CrossWorks Tasking Library

Version: 3.0

Contents

Tasking Library User Guide

This document describes the *CrossWorks Tasking Library* (CTL). The tasking (aka *multitasking*) library provides a multi-priority, preemptive, task switching and synchronization facility. Additionally, it supports a timer, interrupt service routines, and memory-block allocation.

This document is divided into three parts:

- *A whistle-stop tour of the tasking library* introduces the key concepts.
- *Putting the tasking library to use* looks in-depth at the tasking library and how to use it in your applications.
- *Reference information* is a concise reference for each function provided in the tasking library.

Overview

The tasking library enables your application to employ multiple tasks. Tasks are typically used for processing that may suspend execution while other activities occur. For example, you may have a protocol-processing task, a user-interface task, and a data-acquisition task.

Each task has its own *task stack*, which is used to store local variables and function-return information. The task stack is also used to store the CPU execution context when the task isn't executing. The CPU execution context of a task varies between machine architectures; it is typically the subset of the CPU register values that enable a task to be descheduled at any point during its execution.

The process of changing the CPU registers from one task to another is termed *task*switching}. Task switching occurs when a CTL function is called, either from a task or from an interrupt service routine (ISR), and there is a runnable task with higher priority than the executing task. Task switching also occurs when there is a runnable task of the same priority as the executing task, if the executing task has exceeded its time-slice period. If you have more than one runnable task of the same priority, the next task (modulo priority) after the executing task is selected. This is sometimes called *round-robin scheduling*.

There is a single task list and it is kept in priority-sorted order. The task list is updated when tasks are created and deleted, and when their priority changes. The task list is traversed when a CTL function is called that could change the execution state of a task. While the task list is modified or traversed, global interrupts are disabled. Consequently, the length of the interrupt-disable period depends on the number of tasks in the task list, and the priority and type of the task affected by the CTL operation.

If you require a simple, deterministic (sometimes called *real-time*) system, you should ensure that each task has a unique priority. The task switching will always select the highest-priority task that is runnable.

CTL has a pointer to the executing task. There must always be a task executing; if there isn't, a CTL error is signaled. Typically, there will be an idle task that loops and, perhaps, puts the CPU into power-saving mode.

Global interrupts will be enabled when a task switch occurs, so you can safely call tasking library functions while interrupts are disabled.

Task synchronization and resource allocation

The CTL provides several mechanisms to synchronize execution of tasks, to serialize resource access, and to provide high-level communication.

- *Event Sets*: An event set is a word-sized variable, and tasks can wait for its specific bits (representing events) to be set to 1. Events can be used for synchronization and to serialize resource access. Events can be set by interrupt service routines.
- *Semaphores*: A semaphore is a word size variable which tasks can wait for to be non-zero. Semaphores can be used be used for synchronization and to serialize resource access. Semaphores can be signaled by interrupt service routines.

- *Mutexes*: A mutex is a structure that can be used to serialize resource access. Unlike semaphores, mutexes cannot be used by interrupt service routines, but do provide extra features that make mutexes preferable to semaphores for serializing resource access.
- *Message Queues*: A message queue is a structure that enables tasks to post and receive data. Message queues are used to provide a buffered communication mechanism. Messages can be sent by interrupt service routines.
- *Byte Queues*: A byte queue is a specialization of a message queue; i.e., it is a message queue in which the messages are one byte in size. Byte queues can be sent by interrupt service routines.
- *Interrupt Enable and Disable*: The tasking library provides functions that enable and disable the processor's global interrupts. These functions can be used to provide a time-critical, mutual-exclusion facility.

Note that all task synchronization is priority based, i.e., the highest-priority task that is waiting will be scheduled first.

Timer support

If your application can provide a periodic timer interrupt, you can use the timer facility of the CTL. This facility enables time slicing of equal-priority tasks, allows tasks to delay, and provides a timeout capability when waiting for something. The timer is a software counter that is incremented by your timer interrupt. The counter is typically a millisecond counter, but you can change the timer's increment to reduce the interrupt frequency.

Memory allocation support

The CTL provides a simple memory block allocator that can be used in situations for which the standard C **malloc** and **free** functions are either too slow or may block the calling task.

C library support

The CTL provides the functions required of the CrossWorks C library for multi-threading.

Tasks

Each task has a corresponding task structure that contains the following information:

- When the task isn't executing, a pointer to the stack containing the execution context.
- The priority of the task; the lowest priority is 0, the highest is 255.
- The state of the task, runnable or waiting.
- A pointer to the next task.
- If the task is waiting for something, the details of what it is waiting for.
- Thread-specific data such as **errno**.
- A pointer to a null-terminated string that names the task for debugging purposes.

Creating a task

You allocate task structures by declaring them as C variables.

```
CTL_TASK_t mainTask;
```

You create the first task by using ctl_task_init to turn the main program into a task. This function takes a pointer to the task structure that represents the main task, its priority, and a name as parameters.

```
ctl_task_init(&mainTask, 255, "main");
```

This function must be called before any other CrossWorks tasking library calls. The priority (second parameter) must be between 0 (the lowest priority) and 255 (the highest priority). It is advisable to create the first task with the highest priority, which enables it to create other tasks without being descheduled. The name should point to a zero-terminated ASCII string, which is shown in the **Threads** window.

You can create other tasks with the function **ctl_task_run**, which initializes a task structure and may cause a context switch. You supply the same arguments as for **ctl_task_init**, together with the function the task will run and the memory the task will use for its stack.

The function a task will run should take a **void *** parameter and not return any value.

```
void task1Fn(void *parameter)
{
  // task code goes in here
}
```

The **parameter** value is supplied to the function by the **ctl_task_run** call. Note that, when a task function returns, the **ctl_task_die** function is called, terminating the task.

You must allocate the stack for the task as a C array of **unsigned** elements.

```
unsigned task1Stack[CTL_CPU_STATE_WORD_SIZE+32];
```

The stack size you need depends on the CPU (i.e., the number of registers that must be saved), the function calls the task will make, and (again depending on the CPU) the stack used for interrupt service routines. The macro **CTL_CPU_STATE_WORD_SIZE** can be used to determine the number of words required to save the CPU state, the other values you must supply. Running out of stack space is a common problem for multitasking systems, and the resulting error behavior is often misleading. The **ctl_task_run** function will initialize the stack to the word value 0xcdcdcdcd which will make it easier to check the stack's contents with the CrossWorks debugger if problems should occur.

Your **ctl_task_run** function call should look something like this:

```
ctl_task_run(&task1Task,
             12,
             task1Fn,
             0,
             "task1",
             sizeof(task1Stack) / sizeof(unsigned),
             task1Stack,
             0);
```

The first parameter is a pointer to the task structure. The second parameter is the priority (in this case 12) at which the task will start executing. The third parameter is a pointer to the function to execute (in this case **task1Fn**). The fourth parameter is the value supplied to the task function (zero, in this case). The fifth parameter is a null-terminated string that names the task for debug purposes. The sixth parameter is the size of the stack, in words. The seventh parameter is the pointer to the stack. The last parameter is for systems that have a separate *call stack*, and its value is the number of words to reserve for that stack.

Changing a task's priority

You can change the priority of a task using **ctl_task_set_priority**. It takes a pointer to a task structure and the new priority as parameters, and returns the old priority.

```
old_priority = ctl_task_set_priority(&mainTask, 255); // lock scheduler
//
// ... your critical code here
//
ctl_task_set_priority(old_priority);
```

To enable time slicing, you need to set the **ctl_timeslice_period** variable before any task scheduling occurs.

```
ctl_timeslice_period = 100; // time slice period of 100 ms
```

If you want finer control over the scheduling of tasks, you can call **ctl_task_reschedule**. The following example turns **main** into a task and creates a second task. The main task ultimately will be the lowest-priority task that switches the CPU into a power-saving mode when it is scheduled—this satisfies the requirement of always having a task to execute and enables a simple, power-saving system to be implemented.

```
#include <ctl.h>
```

```
void task1(void *p)
{
  // task code; on return, the task will be terminated.
}

static CTL_TASK_t mainTask, task1Task;
static unsigned task1Stack[64];
```

```
int main(void)
{
  // Turn myself into a task running at the highest priority.
  ctl_task_init(&mainTask, 255, "main");

  // Initialize the stack of task1.
  memset(task1Stack, 0xba, sizeof(task1Stack));

  // Prepare another task to run.
  ctl_task_run(&task1Task, 1, task1, 0, "task1",
               sizeof(task1Stack) / sizeof(unsigned),
               task1Stack, 0);

  // Now that all the tasks have been created, go to the lowest priority task.
  ctl_task_set_priority(&mainTask, 0);

  // Main task, if activated because task1 is suspended, just
  // enters low-power mode and waits for task1 to run again
  // (for example, because an interrupt wakes it).
  for (;;)
    {
      // Go into low-power mode.
      sleep();
    }
}
```

Note that, initially, the main task is assigned the highest priority while it creates the other tasks; then it changes its priority to the lowest value. This technique can be used, when multiple tasks are to be created, to ensure all the tasks are created before they start to execute.

Note the use of **sizeof** when passing the stack size to **ctl_task_run**.

Event sets

Event sets are a versatile way to communicate between tasks, manage resource allocation, and synchronize tasks.

An event set is a means to synchronize tasks with other tasks and with interrupt service routines. An event set contains a set of events (one per bit), and tasks can wait for one or more of these bits to be set (i.e., to have the value 1). When a task waits on an event set, the events it is waiting for are matched against the current values—if they match, the task can still execute. If they don't match, the task is put on the task list with details about the event set and the events for which the task is waiting.

You allocate an event set by declaring it as C variable:

```
CTL_EVENT_SET_t e1;
```

A **CTL_EVENT_SET_t** is a synonym for an **unsigned** type. Thus, when an **unsigned** is 16 bits wide, an event set will contain 16 events; and when it consists of 32 bits, an event set will contain 32 events.

An event set must be initialized before any tasks can use it. To initialize an event set, use **ctl_events_init**:

```
ctl_events_init(&e1, 0);
```

You can set and clear events in an event set using the **ctl_events_set_clear** function.

```
ctl_events_set_clear(&e1, 1<<0, 1<<15);
```

This example will set the bit-zero event and clear the bit-15 event. If any tasks are waiting on this event set, the events they are waiting on will be matched against the new event set value, which could cause the task to become runnable.

You can wait for events to be set using **ctl_events_wait**. You can wait for any of the events in an event set to be set (**CTL_EVENT_WAIT_ANY_EVENTS**) or all of the events to be set (**CTL_EVENT_WAIT_ALL_EVENTS**). You can also specify that when events have been set and have been matched that they should be automatically reset (**CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** and **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR**). You can associate a timeout with a wait for an event set to stop your application blocking indefinitely.

```
ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                &e1, 1<<15,
                CTL_TIMEOUT_NONE, 0);
```

This example waits for bit 15 of the event set pointed to by **e1** to become set.

```
if (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                    &e1, 1<<15,
                    CTL_TIMEOUT_DELAY, 1000) == 0)
  {
    // ...timeout occurred
```

```
  }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

You can use **ctl_events_pulse** to set and immediately clear events. A typical use for this would be to wake up multiple threads and reset the events atomically.

Synchronizing with an ISR

The following example illustrates synchronizing a task with a function called from an ISR.

```
CTL_EVENT_SET_t e1;
CTL_TASK_s t1;

void ISRfn()
{
  // ...do work, and then...
  ctl_events_set_clear(&e1, 1<<0, 0);
}

void task1(void *p)
{
  for (;;)
    {
      ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                      &e1, 1<<0,
                      CTL_TIMEOUT_NONE, 0);
      //
      // ...do whatever needs to be done...
      //
      ctl_events_set_clear(&e1, 0, 1<<0);
    }
}
```

Synchronizing with more than one ISR

The following example illustrates synchronizing a task with functions called from two interrupt service routines.

```
CTL_EVENT_SET_t e1;
CTL_TASK_s t1;

void ISRfn1(void)
{
  // do work, and then...
  ctl_events_set_clear(&e1, 1<<0, 0);
}

void ISRfn2(void)
{
  // do work, and then...
  ctl_events_set_clear(&e1, 1<<1, 0);
}
```

```
void task1(void *p)
{
  for (;;)
    {
      unsigned e;
      e = ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR,
                          &e1, (1<<0) | (1<<1),
                          CTL_TIMEOUT_NONE, 0);
      if (e & (1<<0))
        {
          // ISRfn1 completed
        }
      else if (e & (1<<1))
        {
          // ISRfn2 completed
        }
      else
        {
          // error
        }
    }
}
```

Resource serialization with an event set

The following example illustrates resource serialization of two tasks.

```
CTL_EVENT_SET_t e1;

void task1(void)
{
  for (;;)
    {
      // Acquire resource.
      ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR,
                      &e1, 1<<0,
                      CTL_TIMEOUT_NONE, 0);
      // Resource is now been acquired.

      // Now we have exclusive access to the resource.
      do_some_important_work_with_resource();

      // Release acquired resource.
      ctl_events_set_clear(&e1, 1<<0, 0);
      // Resource is now released.
    }
}

void task2(void)
{
  for (;;)
    {
      // Acquire resource.
      ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR,
                      &e1, 1<<0,
                      CTL_TIMEOUT_NONE, 0);
      // Resource is now acquired.

      // Now we have exclusive access to the resource.
```

```
      do_some_important_work_with_resource();

      // Release acquired resource.
      ctl_events_set_clear(&e1, 1<<0, 0);
      // Resource is now been released.
    }
}

void main(void)
{
  // Initialize event set.
  ctl_events_init(&e1, 1<<0);
  // Create tasks and let them run.
}
```

Note that **e1** is initialized with the event set; without this, neither task would acquire the resource.

Semaphores

CTL provides *semaphores* to use for synchronization and resource allocation.

A semaphore is a counter which tasks can wait for to be non-zero. When a semaphore is non-zero and a task waits on it, the semaphore value is decremented and the task continues executing. When a semaphore is zero and a task waits on it, the task will be suspended until the semaphore is signaled. When a semaphore is signaled and no tasks are waiting for it, the semaphore's value is incremented. When a semaphore is signaled and tasks are waiting, one of the tasks is made runnable.

You allocate a semaphore by declaring it as a C variable. For example:

```
CTL_SEMAPHORE_t s1;
```

A **CTL_SEMAPHORE_t** is a synonym for an **unsigned** type, so the maximum value of the counter is dependent upon the word size of the processor (16 or 32 bits).

A semaphore must be initialized done before any tasks can use it. To initialize a semaphore, use **ctl_semaphore_init**:

```
ctl_semaphore_init(&s1, 1);
```

To signal a semaphore, use **ctl_semaphore_signal**:

```
ctl_semaphore_signal(&s1);
```

The highest-priority task waiting on the semaphore pointed at by **s1** will be made runnable by this call. If no tasks are waiting on the semaphore, the semaphore's value is incremented.

To wait for a semaphore with an optional timeout, use **ctl_semaphore_wait**:

```
ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
```

This example will block the task if the semaphore is zero, otherwise it will decrement the semaphore and execution will continue.

```
if (ctl_semaphore_wait(&s1, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000) == 0)
  {
    // ...timeout occurred
  }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

Task synchronization in an interrupt service routine

The following example illustrates synchronizing a task with a function called from an interrupt service routine.

```
CTL_SEMAPHORE_t s1;

void ISRfn()
{
  // Detected something, signal the waiting task.
  ctl_semaphore_signal(&s1);
}

void task1(void *p)
{
  for (;;)
    {
      // Wait for ISR to signal that an event happened.
      ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
      // Deal with the event.
    }
}
```

Resource serialization with semaphore

The following example illustrates resource serialization of two tasks:

```
CTL_SEMAPHORE_t s1 = 1;

void task1(void)
{
  for (;;)
    {
      // Wait for resource.
      ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
      // Resource has now been acquired, do something with it.

      // Now we have exclusive access to the resource.
      do_some_important_work_with_resource();

      // And now release it...
      ctl_semaphore_signal(&s1);
      // Resource is now released.
    }
}

void task2(void)
{
  for (;;)
    {
      ctl_semaphore_wait(&s1, CTL_TIMEOUT_NONE, 0);
      // Resource has now been acquired, do something with it.

      // Now we have exclusive access to the resource.
      do_some_important_work_with_resource();

      // And now release it...
      ctl_semaphore_signal(&s1);
      // Resource has now been released.
    }
}

int main(void)
```

```
{
  // Initialize semaphore.
  ctl_semaphore_init(&s1, 1);
}
```

Note that **s1** is initialized to one; without this, neither task would acquire the resource.

Mutexes

A mutex is a structure that can be used to serialize resource access. Tasks can lock and unlock mutexes. Each mutex has a *lock count* that enables a task to recursively lock the mutex. Tasks must ensure that the number of unlocks matches the number of locks. When a mutex has already been locked by another task, a task that wants to lock it must wait until the mutex becomes unlocked. The task that locks a mutex is assigned a higher priority than any other tasks waiting to lock that mutex; this avoids what is often called *priority inversion*, which can prevent some tasks from ever getting access to a required resource. Mutexes cannot be used by interrupt service routines.

You allocate a mutex by declaring it as a C variable. For example:

```
CTL_MUTEX_t mutex;
```

A mutex must be intitialized before any task can use it. To initialize a mutex, use **ctl_mutex_init** as in this example:

```
ctl_mutex_init(&mutex);
```

You can lock a mutex with an optional timeout by using **ctl_mutex_lock**:

```
ctl_mutex_lock(&mutex, CTL_TIMEOUT_NONE, 0);
```

You can unlock a mutex by using **ctl_mutex_unlock**:

```
ctl_mutex_unlock(&mutex);
```

Note: Only the locking task must unlock a successfully-locked mutex.

Resource serialization with mutex

The following example illustrates resource serialization of two tasks.

```
CTL_MUTEX_t mutex;

void fn1(void)
{
  ctl_lock_mutex(&mutex, CTL_TIMEOUT_NONE, 0);
  ?
  ctl_unlock_mutex(&mutex);
}

void fn2(void)
{
  ctl_lock_mutex(&mutex, CTL_TIMEOUT_NONE, 0);
  ?
  fn1();
  ?
  ctl_unlock_mutex(&mutex);
```

```
}

void task1(void)
{
  for (;;)
    {
      fn2()
    }
}

void task2(void)
{
  for (;;)
    {
      fn1();
    }
}

int main(void)
{
  ?
  ctl_mutex_init(&mutex);
  ?
}
```

Note that **task1** locks the mutex twice by calling **fn2** which then calls **fn1**.

Message queues

Message queues provide buffers between tasks and interrupt service routines.

A *message queue* is a structure that enables tasks to post and receive messages. A *message* is a generic (void) pointer and, as such, can be used to send data that will fit into a pointer type (two or four bytes, depending upon the processor's word size) or to pass a pointer to a block of memory. The message queue uses a buffer to enable a number of posts to be completed without receives occurring. The buffer keeps the posted messages in FIFO order, so the oldest message is received first. When the buffer isn't full, a post will put the message at the back of the queue and the calling task continues execution. When the buffer is full, a post will block the calling task until there is room for the message. When the buffer isn't empty, a receive will return the message from the front of the queue and continue execution of the calling task. When the buffer is empty, a receive will block the calling task until a message is posted.

Initializing a message queue

You allocate a message queue by declaring it as a C variable:

```
CTL_MESSAGE_QUEUE_t m1;
```

A message queue is initialized using **ctl_message_queue_init**:

```
void *queue[20];
?

ctl_message_queue_init(&m1, queue, 20);
```

This example uses a 20-element array for the message queue. The array is a **void *** so pointers to memory or (cast) integers can be communicated via a message queue.

Posting to a message queue

You can post a message to a message queue with an optional timeout by using the **ctl_message_queue_post** function.

```
ctl_message_queue_post(&m1, (void *)45, CTL_TIMEOUT_NONE, 0);
```

This example posts the integer 45 to the message queue.

You can post multiple messages to a message queue with an optional timeout using **ctl_message_queue_post_multi**:

```
if (ctl_message_queue_post_multi(&m1, 4, messages, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000) !
= 4)
  {
    // timeout occurred
  }
```

This example tests the return result to see if the timeout occurred.

If you want to post a message and you cannot afford to block (e.g. inside an interrupt service routine), you can use **ctl_message_queue_post_nb** (or **ctl_message_queue_post_multi_nb** if you want to post multiple messages):

```
if (ctl_message_queue_post_nb(&m1, (void *)45) == 0)
  {
    // queue is full
  }
```

This example tests the return result to see if the post failed.

Receiving from a message queue

You can use **ctl_message_queue_receive** to receive a message with an optional timeout:

```
void *msg;
ctl_message_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
```

This example receives the oldest message in the message queue.

Use **ctl_message_queue_receive_multi** to receive multiple messages from a message queue with an optional timeout:

```
if (ctl_message_queue_multi_receive(&m1, 4, msgs, CTL_TIMEOUT_DELAY, 1000) != 4)
  {
    // timeout occurred
  }
```

This example tests the return result to see if the timeout occurred.

If you want to receive a message and you don't want to block (e.g., when executing interrupt service routine), you can use **ctl_message_queue_receive_nb** (or **ctl_message_queue_receive_multi_nb** to receive multiple messages).

```
if (ctl_message_queue_receive_nb(&m1, &msg) == 0)
  {
    // queue is empty
  }
```

Producer-consumer example

The following example uses a message queue to implement the producer-consumer problem.

```
CTL_MESSAGE_QUEUE_t m1;
void *queue[20];

void task1(void)
{
  ?
```

```
  ctl_message_queue_post(&m1, (void *)i, CTL_TIMEOUT_NONE, 0);
  ?
}

void task2(void)
{
  void *msg;
  ?
  ctl_message_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
  ?
}

int main(void)
{
  ?
  ctl_message_queue_init(&m1, queue, 20);
  ?
}
```

Advanced use

You can associate event flags with a message queue that are set (and similarly cleared) when the message queue is not full and not empty using the function **ctl_message_queue_setup_events**.

For example, you can use this to wait for messages to arrive from multiple message (or byte) queues:

```
CTL_MESSAGE_QUEUE_t m1, m2;
CTL_EVENT_SET_t e;
ctl_message_queue_setup_events(&m1, &e, 1<<0, 1<<1));
ctl_message_queue_setup_events(&m2, &e, 1<<2, 1<<3));
?

switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                        &e, (1<<0) | (1<<2),
                        CTL_TIMEOUT_NONE, 0))
  {
  case 1<<0:
    ctl_message_queue_receive(&m1, …
    break;
  case 1<<2:
    ctl_message_queue_receive(&m2, …
    break;
  }
```

This example sets up and waits for the not-empty event of message queue **m1** and the not-empty event of message queue **m2**. When the wait completes, it reads from the appropriate message queue. Note that you *should not* use a 'with auto clear' event wait type when waiting for events associated with a message queue.

You can use **ctl_message_queue_num_used** to test how many messages are in a message queue and **ctl_message_queue_num_free** to learn how many free messages are in a message queue. With these functions you can poll the message queue:

```
while (ctl_message_queue_num_free(&m1) < 10)
  ctl_task_timeout_wait(ctl_get_current_time() + 1000);
ctl_message_queue_post_multi(&m1, 10, …
```

This example waits for 10 elements to be free before it posts 10 elements.

Byte queues

Byte queues provide byte-based buffers between tasks and interrupt service routines.

A *byte queue* is a structure that enables tasks to post and receive data bytes. The byte queue has a buffer, which enables a number of posts to be completed without receives occurring. The buffer keeps the posted bytes in FIFO order, so the oldest byte is received first. When the buffer isn't full, a post will put the byte at the back of the queue and the calling task continues execution. When the buffer is full, a post will block the calling task until there is room for the byte. When the buffer isn't empty, a receive will return the byte from the front of the queue and continue execution of the calling task. When the buffer is empty, a receive will block the calling task until a byte is posted.

Initializing a byte queue

You allocate a byte queue by declaring it as a C variable:

```
CTL_BYTE_QUEUE_t m1;
```

A byte queue is initialized using **ctl_byte_queue_init**:

```
unsigned char queue[20];
?
ctl_byte_queue_init(&m1, queue, 20);
```

This example uses an 20-element array for the byte queue.

Posting to a byte queue

You can post a byte to a byte queue with an optional timeout using **ctl_byte_queue_post**:

```
ctl_byte_queue_post(&m1, 45, CTL_TIMEOUT_NONE, 0);
```

This example posts the byte 45 to the byte queue.

You can post multiple bytes to a byte queue with an optional timeout using **ctl_byte_queue_post_multi**:

```
if (ctl_byte_queue_post(&m1, 4, bytes, CTL_TIMEOUT_ABSOLUTE, ctl_get_current_time()+1000) !
= 4)
  {
    // timeout occurred
  }
```

This example uses a timeout and tests the return result to see if the timeout occurred.

If you want to post a byte and you don't want to block access (e.g., from an interrupt service routine), you can use **ctl_byte_queue_post_nb** (or **ctl_byte_queue_post_multi_nb** to post multiple bytes).

```
if (ctl_byte_queue_post_nb(&m1, 45) == 0)
  {
    // queue is full
  }
```

This example tests the return result to see if the post failed.

Receiving from a byte queue

You can receive a byte with an optional timeout by using **ctl_byte_queue_receive**:

```
unsigned char msg;
ctl_byte_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
```

This example receives the oldest byte in the byte queue.

You can receive multiple bytes from a byte queue with an optional timeout using **ctl_byte_queue_receive_multi**:

```
if (ctl_byte_queue_receive_multi(&m1, 4, bytes, CTL_TIMEOUT_DELAY, 1000) != 4)
  {
    // timeout occurred
  }
```

This example tests the return result to see if the timeout occurred.

If you want to receive a byte and you don't want to block (e.g., from an interrupt service routine), you can use **ctl_byte_queue_receive_nb** (or **ctl_byte_queue_receive_multi_nb** to receive multiple bytes).

```
if (ctl_byte_queue_receive_nb(&m1, &msg) == 0)
  {
    // queue is empty
  }
```

Producer-consumer example

The following example uses a byte queue to implement the producer-consumer problem.

```
CTL_BYTE_QUEUE_t m1;
void *queue[20];

void task1(void)
{
  ?
  ctl_byte_queue_post(&m1, (void *)i, CTL_TIMEOUT_NONE, 0);
  ?
}

void task2(void)
{
```

```
  void *msg;
  ?
  ctl_byte_queue_receive(&m1, &msg, CTL_TIMEOUT_NONE, 0);
  ?
}

int main(void)
{
  ?
  ctl_byte_queue_init(&m1, queue, 20);
  ?
}
```

Advanced Use

You can associate event flags with a byte queue that are set (and similarly cleared) when the byte queue is not full and not empty using the function **ctl_byte_queue_setup_events**.

For example, you can use this to wait for messages to arrive from multiple byte (or message) queues.

```
CTL_BYTE_QUEUE_t m1, m2;
CTL_EVENT_SET_t e;
ctl_byte_queue_setup_events(&m1, &e, 1<<0, 1<<1);
ctl_byte_queue_setup_events(&m2, &e, 1<<2, 1<<3);
?
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                        &e, (1<<0) | (1<<2),
                        CTL_TIMEOUT_NONE, 0))
  {
    case 1<<0:
      ctl_byte_queue_receive(&m1, …
      break;
    case 1<<2:
      ctl_byte_queue_receive(&m2, …
      break;
  }
```

This example sets up and waits for the not-empty event of byte queue **m1** and the not-empty event of byte queue **m2**. When the wait completes, it reads from the appropriate byte queue. Note that you *must not* use a 'with auto clear' event wait type when waiting on events associated with a byte queue.

You can use **ctl_byte_queue_num_used** to test how many bytes are in a byte queue and **ctl_byte_queue_num_free** to learn how many free bytes are in a byte queue. With these functions, you can poll the byte queue:

```
while (ctl_byte_queue_num_free(&m1) < 10)
  ctl_task_timeout_wait(ctl_get_current_time()+1000);
ctl_byte_queue_post_multi(&m1, 10, …
```

This example waits for 10 elements to be free before it posts 10 elements.

Global interrupts control

CTL provides functions that enable and disable the global interrupt enables of the processor. CTL uses this mechanism when accessing the task list. It can also be used to provide a fast, mutual-exclusion facility for time-critical uses.

You can disable interrupts by using **ctl_global_interrupts_disable** and you can enable interrupts by using **ctl_global_interrupts_enable**.

```
int en = ctl_global_interrupts_disable(); // disable
?
if (en)
  ctl_global_interrupts_enable(); // set to previous state
```

You can call a tasking library function that causes a task switch with global interrupts disabled. The tasking library will ensure that, when the next task is scheduled, global interrupts are enabled.

Timer support

The current time is held as a 32-bit value in the variable **ctl_current_time**. This variable is incremented by the number held in **ctl_time_increment** each time an ISR calls **ctl_increment_tick_from_isr**.

```
void timerISR{void)
{
  ctl_increment_tick_from_isr();
  // Your timer code goes here.
}

int main(void)
{
  ctl_time_increment = 10;
  // User must set up timerISR to be called every 100 ms.
  ?
```

By convention, the timer implements a millisecond counter, but you can set the timer's interrupt-and-increment rate to a value that is appropriate for your application.

You can atomically read **ctl_current_time** by using the **ctl_get_current_time** function on systems whose word size is not 32 bits.

You can use **ctl_timeout_wait** to suspend execution of a task for a fixed period. Note: **ctl_timeout_wait** takes as its parameter the time to resume execution, not the duration: always call this function with **ctl_get_current_time()**+*duration*.

```
ctl_timeout_wait(ctl_get_current_time()+100);
```

This example suspends execution of the calling task for 100 ticks of the **ctl_current_time** variable.

The counter is implemented as a 32-bit number, so you can delay for a maximum of a 31-bit number.

```
ctl_timeout_wait(ctl_get_current_time() + 0x7fffffff);
```

This example suspends execution of the calling task for the maximum possible time.

Interrupt service routines

Interrupt service routines (ISR) can communicate with CTL tasks using a subset of the CTL programming interface. An ISR should not call any of the CTL functions that can block; if your ISR calls a blocking function, **ctl_handle_error** will be called. To detect whether a task or an ISR has called a function, CTL uses the global variable **ctl_interrupt_count**. Interrupt service routines must increment this variable on entry and decrement it on exit. Any CTL functions called by an ISR that require a task reschedule will set the variable **ctl_reschedule_on_last_isr_exit**.

On exit from an interrupt service routine, **ctl_interrupt_count** is decremented to zero and, if **ctl_reschedule_on_last_isr_exit** is set (after resetting **ctl_reschedule_on_last_isr_exit**), a CTL reschedule operation occurs. The support for writing ISRs differs, depending on the target. In general, on entry to an ISR the following is required:

```
// ...preserve register state here
++ctl_interrupt_count;
```

…and, on exit from an ISR:

```
ctl_interrupt_count--;
if (ctl_interrupt_count == 0 && ctl_reschedule_on_last_isr_exit)
  {
    ctl_reschedule_on_last_isr_exit = 0;
    // reschedule
  }
else
  {
    // ...restore register state here
  }
```

Memory areas

Memory areas provide your application with dynamic allocation of fixed-sized memory blocks. Memory areas should be used in preference to the standard C library **malloc** and **free** functions if the calling task cannot block or if memory allocation is done by an ISR.

You allocate a memory area by declaring it as a C variable:

```
CTL_MEMORY_AREA_t m1;
```

Before using a memory area, you must initialize it using **ctl_memory_area_init**:

```
unsigned mem[20];
?
ctl_memory_area_init(&m1, mem, 2, 10);
```

This example uses a 20-element array for the memory area's working storage. The array is split into 10 blocks, each block being two words in size.

To allocate a block from a memory area, use **ctl_memory_area_allocate**. If the memory block cannot be allocated, zero is returned.

```
unsigned *block = ctl_memory_area_allocate(&m1);
if (block)
  {
    // Block has been allocated.
  }
else
  {
    // No block has been allocated.
  }
```

When you have finished with a memory block, use **ctl_memory_area_free** to return it to the memory area *from which it was allocated* so it can be reused:

```
ctl_memory_area_free(&m1, block);
```

You can associate an event flag with the *block available* state of a memory queue to wait for a memory block to become available:

```
CTL_MEMORY_AREA_t m0, m1, m2;
CTL_EVENT_SET_t e;
?
ctl_memory_area_setup_events(&m0, &e, 1<<0);
ctl_memory_area_setup_events(&m1, &e, 1<<1);
ctl_memory_area_setup_events(&m2, &e, 1<<2);
?
switch (ctl_events_wait(CTL_EVENT_WAIT_ANY_EVENTS,
                        &e, (1<<0)|(1<<1)|(1<<2),
                        0, 0))
  {
  case 1<<0:
```

```
    x = ctl_memory_area_allocate(&m0, …
    break;
  case 1<<1:
    x = ctl_memory_area_allocate(&m1, …
    break;
  case 1<<2:
    x = ctl_memory_area_allocate(&m2, …
    break;
  }
```

This example sets up and waits for the block-available events of memory areas **m0**, **m1**, and **m2**. When the wait completes, it attempts to allocate memory from the appropriate memory area. Note that you should *not* use a *with-auto-clear* event wait type when waiting on events associated with a memory area.

Task scheduling example

An example task list could be:

- task1, priority 2, waiting
- task2, priority 1, runnable
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

task2 waits, so task3 is selected to execute:

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

An interrupt occurs that makes task1 runnable, which is higher priority than task3 so task1 executes:

- task1, priority 2, executing
- task2, priority 1, waiting
- task3, priority 1, runnable
- task4, priority 1, runnable
- task5, priority 0, runnable

task1 waits, causing task3 to execute:

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, executing
- task4, priority 1, runnable
- task5, priority 0, runnable

An interrupt occurs and task3 has used its timeslice period, so task4 is selected to execute:

- task1, priority 2, waiting
- task2, priority 1, waiting
- task3, priority 1, runnable
- task4, priority 1, executing
- task5, priority 0, runnable

An interrupt occurs and makes task2 runnable, but task4 hasn't used its timeslice period, so it is left to execute:

- task1, priority 2, waiting
- task2, priority 1, runnable

- task3, priority 1, runnable
- task4, priority 1, executing
- task5, priority 0, runnable

A interrupt occurs and task4 has used its timeslice period:

- task1, priority 2, waiting
- task2, priority 1, executing
- task3, priority 1, runnable
- task4, priority 1, runnable
- task5, priority 0, runnable

ARM implementation details

Processor modes

The ARM implementation of CTL uses System and IRQ processor modes. Other processor modes are not used and, therefore, are available for use by the application. In normal execution, tasks run in System mode with IRQ interrupts enabled.

When CTL requires exclusive access to variables, for example when traversing the task list, IRQ interrupts are disabled. FIQ interrupts are always enabled by CTL. Co-operative context switching is done by changing to IRQ mode (with IRQ interrupts disabled) and, consequently, uses the IRQ mode stack. Preemptive context switching is done from an IRQ handler, which by definition is running in IRQ mode.

Register save order

When a task is not executing, the ARM register context is saved on the task's stack in the following order:

- PSR
- R15
- R14
- R12
- R3–R0
- R11–R4

…with the **stack_pointer** member of the task structure pointing to the **R4** entry which requires 16 words of memory.

For devices that have a VFP with 16 double precision registers the floating point registers are also saved as follows:

- FPSR
- D7–D0
- PSR
- R15
- R14
- R12
- R3–R0
- R11–R4
- D15–D8

…with the **stack_pointer** member of the task structure pointing to the **D8** entry (with 1 added to indicate that the floating point registers have been saved) which requires 49 words of memory.

For devices that have a VFP with 32 double precision registers the floating point registers are also saved as follows:

- FPSR
- D7–D0
- PSR
- R15
- R14
- R12
- R3–R0
- R11–R4
- D31–D8

…with the **stack_pointer** member of the task structure pointing to the **D8** entry (with 3 added to indicate that the 32 double precision floating point registers have been saved) which requires 81 words of memory.

IRQ handler

On entry to an IRQ handler, the **ctl_interrupt_count** variable should be incremented. On exit from an IRQ handler, the **ctl_exit_isr** routine should be called with a parameter in **R0** specifying which registers have been saved by the IRQ handler.

If **ctl_exit_isr**(0) is called, the registers should be saved, as in the following example:

```
irq_handler:
  \vdots
  // store the APCS registers
  sub   lr, lr, #4
  stmfd sp!, {r0-r3, r12, lr}

  // ctl_interrupt_count++
  ldr   r2, =ctl_interrupt_count
  ldrb  r3, [r2]
  add   r3, r3, #1
  strb  r3, [r2]
  \vdots
  // handle interrupt, possibly re-enabling interrupts
  \vdots
  // ctl_exit_isr(0)
  mov   r0, #0
  ldr   r1, =ctl_exit_isr
  bx    r1
```

If `ctl_exit_isr(!0)` is called, the registers should be saved, as in the following example:

```
irq_handler:
  \vdots
  // store all the registers
  stmfd sp!, {r0-r12, lr}
  mrs   r0, spsr
```

```
  stmfd sp!, {r0}

  // ctl_interrupt_count++
  ldr   r2, =ctl_interrupt_count
  ldrb  r3, [r2]
  add   r3, r3, #1
  strb  r3, [r2]
  \vdots
  // handle interrupt, possibly re-enabling interrupts
  \vdots
  // ctl_exit_isr(!0)
  mov   r0, sp
  ldr   r1, =ctl_exit_isr
  bx    r1
```

The first form (`ctl_exit_isr(0)`) is recommended because it uses less stack space and takes fewer machine cycles. The second form (`ctl_exit_isr(!0)`) is provided for backwards compatibility with earlier releases of CTL.

Cortex-M implementation details

Processor modes and interrupts

All CTL threads run in privileged thread mode.

When CTL requires exclusive access to variables—for example when traversing the task list—interrupts are disabled using calls to **ctl_global_interrupts_disable** and **ctl_global_interrupts_enable**.

The default implementation of global interrupt disable and enable for the Cortex-M3/M4 will set or clear the top bit of the **BASEPRI** register. This enables interrupts that have the highest half of the available priority numbers (lowest priority levels) to use CTL API calls. The lowest half of the available priority numbers (highest priority levels) cannot use CTL API calls but will not be disabled during CTL API calls. If, for example, the device has four priority bits, then priority numbers 8 through 15 can be used for interrupts that make CTL API calls and priorities 0 through 7 can be used for interrupts that cannot make CTL API calls.

The default implementation of global interrupts and enable for the Cortex-M0/M1 will set or clear the **PRIMASK** register.

Exceptions

Context switching is implemented using the PendSV exception handler which should be set to run at the lowest exception priority i.e. the highest exception priority number. The SVCall exception handler is not used.

Stacks

CTL threads use the Cortex-M process stack pointer (psp) and exceptions use the main stack pointer (msp). This means that you don't have to allocate space for exceptions in the thread stacks.

Register save order

When a task is not executing, the register context is saved on the task's stack in the following order:

- PSR
- R15
- R14
- R12
- R3–R0
- R11–R4

…with the **stack_pointer** member of the task structure pointing to the **R4** entry which requires 16 words of memory.

For Cortex-M4F the 32 single precision floating point registers are also saved when they have been used by a task. This changes the register context saved on the task's stack to:

- FPSR
- S15–S0
- PSR
- R15
- R14
- R12
- R3–R0
- R11–R4
- S31–S16

…with the **stack_pointer** member of the task structure pointing to the **S16** entry (with 1 added to indicate that the floating point registers have been saved) which requires 49 words of memory.

Interrupt handlers

A Cortex-M interrupt handler that uses CTL services should use the following template code for entry and exit:

```
void SysTick_ISR(void)
{
  ctl_enter_isr();
  ?
  // handle interrupt here
  ?
  ctl_exit_isr();
}
```

The call to **ctl_enter_isr** will increment the **ctl_interrupt_count** and the call to **ctl_exit_isr** will decrement the **ctl_interrupt_count** and, if required, trigger the PendSV exception.

Note that you must ensure that an interrupt handler that uses CTL services cannot interrupt an interrupt handler that does not use CTL services. You can do this by setting the interrupt priority of interrupt handlers that do not use CTL services to be higher than those that do.

Interrupt handler support code (including **ctl_enter_isr** and **ctl_exit_isr**) is not part of CTL, but there are common definitions that are available in `ctl_api.h` and will be defined by code or libraries supplied by the CPU support package you are using.

System timer

The CPU support package you are using will use the Cortex-M SysTick timer to implement the CTL timer. Typically, the timer will be programmed to interrupt at 10 millisecond intervals and increment the CTL timer by 10 to create the millisecond CTL timer.

Complete API reference

This section contains a complete reference to the CrossWorks Tasking Library (CTL) API.

<ctl.h>

API Summary

Bytes queues	
ctl_byte_queue_init	Initialize a byte queue
ctl_byte_queue_num_free	Return number of free bytes in a byte queue
ctl_byte_queue_num_used	Return number of used bytes in a byte queue
ctl_byte_queue_post	Post byte to a byte queue with optional timeout
ctl_byte_queue_post_multi	Post bytes to a byte queue with optional timeout
ctl_byte_queue_post_multi_nb	Post bytes to a byte queue without blocking
ctl_byte_queue_post_multi_uc	Post bytes to a byte queue
ctl_byte_queue_post_nb	Post byte to a byte queue without blocking
ctl_byte_queue_post_uc	Post byte to a byte queue
ctl_byte_queue_receive	Receive a byte from a byte queue with optional timeout
ctl_byte_queue_receive_multi	Receive multiple bytes from a byte queue with optional timeout
ctl_byte_queue_receive_multi_nb	Receive multiple bytes from a byte queue without blocking
ctl_byte_queue_receive_multi_uc	Receive multiple bytes from a byte queue, unconditional
ctl_byte_queue_receive_nb	Receive a byte from a byte queue without blocking
ctl_byte_queue_receive_uc	Receive a byte from a byte queue
ctl_byte_queue_setup_events	Associate events with the not-full and not-empty state of a byte queue
Types	
CTL_BYTE_QUEUE_t	Byte queue struct definition
CTL_ERROR_CODE_t	Error cause
CTL_EVENT_SET_t	Event set definition
CTL_EVENT_WAIT_TYPE_t	Event set wait types
CTL_MEMORY_AREA_t	Memory area struct definition
CTL_MESSAGE_QUEUE_t	Message queue struct definition
CTL_MUTEX_t	Mutex struct definition
CTL_SEMAPHORE_t	Semaphore definition
CTL_STATE_t	Task states

CTL_TASK_t	Task struct definition
CTL_TIMEOUT_t	Type of wait
CTL_TIME_t	Time definition
Mutexes	
ctl_mutex_init	Initialize a mutex
ctl_mutex_lock	Lock a mutex with optional timeout
ctl_mutex_lock_nb	Lock a mutex without blocking
ctl_mutex_lock_uc	Lock a mutex, unconditional
ctl_mutex_unlock	Unlock a mutex
Message queues	
ctl_message_queue_init	Initialize a message queue
ctl_message_queue_num_free	Return number of free elements in a message queue
ctl_message_queue_num_used	Return number of used elements in a message queue
ctl_message_queue_post	Post message to a message queue with optional timeout
ctl_message_queue_post_multi	Post messages to a message queue with optional timeout
ctl_message_queue_post_multi_nb	Post messages to a message queue without blocking
ctl_message_queue_post_multi_uc	Post messages to a message queue
ctl_message_queue_post_nb	Post message to a message queue without blocking
ctl_message_queue_post_uc	Post message to a message queue
ctl_message_queue_receive	Receive message from a message queue with optional timeout
ctl_message_queue_receive_multi	Receive messages from a message queue with optional timeout
ctl_message_queue_receive_multi_nb	Receive messages from a message queue without blocking
ctl_message_queue_receive_multi_uc	Receive messages from a message queue
ctl_message_queue_receive_nb	Receive message from a message queue without blocking
ctl_message_queue_receive_uc	Receive message from a message queue
ctl_message_queue_setup_events	Associate events with the not-full and not-empty state of a message queue
Tasks	
ctl_task_die	Terminate the executing task
ctl_task_init	Create the initial task
ctl_task_remove	Remove a task from the task list

ctl_task_reschedule	Cause a reschedule
ctl_task_restore	Put back a task on to the task list
ctl_task_run	Start a task
ctl_task_set_priority	Set the priority of a task
Memory areas	
ctl_memory_area_allocate	Allocate a block from a memory area
ctl_memory_area_free	Free a memory area block
ctl_memory_area_init	Initialize a memory area
ctl_memory_area_setup_events	Set memory area events
System state variables	
ctl_current_time	The current time in ticks
ctl_interrupt_count	Nested interrupt count
ctl_last_schedule_time	The time (in ticks) of the last task schedule
ctl_reschedule_on_last_isr_exit	Reschedule is required on last ISR exit
ctl_task_executing	The task that is currently executing
ctl_task_list	List of tasks sorted by priority
ctl_task_switch_callout	A function pointer called on a task switch
ctl_time_increment	Current time tick increment
ctl_timeslice_period	Time slice period in ticks
Event sets	
ctl_events_init	Initialize an event set
ctl_events_pulse	Pulse events in an event set
ctl_events_set_clear	Set and clear events in an event set
ctl_events_wait	Wait for events in an event set with optional timeout
ctl_events_wait_nb	Wait for events in an event set without blocking
ctl_events_wait_uc	Wait for events in an event set
Error handling	
ctl_handle_error	Handle a CTL error condition
Semaphores	
ctl_semaphore_init	Initialize a semaphore
ctl_semaphore_signal	Signal a semaphore
ctl_semaphore_wait	Wait for a semaphore with optional timeout
ctl_semaphore_wait_nb	Wait for a semaphore without blocking
ctl_semaphore_wait_uc	Wait for a semaphore
Timer	

ctl_get_current_time	Atomically return the current time
ctl_get_sleep_delay	Return the sleep delay
ctl_increment_tick_from_isr	Increment tick timer
ctl_timeout_wait	Wait until timeout has occurred
Interrupts	
ctl_global_interrupts_disable	Disable global interrupts
ctl_global_interrupts_enable	Enable global interrupts
ctl_global_interrupts_set	Enable/disable interrupts

CTL_BYTE_QUEUE_t

Synopsis

```
typedef struct {
  unsigned char *q;
  unsigned s;
  unsigned front;
  unsigned n;
  CTL_EVENT_SET_t *e;
  CTL_EVENT_SET_t notempty;
  CTL_EVENT_SET_t notfull;
} CTL_BYTE_QUEUE_t;
```

Description

CTL_BYTE_QUEUE_t defines the byte queue structure. The byte queue structure contains:

Member	Description
q	pointer to the array of bytes
s	size of the array of bytes
front	the next byte to leave the byte queue
n	the number of elements in the byte queue
e	the event set to use for the not empty and not full events
notempty	the event number for a not empty event
notfull	the event number for a not full event

CTL_ERROR_CODE_t

Synopsis

```
typedef enum {
  CTL_ERROR_NO_TASKS_TO_RUN,
  CTL_UNSUPPORTED_CALL_FROM_ISR,
  CTL_MUTEX_UNLOCK_CALL_ERROR,
  CTL_UNSPECIFIED_ERROR,
  CTL_STACK_OVERFLOW
} CTL_ERROR_CODE_t;
```

Description

CTL_ERROR_CODE_t defines the set of errors that are detected by the CrossWorks tasking library; the errors are reported by a call to **ctl_handle_error**.

Constant	Description
CTL_ERROR_NO_TASKS_TO_RUN	A reschedule has occurred but there are no tasks which are runnable.
CTL_UNSUPPORTED_CALL_FROM_ISR	An interrupt service routine has called a tasking library function that could block or is otherwise unsupported when called from inside an interrupt service routine.
CTL_MUTEX_UNLOCK_CALL_ERROR	A task called **ctl_mutex_unlock** passing a mutex which it has not locked, or which a different task holds a lock on. Only the task that successfully acquired a lock on a mutex can unlock that mutex.
CTL_STACK_OVERFLOW	Inusufficient space to save the CPU register state to the stack of ctL_currently_executing_task
CTL_UNSPECIFIED_ERROR	An unspecified error has occurred.

CTL_EVENT_SET_t

Synopsis

```
typedef unsigned CTL_EVENT_SET_t;
```

Description

CTL_EVENT_SET_t defines an event set. Event sets are word sized 16 or 32 depending on the machine.

CTL_EVENT_WAIT_TYPE_t

Synopsis

```
typedef enum {
  CTL_EVENT_WAIT_ANY_EVENTS,
  CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR,
  CTL_EVENT_WAIT_ALL_EVENTS,
  CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR
} CTL_EVENT_WAIT_TYPE_t;
```

Description

CTL_EVENT_WAIT_TYPE_t defines how to wait for an event set.

Constant	Description
CTL_EVENT_WAIT_ANY_EVENTS	Wait for any of the specified events to be set in the event set.
CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR	Wait for any of the specified events to be set in the event set and reset (clear) them.
CTL_EVENT_WAIT_ALL_EVENTS	Wait for all of the specified events to be set in the event set.
CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR	Wait for all of the specified events to be set in the event set and reset (clear) them.

See Also

ctl_events_wait

CTL_MEMORY_AREA_t

Synopsis

```
typedef struct {
  unsigned *head;
  CTL_EVENT_SET_t *e;
  CTL_EVENT_SET_t blockavailable;
} CTL_MEMORY_AREA_t;
```

Description

CTL_MEMORY_AREA_t defines the memory area structure. The memory area structure contains:

Member	Description
head	the next free memory block
e	the event set containing the blockavailable event
blockavailable	the blockavailable event

CTL_MESSAGE_QUEUE_t

Synopsis

```
typedef struct {
  void ** q;
  unsigned s;
  unsigned front;
  unsigned n;
  CTL_EVENT_SET_t *e;
  CTL_EVENT_SET_t notempty;
  CTL_EVENT_SET_t notfull;
} CTL_MESSAGE_QUEUE_t;
```

Description

CTL_MESSAGE_QUEUE_t defines the message queue structure. The message queue structure contains:

Member	Description
q	pointer to the array of message queue objects
s	size of the array of message queue objects
front	the next element to leave the message queue
n	the number of elements in the message queue
e	the event set to use for the not empty and not full events
notempty	the event number for a not empty event
notfull	the event number for a not full event

CTL_MUTEX_t

Synopsis

```
typedef struct {
  unsigned lock_count;
  CTL_TASK_t *locking_task;
  unsigned locking_task_priority;
} CTL_MUTEX_t;
```

Description

CTL_MUTEX_t defines the mutex structure. The mutex structure contains:

Member	Description
lock_count	number of times the mutex has been locked
locking_task	the task that has locked the mutex
locking_task_priority	the priority of the task at the time it locked the mutex

CTL_SEMAPHORE_t

Synopsis

```
typedef unsigned CTL_SEMAPHORE_t;
```

Description

CTL_SEMAPHORE_t defines the semaphore type. Semaphores are held in one word, 16 or 32 bits depending on the machine.

CTL_STATE_t

Synopsis

```
typedef enum {
  CTL_STATE_RUNNABLE,
  CTL_STATE_TIMER_WAIT,
  CTL_STATE_EVENT_WAIT_ALL,
  CTL_STATE_EVENT_WAIT_ALL_AC,
  CTL_STATE_EVENT_WAIT_ANY,
  CTL_STATE_EVENT_WAIT_ANY_AC,
  CTL_STATE_SEMAPHORE_WAIT,
  CTL_STATE_MESSAGE_QUEUE_POST_WAIT,
  CTL_STATE_MESSAGE_QUEUE_RECEIVE_WAIT,
  CTL_STATE_MUTEX_WAIT,
  CTL_STATE_SUSPENDED
} CTL_STATE_t;
```

Description

CTL_STATE_t defines the states the task can be on.

Constant	Description
CTL_STATE_RUNNABLE	Task can run.
CTL_STATE_TIMER_WAIT	Waiting for a time value.
CTL_STATE_EVENT_WAIT_ALL	Waiting for all events to be set.
CTL_STATE_EVENT_WAIT_ALL_AC	Waiting for all events to be set with auto clear.
CTL_STATE_EVENT_WAIT_ANY	Waiting for any events to be set.
CTL_STATE_EVENT_WAIT_ANY_AC	Waiting for any events to be set with auto clear.
CTL_STATE_SEMAPHORE_WAIT	Task is waiting for a semaphore.
CTL_STATE_MESSAGE_QUEUE_POST_WAIT	Task is waiting to post to a message queue.
CTL_STATE_MESSAGE_QUEUE_RECEIVE_WAIT	Task is waiting to receive from a message queue.
CTL_STATE_MUTEX_WAIT	Task is waiting for a mutex.
CTL_STATE_SUSPENDED	Task cannot run.

CTL_TASK_t

Synopsis

```
typedef struct {
  unsigned *stack_pointer;
  unsigned *thread_local_storage;
  unsigned *stack_start;
  unsigned char priority;
  unsigned char state;
  unsigned char timeout_occured;
  CTL_TASK_s *next;
  CTL_TIME_t timeout;
  void *wait_object;
  CTL_EVENT_SET_t wait_events;
  void *data;
  CTL_TIME_t execution_time;
  const char *name;
} CTL_TASK_t;
```

Description

CTL_TASK_t defines the task structure. The task structure contains:

Member	Description
stack_pointer	the saved register state of the task when it is not scheduled
thread_local_storage	pointer to the thread local storage of this task
priority	the priority of the task
state	the state of task CTL_STATE_RUNNABLE or (CTL_STATE_*_WAIT_* | CTL_STATE_TIMER_WAIT) or CTL_STATE_SUSPENDED
timeout_occured	1 if a wait timed out otherwise 0 - when state is CTL_RUNNABLE
next	next pointer for wait queue
timeout	wait timeout value or time slice value when the task is executing
wait_object	the event set, semaphore, message queue or mutex to wait on
wait_events	the events to wait for
data	task specific data pointer
execution_time	number of ticks the task has executed for
stack_start	the start (lowest address) of the stack
name	task name

CTL_TIMEOUT_t

Synopsis

```
typedef enum {
  CTL_TIMEOUT_NONE,
  CTL_TIMEOUT_INFINITE,
  CTL_TIMEOUT_ABSOLUTE,
  CTL_TIMEOUT_DELAY,
  CTL_TIMEOUT_NOW
} CTL_TIMEOUT_t;
```

Description

CTL_TIMEOUT_t defines the type of timeout for a blocking function call.

Constant	Description
CTL_TIMEOUT_NONE	No timeout — block indefinitely.
CTL_TIMEOUT_INFINITE	Identical to **CTL_TIMEOUT_NONE**.
CTL_TIMEOUT_ABSOLUTE	The timeout is an absolute time.
CTL_TIMEOUT_DELAY	The timeout is relative to the current time.
CTL_TIMEOUT_NOW	The timeout happens immediately — no rescheduling occurs.

CTL_TIME_t

Synopsis

```
typedef unsigned long CTL_TIME_t;
```

Description

CTL_TIME_t defines the base type for times that CTL uses.

ctl_byte_queue_init

Synopsis

```
void ctl_byte_queue_init(CTL_BYTE_QUEUE_t *q,
                         unsigned char *queue,
                         unsigned queue_size);
```

Description

ctl_byte_queue_init is given a pointer to the byte queue to initialize in **q**. The array that will be used to implement the byte queue pointed to by **queue** and its size in **queue_size** are also supplied.

ctl_byte_queue_num_free

Synopsis

```
unsigned ctl_byte_queue_num_free(CTL_BYTE_QUEUE_t *q);
```

Description

ctl_byte_queue_num_free returns the number of free bytes in the byte queue **q**.

ctl_byte_queue_num_used

Synopsis

```
unsigned ctl_byte_queue_num_used(CTL_BYTE_QUEUE_t *q);
```

Description

ctl_byte_queue_num_used returns the number of used elements in the byte queue **q**.

ctl_byte_queue_post

Synopsis

```
unsigned ctl_byte_queue_post(CTL_BYTE_QUEUE_t *q,
                             unsigned char b,
                             CTL_TIMEOUT_t t,
                             CTL_TIME_t timeout);
```

Description

ctl_byte_queue_post posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the caller will block until the byte can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value.

ctl_byte_queue_post returns zero if the timeout occurred otherwise it returns one.

Note

ctl_byte_queue_post must not be called from an interrupt service routine.

ctl_byte_queue_post_multi

Synopsis

```
unsigned ctl_byte_queue_post_multi(CTL_BYTE_QUEUE_t *q,
                                   unsigned n,
                                   unsigned char *b,
                                   CTL_TIMEOUT_t t,
                                   CTL_TIME_t timeout);
```

Description

ctl_byte_queue_post_multi posts **n** bytes to the byte queue pointed to by **q**. The caller will block until the bytes can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value.

ctl_byte_queue_post_multi returns the number of bytes that were posted.

Note

ctl_byte_queue_post_multi must not be called from an interrupt service routine.

ctl_byte_queue_post_multi does not guarantee that the bytes will be all be posted to the byte queue atomically. If you have multiple tasks posting (multiple bytes) to the same byte queue then you may get unexpected results.

ctl_byte_queue_post_multi_nb

Synopsis

```
unsigned ctl_byte_queue_post_multi_nb(CTL_BYTE_QUEUE_t *q,
                                      unsigned n,
                                      unsigned char *b);
```

Description

ctl_byte_queue_post_multi_nb posts **n** bytes to the byte queue pointed to by **q**.

ctl_byte_queue_post_multi_nb returns the number of bytes that were posted.

ctl_byte_queue_post_multi_uc

Synopsis

```
void ctl_byte_queue_post_multi_uc(CTL_BYTE_QUEUE_t *q,
                                  unsigned n,
                                  unsigned char *b);
```

Description

ctl_byte_queue_post_multi_uc posts **n** bytes to the byte queue pointed to by **q**. The caller will unconditionally block until all bytes are posted.

Note

ctl_byte_queue_post_multi_uc must not be called from an interrupt service routine.

ctl_byte_queue_post_multi_uc does not guarantee that the bytes will be all be posted to the byte queue atomically. If you have multiple tasks posting (multiple bytes) to the same byte queue then you may get unexpected results.

ctl_byte_queue_post_nb

Synopsis

```
unsigned ctl_byte_queue_post_nb(CTL_BYTE_QUEUE_t *q,
                                unsigned char b);
```

Description

ctl_byte_queue_post_nb posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the function will return zero otherwise it will return one.

ctl_byte_queue_post_uc

Synopsis

```
void ctl_byte_queue_post_uc(CTL_BYTE_QUEUE_t *q,
                            unsigned char b);
```

Description

ctl_byte_queue_post_uc posts **b** to the byte queue pointed to by **q**. If the byte queue is full then the caller will unconditionally block until the byte can be posted.

Note

ctl_byte_queue_post_uc must not be called from an interrupt service routine.

ctl_byte_queue_receive

Synopsis

```
unsigned ctl_byte_queue_receive(CTL_BYTE_QUEUE_t *q,
                                unsigned char *b,
                                CTL_TIMEOUT_t t,
                                CTL_TIME_t timeout);
```

Description

ctl_byte_queue_receive pops the oldest byte in the byte queue pointed to by **q** into the memory pointed to by **b**. **ctl_byte_queue_receive** will block if no bytes are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_byte_queue_receive returns zero if a timeout occurs otherwise 1.

Note

ctl_byte_queue_receive must not be called from an interrupt service routine.

ctl_byte_queue_receive_multi

Synopsis

```
unsigned ctl_byte_queue_receive_multi(CTL_BYTE_QUEUE_t *q,
                                      unsigned n,
                                      unsigned char *b,
                                      CTL_TIMEOUT_t t,
                                      CTL_TIME_t timeout);
```

Description

ctl_byte_queue_receive_multi pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed at by **b**. **ctl_byte_queue_receive_multi** will block until the number of bytes are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_byte_queue_receive_multi returns the number of bytes that have been received.

Note

ctl_byte_queue_receive_multi must not be called from an interrupt service routine.

ctl_byte_queue_receive_multi_nb

Synopsis

```
unsigned ctl_byte_queue_receive_multi_nb(CTL_BYTE_QUEUE_t *q,
                                         unsigned n,
                                         unsigned char *b);
```

Description

ctl_byte_queue_receive_multi_nb pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed to by **b**.

ctl_byte_queue_receive_multi_nb returns the number of bytes that have been received.

ctl_byte_queue_receive_multi_uc

Synopsis

```
void ctl_byte_queue_receive_multi_uc(CTL_BYTE_QUEUE_t *q,
                                     unsigned n,
                                     unsigned char *b);
```

Description

ctl_byte_queue_receive_multi_uc pops the oldest **n** bytes in the byte queue pointed to by **q** into the memory pointed at by **b**. **ctl_byte_queue_receive_multi_uc** will unconditionally block until all bytes are received.

Note

ctl_byte_queue_receive_multi_uc must not be called from an interrupt service routine.

ctl_byte_queue_receive_nb

Synopsis

```
unsigned ctl_byte_queue_receive_nb(CTL_BYTE_QUEUE_t *q,
                                   unsigned char *b);
```

Description

ctl_byte_queue_receive_nb pops the oldest byte in the byte queue pointed to by **m** into the memory pointed to by **b**. If no bytes are available the function returns zero otherwise it returns 1.

ctl_byte_queue_receive_uc

Synopsis

```
void ctl_byte_queue_receive_uc(CTL_BYTE_QUEUE_t *q,
                               unsigned char *b);
```

Description

ctl_byte_queue_receive_uc pops the oldest byte in the byte queue pointed to by **q** into the memory pointed to by **b**. **ctl_byte_queue_receive_uc** will unconditionally block if no bytes are available.

Note

ctl_byte_queue_receive_uc must not be called from an interrupt service routine.

ctl_byte_queue_setup_events

Synopsis

```
void ctl_byte_queue_setup_events(CTL_BYTE_QUEUE_t *q,
                                 CTL_EVENT_SET_t *e,
                                 CTL_EVENT_SET_t notempty,
                                 CTL_EVENT_SET_t notfull);
```

Description

ctl_byte_queue_setup_events registers events in the event set **e** that are set when the byte queue **q** becomes **notempty** or becomes **notfull**. No scheduling will occur with this operation, you need to do this before waiting for events.

ctl_current_time

Synopsis

```
CTL_TIME_t ctl_current_time;
```

Description

ctl_current_time holds the current time in ticks. **ctl_current_time** is incremented by **ctl_increment_ticks_from_isr**.

Note

For portable programs without race conditions you should not read this variable directly, you should use **ctl_get_current_time** instead. As this variable is changed by an interrupt, it cannot be read atomically on processors whose word size is less than 32 bits without first disabling interrupts. That said, you can read this variable directly in your interrupt handler as long as interrupts are still disabled.

Note

ctl_current_time is *not* declared volatile because doing so would cause the internal implementation of CTL to be less efficient. We advise you to use the access function **ctl_get_current_time** which provides clean and efficient access to the current time.

See Also

ctl_get_current_time.

ctl_events_init

Synopsis

```
void ctl_events_init(CTL_EVENT_SET_t *e,
                     CTL_EVENT_SET_t set);
```

Description

ctl_events_init initializes the event set **e** with the **set** values.

ctl_events_pulse

Synopsis

```
void ctl_events_pulse(CTL_EVENT_SET_t *e,
                      CTL_EVENT_SET_t set_then_clear);
```

Description

ctl_events_pulse will set the events defined by **set_then_clear** in the event set pointed to by **e**. **ctl_events_pulse** will then search the task list, matching tasks that are waiting on the event set **e**, and make them runnable if the match is successful. The events defined by **set_then_clear** are then cleared.

See Also

ctl_events_set_clear.

ctl_events_set_clear

Synopsis

```
void ctl_events_set_clear(CTL_EVENT_SET_t *e,
                          CTL_EVENT_SET_t set,
                          CTL_EVENT_SET_t clear);
```

Description

ctl_events_set_clear sets the events defined by **set** and clears the events defined by **clear** of the event set pointed to by **e**. **ctl_events_set_clear** will then search the task list, matching tasks that are waiting on the event set **e** and make them runnable if the match is successful.

See Also

ctl_events_pulse.

ctl_events_wait

Synopsis

```
unsigned ctl_events_wait(CTL_EVENT_WAIT_TYPE_t waitType,
                         CTL_EVENT_SET_t *eventSet,
                         CTL_EVENT_SET_t events,
                         CTL_TIMEOUT_t timeoutType,
                         CTL_TIME_t timeout);
```

Description

ctl_events_wait waits for **events** to be set (value 1) in the event set pointed to by **eventSet** with an optional **timeout** applied if **timeoutType** is non-zero.

The **waitType** can be one of:

- **CTL_EVENT_WAIT_ANY_EVENTS** — wait for any of **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** — wait for any of **events** in **eventSet** to be set and reset (clear) them.
- **CTL_EVENT_WAIT_ALL_EVENTS** — wait for all **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR** — wait for all **events** in **eventSet** to be set and reset (clear) them.

ctl_events_wait returns the value pointed to by **eventSet** before any auto-clearing occurred or zero if the **timeout** occurred.

Note

ctl_events_wait must not be called from an interrupt service routine.

ctl_events_wait_nb

Synopsis

```
unsigned ctl_events_wait_nb(CTL_EVENT_WAIT_TYPE_t waitType,
                            CTL_EVENT_SET_t *eventSet,
                            CTL_EVENT_SET_t events);
```

Description

ctl_events_wait_nb waits for **events** to be set (value 1) in the event set pointed to by **eventSet** without blocking.

The **waitType** can be one of:

- **CTL_EVENT_WAIT_ANY_EVENTS** — wait for any of **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** — wait for any of **events** in **eventSet** to be set and reset (clear) them.
- **CTL_EVENT_WAIT_ALL_EVENTS** — wait for all **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR** — wait for all **events** in **eventSet** to be set and reset (clear) them.

ctl_events_wait_nb returns the value pointed to by **eventSet** before any auto-clearing occurred.

Note

ctl_events_wait_nb must not be called from an interrupt service routine.

ctl_events_wait_uc

Synopsis

```
unsigned ctl_events_wait_uc(CTL_EVENT_WAIT_TYPE_t waitType,
                            CTL_EVENT_SET_t *eventSet,
                            CTL_EVENT_SET_t events);
```

Description

ctl_events_wait_uc unconditionally waits for **events** to be set (value 1) in the event set pointed to by **eventSet**.

The **waitType** can be one of:

- **CTL_EVENT_WAIT_ANY_EVENTS** — wait for any of **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ANY_EVENTS_WITH_AUTO_CLEAR** — wait for any of **events** in **eventSet** to be set and reset (clear) them.
- **CTL_EVENT_WAIT_ALL_EVENTS** — wait for all **events** in **eventSet** to be set.
- **CTL_EVENT_WAIT_ALL_EVENTS_WITH_AUTO_CLEAR** — wait for all **events** in **eventSet** to be set and reset (clear) them.

ctl_events_wait_uc returns the value pointed to by **eventSet** before any auto-clearing occurred.

Note

ctl_events_wait_uc must not be called from an interrupt service routine.

ctl_get_current_time

Synopsis

```
CTL_TIME_t ctl_get_current_time(void);
```

Description

ctl_get_current_time atomically reads the value of **ctl_current_time**.

ctl_get_sleep_delay

Synopsis

```
unsigned ctl_get_sleep_delay(void);
```

Description

ctl_get_sleep_delay returns the minimal sleep delay for the tasks on the task list. This is intended for use by tickless CTL implementations.

ctl_global_interrupts_disable

Synopsis

```
int ctl_global_interrupts_disable(void);
```

Description

ctl_global_interrupts_disable disables global interrupts. If **ctl_global_interrupts_disable** is called and interrupts are already disabled then it will return 0. If **ctl_global_interrupts_disable** is called and interrupts are enabled then it will return non-zero which may or may not represent the true interrupt disabled state. ctl_global_interrupts_disable is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_global_interrupts_enable

Synopsis

```
void ctl_global_interrupts_enable(void);
```

Description

ctl_global_interrupts_enable enables global interrupts. ctl_global_interrupts_enable is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_global_interrupts_set

Synopsis

```
int ctl_global_interrupts_set(int enable);
```

Description

ctl_global_interrupts_set disables or enables global interrupts according to the state **enable**. If **enable** is zero, interrupts are disabled and if **enable** is non-zero, interrupts are enabled. If **ctl_global_interrupts_set** is called and interrupts are already disabled then it will return 0. If **ctl_global_interrupts_set** is called and interrupts are enabled then it will return non-zero which may or may not represent the true interrupt disabled state. ctl_global_interrupts_set is used to provide exclusive access to CTL data structures the implementation of it may or may not disable global interrupts.

ctl_handle_error

Synopsis

```
void ctl_handle_error(CTL_ERROR_CODE_t e);
```

Description

ctl_handle_error is a function that you must supply in your application that handles errors detected by the CrossWorks tasking library.

The errors that can be reported in **e** are are described in CTL_ERROR_CODE_t.

ctl_increment_tick_from_isr

Synopsis

```
void ctl_increment_tick_from_isr(void);
```

Description

ctl_increment_tick_from_isr increments **ctl_current_time** by the number held in **ctl_time_increment** and does rescheduling.

Note

ctl_increment_tick_from_isr must only be invoked by an interrupt service routine.

ctl_interrupt_count

Synopsis

```
unsigned char ctl_interrupt_count;
```

Description

ctl_interrupt_count contains a count of the interrupt nesting level. This variable must be incremented immediately on entry to an interrupt service routine and decremented immediately before return from the interrupt service routine.

ctl_last_schedule_time

Synopsis

```
CTL_TIME_t ctl_last_schedule_time;
```

Description

ctl_last_schedule_time contains the time (in ticks) of the last task schedule.

Description

ctl_last_schedule_time contains the time of the last reschedule in ticks.

ctl_memory_area_allocate

Synopsis

```
unsigned *ctl_memory_area_allocate(CTL_MEMORY_AREA_t *memory_area);
```

Description

ctl_memory_area_allocate allocates a block from the initialized memory area **memory_area**. **ctl_memory_area_allocate** returns a block of the size specified in the call to ctl_memory_area_init or zero if no blocks are available.

ctl_memory_area_allocate executes in constant time and is very fast. You can call **ctl_memory_area_allocate** from an interrupt service routine, from a task, or from initialization code.

ctl_memory_area_free

Synopsis

```
void ctl_memory_area_free(CTL_MEMORY_AREA_t *memory_area,
                          unsigned *block);
```

Description

ctl_memory_area_free is given a pointer to a memory area **memory_area** which has been initialized and a **block** that has been returned by **ctl_memory_area_allocate**. The block is returned to the memory area so that it can be allocated again.

ctl_memory_area_init

Synopsis

```
void ctl_memory_area_init(CTL_MEMORY_AREA_t *memory_area,
                          unsigned *memory,
                          unsigned block_size_in_words,
                          unsigned num_blocks);
```

Description

ctl_memory_area_init is given a pointer to the memory area to initialize in **memory_area**. The array that is used to implement the memory area is pointed to by **memory**. The size of a memory block is given supplied in **block_size_in_words** and the number of block is supplied in **num_blocks**.

Note

memory must point to a block of memory that is at least **block_size_in_words** × **num_blocks** words long.

ctl_memory_area_setup_events

Synopsis

```
void ctl_memory_area_setup_events(CTL_MEMORY_AREA_t *m,
                                  CTL_EVENT_SET_t *e,
                                  CTL_EVENT_SET_t blockavailable);
```

Description

ctl_memory_area_setup_events registers the events **blockavailable** in the event set **e** that are set when a block becomes available in the the memory area **m**.

ctl_message_queue_init

Synopsis

```
void ctl_message_queue_init(CTL_MESSAGE_QUEUE_t *q,
                            void **queue,
                            unsigned queue_size);
```

Description

ctl_message_queue_init is given a pointer to the message queue to initialize in **q**. The array that will be used to implement the message queue pointed to by **queue** and its size in **queue_size** are also supplied.

ctl_message_queue_num_free

Synopsis

```
unsigned ctl_message_queue_num_free(CTL_MESSAGE_QUEUE_t *q);
```

Description

ctl_message_queue_num_free returns the number of free elements in the message queue **q**.

ctl_message_queue_num_used

Synopsis

```
unsigned ctl_message_queue_num_used(CTL_MESSAGE_QUEUE_t *q);
```

Description

ctl_message_queue_num_used returns the number of used elements in the message queue **q**.

ctl_message_queue_post

Synopsis

```
unsigned ctl_message_queue_post(CTL_MESSAGE_QUEUE_t *q,
                                void *message,
                                CTL_TIMEOUT_t t,
                                CTL_TIME_t timeout);
```

Description

ctl_message_queue_post posts **message** to the message queue pointed to by **q**. If the message queue is full then the caller will block until the message can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value.

ctl_message_queue_post returns zero if the timeout occurred otherwise it returns one.

Note

ctl_message_queue_post must not be called from an interrupt service routine.

ctl_message_queue_post_multi

Synopsis

```
unsigned ctl_message_queue_post_multi(CTL_MESSAGE_QUEUE_t *q,
                                      unsigned n,
                                      void **messages,
                                      CTL_TIMEOUT_t t,
                                      CTL_TIME_t timeout);
```

Description

ctl_message_queue_post_multi posts **n messages** to the message queue pointed to by **q**. The caller will block until the messages can be posted or, if **timeoutType** is non-zero, the current time reaches **timeout** value.

ctl_message_queue_post_multi returns the number of messages that were posted.

Note

ctl_message_queue_post_multi must not be called from an interrupt service routine.

ctl_message_queue_post_multi function does not guarantee that the messages will be all be posted to the message queue atomically. If you have multiple tasks posting (multiple messages) to the same message queue then you may get unexpected results.

ctl_message_queue_post_multi_nb

Synopsis

```
unsigned ctl_message_queue_post_multi_nb(CTL_MESSAGE_QUEUE_t *q,
                                         unsigned n,
                                         void **messages);
```

Description

ctl_message_queue_post_multi_nb posts **n messages** to the message queue pointed to by **m**.

ctl_message_queue_post_multi_nb returns the number of messages that were posted.

ctl_message_queue_post_multi_uc

Synopsis

```
void ctl_message_queue_post_multi_uc(CTL_MESSAGE_QUEUE_t *q,
                                     unsigned n,
                                     void **messages);
```

Description

ctl_message_queue_post_multi_uc posts **n messages** to the message queue pointed to by **q**. The caller will unconditionally block until all messages are posted.

Note

ctl_message_queue_post_multi_uc must not be called from an interrupt service routine.

ctl_message_queue_post_multi_uc function does not guarantee that the messages will be all be posted to the message queue atomically. If you have multiple tasks posting (multiple messages) to the same message queue, then you may get unexpected results.

ctl_message_queue_post_nb

Synopsis

```
unsigned ctl_message_queue_post_nb(CTL_MESSAGE_QUEUE_t *q,
                                   void *message);
```

Description

ctl_message_queue_post_nb posts **message** to the message queue pointed to by **q**. If the message queue is full then the function will return zero otherwise it will return one.

ctl_message_queue_post_uc

Synopsis

```
void ctl_message_queue_post_uc(CTL_MESSAGE_QUEUE_t *q,
                               void *message);
```

Description

ctl_message_queue_post_uc posts **message** to the message queue pointed to by **q**. If the message queue is full then the caller will unconditionally block until the message can be posted.

Note

ctl_message_queue_post_uc must not be called from an interrupt service routine.

ctl_message_queue_receive

Synopsis

```
unsigned ctl_message_queue_receive(CTL_MESSAGE_QUEUE_t *q,
                                   void **message,
                                   CTL_TIMEOUT_t t,
                                   CTL_TIME_t timeout);
```

Description

ctl_message_queue_receive pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive** will block if no messages are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_message_queue_receive returns zero if a timeout occurs otherwise 1.

Note

ctl_message_queue_receive must not be called from an interrupt service routine.

ctl_message_queue_receive_multi

Synopsis

```
unsigned ctl_message_queue_receive_multi(CTL_MESSAGE_QUEUE_t *q,
                                         unsigned n,
                                         void **messages,
                                         CTL_TIMEOUT_t t,
                                         CTL_TIME_t timeout);
```

Description

ctl_message_queue_receive_multi pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive_multi** will block until all the messages are available unless **timeoutType** is non-zero and the current time reaches the **timeout** value.

ctl_message_queue_receive_multi returns the number of messages that were received.

Note

ctl_message_queue_receive_multi must not be called from an interrupt service routine.

ctl_message_queue_receive_multi_nb

Synopsis

```
unsigned ctl_message_queue_receive_multi_nb(CTL_MESSAGE_QUEUE_t *q,
                                            unsigned n,
                                            void **messages);
```

Description

ctl_message_queue_receive_multi_nb pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**.

ctl_message_queue_receive_multi_nb returns the number of messages that were received.

ctl_message_queue_receive_multi_uc

Synopsis

```
void ctl_message_queue_receive_multi_uc(CTL_MESSAGE_QUEUE_t *q,
                                        unsigned n,
                                        void **messages);
```

Description

ctl_message_queue_receive_multi_uc pops the oldest **n** messages in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive_multi_uc** will unconditionally block until all the messages are received.

Note

ctl_message_queue_receive_multi_uc must not be called from an interrupt service routine.

ctl_message_queue_receive_nb

Synopsis

```
unsigned ctl_message_queue_receive_nb(CTL_MESSAGE_QUEUE_t *q,
                                      void **message);
```

Description

ctl_message_queue_receive_nb pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. If no messages are available the function returns zero otherwise it returns 1.

ctl_message_queue_receive_uc

Synopsis

```
void ctl_message_queue_receive_uc(CTL_MESSAGE_QUEUE_t *q,
                                  void **message);
```

Description

ctl_message_queue_receive_uc pops the oldest message in the message queue pointed to by **q** into the memory pointed to by **message**. **ctl_message_queue_receive_uc** will unconditionally block if no messages are available.

Note

ctl_message_queue_receive_uc must not be called from an interrupt service routine.

ctl_message_queue_setup_events

Synopsis

```
void ctl_message_queue_setup_events(CTL_MESSAGE_QUEUE_t *q,
                                    CTL_EVENT_SET_t *e,
                                    CTL_EVENT_SET_t notempty,
                                    CTL_EVENT_SET_t notfull);
```

Description

ctl_message_queue_setup_events registers events in the event set **e** that are set when the message queue **q** becomes **notempty** or becomes **notfull**. No scheduling will occur with this operation, you need to do this before waiting for events.

ctl_mutex_init

Synopsis

```
void ctl_mutex_init(CTL_MUTEX_t *m);
```

Description

ctl_mutex_init initializes the mutex pointed to by **m**.

ctl_mutex_lock

Synopsis

```
unsigned ctl_mutex_lock(CTL_MUTEX_t *m,
                        CTL_TIMEOUT_t t,
                        CTL_TIME_t timeout);
```

Description

ctl_mutex_lock locks the mutex pointed to by **m** to the calling task. If the mutex is already locked by the calling task then the mutex lock count is incremented. If the mutex is already locked by a different task then the caller will block until the mutex is unlocked. In this case, if the priority of the task that has locked the mutex is less than that of the caller the priority of the task that has locked the mutex is raised to that of the caller whilst the mutex is locked.

If **timeoutType** is non-zero and the current time reaches the **timeout** value before the lock is acquired the function returns zero otherwise it returns one.

Note

ctl_mutex_lock must not be called from an interrupt service routine.

ctl_mutex_lock_nb

Synopsis

```
unsigned ctl_mutex_lock_nb(CTL_MUTEX_t *m);
```

Description

ctl_mutex_lock_nb locks the mutex pointed to by **m** to the calling task. If the mutex is already locked by the calling task then the mutex lock count is incremented. If the mutex is already locked by a different task then zero is returned otherwise 1 is returned.

Note

ctl_mutex_lock_nb must not be called from an interrupt service routine.

ctl_mutex_lock_uc

Synopsis

```
void ctl_mutex_lock_uc(CTL_MUTEX_t *m);
```

Description

ctl_mutex_lock_uc locks the mutex pointed to by **m** to the calling task. If the mutex is already locked by the calling task then the mutex lock count is incremented. If the mutex is already locked by a different task then the caller will unconditionally block until the mutex is unlocked. In this case, if the priority of the task that has locked the mutex is less than that of the caller, the priority of the task that has locked the mutex is raised to that of the caller whilst the mutex is locked.

Note

ctl_mutex_lock_uc must not be called from an interrupt service routine.

ctl_mutex_unlock

Synopsis

```
void ctl_mutex_unlock(CTL_MUTEX_t *m);
```

Description

ctl_mutex_unlock unlocks the mutex pointed to by **m**. The mutex must have previously been locked by the calling task. If the calling task's priority has been raised (by another task calling **ctl_mutex_unlock** whilst the mutex was locked), then the calling tasks priority will be restored.

Note

ctl_mutex_unlock must not be called from an interrupt service routine.

ctl_reschedule_on_last_isr_exit

Synopsis

```
unsigned char ctl_reschedule_on_last_isr_exit;
```

Description

ctl_reschedule_on_last_isr_exit is set to a non-zero value if a CTL call is made from an interrupt service routine that requires a task reschedule. This variable is checked and reset on exit from the last nested interrupt service routine.

ctl_semaphore_init

Synopsis

```
void ctl_semaphore_init(CTL_SEMAPHORE_t *s,
                        unsigned value);
```

Description

ctl_semaphore_init initializes the semaphore pointed to by **s** to **value**.

ctl_semaphore_signal

Synopsis

```
void ctl_semaphore_signal(CTL_SEMAPHORE_t *s);
```

Description

ctl_semaphore_signal signals the semaphore pointed to by **s**. If tasks are waiting for the semaphore then the highest priority task will be made runnable. If no tasks are waiting for the semaphore then the semaphore value will be incremented.

ctl_semaphore_wait

Synopsis

```
unsigned ctl_semaphore_wait(CTL_SEMAPHORE_t *s,
                            CTL_TIMEOUT_t t,
                            CTL_TIME_t timeout);
```

Description

ctl_semaphore_wait waits for the semaphore pointed to by **s** to be non-zero. If the semaphore is zero then the caller will block unless **timeoutType** is non-zero and the current time reaches the **timeout** value. If the timeout occurred **ctl_semaphore_wait** returns zero otherwise it returns one.

Note

ctl_semaphore_wait must not be called from an interrupt service routine.

ctl_semaphore_wait_nb

Synopsis

```
unsigned ctl_semaphore_wait_nb(CTL_SEMAPHORE_t *s);
```

Description

ctl_semaphore_wait_nb waits for the semaphore pointed to by **s** without blocking. Returns returns one on success.

Note

ctl_semaphore_wait_nb must not be called from an interrupt service routine.

ctl_semaphore_wait_uc

Synopsis

```
void ctl_semaphore_wait_uc(CTL_SEMAPHORE_t *s);
```

Description

ctl_semaphore_wait_uc unconditionally waits for the semaphore pointed to by **s** to be non-zero. If the semaphore is zero then the caller will block.

Note

ctl_semaphore_wait_uc must not be called from an interrupt service routine.

ctl_task_die

Synopsis

```
void ctl_task_die(void);
```

Description

ctl_task_die terminates the currently executing task and schedules the next runnable task.

ctl_task_executing

Synopsis

```
CTL_TASK_t *ctl_task_executing;
```

Description

ctl_task_executing points to the **CTL_TASK_t** structure of the currently executing task. The **priority** field is the only field in the **CTL_TASK_t** structure that is defined for the task that is executing. It is an error if **ctl_task_executing** is **NULL**.

ctl_task_init

Synopsis

```
void ctl_task_init(CTL_TASK_t *task,
                   unsigned char priority,
                   const char *name);
```

Description

ctl_task_init turns the main program into a task. This function takes a pointer in **task** to the **CTL_TASK_t** structure that represents the main task, its **priority** (0 is the lowest priority, 255 the highest), and a zero-terminated string pointed by **name**. On return from this function global interrupts will be enabled.

The function must be called before any other CrossWorks tasking library calls are made.

ctl_task_list

Synopsis

```
CTL_TASK_t *ctl_task_list;
```

Description

ctl_task_list points to the **CTL_TASK_t** structure of the highest priority task that is not executing. It is an error if **ctl_task_list** is **NULL**.

ctl_task_remove

Synopsis

```
void ctl_task_remove(CTL_TASK_t *task);
```

Description

ctl_task_remove removes the task **task** from the waiting task list. Once you you have removed a task the only way to re-introduce it to the system is to call **ctl_task_restore**.

ctl_task_reschedule

Synopsis

```
void ctl_task_reschedule(void);
```

Description

ctl_task_reschedule causes a reschedule to occur. This can be used by tasks of the same priority to share the CPU without using timeslicing.

ctl_task_restore

Synopsis

```
void ctl_task_restore(CTL_TASK_t *task);
```

Description

ctl_task_restore adds a task **task** that was removed (using **ctl_task_remove**) onto the task list and do scheduling.

ctl_task_run

Synopsis

```
void ctl_task_run(CTL_TASK_t *task,
                  unsigned char priority,
                  void (*entrypoint)(void *),
                  void *parameter,
                  const char *name,
                  unsigned stack_size_in_words,
                  unsigned *stack,
                  unsigned call_size_in_words);
```

Description

ctl_task_run takes a pointer in **task** to the **CTL_TASK_t** structure that represents the task. The **priority** can be zero for the lowest priority up to 255 which is the highest. The **entrypoint** parameter is the function that the task will execute which has the **parameter** passed to it.

name is a pointer to a zero-terminated string used for debug purposes.

The start of the memory used to hold the stack that the task will execute in is **stack** and the size of the memory is supplied in **stack_size_in_words**. On systems that have two stacks (e.g. Atmel AVR) then the **call_size_in_words** parameter must be set to specify the number of stack elements to use for the call stack.

ctl_task_set_priority

Synopsis

```
unsigned char ctl_task_set_priority(CTL_TASK_t *task,
                                    unsigned char priority);
```

Description

ctl_task_set_priority changes the priority of **task** to **priority**. The priority can be 0, the lowest priority, to 255, which is the highest priority.

You can change the priority of the currently executing task by passing **ctl_task_executing** as the **task** parameter.

ctl_task_set_priority returns the previous priority of the task.

ctl_task_switch_callout

Synopsis

```
void (*ctl_task_switch_callout)(CTL_TASK_t *);
```

Description

ctl_task_switch_callout contains a pointer to a function that is called (if it is set) when a task schedule occurs. The task that will be scheduled is supplied as a parameter to the function (**ctl_task_executing** will point to the currently scheduled task).

Note that the callout function is called from the CTL scheduler and as such any use of CTL services whilst executing the callout function has undefined behavior.

Note

Because this function pointer is called in an interrupt service routine, you should assign it before interrupts are started or with interrupts turned off.

ctl_time_increment

Synopsis

```
unsigned ctl_time_increment;
```

Description

ctl_time_increment contains the value that **ctl_current_time** is incremented when **ctl_increment_tick_from_isr** is called.

ctl_timeout_wait

Synopsis

```
void ctl_timeout_wait(CTL_TIME_t timeout);
```

Description

ctl_timeout_wait takes the **timeout** (not the duration) as a parameter and suspends the calling task until the current time reaches the timeout.

Note

ctl_timeout_wait must not be called from an interrupt service routine.

ctl_timeslice_period

Synopsis

```
CTL_TIME_t ctl_timeslice_period;
```

Description

ctl_timeslice_period contains the number of ticks to allow a task to run before it will be preemptively rescheduled by a task of the same priority. The variable is set to zero by default so that only higher priority tasks will be preemptively scheduled.