cadence®

Xtensa® XOS

Reference Manual

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Issue Date: 10/2018

Release: RI-2018.0

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Tables

1. Preface

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- $ represents your shell prompt, in user-session examples.
- **literal_input** indicates literal command-line input.
- `variable` indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- *... output ...* indicates unspecified program output.
- *[optional-`variable`]* indicates an optional parameter.
- [`variable`] indicates a parameter within literal square-braces.
- {`variable`} indicates a parameter within literal curly-braces.
- (`variable`) indicates a parameter within literal parentheses.
- | means OR.
- *(var1* | *var2)* indicates a required choice between one of multiple parameters.
- *[var1* | *var2]* indicates an optional choice between one of multiple parameters.
- *var1 [, varn]*∗ indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- 0x at the beginning of a value indicates a hexadecimal value.
- b means bit.
- B means byte.
- flush is deprecated due to potential ambiguity (it may mean write-back or discard).
- Mb means megabit.
- MB means megabyte.
- PC means program counter.
- word means 4 bytes.

Changes from the Previous Version

2. Introduction

The XOS embedded kernel from Cadence is designed for efficient operation on embedded systems built using the Xtensa architecture. The goal of XOS is to provide a redistributable, royalty-free reference platform that users can build upon for their applications. Although various parts of XOS will continue to be tuned for efficient performance on the Xtensa hardware, the majority of the code is written in standard C and is not Xtensa-specific. XOS is distributed in source code form, as well as prebuilt libraries matching your target core configuration. Since there are several configuration options, each of which has an impact on performance and/or memory footprint, there is no "ideal" default configuration. It is expected that many users will rebuild XOS using the configuration settings that work best for them.

In addition to the core system and thread management operations, XOS also provides support for primitives such as mutexes, semaphores, message queues, etc. These modules are implemented in a way that imposes minimal or no speed/space penalty if not used. For a list of modules supported in the current release, see the Features section.

2.1 Goals

XOS has been designed and implemented with the following major goals in mind.

- Fastest possible context switching performance
- Fast operation for common actions (e.g. mutex lock/unlock, semaphore put/wait).
- Smallest possible footprint based on used features. Unused features should not contribute anything to memory footprint.
- No dynamic memory allocation in the core OS, able to run without any memory allocator.
- No compile-time limits on number of OS objects (tasks, mutexes, semaphores, etc.).
- Debug and trace features built in, but can be completely compiled out. Zero overhead if not used.
- Takes advantage of available hardware features to minimize power / energy consumption.

2.2 Requirements

XOS requires some Xtensa configuration options to be present to function properly. These are:

- Exception Option (XEA2 or later, neither XEA1 nor XEAX are supported).
- The XSR instruction (this is part of the core architecture but was not present until T1040).

In addition, for preemptive multithreading support, at least one of the following is required:

- Timer Option with at least one timer interrupt at level <= EXCM_LEVEL.
- Interrupt Option with at least one interrupt source at level <= EXCM_LEVEL.

XOS can run without timers; however no timing services will be available. XOS can also run without any interrupts in the system - in that case only cooperative multithreading will be possible since there will be no preemption source in the system. All other options may or may not be present. At this time, there is no option which precludes the use of XOS (other than XEA1 or XEAX).

2.3 Licensing

XOS is licensed under an MIT-style open source license. For the full text of the license terms refer to the files in the source distribution.

2.4 Features

The following major features are available in this release:

- Support for both windowed and CALL0 ABIs.
- Preemptive multithreading.
- Support for context switching of coprocessor states.
- Some support for run-to-completion (RTC) threads.
- Support for level 1, medium priority, and high priority interrupts.
- Dynamic tick support.
- Timer API.
- Mutex API.
- Semaphore API.
- Event API.
- Message Queue API
- Stopwatch API
- System Log support.
- Statistics support.
- Thread safe C library - both newlib and xclib are supported.
- Basic test suite / sample programs.

3. Installing and Configuring XOS

3.1 XOS Installation

XOS sources are installed along with the Xtensa Tools. The sources can be found under

`<xtensa_tools_root>/xtensa-elf/src/xos`

where `<xtensa_tools_root>` is the directory where the Xtensa Tools have been installed. The sources are self-contained and can be copied elsewhere by copying the directory tree under the xos directory.

The XOS header files are also present in

`<xtensa_tools_root>/xtensa-elf/include/xtensa`

This is where the compiler will find them if you use the standard include mechanism:

```
#include <xtensa/xos.h>
```

If you modify the header files, then you can copy the updated files into this location or point the compiler to your updated files. This relocation can be done by using the –I compiler option to specify the new location. For example, if your updated versions of the header files are at location /home/work/myxos, then use:

```
-I/home/work/myxos/include
```

to specify the new location to the compiler. If you edit xos_params.h to change the configuration options, then remember to include the modified version of this file in your application as well.

Three versions of the XOS library are provided prebuilt with your Xtensa software installation.

File Name	Description
`libxos.a`	Standard version built with the default configuration options. This version is built with the compiler flags “-O2 –g” and is suitable for development use but is also optimized to a reasonably high level.
`libxos-debug.a`	Debug version built with the same options and compiler flags as above but with the XOS_DEBUG_ALL option enabled. This version has all debug support enabled.

`libxos-ipa.a`	Heavily optimized version built with the compiler flags "-O3 –ipa". This version is compiled without debug information and is not suitable for debugging. Note that your application code must also be compiled and linked with the "-ipa" flag to gain the full benefit of optimizations.

3.2 Changing the Build Configuration

XOS configuration options are controlled by a header file. The various options are listed below. XOS must be rebuilt after any change in the configuration options. The build process is just like any other library; there are no special steps required. A makefile is provided for command-line and scripted builds. An Xtensa Xplorer project is also available.

To minimize code size, the XOS library should be compiled with the `-ffunction-sections` compiler option. When used together with the `--gc-sections` linker option, all unused code is removed from the final executable.

The XOS configuration options can be modified by either editing xos_params.h, or by overriding the settings in this file from the build command line. The latter can be done from a makefile, or by defining the overrides in the Xtensa Xplorer project settings. In either case, make sure that the same set of options has been applied to building the XOS library and your application. Otherwise, runtime errors and/or unexpected behavior may occur because of the mismatch between the XOS code and your application.

3.3 Rebuilding XOS using command line tools

1. Open a command shell window depending on your operating system.
2. Change directories to the 'src' directory under the XOS installation directory.
3. Edit the makefile in this directory to specify the build override options as needed (or edit the xos_params.h file).
4. Run "`xt-make clean`" followed by "`xt-make all`" using the makefile in this directory.
5. The compiler and assembler options can be overridden by specifying CFLAGS and ASFLAGS on the make command line. See the makefile and the Xtensa C compiler user guide for details of the various available options.
6. The makefile supports some shortcut options for building the library in specific ways. See the MODE=xxx options in the makefile for details.

3.4 Rebuilding XOS using Xtensa Xplorer

The XOS installation does not contain an Xplorer workspace for rebuilding XOS. You can create one yourself or ask your technical representative for assistance. If you create your own build process for XOS, remember that the use of the following compiler options is required.

- `-mno-coproc`
  - Disallow use of coprocessor instructions and registers in XOS code.
- `-mlongcalls`
  - Convert direct calls to indirect calls where necessary.
- `-mno-l32r-flix`
  - Disallow FLIXing of L32R instructions.

3.5 Configuration Options

Table 3.1: Configuration Options

Configuration Option	Controls
XOS_NUM_PRIORITY	Number of thread priority levels (max 32). The default is 16.
XOS_CLOCK_FREQ	CPU clock frequency (in Hz) defined at compile time. The clock frequency can also be set at run time.
XOS_INT_STACK_SIZE	Interrupt stack size in bytes. The default size is 8 KB.
XOS_DEBUG_ALL	Enables debug features (sub-flags enable specific debug features).
XOS_OPT_STATS	Enables the core statistics module. Enabled by default.
XOS_OPT_MSGQ_STATS	Enables statistics for the message queue module. Disabled by default.
XOS_OPT_BLOCKMEM_STATS	Enables statistics for the block memory module. Disabled by default.
XOS_OPT_INTERRUPT_SWPRI	If enabled, then numerically higher interrupts at the same priority level are left enabled while handling low and medium priority interrupts. See Interrupt and Exception Dispatch for more details. Enabled by default for XEA2, not meaningful for XEA3.
XOS_OPT_STACK_CHECK	Enables thread stack overrun checking. Enabled by default if XOS_DEBUG_ALL is enabled, else disabled by default.
XOS_OPT_THREAD_SAFE_CLIB	Enables thread safety for the C library. The size of the thread control block (TCB) does increase to accommodate the per-thread context for the library. Enabled by default for newlib and xclib.

XOS_OPT_TIMER_WAIT	Enables waiting on a timer object. Enabled by default.
XOS_OPT_TIME_SLICE	Enables time slicing between multiple threads at the same priority. Time slicing happens every XOS_TIME_SLICE_TICKS timer ticks if enabled. Enabled by default.
XOS_TIME_SLICE_TICKS	Specify the number of timer ticks per time slice interval. Used only if XOS_OPT_TIME_SLICE is enabled. The default is 1, the range is 1-50.
XOS_OPT_WAIT_TIMEOUT	Enables timeouts on waits (e.g. mutex lock attempt). Enabled by default.
XOS_OPT_MUTEX_PRIORITY	Enables priority inversion protection. Enabled by default.

Notes:

1. XOS allows the CPU clock frequency to be specified at run time (boot time) for additional flexibility.
2. If changing the interrupt stack size, make sure that the new stack size is adequate for worst case interrupt nesting and/or nested function calls from interrupt handlers.
3. XOS_OPT_STATS, XOS_OPT_TIMER_WAIT, and XOS_OPT_WAIT_TIMEOUT all require a hardware timer to be present.
4. XOS_OPT_THREAD_SAFE_CLIB works with the newlib and xclib libraries. Check your core configuration to see which C library has been selected.

4. The XOS System Model

XOS is built on the single-application, multi-thread model. XOS is built as a library and statically linked to the application code to generate a single executable file. All code and data in the application share the same address space. Application threads have full visibility into global data. At this time there is no support for thread-local static data (only stack variables are thread local). XOS code runs at the same privilege level as the application code, and there is no protection for XOS data from accidental or malicious corruption by application code. XOS system calls are regular function calls, not traps. This makes for faster execution and smaller code size. Selected system calls may be inlined for performance reasons. In the future, if it is required to run the XOS kernel in a privileged mode, such support will be added without altering the API.

XOS itself does not create any threads except the "idle" thread, which is not really a thread in the sense that it has no state and does no useful work. Application threads may be created and destroyed any number of times at run time. There is no limit on the number of threads that can exist - this is limited only by memory available for thread control blocks and thread stacks.

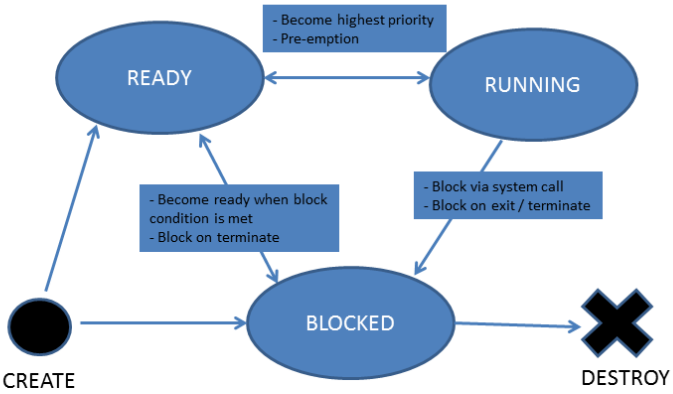

Figure 4-1. XOS Thread States.

Threads can exist in one of three possible states - Ready, Running, or Blocked. When a thread is created, it is put into either the Ready state or the Blocked state, depending on what was specified during creation.

- A thread in the Ready state is placed at the end of the ready queue for its priority level, and eventually will get to run, at which time it will go into the Running state.
- A Running thread may execute a blocking system call and go into the Blocked state. It can also be pre-empted by a thread at a higher priority and go back into the Ready state.
- A thread in the Blocked state does not go back into the Ready state until the blocking condition is satisfied.

XOS supports multiple priority levels. The number of priority levels is configurable at build time. A higher priority number denotes a higher priority thread - zero is the lowest priority level. The scheduler will ensure that at any instant, if the current running thread has priority P, then no thread at a priority $>$ P is ready in the system. (NOTE: This is applicable to a single-CPU system, but likely to change when we start supporting multiple CPUs). If such a thread does become

ready, it will immediately pre-empt the previous thread and become the running thread. If there are multiple threads at the same priority level, they will be run round-robin or be time-sliced, depending on a configuration parameter.

In round-robin mode, the following conditions must hold true for a ready thread at priority level P to become running:

- There must be no threads at priority $>$ P in the ready state
- The selected thread must be at the head of its ready queue
- The thread currently executing must have yielded the CPU

In time slice mode, the following conditions must hold true in the same situation:

- There must be no threads at priority $>$ P in the ready state
- The selected thread must be at the head of its ready queue
- The thread currently executing must have used up its time slice OR have yielded the CPU

The only exception to these cases is when preemption is disabled by the application. The thread that disables preemption will continue to run until it re-enables preemption or yields the CPU.

4.1 Initialization and Startup

At power on or reset, the processor starts executing code from the Reset vector. This vectors to the reset handler, which is normally the default reset handler provided with XTOS. The user can customize the reset handler as they wish. Eventually, the reset handler code must call `_start`, which is a piece of code that initializes the C/C++ runtime library and a few other things. Eventually, `_start` calls `main()` which is the entry point for the application. Up to this point, XOS has not been involved in system startup.

From the `main()` function, the application must initialize the XOS subsystem and start multitasking. This is when XOS takes over control of the processor and decides which thread to run. There are two options available at this point.

4.1.1 Starting up with main() as a thread

If you call xos_start_main(), then two threads are created, the idle thread and the "main" thread. The "main" thread is just the code running within the `main()` function. In other words, the application's `main()` function becomes its first user thread. After creating the threads and initializing XOS, xos_start_main() returns to its caller, which can then do anything that a user thread is allowed to do. New threads may be created and activated beyond this point.

There are some differences between a regular thread and the “main” thread. The “main” thread’s stack remains on the startup stack, which is the stack set up by the `_start` setup code. This is usually located at the top of available memory. Since execution has to return from xos_start_main(), the stack cannot be relocated inside that function. So there is no way to specify either the location of the stack for “main”, or the stack size. The stack for this thread grows downward, while the C library heap grows upward. This growth can become a problem in situations where there is a lot of dynamic memory allocation. Secondly, because of the way this thread is created, space to save the TIE state cannot be allocated at the top of the stack. So, XOS allocates a static buffer to store the “main” thread’s TIE state. Last but not least, if the “main” thread ever returns, then `exit()` will be called to terminate the application. Thus, you must take care that `main()` never returns.

4.1.2 Starting up with main() not a thread

If you call xos_start(), the behavior is different. xos_start() will create only the idle thread and then start the scheduler. It will not return to its caller, so execution will never return to `main()`. In this mode, at least one thread must be created before calling xos_start(), or else there will be nothing but the idle thread to run when the scheduler is started. The advantage of this mode is that the startup stack (main’s stack) is freed up, and no memory is reserved for the “main” thread’s TCB and coprocessor/TIE state save area. Also, the possibility of `main()` returning and calling `exit()` to halt the system is eliminated.

4.2 Starting the System Timer

If a timer is to be used, then xos_start_system_timer() must be called to set up and start the system timer. This can be called before or after starting XOS. The timer number to use can be explicitly specified, or a timer number of -1 can be specified to indicate that XOS should automatically select the timer.

If the system clock frequency is to be detected at runtime, this should be done and the frequency indicated to XOS before XOS is started. The function xos_set_clock_freq() should be called to set the system clock frequency.

XOS objects can be created before xos_start() or xos_start_main() is called. Interrupt and exception handlers can be registered, but interrupts will not be enabled until multitasking is started.

5. Core Functions

The XOS core functions are not configurable and are not optional. This functionality is always present. This includes XOS startup, exception and interrupt handling, runtime error handling, and some miscellaneous functions.

Table 5.1: Core Functions

API Call	Description
xos_get_int_pri_level()	Get current CPU interrupt priority level.
xos_set_int_pri_level()	Set current CPU interrupt priority level.
xos_restore_int_pri_level()	Restore interrupt priority level to that returned by xos_set_int_pri_level().
xos_disable_interrupts()	Disable low and medium priority interrupts.
xos_restore_interrupts()	Restore CPU's interrupt enable state.
xos_interrupt_enable()	Enable specific interrupt.
xos_interrupt_disable()	Disable specific interrupt.
xos_register_interrupt_handler()	Register an interrupt handler for specified interrupt.
xos_register_hp_interrupt_handler()	Register a high priority interrupt handler for a specified level (XEA2 only).
xos_register_exception_handler()	Register exception handler for specified exception.
xos_fatal_error()	Report a fatal error and halt.
xos_register_fatal_error_handler()	Register a fatal error handler function.
XOS_ASSERT()	Assert if the expression is false (call xos_fatal_error()). Compiled out if XOS_DEBUG not enabled.

6. Managing and Scheduling Threads

The thread module provides services to create and manage threads. As described earlier, an XOS thread is a sequence of execution with its own state that is managed by the scheduler. The thread API calls help to initialize thread state, execute threads, change their runtime priority, and destroy them. There are also functions to get and set various thread properties.

6.1 Creating Threads

Threads may be created before or after starting XOS. However, if you are going to call xos_start(), then at least one thread must be created before calling xos_start(). If this is not done, then xos_start() will create the idle thread and start the scheduler, and control will never return to the application code. If xos_start_main() is called, it creates the 'main' thread out of the calling function, which is normally the main() function of the application. This becomes the first user thread in the system. The 'idle' thread is a pseudo-thread which is assigned the lowest priority in the system. It runs only when there is no other thread ready to be run. Unlike a real thread it has no state and no stack, and is used to place the CPU into a low power state until there is actual work to be done.

Note that the memory required for the thread instance (its stack and thread control block) must be provided as input to this function. This memory must be allocated by the application and must remain allocated for the lifetime of the thread. The Thread Control Block (TCB) is a fixed size structure for a specific CPU configuration, and is described by the structure XosThread. The stack size depends on the thread's purpose and the system design. See the later section Stack Usage for more details of stack sizing. A call to xos_thread_create() can cause a context switch to the newly created thread, if it happens to be the highest priority thread that is ready. This context switch can be avoided, if desired, by creating the thread in a suspended state. In that case, the thread must be unsuspended explicitly at some point in the future to get it to run.

6.2 Running Threads

Once a thread is ready to run, it is put on the ready queue at its current priority level. The scheduler picks this thread to run when it gets to the head of its ready queue and no other higher priority threads are ready to run. Once running, the thread continues to run until it either yields the CPU, is preempted by a higher priority thread, or is time sliced with another thread at the same priority. Multiple threads of the same priority are time-sliced by the scheduler if time slicing is enabled (via XOS_OPT_TIME_SLICE). The threads are scheduled in round robin fashion, to ensure that none are starved. Time slicing, if enabled, occurs on every timer tick. Note that time slicing will not happen on interrupts other than the timer tick interrupt. However, fairness via time slicing cannot be guaranteed when dynamic ticks are enabled, as timer ticks

may occur at long and irregular intervals. Preemption by higher priority threads occurs during timer or other interrupt handling. Whenever the OS is returning from interrupt handling to normal mode, the scheduler checks to see if a higher priority thread is ready. If there is such a thread, then execution switches to that thread. Preemption can also occur because of system calls to change the state of shared objects, such as releasing a mutex, signaling a semaphore, etc. Such actions can unblock threads at higher priorities, and they will immediately become ready to run.

6.3 Suspending and Terminating Threads

Calling xos_thread_delete() will remove a thread and free up all its resources. However, this function can only be called on a thread that has already exited. Threads can exit in three ways. A thread can terminate itself by calling xos_thread_exit(). A thread can also return from its entry function to terminate; this, too, results in a call to xos_thread_exit(). A thread can terminate another thread by calling xos_thread_abort() on it. In all cases, the thread will run its exit handler, if one has been specified. The exit handler is a user-defined function that will be called when the thread is exiting. Typically, it is used to free up resources, record the results of the thread's activity, etc. An exit handler does not need to explicitly signal other threads that may be waiting for this thread to terminate. That action is performed by xos_thread_join(). A thread can call xos_thread_join() to wait on another thread's termination. When the target thread terminates, the waiting thread will be woken up with a return value equal to the terminating thread's return code. Of course, there may be cases where calling xos_thread_join() is impractical. In those situations, the exit handler can, for example, set an event bit or send a message to a waiting thread to indicate termination. The exit handler runs in the context of the exiting thread.

Calling xos_thread_abort() can have unexpected side effects, so it should be used carefully. An aborted thread directly executes the exit handler (if any) once it is scheduled, abandoning whatever it may have been doing. If it was blocked waiting for something, the wait is terminated. If the thread owns one or more mutexes, they will not be freed up by the abort process. You must make sure that any owned mutexes are freed properly. You must also make sure that any memory allocated via malloc() or equivalent calls is freed up properly. This can best be done in the exit handler. It is important to remember that the target thread may not have finished exiting when the call to xos_thread_abort() returns. This is because the thread being aborted is (by definition) not running when the call is made, and even if woken up, may not get scheduled right away. The only way to reliably detect whether the thread has exited is to call xos_thread_join() or xos_thread_get_state(). The latter function never blocks, so it can be called from interrupt handlers.

A running thread can release the CPU by calling xos_thread_yield(). This causes the thread to be queued at the tail end of the ready queue for its priority level. The next ready thread at the same priority level is then picked to run. If there are no other ready threads at the same priority level, then the thread is resumed immediately. Notice that yielding the CPU does not allow lower priority threads to execute.

A thread may suspend itself (or be suspended) for an indefinite period by calling

xos_thread_suspend(). If a thread is suspended unconditionally via xos_thread_suspend(), the only way to resume it is by calling xos_thread_resume(). Obviously, a thread suspended in such a way cannot resume itself. Both suspend and resume can be called from an interrupt handler. A thread may also be suspended when calling a blocking function to acquire a resource or wait for an event. These cases are discussed in detail in later sections.

6.4 Priorities and Preemption

A thread's priority can be changed while it is running by calling xos_thread_set_priority(). This can be called by the thread itself or from another thread, or from an interrupt handler. A change in priority can lead to an immediate context switch if a new thread now becomes higher priority than the currently executing thread. A thread's current priority is obtained by calling xos_thread_get_priority().

Thread preemption can be temporarily disabled by calling xos_preemption_disable(). This stays in effect until xos_preemption_enable() is called. The calls can be nested, so multiple calls to xos_preemption_disable() will require an equal number of calls to xos_preemption_enable() to restore preemption. The exception is if a thread exits or encounters a fatal error while preemption is disabled; in that case preemption is enabled by the scheduler.

Table 6.1: Thread Functions

API Call	Description
xos_start()	Init XOS and start multithreading.
xos_start_main()	Init XOS and convert main() into a thread.
xos_thread_create()	Create a new thread.
xos_threadp_set_cp_mask()	Set the coprocessor mask for a thread, in thread creation parameters.
xos_threadp_set_preemption_priority()	Set the preemption priority for a thread, in thread creation parameters.
xos_threadp_set_exit_handler()	Set the exit handler function for a thread, in thread creation parameters.
xos_thread_delete()	Destroy the thread, free up resources.
xos_thread_abort()	Force the thread to terminate.
xos_thread_exit()	Exit the current thread.
xos_thread_join()	Wait for a specified thread to terminate.
xos_thread_yield()	Yield the CPU.

xos_thread_suspend()	Suspend the specified thread.
xos_thread_resume()	Resume (make ready) the specified thread.
xos_thread_get_priority()	Get current priority of thread.
xos_thread_set_priority()	Set current priority of thread.
xos_thread_set_exit_handler()	Set exit handler function for thread.
xos_thread_id()	Return ID (handle) of current thread.
xos_thread_get_name()	Return thread name.
xos_thread_set_name()	Set thread name.
xos_thread_cp_mask()	Get list of CPs that this thread can touch.
xos_thread_get_wake_value()	Get last wake value for the thread.
xos_thread_get_event_bits()	Get current value of event bits for the thread.
xos_thread_get_state()	Get the state of the thread.
xos_preemption_disable()	Disable thread preemption.
xos_preemption_enable()	Re-enable thread preemption.
xos_thread_get_stats()	Get runtime statistics for the thread.
xos_get_cpu_load()	Calculate and return the CPU use % for all threads in the system.

7. Using Timers

The timer module provides time-related services for threads, such as timed delays and timer callback functions. It provides the default preemption source, as the timer interrupt is used to trigger scheduling and time slicing. The timer module also provides the low level support for runtime statistics services. To enable timer support, the hardware must be configured with the timer option, and at least one timer must be present with the timer's interrupt at level <= EXCM_LEVEL. The timer module must be initialized during startup by calling xos_start_system_timer(). This can be called before or after multitasking has been started. The timer ID and the tick interval are specified as arguments to this function.

7.1 Selecting the System Timer

The Xtensa processor can be configured with up to three internal timers. XOS can use any internal timer whose interrupt is at a priority level <= EXCM_LEVEL. The application can explicitly specify which timer XOS should use (timer ID 0, 1 or 2), or it can allow XOS to automatically select the first suitable timer by specifying a timer ID of -1. In the latter case, XOS will select the highest priority timer that meets its requirements. The ID of the selected timer can be found by calling xos_get_system_timer_num().

XOS also supports the use of an external timer. This requires some customization. See Customizing Timer Handling And Using External Timers for details of the steps required to utilize an external timer.

7.2 Using Dynamic Tick Interval

XOS is designed to support timing services without a fixed-interval timer tick. This dynamic tick support ("tickless operation") can be enabled at runtime by setting the tick period to zero in the call to xos_start_system_timer(). Fixed-interval timer ticks are also supported. The choice of which mode to use is dictated by the needs of the application. For a system which spends a lot of time in idle mode, or does not use frequent timer events, the dynamic tick mode can help reduce timer overhead and also save power during idle periods by waking up the CPU less frequently. For a system that runs more or less continuously or requires frequent timer events, a fixed-period tick may be better.

Dynamic tick mode allows thread sleeps and wait timeouts to be performed with improved granularity and variance. For example, with a fixed tick interval of 5 msec a thread cannot be made to sleep consistently for 2 msec. It will be woken only on a tick boundary, and end up sleeping between 2 and (2+5) msec. With dynamic ticking the thread can be made to sleep for 2 msec with much less variance; the next tick can be scheduled exactly 2 msec out. Note that the

interval may not be exactly 2 msec: the timer interrupt handler may be delayed by other higher priority interrupts, cache misses, etc. The timer handler itself has a small additional overhead. Also, to avoid too-frequent timer activity, events are clubbed together if they are less than 10,000 cycles apart (they are handled on the same timer tick). At a clock frequency of 500 MHz this can cause a variance of up to 20 usec.

Any change in thread states caused by timer activity will trigger a scheduling event. For example if a thread is woken from sleep by the tick handler, if its priority is higher than that of the currently active thread, it will be scheduled to run right away (on return from the interrupt). If a wait timeout expires and a thread becomes unblocked, the same rule applies.

7.3 Keeping Track of Time

The system tick count is maintained as a 64-bit counter. A 32-bit counter is not large enough to cover possible use cases. For example, at a 1 millisecond tick rate, a 32-bit counter will overflow after approximately 50 days. The tick count does not correlate with elapsed time if dynamic ticks are enabled. However, XOS also keeps track of the CPU cycle count, which is used to compute elapsed real time so that is not affected by dynamic ticking. The current cycle count may be retrieved by calling xos_get_system_cycles(). The elapsed time may be read and set by calling gettimeofday() and settimeofday(). These two functions are related to the C runtime library. See the C runtime library documentation for details of usage.

All internal time tracking is done in terms of CPU clock cycles. Since 32-bit counters will likely overflow very quickly at high clock rates, XOS uses 64-bit counters to track time internally. The performance penalty due to 64-bit arithmetic is negligible. The CPU clock frequency is a property of the platform and cannot be detected by XOS. There needs to be some platform-specific code to detect the clock frequency and inform XOS via xos_set_clock_freq(). This is required so that XOS can convert cycle counts to real time. It is also possible to hardcode the clock frequency at build time by defining XOS_CLOCK_FREQ appropriately in xos_params.h. Dynamic clock frequency changes at runtime are currently not supported by XOS.

Keeping track of elapsed time becomes more complex if the CPU core can be powered down. During the power down interval, the internal timer does not count, so the CPU cycle count can no longer be used to compute elapsed time. An external timer that continues to count can be used for such cases. In a future version, XOS will provide a way for the application to inform it of the duration of the power down interval.

7.4 Using Time Functions

Timer functionality is provided through XOS timer objects. A timer object must be initialized by calling xos_timer_init() before first use. A timer object can be started and stopped at any time by calling xos_timer_start() and xos_timer_stop(). Timers may be one-shot or periodic. The function xos_timer_get_period() can be used to query the repetition period for a periodic timer,

and xos_timer_set_period() can be used to change the period, or to convert a one-shot timer into a periodic timer (and vice versa).

Calling xos_timer_restart() will reset and restart the timer. As an example, you might set up a timer with a 2-second expiry to detect that something timed out. Every time the 'something' responds, you call the restart function to push out the expiry another 2 seconds. As long as responses keep coming in on time, the timer never expires. Notice that calling xos_timer_restart() with a new timeout value will change the period of a periodic timer. If you want to restart the timer but keep the same period, then use xos_timer_reset(&timer, xos_timer_get_period(&timer)).

An optional callback function may be provided to the timer. When the timer expires, the callback function is invoked. The callback function is invoked from an interrupt handler, and therefore does not run in the context of the thread that requested the callback. The callback runs using the system interrupt stack. Timers can also be waited upon. A thread can call xos_timer_wait() to block itself until the timer expires or is cancelled. There is a small space and time penalty for using the wait feature. This feature can be disabled by setting XOS_OPT_TIMER_WAIT to 0 in xos_params.h.

The xos_thread_sleep() group of functions can be used to suspend a thread for a specified duration. The duration can be specified either in cycles or in time units.

Table 7.1: Time Functions

API Call	Description
xos_set_clock_freq()	Set system clock frequency.
xos_get_clock_freq()	Get current system clock frequency.
xos_start_system_timer()	Initialize timer support and start system timer.
xos_get_system_timer_num()	Get the ID of the system timer.
xos_timer_init()	Initialize a timer object.
xos_timer_start()	Start the timer object.
xos_timer_stop()	Stop the timer object.
xos_timer_restart()	Restart timer with new duration/period.
xos_timer_is_active()	Check if the timer is active.
xos_timer_get_period()	Get the repetition period for a periodic timer.
xos_timer_set_period()	Set the repetition period for a periodic timer.
xos_get_system_cycles()	Get current system cycle count.
xos_thread_sleep()	Suspend thread for specified number of CPU cycles.

xos_thread_sleep_msec()	Suspend thread for specified number of msec.
xos_thread_sleep_usec()	Suspend thread for specified number of usec.
xos_timer_wait()	Block calling thread on timer until it expires.
gettimeofday()	Get time and timezone information.
settimeofday()	Set time and timezone information.

8. Mutexes

Mutexes are program objects that allow multiple threads to negotiate exclusive access to a shared resource. They are used to allow only one thread at a time to access the shared resource. They can also be used as locks to serialize access to critical portions of code. XOS mutexes have an associated count in order to support recursive locking. The mutexes can be waited upon by multiple threads concurrently. Mutexes can be locked and unlocked from interrupt handlers. They cannot be waited upon by interrupt handlers, since interrupt code is not allowed to make blocking calls. The mutex module is optional and is included only if required.

XOS supports priority inversion protection for mutexes. Priority inversion refers to a condition where a higher-priority thread (H) waits on a lower-priority thread (L) because the lower-priority thread owns a mutex that the higher-priority thread needs. Since the thread L can be preempted by other threads at priorities higher than L but lower than H, thread H can suffer a potentially unbounded delay. Priority inversion can be protected against by enabling the configuration option XOS_OPT_MUTEX_PRIORITY (this option is enabled by default). Enabling this option provides two mechanisms: priority ceiling and priority inheritance. The priority ceiling for a mutex can be specified when the mutex is created. Specifying a priority of zero disables the priority ceiling. Priority inheritance is automatic when the configuration option is enabled. For a more detailed discussion see Preventing Priority Inversion.

A mutex must be initialized by calling xos_mutex_create() before it can be used. Initialization leaves the mutex in the unlocked state. The lock count is tracked by a signed 32-bit counter, so a maximum of INT32_MAX recursive lock calls are possible. Overflow checking is done only in debug mode.

A thread can attempt to lock a mutex by using one of three API calls.

- A call to xos_mutex_lock() will block indefinitely until the lock becomes available. It returns only after successfully owning the lock.
- A call to xos_mutex_lock_timeout() will block until the mutex is available or the timeout expires. The timeout function is larger and slower, but some applications require it.
- A non-blocking option is xos_mutex_trylock(). This call returns immediately and indicates success or failure. This call can be used from an interrupt handler.

The call xos_mutex_test() checks the state of the mutex and returns, returning 0 if the mutex is unlocked and 1 if otherwise. This call does not block, and does not attempt to lock the mutex. Keep in mind that xos_mutex_test() only provides a snapshot of the mutex state. Just because it returned 0, does not mean that a call to a lock function will not block. Another thread or threads may execute a lock on the mutex between calling xos_mutex_test() and xos_mutex_lock().

A call to xos_mutex_unlock() will unlock the mutex. This can be called from a thread or an interrupt handler. Unlocking succeeds only if the mutex is owned by the calling thread. If called from an interrupt handler, the mutex must be owned by the thread preempted by the handler. A

successful unlock operation always decrements the lock count by 1. Since the mutexes support recursive locking, an equal number of lock and unlock operations are required for the lock to become free. If the mutex becomes free (the lock count goes to 0) and there are threads waiting on the mutex, then exactly one thread will be woken up by the unlock operation. If there are multiple waiters, the thread that gets woken up is determined by a mutex attribute specified at creation time. If the mutex was created with the flag XOS_MUTEX_WAIT_PRIORITY, then the highest priority thread will be woken. If it was created with the flag XOS_MUTEX_WAIT_FIFO, then the threads will be woken in the order in which they blocked on the mutex. Note that if there are multiple threads at the same priority, then XOS_MUTEX_WAIT_PRIORITY does not guarantee any specific order in which they are woken.

Calling xos_mutex_delete() will destroy a mutex object. The mutex must either be in the unlocked state or be owned by the thread that calls xos_mutex_delete(). If there are any threads blocked waiting for the mutex, they will be woken up with an error code indicating that the object they were waiting on has been deleted.

Table 8.1: Mutex Functions

API Call	Description
xos_mutex_create()	Create a new mutex object.
xos_mutex_delete()	Delete a mutex object, unblock waiting threads.
xos_mutex_lock()	Lock the mutex, wait until available.
xos_mutex_lock_timeout()	Lock the mutex, wait until available or timeout expires.
xos_mutex_unlock()	Unlock the mutex, wake one waiting thread (if any).
xos_mutex_trylock()	Try to lock mutex but return immediately if failed.
xos_mutex_test()	Check state of mutex.

9. Semaphores

Semaphores are constructs that can be used to control access to a common resource from multiple threads. They can also be used to synchronize execution between multiple threads, or between an interrupt handler and a thread. Semaphores have an associated count that controls the degree of access to the shared resource. XOS semaphores can be both waited upon and signaled by multiple threads concurrently. Semaphores can be signaled from interrupt handlers. They cannot be waited upon from interrupt handlers, since interrupt code is not allowed to make blocking calls. The semaphore module is optional and is included only if required.

A semaphore must be initialized by calling xos_sem_create() before it can be used. The semaphore's initial count can be set to any desired value during initialization. If the initial count is set to N, then N get operations will succeed before the consumer thread is blocked on a get request (if no put operations are done during this time). A semaphore's count is tracked by an unsigned 32-bit counter, so a maximum count of UINT32_MAX is possible. There is no overflow checking as this maximum is expected to be far more than will ever be used in actual applications.

A thread can wait on a semaphore using one of three API calls. A call to xos_sem_get() will block indefinitely until the semaphore is signaled. A call to xos_sem_get_timeout() will block until the semaphore is signaled or the timeout expires. The timeout function is larger and somewhat slower, but some applications require it. A non-blocking option is xos_sem_tryget(). This call will return immediately and indicate success or failure. This can be used from an interrupt handler. The call xos_sem_test() checks the state of the semaphore and returns, returning the semaphore count (0 means the semaphore is not signaled). This call does not block, and does not decrement the semaphore. Keep in mind that xos_sem_test() only provides a snapshot of the semaphore's state. Just because it returned a nonzero count does not mean that a call to a get function will not block. Another thread or threads may execute a get on the semaphore between calling xos_sem_test() and xos_sem_get().

A call to xos_sem_put() will signal the semaphore. This can be called from a thread or an interrupt handler. A put operation will always increment the count by 1. If there are threads waiting on the semaphore, a put operation will wake up exactly one thread. If there are multiple waiters, the thread that gets woken up is determined by a semaphore attribute specified at creation time. If the semaphore was created with the flag XOS_SEM_WAIT_PRIORITY, then the highest priority thread will be woken. If it was created with the flag XOS_SEM_WAIT_FIFO, then the threads will be woken in the order in which they blocked on the semaphore. Note that if there are multiple threads at the same priority, then XOS_SEM_WAIT_PRIORITY does not guarantee any specific order in which they are woken.

xos_sem_put_max() works just like xos_sem_put(), except that the semaphore is incremented only if the specified maximum count will not be exceeded.

Calling xos_sem_delete() will destroy a semaphore object. If there are any threads blocked waiting for the semaphore, they will be woken up with an error code indicating that the object they were waiting on has been deleted.

Table 9.1: Semaphore Functions

API Call	Description
xos_sem_create()	Create the semaphore and specify its properties.
xos_sem_delete()	Delete the semaphore. Will unblock all waiting threads.
xos_sem_get()	Decrement the semaphore count or block until able to do so.
xos_sem_get_timeout()	Like xos_sem_get(), except that a timeout can be specified for the wait.
xos_sem_put()	Signal the semaphore (increment count).
xos_sem_put_max()	Signal the semaphore only if the specified max count is not exceeded.
xos_sem_tryget()	Try to decrement the semaphore count but return immediately if failed.
xos_sem_test()	Check the value of the semaphore, but don't attempt to decrement it.

10. Events

An event is a group of bits that can be set, cleared, and waited upon in various combinations. Events can be used for synchronization between threads, or between threads and interrupt handlers. XOS event objects can be both waited upon and signaled by multiple threads concurrently. Event objects can be signaled from interrupt handlers. They cannot be waited upon from interrupt handlers, as interrupt code is not allowed to make blocking calls.

An event object must be initialized by calling xos_event_create() before first use. It can contain up to 32 bits of state. The active bits of an event are specified in the call to xos_event_create(). Only the active bits can be signaled or waited upon. For example, an event can be created with only the four least significant bits active.

A thread (or interrupt handler) can set or clear any combination of active bits in the event, by calling xos_event_set() and xos_event_clear(). It is possible to set some bits and clear others at the same time by calling xos_event_clear_and_set().

A thread can wait on an event by specifying a nonzero bitmask. The bitmask can be applied in ALL mode by calling xos_event_wait_all(), or in ANY mode by calling xos_event_wait_any(). In ALL mode, all the bits of the bitmask must match to satisfy the wait. In ANY mode, any of the bits matching will satisfy the wait. Event bits are sticky, so if the bits are set before a thread waits on them, then the wait will be immediately satisfied. Event bits have no count capability – if a bit is set multiple times and then cleared once, it will still be cleared to zero. Multiple threads can wait on the same set of bits. They will all be woken when the wait condition is satisfied. Both wait functions can accept an optional timeout parameter. If a timeout is specified, then the wait will be canceled if the timeout expires before the required bits have been signaled. Timeout processing imposes some overhead, so this feature is enabled only if the option XOS_OPT_WAIT_TIMEOUT is turned on.

Event bits can be read without blocking by calling xos_event_get(). This call returns a snapshot of the state of the event. This can be used as a way to test event bits from interrupt code, which cannot make blocking calls.

The function xos_event_set_and_wait() can be used to atomically set a specified set of bits and wait on another set of bits. This function operates in ALL mode, requiring all the bits of the wait set to be signaled before the wait is satisfied.

If the XOS_EVENT_AUTO_CLEAR flag is specified during event creation, then the event object automatically clears the bits used to satisfy a thread executing a blocking wait. This happens whether the wait actually blocks or not. Note that only those bits which actually signal a thread are cleared. For example, if a thread was waiting on the bitmask 0xE, and the signaling thread set the bits to 0xF, then after waking the waiter thread, the bits corresponding to 0xE will be cleared, but bit 0 will be left as set, since it did not participate in waking the thread. Nonblocking reads of the event state do not clear any bits.

When a thread reads the state of an event, a copy of the event's bits is stored in the thread's TCB. This copy may be accessed by calling xos_thread_get_event_bits(). This value is updated

every time the thread makes a blocking call on an event, even if the call does not actually block. This can be useful in cases where the thread is woken by an auto-clear event. Even though the event bits get cleared when the thread is woken, the copy in the thread's TCB reflects the state of the bits before the auto-clear operation. Thus it is possible to detect exactly which bits woke the thread, in cases where the ANY wait mode was used. This value is not updated if the wait times out. It is also not updated by a call to xos_event_get().

NOTE: It is good practice to avoid cases where multiple threads wait on some set of partially overlapping bits. For example, say thread 1 waits on bits A and B, and thread 2 waits on bits B and C. If bits A and B get signaled, thread 1 can wake up and clear both bits. Later, if bit C gets signaled, thread 2 will not be woken up, since bit B is now not set.

NOTE: It is important to remember that events have no counting capability. So they should not be used in situations where the "writer" can potentially signal the event more than once while the "reader" does a single wait, and the number of signals is important. In such cases, a mechanism such as a counting semaphore should be used.

Table 10.1: Event Functions

API Call	Description
xos_event_create()	Create the event object. Specify the group of valid bits.
xos_event_delete()	Destroy an event object. Will unblock all waiting threads.
xos_event_set()	Set the specified group of bits.
xos_event_clear()	Clear the specified group of bits.
xos_event_clear_and_set()	Clear one group and set another group, as one action. The groups may overlap.
xos_event_get()	Read the state of the event bits without blocking.
xos_event_wait_all()	Wait for all of a group of bits to be set.
xos_event_wait_all_timeout()	Like xos_event_wait_all(), except that a timeout can be specified for the wait.
xos_event_wait_any()	Wait for any of a group of events to be set.
xos_event_wait_any_timeout()	Like xos_event_wait_any(), except that a timeout can be specified for the wait.
xos_event_set_and_wait()	Atomically set a group of bits and wait on another group of bits.

11. Condition Objects

Condition objects (or condition variables) allow threads to block and wait for a specific condition to become true. The condition is evaluated by a supplied condition function, and the evaluation is performed every time the condition object is signaled by another thread or by an interrupt handler. Whenever the evaluation function returns true, the thread is moved into the ready state. If no condition function is specified, then the thread will be made ready the first time that the condition is signaled after it has blocked. Note that ordering is important: if a condition is signaled before a thread starts waiting on it, then that signal has no effect on this waiting thread. This is unlike semaphores, for example.

Multiple threads may wait on the same condition object with each having supplied a different condition evaluation function, or none. When the condition is signaled, it is supplied a value and this value is passed on to each evaluation function, along with an optional argument that can be specified during the wait call.

Condition objects are initialized by calling xos_cond_create(). A wait on a condition object is performed by calling xos_cond_wait(). Since this is a call that may block, it is not allowed from interrupt handlers. The call to xos_cond_wait() can optionally specify an evaluation function and one optional argument for this function. A condition is signaled by calling xos_cond_signal() with a signal value. The value will be passed to all available evaluation functions in the wait queue of the condition. All the waiting threads that satisfy the wake criteria will be unblocked and made ready. Note that while the condition wait queue does have priority ordering, the selection of threads to be woken is dependent on the condition evaluation. If multiple threads are woken, the next one to be scheduled will be the one at highest priority. While xos_cond_signal() may wake any number of threads, xos_cond_signal_one() will wake at most one thread. It scans the wait queue until a waiting thread is found that can be woken.

An atomic release of a mutex followed by a wait on a condition can be accomplished by calling xos_cond_wait_mutex(). The mutex passed to this function must have been locked by the calling thread. The function releases the mutex, waits on the condition, and re-locks the mutex before returning. xos_cond_wait_mutex_timeout() does the same, except that a timeout for the condition wait can be specified. These functions, together with the signal functions, make it easier to emulate POSIX pthreads condition variables.

A condition object is destroyed by calling xos_cond_delete(). This call will wake up all waiting threads, with an error code indicating that the object they were waiting on has been deleted.

Table 11.1: Condition Functions

API Call	Description
xos_cond_create()	Initialize the condition object.
xos_cond_delete()	Destroy the condition object. Any waiting threads will be unblocked.
xos_cond_wait()	Wait on the condition for it to become true.
xos_cond_signal()	Signal the condition, wake all waiting threads.
xos_cond_signal_one()	Signal the condition, wake one waiting thread.
xos_cond_wait_mutex()	Atomically release mutex and wait on condition.
xos_cond_wait_mutex_timeout()	As above, except a timeout can be specified for the wait.

12. Message Queues

The message queue module implements a multi-writer multi-reader queue. It is completely thread-safe and can be used by interrupt handlers. Messages are copied into the queue so the messages can be freed or reused as soon as the API calls return. The queue contains storage for a fixed number of messages, this number being defined at queue creation time.

It is important to remember that even though the messages are copied into the queue, any data that they may refer to is not copied. For example, if a message is a pointer to a shared data structure, then that data structure must not be altered by the sender until the receiver has had a chance to read the message and access the data. A common use of message queues is to share data buffers by sending the buffer pointer as a message. In such cases, synchronization can be maintained by using a return queue to return the buffer pointers after they have been consumed by the receiver. Once the sender receives a pointer back, it knows that it is safe to reuse the buffer pointed to.

The queue is designed to support use from interrupt handlers. Interrupt handlers are guaranteed not to block when trying to send or receive a message. Calls to get or put a message will return immediately if there are no waiting messages or if the queue is full. One common use case is for a device to receive data and generate an interrupt. The interrupt handler copies the data into a memory buffer and then sends a message to a user thread to further process the data. If the message queue is properly sized, then the interrupt handler should always be able to put a message into the queue.

A message queue is created by calling xos_msgq_create(). Storage for the queue must be allocated by the application. There are two macros that can be used for allocating storage. The XOS_MSGQ_ALLOC macro allocates a static queue, while the XOS_MSGQ_SIZE macro computes the number of bytes to be allocated for dynamic allocation. Messages must be a multiple of four bytes long (padded if necessary) and the message buffers must be 4-byte aligned. Since the messages are copied, the put/get operations timings are dependent on the length of the message. At queue creation time, it is possible to specify whether waiting threads will be woken in FIFO order or priority order.

A message is written into a queue by calling xos_msgq_put(). This call can block if the queue is full. However, if called from an interrupt handler, this function will return immediately even if it fails to put the message into the queue. Calling xos_msgq_put_timeout() will block if needed, but only until the timeout expires. This is intended for use in threads that cannot afford to block indefinitely. Putting a message into a queue will wake up one thread if there are threads waiting on the queue. If the message queue was created with the flag XOS_MSGQ_WAIT_PRIORITY, then the highest priority waiting thread will be woken. If it was created with the flag XOS_MSGQ_WAIT_FIFO, then threads will be woken in the order in which they blocked on the queue.

Messages are read by calling either xos_msgq_get() or xos_msgq_get_timeout(). A read operation can also wake up a thread if there was one waiting to write into the queue. The wake processing is the same as that described above for writing messages. Both of these

calls can block if called from a thread, but will never block if called from an interrupt handler. Another way to check the state of the queue without blocking is to call either xos_msgq_full() or xos_msgq_empty(). Both these functions will return true if the corresponding condition is satisfied. Keep in mind that these two calls just check the instantaneous state of the queue, and the queue state can change at any time.

A queue is destroyed by calling xos_msgq_delete(). This call will wake up any waiting threads with an error code indicating that the queue has been destroyed. After the queue is destroyed, the memory allocated to it can be freed up or used for other purposes.

Table 12.1: Message Queue Functions

API Call	Description
xos_msgq_create()	Initialize a message queue.
xos_msgq_delete()	Delete a message queue.
xos_msgq_put()	Put a message into the specified queue, wait until space is available.
xos_msgq_put_timeout()	Like xos_msgq_put(), except that a timeout can be specified for the wait.
xos_msgq_get()	Get a message from the specified queue, wait until a message is available.
xos_msgq_get_timeout()	Like xos_msgq_get(), except that a timeout can be specified for the wait.
xos_msgq_empty()	Check if the message queue is empty.
xos_msgq_full()	Check if the message queue is full.

13. Block Memory Pools

A block memory pool object manages a collection of fixed-size blocks of memory. These blocks can be individually allocated and freed by any thread. Interrupt code can also allocate and free blocks, subject to some restrictions. Block memory pools are global and can be accessed by any thread in the system regardless of which thread created it. The pool object is thread safe so no external synchronization is needed to access it. The pool is also waitable, so a thread can wait if there are no free blocks available.

The block memory pool has been designed to be lightweight and low overhead, with fast allocate and free operations. Both allocate and free run in constant time independent of the block size and pool size. Blocks are allocated in LIFO order, that is, the most recently freed block will be allocated first. This can help with cache utilization. There is no extra memory overhead for block management other than the pool object itself. A semaphore is used internally to implement thread waits. Thus waiting threads can be in FIFO or priority order.

There is no limitation on the number of memory pools that can be created. One instance of an XosBlockPool object is required for each pool, and this object need not reside adjacent to or even in the same memory region as the pool memory. For instance a pool created from external memory could be managed by a pool object residing in local data memory for fast access.

A block memory pool is initialized by calling xos_block_pool_init(). There is no restriction on the block size other than that it has to be a multiple of 4 bytes. Block allocations are performed by calling xos_block_alloc(). This function cannot be called from interrupt context since it can suspend the caller. For nonblocking allocation attempts, xos_block_try_alloc() must be called. This function returns immediately even if it fails to allocate memory. A block of memory is freed by calling xos_block_free(). The internal consistency of a block pool can be checked by calling xos_block_pool_check(). This function will perform a check only if XOS_BLOCKMEM_DEBUG has been defined to be nonzero.

Table 13.1: Block Memory Pool Functions

API Call	Description
xos_block_pool_init()	Initialize a block memory pool.
xos_block_alloc()	Allocate a block of memory, wait until available.
xos_block_try_alloc()	Try to allocate a block of memory, does not wait.
xos_block_free()	Free an allocated memory block.
xos_block_pool_check()	Check internal consistency of memory pool.

14. Tracking Time With Stopwatches

A stopwatch object allows tracking of elapsed time and time intervals. It is designed such that operations upon it are very lightweight and add minimal overhead to time critical activities. The stopwatch object tracks time in terms of CPU cycles. This can be converted into time units by using the helper functions provided in the timer module. The stopwatch module requires the presence of the system timer function to keep track of the system cycle count. Memory for the stopwatch objects must be allocated by the application.

A stopwatch object is initialized by calling xos_stopwatch_init(). Once initialized, it can be started at any time by calling xos_stopwatch_start(), and stopped by calling xos_stopwatch_stop(). The stopwatch accumulates the elapsed cycles between every start/stop pair. Thus, you can repeatedly start and stop the stopwatch to accumulate the time spent over multiple operations, or multiple repetitions of the same operation. The accumulated count can be read by calling xos_stopwatch_count(). A stopwatch may also be used to check how much time has elapsed from a certain point in time. Call xos_stopwatch_start() on the stopwatch at the desired point in time. Then, call xos_stopwatch_elapsed() to determine how much time has elapsed since then. Calling xos_stopwatch_clear() deactivates the stopwatch, if necessary, and clears it.

Stopwatch operations are not thread-safe, so a single stopwatch object should not be used from multiple threads unless mutual exclusion is otherwise guaranteed by the design of the application.

The stopwatch cycle counter is an unsigned 64-bit quantity, so it can handle very large counts and periods. The cycle counts can be converted to elapsed time by using the timer helper functions such as xos_cycles_to_secs().

Table 14.1: Stopwatch Functions

API Call	Description
xos_stopwatch_init()	Initialize a stopwatch.
xos_stopwatch_start()	Activate a stopwatch.
xos_stopwatch_stop()	Deactivate a stopwatch and update accumulated count.
xos_stopwatch_count()	Get stopwatch accumulated count.
xos_stopwatch_elapsed()	Get elapsed time since the stopwatch was started.
xos_stopwatch_clear()	Clear the stopwatch and deactivate it if needed.

15. System Log

The system log is designed to be a very lightweight mechanism to record events for later examination. The overhead of writing an entry in the system log is small enough that this can be done even from speed-critical code. There is only a single system log, and it is not created by default. If it is required, then memory for the log must be allocated by the application. The log module does not impose any memory or performance penalty if not used by the application.

System log entries are of fixed size. Each entry contains a timestamp (in cycles) and two words of user-specified data. The log is enabled when created and wraps around when full so that only the most recent log entries are retained. API calls are provided to walk through the log and examine each entry.

System log entries contain a 32-bit timestamp instead of a full 64-bit timestamp as used elsewhere in the system. This is done to save both space and time. Since the log entries are ordered sequentially, it is easy to detect counter rollover and compensate for it.

The system log is created by calling xos_syslog_init(). This function must be given a pointer to the memory allocated for the log, and a count of the number of entries. The macro XOS_SYSLOG_SIZE helps compute the space required for a given number of entries. Memory for the log may be allocated statically or dynamically. The log is automatically enabled on initialization. Logging can be disabled by calling xos_syslog_disable(), and re-enabled by calling xos_syslog_enable(). A log entry is written by calling xos_syslog_write(). This call automatically advances the write pointer to the next entry, and wraps around to the beginning of the log if it reaches the end. If the log has not been created or is disabled, then xos_syslog_write() will do nothing.

To begin reading the log contents, xos_syslog_get_first() must be called to read out the first entry. Then xos_syslog_get_next() must be called to read subsequent entries until the end of the log is reached. It is recommended that the log be disabled while it is being read out. The log can be cleared by calling xos_syslog_clear(). This clears all written entries and sets the write pointer to the start of the log.

Table 15.1: System Log Functions

API Call	Description
xos_syslog_init()	Initialize the system log.
xos_syslog_clear()	Clear the system log.
xos_syslog_disable()	Disable logging to the system log.
xos_syslog_enable()	Enable logging to the system log.

xos_syslog_write()	Write an entry into the syslog.
xos_syslog_get_first()	Read the first (oldest) entry in the system log.
xos_syslog_get_next()	Read the next entry in the system log.

16. Tracking Runtime Statistics

XOS is capable of tracking runtime statistics when so configured. The build option XOS_OPT_STATS controls the enabling of runtime statistics collection. If this is enabled, there is a small performance penalty and code size increase. If not enabled, there is no code size or performance impact on the system. In this release, the following statistics per thread are tracked:

- The cycle count: the number of CPU cycles used by the thread. This number is not exact but an approximation, because tracking the exact cycle count is not feasible. It includes time spent in XOS calls made from the thread. It also includes time spent in high level interrupt handlers that have preempted this thread. During context switch, XOS attempts to allocate the time spent in saving the outgoing thread's context to the outgoing thread, and the time spent in restoring the incoming thread's context to the incoming thread.
- The number of preemptive context switches: the number of times this thread was preempted and resumed.
- The number of cooperative context switches: the number of times this thread was blocked and resumed. A thread may block because a resource is unavailable (e.g. mutex), because it sleeps (timer) or because it yields the CPU.

The above statistics are also kept for the idle thread. In this case, the number of context switches is not meaningful. The cycle count is a measure of how long the system was idle.

XOS also tracks the number of cycles spent in handling low and medium level interrupts. This is not an exact count, as some cycles spent in saving the CPU context are attributed to the interrupted thread. Cycles spent in restoring the state are also attributed to the thread. There is currently no mechanism to instrument high level interrupt handlers. It is assumed that these handlers are all time-critical and additional instrumentation will slow down their operation to unacceptable levels. Cycles spent in high level interrupt handlers are attributed to whatever thread (or low/medium level interrupt handler) that they happened to interrupt.

Enabling statistics does have some impact on performance. The code size increase is minimal, but interrupt dispatch and context switching both take somewhat longer. The amount of the increase depends on the system configuration.

Per-thread statistics can be read out for a specified thread by calling xos_thread_get_stats(). This function will fill in an XosThreadStats structure. CPU load statistics for the entire system can be retrieved by calling xos_get_cpu_load(). This function returns the statistics for all threads, and also computes the percentage of CPU time used by each thread. Because of this computation, this function can take some time to complete. The idle thread and interrupt processing are also covered by these functions.

In addition, message queues can be enabled to track per-queue statistics by defining the build option XOS_OPT_MSGQ_STATS to be nonzero at build time. This tracks the number of sends

and receives and the number of block events due to calls to send and receive for each queue. Enabling this option increases the size of queue objects.

17. Interrupt and Exception Handling

The interrupt and exception dispatch mechanism in XOS was designed to make dispatching as fast as possible while still retaining flexibility for users to install their own custom handlers. The various interrupt and exception types are handled as follows.

17.1 Interrupts in XEA3

In Xtensa Exception Architecture v3 (XEA3) no distinction is made between low, medium and high priority interrupt levels. Interrupts at all levels up to the highest (NMI) level can be dispatched to handlers written in C. They are allowed to make XOS system calls and can access system objects such as queues, semaphores, etc. However, they are not allowed to make any calls that could block, for example, calling xos_mutex_lock() is not allowed. The scheduler is invoked at the end of interrupt processing just before returning control to user code. At this time, if a new thread has become the highest priority ready thread, a context switch will occur.

Interrupts are dispatched by the XOS dispatcher and the handlers are found from a handler table. Handlers are installed into the handler table by calling xos_register_interrupt_handler(). The dispatcher cannot be replaced without modifying XOS.

As there is no concept of a high priority interrupt any longer, handlers can only be specified per interrupt, not per level. Since all levels can be handled in C, the value of EXCM_LEVEL is set to the highest possible level (NMI level).

This page lists the system calls that cannot be called from interrupt handlers.

17.2 Interrupts in XEA2

In Xtensa Exception Architecture v2 (XEA2) there are two categories of interrupts. High priority interrupts are those at level higher than EXCM_LEVEL. These cannot be handled in C without extra processing and overhead. Low priority (level 1) and medium priority (level <= EXCM_LEVEL) interrupts can be handled in C. EXCM_LEVEL is a user selectable, configuration-specific value.

17.3 High Priority Interrupts (XEA2)

There is a dispatcher for each high priority interrupt level. This dispatcher looks up a table for an application-defined handler at that level and transfers control to it. This handler must return to the dispatcher when finished. An application can install a custom handler for a high priority interrupt

by calling the function xos_register_hp_interrupt_handler(). This approach provides application flexibility and ease of development. However, it does take a few extra cycles. Applications that cannot afford even these few cycles can define their own dispatchers, which will override the default dispatchers at build time. These user dispatchers can be written to be as efficient as possible. If user dispatchers are written to override the defaults, then, of course, any handler installed via xos_register_hp_interrupt_handler() is never called.

17.4 Medium and Low Priority Interrupts (XEA2)

These interrupts are dispatched by the XOS dispatcher and the handlers are found from a handler table. Handlers are installed into the handler table by calling xos_register_interrupt_handler(). The dispatcher for these levels cannot be replaced without modifying XOS.

Low and medium priority interrupt handlers can be written in C. They are allowed to make XOS system calls and can access system objects such as queues, semaphores, etc. However, they are not allowed to make any calls that could block, for example, calling xos_mutex_lock() is not allowed. The scheduler is invoked at the end of interrupt processing just before returning control to user code. At this time, if a new thread has become the highest priority ready thread, a context switch will occur.

This page lists the system calls that cannot be called from interrupt handlers.

17.5 Interrupt Nesting and Software Prioritization

XOS supports nested interrupt handling. While an interrupt is being handled, if another interrupt at a higher priority level is asserted then the higher priority one will be taken immediately, and it will preempt the handling of the lower priority interrupt. For example, while an interrupt at priority level 1 is being handled, all enabled interrupts at level 2 or higher are allowed to be taken. The only exception to this is a brief period of time during interrupt dispatch. During this time all interrupts are disabled while the CPU state is saved, the stack is switched (if necessary) and the interrupt to be handled is selected.

Software prioritization of interrupts is not supported for XEA3. The interrupt controller takes care of interrupt prioritization and fairness. The following discussion is relevant for XEA2 only.

If multiple interrupts at the same priority level are asserted at the same time, then the numerically highest one is handled first. For example, say interrupts numbers 5 and 10, both at priority level 2, are asserted. The dispatch process will save the CPU state, switch to the interrupt stack, and then examine the pending interrupts. It will select the numerically higher interrupt 10 first.

However, if interrupt 10 were to be asserted just after the interrupt selection step above, then it would not be handled until the processing for interrupt 5 was finished. If this behavior is not desired, then software prioritization of interrupts should be enabled via XOS_OPT_INTERRUPT_SWPRI. This option creates virtual priority levels for interrupts at

the same hardware priority level. For the example above, it would assign interrupt 10 a virtual priority level higher than interrupt 5 (but lower than any interrupt at hardware priority level 3). Now if interrupt 10 were to be asserted at any time during the processing of interrupt 5 it would be taken immediately.

The following table illustrates a more complex example.

Interrupt Number	HW Priority Level	SW Priority Level
1	1	1
2	3	3.0
3	2	2.0
4	2	2.1
5	2	2.2
6	3	3.1
7	4	4

When XOS_OPT_INTERRUPT_SWPRI is enabled, interrupt 5 can preempt both interrupts 3 and 4. However it cannot preempt interrupts 2, 6 and 7. Interrupt 6 can preempt interrupt 2, but not interrupt 7.

17.6 Interrupt Stack

XOS uses a separate dedicated stack for interrupt processing. It switches to this stack when taking an interrupt from user code. Further nested interrupts continue to use this stack. All interrupt handlers and functions called by these handlers use the interrupt stack, so the sizing of the interrupt stack is very important. Making the stack excessively large will waste memory, but making it too small will cause the system to fail. The configuration parameter XOS_INT_STACK_SIZE is used to specify the stack size in bytes.

If nested interrupts can happen, then the worst case stack size must take into account the maximum level of interrupt nesting. Using software prioritization increases the effective number of levels, so the stack needs to be larger. The amount of stack to be used by each handler (and the functions it calls) also needs to be taken into account. It is good practice to avoid making deep function calls from interrupt handlers. It is also recommended to not allocate large data structures on the stack in functions called from interrupt handlers.

The interrupt stack must exist even for hardware configurations that do not have any interrupts, because it is also used as the idle thread's stack. When no interrupts are configured, the stack size can be very small. The default is 32 bytes.

17.7 Handling Exceptions

Exceptions are handled very similarly to low and medium priority interrupts. A system-wide exception handler table is maintained by XOS, with one handler for each exception type. At startup, every entry in the table points to the default handler, xos_default_exception_handler. User code can install its own handler for a specific exception by calling xos_register_exception_handler(). However, note that some exceptions cannot be taken over by user handlers. These are - window overflow and underflow, coprocessor, alloca, and (for XEA2) Level 1 interrupt. XOS needs to handle these exceptions itself for proper system operation. User exception handlers can be written in C, as the exception dispatcher sets up the processor state correctly to invoke C code. The handler function is passed a pointer to an XosExcFrame structure that contains information about the exception. If the handler function returns, the instruction that caused the exception will be retried. If the exception condition was not corrected by the handler, then this will cause the exception to be taken again. The handler can of course change the saved PC to point elsewhere.

The XOS default exception handler does not return. If the exception occurred in the context of a thread, then the thread is terminated with a return code of XOS_ERR_UNHANDLED_EXCEPTION. If the thread has an exit handler it will be called, and any threads waiting on the faulting thread will be awakened. If the exception occurred in system context, it is considered a fatal error and the fatal error handler is called.

Per-thread exception handlers are not supported. Any user handlers that are installed must be able to handle being invoked in the context of any thread or in the context of the system. A user handler can be unregistered by calling xos_register_exception_handler() with a null value for the handler address.

The XosExcFrame structure contains the exception type and the exception memory address, if any. It also contains the saved PC, PS, and AR registers for the faulting location. The exception handler can change any of these registers in the exception frame, and the changes will be applied when control is returned to the point of the exception.

18. Understanding Coprocessor State

For each thread, memory must be allocated to save the state of user TIE, along with the state(s) of any coprocessors that the thread uses. This memory is allocated on the thread stack. The stacks must be sized to allow for this extra space. Only those coprocessors that have actually been used by a thread will be saved and restored. However, a thread must have space allocated for saving all the coprocessors in the system, since it can potentially make use of all the coprocessors. The exception to this space allocation requirement is if a thread does not use any coprocessors at all. This exception can be indicated to XOS by the flag XOS_THREAD_NO_CP specified at thread creation time. No space is then reserved for coprocessor state on the stack. Such threads can use a smaller stack size. You cannot specify a subset of coprocessors to be used by a thread, if there is more than one coprocessor in the system. See the Stack Usage section for more details.

State for a coprocessor is saved only when necessary. The necessity is determined by tracking the ownership of each coprocessor. As an example, assume a thread A that uses CP1 and CP4, and a thread B that uses only CP4. When thread A is made running, it uses CP1 and CP4 and this is tracked by the OS. Eventually, A is evicted and other threads run, but when A is evicted, the OS saves the ownership state and disables all coprocessors. So long as no other thread wants to use either of these coprocessors, their state is not saved. If A is scheduled again, it can run and use the coprocessors right away. Now if B is scheduled and wants to use CP4, it will cause a coprocessor exception. The exception handler sees that CP4 is owned by thread A, so it saves the coprocessor state in A's save area and then gives ownership of CP4 to thread B. Since thread B did not have any saved state for CP4, none is restored. Now when thread A gets to run again, it does not possess ownership of CP4, so when it tries to access the coprocessor an exception is again generated. The exception handler this time saves the coprocessor state into B's save area, restores A's old state from A's save area, and gives ownership to thread A.

19. Thread-safe C library

XOS supports the use of a thread-safe C runtime library. This is a user-selectable option. Both newlib and xclib libraries are supported. The per-thread context data for the library is allocated as part of the TCB (if this option is enabled). The thread-safe locking is implemented by the use of mutexes. XOS also provides the _sbrk_r() function to support dynamic allocation from the heap, using C library functions such as malloc(). The default heap area and bounds are determined by linker symbols generated at build time by the standard LSPs. Note that malloc/free etc. will still work if thread safety is not enabled, but it will be the application's responsibiity to make sure that these functions are called in a thread-safe manner.

To enable thread safety, the configuration option XOS_OPT_THREAD_SAFE_CLIB must be set to a nonzero value. In the default configuration, the type of C library present is detected at build time and if either newlib or xclib are detected, this option is automatically enabled. Enabling this option does increase both memory use and code size, and therefore should be considered carefully.

The use of thread-safe libraries requires extra memory in the TCB. There is a very small runtime penalty, which is not expected to be significant. The newlib library requires more memory per thread than xclib. The exact amount of memory can vary from one release of the Xtensa tool chain to another, so no assumptions should be made. If it is known that the application will not call C library functions that maintain internal state from multiple threads, then the single-threaded version of the library can be used. Refer to the C library documentation for details of which functions are in this category.

Note that file I/O operations other than on stdin / stdout / stderr are not supported by XOS. Any other standard input/output operations will have to be supported by other libraries or by user code.

20. Stack Usage

XOS uses the thread stack to store coprocessor / TIE state for the thread while the thread is not running. In addition, an exception frame is created on the active stack when handling exceptions or interrupts. Adequate extra space must be allocated on the stack to avoid stack corruption. XOS defines some constants to make this easier.

- XOS_STACK_EXTRA is the amount of extra space, in bytes, that must be set aside for state saving, if the thread uses coprocessors / TIE resources. You must add this to the amount of stack space needed by the thread for its own purposes to compute the total stack size.
- XOS_STACK_EXTRA_NO_CP is the amount of extra space required if the thread is guaranteed to not use any coprocessors.

The minimum recommended stack sizes (in bytes) are specified by XOS_STACK_MIN_SIZE for threads that use coprocessors and XOS_STACK_MIN_SIZE_NO_CP for threads that will not use coprocessors. These sizes are recommended but not enforced, so you can specify smaller stack sizes if you are sure that they will work for you under all operating conditions. However, XOS does enforce absolute minimum stack sizes. Currently, these minimums are (XOS_STACK_EXTRA + 32) for threads that use coprocessors, and (XOS_STACK_EXTRA_NO_CP + 32) for threads that will not use coprocessors.

It is recommended that development be started by specifying at least XOS_STACK_MIN_SIZE for every thread stack, and optimization for size be done later in the development process. Stack checking should be enabled during development. The xt-stack-usage tool is also helpful in analyzing stack requirements.

21. Managing Memory

XOS does not provide any memory management capability beyond the simple block memory pool. More sophisticated (but slower) management is available in the C runtime library. This provides the standard malloc / free interface. There is one system-wide heap from which all allocation requests are satisfied. For the newlib and xclib libraries, memory allocation and freeing can be made thread-safe. However, the memory allocator does not track ownership of allocated memory by thread.

XOS itself does not depend on the presence of a memory allocator. It is designed to operate without any access to dynamic memory allocation. All internal data required by XOS is statically allocated. User-visible objects (threads, stacks, semaphores, message queues, etc.) must be allocated by the application, either statically or dynamically.

22. Customizing Timer Handling And Using External Timers

XOS provides hooks that allow you to change how the system timer is set up and handled. It also allows you to use an external timer instead of using one of the Xtensa internal timers. You must provide your own custom versions of the following functions to make this work. The default versions of these functions can be used as a reference to understand how they work. Dynamic tick mode is handled internally by XOS, so no extra work is required on your part.

xos_system_timer_select() - XOS calls this function to select which timer to use as the system timer. Your implementation of this function must choose one of the internal timers or an external timer. It must return XOS_OK on success, and return the timer ID in the output parameter 'psel'. The timer ID is one of XOS_SYS_TIMER_0, XOS_SYS_TIMER_1, XOS_SYS_TIMER_2 or XOS_SYS_TIMER_EXTERNAL.

xos_system_timer_init() - XOS calls this function to initialize and start the system timer. Your implementation must take all the steps required for this, including setting up your interrupt handler and enabling the timer interrupt. XOS also provides the start value for the timer in this call, as the number of CPU clock cycles to the first timer interrupt.

xos_system_timer_set() - XOS calls this function to set up the next timer interrupt. It specifies the number of CPU clock cycles to the next timer interrupt. Note that this number may vary even for fixed-period tick mode, as XOS may make adjustments to compensate for internal delays.

Your timer interrupt handler must call xos_tick_handler() on every timer tick. This function handles XOS internal processing, and computes the time to the next tick. It will call xos_system_timer_set() to set up the next tick. Do not set up the next tick in your interrupt handler.

Treat the CPU clock cycle counts as being from the instant that your functions are called. If you are using one of the internal timers, see the default code for how the timer is set up. For an external timer, XOS makes no assumptions about its properties, except that it should be able to handle 32-bit CPU cycle counts. The external timer can be an up or down counter. It can also run at a different frequency, in which case your implementation of the above functions will need to do the appropriate conversion.

The default versions of the above functions are defined as weak symbols, so you do not have to rebuild XOS to substitute your custom implementation. Simply provide your version at link time to override the default.

23. Preventing Priority Inversion

A major problem with using mutexes (binary semaphores) for controlling access to critical resources is unbounded priority inversion. Priority inversion occurs when a low-priority thread owns a mutex, and a high-priority thread is forced to wait on the mutex until the low-priority thread releases it. If, prior to releasing the mutex, the low-priority thread is preempted by one or more mid-priority threads, then unbounded priority inversion has occurred because the delay of the high-priority thread is no longer predictable. In such cases it will not be possible to predict whether the high-priority thread will meet its deadline.

Consider a high-priority thread H, a low-priority thread L, and one or more threads with priorities M1, M2, ... in between H and L. If L acquires a mutex K, and subsequently H tries to acquire it, H will be suspended waiting for K. Since L can be preempted by any of M1, M2 etc. we cannot determine when L will release the mutex, and so we cannot determine when H will resume.

Sharing a critical resource between high and low priority threads is not a desirable design practice. It is better to share a resource only among equal priority threads or to access the resource through a single server thread. However, this is not always practicable and in reality, design situations occur when priority inversion becomes possible. For robust operation, it is best to provide mechanisms to prevent priority inversion. Two popular methods of doing so are priority inheritance and priority ceiling.

23.1 Priority inheritance

Priority inheritance is the process by which a lower-priority thread owning a mutex inherits the priority of the highest-priority thread that is waiting on the same mutex. Since the owner of the mutex is always known, when a thread blocks on the mutex we can check if the blocked thread has a higher priority than the owner thread. If it does, then the priority of the owner thread is raised to be equal to the priority of the blocked thread for as long as it owns the mutex. This guarantees that the owner will not be preempted by any thread at a lower priority than the high-priority blocked thread.

23.2 Priority ceiling

In this method, the shared mutex is assigned a priority ceiling when it is created, and this ceiling is usually set equal to the priority of the highest-priority thread that can own the mutex. Whenever a thread gets to own the mutex, its priority is raised to the mutex's priority ceiling if needed, and stays for as long as it owns the mutex. This ensures that no other mid-priority thread can preempt it, and also ensures that no other thread wanting the mutex can preempt it. This method minimizes the possible number of extra context switches. This method also forces all threads to

be at the same priority while owning a resource, which is good design practice and improves determinism.

If all the mutexes in a group are assigned the same priority ceiling, then once a thread gets to own one mutex in the group, it gets access to the entire group. This is because once a thread owns one mutex, its priority is raised and it cannot be preempted by any thread wanting to acquire a mutex in the same group. Secondly, this thread could not be running if another thread had already owned another mutex from this group.

23.3 XOS implementation

Priority inversion protection in XOS combines both priority ceiling and priority inheritance. If the XOS_OPT_MUTEX_PRIORITY option is enabled, then priority inheritance is automatic. Priority ceiling can be enabled by specifying the ceiling for a mutex when it is created. This allows the most flexible implementation. If mutex priorities are set sufficiently high then priority inheritance is effectively disabled. On the other hand, setting a mutex priority to zero effectively disables priority ceiling for that mutex. A combination of the two can be used if desired to implement correct behavior and minimize the number of extra context switches.

IMPORTANT: The function xos_mutex_lock_timeout() does not implement priority inheritance. The combination of a timed wait and priority elevation can lead to scenarios that are difficult to analyze and verify. Priority ceiling does continue to work.

24. Debugging

You can enable additional consistency and error checking by defining XOS_DEBUG_ALL to be nonzero at build time. Debugging for individual modules can be enabled by defining, for example, XOS_THREAD_DEBUG for the thread module. See the file xos_params.h for the available options. Enabling the debug option carries a penalty in both code size and run time.

When runtime assertions fail, xos_fatal_error() is called. This function will call a user-supplied handler if one has been provided via xos_register_fatal_error_handler(). The default behavior is to print a message (if XOS_DEBUG_ALL is enabled) and halt the system.

Runtime stack checking can be enabled by defining XOS_OPT_STACK_CHECK to be nonzero. This turns on stack overflow checking whenever there is a context switch. It does not prevent stack overflow or detect other kinds of stack corruption.

24.1 XOS debug output

When the XOS library is built with the XOS_DEBUG option enabled, large amounts of debug output may be generated from XOS code. This debug output is enabled by default, and is sent to the standard output device. If you want to suppress this debug output without having to rebuild your executable, set the XOS global variable "xos_debug_print" to zero in your code or via the debugger. Setting this variable to any nonzero value will enable the debug output. Note that this variable exists only when the XOS library has been built with XOS_DEBUG enabled.

24.2 Using libxtutil

When XOS debugging features are enabled, it uses the libxtutil library for formatted output. Thus, this library must also be linked into the application. This library provides functions compatible with the C standard library (such as xt_printf() instead of printf()), although not all features are supported. In particular, xt_printf() does not support floating point formats and some other format options. The libxtutil functions are lightweight versions with much smaller code and data memory requirements and they are also thread-safe. For more details on included functions refer to the file xtensa/xtutil.h.

25. Interrupt Handler Restrictions

Interrupt handlers are subject to the limitations described in the “Handling Interrupts” section of the System Software Reference Manual. In particular, for code that can be executed from interrupt handlers:

- Avoid using floating point types (only relevant here if the FPU option is configured).
- Avoid using XCC vectorization options, such as -LNO:simd.
- Avoid using the XCC option –mcoproc.
- Avoid using intrinsics that might access custom TIE registers and states.

The following functions cannot be called from interrupt handlers:

- xos_start()
- xos_start_main()
- xos_thread_exit()
- xos_thread_join()
- xos_thread_yield()
- xos_thread_sleep()
- xos_get_cpu_load() (not recommended)
- xos_start_system_timer()
- xos_timer_wait()
- xos_cond_wait()
- xos_mutex_lock()
- xos_mutex_lock_timeout()
- xos_sem_get()
- xos_sem_get_timeout()
- xos_event_wait_any()
- xos_event_wait_any_timeout()
- xos_event_wait_all()

- xos_event_wait_all_timeout()
- xos_event_set_and_wait()
- xos_block_alloc()

26. XOS Initialization Sample Code

Here are two small examples to illustrate XOS initialization and startup. The first one illustrates the case when main() is not converted into a thread.

```
#define STACK_SIZE (XOS_STACK_MIN_SIZE + 0x1000)

XosThread thread_tcb;
uint8_t   thread_stack[STACK_SIZE];

int32_t
thread_func(void * arg, int32_t unused)
{
    int32_t count = 0;

    puts("Thread starting.");

    while (1) {
        xos_thread_sleep(1000);
        count++;
        printf("Count = %d\n", count);
    }

    return 0;
}

int
main()
{
    int32_t ret;

    // Set clock frequency before calling xos_start().
    xos_set_clock_freq(XOS_CLOCK_FREQ);

    // Select and start system timer.
    xos_start_system_timer(-1, 0);

    // Create at least one thread before calling xos_start().
    ret = xos_thread_create(&thread_tcb,
                            0,
                            thread_func,
                            0,
                            "demo",
                            thread_stack,
                            STACK_SIZE,
                            7,
                            0,
                            0);

    // Start multitasking.
    xos_start(0);
    // Should never get here.
    return -1;
}
```

This example illustrates the case where main() is converted into a thread.

```
#define STACK_SIZE (XOS_STACK_MIN_SIZE + 0x1000)
```

```
XosThread thread_tcb;
uint8_t   thread_stack[STACK_SIZE];

int32_t
thread_func(void * arg, int32_t unused)
{
    int32_t count = 0;

    puts("Thread starting.");

    while (1) {
        xos_thread_sleep(1000);
        count++;
        printf("Count = %d\n", count);
    }

    return 0;
}

int
main()
{
    int32_t ret;

    // Set clock frequency before calling xos_start_main().
    xos_set_clock_freq(XOS_CLOCK_FREQ);

    // Select and start system timer.
    xos_start_system_timer(-1, 0);

    // Start multitasking.
    xos_start_main("main", 5, 0);

    // Create a thread after control returns.
    ret = xos_thread_create(&thread_tcb,
                            0,
                            thread_func,
                            0,
                            "demo",
                            thread_stack,
                            STACK_SIZE,
                            7,
                            0,
                            0);

    // Do not return from here.
    while (1);

    return 0;
}
```

27. Data Structure Index

27.1 Data Structures

Here are the data structures with brief descriptions:

28. File Index

28.1 File List

Here is a list of all documented files with brief descriptions:

29. Data Structure Documentation

29.1 XosThread Struct Reference

30. File Documentation

30.1 xos.h File Reference

Macros

- #define XOS_VERSION_STRING "2.03"

  XOS version string.
- #define XosExcFrame int

Typedefs

- typedef void(XosIntFunc) (void ∗arg)
- typedef int32_t(XosPrintFunc) (void ∗arg, const char ∗fmt,...)
- typedef void(XosFatalErrFunc) (int32_t errcode, const char ∗errmsg)

Functions

- void xos_fatal_error (int32_t errcode, const char ∗errmsg)
- XosExcHandlerFunc ∗ xos_register_exception_handler (uint32_t exc, XosExcHandlerFunc ∗handler)
- XosFatalErrFunc ∗ xos_register_fatal_error_handler (XosFatalErrFunc ∗handler)
- int32_t xos_register_interrupt_handler (uint32_t num, XosIntFunc ∗handler, void ∗arg)
- int32_t xos_unregister_interrupt_handler (uint32_t num)
- int32_t xos_register_hp_interrupt_handler (uint32_t level, void ∗handler)
- void xos_interrupt_enable (uint32_t intnum)
- bool xos_interrupt_enabled (uint32_t intnum)
- void xos_interrupt_disable (uint32_t intnum)
- static uint32_t xos_get_int_pri_level (void)
- static uint32_t xos_set_int_pri_level (uint32_t level)
- static void xos_restore_int_pri_level (const uint32_t oldval)
- static uint32_t xos_disable_interrupts (void)
- static void xos_restore_interrupts (uint32_t rval)

30.1.1 Macro Definition Documentation

30.1.1.1 #define XosExcFrame int

Exception handler function pointer type.

30.1.2 Typedef Documentation

30.1.2.1 typedef void(XosFatalErrFunc) (int32_t errcode, const char *errmsg)

Fatal error handler function pointer type.

30.1.2.2 typedef void(XosIntFunc) (void *arg)

Interrupt handler function pointer type.

30.1.2.3 typedef int32_t(XosPrintFunc) (void *arg, const char *fmt,...)

Print handler function pointer type.

30.1.3 Function Documentation

30.1.3.1 static uint32_t xos_disable_interrupts (void) [inline], [static]

Disable all interrupts that can interact directly with the OS. This is a convenience function, shorthand for setting the IPL to XOS_MAX_OS_INTLEVEL.

Returns: A value that can be used to restore the previous priority level by calling xos_restore_interrupts(). This value should be treated as opaque by application code, and should be passed unchanged to the restore function.

30.1.3.2 void xos_fatal_error (int32_t *errcode,* const char * *errmsg*)

Reports a fatal error and halts XOS operation, i.e. halts the system. This function will call a user-registered error handler (if one has been set) and then halt the system. The user handler may do system-specific things such as record the error reason in nonvolatile memory etc.

Parameters

errcode	Error code. May be any user defined value $<$ 0. Values $>$=0 are reserved for use by the system.
errmsg	Optional text string describing the error.

Returns

This function does not return.

30.1.3.3 static uint32_t xos_get_int_pri_level (void) [inline], [static]

Get the CPU's current interrupt priority level. Interrupts at or below this priority level are blocked.

Returns

Returns the current IPL, ranging from 0 to XCHAL_NUM_INTLEVELS.

30.1.3.4 void xos_interrupt_disable (uint32_t *intnum*)

Disable a specific individual interrupt, by interrupt number.

This is the counterpart to xos_interrupt_enable(). See the description of xos_interrupt_enable() for further comments and notes.

Parameters

intnum	Interrupt number to disable. Must range between 0-31.

Returns

Returns nothing.

30.1.3.5 void xos_interrupt_enable (uint32_t *intnum*)

Enable a specific interrupt, by interrupt number. The state (enabled vs. disabled) of individual interrupts is global, i.e. not associated with any specific thread. Depending on system options and implementation, this state may be stored in one of two ways:

- directly in the INTENABLE register, or
- in a global variable (this is generally the case when INTENABLE is used not just to control what interrupts are enabled globally, but also for software interrupt prioritization within an interrupt level, effectively providing finer grained levels; in this case XOS takes care to update INTENABLE whenever either the global enabled-state variable or the per-thread fine-grained-level variable change). Thus it is best to never access the INTENABLE register directly.

To modify thread-specific interrupt priority level, use one of:

- xos_set_int_pri_level()
- xos_restore_int_pri_level()
- xos_disable_interrupts()
- xos_restore_interrupts()

NOTE: To refer to a specific external interrupt number (BInterrupt pin), use HAL macro XCHAL_EXTINTx_NUM where 'x' is the external interrupt number. For example, to enable external interrupt 3 (BInterrupt[3]), you can use:

```
xos_interrupt_enable( XCHAL_EXTINT3_NUM );
```

Parameters

intnum	Interrupt number to enable. Must range between 0-31.

Returns

Returns nothing.

30.1.3.6 bool xos_interrupt_enabled (uint32_t *intnum*)

Check whether the specified interrupt is enabled. See the description of xos_interrupt_enable() for further comments and notes.

Parameters

intnum	Interrupt number to query.

Returns

Returns true if the interrupt is enabled, else returns false.

30.1.3.7 XosExcHandlerFunc* xos_register_exception_handler (uint32_t *exc,* XosExcHandlerFunc * *handler*)

Install a user defined exception handler for the specified exception type. This will override the default XOS exception handler. The handler is a C function that is passed one parameter – a pointer to the exception frame. The exception frame is allocated on the stack of the thread that caused the exception, and contains saved state and exception information. For details of the exception frame see the structure XosExcFrame.

Parameters

exc	Exception type (number) to override. The exception numbers are enumerated in <xtensa/corebits.h>.
handler	Pointer to handler function to be installed. To revert to the default handler, pass NULL.

Returns

Returns a pointer to previous handler installed, if any.

30.1.3.8 XosFatalErrFunc* xos_register_fatal_error_handler (XosFatalErrFunc * *handler*)

Install a user defined fatal error handler. This function will be called if a fatal error is reported either by user code or by XOS itself. It will be passed the same arguments that are passed to xos_fatal_error().

The handler need not return. It should make minimal assumptions about the state of the system. In particular, it should not assume that further XOS system calls will succeed.

Parameters

handler	Pointer to handler function to be installed.

Returns

Returns a pointer to previous handler installed, if any.

30.1.3.9 int32_t xos_register_hp_interrupt_handler (uint32_t *level,* void * *handler*)

Register a high priority interrupt handler for interrupt level "level". This function is available only for configurations using XEA2.

Unlike low and medium priority interrupt handlers, high priority handlers are not installed for a specific interrupt number, but for an interrupt level. The level must be above XCHAL_EXCM_LEVEL. The handler function must be written in assembly since C handlers are not supported for levels above XCHAL_EXCM_LEVEL. The handler function must preserve all registers except a0, and must return to the dispatcher via a "ret" instruction, not "rfi".

NOTE: This method of dispatch takes a few cycles of overhead. If you wish to save even these cycles, then you can define your own dispatch function to override the built-in dispatcher. See xos_handlers.S for more details.

Parameters

level	The interrupt level to be handled.
handler	Pointer to handler function.

Returns

Returns XOS_OK if successful, else error code.

30.1.3.10 int32_t xos_register_interrupt_handler (uint32_t *num,* XosIntFunc * *handler,* void * *arg*)

Register a handler function to call when interrupt "num" occurs.

For level-triggered and timer interrupts, the handler function will have to clear the source of the interrupt before returning, to avoid infinitely retaking the interrupt. Edge-triggered and software interrupts are automatically cleared by the OS interrupt dispatcher (see xos_handlers.S).

Parameters

num	Xtensa internal interrupt number (0..31). To refer to a specific external interrupt number (BInterrupt pin), use HAL macro XCHAL_EXTINTx_NUM where 'x' is the external number.
handler	Pointer to handler function.
arg	Argument passed to handler.

Returns

Returns XOS_OK if successful, else error code.

30.1.3.11 static void xos_restore_int_pri_level (const uint32_t *oldval*) [inline], [static]

Restores the CPU to a previously saved interrupt priority level. This level must have been obtained by calling xos_set_int_pri_level().

Parameters

oldval	Return value from xos_set_int_pri_level().

Returns

Returns nothing.

30.1.3.12 static void xos_restore_interrupts (uint32_t *rval*) [inline], [static]

Restore the CPU's previously saved interrupt status. This is a convenience function, the counterpart to xos_disable_interrupts().

Returns

rval Return value from xos_disable_interrupts().
Returns nothing.

30.1.3.13 static uint32_t xos_set_int_pri_level (uint32_t *level*) [inline], [static]

Set the CPU's interrupt priority level to the specified level, but only if the current IPL is below the one requested. This function will never cause the interrupt priority level to be lowered from the current level. Call this function to block interrupts at or below the specified priority level.

When setting the IPL temporarily (such as in a critical section), call xos_set_int_pri_level(), execute the critical code section, and then call xos_restore_int_pri_level().

The interrupt priority level is part of the thread context, so it is saved and restored across context switches. To enable and disable individual interrupts globally, use the functions xos_interrupt_enable() and xos_interrupt_disable() instead.

NOTE: It is usually not required to disable interrupts at a level higher than that of the highest priority interrupt that interacts with the OS (i.e. calls into XOS such that threads may be woken / blocked / reprioritized / switched, or otherwise access XOS data structures). In XOS, that maximum level is XOS_MAX_OS_INTLEVEL, which defaults to XCHAL_EXCM_LEVEL. This may be modified by editing xos_params.h and rebuilding XOS.

Parameters

level	The new interrupt priority level (IPL).

Returns

Returns a value that can be used to restore the previous priority level by calling xos_restore_int_pri_level(). This value should be treated as opaque by application code, and should be passed unchanged to the restore function.

30.1.3.14 int32_t xos_unregister_interrupt_handler (uint32_t *num*)

Unregister a handler function for interrupt "num". If no handler was installed, this function will have no effect.

Parameters

num	Xtensa internal interrupt number (0..31). To refer to a specific external interrupt number (BInterrupt pin), use HAL macro XCHAL_EXTINTx_NUM where 'x' is the external number.

Returns

Returns XOS_OK if successful, else error code.

30.2 xos_blockmem.h File Reference

Data Structures

- struct XosBlockPool

Macros

- #define XOS_BLOCKMEM_WAIT_PRIORITY 0x0000U

  Wake waiters in priority order (default)
- #define XOS_BLOCKMEM_WAIT_FIFO 0x0001U

  Wake waiters in FIFO order.

Functions

- int32_t xos_block_pool_init (XosBlockPool ∗pool, void ∗mem, uint32_t blocksize, uint32_t nblocks, uint32_t flags)
- void ∗ xos_block_alloc (XosBlockPool ∗pool)
- void ∗ xos_block_try_alloc (XosBlockPool ∗pool)
- int32_t xos_block_free (XosBlockPool ∗pool, void ∗mem)
- int32_t xos_block_pool_check (XosBlockPool ∗pool)

30.2.1 Data Structure Documentation

30.2.1.1 struct XosBlockPool

XosBlockPool object.

Data Fields

uint32_t	flags	Properties.
uint32_t ∗	head	Pointer to first free block.
uint32_t	nblks	Number of blocks in the pool.
XosSem	sem	Semaphore to keep count / block.

30.2.2 Function Documentation

30.2.2.1 void* xos_block_alloc (XosBlockPool * *pool*)

Allocate a single block from the pool. If no block is available, the calling thread waits until memory becomes available. This function cannot be called from interrupt context because it can block.

Parameters

pool	Pointer to the memory pool to allocate from.

Returns

Returns pointer to the allocated block on success, else XOS_NULL.

30.2.2.2 int32_t xos_block_free (XosBlockPool * pool, void * mem)

Free a single block back to the pool. The block must have been allocated from the same pool.

Parameters

pool	Pointer to the memory pool to free into.
mem	Pointer to the block to be freed.

Returns

Returns XOS_OK on success, XOS_ERR_ILLEGAL_OPERATION if the block does not belong in this pool, else error code.

NOTE: Freeing a block can potentially wake up a higher priority thread and cause a context switch.

30.2.2.3 int32_t xos_block_pool_check (XosBlockPool * *pool*)

Verify that the state of the pool is consistent. The check happens only if XOS_BLOCKMEM_DEBUG is defined and is nonzero.

Parameters

pool	Pointer to memory pool to be checked.

Returns

Returns XOS_OK on success, XOS_ERR_INTERNAL_ERROR on error.

30.2.2.4 int32_t xos_block_pool_init (XosBlockPool * *pool*, void * *mem*, uint32_t *blocksize*, uint32_t *nblocks*, uint32_t *flags*)

Initialize a block memory pool.

Parameters

pool	Pointer to block pool object.
mem	Pointer to the memory area that the pool will use. This area must be at least (blocksize * nblocks) bytes in size.
blocksize	Size of each block in bytes. Must be a multiple of 4.
nblocks	The number of blocks to be created.
flags	Creation flags: • XOS_BLOCKMEM_WAIT_FIFO – queue waiting threads in fifo order. • XOS_BLOCKMEM_WAIT_PRIORITY – queue waiting threads by priority. This is the default.

Returns

Returns XOS_OK on success, else error code.

30.2.2.5 void* xos_block_try_alloc (XosBlockPool * *pool*)

Nonblocking version of xos_block_alloc(). Returns immediately on failure.

Parameters

pool	Pointer to the memory pool to allocate from.

Returns

Returns pointer to the allocated block on success, else XOS_NULL.

30.3 xos_cond.h File Reference

Data Structures

- struct XosCond

Functions

- void xos_cond_create (XosCond *cond)
- void xos_cond_delete (XosCond *cond)
- int32_t xos_cond_wait (XosCond *cond, XosCondFunc *cond_fn, void *cond_arg)
- int32_t xos_cond_wait_mutex (XosCond *cond, struct XosMutex *mutex)
- int32_t xos_cond_wait_mutex_timeout (XosCond *cond, struct XosMutex *mutex, uint64_t to_cycles)
- static int32_t xos_cond_signal (XosCond *cond, int32_t sig_value)
- static int32_t xos_cond_signal_one (XosCond *cond, int32_t sig_value)

30.3.1 Data Structure Documentation

30.3.1.1 struct XosCond

Condition object.

Data Fields

XosThreadQueue	queue	Queue of waiters.

30.3.2 Function Documentation

30.3.2.1 void xos_cond_create (XosCond * *cond*)

Initialize a condition object before first use. The object must be allocated by the caller.

Parameters

cond	Pointer to condition object.

Returns

Returns nothing.

30.3.2.2 void xos_cond_delete (XosCond * *cond*)

Destroy a condition object. Must have been previously created by calling xos_cond_create().

Parameters

cond	Pointer to condition object.

Returns

Returns nothing.

30.3.2.3 static int32_t xos_cond_signal (XosCond * *cond*, int32_t *sig_value*) [inline], [static]

Trigger the condition: wake all threads waiting on the condition, if their condition function evaluates to true (non-zero). If there is no condition function for a thread then it is automatically awakened.

The condition object must have been initialized before first use by calling xos_cond_create().

Parameters

cond	Pointer to condition object.
sig_value	Value passed to all waiters, returned by xos_cond_wait().

Returns

Returns number of woken threads on success, else error code.

NOTE: Signaling a condition that has no waiters has no effect on it, and the signal is not remembered. Any thread that waits on it later must be woken by another call to xos_cond_signal() or xos_cond_signal_one().

30.3.2.4 static int32_t xos_cond_signal_one (XosCond * *cond*, int32_t *sig_value*) [inline], [static]

Trigger the condition: wake one thread waiting on the condition, if its condition function evaluates to true (non-zero). If there is no condition function for a thread then it is automatically awakened.

The condition object must have been initialized before first use by calling xos_cond_create().

Parameters

cond	Pointer to condition object.
sig_value	Value passed to woken thread, returned by xos_cond_wait().

Returns

Returns 1 on success, else error code.

NOTE: Signaling a condition that has no waiters has no effect on it, and the signal is not remembered. Any thread that waits on it later must be woken by another call to xos_cond_signal() or xos_cond_signal_one().

30.3.2.5 int32_t xos_cond_wait (XosCond * *cond*, XosCondFunc * *cond_fn*, void * *cond_arg*)

Wait on a condition: block until the condition is satisfied. The condition is satisfied when xos_cond_signal() or xos_cond_signal_one() is called on this condition *and* the condition callback function returns non-zero. If there is no callback function, then the condition is automatically satisfied.

The condition object must have been initialized before first use by calling xos_cond_create().

Parameters

cond	Pointer to condition object.
cond_fn	Pointer to a function, called by xos_cond_signal(), that should return non-zero if this thread is to be resumed. The function is invoked as: `(*cond_fn)(cond_arg, sig_value)`.
cond_arg	Argument passed to cond_fn.

Returns

Returns the value passed to xos_cond_signal(), or error code.

30.3.2.6 int32_t xos_cond_wait_mutex (XosCond * *cond*, struct XosMutex * *mutex*)

Atomically release the mutex and block until the condition is satisfied. The condition is satisfied when xos_cond_signal() or xos_cond_signal_one() is called on this condition. This function does not allow specifying a condition callback function.

The condition and mutex objects must have been initialized before this call.

Parameters

cond	Pointer to condition object.
mutex	Pointer to mutex object. The mutex must have been locked by the calling thread.

Returns

Returns the value passed to xos_cond_signal(), or error code.

30.3.2.7 int32_t xos_cond_wait_mutex_timeout (XosCond * *cond*, struct XosMutex * *mutex*, uint64_t *to_cycles*)

Atomically release the mutex and block until the condition is satisfied or the timeout expires. The condition is satisfied when xos_cond_signal() or xos_cond_signal_one() is called on this condition. This function does not allow specifying a condition callback function.

The condition and mutex objects must have been initialized before this call.

Parameters

cond	Pointer to condition object.
mutex	Pointer to mutex object. The mutex must have been locked by the calling thread.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns the value passed to xos_cond_signal(), or error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

30.4 xos_errors.h File Reference

Enumerations

- enum

Functions

- static bool IS_XOS_ERRCODE (int32_t val)

30.4.1 Enumeration Type Documentation

30.4.1.1 anonymous enum

List of XOS error codes. All error codes are negative integers, except for XOS_OK which is zero. XOS error codes occupy the range from -65536 up to -1. The function IS_XOS_ERRCODE() can be used to check if a value lies within the error code range.

Enumerator

XOS_ERR_NOT_FOUND Object not found.

XOS_ERR_INVALID_PARAMETER Function parameter is invalid.

XOS_ERR_LIMIT Limit exceeded.

XOS_ERR_NOT_OWNED Object not owned by caller.

XOS_ERR_MUTEX_LOCKED Mutex is already locked.

XOS_ERR_MUTEX_NOT_OWNED Mutex not owned by caller.

XOS_ERR_MUTEX_ALREADY_OWNED Mutex already owned by caller.

XOS_ERR_MUTEX_DELETE Mutex being waited on has been deleted.

XOS_ERR_COND_DELETE Condition being waited on has been deleted.

XOS_ERR_SEM_DELETE Semaphore being waited on has been deleted.

XOS_ERR_SEM_BUSY Semaphore is not available.

XOS_ERR_EVENT_DELETE Event being waited on has been deleted.

XOS_ERR_MSGQ_FULL Message queue is full.

XOS_ERR_MSGQ_EMPTY Message queue is empty.

XOS_ERR_MSGQ_DELETE Message queue being waited on has been deleted.

XOS_ERR_TIMER_DELETE Timer being waited on has been deleted.

XOS_ERR_CONTAINER_NOT_RTC Containing thread not of RTC type.

XOS_ERR_CONTAINER_NOT_SAME_PRI Containing thread not at same priority.

XOS_ERR_STACK_TOO_SMALL Thread stack is too small.

XOS_ERR_CONTAINER_ILLEGAL Illegal container thread.

XOS_ERR_ILLEGAL_OPERATION This operation is not allowed.

XOS_ERR_THREAD_EXITED The thread has already exited.

XOS_ERR_NO_TIMER No suitable timer found.

XOS_ERR_FEATURE_NOT_PRESENT This feature is disabled or not implemented.

XOS_ERR_TIMEOUT Wait timed out.

XOS_ERR_STACK_OVERRUN Possible thread stack corruption detected.

XOS_ERR_UNHANDLED_INTERRUPT No handler for interrupt.

XOS_ERR_UNHANDLED_EXCEPTION No handler for exception.

XOS_ERR_INTERRUPT_CONTEXT Operation is illegal in interrupt context.

XOS_ERR_THREAD_BLOCKED Thread already blocked.

XOS_ERR_ASSERT_FAILED Runtime assertion failure.

XOS_ERR_CLIB_ERR Error in C library thread safety module.

XOS_ERR_INTERNAL_ERROR XOS internal error.

30.4.2 Function Documentation

30.4.2.1 static bool IS_XOS_ERRCODE (int32_t *val*) [inline],[static]

Check if a value is a valid XOS error code.

Parameters

val	Value to check

Returns

Returns true if 'val' is in the XOS error code range.

30.5 xos_event.h File Reference

Data Structures

- struct XosEvent

Macros

- #define XOS_EVENT_AUTO_CLEAR 0x0010U

  Auto-clear bits on thread signal.

Functions

- void xos_event_create (XosEvent ∗event, uint32_t mask, uint16_t flags)
- void xos_event_delete (XosEvent ∗event)
- int32_t xos_event_set (XosEvent ∗event, uint32_t bits)
- int32_t xos_event_clear (XosEvent ∗event, uint32_t bits)
- int32_t xos_event_clear_and_set (XosEvent ∗event, uint32_t clr_bits, uint32_t set_bits)
- int32_t xos_event_get (XosEvent ∗event, uint32_t ∗pstate)
- int32_t xos_event_wait_all (XosEvent ∗event, uint32_t bits)
- int32_t xos_event_wait_all_timeout (XosEvent ∗event, uint32_t bits, uint64_t to_cycles)
- int32_t xos_event_wait_any (XosEvent ∗event, uint32_t bits)
- int32_t xos_event_wait_any_timeout (XosEvent ∗event, uint32_t bits, uint64_t to_cycles)
- int32_t xos_event_set_and_wait (XosEvent ∗event, uint32_t set_bits, uint32_t wait_bits)

30.5.1 Data Structure Documentation

30.5.1.1 struct XosEvent

Event object.

Data Fields

uint32_t	bits	Event bits.
uint16_t	flags	Properties.
uint32_t	mask	Specifies which bits are valid.
uint16_t	pad	Padding.

XosThreadQueue	waitq	Queue of waiters.

30.5.2 Function Documentation

30.5.2.1 int32_t xos_event_clear (XosEvent * *event*, uint32_t *bits*)

Clear the specified bits in the specified event. Propagates the bit states to all waiting threads and wakes them if needed.

Parameters

event	Pointer to event object.
bits	Mask of bits to clear. Every bit that is set in the mask will be cleared from the event. Bits not set in the mask will not be modified by this call. To clear all the bits in an event use the constant XOS_EVENT_BITS_ALL.

Returns

Returns XOS_OK on success, else error code.

30.5.2.2 int32_t xos_event_clear_and_set (XosEvent * *event*, uint32_t *clr_bits*, uint32_t *set_bits*)

Clear and set the specified bits in the specified event. The two steps are combined into one update, so this is faster than calling xos_event_clear() and xos_event_set() separately. Only one update is sent out to waiting threads.

Parameters

event	Pointer to event object.
clr_bits	Mask of bits to clear. The clear operation happens before the set operation.
set_bits	Mask of bits to set.

Returns

Returns XOS_OK on success, else error code.

30.5.2.3 void xos_event_create (XosEvent * *event*, uint32_t *mask*, uint16_t *flags*)

Initialize an event object before first use.

Parameters

event	Pointer to event object.
mask	Mask of active bits. Only these bits can be signaled.
flags	Creation flags: • XOS_EVENT_AUTO_CLEAR – Automatically clear the set bits that signal a waiting thread.

Returns

Returns nothing.

30.5.2.4 void xos_event_delete (XosEvent * *event*)

Destroy an event object. Must have been previously created by calling xos_event_create().

Parameters

event	Pointer to event object.

Returns

Returns nothing.

30.5.2.5 int32_t xos_event_get (XosEvent * *event*, uint32_t * *pstate*)

Get the current state of the event object. This is a snapshot of the state of the event at this time.

Parameters

event	Pointer to event object.
pstate	Pointer to a uint32_t variable where the state will be returned.

Returns

Returns XOS_OK on success, else error code.

30.5.2.6 int32_t xos_event_set (XosEvent * *event*, uint32_t *bits*)

Set the specified bits in the specified event. Propagates the bit states to all waiting threads and wakes them if needed.

Parameters

event	Pointer to event object.
bits	Mask of bits to set. Bits not set in the mask will not be modified by this call. To set all the bits in the event, use the constant XOS_EVENT_BITS_ALL.

Returns

Returns XOS_OK on success, else error code.

30.5.2.7 int32_t xos_event_set_and_wait (XosEvent * *event*, uint32_t *set_bits*, uint32_t *wait_bits*)

Atomically set a specified group of bits, then wait for another specified group of bits to become set.

Parameters

event	Pointer to event object.
set_bits	Group of bits to set.
wait_bits	Group of bits to wait on. All the bits in the group will have to get set before the wait is satisfied.

Returns

Returns XOS_OK on success, else error code.

30.5.2.8 int32_t xos_event_wait_all (XosEvent * *event*, uint32_t *bits*)

Wait until all the specified bits in the wait mask become set in the given event object.

Parameters

event	Pointer to event object.
bits	Mask of bits to test.

Returns

Returns XOS_OK on success, else error code.

30.5.2.9 int32_t xos_event_wait_all_timeout (XosEvent * *event*, uint32_t *bits*, uint64_t *to_cycles*)

Wait until all the specified bits in the wait mask become set in the given event object, or the timeout expires.

Parameters

event	Pointer to event object.
bits	Mask of bits to test.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns XOS_OK on success, XOS_ERR_TIMEOUT on timeout, else error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

30.5.2.10 int32_t xos_event_wait_any (XosEvent * *event*, uint32_t *bits*)

Wait until any of the specified bits in the wait mask become set in the given event object.

Parameters

event	Pointer to event object.
bits	Mask of bits to test.

Returns

Returns XOS_OK on success, else error code.

30.5.2.11 int32_t xos_event_wait_any_timeout (XosEvent * *event*, uint32_t *bits*, uint64_t *to_cycles*)

Wait until any of the specified bits in the wait mask become set in the event object, or the timeout expires.

Parameters

event	Pointer to event object.
bits	Mask of bits to test.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns XOS_OK on success, XOS_ERR_TIMEOUT on timeout, else error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

30.6 xos_msgq.h File Reference

Data Structures

- struct XosMsgQueue

Macros

- #define XOS_MSGQ_WAIT_PRIORITY 0x0000U

  Wake waiters in priority order (default)
- #define XOS_MSGQ_WAIT_FIFO 0x0001U

  Wake waiters in FIFO order.
- #define XOS_MSGQ_ALLOC(name, num, size)

Functions

- int32_t xos_msgq_create (XosMsgQueue ∗msgq, uint16_t num, uint32_t size, uint16_t flags)
- int32_t xos_msgq_delete (XosMsgQueue ∗msgq)
- int32_t xos_msgq_put (XosMsgQueue ∗msgq, const uint32_t ∗msg)
- int32_t xos_msgq_put_timeout (XosMsgQueue ∗msgq, const uint32_t ∗msg, uint64_t to_cycles)
- int32_t xos_msgq_get (XosMsgQueue ∗msgq, uint32_t ∗msg)
- int32_t xos_msgq_get_timeout (XosMsgQueue ∗msgq, uint32_t ∗msg, uint64_t to_cycles)
- int32_t xos_msgq_empty (const XosMsgQueue ∗msgq)
- int32_t xos_msgq_full (const XosMsgQueue ∗msgq)

30.6.1 Data Structure Documentation

30.6.1.1 struct XosMsgQueue

XosMsgQueue object.

Data Fields

uint16_t	count	**of messages queue can hold**
uint16_t	flags	queue flags
uint16_t	head	write pointer
uint32_t	msg[1]	first word of message buffer
uint32_t	msize	message size in bytes
XosThreadQueue	readq	reader wait queue
uint16_t	tail	read pointer
XosThreadQueue	writeq	writer wait queue

30.6.2 Macro Definition Documentation

30.6.2.1 #define XOS_MSGQ_ALLOC(*name, num, size*)

Value:

```
static uint8_t name ## _buf[ sizeof(XosMsgQueue) + ((num) * (size)) ]; \
    XosMsgQueue * name = (XosMsgQueue *) name ## _buf;
```

Use these macros to statically or dynamically allocate a message queue. XOS_MSGQ_ALLOC allocates a static queue, while XOS_MSGQ_SIZE can be used to allocate memory via malloc() etc.

Static: this allocates a queue named "testq", containing 10 messages, each 16 bytes long.

```
XOS_MSGQ_ALLOC(testq, 10, 16);
```

Dynamic: this allocates a queue named "testq", containing 10 messages, each 16 bytes long.

```
XosMsgQueue * testq = malloc( XOS_MSGQ_SIZE(10, 16) );
```

Parameters

name	The queue name, i.e. the name of the pointer to the queue. Used as the queue handle in queue API calls.

num	Number of messages to allocate in queue. Must be > 0.
size	Message size in bytes. Must be > 0 and multiple of 4.

30.6.3 Function Documentation

30.6.3.1 int32_t xos_msgq_create (XosMsgQueue * *msgq*, uint16_t *num*, uint32_t *size*, uint16_t *flags*)

Create the message queue object. Memory for the queue must be allocated by the caller, either statically or via dynamic allocation. See the macros XOS_MSGQ_ALLOC and XOS_MSGQ_SIZE for examples.

Parameters

msgq	Handle (pointer) to message queue.
num	Number of messages allocated in queue. Must be > 0.
size	Message size in bytes. Must be > 0 and multiple of 4.
flags	Queue flags: • XOS_MSGQ_WAIT_FIFO - blocked threads will be woken in FIFO order. • XOS_MSGQ_WAIT_PRIORITY - blocked threads will be woken in priority order (default).

Returns

Returns XOS_OK on success, else error code.

30.6.3.2 int32_t xos_msgq_delete (XosMsgQueue * msgq)

Destroys the specified queue. Any waiting threads are unblocked with an error return. Any messages in the queue will be lost.

Parameters

msgq	Pointer to message queue.

Returns

Returns XOS_OK on success, else error code.

30.6.3.3 int32_t xos_msgq_empty (const XosMsgQueue * *msgq*)

Check if the queue is empty.

Parameters

msgq	Pointer to message queue.

Returns

Returns nonzero if queue is empty, zero if queue is not empty.

30.6.3.4 int32_t xos_msgq_full (const XosMsgQueue * *msgq*)

Check if the queue is full.

Parameters

msgq	Pointer to message queue.

Returns

Returns nonzero if queue is full, zero if queue is not full.

30.6.3.5 int32_t xos_msgq_get (XosMsgQueue * msgq, uint32_t * msg)

Get a message from the queue. The message contents are copied into the buffer that must be provided. If no message is available, this function will block if called from a thread, but will return immediately if called from an interrupt handler.

Parameters

msgq	Pointer to message queue.
msg	Pointer to message buffer.

Returns

Returns XOS_OK on success, else error code.

30.6.3.6 int32_t xos_msgq_get_timeout (XosMsgQueue * *msgq*, uint32_t * *msg*, uint64_t *to_cycles*)

Get a message from the queue. The message contents are copied into the buffer that must be provided. If no message is available, this function will block if called from a thread, but will return immediately if called from an interrupt handler. The thread will be unblocked when a message arrives in the queue or the timeout expires.

Parameters

msgq	Pointer to message queue.
msg	Pointer to message buffer.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns XOS_OK on success, XOS_ERR_TIMEOUT on timeout, else error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

30.6.3.7 int32_t xos_msgq_put (XosMsgQueue * msgq, const uint32_t * msg)

Put a message into the queue. The message contents are copied into the next available message slot. If no space is available, this function will block if called from a thread, but will return immediately if called from an interrupt handler.

Parameters

msgq	Pointer to message queue.
msg	Pointer to message buffer.

Returns

Returns XOS_OK on success, else error code.

30.6.3.8 int32_t xos_msgq_put_timeout (XosMsgQueue * *msgq*, const uint32_t * *msg*, uint64_t *to_cycles*)

Put a message into the queue. The message contents are copied into the next available message slot. If no space is available, this function will block if called from a thread, but will return immediately if called from an interrupt handler. The thread will be unblocked when space frees up in the queue or the timeout expires.

Parameters

msgq	Pointer to message queue.
msg	Pointer to message buffer.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns XOS_OK on success, XOS_ERR_TIMEOUT on timeout, else error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

30.7 xos_mutex.h File Reference

Data Structures

- struct XosMutex

Macros

- #define XOS_MUTEX_WAIT_PRIORITY 0x0000U

  Wake waiters in priority order (default)
- #define XOS_MUTEX_WAIT_FIFO 0x0001U

  Wake waiters in FIFO order.

Functions

- int32_t xos_mutex_create (XosMutex *mutex, uint32_t flags, int8_t priority)
- int32_t xos_mutex_delete (XosMutex *mutex)
- int32_t xos_mutex_lock (XosMutex *mutex)
- int32_t xos_mutex_lock_timeout (XosMutex *mutex, uint64_t to_cycles)
- int32_t xos_mutex_unlock (XosMutex *mutex)
- int32_t xos_mutex_trylock (XosMutex *mutex)
- static int32_t xos_mutex_test (const XosMutex *mutex)

30.7.1 Data Structure Documentation

30.7.1.1 struct XosMutex

XosMutex object.

Data Fields

uint32_t	flags	Properties.
int32_t	lock_count	For recursive locking.
XosThread *	owner	Owning thread (null if unlocked).
XosThreadQueue	waitq	Queue of waiters.

30.7.2 Function Documentation

30.7.2.1 int32_t xos_mutex_create (XosMutex * *mutex*, uint32_t *flags*, int8_t *priority*)

Initialize a mutex object before first use.

Parameters

mutex	Pointer to mutex object.
flags	Creation flags: • XOS_MUTEX_WAIT_FIFO – Queue waiting threads in fifo order. • XOS_MUTEX_WAIT_PRIORITY – Queue waiting threads by priority. This is the default.
priority	Mutex's priority ceiling. This is used only if option XOS_OPT_MUTEX_PRIORITY is enabled. Legal values are from 0 .. (XOS_NUM_PRIORITY - 1). A value of zero disables priority ceiling.

Returns

Returns XOS_OK on success, else error code.

30.7.2.2 int32_t xos_mutex_delete (XosMutex * *mutex*)

Destroy a mutex object. Must have been previously initialized by calling xos_mutex_create().

Parameters

mutex	Pointer to mutex object.

Returns

Returns XOS_OK on success, else error code.

30.7.2.3 int32_t xos_mutex_lock (XosMutex * *mutex*)

Take ownership of the mutex: block until the mutex is owned. The mutex must have been initialized.

Parameters

mutex	Pointer to mutex object.

Returns

Returns XOS_OK on success, else error code.

30.7.2.4 int32_t xos_mutex_lock_timeout (XosMutex * *mutex*, uint64_t *to_cycles*)

Take ownership of the mutex: block until the mutex is owned or the timeout expires. The mutex must have been initialized.

Parameters

mutex	Pointer to mutex object.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns XOS_OK on success, XOS_ERR_TIMEOUT on timeout, else error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

IMPORTANT: This function does not implement priority inheritance. The combination of a timed wait and priority elevation can lead to scenarios that are difficult to analyze and verify.

30.7.2.5 static int32_t xos_mutex_test (const XosMutex * *mutex*) [inline], [static]

Return the state of the mutex (locked or unlocked) but do not attempt to take ownership. The mutex must have been initialized.

Parameters

mutex	Pointer to mutex object.

Returns

Returns 0 if the mutex is unlocked, 1 if it is locked, -1 on error.

30.7.2.6 int32_t xos_mutex_trylock (XosMutex * *mutex*)

Try to take ownership of the mutex, but do not block if the mutex is taken. Return immediately. The mutex must have been initialized.

Parameters

mutex	Pointer to mutex object.

Returns

Returns XOS_OK on success (mutex owned), else error code.

30.7.2.7 int32_t xos_mutex_unlock (XosMutex * *mutex*)

Release ownership of the mutex. The mutex must have been initialized and must be owned by the calling thread. If released from an interrupt handler the mutex must be owned by the thread that was preempted.

Parameters

mutex	Pointer to mutex object.

Returns

Returns XOS_OK on success, else error code.

30.8 xos_params.h File Reference

Macros

- #define XOS_NUM_PRIORITY 16
- #define XOS_DEBUG_ALL 0
- #define XOS_OPT_STATS 1
- #define XOS_OPT_MSGQ_STATS 0
- #define XOS_OPT_BLOCKMEM_STATS 0
- #define XOS_INT_STACK_SIZE 32
- #define XOS_MAX_OS_INTLEVEL XCHAL_NUM_INTLEVELS
- #define XOS_OPT_STACK_CHECK 0
- #define XOS_CLOCK_FREQ 1000000
- #define XOS_OPT_INTERRUPT_SWPRI 1
- #define XOS_OPT_THREAD_SAFE_CLIB 1
- #define XOS_OPT_WAIT_TIMEOUT 1
- #define XOS_OPT_TIMER_WAIT 1
- #define XOS_OPT_TIME_SLICE 1
- #define XOS_TIME_SLICE_TICKS 1
- #define XOS_BLOCKMEM_DEBUG 0
- #define XOS_OPT_MUTEX_PRIORITY 1
- #define XOS_USE_INT_WRAPPER 0
- #define XOS_OPT_LOG_SYSEVENT 0

30.8.1 Macro Definition Documentation

30.8.1.1 #define XOS_BLOCKMEM_DEBUG 0

Set this option to 1 to enable debugging of block memory pool operations. This option will enable additional checks during alloc/free, and will also enable the internal consistency check via xos_block_pool_check().

Enabling this option will add a small overhead to alloc/free operations.

30.8.1.2 #define XOS_CLOCK_FREQ 1000000

Set XOS_CLOCK_FREQ to the system clock frequency if this is known ahead of time. Otherwise, call xos_set_clock_freq() to set it at run time.

30.8.1.3 #define XOS_DEBUG_ALL 0

Debug flags - Set to 1 to enable debug mode (and more verbose operation). Can be set individually, or define XOS_DEBUG_ALL=1 to enable all of them.

- XOS_DEBUG – Generic OS debug
- XOS_COND_DEBUG – Condition objects debug
- XOS_EVENT_DEBUG – Event objects debug
- XOS_MSGQ_DEBUG – Message queue debug
- XOS_MUTEX_DEBUG – Mutex objects debug
- XOS_SEM_DEBUG – Semaphore objects debug
- XOS_THREAD_DEBUG – Thread module debug
- XOS_TIMER_DEBUG – Timer module debug

WARNING: Enabling one or more of these flags will affect system performance and timing.

NOTE: Not all of these have been fully implemented.

30.8.1.4 #define XOS_INT_STACK_SIZE 32

Size of interrupt stack in bytes. Shared by all interrupt handlers. Must be sized to handle worst case nested interrupts. This is also used by the idle thread so must exist even if interrupts are not configured.

30.8.1.5 #define XOS_MAX_OS_INTLEVEL XCHAL_NUM_INTLEVELS

Default maximum interrupt level at which XOS primitives may be called. It is the level at which interrupts are disabled by default. See also description of xos_set_int_pri_level().

30.8.1.6 #define XOS_NUM_PRIORITY 16

Number of thread priority levels. At this time XOS supports a maximum of 32 priority levels (0 - 31).

30.8.1.7 #define XOS_OPT_BLOCKMEM_STATS 0

Set this option to 1 to enable statistics tracking for block memory pools. Enabling this causes block pool objects to increase in size and adds a small overhead to alloc/free operations.

30.8.1.8 #define XOS_OPT_INTERRUPT_SWPRI 1

Set this option to 1 to enable software prioritization of interrupts. The priority scheme applied is that a higher interrupt number at the same level will have higher priority.

30.8.1.9 #define XOS_OPT_LOG_SYSEVENT 0

Setting this option to 1 enables system event logging. The event log must be configured and enabled for event logging to work. If the event log is not enabled, this option will do nothing.

30.8.1.10 #define XOS_OPT_MSGQ_STATS 0

Set this option to 1 to enable statistics tracking for message queues. Enabling this causes message queue objects to increase in size and adds some overhead to message queue processing.

30.8.1.11 #define XOS_OPT_MUTEX_PRIORITY 1

Set this option to 1 to enable priority inversion protection for mutexes. This option must be enabled to use mutex priority inheritance and priority protection.

Enabling this option will add to both code and data size and runtime.

30.8.1.12 #define XOS_OPT_STACK_CHECK 0

Set this to 1 to enable stack checking. The stack is filled with a pattern on thread creation, and the stack is checked at certain times during system operation. WARNING: Enabling this option can have some impact on runtime performance.

30.8.1.13 #define XOS_OPT_STATS 1

Set this option to 1 to enable runtime statistics collection for XOS. NOTE: Enabling this option does have some impact on runtime performance and OS footprint.

30.8.1.14 #define XOS_OPT_THREAD_SAFE_CLIB 1

Set this option to 1 to use the thread-safe version of the C runtime library. You may need to enable this if you call C library functions from multiple threads – see the documentation for the relevant C library to determine if this is necessary. This option increases the size of the TCB. NOTE: At this time only the newlib and xclib libraries are supported for thread safety.

30.8.1.15 #define XOS_OPT_TIME_SLICE 1

Set this option to 1 to enable time-slicing between multiple threads at the same priority. If this option is enabled then after every slice period the timer handler will switch out the current thread if there is another ready thread at the same priority, and allow the latter thread to run. Execution will be round robin switched among all threads at the same priority.

The time slice interval is a multiple of the timer tick period, and is determined by the value of XOS_TIME_SLICE_TICKS.

This feature is most useful if fixed duration timer ticks are used. If dynamic ticking is enabled, then time slicing will work unpredictably because the interval between ticks will vary. In such cases it may be better to turn time slicing off.

30.8.1.16 #define XOS_OPT_TIMER_WAIT 1

Set this option to 1 to enable threads waiting on timer objects. If this feature is not used, turning it off will make timer objects smaller, and reduce the time taken by timer expiry processing (by a small amount).

30.8.1.17 #define XOS_OPT_WAIT_TIMEOUT 1

Set this option to 1 to enable the wait timeout feature. This allows waits on waitable objects to expire after a specified timeout.

30.8.1.18 #define XOS_TIME_SLICE_TICKS 1

Number of timer ticks per time slice interval. This setting is used only if the option XOS_OPT_TIME_SLICE is enabled. Setting this to a higher number will increase the time slice interval. The valid range for this setting is 1 to 50. Specifying a number outside the range will cause it to be capped within the range limit.

30.8.1.19 #define XOS_USE_INT_WRAPPER 0

Setting this option to 1 causes all interrupts to go to a wrapper function which then dispatches them to the appropriate handlers. NOTE: This is an experimental feature.

30.9 xos_semaphore.h File Reference

Data Structures

- struct XosSem

Macros

- #define XOS_SEM_WAIT_PRIORITY 0x0000U

  Wake waiters in priority order (default)
- #define XOS_SEM_WAIT_FIFO 0x0001U

  Wake waiters in FIFO order.

Functions

- int32_t xos_sem_create (XosSem ∗sem, uint32_t flags, uint32_t initial_count)
- int32_t xos_sem_delete (XosSem ∗sem)
- int32_t xos_sem_get (XosSem ∗sem)
- int32_t xos_sem_get_timeout (XosSem ∗sem, uint64_t to_cycles)
- int32_t xos_sem_put (XosSem ∗sem)
- int32_t xos_sem_put_max (XosSem ∗sem, uint32_t max)
- int32_t xos_sem_tryget (XosSem ∗sem)
- static uint32_t xos_sem_test (const XosSem ∗sem)

30.9.1 Data Structure Documentation

30.9.1.1 struct XosSem

XosSem object.

Data Fields

uint32_t	count	Current count.
uint32_t	flags	Properties.
XosThreadQueue	waitq	Queue of waiters.

30.9.2 Function Documentation

30.9.2.1 int32_t xos_sem_create (XosSem * *sem*, uint32_t *flags*, uint32_t *initial_count*)

Initialize a semaphore object before first use.

Parameters

sem	Pointer to semaphore object.
flags	Creation flags: • XOS_SEM_WAIT_FIFO – queue waiting threads in fifo order. • XOS_SEM_WAIT_PRIORITY – queue waiting threads by priority. This is the default. • XOS_SEM_PRIORITY_INV – protect against priority inversion.
initial_count	Initial count for semaphore on creation.

Returns

Returns XOS_OK on success, else error code.

NOTE: XOS_SEM_PRIORITY_INV is NOT supported in the current release. It will be supported in a future release.

30.9.2.2 int32_t xos_sem_delete (XosSem * *sem*)

Destroy a semaphore object. Must have been previously created by calling xos_sem_create().

Parameters

sem	Pointer to semaphore object.

Returns

Returns XOS_OK on success, else error code.

30.9.2.3 int32_t xos_sem_get (XosSem * sem)

Decrement the semaphore count: block until the decrement is possible. The semaphore must have been initialized.

Parameters

sem	Pointer to semaphore object.

Returns

Returns XOS_OK on success, else error code.

30.9.2.4 int32_t xos_sem_get_timeout (XosSem * *sem*, uint64_t *to_cycles*)

Decrement the semaphore count: block until the decrement is possible or the timeout expires. The semaphore must have been initialized.

Parameters

sem	Pointer to semaphore object.
to_cycles	Timeout in cycles. Convert from time to cycles using the helper functions provided in xos_timer. A value of zero indicates no timeout.

Returns

Returns XOS_OK on success, XOS_ERR_TIMEOUT on timeout, else error code.

NOTE: If XOS_OPT_WAIT_TIMEOUT is not enabled, then the timeout value is ignored, and no timeout will occur.

30.9.2.5 int32_t xos_sem_put (XosSem * sem)

Increment the semaphore count. The semaphore must have been initialized. Remember that this action may wake up a waiting thread, and if that thread is higher priority then there will be an immediate context switch.

Parameters

sem	Pointer to semaphore object.

Returns

Returns XOS_OK on success, else error code.

30.9.2.6 int32_t xos_sem_put_max (XosSem * *sem*, uint32_t *max*)

Increment the semaphore count only if the specified maximum count will not be exceeded. The semaphore must have been initialized. This action could wake up a waiting thread, and if that thread is higher priority there will be an immediate context switch.

Parameters

sem	Pointer to semaphore object.
max	Maximum count of semaphore to enforce.

Returns

Returns XOS_OK on success, XOS_ERR_LIMIT if the limit would be exceeded, else error code.

30.9.2.7 static uint32_t xos_sem_test (const XosSem * *sem*) [inline], [static]

Return the count of the semaphore but do not attempt to decrement it. The semaphore must have been initialized.

Parameters

sem	Pointer to semaphore object.

Returns

Returns semaphore count, -1 on error.

30.9.2.8 int32_t xos_sem_tryget (XosSem * *sem*)

Try to decrement the semaphore, but do not block if the semaphore count is zero. Return immediately. The semaphore must have been initialized.

Parameters

sem	Pointer to semaphore object.

Returns

Returns XOS_OK on success (semaphore decremented), else error code.

30.10 xos_stopwatch.h File Reference

Data Structures

- struct XosStopwatch

Functions

- static void xos_stopwatch_init (XosStopwatch ∗sw)
- static void xos_stopwatch_start (XosStopwatch ∗sw)
- static void xos_stopwatch_stop (XosStopwatch ∗sw)
- static uint64_t xos_stopwatch_count (const XosStopwatch ∗sw)
- static uint64_t xos_stopwatch_elapsed (const XosStopwatch ∗sw)
- static void xos_stopwatch_clear (XosStopwatch ∗sw)

30.10.1 Data Structure Documentation

30.10.1.1 struct XosStopwatch

XosStopwatch object.

Data Fields

uint16_t	active	Active flag (nonzero when active)
uint64_t	start	Starting system cycle count.
uint64_t	total	Total accumulated cycle count.

30.10.2 Function Documentation

30.10.2.1 static void xos_stopwatch_clear (XosStopwatch * sw) [inline], [static]

Clears a stopwatch. Resets the accumulated count to zero, and deactivates it if active.

Parameters

sw	Pointer to a stopwatch object.

Returns

Returns nothing.

30.10.2.2 static uint64_t xos_stopwatch_count (const XosStopwatch * *sw*) [inline], [static]

Get stopwatch accumulated count.

Parameters

sw	Pointer to a stopwatch object.

Returns

Returns the accumulated count.

30.10.2.3 static uint64_t xos_stopwatch_elapsed (const XosStopwatch * sw) [inline], [static]

Get elapsed time since stopwatch was started. If not started, returns zero.

Parameters

sw	Pointer to a stopwatch object.

Returns

Returns elapsed time in cycles.

30.10.2.4 static void xos_stopwatch_init (XosStopwatch * sw) [inline], [static]

Initialize a stopwatch object.

Parameters

sw	Pointer to a stopwatch object.

Returns

Returns nothing.

30.10.2.5 static void xos_stopwatch_start (XosStopwatch * *sw*) [inline], [static]

Start a stopwatch. Starts cycle counting. Note that this does not necessarily start counting from zero. The current run (start-to-stop interval) will just get added to the accumulated count in the stopwatch if any. To reset the accumulated count, use xos_stopwatch_clear().

Parameters

sw	Pointer to a stopwatch object.

Returns

Returns nothing.

30.10.2.6 static void xos_stopwatch_stop (XosStopwatch * *sw*) [inline], [static]

Stop a stopwatch. Stops cycle counting and updates total.

Parameters

sw	Pointer to a stopwatch object.

Returns

Returns nothing.

30.11 xos_syslog.h File Reference

Data Structures

- struct XosSysLogEntry
- struct XosSysLog

Macros

- #define XOS_SYSLOG_SIZE(num_entries) (sizeof(XosSysLog) + (((num_entries) - 1) ∗ sizeof(XosSysLogEntry)))

Functions

- static void xos_syslog_init (void ∗log_mem, uint16_t num_entries)
- static void xos_syslog_clear (void)
- static void xos_syslog_enable (void)
- static void xos_syslog_disable (void)
- static void xos_syslog_write (uint32_t param1, uint32_t param2)
- static int32_t xos_syslog_get_first (XosSysLogEntry ∗entry)
- static int32_t xos_syslog_get_next (XosSysLogEntry ∗entry)

30.11.1 Data Structure Documentation

30.11.1.1 struct XosSysLogEntry

System log entry structure.

Data Fields

struct XosSysLogEntry ∗	next	Link to next entry.
uint32_t	param1	User defined value.
uint32_t	param2	User defined value.
uint32_t	timestamp	Timestamp in clock cycles.

30.11.1.2 struct XosSysLog

System log structure.

Data Fields

XosSysLogEntry	entries[1]	First entry.
uint16_t	flags	Flags.
XosSysLogEntry *	next	Next write position.
uint16_t	size	Number of entries.

30.11.2 Macro Definition Documentation

30.11.2.1 #define XOS_SYSLOG_SIZE(*num_entries*) (sizeof(XosSysLog) + (((num_entries) - 1) * sizeof(XosSysLogEntry)))

Use this macro to compute how much memory to allocate for the syslog.

30.11.3 Function Documentation

30.11.3.1 static void xos_syslog_clear (void) [inline],[static]

Reset the syslog. All entries made up to now are abandoned and the write pointer is set to the first entry location.

No parameters.

Returns

Returns nothing.

30.11.3.2 static void xos_syslog_disable (void) [inline], [static]

Disable logging to the syslog. It is sometimes useful to disable logging while the log is being examined or dumped.

No parameters.

Returns

Returns nothing.

30.11.3.3 static void xos_syslog_enable (void) [inline], [static]

Enable logging to the syslog. This function needs to be called only if logging had been previously disabled via xos_syslog_disable(), since initializing the syslog automatically enables it.

No parameters.

Returns

Returns nothing.

30.11.3.4 static int32_t xos_syslog_get_first (XosSysLogEntry * *entry*) [inline], [static]

Read the first (oldest) entry in the syslog. Will return an error if the log has not been created or is empty. Storage to copy the entry must be provided by the caller.

Parameters

entry	Pointer to storage where the entry data will be copied. This pointer must be passed to xos_syslog_get_next().

Returns

Returns XOS_OK on success, else error code.

30.11.3.5 static int32_t xos_syslog_get_next (XosSysLogEntry * *entry*) [inline], [static]

Get the next sequential entry from the syslog. This function must be called only after xos_syslog_get_first() has been called.

Parameters

entry	Pointer to storage where entry data will be copied. Must be the same pointer that was passed in the call to xos_syslog_get_first(), as it is used to keep track of the current position.

Returns

Returns XOS_OK on success, else error code.

30.11.3.6 static void xos_syslog_init (void * *log_mem,* uint16_t *num_entries*) [inline], [static]

Initialize the syslog. Initializing the log also enables it. The system log always wraps around when full and overwrites the oldest entries.

Parameters

log_mem	Pointer to allocated memory for the log.
num_entries	The number of entries that the log can contain.

Returns

Returns nothing.

30.11.3.7 static void xos_syslog_write (uint32_t *param1,* uint32_t *param2*) [inline], [static]

Write an entry into the syslog. This function does disable all interrupts since logging can be done from interrupt handlers as well. It will write into the log only if the log exists and is enabled.

Parameters

param1	User defined value.
param2	User defined value.

Returns

Returns nothing.

30.12 xos_thread.h File Reference

Data Structures

- struct XosThreadQueue
- union XosFrame
- struct XosThread
- struct XosThreadParm
- struct XosRtcThread
- struct XosThreadStats

Macros

- #define XOS_THREAD_SUSPEND 0x0001U

  Create suspended instead of ready.
- #define XOS_THREAD_RTC 0x0002U

  Run-to-completion thread.
- #define XOS_THREAD_NO_CP 0x0004U

  Thread does not use coprocessors.

Typedefs

- typedef int32_t(XosThreadFunc) (void *arg, int32_t wake_value)
- typedef int32_t(XosCondFunc) (void *arg, int32_t sig_value, XosThread *thread)
- typedef int32_t(XosThdExitFunc) (int32_t exitcode)

Enumerations

- enum xos_thread_state_t

Functions

- static void xos_threadp_set_cp_mask (XosThreadParm *parms, uint16_t cp_mask)
- static void xos_threadp_set_preemption_priority (XosThreadParm *parms, int8_t preempt_pri)
- static void xos_threadp_set_exit_handler (XosThreadParm *parms, XosThdExitFunc *handler)
- int32_t xos_thread_create (XosThread *thread, XosThread *container, XosThreadFunc *entry, void *arg, const char *name, void *stack, uint32_t stack_size, int8_t priority, XosThreadParm *parms, uint32_t flags)

- int32_t xos_thread_delete (XosThread *thread)
- int32_t xos_thread_abort (XosThread *thread, int32_t exitcode)
- void xos_thread_exit (int32_t exitcode)
- int32_t xos_thread_join (XosThread *thread, int32_t *p_exitcode)
- void xos_thread_yield (void)
- int32_t xos_thread_suspend (XosThread *thread)
- int32_t xos_thread_resume (XosThread *thread)
- static int8_t xos_thread_get_priority (XosThread *thread)
- int32_t xos_thread_set_priority (XosThread *thread, int8_t priority)
- static const char * xos_thread_get_name (XosThread *thread)
- static int32_t xos_thread_set_name (XosThread *thread, const char *name)
- int32_t xos_thread_set_exit_handler (XosThread *thread, XosThdExitFunc *func)
- static XosThread * xos_thread_id (void)
- static uint16_t xos_thread_cp_mask (XosThread *thread)
- static int32_t xos_thread_get_wake_value (XosThread *thread)
- static uint32_t xos_thread_get_event_bits (void)
- xos_thread_state_t xos_thread_get_state (XosThread *thread)
- uint32_t xos_preemption_disable (void)
- uint32_t xos_preemption_enable (void)
- void xos_start (uint32_t flags)
- void xos_start_main (const char *name, int8_t priority, uint32_t flags)
- int32_t xos_thread_get_stats (const XosThread *thread, XosThreadStats *stats)
- int32_t xos_get_cpu_load (XosThreadStats *stats, int32_t *size, int32_t reset)
- int32_t xos_set_sched_interrupt (int32_t intnum)

30.12.1 Data Structure Documentation

30.12.1.1 struct XosThreadQueue

30.12.1.2 union XosFrame

30.12.1.3 struct XosThreadParm

30.12.1.4 struct XosRtcThread

30.12.1.5 struct XosThreadStats

Per-thread stats structure. Note that the CPU use % is approximate, both because of cycle counting and because of integer division. So all the threads' CPU % will not add up to exactly 100%.

Data Fields

uint32_t	cpu_pct	CPU use % for this thread.
uint64_t	cycle_count	Number of cycles consumed.
uint32_t	normal_switches	Number of non-preemptive switches.
uint32_t	preempt_switches	Number of preemptive switches.
XosThread *	thread	Thread handle (or pseudo-handle)

30.12.2 Typedef Documentation

30.12.2.1 typedef int32_t(XosCondFunc) (void *arg, int32_t sig_value, XosThread *thread)

Condition evaluation callback function pointer type.

30.12.2.2 typedef int32_t(XosThdExitFunc) (int32_t exitcode)

Thread exit handler function pointer type.

30.12.2.3 typedef int32_t(XosThreadFunc) (void *arg, int32_t wake_value)

Thread entry function pointer type.

30.12.3 Enumeration Type Documentation

30.12.3.1 enum xos_thread_state_t

Enum values for thread state.

Enumerator

XOS_THREAD_STATE_INVALID Invalid thread.

XOS_THREAD_STATE_BLOCKED Thread is blocked.

XOS_THREAD_STATE_READY Thread is ready to run.

XOS_THREAD_STATE_RUNNING Thread is running.

XOS_THREAD_STATE_EXITED Thread has exited.

30.12.4 Function Documentation

30.12.4.1 int32_t xos_get_cpu_load (XosThreadStats * *stats*, int32_t * *size*, int32_t *reset*)

Get CPU loading statistics for the system. This function computes the CPU percentage use for all threads in the system (including the idle thread and the 'interrupt thread' (interrupt context). It also returns the cycle count and number of context switches for each thread. Statistics are only available if XOS_OPT_STATS has been enabled. Otherwise, the function will return XOS_OK, but the structure contents will be undefined.

IMPORTANT: The entry for interrupt context does not contain a real thread handle. It uses the pseudo-handle XOS_THD_STATS_INTR to indicate that this entry reports interrupt statistics. This pseudo-handle cannot be used for any other thread operations or queries.

NOTE: This function disables interrupts while traversing the thread list. It does not leave interrupts disabled during the computations, as that can take a fair amount of time.

Parameters

stats	Pointer to an array of XosThreadStats structures. The array must be large enough to accommodate all threads in the system.
size	The number of elements available in the array. If this is smaller than the number of threads plus one (for the interrupt context) then XOS_ERR_INVALID_PARAMETER will be returned and '*size' will be set to the minimum number of elements required. On a successful return, '*size' is set to the number of elements actually filled in.
reset	If nonzero, then thread stats counters are reset after reading. This is useful if you want to track the stats so as to get a better idea of current system loading. E.g. calling this function once a second with 'reset' nonzero will provide CPU load information for the last second on each call.

Returns

Returns XOS_OK on success, else error code. In particular, XOS_ERR_INVALID_PARAMETER will be returned if the output buffer is too small.

30.12.4.2 uint32_t xos_preemption_disable (void)

Disable thread preemption. Prevents context switching to another thread. However, interrupt handlers will still continue to be run. Multiple calls will nest, and the same number of calls to xos_preemption_enable() will be required to re-enable preemption. If the calling thread yields the CPU or exits without enabling preemption, it will cause a system halt. If the calling thread encounters a fatal error, preemption will be enabled during fatal error handling.

Returns

Returns the new value of preemption disable flag after this call.

NOTE: Cannot be called from interrupt context.

30.12.4.3 uint32_t xos_preemption_enable (void)

Enable thread preemption. Has no effect if preemption was already enabled. Otherwise, it decrements the value of the preemption disable flag and if the value goes to zero, enables preemption.

Returns

Returns the new value of preemption disable flag after this call.

NOTE: If scheduling gets enabled, it may cause an immediate context switch if higher priority threads are ready.

30.12.4.4 int32_t xos_set_sched_interrupt (int32_t *intnum*)

Specify which software interrupt the scheduler should use. This is only required for XEA3-based systems. The scheduler will use the interrupt to trigger context switch as needed. If you do not specify an interrupt number, then XOS will automatically select the first (i.e. numerically smallest) free software interrupt that it finds during startup.

Parameters

intnum	SW interrupt number to use for scheduling.

Returns

Returns XOS_OK on success, XOS_ERR_INVALID_PARAMETER if the specified interrupt is not a software interrupt.

IMPORTANT: If this function is called, it must be called before either xos_start() or xos_start_main() has been called. Once XOS is started the interrupt number cannot be changed. Also, once the scheduler interrupt has been selected, it cannot be used for any other purpose.

30.12.4.5 void xos_start (uint32_t *flags*)

Initialize XOS thread support and start scheduler.

Must be called from main() before calling any other thread function. This function initializes thread support, creates the idle thread, and starts the scheduler. It does not return to its caller. This means that at least one user thread must be created before calling xos_start(). Otherwise, the scheduler will run the idle thread since it will be the only thread in the system, and no other thread can be created.

NOTE: This function does not initialize timer/tick support. For timer services to be available xos_start_system_timer() must be called.

NOTE: xos_start() and xos_start_main() are exclusive, both cannot be called within the same application.

Parameters

flags	Currently unused (pass 0).

Returns

Does not return.

30.12.4.6 void xos_start_main (const char * *name*, int8_t *priority*, uint32_t *flags*)

Initialize XOS thread support and create init (main) thread.

Must be called from main() before calling any other thread function. This function converts the caller into the 'main' or 'init' thread, and returns to the caller after completing initialization.

NOTE: This function does not initialize timer/tick support. For timer services to be available xos_start_system_timer() must be called.

NOTE: xos_start_main() and xos_start() are exclusive, both cannot be called within the same application.

Parameters

name	Name of main thread (see xos_thread_create()).
priority	Initial priority of main thread.
flags	Currently unused (pass 0).

Returns

Returns nothing.

30.12.4.7 int32_t xos_thread_abort (XosThread * *thread,* int32_t *exitcode*)

Force the thread to terminate. The thread execution is aborted, but exit processing will still happen, i.e. the exit handler (if any) will be run. After termination, any other threads waiting on this thread are notified. This function cannot be called on the current thread.

Parameters

thread	Handle of thread to be aborted.
exitcode	Exit code returned to any waiting threads.

Returns

Returns XOS_OK on success, else error code.

NOTE: If the thread is blocked waiting for something, the wait is aborted and the thread is made ready. NOTE: The thread is not guaranteed to have exited when this call returns. It will be made ready and set up for exit processing, but when the exit processing will actually happen depends on the state of the system and the priority of the thread being aborted.

30.12.4.8 static uint16_t xos_thread_cp_mask (XosThread * *thread*) [inline], [static]

Return the coprocessor mask for the specified thread.

Parameters

thread	Handle of thread being queried.

Returns

Returns the mask for the specified thread if available, else 0.

30.12.4.9 int32_t xos_thread_create (XosThread * *thread*, XosThread * *container*, XosThreadFunc * *entry*, void * *arg*, const char * *name*, void * *stack*, uint32_t *stack_size*, int8_t *priority*, XosThreadParm * *parms*, uint32_t *flags*)

Create a new thread. If the thread is not created suspended, then it will be made ready as soon as it is created, and will immediately run if it is the highest priority non-blocked thread in the system.

Parameters

thread	Pointer to the thread descriptor (an otherwise unused XosThread structure, usually allocated by the caller for the lifetime of the thread, for example as a global variable).
container	Pointer to separate thread acting as "container" for this one. At the moment, this is only meaningful for run-to-completion (RTC) threads (identified with the XOS_THREAD_RTC flag), in which case the container must have the same priority and also be an RTC thread. (The priority restriction may be lifted in a future implementation, with appropriate constraints on dynamic reprioritization of the created thread).
entry	Thread entry function, takes one argument.
arg	Argument "void*" that is passed to the thread function.
name	Unique name of the thread, for debug/display purposes. This string must be valid for the lifetime of the thread (only a pointer to it is stored in the thread control block). Typically consists of identifier chars with no spaces.
stack	Base of initial stack for the thread, allocated by the caller. Need not be aligned (initial stack pointer will be computed and aligned from given stack base and size). Required argument, except for run-to-completion threads when container is non-NULL, in which case the container's stack is used and this argument must be NULL.
stack_size	Size of the stack, in bytes. NOTE: stack should be at least XOS_STACK_EXTRA bytes plus whatever the thread actually needs if the thread will use coprocessors/TIE state. If the thread will not touch the coprocessors, then it should be XOS_STACK_EXTRA_NO_CP plus whatever the thread actually needs. Recommended minimum stack sizes are defined by the constants XOS_STACK_MIN_SIZE and XOS_STACK_MIN_SIZE_NO_CP.

For run-to-completion threads where container is non-NULL, stack_size specifies the minimum stack size required for the thread; it should be smaller or equal to the container's stack.

Parameters

priority	Initial thread priority. From 0 .. XOS_MAX_PRIORITY - 1. Higher numbers are higher priority.

parms	Pointer to extra parameters structure, or 0 if none given. Use xos_thread_p_***() functions to set parameters in the structure.
flags	Option flags: • XOS_THREAD_SUSPEND – Leave thread suspended instead of making it ready. The thread can be made ready to run later by calling xos_thread_resume(). • XOS_THREAD_RTC – Run-to-completion thread. • XOS_THREAD_NO_CP – Thread does not use coprocessors. No coprocessor state will be saved for this thread. Threads that have this flag set will not allocate any storage for saving coprocessor state and so can have smaller stacks.

NOTE: xos_start_main() calls xos_thread_create() to convert main() into the 'main' thread.

Returns

Returns XOS_OK if successful, error code otherwise.

30.12.4.10 int32_t xos_thread_delete (XosThread * *thread*)

Remove thread and free up all resources. Thread must have exited already. After this call returns, all resources allocated to the thread (e.g. TCB, stack space, etc.) can be reused.

Parameters

thread	Handle of thread to be deleted.

Returns

Returns XOS_OK on success, else error code.

NOTE: A thread cannot call this on itself.

30.12.4.11 void xos_thread_exit (int32_t *exitcode*)

Exit the current thread. The exit handler (if any) will be run before the thread terminates.

Parameters

exitcode	Exit code to be returned to any waiting threads.

Returns

This function does not return.

NOTE: This is automatically called if the thread returns from its entry function. The entry function's return value will be passed as the exit code.

30.12.4.12 static uint32_t xos_thread_get_event_bits (void) [inline], [static]

Return the current value of the event bits for the current thread. This function takes no parameters.

Returns

Returns the current value of the event bits. The event bits are set when the thread is woken from an event wait. They will not change while the thread is running. There is no way to determine when the event bits were last updated.

30.12.4.13 static const char* xos_thread_get_name (XosThread * *thread*) [inline], [static]

Return the name of the specified thread.

Parameters

thread	Handle of thread being queried. A thread can use the special handle XOS_THREAD_SELF to specify itself.

Returns

Returns a pointer to the name string if available, else NULL.

30.12.4.14 static int8_t xos_thread_get_priority (XosThread * *thread*) [inline], [static]

Get the priority of the specified thread. This returns the priority of the queried thread at this instant, however this can change at any time due to other activity in the system.

Parameters

thread	Handle of thread being queried. A thread can use the special handle XOS_THREAD_SELF to query itself.

Returns

Returns the thread's current priority, or -1 if the thread handle is not valid.

30.12.4.15 xos_thread_state_t xos_thread_get_state (XosThread * *thread*)

Return the state of the specified thread.

Parameters

thread	Handle of thread being queried.

Returns

Returns one of the following values:

- XOS_THREAD_STATE_RUNNING – The thread is currently running.
- XOS_THREAD_STATE_READY – The thread is ready to run.
- XOS_THREAD_STATE_BLOCKED – The thread is blocked on something.
- XOS_THREAD_STATE_INVALID – The thread handle is invalid.
- XOS_THREAD_STATE_EXITED – The thread has exited.

30.12.4.16 int32_t xos_thread_get_stats (const XosThread * *thread*, XosThreadStats * *stats*)

Get the thread statistics for the specified thread. Statistics are only available if XOS_OPT_STATS has been enabled. Otherwise, the function will return XOS_OK, but the structure contents will be undefined.

Parameters

thread	Handle of thread being queried. The following special pseudo-handles can be used: • XOS_THD_STATS_IDLE – stats for idle thread • XOS_THD_STATS_INTR – stats for interrupt processing
stats	Pointer to XosThreadStats struct to be filled in.

Returns

Returns XOS_OK on success, else error code.

NOTE: Can be called from interrupt context. NOTE: This call will not fill in the "thread" and "cpu_pct" fields in the "stats" structure. The thread handle is already known, and calculating the CPU loading can take quite a bit of time so is not done here.

30.12.4.17 static int32_t xos_thread_get_wake_value (XosThread * *thread*) [inline], [static]

Return the wake value for the specified thread.

Returns

thread Handle of thread being queried.
Returns The last set wake value. There is no way to detect what action set the wake value and when.

30.12.4.18 static XosThread* xos_thread_id (void) [inline], [static]

Return the ID (handle) of the current thread.

Returns

Returns the handle of the current thread. This handle can be used in all XOS system calls.

NOTE: If called from interrupt context, returns the handle of the thread that was preempted.

30.12.4.19 int32_t xos_thread_join (XosThread * *thread,* int32_t * *p_exitcode*)

Wait until the specified thread exits and get its exit code. If the thread has exited already, an error will be returned.

Parameters

thread	The thread to wait for. Cannot be "self", i.e. one cannot wait on one's own exit.
p_exitcode	If not null, the exit code will be returned here.

Returns

Returns XOS_OK on success, else error code.

30.12.4.20 int32_t xos_thread_resume (XosThread * *thread*)

Resume a suspended thread. If the thread is not suspended or is blocked on some other condition then this function will do nothing. Otherwise, it will be made ready, and this can cause an immediate context switch if the thread is at a higher priority than the calling thread.

Parameters

thread	Handle of thread being resumed.

Returns

Returns XOS_OK on success, else error code.

30.12.4.21 int32_t xos_thread_set_exit_handler (XosThread * *thread,* XosThdExitFunc * *func*)

Set an exit handler for the specified thread. The exit handler is run when the thread terminates, either by calling xos_thread_exit() or by returning from its entry function. It will also be called if the thread is being terminated due to e.g. an unhandled exception.

The handler must be a function defined as e.g.:

```
int32_t exit_handler(int32_t exitcode);
```

The exit handler runs in the context of the exiting thread, and can call system services. It is provided with a single parameter which is the thread's exit code (the exit code may be set to an error code if the thread is being terminated due to an error or exception). The handler must return a value which will be set as the thread's exit code.

Parameters

thread	Handle of the thread for which the handler is to be installed. A thread can use the special handle XOS_THREAD_SELF to specify itself.
func	Pointer to exit handler function. To clear an existing handler, pass NULL as the pointer.

Returns

Returns XOS_OK on success, else error code.

30.12.4.22 static int32_t xos_thread_set_name (XosThread * *thread,* const char * *name*) [inline],[static]

Set the name of the specified thread.

Parameters

thread	Handle of thread whose name is to be set. A thread can use the special handle XOS_THREAD_SELF to specify itself.
name	Pointer to the new name string. The string is not copied, only the pointer is saved. So the string must be persistent for the life of the thread.

Returns

Returns XOS_OK on success, else error code.

30.12.4.23 int32_t xos_thread_set_priority (XosThread * *thread*, int8_t *priority*)

Set the priority of the specified thread. The thread must exist.

Parameters

thread	Handle of thread being affected. A thread can use the special handle XOS_THREAD_SELF to specify itself.
priority	The new priority level to be set.

Returns

Returns XOS_OK on success, else error code.

NOTE: Calling this function can result in a scheduler activation, and the caller may be suspended as a result.

30.12.4.24 int32_t xos_thread_suspend (XosThread * *thread*)

Suspend the specified thread. The thread will remain suspended until xos_thread_resume() has been called on it. If the thread is already blocked on some other condition, then this function will return an error.

Parameters

thread	Handle of thread being suspended. A thread can use the special handle XOS_THREAD_SELF to suspend itself.

Returns

Returns XOS_OK on success, else error code.

30.12.4.25 void xos_thread_yield (void)

Yield the CPU to the next thread in line. The calling thread remains ready and is placed at the tail of the ready queue at its current priority level. If there are no threads at the same priority level that are ready to run, then this call will return immediately.

Returns

Returns nothing.

30.12.4.26 static void xos_threadp_set_cp_mask (XosThreadParm * *parms,* uint16_t *cp_mask*) [inline],[static]

Set thread creation parameter: the group of coprocessors that this thread will use. This must be set during thread creation, and cannot be changed after the thread has been created. Defining this allows reduction of memory usage (for CP state saving) in some circumstances, and can also speed up the context switch time.

NOTE: Support for this is not currently implemented. If a thread uses any coprocessor, space for all coprocessors must be reserved.

Parameters

parms	Thread creation parameter structure. Must be allocated by the caller.
cp_mask	Bitmask of coprocessors thread is allowed to use. Bit 0 for coprocessor 0, etc.

Returns

Returns nothing.

30.12.4.27 static void xos_threadp_set_exit_handler (XosThreadParm * *parms*, XosThdExitFunc * *handler*) [inline], [static]

Set thread creation parameter: thread exit handler.

Parameters

parms	Thread creation parameter structure. Must be allocated by caller.
handler	Exit handler function.

Returns

Returns nothing.

30.12.4.28 static void xos_threadp_set_preemption_priority (XosThreadParm * *parms*, int8_t *preempt_pri*) [inline], [static]

Set thread creation parameter: thread pre-emption priority.

Parameters

parms	Thread creation parameter structure. Must be allocated by caller.
preempt_pri	Thread pre-emption blocking priority. From 0 .. XOS_NUM_PRIORITY - 1. Must be greater or equal to the thread priority (if not, is automatically set to thread priority).

Returns

Returns nothing.

30.13 xos_timer.h File Reference

Data Structures

- struct XosTimer

Macros

- #define XOS_SYS_TIMER_0 0

  Internal timer 0.
- #define XOS_SYS_TIMER_1 1

  Internal timer 1.
- #define XOS_SYS_TIMER_2 2

  Internal timer 2.
- #define XOS_SYS_TIMER_EXTERNAL -2

  External timer.
- #define XOS_SYS_TIMER_NONE -1

  No system timer selected.

Typedefs

- typedef void(XosTimerFunc) (void *arg)

Functions

- static void xos_set_clock_freq (uint32_t freq)
- static uint32_t xos_get_clock_freq (void)
- int32_t xos_start_system_timer (int32_t timer_num, uint32_t tick_period)
- int32_t xos_get_system_timer_num (void)
- void xos_timer_init (XosTimer *timer)
- int32_t xos_timer_start (XosTimer *timer, uint64_t when, uint32_t flags, XosTimerFunc *func, void *arg)
- int32_t xos_timer_stop (XosTimer *timer)
- int32_t xos_timer_restart (XosTimer *timer, uint64_t when)
- static int32_t xos_timer_is_active (const XosTimer *timer)
- static uint64_t xos_timer_get_period (const XosTimer *timer)
- int32_t xos_timer_set_period (XosTimer *timer, uint64_t period)
- static uint64_t xos_get_system_cycles (void)
- int32_t xos_thread_sleep (uint64_t cycles)

- static int32_t xos_thread_sleep_msec (uint64_t msecs)
- static int32_t xos_thread_sleep_usec (uint64_t usecs)
- int32_t xos_timer_wait (XosTimer *timer)
- void xos_tick_handler (void)
- int32_t xos_system_timer_select (int32_t timer_num, int32_t *psel)
- void xos_system_timer_init (uint32_t cycles)
- void xos_system_timer_set (uint32_t cycles)
- uint32_t xos_system_timer_get (void)

- static uint64_t xos_cycles_to_secs (uint64_t cycles)
- static uint64_t xos_cycles_to_msecs (uint64_t cycles)
- static uint64_t xos_cycles_to_usecs (uint64_t cycles)
- static uint64_t xos_secs_to_cycles (uint64_t secs)
- static uint64_t xos_msecs_to_cycles (uint64_t msecs)
- static uint64_t xos_usecs_to_cycles (uint64_t usecs)

30.13.1 Data Structure Documentation

30.13.1.1 struct XosTimer

Timer event structure. Used to track pending timer events.

Data Fields

bool	active	Set if active (in some list of events).
void *	arg	Argument to pass to called function.
uint64_t	delta	Delta for next re-trigger, 0 if none.
XosTimerFunc *	fn	Function to call when timer expires.
struct XosTimer *	next	Pointer to next event in list.
uint64_t	when	Time (clock cycles) at which to trigger.

30.13.2 Typedef Documentation

30.13.2.1 typedef void(XosTimerFunc) (void *arg)

Function pointer type for timer callbacks.

30.13.3 Function Documentation

30.13.3.1 static uint64_t xos_cycles_to_msecs (uint64_t *cycles*) [inline],[static]

Converts CPU cycles to time in milliseconds.

Parameters

cycles	Number of CPU cycles.

Returns

Equivalent number of milliseconds (truncated to integer).

30.13.3.2 static uint64_t xos_cycles_to_secs (uint64_t *cycles*) [inline],[static]

Converts CPU cycles to time in seconds.

Parameters

cycles	Number of CPU cycles.

Returns

Equivalent number of seconds (truncated to integer).

30.13.3.3 static uint64_t xos_cycles_to_usecs (uint64_t *cycles*) [inline],[static]

Converts CPU cycles to time in microseconds.

Parameters

cycles	Number of CPU cycles.

Returns

Equivalent number of microseconds (truncated to integer).

30.13.3.4 static uint32_t xos_get_clock_freq (void) [inline], [static]

Get current system clock frequency.

Returns

Returns current system clock frequency in cycles per second.

30.13.3.5 static uint64_t xos_get_system_cycles (void) [inline], [static]

Get the current system cycle count. This accounts for the periodic rollover of the 32-bit CCOUNT cycle counter and returns a 64-bit value.

Returns

Returns the current system cycle count.

30.13.3.6 int32_t xos_get_system_timer_num (void)

Get the timer number of the system timer. Useful mainly when XOS has been allowed to choose its own timer via xos_start_system_timer(). Not valid if called before the system timer has been started.

Returns

Returns one of XOS_SYS_TIMER_0, XOS_SYS_TIMER_1, XOS_SYS_TIMER_2 or XOS_SYS_TIMER_EXTERNAL, or XOS_SYS_TIMER_NONE.

30.13.3.7 static uint64_t xos_msecs_to_cycles (uint64_t *msecs*) [inline], [static]

Converts time in milliseconds to CPU cycle count.

Parameters

msecs	Number of milliseconds.

Returns

Equivalent number of CPU cycles.

30.13.3.8 static uint64_t xos_secs_to_cycles (uint64_t *secs*) [inline], [static]

Converts time in seconds to CPU cycle count.

Parameters

secs	Number of seconds.

Returns

Equivalent number of CPU cycles.

30.13.3.9 static void xos_set_clock_freq (uint32_t *freq*) [inline], [static]

Set system clock frequency. This is expected to be set only once, and only if the clock frequency is not known at compile time.

Parameters

freq	Frequency in cycles per second.

Returns

Returns nothing.

30.13.3.10 int32_t xos_start_system_timer (int32_t *timer_num,* uint32_t *tick_period*)

Initialize timer support and start the system timer. This function must be called before calling any other timer function.

NOTE: The smaller the tick period, the more precisely delays can be specified using timers. However, we also need to make the tick period large enough to allow time both to execute the tick timer interrupt handler and for the application to make reasonable forward progress. If tick_period is too small, the timer interrupt may re-trigger before the timer interrupt handler has returned to the application, thus keeping the processor busy in constantly executing the timer interrupt handler without leaving any cycles for the application. Or, the application might get some cycles but only a fraction of what is spent in the timer interrupt handler, thus severely impacting application performance.

The exact number of cycles needed to execute the timer interrupt handler is not specified here. It depends on many factors (e.g. use of caches, various processor configuration options, etc) and can vary by orders of magnitude. Also note that the time to execute this handler is variable: when timers expire upon a given tick timer interrupt, their respective timer handler functions are called from within the interrupt handler.

Parameters

timer_num	Which Xtensa timer to use (0..2). This timer must exist and be configured at level 1 or at a medium-priority interrupt level (<=EXCM_LEVEL). If 'timer_num' is -1, then this function will automatically choose the highest priority timer that is suitable for use. This value will be passed to xos_system_timer_select().
tick_period	Number of clock (CCOUNT) cycles between ticks. Must range between 0 and UINT32_MAX. Zero is used to specify dynamic tick (tickless) mode.

Returns

Returns XOS_OK on success, else error code.

30.13.3.11 uint32_t xos_system_timer_get (void)

Reads the current value of the system timer. The value returned is in CPU cycles. This function can be overridden to provide custom timer processing or to support an external timer.

Returns

Returns the number of CPU cycles elapsed since the timer last rolled over or was started.

30.13.3.12 void xos_system_timer_init (uint32_t *cycles*)

Starts the system timer and sets up the first interrupt. This function can be overridden to provide custom timer processing or to support an external timer.

Parameters

cycles	The number of CPU cycles from now when the first interrupt must occur.

30.13.3.13 int32_t xos_system_timer_select (int32_t *timer_num,* int32_t * *psel*)

Selects the timer to use. The selection may be one of the internal timers or an external timer. The default implementation selects an internal timer. This function can be overridden to provide custom timer processing or to support an external timer.

Parameters

timer_num	The internal timer number to select (0-2) or -1 to auto-select a timer. This parameter can be ignored by custom implementations that use an external timer.
psel	Pointer to a location where the selected timer ID must be returned. The timer ID must be one of XOS_SYS_TIMER_0, XOS_SYS_TIMER_1, XOS_SYS_TIMER_2 or XOS_SYS_TIMER_EXTERNAL.

Returns

Returns XOS_OK on success, else error code.

30.13.3.14 void xos_system_timer_set (uint32_t *cycles*)

Sets the next trigger value of the system timer. The parameter 'cycles' is the number of CPU cycles from now when the interrupt must occur. This function can be overridden to provide custom timer processing or to support an external timer.

Parameters

cycles	The number of CPU cycles from now when the next interrupt must occur.

30.13.3.15 int32_t xos_thread_sleep (uint64_t *cycles*)

Put calling thread to sleep for at least the specified number of cycles. The actual number of cycles spent sleeping may be larger depending upon the granularity of the system timer. Once the specified time has elapsed the thread will be woken and made ready.

Parameters

cycles	Number of system clock cycles to sleep.

Returns

Returns XOS_OK on success, else error code.

30.13.3.16 static int32_t xos_thread_sleep_msec (uint64_t *msecs*) [inline],[static]

Put calling thread to sleep for at least the specified number of msec. The actual amount of time spent sleeping may be larger depending upon the granularity of the system timer. Once the specified time has elapsed the thread will be woken and made ready.

Returns

msecs The number of milliseconds to sleep.
Returns XOS_OK on success, else error code.

30.13.3.17 static int32_t xos_thread_sleep_usec (uint64_t *usecs*) [inline],[static]

Put calling thread to sleep for at least the specified number of usec. The actual amount of time spent sleeping may be larger depending upon the granularity of the system timer. Once the specified time has elapsed the thread will be woken and made ready.

Returns

usecs The number of microseconds to sleep.
Returns XOS_OK on success, else error code.

30.13.3.18 void xos_tick_handler (void)

This function handles XOS timer tick processing. It must be called by the timer interrupt handler on every timer interrupt. This function computes the time to the next tick and sets it up by calling xos_system_timer_set().

30.13.3.19 static uint64_t xos_timer_get_period (const XosTimer * *timer*) [inline], [static]

Get the repeat period for a periodic timer. For a one-shot timer this will return zero. The period is reported in system clock cycles.

Parameters

timer	Pointer to timer object.

Returns

Returns period in cycles, or zero for non-periodic timers.

30.13.3.20 void xos_timer_init (XosTimer * *timer*)

Initialize timer object.

Parameters

timer	Pointer to timer event structure.

Returns

Returns nothing.

NOTE: This function should not be called on a timer object once it has been activated.

30.13.3.21 static int32_t xos_timer_is_active (const XosTimer * *timer*) [inline], [static]

Check if the timer is active. The timer is active if it has been started and not yet expired or canceled.

Parameters

timer	Pointer to timer object.

Returns

Returns non-zero if the timer is active, else zero.

30.13.3.22 int32_t xos_timer_restart (XosTimer * *timer,* uint64_t *when*)

Reset and restart the timer.

The timer is reset to go off at time "when" from now. If the timer was not active, it will be activated. If the timer was active, it will be restarted. If the timer is periodic, the period will be set to "when". The timer object must have been initialized at some point before this call.

Parameters

timer	Pointer to timer object.
when	Number of cycles from now that the timer will expire.

Returns

Returns XOS_OK on success, else error code.

30.13.3.23 int32_t xos_timer_set_period (XosTimer * *timer,* uint64_t *period*)

Set the repeat period for a periodic timer. The period must be specified in system clock cycles.

If the timer is active, the change in period does not take effect until the timer expires at least once after this call. Note that setting a period of zero will effectively turn a periodic timer into a one-shot timer. Similarly, a one-shot timer can be turned into a periodic timer.

Parameters

timer	Pointer to timer object.
period	Repeat period in system clock cycles.

Returns

Returns XOS_OK on success, else error code.

30.13.3.24 int32_t xos_timer_start (XosTimer * *timer,* uint64_t *when,* uint32_t *flags,* XosTimerFunc * *func,* void * *arg*)

Start the timer, and when the timer expires, call the specified function (invoke (*fn)(arg)). If the timer is periodic, it will be automatically restarted when it expires.

The specified timer event structure must have been initialized before first use by calling xos_timer_init().

The callback function will be called in an interrupt context. Hence it is NOT safe to use any coprocessors in the function, including the FPU. If a coprocessor must be used, then its state must be saved and restored across its use.

NOTE: If you are using the timer only to wait on (via xos_timer_wait()) then it is not necessary to specify a callback function. You should pass NULL for the callback function and zero for the callback argument.

Parameters

timer	Pointer to timer event structure. Must have been initialized. May be active or not.
when	When to call the function (see flags).
flags	Set of option flags XOS_TIMER_* The following flags are mutually exclusive: • XOS_TIMER_DELTA – when is number of cycles from [see below] (default) • XOS_TIMER_PERIODIC – when is number of cycles from [see below], and timer continually re-triggers at that interval • XOS_TIMER_ABSOLUTE – when is absolute value of cycle count The following flags are mutually exclusive: • XOS_TIMER_FROM_NOW – *DELTA and *PERIODIC are relative to now (default) • XOS_TIMER_FROM_LAST – *DELTA and *PERIODIC are relative to the timer event's last specified expiry time (usually in the future if active, in the past if not, absolute 0 if was never activated).

func	Function to call (called in timer interrupt context). This argument is optional. Specify NULL if no function is to be called.
arg	Argument passed to callback function. Only relevant if 'func' is not NULL.

Returns

Returns XOS_OK on success, else error code.

30.13.3.25 int32_t xos_timer_stop (XosTimer * *timer*)

Stop the timer and remove it from the list of active timers. Has no effect if the timer is not active. Any waiting threads are woken up.

The timer structure must have been initialized at least once, else its contents are undefined and can lead to unpredictable results.

Parameters

timer	Pointer to timer object.

Returns

Returns XOS_OK on success, else error code.

30.13.3.26 int32_t xos_timer_wait (XosTimer * *timer*)

Wait on a timer until it expires or is cancelled. The calling thread will be blocked. The timer must be active. NOTE: This operation is only available if XOS_OPT_TIMER_WAIT is set to 1 in the configuration options.

Parameters

timer	Pointer to timer object.

Returns

Returns XOS_OK on normal timeout, else an error code.

30.13.3.27 static uint64_t xos_usecs_to_cycles (uint64_t *usecs*) [inline], [static]

Converts time in microseconds to CPU cycle count.

Parameters

usecs	Number of microseconds.

Returns

Equivalent number of CPU cycles.

30.14 xos_types.h File Reference

Macros

- #define XOS_NULL ((void *)0)
- #define bool _Bool
- #define false 0

  XOS definition of 'false'.
- #define true 1

  XOS definition of 'true'.

30.14.1 Macro Definition Documentation

30.14.1.1 #define bool _Bool

XOS definition of 'bool' type. Some C libraries do not support stdbool.h.

30.14.1.2 #define XOS_NULL ((void *)0)

XOS define for NULL value. This makes the NULL value independent of the C library (not all of them define NULL the same way).

Index