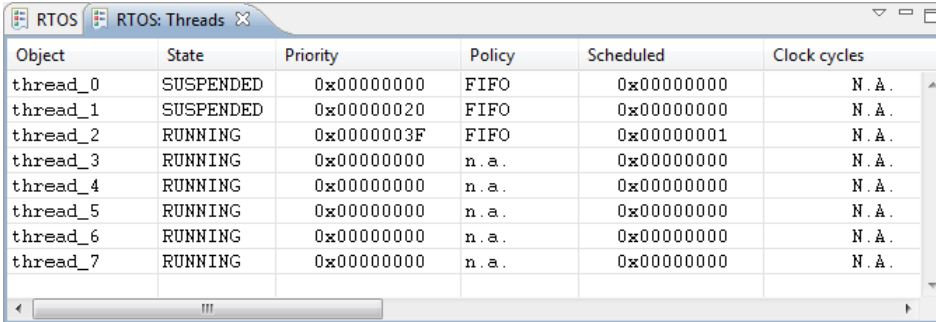
4. In the **ORTI file** field, specify the name of the ORTI file (`swplatform.orti`).
5. In the **KSM module** field, click **Default** or browse for the file `osek_radm.dll` in the `bin` directory of the toolset.

The debugger RTOS view (**Window » Show View » Other... » TASKING » RTOS**) allows inspection of all POSIX resources including threads, clocks, timers, message queues and signals.

For example, to show the threads:

- From the View Menu button (∇), select **OSEK/ORTI » TASKING POSIX » Threads**.

The RTOS: Threads view appears.



The screenshot shows a window titled "RTOS: Threads" with a table listing the state and properties of various threads. The table has six columns: Object, State, Priority, Policy, Scheduled, and Clock cycles. The threads listed are thread_0 through thread_7. thread_0 and thread_1 are in a SUSPENDED state, while threads 2 through 7 are in a RUNNING state. The Priority column shows hexadecimal values, and the Policy column shows FIFO or n.a. The Scheduled column shows hexadecimal values, and the Clock cycles column shows N.A.

Object	State	Priority	Policy	Scheduled	Clock cycles
thread_0	SUSPENDED	0x00000000	FIFO	0x00000000	N.A.
thread_1	SUSPENDED	0x00000020	FIFO	0x00000000	N.A.
thread_2	RUNNING	0x0000003F	FIFO	0x00000001	N.A.
thread_3	RUNNING	0x00000000	n.a.	0x00000000	N.A.
thread_4	RUNNING	0x00000000	n.a.	0x00000000	N.A.
thread_5	RUNNING	0x00000000	n.a.	0x00000000	N.A.
thread_6	RUNNING	0x00000000	n.a.	0x00000000	N.A.
thread_7	RUNNING	0x00000000	n.a.	0x00000000	N.A.

