# SYS/BIOS POSIX Thread (pthread) Support

> **NOTE**
>
> Note, this page is obsolete as of the SYS/BIOS 6.53 release. While POSIX-on-SYS/BIOS continues to be supported (and based on the same underlying implementation and features this page describes), the implementation and delivery mechanism has evolved, making some of the details below wrong for more recent releases. Please refer to the docs\tiposix\Users_Guide.html page in the SYS/BIOS installation for more current details.

SYS/BIOS (v6.42.01 and higher) provides a subset of the POSIX thread (pthread) APIs. These include pthread threads, mutexes, read-write locks, barriers, and condition variables. The pthread APIs can simplify porting applications from a POSIX environment to SYS/BIOS, as well as allowing the same code to be compiled to run in a POSIX environment and with SYS/BIOS. As the pthread APIs are built on top of the SYS/BIOS Task and Semaphore modules, some of the APIs can be called from SYS/BIOS Tasks.

This page provides brief descriptions of the APIs supported by SYS/BIOS and any SYS/BIOS-specific details. For more detail, see the official specification (http://pubs.opengroup.org/onlinepubs/9699919799/) and documentation of the generic implementation (https://computing.llnl.gov/tutorials/pthreads/).

The POSIX APIs supported for SYS/BIOS have now been ported to the FreeRTOS operating system, with some limitations. The supported POSIX APIs for FreeRTOS are described at the end of this document.

## Contents

# Adding SYS/BIOS Support for POSIX Threads

Use the ti.sysbios.posix.Settings module to add pthread support to your application. In your application .cfg file, add the following line:

```
xdc.useModule('ti.sysbios.posix.Settings');
```

Include the SYS/BIOS pthread header file in your .c files:

```
#include <ti/sysbios/posix/pthread.h>
```

The Settings module has one public configuration parameter, supportsMutexPriority. This is a Boolean parameter that defaults to false. If set to true, the priority inheritance and priority ceiling mutex protocols will be available. Setting this parameter to true increases code size, so if these mutex protocols are not needed, supportsMutexPriority should be left as false.

Example configuration:

```
var Settings = xdc.useModule('ti.sysbios.posix.Settings');
Settings.supportsMutexPriority = true;
```

# POSIX Threads

SYS/BIOS supports both pthread and pthread_attrs APIs for POSIX threads.

Since each SYS/BIOS pthread has an underlying Task object, some of the SYS/BIOS Task APIs can be called from a pthread. These include Task_sleep(), Task_yield(), Task_self(), and Task_getPri(). Some Task APIs, such as Task_setPri(), should never be called from a pthread.

Many of the pthread APIs must be called from a Task context, and therefore, cannot be called from main(). For example, pthread_create() can be called from main(), but pthread_join() cannot. A BIOS Task can call any pthread_mutexattr API, and can create and destroy a pthread mutex, but cannot call any other pthread_mutex APIs.

## pthread_attr APIs

The pthread_attr_ APIs are used for setting/getting the attributes to be used when a thread is created. They do not modify the attributes of a thread that has already been created.

### Initialize and destroy attributes

```
int pthread_attr_init(pthread_attr_t *attr)
int pthread_attr_destroy(pthread_attr_t *attr)
```

pthread_attr_init() initializes the pthread_attr_t object pointed to by *attr* with default values. The stack size is initialized to Task_defaultStackSize and the priority to 1. The other pthread_attr_ functions can be used to change the defaults.

Return values:

- 0 – Success

pthread_attr_destroy() destroys the pthread_attr_t object pointed to by attr.

### Get and set the thread attributes detach state

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)
```

The detach state of the thread attributes can be PTHREAD_CREATE_DETACHED (detached) or PTHREAD_CREATE_JOINABLE (joinable). The default set by pthread_attr_init() is PTHREAD_CREATE_JOINABLE. When a detached thread is created, its resources can be reused as soon as the thread terminates. When a joinable thread terminates, its resources are not released until another thread joins with it. If no thread calls pthread_join() for a joinable thread, a memory leak will occur, so if you do not want a thread to be joined, create it as a detached thread.

Return values:

- 0 – Success.
- EINVAL – The detachstate passed to pthread_attr_setdetachstate() is not PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE.

### Get and set stack attributes

```
int pthread_attr_getstacksize(const pthread_attr_t *attr, size_t *stacksize)
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)
```

```
int pthread_attr_getguardsize(const pthread_attr_t *attr, size_t *guardsize)
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize)
```

```
int pthread_attr_getstack(const pthread_attr_t *attr,
                          void **stackaddr, size_t *stacksize)
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr,
                          size_t stacksize)
```

pthread_attr_init() sets the stack address attribute to NULL and stack size to Task_defaultStackSize. A stack address of NULL specifies that the thread's stack will be allocated when the thread is created. The size of the stack allocated will be *stacksize* + *guardsize*.

The *guardsize* attribute specifies the amount of extra memory allocated at the overflow end of the stack to protect from stack overflows. The default guardsize is 0. You can also create the equivalent of a non-zero guardsize by increasing the stack size.

SYS/BIOS does not check to see if the thread's stack pointer has gone into the guard area, so it is up to the application to check for this. There are stack overrun checks in SYS/BIOS, which you can find in the ti.sysbios.knl.Task documentation.

If you want to provide the memory for the stack, you can use pthread_attr_setstack(). In that case, the guardsize will be ignored.

Return values:

- 0 – Success.

### Get and set thread priority attributes

```
int pthread_attr_getschedparam(const pthread_attr_t *attr,
       struct sched_param *schedparam)
int pthread_attr_setschedparam(pthread_attr_t *attr,
       const struct sched_param *schedparam)
```

The sched_param structure has just one field for the thread's priority:

```
struct sched_param {
  int sched_priority; /* Thread execution priority */
};
```

These APIs set the thread priority in the *attr* that will be used to create a thread. After a thread has been created, you can use the pthread_getschedparam() and pthread_setschedparam() APIs to get and set the priority of an existing thread.

The priorities allowed are {-1, 1, ... ,Task_numPriorities – 1}. Programs can use sched_get_priority_min() and sched_get_priority_max() instead of using SYS/BIOS-specific values. sched_get_priority_min() returns the lowest priority, 1, and sched_get_priority_max() returns the highest priority, Task_numPriorities – 1.

A priority value of -1 prevents the thread from being scheduled immediately upon creation. In this case the thread will be put on the Task "inactive" queue. The priority can then be set later to allow the thread to be scheduled.

Return values:

- 0 – Success.
- EINVAL – pthread_attr_setschedparam() was passed a priority that is out of range.

## pthread APIs

The pthread_ APIs allow you to create, join, detach, cancel, exit, and cleanup threads.

### Create thread

```
int pthread_create(pthread_t *newthread, const pthread_attr_t *attr,
   void * (*startroutine)(void *), void *arg)
```

This function starts a new thread running the *startroutine* function, with the parameter *arg* passed to *startroutine*. If *attr* is NULL, the thread is created with default pthread attributes.

Return values:

- 0 – Success.
- ENOMEM – Failed to allocate memory for thread resources.

### Detach or join a thread

```
int pthread_detach(pthread_t pthread)
int pthread_join(pthread_t th, void **thread_return)
```

If a thread was created with the PTHREAD_CREATE_JOINABLE attribute, another thread can call pthread_join() to block until the thread exits or is cancelled. The thread's return value will be passed in *thread_return*, unless the thread was cancelled, in which case PTHREAD_CANCELED will be passed.

If a thread is joinable and no other thread has called pthread_join() with it, the thread can be changed to detached, with pthread_detach(). Once a thread is in the detached state, it can never be joined.

Return values for pthread_detach():

- 0 – Success.
- EINVAL – pthread_detach() was called after another thread has called pthread_join() on the thread.

Return values for pthread_join():

- 0 – Success.
- EINVAL – pthread_join() was called on a thread that another thread has already called pthread_join() on the thread.
- EINVAL – pthread_join() was called on a thread that is detached.
- EDEADLK – pthread_join() was called on a thread that has called pthread_join() on the calling thread.

### Cancel, exit, and cleanup

```
int pthread_setcancelstate(int state, int *oldstate)
int pthread_cancel(pthread_t pthread)
void pthread_cleanup_pop(int execute)
void pthread_cleanup_push(void (*fxn)(void *), void *arg)
```

A thread can set its cancel state to PTHREAD_CANCEL_ENABLE to allow another thread to cancel it. To prevent cancellation, the thread can set its cancel state to PTHREAD_CANCEL_DISABLE. When a thread is cancelled, any cleanup handlers that it pushed with pthread_cleanup_push() will be popped and run. If the thread is joinable, a thread that has called pthread_join() will unblock, and the cancelled thread will go on the Task "inactive" queue. Its resources will be freed in pthread_join(). If the thread is not joinable, it will be deleted immediately.

SYS/BIOS supports only asynchronous cancellation, and not deferred cancellation. The reason for this is that functions that are required by POSIX to be cancellation points, for example printf(), are not cancellation points in SYS/BIOS.

```
void pthread_exit(void *ptr)
```

A thread can exit by calling pthread_exit(), passing a return value that will be available to pthread_join() if the thread is joinable. When a thread exits, any cleanup handlers pushed with pthread_cleanup_push(), will be popped and run. If the thread is joinable, it will go on the Task "inactive" queue, and its resources will be freed in pthread_join(). If the thread is detached, it will be freed up immediately.

The cleanup handlers are defined as macros:

```
#define pthread_cleanup_push(fxn, arg) \
   do { \
        struct _pthread_cleanup_context _pthread_clup_ctx; \
        _pthread_cleanup_push(&_pthread_clup_ctx, (fxn), (arg))
```

```
#define pthread_cleanup_pop(execute) \
        _pthread_cleanup_pop(&_pthread_clup_ctx, (execute)); \
   } while (0)
```

The '{' and '}', in the macro expansions, force the push and pop macros to be paired in the same scope. Variables declared within the scope of the paired calls will only be visible within that scope.

Return values:

- 0 – Success
- EINVAL – pthread_setcancelstate() was passed a value other than PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

### Get and set a thread's priority

```
int pthread_getschedparam(pthread_t thread, int *policy,
       struct sched_param *param)
int pthread_setschedparam(pthread_t pthread, int policy,
       const struct sched_param *param)
```

The sched_priority field of the sched_param structure is used for the thread's priority:

```
struct sched_param {
   int sched_priority; /* Thread execution priority */
};
```

The priorities allowed are {-1, 1, … , Task_numPriorities – 1}. Programs can use sched_get_priority_min() and sched_get_priority_max() instead of referring to SYS/BIOS specific values. sched_get_priority_min() returns the lowest priority, 1, and sched_get_priority_max() returns the highest priority, Task_numPriorities – 1.

A priority value of -1 makes the thread inactive. The thread can later be made active by setting its priority to a value in the range {1, … , Task_numPriorities – 1}.

**NOTE:** pthread_getschedparam() returns the priority that was set through the pthread_attr_t attributes when the thread was created or through pthread_setschedparam() if that was called after the thread was created. If the thread is running at a higher priority because it owns a PTHREAD_PRIO_PROTECT or PTHREAD_PRIO_INHERIT mutex, the sched_priority returned will not reflect this. To determine the priority that the thread is actually running at, you can call Task_getPri(Task_self()) within the thread. However, you should never call Task_setPri(Task_self()) on a pthread!

Since SYS/BIOS has only a priority-based scheduling policy, the policy parameter to these APIs is ignored.

Return values:

- 0 – Success.
- EINVAL – pthread_setschedparam() was passed a priority that is out of range.

### Miscellaneous other thread APIs

```
int pthread_equal(pthread_t pt1, pthread_t pt2)
```

Returns:

- 0 – pt1 and pt2 are different threads
- 1 – pt1 and pt2 are the same threads

```
int pthread_once(pthread_once_t *once, void (*initFxn)(void))
```

pthread_once() runs an initialization routine atomically and once only. For example, suppose a set of threads all require some initialization function to be run, but this function should be run only once (for example, because it initializes a mutex). Each of these threads can call pthread_once(), passing the initialization function. The *once* parameter should be a variable, initialized to PTHREAD_ONCE_INIT, that is globally visible to all the threads calling pthread_once() with the same initFxn.

Return values:

- 0 – Success.

```
pthread_t pthread_self(void)
```

This function returns the thread handle of the calling thread, which can then be passed to other pthread APIs.

Returns:

- Handle of the calling thread.

# POSIX Mutexes

A mutex (for "mutual exclusion") is a data structure used to allow multiple threads to share a resource without accessing it simultaneously.

## Mutex Types

The following pthread mutex types are supported in SYS/BIOS:

**PTHREAD_MUTEX_NORMAL** – When this type of mutex is held, subsequent calls to lock the mutex will block, until the owner releases it. If the owner holds the mutex and tries to lock it again without first releasing it, a deadlock situation will occur. This type of mutex is analogous to a SYS/BIOS binary semaphore.

**PTHREAD_MUTEX_RECURSIVE** – A recursive mutex can be locked again by the owner of the mutex. The owner must unlock the mutex the same number of times it locked it. This type of mutex is similar to SYS/BIOS Gates, with the addition of timeouts when trying to lock the mutex.

**PTHREAD_MUTEX_ERRORCHECK** – This type of mutex is identical to the PTHREAD_MUTEX_NORMAL, with the addition of returning an error if a deadlock condition is detected. The only deadlock detection supported is when the calling thread already owns the mutex. More complex deadlock situations will not be detected. For example, the following scenario will not be detected:

```
Thread 1                Thread 2
Locks mutex A           Locks mutex B
Try to lock mutex B     Try to lock mutex A
```

**PTHREAD_MUTEX_DEFAULT** – This will be equivalent to PTHREAD_MUTEX_NORMAL.

## Mutex Protocols

The default SYS/BIOS pthread library supports these mutex protocols:

**PTHREAD_PRIO_NONE** – The owner of a mutex of this protocol will not have its priority boosted if a higher priority thread tries to lock the mutex.

**PTHREAD_PRIO_INHERIT** – If a higher priority thread blocks trying to lock a PTHREAD_PRIO_INHERIT mutex owned by a lower priority thread, the lower priority thread's priority will be boosted to the priority of the higher priority thread.

**PTHREAD_PRIO_PROTECT** – A mutex of this protocol has a priority ceiling set by the application. The priority ceiling should be set to the priority of the highest priority thread that will lock the mutex. Any thread that locks the mutex will automatically have its priority raised to the priority ceiling of the mutex.

The SYS/BIOS pthread library can be configured to only support the PTHREAD_PRIO_NONE protocol. This can reduce code size. See the section on the Settings Module below.

## pthread_mutexattr APIs

The pthread_mutexattr_ APIs are used for setting/getting the attributes to be used when a mutex is created. They do not modify the attributes of a mutex that has already been created.

The following pthread_mutexattr APIs are supported in SYS/BIOS.

### Initialize and de-initialize a pthread_mutexattr_t object

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr)
int pthread_mutexattr_init(pthread_mutexattr_t *attr)
```

The pthread_mutexattr_init() call initializes the object pointed to by attr with default values. Use the other pthread_mutexattr APIs to change attributes.

Returns:

- 0 - Success

### Get and set mutex type, protocol, and priority ceiling attributes

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type)
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *attr,
      int *prioceiling)
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,
      int *protocol)
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
      int prioceiling)
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol)
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type)
```

Returns:

- 0 – Success
- EINVAL – pthread_mutexattr_setprioceiling() was passed a prioceiling value less than 1 or greater than or equal to Task_numPriorities.
- EINVAL – pthread_mutexattr_setprotocol() was passed a protocol value that is not one of the supported mutex protocols.
- EINVAL – pthread_mutexattr_settype() was passed an invalid value for mutex type.
- ENOSYS – If the SYS/BIOS pthread library is configured with minimal support, pthread_mutexattr_getprioceiling(), pthread_mutexattr_setprioceiling(), pthread_mutexattr_getprotocol(), and pthread_mutexattr_setprotocol() will return ENOSYS.

# pthread_mutex APIs

The pthread_mutex_ APIs allow you to initialize, de-initialize, lock, unlock, and get information about mutexes.

### Initialize and de-initialize mutexes

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
int pthread_mutex_init(pthread_mutex_t *mutex,
      const pthread_mutexattr_t *attr)
```

Returns:

- 0 – Success
- ENOMEM – Mutex initialization failed to allocate memory for the mutex object.

### Get and set mutex priority ceilings

```
int pthread_mutex_getprioceiling(const pthread_mutex_t *mutex,
      int *prioceiling)
int pthread_mutex_setprioceiling(pthread_mutex_t *mutex,
      int prioceiling, int *oldceiling)
```

These APIs are applicable to a PTHREAD_PRIO_PROTECT protocol mutex only. When the priority ceiling is set, the mutex is locked, so pthread_mutex_setprioceiling() is a blocking call. If the mutex type is PTHREAD_MUTEX_NORMAL and the calling thread already owns the mutex, pthread_mutex_setprioceiling() will deadlock. If the mutex is recursive and the calling thread already owns the mutex, the thread's priority will be bumped up to the new priority ceiling, if it is higher than the thread's current priority.

Returns:

- 0 – Success.
- ENOSYS – The SYS/BIOS pthread library was configured to not support PTHREAD_PRIO_PROTECT protocol mutexes.
- EINVAL – The protocol of the mutex is not PTHREAD_PRIO_PROTECT.
- EINVAL – The priority passed to pthread_mutex_getprioceiling() is less than 1 or greater than Task_numPriorities – 1.
- EINVAL – The thread calling pthread_mutex_setprioceiling() already owns the mutex (must be of type PTHREAD_MUTEX_RECURSIVE), and the new priority ceiling is lower than the thread's priority.
- EDEADLK – The calling thread of pthread_mutex_setprioceiling() already owns the mutex and the mutex type is PTHREAD_MUTEX_ERRORCHECK.

### Lock a mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
      const struct timespec *abstime)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

The function pthread_mutex_lock() blocks until the mutex is acquired by the calling thread. The non-blocking function, pthread_mutex_trylock(), returns immediately, and if the mutex is available, the calling thread will now own it.

The pthread_mutex_timedlock() API blocks only until the specified time has been reached or the mutex becomes available, whichever happens first. Note that the time passed to pthread_mutex_timedlock() is an absolute time, not a relative timeout. The timespec structure is defined in ti/sysbios/posix/time.h:

```
struct timespec {
   time_t  tv_sec;   /* Seconds */
   long    tv_nsec;  /* Nanoseconds */
};
```

To fill in the timespec object, SYS/BIOS has a clock_gettime() API:

```
#include <ti/sysbios/posix/pthread.h>

struct timespec      abstime;
```

```
clock_gettime(CLOCK_REALTIME, &abstime);
```

Add the relative timeout value to abstime. For example:

```
abstime.tv_sec++;   /* A one second timeout */
```

Or

```
abstime.tv_nsec += 1000000;  /* A one msec timeout */
```

The *tv_nsec* value must be greater than or equal to 0, and less than 1,000,000,000, otherwise the function returns -1.

Timeouts for pthread_mutex_timedlock() must be based on the CLOCK_REALTIME clock. The CLOCK_REALTIME time is based on the SYS/BIOS ti/sysbios/hal/Seconds module. Since the Seconds count can be changed at run-time with the Seconds_set() API, time based on Seconds is not necessarily monotonic. Additionally, the Seconds count must be initialized before calling clock_gettime(CLOCK_REALTIME,...). This can be done in main() with a call to clock_settime(CLOCK_REALTIME, val) (or by calling Seconds_set()). Normally, you would pass the number of seconds elapsed since the beginning of the Epoch (Jan 1, 1970) to clock_settitme() or Seconds_set(). However, if you don't care about absolute time, you can just initialize the time to 0 as follows:

```
#include <ti/sysbios/posix/time.h>

/* Initialize the CLOCK_REALTIME clock */
clock_settime(CLOCK_REALTIME, 0);
```

clock_settime() can be called in main(), before BIOS_start() is called. Note that due to the granularity of the Seconds count and Clock ticks (typically one millisecond), the relative timeouts may not be precise. If supported on the device, the Seconds count runs off a real-time clock, which may have a low frequency (for example, 32KHz). If the device does not have a real-time clock, the Seconds count runs off the SYS/BIOS Clock tick, which is typically incremented every one millisecond. Therefore, timeouts may be fairly coarse.

Note on priority inheritance: If the priority, p, of a thread that blocks in a mutex lock call is higher than the mutex owner's priority, the priority of the mutex owner will be boosted to p. If the mutex owner is also blocked in a mutex lock call, the priority of the thread owning the mutex it is waiting on will also be raised to p. The priority raising will continue, recursively, until an owner thread is not blocked or has priority greater than or equal to p. For example, in the chain of threads in the figure below, thread $T_i$ is blocked on mutex $M_i$+1 (1 <= i < n), and thread $T_i$ owns mutex $M_i$ (1 < i <= n). The priority of thread $T_i$ I will be raised to p, if it is less than p.

$T_1$ pthread_mutex_lock($M_2$) --> $T_2$ (owner of $M_2$) --> $T_3$ (owner of $M_3$) ... --> $T_n$ (owner of $M_n$) not blocked.

There is no error checking for the situation in which one of the threads in the chain above owns a PTHREAD_PRIO_PROTECT mutex, and the inherited priority of the thread is higher than the priority ceiling of the mutex.

If pthread_mutex_timedlock() times out, and this call caused other threads' priorities to be raised, the priorities of these threads will be re-adjusted.

Returns:

- 0 – Success (Calling thread now owns the mutex)
- EINVAL – A bad time value was passed to pthread_mutex_timedlock().
- EINVAL – The mutex protocol is PTHREAD_PRIO_PROTECT and the priority of the calling thread is higher than the priority ceiling of the mutex.
- EDEADLK – The mutex type is PTHREAD_MUTEX_ERRORCHECK and the calling thread already owns the mutex.
- EBUSY – Returned by pthread_mutex_trylock() if the mutex is not available.
- ETIMEDOUT – The function pthread_mutex_timedwait() timed out before acquiring the mutex.

## Unlock a mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

This function unlocks a previously locked mutex. If the mutex is of type PTHREAD_MUTEX_RECURSIVE, each successful mutex lock call must be matched by a pthread_mutex_unlock() call.

Returns:

- 0 – Success
- EPERM – The calling thread does not own the mutex.

# POSIX Barriers

Pthread barriers are a way of synchronizing threads. A barrier is initialized with pthread_barrier_init() to a count, equal to the number of threads to synchronize with the barrier. The first count – 1 threads to call pthread_barrier_wait() will block. When the next thread calls pthread_barrier_wait(), all threads become unblocked.

## pthread_barrierattr APIs

```
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr)
int pthread_barrierattr_init(pthread_barrierattr_t *attr)
```

Returns:

- 0 – Success.

## pthread_barrier APIs

```
int pthread_barrier_destroy(pthread_barrier_t *barrier)
int pthread_barrier_init(pthread_barrier_t *barrier,
       constpthread_barrierattr_t *attr, unsigned count)
int pthread_barrier_wait(pthread_barrier_t *barrier)
```

Returns:

- 0 – Success
- PTHREAD_BARRIER_SERIAL_THREAD – This is returned to the last thread that called pthread_barrier_wait(), unblocking the first count – 1 threads.

# POSIX Condition Variables

Please refer to any POSIX documentation for a description and usage of condition variables. The following APIs are supported by SYS/BIOS.

## pthread_condattr APIs

```
int pthread_condattr_destroy(pthread_condattr_t *attr)
int pthread_condattr_init(pthread_condattr_t * attr)
```

Returns:

- 0 – Success

## pthread_cond APIs

### Initialization and destruction

```
int pthread_cond_destroy(pthread_cond_t *cond)
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)
```

Returns:

- 0 – Success.

### Waiting on a condition variable

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
       const struct timespec *abstime)
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Please see the section on mutexes for a description of the timeout parameter passed to pthread_cond_timedwait(). This is an absolute time value, not a relative time. Only the CLOCK_REALTIME clock is supported for timed waits.

Returns:

- 0 – Success
- EINVAL – An invalid timeout value was passed to pthread_cond_timedwait(): abstime->nsec < 0 or >= 1,000,000,000.
- ETIMEDOUT – pthread_cond_timedwait() timed out.

### Signaling a condition variable

```
int pthread_cond_broadcast(pthread_cond_t *cond)
int pthread_cond_signal(pthread_cond_t *cond)
```

Returns:

- 0 – Success.

# POSIX Read-Write Locks

Read-write locks allow access by multiple readers or one writer to the lock. When a writer owns the lock, no other writers and no readers can acquire the lock. When a reader owns the lock, other readers can also acquire the lock, but no writer can acquire the lock. SYS/BIOS does not give preference to writers; when the lock is available, it goes to the highest priority thread waiting on the lock.

## pthread_rwlockattr APIs

```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr)
int pthread_rwlockattr_init(pthread_rwlockattr_t * attr)
```

Returns:

- 0 – Success.

## pthread_rwlock APIs

### Initialize and destroy locks

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
       const pthread_rwlockattr_t *attr)
```

Returns:

- 0 – Success.

### Acquire a lock for reading

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,
       const struct timespec *abstime)
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock)
```

See the section on mutexes for a description of timed waits. Multiple threads can own the lock for reading. Only the CLOCK_REALTIME clock is supported for pthread_rwlock_timedrdlock().

Returns:

- 0 – Success. The calling thread owns the lock for reading.
- EBUSY – Returned by pthread_rwlock_tryrdlock() if the lock is owned by a writer.
- EINVAL – A bad timeout value passed to pthread_rwlock_timedrdlock().
- ETIMEDOUT – pthread_rwlock_timedrdlock() timed out waiting for the lock.

### Acquire a lock for writing

```
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,
       const struct timespec *abstime)
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock)
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock)
```

See the section on mutexes for a description on timed waits. Only the CLOCK_REALTIME clock is supported for pthread_rwlock_timedwrlock().

Returns:

- 0 – Success. The calling thread owns the lock for writing.
- EINVAL – A bad timeout value was passed to pthread_rwlock_timedwrlock().
- EBUSY – Returned by pthread_rwlock_trywrlock() when the lock is owned by readers or another writer.
- ETIMEDOUT – pthread_rwlock_timedwrlock() timed out waiting for the lock.

### Unlock a read-write lock

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock)
```

# Clock APIs

The following clock APIs are supported in SYS/BIOS:

```
#include <ti/sysbios/posix/time.h>
```

```
int    clock_gettime(clockid_t clockId, struct timespec *ts);
int    clock_settime(clockid_t clockId, const struct timespec *ts);
```

If you plan to use these APIs, include the header file <ti/sysbios/posix/pthread.h>. The clockId parameter can be CLOCK_MONOTONIC or CLOCK_REALTIME. The CLOCK_REALTIME clock is implemented using the SYS/BIOS ti.sysbios.hal.Seconds module. clock_settime() must be called before calling clock_gettime(CLOCK_REALTIME, ts). For CLOCK_MONOTONIC, the implementation of clock_gettime() uses the SYS/BIOS ti.sysbios.knl.Clock system clock. Calling clock_settime(CLOCK_MONOTONIC, ts) returns -1.

# Message Queues

SYS/BIOS versions 6.46.00 and higher support the following message queue APIs:

```
int mq_close(mqd_t mqdes);
int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat);
mqd_t mq_open(const char *name, int oflags, ...);
long mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
       unsigned int *msg_prio);
int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
       unsigned int msg_prio);
int mq_setattr(mqd_t mqdes, const struct mq_attr *mqstat,
       struct mq_attr *omqstat);
long mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len,
       unsigned int *msg_prio, const struct timespec *abstime);
int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len,
       unsigned int msg_prio, const struct timespec *abstime);
int mq_unlink(const char *name);
```

Include the header file <ti/sysbios/posix/mqueue.h> to use these APIs. Note: Only the CLOCK_REALTIME clock is supported for timed send and receive calls.

# Semaphores

The following semaphore APIs are supported in SYS/BIOS:

```
int    sem_destroy(sem_t *sem);
int    sem_getvalue(sem_t *sem, int *value);
int    sem_init(sem_t *sem, int pshared, unsigned value);
int    sem_post(sem_t *sem);
int    sem_timedwait(sem_t *sem, const struct timespec *abstime);
int    sem_trywait(sem_t *sem);
int    sem_wait(sem_t *sem);
```

If you plan to use these APIs, include the header file <ti/sysbios/posix/semaphore.h>. Refer to the POSIX standards documentation (http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/semaphore.h.html) for a description of these APIs.

Note: Only the CLOCK_REALTIME clock is supported for sem_timedwait().

# Timers

SYS/BIOS versions 6.42.02 and higher support the following timer APIs:

```
int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid);
int timer_delete(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *its);
int timer_settime(timer_t timerid, int flags, const struct itimerspec *value, struct itimerspec *ovalue);
```

### Create and Delete a Timer

```
int timer_create(clockid_t clockid, struct sigevent *sev, timer_t *timerid)
int timer_delete(timer_t timerid)
```

The timer_create function creates a new timer object. A handle to the newly created timer is stored in the *timerid* parameter. The *clockid* parameter (CLOCK_MONOTONIC or CLOCK_REALTIME) determines the clock used for timer_settime() and timer_gettime(). Be sure to use the same clock id passed to timer_create() when computing timeouts passed to timer_settime().

The *sev* parameter points to a sigevent structure (below), and must not be NULL, since it contains the notification function that is called when the timer expires. Since SYS/BIOS does not have support for signals, the *sigev_signo* parameter in the structure is not used. However, the SYS/BIOS implementation of timer allows the application to specify whether the notification function is called from Swi context or from Task context. If the *sigev_notify* field is set to *SIGEV_SIGNAL*, the notification will be called from a Swi. In this case, the notification function must not make any blocking calls.

If *sigev_notify* is set to *SIGEV_THREAD*, a thread will be created that calls the notification function. The thread will be deleted when the timer is deleted.

```
typedef struct sigevent {
    int sigev_notify;
    int sigev_signo;
    sigval sigev_value;
    void (*sigev_notify_function)(sigval val);
    pthread_attr_t *sigev_notify_attributes;
} sigevent;
```

Return values:

- 0 – Success.
- -1 - An invalid clock id was passed to timer_create() or memory allocation failed.

**Note:**

For a *SIGEV_SIGNAL* timer, the notification function must not be NULL. Since the notification function is called in a **Swi** context, it must not make any blocking calls.

For a *SIGEV_THREAD* timer, a thread is created to call the notification. If the sigev_notify_attributes are NULL, the thread is created with default thread attributes, so its priority will be low. If you want your notification to run at a high priority, make sure to set the priority in *sigev_notify_attributes*.

**Example**

This is an example of timer_create() code that builds for both Linux and SYS/BIOS. As such, some of the fields that are ignored by the SYS/BIOS timer implementation, are set. The notification function, *notifyFxn*, is called when the timer expires. The address of the variable, *notifyCount* is passed to *notifyFxn*. The *clockid* parameter is ignored in the SYS/BIOS implementation. Note that the header file, ti/sysbios/posix/pthread.h must be included.

```
#include <ti/sysbios/posix/pthread.h>
```

```
sigevent    sev;
int         retc;
timer_t     timer;
```

```
sev.sigev_notify = SIGEV_SIGNAL;
sev.sigev_value.sival_ptr = &notifyCount;
sev.sigev_notify_function = &notifyFxn;
sev.sigev_notify_attributes = NULL;
```

```
retc = timer_create(CLOCK_MONOTONIC, &sev, &timer);
...
timer_delete(timer);
```

```
void notifyFxn(sigval val)
{
    /* This notification can be safely called from Swi context */
    int *count = (int *)val.sival_ptr;
    *count = *count + 1;
}
```

**Example**

This example creates a timer whose notification is called from a thread.

```
#include <ti/sysbios/posix/pthread.h>
```

```
pthread_attr_t      pattrs;
struct sched_param  priParam;
sigevent    sev;
int         retc;
timer_t     timer;
```

```
retc = pthread_attr_init(&pattrs);
priParam.sched_priority = TIMERPRI;
retc = pthread_attr_setschedparam(&pattrs, &priParam);
```

```
sev.sigev_notify = SIGEV_THREAD;
sev.sigev_value.sival_int = 1;
sev.sigev_notify_function = &timerFxn;
sev.sigev_notify_attributes = &pattrs;
```

```
retc = timer_create(CLOCK_MONOTONIC, &sev, &timer);
...
timer_delete(timer);
```

```
void timerFxn(sigval val)
{
    timerCount++;
```

```
    Semaphore_pend(biosSem, BIOS_WAIT_FOREVER);
    globalCount++;
    Semaphore_post(biosSem);
}
```

## Get and Set a Timer's Expiration Time

The following APIs get and set a timer's expiration. The *timespec* and *itimerspec* are shown below for reference.

```
int timer_gettime(timer_t timerid, struct itimerspec *its);
int timer_settime(timer_t timerid, int flags, const struct itimerspec *value, struct itimerspec *ovalue);
```

```
struct timespec {
    time_t  tv_sec;   /* Seconds */
    long    tv_nsec;  /* Nanoseconds */
};
```

```
struct itimerspec {
    struct timespec it_interval;  /* Timer interval */
    struct timespec it_value;     /* Timer expiration */
};
```

Returns:

- 0 - Success
- -1 - A bad time value was passed to timer_settime()

The timer_settime() API can be passed either a relative time or an absolute time when the timer expires. When this API is called, the timer will be armed if the *value->it_value* parameter specifies a non-zero time. If *value->it_value* specifies a time of 0, the timer will be dis-armed. A non-zero *value->it_interval* will cause the timer to fire periodically at the specified interval.

#### Example

Set a timer to go off periodically every 500 milliseconds:

```
#include <ti/sysbios/posix/pthread.h>
```

```
    timer_t           timer;
    struct itimerspec its;
    struct itimerspec oldIts;
    struct timespec   abstime;
    int               retc;
```

```
    its.it_interval.tv_sec = 0;
    its.it_interval.tv_nsec = 500000000;
    its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = 500000000;
```

```
    retc = timer_settime(timer, 0, &its, NULL);
```

Set the timer to go off after 1 second, using absolute time:

```
#include <ti/sysbios/posix/pthread.h>
```

```
    clock_gettime(CLOCK_MONOTONIC, &abstime);
```

```
    its.it_interval.tv_sec = 0;
    its.it_interval.tv_nsec = 0;
    its.it_value.tv_sec = abstime.tv_sec + 1;
    its.it_value.tv_nsec = abstime.tv_nsec;
```

```
    retc = timer_settime(timer, TIMER_ABSTIME, &its, NULL);
```

Stop the timer:

```
    its.it_value.tv_sec = 0;
    its.it_value.tv_nsec = 0;
```

```
    retc = timer_settime(timer, 0, &its, &oldIts);
```

# Summary of Pthread APIs supported in SYS/BIOS

The following tables show pthread APIs supported in SYS/BIOS:

| pthread API | Summary | Callable from main() | Callable from Task |
|---|---|---|---|
| **pthead_attr_** | | | |
| pthead_attr_destroy | De-initialize a pthread attributes object | Yes | Yes |
| pthead_attr_getdetachstate | Get the detach state of the pthread attributes | Yes | Yes |
| pthead_attr_getguardsize | Get the stack guard size attribute | Yes | Yes |
| ~~pthead_attr_getinheritsched~~ | Not supported - SYS/BIOS scheduling policy is fixed | | |
| pthead_attr_getschedparam | Get pthread priority attribute | Yes | Yes |
| ~~pthead_attr_getschedpolicy~~ | Not supported - SYS/BIOS scheduling policy is fixed | | |
| ~~pthead_attr_getscope~~ | Not supported - SYS/BIOS has no notion of processes | | |
| pthead_attr_getstack | Get stack size and address attributes | Yes | Yes |
| pthead_attr_getstacksize | Get stack size attribute | Yes | Yes |
| pthead_attr_init | Initialize a pthread attributes object | Yes | Yes |
| pthead_attr_setdetachstate | Set the detach state. Determines whether or not the thread can be joined. | Yes | Yes |
| pthead_attr_setguardsize | Set the stack guardsize. Equivalent to increasing stack size when guardsize is non-zero | Yes | Yes |
| ~~pthead_attr_setinheritsched~~ | Not supported - SYS/BIOS scheduling policy is fixed | | |
| pthead_attr_setschedparam | Use to set the pthread priority | Yes | Yes |
| ~~pthead_attr_setschedpolicy~~ | Not supported - SYS/BIOS scheduling policy is fixed | | |
| ~~pthead_attr_setscope~~ | Not supported - SYS/BIOS has no notion of processes | | |
| pthead_attr_setstack | Set the stack size and address | Yes | Yes |
| pthead_attr_setstacksize | Set the stack size | Yes | Yes |
| | | | |
| **pthread_** | | | |
| pthread_cancel | Terminate a thread | No | Yes |
| pthread_cleanup_pop | Pop a cleanup handler | No | No |
| pthread_cleanup_push | Push a cleanup handler | No | No |
| pthread_create | Create a pthread | Yes | Yes |
| pthread_detach | Make a pthread not joinable | No | Yes |
| pthread_equal | Determine equality of two pthread handles | Yes | Yes |
| pthread_exit | Terminate the calling thread | No | No |
| ~~pthread_getconcurrency~~ | Not supported | | |
| ~~pthread_getcpuclockid~~ | Not supported | | |
| pthread_getschedparam | Use to get priority of a pthread | No | Yes |
| pthread_getspecific | Get thread-specific data for calling thread | No | No |
| pthread_join | Wait for pthread to terminate | No | Yes |
| pthread_key_create | Create a thread-specific data key | No | No |
| pthread_key_delete | Delete a thread-specific data key | No | No |
| pthread_once | Run an initialization routine once | No | Yes |
| pthread_self | Get the handle of the calling pthread | No | No |
| pthread_setcancelstate | Control the canelability of the calling pthread | No | No |
| ~~pthread_setcanceltype~~ | Not supported. Only asynchronous cancellation is supported. | | |
| ~~pthread_setconcurrency~~ | Not supported | | |
| pthread_setschedparam | Use to set priority of a pthread | No | No |
| ~~pthread_setschedprio~~ | Not supported. Use pthread_setschedparam to set priority | | |
| pthread_setspecific | Set thread-specific data for calling thread | No | No |
| ~~pthread_testcancel~~ | Not supported | | |
| | | | |
| **pthread_barrierattr_** | | | |
| pthread_barrierattr_destroy | De-initialize pthread_barrier attrs | Yes | Yes |
| ~~pthread_barrierattr_getpshared~~ | Not supported. SYS/BIOS has no notion of processes | | |
| pthread_barrierattr_init | Initialize pthread_barrier attrs | Yes | Yes |
| ~~pthread_barrierattr_setpshared~~ | Not supported. SYS/BIOS has no notion of processes | | |
| | | | |
| **pthread_barrier_** | | | |
| pthread_barrierdestroy | Free resources allocated for a pthread_barrier object | Yes | Yes |
| pthread_barrierinit | Allocate resources for a pthread_barrier object | Yes | Yes |
| pthread_barrierwait | Wait on a pthread_barrier | No | Yes |
| | | | |
| | | | |

| | | | |
|---|---|---|---|
| **pthread_condattr_** | | | |
| pthread_condattrdestroy | De-initialize pthread_cond attributes | Yes | Yes |
| ~~pthread_condattrgetclock~~ | Not supported | | |
| ~~pthread_condattrgetpshared~~ | Not supported | | |
| pthread_condattrinit | Initialize pthread_cond attributes | Yes | Yes |
| ~~pthread_condattrsetclock~~ | Not supported | | |
| ~~pthread_condattrsetpshared~~ | Not supported | | |
| | | | |
| **pthread_cond_** | | | |
| pthread_cond_broadcast | Unblock all threads blocked on a condition variable | No | Yes |
| pthread_cond_destroy | Free resources allocated for a condition variable | Yes | Yes |
| pthread_cond_init | Allocte and initialize a condition variable | Yes | Yes |
| pthread_cond_signal | Unblock a thread waiting on a condition variable | No | Yes |
| pthread_cond_timedwait | Wait on a condition variable with a timeout | No | Yes |
| pthread_cond_wait | Wait on a condition variable | No | Yes |
| | | | |
| **pthread_mutexattr_** | | | |
| pthread_mutexattr_destroy | De-initialize pthread_mutex attributes | Yes | Yes |
| pthread_mutexattr_getprioceiling | Get the priority ceiling of the mutex attributes. Not available with minimal pthread configuration. | Yes | Yes |
| pthread_mutexattr_getprotocol | Get to protocol of the mutex attributes. Not available with minimal pthread configuration. | Yes | Yes |
| ~~pthread_mutexattr_getpshared~~ | Not supported | | |
| pthread_mutexattr_gettype | Get the type of the mutex attributes (eg, normal, recursive) | None | |
| pthread_mutexattr_init | Initialize pthread_mutex attributes | None | |
| pthread_mutexattr_setprioceiling | Set the priority ceiling of the mutex attributes. Not available with minimal pthread configuration. | Yes | Yes |
| pthread_mutexattr_setprotocol | Set the protocol ceiling of the mutex attributes. Not available with minimal pthread configuration. | Yes | Yes |
| ~~pthread_mutexattr_setpshared~~ | Not supported | | |
| pthread_mutexattr_settype | Set the type of the mutex attributes (eg, normal, recursive) | Yes | Yes |
| | | | |
| **pthread_mutex_** | | | |
| pthread_mutex_destroy | Free resources allocated for a pthread mutex | Yes | Yes |
| pthread_mutex_getprioceiling | Get the priority ceiling of a mutex with protocol PTHREAD_PRIO_PROTECT. Not available with minimal pthread configuration. | Yes | Yes |
| pthread_mutex_init | Allocate and initialize a pthread mutex | Yes | Yes |
| pthread_mutex_lock | Lock a mutex. Blocks until mutex is available. | No | No |
| pthread_mutex_setprioceiling | Set the priority ceiling of a mutex with protocol PTHREAD_PRIO_PROTECT. May change the priority of the calling thread if it owns the mutex. Not available with minimal pthread configuration. | No | Yes |
| pthread_mutex_timedlock | Wait for a mutex with a timeout. | No | No |
| pthread_mutex_trylock | Lock a mutex if it is available returns without blocking. | No | No |
| pthread_mutex_unlock | Unlock a mutex owned by the calling thread. | No | No |
| | | | |
| **pthread_rwlock_attr_** | | | |
| pthread_rwlock_attr_destroy | De-initialize pthread rwlock attributes | Yes | Yes |
| ~~pthread_rwlock_attr_getpshared~~ | Not supported | | |
| pthread_rwlock_attr_init | Initialize pthread rwlock attributes | Yes | Yes |
| ~~pthread_rwlock_attr_setpshared~~ | Not supported | | |
| | | | |
| **pthread_rwlock_** | | | |
| pthread_rwlock_destroy | Free resources allocated for a pthread rwlock object | Yes | Yes |
| pthread_rwlock_init | Allocate resources and initialize a pthread rwlock object | Yes | Yes |
| pthread_rwlock_rdlock | Acquire rwlock for reading. Blocks until lock is acquired. | No | Yes |
| pthread_rwlock_timedrdlock | Lock rwlock for reading with a timeout | No | Yes |
| pthread_rwlock_timedwrlock | Lock rwlock for writing with a timeout | No | Yes |
| pthread_rwlock_tryrdlock | Lock rwlock for reading if available. Returns immediately | No | Yes |
| pthread_rwlock_trywrlock | Lock rwlock for writing if available. Returns immediately | No | Yes |
| pthread_rwlock_unlock | Unlock a previously locked rwlock | No | Yes |
| pthread_rwlock_wrlock | Lock rwlock for writing. Blocks until lock is acquired. | No | Yes |

# Other POSIX APIs supported in SYS/BIOS

SYS/BIOS also has support for POSIX semaphores and timers (BIOS versions 6.42.02 and higher). The table below lists the supported semaphore and timer APIs.

| API | Summary | Callable from main() | Callable from Task |
|---|---|---|---|
| **clock** | | | |
| clock_gettime | Get the current time. | Yes | Yes |
| clock_nanosleep | Sleep using specified clock. | No | Yes |
| clock_settime | Set the current time for the CLOCK_REALTIME clock. | Yes | Yes |
| **message queue** | | | |
| mq_close | Close a message queue. | Yes | Yes |
| mq_getattr | Get message queue attributes. | Yes | Yes |
| mq_open | Open a message queue. | Yes | Yes |
| mq_receive | Wait for a message. | No | Yes |
| mq_send | Send a message (may block if queue is full). | No | Yes |
| mq_setattr | Set message queue attributes. | Yes | Yes |
| mq_timedreceive | Timed wait for a message. | No | Yes |
| mq_timedsend | Timed wait to send a message. | No | Yes |
| mq_unlink | Last call frees message queue. | Yes | Yes |
| **semaphore** | | | |
| sem_destroy | De-initialize a semaphore. | Yes | Yes |
| sem_getvalue | Get the semaphore count. | Yes | Yes |
| sem_init | Initialize a semaphore. | Yes | Yes |
| sem_post | Post a semaphore. | Yes | Yes |
| sem_timedwait | Pend on a semaphore with a timeout. | No | Yes |
| sem_trywait | Acquire a semaphore if it is available. | No | Yes |
| sem_wait | Pend on a semaphore. | No | Yes |
| **timer** | | | |
| timer_create | Create a timer. | Yes | Yes |
| timer_delete | Delete a timer. | Yes | Yes |
| timer_gettime | Get a timer's time to expiration. | Yes | Yes |
| timer_settime | Set a timer's time to expiration. | Yes | Yes |

# POSIX for FreeRTOS

The same Posix APIs that are supported for SYS/BIOS are also supported for FreeRTOS, except the following.

- pthread_attr_setstack() – There is no way to pass a stack to xTaskCreate().

Priority inheritance for mutexes is handled by FreeRTOS. The priority of a task that owns a mutex is temporarily raised, if a task of higher priority attempts to acquire the mutex. Therefore, mutex protocols are not supported in FreeRTOS Posix.

- pthread_mutexattr_getprotocol(), pthread_mutexattr_setprotocol() - not supported for FreeRTOS
- pthread_mutexattr_getprioceiling(), pthread_mutexattr_setprioceiling() - not supported for FreeRTOS

- clock_settime() – CLOCK_REALTIME is not supported, only CLOCK_MONOTONIC is supported. clock_settime() returns -1. clock_gettime() calls xTaskGetTickCount() (there is no check on the clock id).

Other differences between FreeRTOS POSIX and SYS/BIOS POSIX:

- timer_settime() – This is a blocking call, unlike in SYS/BIOS.

- Task cleanup is done by the idle thread in FreeRTOS. If pthreads are deleted, make sure the idle task gets a chance to run to free RTOS allocated memory for the deleted thread.

{{

1. switchcategory:MultiCore=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum

Keystone=

- For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum
- For questions related to the BIOS MultiCore SDK (MCSDK),

C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments*

DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article* ***SYS/BIOS***

MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the*

OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article* ***SYS/BIOS***

OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article*

MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only*

*For technical s please post you questions at http://e2e.ti.cor Please post on comments abo article* ***SYS/BI POSIX Thread (pthread) Sup*** *here.*

}}

Please post only comments related to the article **SYS/BIOS POSIX Thread (pthread) Support** here.

please use the BIOS Forum

Please post only comments related to the article **SYS/BIOS POSIX Thread (pthread) Support** here.

*about the article* ***SYS/BIOS POSIX Thread (pthread) Support*** *here.*

***POSIX Thread (pthread) Support*** *here.*

*article* ***SYS/BIOS POSIX Thread (pthread) Support*** *here.*

***POSIX Thread (pthread) Support*** *here.*

***SYS/BIOS POSIX Thread (pthread) Support*** *here.*

*comments about the article* ***SYS/BIOS POSIX Thread (pthread) Support*** *here.*

## Links

Amplifiers & Linear
Audio
Broadband RF/IF & Digital Radio
Clocks & Timers
Data Converters
DLP & MEMS
High-Reliability
Interface
Logic
Power Management
Processors

- ARM Processors
- Digital Signal Processors (DSP)
- Microcontrollers (MCU)
- OMAP Applications Processors

Switches & Multiplexers
Temperature Sensors & Control ICs
Wireless Connectivity