STMicroelectronics

OS21

User manual

7358306 Rev V

August 2010

www.st.com

BLANK

User manual

OS21

Introduction

OS21 is a royalty free, light weight, multitasking operating system. It is based on the existing OS20 API and is intended for applications where small footprint, and excellent real time responsiveness are required. It provides a multi-priority preemptive scheduler, with low context switch and interrupt handling latencies.

OS21 provides portable APIs to handle task, memory, messaging, interrupts, exceptions, synchronization, and time management. It also provides target specific APIs for various chip devices.

Multi-tasking is widely accepted as an optimal method of implementing real-time systems. Applications may be broken down into several independent tasks which co-ordinate their use of shared system resources, such as memory and CPU time. External events arriving from peripheral devices are made known to the system through interrupts.

The OS21 real-time kernel provides comprehensive multi-tasking services. Tasks synchronize their activities and communicate with each other using semaphores and message queues. Real world events are handled using interrupt routines and communicated to tasks using semaphores. Memory allocation for tasks is selectively managed by OS21 or the user. Tasks may be given priorities and are scheduled accordingly. Timer functions are provided to implement time and delay functions.

Virtual memory provides a way of controlling memory access. Out-of-course events can be dealt with by exception handlers. A power management framework is provided.

Contents

Preface

Document identification and control

Each book carries a unique ADCS identifier of the form:

ADCS *nnnnnnnx*

where *nnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on a document, the complete identification ADCS *nnnnnnnx* should be quoted.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- `Sample code`, `keyboard input` and `file names`,
- *Variables* and *`code variables`*,
- *`code comments`*,
- **Screens**, **windows** and **dialog boxes**,
- **Instructions**.

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,
- PIN NAMES and SIGNAL NAMES.

Software notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1. Terminal strings of the language, that is, strings not built up by rules of the language, are printed in teletype font. For example, `void`.
2. Nonterminal strings of the language, that is, strings built up by rules of the language, are printed in italic teletype font. For example, *`name`*.
3. If a nonterminal string of the language starts with a nonitalicized part, it is equivalent to the same nonterminal string without that nonitalicized part. For example, `vspace-`*`name`*.
4. Each phrase definition is built up using a double colon and an equals sign to separate the two sides (‘`::=`’).
5. Alternatives are separated by vertical bars (‘`|`’).
6. Optional sequences are enclosed in square brackets (‘`[`’ and ‘`]`’).
7. Items which may be repeated appear in braces (‘`{`’ and ‘`}`’).

1 OS21 overview

The OS21 kernel features:

- multi-priority preemptive scheduling based on 256 levels of priority
- semaphores
- mutexes
- message queues
- high resolution timers
- memory management
- interrupt handling
- virtual memory
- exception handling
- very small memory requirement
- power management framework

Each OS21 service can be used largely independently of any other service and this division into different services is seen in several places.

- Each service has its own header file, which defines all the variables, macros, types and functions for that service, see *Table 1*.
- All the symbols defined by a service have the service name as the first component of the name, see *Table 1*.

Table 1. OS21 include files

Header	Description
`os21.h`	Main include file
`os21/cache.h`	Cache functions
`os21/callback.h`	Callback functions
`os21/event.h`	Event flag functions
`os21/exception.h`	Exception functions
`os21/interrupt.h`	Interrupt functions
`os21/kernel.h`	Kernel functions
`os21/message.h`	Message handling functions
`os21/mutex.h`	Mutex functions
`os21/ostime.h`	Timer functions
`os21/partition.h`	Memory functions
`os21/power.h`	Power management
`os21/profile.h`	Profiler functions
`os21/semaphore.h`	Semaphore functions
`os21/task.h`	Task functions
`os21/typedefs.h`	OS21 types

Table 1. OS21 include files (continued)

Header	Description
os21/[target]	Target specific files
os21/vmem.h	Virtual memory

By including the header file os21.h, all the above header files are automatically included. The target specific API files in *Table 1* include typedefs and APIs relating to register contexts and interrupts.

1.1 Naming

All the functions in OS21 follow a common naming scheme. This is:

service_action[*_qualifier*]

where *service* is the service name, which groups all the functions, and *action* is the operation to be performed. *qualifier* is an optional keyword which is used where there are different styles of operation.

1.2 How this document is organized

The division of OS21 functions into services is also used in this manual. Each of the major service types is described separately, using a common layout:

- an overview of the service, and the facilities it provides
- a list of the macros, types and functions defined by the service header file
- a detailed description of each of the functions in the service

The remaining sections of this chapter describe the main concepts on which OS21 is founded.

1.3 Differences between OS20 and OS21

OS20 contains many aspects which relate specifically to the ST20 CPU, in its various versions. These aspects of the API are not present in OS21 (for example, channels and high priority processes).

Where parts of the OS20 API have grown to exploit facilities which exist solely on the ST20, OS21 preserves the interface, but the functionality is the same as the root API call. An example is the _timeout() functions for semaphores and message queues. These are generic calls, but OS20 also provides non _timeout() versions, which are mapped directly to ST20 hardware semaphores. OS21 preserves the API, but the non _timeout() versions map directly on to the generic calls.

OS20 uses header files with 8.3 names. OS21 is not constrained by this limitation, and uses meaningful names, which will not clash with other headers.

The following classes of API calls are common between OS20 and OS21, with OS21 presenting either exactly the same interface, or a super set:

- kernel API
- memory and partitions
- task and scheduler APIs
- semaphore API
- message API
- time API

In all the above APIs there is one notable difference between OS20 and OS21. OS21 no longer supports the `_init()` family of calls. OS21 presents an enhanced partition API, which is a super set of the OS20 API, and provides much of functionality of the `_init()` calls. See *Chapter 3: Memory and partitions on page 25*.

OS21 has also added the following.

- Mutexes, a new class of synchronization object. These offer extra facilities beyond simple binary semaphores. See *Chapter 7: Mutexes on page 110*.
- The concept of event flags, which allow tasks to wait on a combination of events occurring. See *Chapter 8: Event flags on page 119*.
- A portable interrupt API. See *Chapter 11: Interrupts on page 145*.
- A portable cache API. See *Chapter 12: Caches and memory areas on page 163*.
- A portable virtual memory API. See *Chapter 13: Virtual memory on page 173*.
- A portable exception API. See *Chapter 14: Exceptions on page 180*.
- A power management API. See *Chapter 16: Power management on page 192*.

There has been a minor change to the behavior of priority semaphores between OS20 and OS21. When a task is queued on a priority semaphore in OS20, its position in the queue is determined statically when the task is queued. Subsequent modification of the task's priority will not change its position in the queue, hence the priority queue can become unordered. In OS21 this has been fixed; changing a task's priority while it is on a priority ordered queue moves the task to the appropriate place in the queue, so correct ordering of the queue is maintained.

OS21's concept of time differs to that of OS20, however, providing the mandated OS20 time manipulation functions are used, compatibility is retained. OS20 represents time (in the form of clock ticks) as a 32-bit quantity. This results in a limited timer range, and the notion of timer wrapping. In OS21 this range is extended by representing clock ticks as a signed 64-bit quantity. This eliminates the clock range restrictions of OS20 and also timer wrap.

Cache API calls differ from earlier OS20 versions. Prior to v.3.0.2 of OS21, cache API calls are defined as target specific. In v.3.0.2 and following, cache API calls are generic across all platforms.

New target specific APIs may be added to OS21 which are not present in OS20, or OS21 on other targets.

OS21 uses two symbolic constants to represent success and failure. They are defined as follows:

`OS21_SUCCESS`	The value 0
`OS21_FAILURE`	The value -1

OS20 used two pre-defined memory partitions which the user could access, the `system_partion` and the `internal_partion`. OS21 does not have any pre-defined memory partitions. The system heap in OS21 can either be managed by the C runtime library routines such as `malloc()` and `free()`, or by OS21.

1.4 Classes and objects

OS21 uses an object oriented style of programming. This will be familiar to users of C++, however it is useful to understand how this has been applied to OS21, and how it has been implemented in the C language.

Each of the major services of OS21 is represented by a class, for example:

- memory partitions
- tasks
- semaphores
- mutexes

A class is a purely abstract concept, which describes a collection of data items and a list of operations which can be performed on it. An object represents a concrete instance of a particular class. An object consists of a data structure in memory which describes the current state of the object, with information to describe how operations applied to that object will affect it and the rest of the system.

For many classes within OS21, there are different flavors. For example, the semaphore class has FIFO and priority flavors. When a particular object is created, the flavor required must be specified by using a qualifier on the object creation function that is fixed for the lifetime of that object. All the operations specified by a particular class can be applied to all objects of that class, however, how they behave may depend on the flavor of that class. So, the exact behavior of `semaphore_wait()` depends on whether it is applied to a FIFO or priority semaphore object.

Once an object has been created, all the data which represents that object is encapsulated within it. Functions are provided to modify or retrieve this data.

To provide this abstraction within OS21, using only standard C language features, most functions which operate on an object take the address of the object as their first parameter. This provides a level of type checking at compile time, for example, to ensure that a message queue operation is not applied to a semaphore. The only functions which are applied to an object, and which do not take the address of the object as a first parameter are those where the object in question can be inferred. For example, when an operation can only be applied to the current task, there is no need to specify its address.

1.4.1 Object lifetime

All objects can be created using the *`class`*`_create` or *`class`*`_create_p` functions. These allocate whatever memory is required to store the object, and return a pointer to the object. The pointer can then be used in all subsequent operations on that object.

When using *`class`*`_create` calls, the memory for the object structure is allocated from the `system` partition. Therefore this partition must be initialized (by calling `kernel_initialize()`) before any *`class`*`_create` calls are made. *Chapter 3: Memory and partitions on page 25* describes the `system` partition in more detail.

When using the *class*_create_p calls, OS21 allocates space from a user nominated partition.

The number of objects which can be created is only limited to the available memory, there are no fixed size lists within OS21's implementation.

When an object is no longer required, it must be deleted by calling the appropriate *class*_delete function. If objects are not deleted and memory is reused, OS21 and the debugger's knowledge of valid objects becomes corrupted.

Using the appropriate *class*_delete function has several effects.

- The object is removed from any lists within OS21, so will no longer appear in the debugger's list of known objects.
- The memory allocated for the object will be freed back to the appropriate partition.

Note: The objects created using both *class*_create *and* *class*_create_p *are deleted using* *class*_delete*.*

A deleted object cannot continue to be used. Any attempt to use a deleted object results in undefined behavior.

1.5 Defining memory partitions

Memory blocks are allocated and freed from memory partitions for dynamic memory management. OS21 supports three pre-defined types of memory partition, **heap**, **fixed**, and **simple**, as described in *Chapter 3: Memory and partitions on page 25*. The different styles of memory partition allow trade-offs between execution times and memory utilization. In addition to these pre-defined partition types, OS21 allows for user defined partition types to be easily created, should a different allocator be more appropriate for an application (for example, the buddy algorithm).

An important use of memory partitions is for object allocation. When using the *class*_create versions of the library functions to create objects, OS21 allocates memory for the object from the pre-defined memory **system** partition. This partition must be defined (by calling kernel_initialize()) before any of the create_ functions are called. When using the *class*_create_p versions of the library functions to create objects, the user can specify which partition to allocate from.

The standard C runtime memory allocation routines (for example, malloc() and free()) can be used, and these work on the system heap as normal.

1.6 Tasks

Tasks are the main elements of the OS21 multi-tasking facilities. A task describes the behavior of a discrete, separable component of an application. It behaves like a separate program, except that it can communicate with other tasks. New tasks may be generated dynamically by any existing task. There is no limit on the number of tasks in the system, beyond physical memory limitations.

Each task has its own data area in memory, including its own stack and the current state of the task. These data areas can be allocated by OS21 from the **system** partition or specified by the user. The code, global static data area and heap area are all shared between tasks. Two tasks may use the same code with no penalty. Sharing static data between tasks must

be done with care, and is not recommended as a means of communication between tasks without explicit synchronization.

Applications can be broken into any number of tasks provided there is sufficient memory. The overhead for generating and scheduling tasks is small in terms of processor time and memory.

Tasks are described in more detail in *Chapter 4: Tasks on page 44*.

1.7 Priority

The order in which tasks are run is governed by each task's **priority**. Normally the task which has the highest priority is the task which runs. All tasks of lower priority are prevented from running until the highest priority task deschedules.

If desired, when there are two or more tasks of the same priority waiting to run, they can each be run for a short period, dividing the use of the CPU between them. This is called **timeslicing**.

A task's priority is set when the task is created, although it may be changed later. OS21 provides the user with 256 levels of priority.

To implement multi-priority scheduling, OS21 uses a scheduling kernel which must be installed and started, before any tasks are created. This is described in *Chapter 2: Kernel on page 17*.

1.8 Semaphores

OS21 uses semaphores to synchronize multiple tasks. They are used to ensure mutual exclusion and control access to a shared resource.

Semaphores are also used for synchronization between interrupt handlers and tasks. Semaphores are described in more detail in *Chapter 6: Semaphores on page 100*.

1.9 Mutexes

OS21 uses mutexes to create critical regions. Mutexes can only be owned by one task at a time, but also allow an owning task to take a mutex multiple times without deadlock. They provide simple FIFO queuing of tasks, or priority based queuing with priority inversion correction. Mutexes are described in detail in *Chapter 7: Mutexes on page 110*.

1.10 Event flags

OS21 provides event flags, which allow tasks to wait for an arbitrary combination of events to occur. See *Chapter 8: Event flags on page 119*.

1.11 Message queues

Message queues provide a buffered communication method for tasks, described in *Chapter 9: Message handling on page 129*.

1.12 Clocks

OS21 provides several clock functions to read the current time, to pause the execution of a task until a specified time and to time-out an input communication. *Chapter 10: Real-time clocks on page 140* provides an overview of how time is handled in OS21. Time-out related functions are described in *Chapter 4: Tasks on page 44*, *Chapter 6: Semaphores on page 100* and *Chapter 9: Message handling on page 129*.

OS21 provides a high resolution timer by efficiently using the hardware timer provided on the device.

1.13 Interrupts

A comprehensive set of interrupt handling functions is provided by OS21 to enable external events to interrupt the current task. These functions are described in *Chapter 11: Interrupts on page 145*.

1.14 Virtual memory

A set of functions to support virtual memory is supplied. These may be used to control memory access and create portable device drivers. See *Chapter 13: Virtual memory on page 173*.

1.15 Exceptions

An exception is an out of course (unexpected) event which causes the CPU to jump to an exception handling routine. Many exceptions are fatal, but other exceptions may require software intervention before the CPU can continue from the PC address where the exception was generated. Support is provided in OS21 to the user to deal with exceptions. These functions are described in *Chapter 14: Exceptions on page 180*.

1.16 Caches

A comprehensive set of cache handling functions is provided by OS21 to enable external events to gain control of the CPU. Prior to R.3.0.2 of OS21, these functions are described in the *OS21 implementation specific documentation*. For R.3.0.2 and following, these functions are described in this manual. See *Chapter 12: Caches and memory areas on page 163*.

1.17 Power management

OS21 defines a number of different power levels and a mechanism for transitioning between power levels, including waking up from standby mode. See *Chapter 16: Power management on page 192*.

1.18 Board support packages

Platform specific differences are held within Board Support Package (BSP) libraries. A board support library should be linked with the application and the OS21 library at final link time. The source for each BSP library shipped with OS21 is supplied as part of the product, enabling customers to modify them or create their own as necessary.

The source files are held in a subdirectory tree under the main OS21 directory:

`.../os21/src/`*`target_cpu`*`/bsp/*`

The `bsp` directory contains the source and makefiles required to build the BSP.

A BSP library provides target specific data and code. The precise nature of the data in the BSP is target specific. Every BSP exports two functions: `bsp_initialize()` and `bsp_start()`.`bsp_initialize()` is called by the OS21 kernel at initialization time, and provides a place for users to insert code which is executed just before the kernel comes up. `bsp_start()` is called when the OS21 kernel starts, and allows users a hook to perform any final initialization required by the target.

For details of the Board Support Package features that are common to all Board Support Packages, see *Chapter 17: Board support package on page 207*. For a description of the Board Support Packages (BSPs) for each supported OS21 target, see the *OS21 implementation specific documentation*.

2 Kernel

To implement multi-priority scheduling, OS21 uses a small scheduling kernel. This is a piece of code which makes scheduling decisions based on the priority of the tasks in the system. The kernel ensures that the current running task is always the one with the highest scheduling priority.

2.1 Kernel implementation

The kernel maintains two important pieces of information:

- the identity of the currently executing task (and therefore the priority currently being executed)
- a list of all the tasks which are currently ready to run

The kernel is invoked whenever a scheduling decision has to be made. This is on four possible occasions.

- When a task is about to be scheduled, the scheduler is called to determine if the new task is of higher priority than the currently executing task. If it is, the state of the current task is saved, and the new one is installed in its place, so the new task starts to run. This is termed **pre-emption** because the new task has preempted the old one.
- When a task deschedules for example, it waits on a message queue which does not have any messages available, then the scheduler is invoked to decide which task to run next. The kernel examines the list of tasks which are ready to run, and picks the one with the highest priority.
- Periodically the scheduler may be called to timeslice the currently executing task. If there are other tasks which are of the same priority as the current task, the state of the current task is saved onto the back of the current priority queue, and the task at the front of the queue is installed in its place. Therefore all tasks at the same priority have an opportunity to run.
- When an interrupt has been serviced, and there are no other lower-priority interrupts being serviced, the kernel is called to see if a reschedule is required. For example, an interrupt handler may have signalled a semaphore so that a higher priority task becomes ready to run when the interrupt handler completes.

This ensures that it is always the highest priority task running.

2.2 OS21 kernel

The only operation which can be performed on the OS21 kernel is its installation and start. This is done by calling the functions `kernel_initialize()` and `kernel_start()` which is usually performed as the first operation in `main()`:

```
if (kernel_initialize(&kernel_init_struct) != OS21_SUCCESS) {
  printf ("Error : initialise. kernel_initialize failed\n");
  exit (EXIT_FAILURE);
}

if (kernel_start() != OS21_SUCCESS) {
  printf("Error: initialize. kernel_start failed\n");
  exit(EXIT_FAILURE);
}
```

2.3 Kernel API summary

All the definitions related to the kernel can be obtained by including the header file `os21.h`, which itself includes the header file `kernel.h`. See *Table 2* for a complete list.

Table 2. Functions defined in kernel.h

Function	Description
`kernel_board()`	Returns the name of the board
`kernel_chip()`	Returns the name of the chip type
`kernel_cpu()`	Returns the name of the CPU type
`kernel_idle()`	Returns the amount of idle time on the CPU
`kernel_initialize()`	Initializer for preemptive scheduling
`kernel_printf()`	Outputs a string
`kernel_start()`	Starts preemptive scheduling regime
`kernel_time()`	Returns amount of kernel up-time
`kernel_timeslice()`	Turns timeslicing on and off
`kernel_version()`	Returns the OS21 version string

The `initialize` and `start` functions must be called only once from the main body of the application.

2.4 Kernel function definitions

kernel_board

Return the name of the board on which the application is running

Definition:
```
#include <os21.h>
const char* kernel_board(void);
```

Arguments: None

Returns: Returns a pointer to a string describing the board on which the application is running.

Errors: None

Context: Callable from task or system context.

Description: `kernel_board()` returns a pointer to a string which gives a readable description of the board on which the application is currently running.

See also: `kernel_version`, `kernel_cpu`

kernel_chip

Return the name of the chip type on which the application is running

Definition:
```
#include <os21.h>
const char* kernel_chip(void);
```

Arguments: None

Returns: Returns a pointer to a string describing the chip type on which the application is running.

Errors: None.

Context: Callable from task or system context.

Description: `kernel_chip()` returns a pointer to a string which gives a readable description of the chip type on which the application is currently running.

See also: `kernel_version`, `kernel_board`, `kernel_cpu`

kernel_cpu

Return the name of the CPU type on which the application is running

Definition:
```
#include <os21.h>
const char* kernel_cpu(void);
```

Arguments: None

Returns: Returns a pointer to a string describing the CPU type on which the application is running.

Errors: None.

Context: Callable from task or system context.

Description: `kernel_cpu()` returns a pointer to a string which gives a readable description of the CPU type on which the application is currently running.

See also: `kernel_version`, `kernel_board`, `kernel_chip`

kernel_idle

Return the kernel idle time

Definition:
```
#include <os21.h>
osclock_t kernel_idle(void);
```

Arguments: None

Returns: This function returns a time value indicating the kernel idle time.

Errors: None

Context: Callable from task or system context.

Description: `kernel_idle()` passes back a time value indicating the amount of time that the kernel has been idle; that is the time not executing tasks. Idle time occurs when there is no valid task to run, or interrupt pending.

The idle time is measured by recording the accumulation of intervals between the time when the kernel becomes idle and the time when it becomes active again.

See also: `kernel_time`

kernel_initialize

Initialize for preemptive scheduling

Definition:

```
#include <os21.h>
int kernel_initialize(
  kernel_initialize_t* init);
```

Arguments:

`kernel_initialize_t* init`	Address of kernel initialization structure, or `NULL`

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: Failure is caused by insufficient space to create the necessary data structures.

Context: Callable from task only.

Description: `kernel_initialize()` must be called before any tasks are created. It creates and initializes the task and queue data structures. After the structures are created the calling task is initialized as the root task in the system. This function should only be called once. Calling it multiple times has no effect. `kernel_initialize()` is also responsible for calling `bsp_initialize()` to initialize the BSP.

The `kernel_initialize_t` structure passed to this call provides OS21 with its required memory regions. If the `system_heap_base` is `NULL`, and the `system_heap_size` is 0, OS21 uses the usual 'C' runtime heap as its system heap (managed by C runtime library routines such as `malloc()`, `free()`). If a system heap memory region is provided with this structure, OS21 takes over management of it directly.

If the `system_stack_base` is `NULL`, and the `system_stack_size` is 0, OS21 allocates its system stack from the system heap. The size of the system stack in this case is a platform-specific default value.

If `NULL` is specified for either base pointer, with an associated non-zero size, OS21 allocates the required memory from the 'C' runtime heap.

If the `init` parameter is passed as a `NULL` pointer, OS21 uses the 'C' runtime heap for its system heap, and allocates a default sized system stack.

The definition of a `kernel_initialize_t` is:

```
typedef struct {
  unsigned char* system_stack_base;
  size_t system_stack_size;
  unsigned char* system_heap_base,
  size_t system_heap_size,
} kernel_initialize_t;
```

See also: `kernel_start`

kernel_printf

Output a string

Definition:
```
#include <os21.h>
void kernel_printf(const char * fmt, ...);
```

Arguments:

`fmt`	The string to output.

Returns: None.

Errors: None.

Context: Callable from task or system context.

Description: `kernel_printf()` outputs a string in a similar manner to the C run-time function `printf()`. `kernel_printf()` can be called from any context and is guaranteed not to block. It is not guaranteed that the message will be output. For example, if some I/O resource is busy at the point at which the call to `kernel_printf()` is made, the message may not be output.

`kernel_printf()` supports some, but not all, of the familiar `printf()` formats (`%d`, `%u`,`%p`, `%x`, `%c`, `%s` and `%%` are supported). Field width specifiers are not supported.

kernel_start

Starts preemptive scheduling regime

Definition:
```
#include <os21.h>
int kernel_start(void);
```

Arguments: None

Returns: `OS21_SUCCESS` or `OS21_FAILURE`

Errors: Failure is caused by insufficient memory, or `kernel_initialize()` not having been called previously.

Context: Callable from task only.

Description: `kernel_start()` must be called before any tasks are created. On return from the function the preemptive scheduler is running. The calling function is installed as the first OS21 task and is running at `MAX_USER_PRIORITY`.

`kernel_start()` is also responsible for calling `bsp_start()` in the BSP, to allow any BSP specific start actions to be performed.

Note: Before calling this function, `kernel_initialize()` *must have been called.* `kernel_start()` *should only be called once.*

kernel_time

Return the kernel up-time

Definition:
```
#include <os21.h>
osclock_t kernel_time(void);
```

Arguments: None

Returns: A clock value indicating how long has elapsed since the kernel started executing.

Errors: None

Context: Callable from task or system context.

Description: `kernel_time()` returns the kernel up-time, indicating the elapsed time that the kernel has been running; that is, the time spent executing code or in idle state.

The kernel up-time is the time from when the kernel was successfully started to the time when the `kernel_time()` call is made.

See also: `kernel_idle`

kernel_timeslice

Turn on or off timeslicing

Definition:
```
#include <os21.h>
void kernel_timeslice(
  int on);
```

Arguments:

`int on`	`OS21_TRUE` to turn on timeslicing, `OS21_FALSE` to turn off timeslicing

Returns: None

Errors: None

Context: Callable from task or system context.

Description: `kernel_timeslice()` can be called after `kernel_start()` has been called and turns timeslicing on or off. If an application has distinct priorities for each task, there is no requirement for timeslicing. An application that runs without timeslicing, spends less time executing OS21 kernel code and is therefore more efficient than one that uses timeslicing.

See also: `kernel_start`

kernel_version

Return the OS21 version number

Definition:
```
#include <os21.h>
const char* kernel_version(void);
```

Arguments: None

Returns: Returns a pointer to the OS21 version string.

Errors: None

Context: Callable from task or system context.

Description: `kernel_version()` returns a pointer to a string which gives the OS21 version number. This string takes the form: *`{major number}.{minor number}.{patch number} [text]`*. That is, a major, minor and release number, separated by decimal points, and optionally followed by a space and a printable text string.

See also: `kernel_initialize`

3 Memory and partitions

Memory management on many embedded systems is vital, because available memory is often small and must be used efficiently. Therefore three different styles of memory management have been provided with OS21, with the ability for users to define their own memory managers; see *Section 3.2: Allocation strategies on page 26*. These give the user flexibility in controlling how memory is allocated, allowing a space/time trade-off to be performed.

3.1 Partitions

The job of memory management is to allow the application program to allocate and free blocks of memory from a larger block of memory, which is under the control of a memory allocator. In OS21 these concepts have been combined into a **partition**, which has three properties:

- the block of memory for which the partition is responsible
- the current state of allocated and free memory
- the algorithm to use when allocating and freeing memory

The method of allocating/deallocating memory is the same whatever style of partition is used, only the algorithm used (and therefore the interpretation of the partition data structures) changes.

There is nothing special about the memory which a partition manages. It can be a static or local array, or an absolute address which is known to be free. It can also be a block allocated from another partition, see the example given in the description of *partition_delete on page 41*. This is useful by avoiding having to explicitly free all the blocks allocated:

- allocate a block from a partition, and create a second partition to manage it
- allocate memory from the partition as normal
- when finished, rather than freeing all the allocated blocks individually, free the whole partition (as a block) back to the partition from which it was first allocated

The OS21 system of partitions can also be exploited to build fault-tolerance into an application. This is done by implementing different parts of the application, using different memory partitions. Therefore, if a fault occurs in one part of the application it does not necessarily effect the whole application.

3.2 Allocation strategies

Three types of partition are directly supported in OS21:

Heap

Heap partitions use the same style of memory allocator as the traditional C runtime `malloc` and `free` functions. Variable sized blocks can be allocated, with the requested size of memory being allocated by `memory_allocate`, and the first available block of memory is returned to the user. Blocks of memory may be deallocated using `memory_deallocate`, in which case they are returned to the partition for re-use. When blocks are freed, if there is a free block before or after it, it is combined with that block to allow larger allocations.

Although the heap style of allocator is very versatile, it does have some disadvantages. It is not deterministic, the time taken to allocate and free memory is variable because it depends on the previous allocations/deallocations performed and lists have to be searched. Also, the overhead (additional memory which the allocator consumes for its own use) is quite high, with several additional words being required for each allocation.

Fixed

The **fixed** partition overcomes some of these problems, by fixing the size of the block which can be allocated when the partition is created, using `partition_create_fixed` or `partition_create_fixed_p`. This means that allocating and freeing a block takes constant time (that is, it is deterministic) and there is a very small memory overhead. Therefore this partition ignores the *`size`* argument when an allocation is preformed by `memory_allocate` and uses instead the *`size`* argument which was specified when the partition was created using either `partition_create_fixed` or `partition_create_fixed_p`.

Blocks of memory may be deallocated using `memory_deallocate`, in which case they are returned to the partition for re-use.

Simple

The **simple** partition is a trivial allocator, which just increments a pointer to the next available block of memory. This means that it is impossible to free any memory back to the partition, but there is no wasted memory when performing memory allocations. Therefore this partition is ideal for allocating internal memory. Variable sized blocks of memory can be allocated, with the size of block being defined by the argument to `memory_allocate` and the time taken to allocate memory is constant.

The properties of the three partition types are summarized in *Table 3*.

Table 3. Partition properties

Properties	Heap	Fixed	Simple
Allocation method	As requested by `memory_allocate` or `memory_reallocate`	Fixed at creation by `partition_create_fixed` or `partition_create_fixed_p`	As requested by `memory_allocate` or `memory_reallocate`
Deallocation possible	Yes	Yes	No
Deterministic	No	Yes	Yes

OS21 also allows the user to create new partition types which can implement any allocation scheme. This is supported by the `partition_create_any()` API, which allows the user to register a new type of partition manager.

3.3 Predefined partitions

Unlike OS20, OS21 does not have any predefined partitions. The system heap can be managed by the traditional C runtime routines (such as `malloc()`), or by OS21. The system heap is used by all `_create()` calls to allocate control structures.

3.4 Obtaining information about partitions

When memory is dynamically allocated it is important to have knowledge of how much memory is used or how much memory is available in a partition. The status of a partition can be retrieved with a call to the following function:

```
#include <os21.h>
int partition_status(
  partition_t* partition,
  partition_status_t* status,
  partition_status_flags_t flags);
```

The information returned includes the total memory used, the total amount of free memory, the largest block of free memory and whether the partition is in a valid state.

`partition_status()` returns the status of **heap**, **fixed** and **simple** partitions by storing the status into the `partition_status_t` structure which is passed as a pointer to `partition_status()`.

For **fixed** partitions the largest free block of memory is always the same as the block size of the requested partition.

3.5 Creating a new partition type

OS21 allows a user to create partitions with their own allocation strategies. The user supplies `allocate`, `deallocate`, `reallocate` and `status` functions to `partition_create_any()` or `partition_create_any_p()`, with the required amount of extra storage they need for their control structure (private state). These calls create a `partition_t` structure that incorporates enough room for their private control structure. This is allocated either from the system heap, or from the nominated partition if `partition_create_any_p()` is used.

The user then calls `partition_private_state()` passing in the returned `partition_t` pointer, to get a pointer to their private state, so it can be initialized.

Any memory requests involving this partition are vectored to the user supplied routines.

3.6 Traditional ‘C’ memory management

The traditional ‘C’ heap management routines (such as `malloc()`, `realloc()` and `free()`) are all still available from **newlib**. The calls are task aware, and can be used to manage the ‘C’ runtime heap. `malloc()` can be used to allocate chunks of memory from the ‘C’ runtime heap, then pass this memory to OS21 (using the `kernel_initialize()` call), or to a partition manager for it to manage (using a `partition_create_*()` call).

3.7 Partition API summary

All the definitions related to memory partitions can be obtained by including the header file, `os21.h`, which itself includes the header file `partition.h`. See *Table 4* and *Table 5.* for a complete list.

Table 4. Functions defined in partition.h

Function	Description
`memory_allocate()`	Allocates a block of memory from a partition
`memory_allocate_clear()`	Allocates a block of memory from a partition and clear to zero
`memory_deallocate()`	Frees a block of memory back to a partition
`memory_reallocate()`	Reallocates a block of memory from a partition
`partition_create_any()`	Creates a user partition
`partition_create_any_p()`	Creates a user partition
`partition_create_fixed()`	Creates a fix partition
`partition_create_fixed_p()`	Creates a fix partition
`partition_create_heap()`	Creates a heap partition
`partition_create_heap_p()`	Creates a heap partition
`partition_create_simple()`	Creates a simple partition
`partition_create_simple_p()`	Creates a simple partition
`partition_delete()`	Deletes a partition
`partition_private_state()`	Returns a user partition’s private state pointer
`partition_status()`	Gets the status of a partition

All functions are callable from an OS21 task.

Table 5. Types defined by partition.h

Type	Description
`memory_allocate_fn`	Memory allocator function
`memory_deallocate_fn`	Memory deallocator function
`memory_reallocate_fn`	Memory reallocator function
`memory_status_fn`	Memory status function

Table 5. Types defined by partition.h (continued)

Type	Description
`partition_t`	A memory partition
`partition_status_flags_t`	Additional flags for `partition_status`

3.8 Memory and partition function definitions

memory_allocate

Allocate a block of memory from a partition

Definition:

```
#include <os21.h>
void* memory_allocate(
  partition_t *part,
  size_t size);
```

Arguments:

`partition_t *part`	The partition from which to allocate memory
`size_t size`	The number of bytes to allocate

Returns: A pointer to the allocated memory, or `NULL` if there is insufficient memory available.

Errors: If there is insufficient memory for the allocation, it fails and returns `NULL`.

Context: Callable from task only.

Description: `memory_allocate()` allocates a block of memory of size bytes from partition `part`. It returns the address of a block of memory of the required size, which is suitably aligned to contain any type.

Note: If a null pointer is specified for `part`, instead of a valid partition pointer, the C runtime heap is used.

This function calls the memory allocator associated with the partition `part`. For a full description of the algorithm, see the description of the appropriate partition creation function.

See also: `memory_deallocate`, `memory_reallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_fixed`, `partition_create_fixed_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_simple`, `partition_create_simple_p`

memory_allocate_clear

Allocate and zero a block of memory from a partition

Definition:

```
#include <os21.h>
void* memory_allocate_clear(
  partition_t *part,
  size_t nelem,
  size_t elsize);
```

Arguments:

`partition_t *part`	The partition from which to allocate memory
`size_t nelem`	The number of elements to allocate
`size_t elsize`	The size of each element in bytes

Returns: A pointer to the allocated memory, or `NULL` if there is insufficient memory available.

Errors: If there is insufficient memory for the allocation, it fails and returns `NULL`.

Context: Callable from task only.

Description: `memory_allocate_clear()` allocates a block of memory large enough for an array of `nelem` elements, each of size `elsize` bytes, from partition `part`. It returns the base address of the array, which is suitably aligned to contain any type. The memory is initialized to zero.

Note: If a null pointer is specified for `part`*, instead of a valid partition pointer, the C runtime heap is used.*

This function calls the memory allocator associated with the partition `part`. For a full description of the algorithm, see the description of the appropriate partition creation function.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

memory_deallocate

Free a block of memory back to a partition

Definition:

```
#include <os21.h>
void memory_deallocate(
  partition_t *part,
  void* block);
```

Arguments:

`partition_t *part`	The partition to which memory is freed
`void* block`	The block of memory to free

Returns: None

Errors: None

Context: Callable from task only.

Description: `memory_deallocate()` returns a block of memory at `block`, back to partition `part`. The memory must have been originally allocated from the same partition to which it is being freed.

Note: If a null pointer is specified for `part`*, instead of a valid partition pointer, the C runtime heap is used.*

This function calls the memory allocator associated with the partition `part`. For a full description of the algorithm, see the description of the appropriate partition creation function.

See also: `memory_allocate`, `memory_reallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

memory_reallocate

Reallocate a block of memory from a partition

Definition:

```
#include <os21.h>
void* memory_reallocate(
  partition_t *part,
  void* block,
  size_t size);
```

Arguments:

`partition_t *part`	The partition to reallocate
`void* block`	The current memory block
`size_t size`	The number of bytes to allocate

Returns: A pointer to the allocated memory, or `NULL` if there is insufficient memory available.

Errors: If there is insufficient memory for the allocation, it fails and returns `NULL`.

Context: Callable from task only.

Description: `memory_reallocate()` changes the size of a memory block allocated from a partition, preserving the current contents.

If `block` is `NULL`, the function behaves like `memory_allocate` and allocates a block of memory. If `size` is 0 and `block` is not `NULL`, the function behaves like `memory_deallocate()` and frees the block of memory back to the partition.

For fixed and heap partitions, if `block` is not `NULL` and `size` is not 0, the block of memory is reallocated.

Note: `block` *must have been allocated from* `part` *originally.*

Note: If a null pointer is specified for `part`*, instead of a valid partition pointer, the C runtime heap is used.*

`memory_reallocate()` calls the memory allocator associated with the partition `part`. For a full description of the algorithm, see the description of the appropriate partition initialization function.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

partition_create_any

Create a user defined partition

Definition:

```
#include <os21.h>
partition_t* partition_create_any(
  size_t private_state_size,
  mem_allocate_fn allocate,
  mem_deallocate_fn deallocate,
  mem_reallocate_fn reallocate,
  mem_status_fn status);
```

Arguments:

`size_t private_state_size`	Amount of state this allocator requires
`mem_allocate_fn allocate`	Memory allocation routine
`mem_deallocate_fn deallocate`	Memory deallocate routine
`mem_reallocate_fn reallocate`	Memory reallocate routine
`mem_status_fn status`	Memory status routine

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If there is insufficient memory to allocate the control structure the routine returns `NULL`.

Context: Callable from task only.

Description: `partition_create_any()` creates a memory partition where the allocation strategy is user defined. The `partition_t` describing this partition is allocated from the system heap.

Memory is allocated and freed back to this partition using `memory_allocate()`, `memory_deallocate()` and `memory_reallocate()`, which are vectored to the user supplied routines.

The prototypes for the management routines are:

```
typedef void* (*memory_allocate_fn) (
  void* state,
  size_t size);
typedef void (*memory_deallocate_fn) (
  void * state,
  void * ptr);
typedef void* (*memory_reallocate_fn) (
  void* state,
  void* ptr,
  size_t size);
typedef int (*mem_status_fn) (
  void* state,
  partition_status_t *status,
  partition_status_flags_t flags);
```

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `patition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`, `partition_private_state`

partition_create_any_p

Create a user defined partition

Definition:

```
#include <os21.h>
partition_t* partition_create_any_p(
  partition_t* partition,
  size_t private_state_size,
  mem_allocate_fn allocate,
  mem_deallocate_fn deallocate,
  mem_reallocate_fn reallocate,
  mem_status_fn status);
```

Arguments:

`partition_t* partition`	Partition from which to allocate `partition_t`
`size_t private_state_size`	Amount of state this allocator requires
`mem_allocate_fn allocate`	Memory allocation routine
`mem_deallocate_fn deallocate`	Memory deallocate routine
`mem_reallocate_fn reallocate`	Memory reallocate routine
`mem_status_fn status`	Memory status routine

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If there is insufficient memory to allocate the control structure the routine returns `NULL`.

Context: Callable from task only.

Description: `partition_create_any_p()` creates a memory partition where the allocation strategy is user defined. The `partition_t` describing this partition is allocated from the specified `partition`.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

Memory is allocated and freed back to this partition using `memory_allocate()`, `memory_deallocate()` and `memory_reallocate()`, which are vectored to the user supplied routines.

The prototypes for the management routines are:

```
typedef void* (*memory_allocate_fn) (
  void* state,
  size_t size);
typedef void (*memory_deallocate_fn) (
  void * state,
  void * ptr);
typedef void* (*memory_reallocate_fn) (
  void* state,
  void* ptr,
  size_t size);
```

```
typedef int (*mem_status_fn) (
  void* state,
  partition_status_t *status,
  partition_status_flags_t flags);
```

See also: memory_allocate, memory_deallocate, partition_create_any, partition_create_simple, partition_create_simple_p, patition_create_heap, partition_create_heap_p, partition_create_fixed, partition_create_fixed_p, partition_private_state

partition_create_fixed

Create a fixed size partition

Definition:

```
#include <os21.h>
partition_t* partition_create_fixed(
  void* memory,
  size_t memory_size,
  size_t block_size);
```

Arguments:

`void* memory`	The start address for the memory partition
`size_t memory_size`	The size of the memory block in bytes
`size_t block_size`	The size of the block to allocate from the partition

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If the amount of memory is insufficient it fails and returns `NULL`.

Context: Callable from task only.

Description: `partition_create_fixed()` creates a memory partition where the size of the blocks which can be allocated is fixed when the partition is created. Only the amount of memory requested is allocated, with no overhead for the partition manager. Allocating and freeing simply involves removing and adding blocks to a linked list, and so takes constant time.

Memory is allocated and freed back to this partition using `memory_allocate()` and `memory_deallocate()`. `memory_allocate()` must specify the same or smaller block size as was used when the partition was created, otherwise the allocation fails. `memory_reallocate()` has no effect.

See also: memory_allocate, memory_deallocate, partition_create_any, partition_create_any_p, partition_create_simple, partition_create_simple_p, partition_create_heap, partition_create_heap_p, partition_create_fixed_p

partition_create_fixed_p

Create a fixed size partition

Definition:

```
#include <os21.h>
partition_t* partition_create_fixed_p(
  partition_t* partition,
  void* memory,
  size_t memory_size,
  size_t block_size);
```

Arguments:

`partition_t* partition`	Partition from which to allocate `partition_t`
`void* memory`	The start address for the memory partition
`size_t memory_size`	The size of the memory block in bytes
`size_t block_size`	The size of the block to allocate from the partition

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If the amount of memory is insufficient it fails and returns `NULL`.

Context: Callable from task only.

Description: `partition_create_fixed_p()` creates a memory partition where the size of the blocks which can be allocated is fixed when the partition is created. Only the amount of memory requested is allocated, with no overhead for the partition manager. Allocating and freeing simply involves removing and adding blocks to a linked list, and so takes constant time. The `partition_t` is allocated from the specified partition.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

Memory is allocated and freed back to this partition using `memory_allocate()` and `memory_deallocate()`. `memory_allocate()` must specify the same or smaller block size as was used when the partition was created, otherwise the allocation fails. `memory_reallocate()` has no effect.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`

partition_create_heap

Create a heap partition

Definition:

```
#include <os21.h>
partition_t* partition_create_heap(
  void* memory,
  size_t size);
```

Arguments:

`void* memory`	The start address for the memory partition
`size_t size`	The size of the memory block in bytes

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If the amount of memory is insufficient it fails and returns `NULL`.

Context: Callable from task only.

Description: `partition_create_heap()` creates a memory partition with the semantics of a heap. This means that variable size blocks of memory can be allocated and freed back to the memory partition. Only the amount of memory requested is allocated, with a small overhead on each block for the partition manager. Allocating and freeing requires searching through lists, and so the length of time depends on the current state of the heap.

Memory is allocated and freed back to this partition using `memory_allocate()` and `memory_deallocate()`. `memory_reallocate()` is implemented efficiently. Reducing the size of a block is always done without copying, and expanding only results in a copy if the block cannot be expanded because subsequent memory locations have been allocated.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

partition_create_heap_p

Create a heap partition

Definition:

```
#include <os21.h>
partition_t* partition_create_heap_p(
  partition_t* partition,
  void* memory,
  size_t size);
```

Arguments:

`partition_t* partition`	Partition from which to allocate control structure
`void* memory`	The start address for the memory partition
`size_t size`	The size of the memory block in bytes

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If the amount of memory is insufficient it fails and returns `NULL`.

Context: Callable from task only.

Description: `partition_create_heap_p()` creates a memory partition with the semantics of a heap. This means that variable size blocks of memory can be allocated and freed back to the memory partition. Only the amount of memory requested is allocated, with a small overhead on each block for the partition manager. Allocating and freeing requires searching through lists, and so the length of time depends on the current state of the heap. The new `partition_t` structure is allocated from the specified existing partition.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

Memory is allocated and freed back to this partition using `memory_allocate()` and `memory_deallocate()`. `memory_reallocate()` is implemented efficiently. Reducing the size of a block is always done without copying, and expanding only results in a copy if the block cannot be expanded because subsequent memory locations have been allocated.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_fixed`, `partition_create_fixed_p`

partition_create_simple

Create a simple partition

Definition:

```
#include <os21.h>
partition_t* partition_create_simple(
  void* memory,
  size_t size);
```

Arguments:

`void* memory`	The start address for the memory partition
`size_t size`	The size of the memory block in bytes

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If the amount of memory is insufficient it fails and returns `NULL`.

Context: Callable from task only.

Description: `partition_create_simple()` creates a memory partition with allocation only semantics. This means that memory can only be allocated from the partition, attempting to free it back has no effect. Only the amount of memory requested is allocated, with no overhead. Allocation involves checking if there is space left in the partition, and incrementing a pointer, so is very efficient and takes constant time.

Memory is allocated from this partition using `memory_allocate()`. Calling `memory_deallocate()` on this partition has no effect. As there is no record of the original allocation size, `memory_reallocate()` cannot know whether the block is growing or shrinking, and so always returns `NULL`.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

partition_create_simple_p

Create a simple partition

Definition:

```
#include <os21.h>
partition_t* partition_create_simple_p(
  partition_t* partition,
  void* memory,
  size_t size);
```

Arguments:

`partition_t* partition`	Partition from which to allocate control structure
`void* memory`	The start address for the memory partition
`size_t size`	The size of the memory block in bytes

Returns: The partition identifier or `NULL` if an error occurs.

Errors: If the amount of memory is insufficient it fails and returns `NULL`.

Context: Callable from task only.

Description: `partition_create_simple_p()` creates a memory partition with allocation only semantics. This means that memory can only be allocated from the partition, attempting to free it back has no effect. Only the amount of memory requested is allocated, with no overhead. Allocation involves checking if there is space left in the partition, and incrementing a pointer, so is very efficient and takes constant time. The new `partition_t` structure is allocated from the specified existing partition.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

Memory is allocated from this partition using `memory_allocate()`. Calling `memory_deallocate()` on this partition has no effect. As there is no record of the original allocation size, `memory_reallocate()` cannot know whether the block is growing or shrinking, and so always returns `NULL`.

See also: `memory_allocate`, `memory_deallocate`, `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

partition_delete

Delete a partition

Definition:

```
#include <os21.h>
void partition_delete(
  partition_t* partition );
```

Arguments:

`partition_t* partition`	Partition to delete

Returns: None

Errors: None

Context: Callable from task only.

Description: This function allows a partition to be deleted. It frees the data structure used to manage the partition (`partition_t`).

Deleting the memory that forms the partition is the responsibility of the user. The block of memory being managed by the partition is unaffected by `partition_delete()`.

See also: `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

partition_private_state

Return the address of a partition's private state structure

Definition:

```
#include <os21.h>
void * partition_private_state(
  partition_t* partition );
```

Arguments:

`partition_t* partition`	Partition to query

Returns: The address of the partition's private state information, or `NULL`.

Errors: Returns `NULL` if a null partition is specified.

Context: Callable from task only.

Description: This function allows the address of a partition's private state data to be returned. This is required when implementing a new partition management scheme.

See also: `partition_create_any`, `partition_create_any_p`

partition_status

Get status of a partition

Definition:

```
#include <os21.h>
int partition_status(
  partition_t* partition,
  partition_status_t* status,
  partition_status_flags_t flags);
```

Arguments:

`partition_t* partition`	A pointer to a partition
`partition_status_t* status`	A pointer to a buffer to save to
`partition_status_flags_t flags`	Reserved for future use, `flags` should be set to zero

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: Returns `OS21_FAILURE` if `status` is `NULL`, or if `partition` has not been initialized using one of the `_create or _create_p` functions. Partitions previously deleted with `partition_delete()` also return `OS21_FAILURE`.

Context: Callable from task only.

Description: `partition_status()` checks the status of the partition by checking that the partition is not corrupt and also by calculating the memory usage of the partition. Memory usage includes the amount of memory used, memory available and largest available block of memory.

`partition` is a pointer to a partition which `partition_status()` references to calculate memory usage. `status` is a pointer to a structure which `partition_status()` uses to store the results.

Note: If `NULL` is passed as the partition pointer, the status is filled in as best it can be by interrogating the C runtime heap manager.

Table 6 shows the layout of the structure `partition_status_t`.

Table 6. Layout of structure partition_status_t

Name	Description
`partition_status_state`	Partition state (See *Table 7*)
`partition_status_type`	Type of partition (See *Table 8*)
`partition_status_size`	Total number of bytes within partition
`partition_status_free`	Total number of bytes free within partition
`partition_status_free_largest`	Total number of bytes within the largest free block in partition
`partition_status_used`	Total number of bytes which are allocated/in use within the partition

Table 7 shows all the possible values which are available to the field `partition_status_state`.

Table 7. Flag values for partition_status_state

Flag	Flag description
`partition_status_state_valid`	Partition is valid
`partition_status_state_invalid`	Partition is corrupt

Table 8 shows all the possible values which are available to the field `partition_status_type`.

Table 8. Values for partition_status_type

Flag	Flag description
`partition_status_type_simple`	Partition is a **simple** partition
`partition_status_type_fixed`	Partition is a **fixed** partition
`partition_status_type_heap`	Partition is a **heap** partition
`partition_status_type_any`	Partition has user defined semantics

If `partition_status()` returns successfully then the structure pointed to by `status` contains statistics about the partition `partition`.

`partition_status_state` is set to `partition_status_state_valid` if the partition is valid. Otherwise it is set to `partition_status_state_invalid`.

`partition_status_type` depending on the type of partition contains one of the flags as shown in *Table 8*.

`partition_status_size` contains the size of the partition in bytes. The size of a partition is defined when a partition is initialized using the `_create/_create_p` functions, therefore `partition_status_size` does not change with subsequent calls to `partition_status()`.

`partition_status_used` is the total number of bytes allocated in the partition.

`partition_status_free` is the number of free bytes available in the partition.

`partition_status_free_largest` is the size of the largest free block of memory in the partition.

`partition_status_used` is the total number of bytes used in the partition.

The results provided by `partition_status()` may differ slightly for each partition type, for example, **heap** and **fixed** partitions incur a memory overhead with each allocation/deallocation, these overheads are taken into account in the results. See *Table 3: Partition properties on page 26*.

See also: `partition_create_any`, `partition_create_any_p`, `partition_create_simple`, `partition_create_simple_p`, `partition_create_heap`, `partition_create_heap_p`, `partition_create_fixed`, `partition_create_fixed_p`

4 Tasks

Tasks are separate threads of control, which run independently. A task describes the behavior of a discrete, separable component of an application, behaving like a separate program, except that it can communicate with other tasks. New tasks may be generated dynamically by any existing task.

Applications can be broken into any number of tasks provided there is sufficient memory. When a program starts, there is a single main task in execution. Other tasks can be started as the program executes. These other tasks can be considered to execute independently of the main task, but share the processing capacity of the processor.

4.1 OS21 tasks

A task consists of a data structure, stack and a section of code. A task's data structure is known as its **state** and its exact content and structure are processor dependent. This structure is known as a `task_t` structure. It includes the task state (being created, executing, terminated) and the stack range (used for stack checking).

A task is identified by its `task_t` structure and this should always be used when referring to the task. A pointer to the `task_t` structure is called the task's ID, see *Section 4.11 on page 49*.

The code for the task to execute is provided by the user function. To create a task, the `task_t` data structure must be allocated and initialized and a stack and function must be associated with them. This is done using the `task_create()` or `task_create_p()` functions depending on whether the user wishes to control the allocation of the data structures or not. See *Section 4.4: Creating and running a task on page 46*.

Note: OS21 does not provide the equivalent of OS20 high priority hardware processes. Only the tasking model of OS20 is preserved in OS21.

4.2 OS21 priorities

The number of OS21 task priorities and the highest and lowest task priorities are defined using the macros in the header file `os21/task.h`, see *Section 4.19: Task API summary on page 55*. Numerically higher priorities preempt lower priorities for example, 3 is a higher priority than 2.

A task's initial priority is defined when it is created, see *task_create on page 58*. The only task which does not have its priority defined in this way is the root task, that is, the task which starts OS21 running by calling `kernel_start()`. This task starts running with the highest priority available, `MAX_USER_PRIORITY`.

If a task needs to know the priority it is running at or the priority of another task, it can call the following function:

```
int task_priority (task_t* task)
```

`task_priority()` retrieves the OS21 priority of the task specified by `task` or the priority of the currently active task if `task` is `NULL`.

The priority of a task can be changed using the `task_priority_set()` function:

```
int task_priority_set (
   task_t* task,
   int priority);
```

`task_priority_set()` sets the priority of the task specified by `task`, or of the currently active task if `task` is `NULL`. If this results in the current task's priority falling below that of another task which is ready to run, or a ready task now has a priority higher than the current task's, tasks may be rescheduled.

4.3 Scheduling

An active task may either be running or waiting to run. OS21 ensures the following.

- The currently executing task is always the one with the highest priority.
  If a task with a higher priority becomes ready to run, the OS21 scheduler saves the current task's state and makes the higher priority task the current task. The current task runs to completion unless it is preempted by a higher priority task, and so on. Once a task has completed, the next highest priority task starts executing.
- If timeslicing has been enabled, tasks of equal priority are timesliced to ensure they all get a chance to run. Each task of the same priority level executes in turn for a period of time known as a **timeslice**.

The kernel scheduler can be prevented from preempting or timeslicing the current task, by using the following pair of functions:

```
void task_lock (void);
void task_unlock (void);
```

These functions should always be called as a pair and can be used to create a critical region where one task is prevented from preempting another. Calls to `task_lock()` can be nested, and the lock is not released until an equal number of calls to `task_unlock()` have been made. Once `task_unlock()` is called, the scheduler starts running the highest **priority task available. T**his may not be the task which called `task_unlock()`.

If a task voluntarily deschedules, for example by calling `semaphore_wait()`, the critical region is unlocked and normal scheduling resumes. When the task resumes (if for example it acquired the semaphore), the critical region is reinstated by OS21.

When this lock is in place the task can still be interrupted by interrupt handlers. Interrupts can be prevented from interrupting the task by using the `interrupt_mask()` or `interrupt_mask_all()` functions. Any task that is made runnable as a result cannot pre-empt the locked task, even if it is of higher priority. The pre-emption only occurs when the `task_unlock()` call is made.

4.4 Creating and running a task

The following functions are provided for creating and starting a task running:

```
#include <os21.h>
task_t* task_create (
  void (*function)(void*),
  void* param,
  int stack_size,
  int priority,
  const char* name,
  task_flags_t flags);
```

```
#include <os21.h>
task_t* task_create_p(
  partition_t* partition,
  void (*function)(void*),
  void* param,
  partition_t* stack_partition,
  int stack_size,
  int priority,
  const char* name,
  task_flags_t flags);
```

Both functions set up a task and start the task running at the specified function. This is done by initializing the data structure `task_t` and associating a function with it.

Using either `task_create()` or `task_create_p()`, the function is passed in as a pointer to the task's entry point. Both functions take a single pointer to be used as the argument to the user function. A cast to `void*` should be performed to pass in a single word sized parameter (for example, an `int`). Otherwise a data structure should be set up.

The functions differ in how the task's data structure is allocated. `task_create()` allocates memory for the task's stack, control block `task_t` from the system heap, whereas `task_create_p()` enables the user to specify a specific memory partition from which to allocate.

`task_create()` and `task_create_p()` both require the stack size to be specified. Stack is used for a function's local variables and parameters and to save the register context when the task is preempted.

Both functions require an OS21 priority level to be specified for the task and a name to be associated with the task for use by the debugger. The priority levels are defined in the header file `os21/task.h` by the macros `OS21_PRIORITY_LEVELS`, `MAX_USER_PRIORITY` and `MIN_USER_PRIORITY`, see *Section 4.19: Task API summary on page 55*.

4.5 Synchronizing tasks

Tasks synchronize their actions with each other using semaphores and mutexes, as described in *Chapter 6: Semaphores on page 100* and *Chapter 7: Mutexes on page 110*.

4.6 Communicating between tasks

Tasks communicate with each other using message queues, as described in *Chapter 9: Message handling on page 129*.

4.7 Timed delays

The following two functions cause a task to wait for a certain length of time measured in ticks of the timer.

```
void task_delay(osclock_t delay);
void task_delay_until(osclock_t delay);
```

Both functions wait for a period of time and then return. `task_delay_until()` waits until the given absolute reading of the timer is reached. If the requested time is before the present time, the task does not wait.

`task_delay()` waits until the given time has elapsed, that is, it delays execution for the specified number of timer ticks. If the time given is negative, no delay takes place.

`task_delay()` or `task_delay_until()` may be used for data logging or causing an event at a specific time. A high priority task can wait until a certain time; when it wakes it preempts any lower priority task that is running and performs the time-critical function.

When initiating regular events, such as for data logging, it may be important not to accumulate errors in the time between ticks. This is done by repeatedly adding to a time variable rather than rereading the start time for the delay.

For example, to initiate a regular event every `delay` ticks:

```
#include <os21.h>
osclock_t time;
time = time_now();
for (;;)
{
   time = time_plus (time, delay);
   task_delay_until(time);
   initiate_regular_event ();
}
```

4.8 Rescheduling

Sometimes, a task needs to voluntarily give up control of the CPU so that another task at the same priority can execute, that is, terminate the current timeslice. This is achieved with the functions:

```
void task_reschedule (void);
void task_yield (void);
```

These provide a clean way of suspending execution of a task in favor of the next task on the scheduling list, but without losing priority. The task which executes `task_reschedule()` or `task_yield()` is added to the back of the scheduling list and the task at the front of the scheduling list is promoted to be the new current task.

The only difference between `task_reschedule()` and `task_yield()` is that `task_reschedule()` has no effect if a `task_lock()` is in effect, where as `task_yield()` will always yield the CPU.

A task may be inadvertently rescheduled when the `task_priority_set()` function is used, see *task_priority_set on page 73*.

4.9 Suspending tasks

Normally a task only deschedules when it is waiting for an event, such as for a semaphore to be signalled. This requires that the task calls a function indicating that it is willing to deschedule at that point (for example, by calling `semaphore_wait()`). However, sometimes it is useful to be able to control a task, causing it to forcibly deschedule, without it explicitly indicating that it is willing to be descheduled. This can be done by **suspending** the task.

When a task is suspended, it stops executing immediately. When the task should start executing again, another task must **resume** it. When it is resumed the task is unaware it has been suspended, other than the time delay.

Task suspension is in addition to any other reason that a task is descheduled. Therefore a task which is waiting on a semaphore, and which is then suspended, does not start executing again until both the task is resumed, and the semaphore is signalled, although these can occur in any order.

A task is suspended using the call:

```
int task_suspend(task_t* task)
```

where `task` is the task to be suspended. A task may suspend itself by specifying `task` as `NULL`. The result is `OS21_SUCCESS` if the task was successfully suspended, `OS21_FAILURE` if it failed. This call fails if the task has terminated. A task may be suspended multiple times by executing several calls to `task_suspend()`. It does not start executing again until an equal number of `task_resume()` calls have been made.

A task is resumed using the call:

```
int task_resume(task_t* task)
```

where `task` is the task to be resumed. The result is `OS21_SUCCESS` if the task was successfully resumed, `OS21_FAILURE` if it failed. The call fails if the task has terminated, or is not suspended.

It is also possible to specify that when a task is created, it should be immediately suspended, before it starts executing. This is done by specifying the flag `task_flags_suspended` when calling `task_create()` or `task_create_p()`. This can be useful to ensure that initialization is carried out before the task starts running. The task is resumed in the usual way, by calling `task_resume()`, and it starts executing from its entry point.

4.10 Killing a task

Normally a task runs to completion and then exits. It may also choose to exit early by calling `task_exit()`. However, it is also possible to force a task to exit early, using the function:

```
int task_kill(
  task_t* task,
  int status,
  task_kill_flags_t flags);
```

This stops the task immediately, causes it to run the exit handler (if there is one), and exit.

Sometimes it may be desirable for a task to prevent itself being killed temporarily, for example, while it owns a mutual exclusion semaphore. To do this, the task can make itself immortal by calling:

```
void task_immortal(void);
```

and once it is willing to be killed again calling:

```
void task_mortal(void);
```

While the task is immortal, it cannot be killed. However, if an attempt was made to kill the task while it was immortal, it dies immediately when it makes itself mortal again by calling `task_mortal()`.

Calls to `task_immortal()` and `task_mortal()` nest correctly, so the same number of calls must be made to both functions before the task becomes mortal.

4.11 Getting the current task's id

Several functions are provided for obtaining details of a specified task. The following function returns a pointer to the task structure of the current task:

```
task_t* task_id (void)
```

This function may be used with `task_wait()`, see *task_wait on page 87*. The function:

```
const char* task_name(task_t *task);
```

returns the name of the specified task, or if `task` is `NULL`, the current task. (The task's name is set when the task is created).

4.12 Stack usage

A common problem when developing applications is not allocating enough stack for a task, or the need to tune stack allocation to minimize memory wastage. OS21 provides a couple of techniques which can be used to address this.

The first technique is to enable stack checking in the compiler. This adds an additional function call at the start of each of the user's functions, just before any additional stack is allocated. The called stack check function can then determine whether there is sufficient space available for the function which is about to execute.

OS21 does not support GCC stack checking.

Although stack checking has the advantage that a stack overflow is reported immediately it occurs, it has several problems:

- there is a run time cost incurred every function call to perform the check
- it cannot report on functions which are not recompiled with stack checking enabled

An alternative technique is to determine experimentally, how much stack a task uses by giving the task a large stack initially, running the code, and then seeing how much stack has been used. To support this, OS21 normally fills a task's stack with a known value. As the task runs it writes its own data into the stack, altering this value, and later the stack can be inspected to determine the highest address which has not been altered.

To support this, OS21 provides the function:

```
int task_status(
  task_t* task,
  task_status_t* status,
  task_status_flags_t flags);
```

This function can be used to determine information about the task's status, for example the base and size of the stack specified when the task was created.

Stack filling is enabled by default, however, in some cases the user may want to control it, so two functions are provided:

```
int task_stack_fill(task_stack_fill_t* fill);
```

returns details about the current stack fill settings, and:

```
int task_stack_fill_set(task_stack_fill_t* fill);
```

allows them to be altered. Stack filling can be enabled or disabled, or the fill value can be changed. By default it is enabled, and the fill value set to `0x12345678`.

By placing a call to `task_stack_fill_set()` in a start-up function, before the OS21 kernel is initialized, it is possible to control the filling of the root task's stack.

To determine how much stack has been used `task_status()` can be called, with the `flags` parameter set to `task_status_flags_stack_used`. For this to work correctly, task stack filling must have been enabled when the task was created, and the fill value must have the same value as the one which was in effect when the task was created.

4.13 Task data

4.13.1 Application data

OS21 provides one word of **task-data** per task. This can be used by the application to store data which is specific to the task, but which needs to be accessed uniformly from multiple tasks.

This is typically used to store data which is required by a library, when the library can be used from multiple tasks but the data is specific to the task. For example, a library which manages an I/O channel may be called by multiple tasks, each of which has its own I/O buffers. To avoid having to pass an I/O descriptor into every call it could be stored in task-data.

Although only one word of storage is provided, this is usually treated as a pointer, which points to a user defined data structure which can be as large as required.

Two functions provide access to the task-data pointer:

```
void* task_data_set (task_t* task, void* new_data);
```

sets the task-data pointer of the task specified by `task`.

```
void* task_data (task_t* task);
```

retrieves the task-data pointer of the task specified by `task`.

If `task` is `NULL`, both functions use the currently active task.

When a task is first created (including the root task), its task-data pointer is set to `NULL` (0). For example:

```
typedef struct {
   char buffer[BUFFER_SIZE];
   char* buffer_next;
   char* buffer_end;
} ptd_t;
char buffer_read(void)
{
   ptd_t *ptd;
   ptd = task_data(NULL);
   if (ptd->buffer_next == ptd->buffer_end) {
      ... fill buffer ...
   }
   return *(ptd->buffer_next++);
}
int main()
{
   ptd_t *ptd;
   task_t *task;
   ... create a task ...
   ptd = memory_allocate(some_partition, sizeof(ptd_t));
   ptd->buffer_next = ptd->buffer_end = ptd->buffer;
   task_data_set(task, ptd);
}
```

Note: `kernel_start()` *must be called before* `memory_allocate()` *to prevent the function failing.*

4.13.2 Library data

OS21 also provides a facility to manage multiple instances of task private data. This is to enable libraries to store their own per task private data. Two function calls provide access to this facility:

```
void* task_private_data(
   task_t* task,
   void * cookie );

int task_private_data_set(
   task_t* task,
   void* data,
   void* cookie,
   void (*destructor)( void* data ));
```

This API allows a client to allocate and associate a block of data with a given `task`, under a unique `cookie` identifier. The `cookie` is typically the address of some object in the client library to guarantee uniqueness.

`task_private_data()` returns `NULL` if no data has been registered under the given `cookie`, otherwise it returns the address of the private data block.

`task_private_data_set()` is used to request that a block of `data` be associated with the given `task` under the given `cookie`. Only one data block can be registered under a given cookie for a given task. The `destructor` parameter is the address of a routine which OS21 calls when the task is deleted. The `destructor` is called with the address of the task private data allocated by the library, and it has the responsibility to deallocate this data.

If the `task` parameter is `NULL`,the current task is used for the operation.

If `task_private_data()` or `task_private_data_set()` are called before kernel initialization, the operations are performed on the root task.

4.14 Task termination

A task terminates when it returns from the task's entry point function.

A task may also terminate by using the following function:

```
void task_exit(int param);
```

In the latter case an exit status can be specified. When the task returns from its entry point function, the exit status is 0. If `task_exit()` is called, the exit status is specified as the parameter. This value is then made available to the **onexit** handler if one has been installed (see the following example), and also by using the `task_status()` call.

Just before the task terminates (either by returning from its entry point function, or calling `task_exit()`), it calls an **onexit** handler(s). These functions allow any application-specific tidying up to be performed before the task terminates. **onexit** handlers are installed by calling one of these two functions:

```
task_onexit_fn_t task_onexit_set(
   task_onexit_fn_t fn);
int task_private_onexit_set(
   task_t* task,
   task_onexit_fn_t fn);
```

The **onexit** handler function must have a prototype of:

```
void onexit_handler(
   task_t *task,
   int param)
```

When a handler function is called, `task` specifies the task which has exited, and `param` is the task's exit status.

`task_onexit_set()` registers a global **onexit** handler, which is called when any task exits. Only one global **onexit** handler may be registered. This call returns the address of the previously registered handler.

`task_private_onexit_set()` registers an **onexit** handler for the given task. Handlers registered with this API are only called when the specified task terminates. This API allows multiple handlers to be registered. OS21 invokes the handlers in the reverse order to the order they were registered with the task.

All task private **onexit** handlers are called before the global one.

The following code example shows how a task's exit code can be stored in its task-data (see *Section 4.13: Task data on page 51*), and retrieved later by another task which is notified of the termination through `task_wait()`.

4.15 Waiting for termination

The following function waits until one of a list of tasks terminates or the specified timeout period is reached:

```
int task_wait(
   task_t **tasklist,
   int ntasks,
   const osclock_t *timeout);
```

Timeouts for tasks are implemented using hardware and do not increase the application's code size. Any task can wait for any other asynchronous task to complete. A parent task should, for example, wait for any children to terminate. In this case, `task_wait()` can be used inside a loop.

After `task_wait()` has indicated that a particular task has completed, any of the task's data including any memory dynamically loaded or allocated from the heap and used for the task's stack, can be freed. The task's state that is, its control block `task_t` may also be freed. (`task_delete` can be used to free `task_t`, see *Section 4.17: Deleting a task on page 54*).

The `timeout` period for `task_wait()` may be expressed as a certain number of ticks or it may take one of two values: `TIMEOUT_IMMEDIATE` indicates that the function should return immediately, even if no tasks have terminated, and `TIMEOUT_INFINITY` indicates that the function should ignore the timeout period, and only return when a task terminates. The header file `os21/ostime.h` must be included when using this function, see *Section 4.19: Task API summary on page 55*.

4.16 Getting a task's exit status

A task's exit status is available once the task has terminated, through the `task_status()` function.

4.17 Deleting a task

A task can be deleted using the `task_delete()` function:

```
#include <os21.h>
int task_delete(
  task_t* task);
```

This removes the task from the list of known tasks and allow its stack and data structures to be reused.

`task_delete()` calls `memory_deallocate()` to free the task's state and the task's stack.

A task must have terminated before it can be deleted, if it has not `task_delete` fails.

4.18 Enumerating all tasks

All tasks on the system can be enumerated with the `task_list_next()` function:

```
#include <os21.h>
task_t* task_list_next(
  task_t* task);
```

This returns successive OS21 task descriptors for each call, returning `NULL` when the end of the list of tasks has been reached. Passing in a `NULL` pointer returns the first task on the list.

Note: There is no synchronization with the task list structure implied with this call. The call should wrap calls to this function with `task_lock()` *and* `task_unlock()` *to guarantee a consistent list of tasks are returned.*

4.19 Task API summary

All the definitions related to tasks are obtained by including the header file `os21.h`, which itself includes the header file `task.h`. See *Table 9*, *Table 10* and *Table 11* for a complete list.

Table 9. Functions defined in task.h

Function	Description
`task_context()`	Returns the current execution context
`task_create()`	Creates an OS21 task
`task_create_p()`	Creates an OS21 task using specific partitions
`task_data()`	Retrieves a task's data pointer
`task_data_set()`	Sets a task's data pointer
`task_delay()`	Delays the calling task for a period of time
`task_delay_until()`	Delays the calling task until a specified time
`task_delete()`	Deletes a task
`task_exit()`	Exits the current task
`task_id()`	Returns the current task's ID
`task_immortal()`	Makes the current task immortal
`task_kill()`	Kills a task
`task_list_next()`	Returns the next task in the list
`task_lock()`	Locks current task to prevent task rescheduling
`task_lock_task()`	Locks any task
`task_mortal()`	Makes the current task mortal
`task_name()`	Returns the task's name
`task_onexit_set()`	Sets up a function to be called when a task exits
`task_priority()`	Returns a task's priority
`task_priority_set()`	Sets a task's priority
`task_private_data()`	Retrieves some task private data
`task_private_data_set()`	Registers some task private data
`task_private_onexit_set()`	Registers a task private onexit handler.
`task_reschedule()`	Current task yields the CPU if not locked
`task_resume()`	Resumes a suspended task
`task_stack_fill()`	Returns the task fill configuration
`task_stack_fill_set()`	Sets the task stack fill configuration
`task_stackinfo()`	Obtain basic task stack information
`task_stackinfo_set()`	Set basic task stack information
`task_status()`	Returns status information about the task
`task_suspend()`	Suspends a task

Table 9. Functions defined in task.h (continued)

Function	Description
task_unlock()	Unlocks current task to allow task rescheduling
task_unlock_task()	Unlocks any task
task_wait()	Waits until one of a list of tasks completes
task_yield()	Current task unconditionally yields the CPU

Table 10. Types defined in task.h

Type	Description
task_context_t	Execution context
task_flags_t	Additional flags for task_create() and task_create_p()
task_kill_flags_t	Additional flags for task_kill
task_onexit_fn_t	Function to be called on task exit
task_state_t	State of a task (for example, active, deleted)
task_stack_fill_state_t	Whether stack filling is enabled or disabled
task_stack_fill_t	Stack filling state (specifies enables and value)
task_status_flags_t	Additional flags for task_status
task_status_t	Result of task_status
task_t	A task's state

Table 11. Macros defined in task.h

Macro	Description
OS21_PRIORITY_LEVELS	Number of OS21 priority levels (default is 256)
MAX_USER_PRIORITY	Highest user task priority (default is 255)
MIN_USER_PRIORITY	Lowest task priority (default is 0)

4.20 Task function definitions

task_context

Return the current execution context

Definition:

```
#include <os21.h>
task_context_t task_context(
  task_t **task,
  int* interrupt_info);
```

Arguments:

`task_t **task`	Where to return the task descriptor
`int* interrupt_info`	Where to return the platform specific interrupt information

Returns: Returns whether the function was called from a task or system context, or if the OS21 kernel has not started yet.

Errors: None

Context: Callable from task or system context.

Description: The `task_context` function returns a description of the context from which it is called. This can be task context, system context or no context (called before the kernel has started). This is indicated by one of the following values.

- If the function was called before OS21 has been started (by calling `kernel_start()`), it returns `task_context_none`.
- If the function was called from an OS21 task, it returns `task_context_task`. If `task` is not `NULL`, the corresponding `task_t *` is written into the variable pointed to by `task`.
- If the function was called from an interrupt handler, it returns `task_context_system`. If `interrupt_info` is not `NULL`, platform specific interrupt information is written into the variable pointed to by `interrupt_info`. This interrupt information is guaranteed to be non-zero.
- If the function was called from an exception handler, it returns `task_context_system`. If `interrupt_info` is not `NULL`, `0` is written into the variable pointed to by `interrupt_info`.

task_create

Create an OS21 task

Definition:

```
#include <os21.h>
task_t* task_create(
  void (*function)(void*),
  void* param,
  size_t stack_size,
  int priority,
  const char* name,
  task_flags_t flags);
```

Arguments:

`void (*function)(void*)`	Pointer to the task's entry point
`void* param`	The parameter which is passed into `function`
`size_t stack_size`	Required stack size for the task, in bytes
`int priority`	Task's scheduling priority in the range `MIN_USER_PRIORITY` to `MAX_USER_PRIORITY`
`const char* name`	The name of the task, to be used by the debugger
`task_flags_t flags`	Various flags which affect task behavior

Returns: Returns a pointer to the task structure if successful or `NULL` otherwise. The returned structure pointer should be assigned to a local variable for future use.

Errors: Returns a `NULL` pointer if an error occurs because the task's priority is invalid, the stack is not big enough, or there is insufficient memory for the task's data structures or stack.

Context: Callable from task only.

Description: `task_create()` sets up a function as an OS21 task and starts the task executing. It returns a pointer to the task control block, `task_t`, which is subsequently used to refer to the task.

`function` is a pointer to the function which is to be the entry point of the task.

`stack_size` is the size of the stack space required in bytes. It is important that enough stack space is requested, if not, the results of running the task are undefined. `task_create` automatically allocates the stack from the system heap.

OS21 mandates a minimum task stack on each platform, which is given by the value `OS21_DEF_MIN_STACK_SIZE`. Although this value is defined separately for each platform, the correct value is obtained from the `os21.h` header file:

```
#include <os21.h>
```

If you are sure of your task's stack requirements, you can override the enforcement of this check by specifying `task_flags_no_min_stack_size` in the `flags` parameter. With this flag set there is no minimum stack size enforced.

`param` is a pointer to the arguments to `function`. If `function` has several parameters, these should be combined into a structure and the address of the

structure provided as the argument to `task_create()`. When the task is started it begins executing as if `function` were called with the single argument `param`.

The task's data structures are also allocated by `task_create()` from the system heap.

`priority` is the task's scheduling priority.

`name` is the name of the task, which is passed to the debugger (if present) so that the task can be correctly identified in the debugger's task list.

`flags` is used to give additional information about the task. Normally `flags` should be specified as 0, which results in the default behavior, however, the value `task_flags_suspended` can be used to create tasks which are initially suspended. This means that the task does not run until it is resumed using the `task_resume()` call.

Current possible values for `flags` are:

`0`	Create a runnable OS21 task (default).
`task_flags_suspended`	Create a suspended OS21 task.
`task_flags_no_min_stack_size`	Do not enforce minimum stack size checks.

Example:

```
struct sig_params{
   semaphore_t *Ready;
   int Count;
};
void signal_task(void* p)
{
   struct sig_params* Params = (struct sig_params*)p;
   int j;
   for (j = 0; j < Params->Count; j++) {
      semaphore_signal (Params->Ready);
      task_delay(ONE_SECOND);
   }
}
foo(void) {
   task_t* Task;
   struct sig_params params;
   Task = task_create (signal_task, &params,
        USER_WS_SIZE, USER_PRIORITY, "Signal", 0);
   if (Task == NULL) {
      printf ("Error : create. Unable to create task\n");
      exit (EXIT_FAILURE);
   }
   ...
}
```

task_create_p

Create an OS21 task

Definition:

```
#include <os21.h>
task_t* task_create_p(
  partition_t* partition,
  void (*function)(void*),
  void* param,
  partition_t* stack_partition,
  size_t stack_size,
  int priority,
  const char* name,
  task_flags_t flags);
```

Arguments:

`partition_t* partition`	The partition from which to allocate control structures
`void (*function)(void*)`	Pointer to the task's entry point
`void* param`	The parameter which is passed into `function`
`partition_t* stack_partition`	The partition from which to allocate the stack
`size_t stack_size`	Required stack size for the task, in bytes
`int priority`	Task's scheduling priority in the range `MIN_USER_PRIORITY` to `MAX_USER_PRIORITY`
`const char* name`	The name of the task, to be used by the debugger
`task_flags_t flags`	Various flags which affect task behavior

Returns: Returns a pointer to the task structure if successful or `NULL` otherwise. The returned structure pointer should be assigned to a local variable for future use.

Errors: Returns a `NULL` pointer if an error occurs because:

- the task's priority is invalid
- there is insufficient memory for the task's data structures or stack
- either of the two partitions contain memory from a non-fixed virtual address mapping

Context: Callable from task only.

Description: `task_create_p()` sets up a function as an OS21 task and starts the task executing. `task_create_p()` returns a pointer to the task control block `task_t`, which is subsequently used to refer to the task.

`partition` specifies where to allocate the control structures from.

`function` is a pointer to the function which is to be the entry point of the task.

`stack_partition` is the partition from which to allocate the stack.

Note: If a null pointer is specified for `partition` *or* `stack_partition`*, instead of a valid partition pointer, the C runtime heap is used.*

Note: `task_create_p()` will fail and return `NULL` if either of the two partitions contain memory from a non-fixed virtual address mapping.

`stack_size` is the size of the stack space required in bytes. It is important that enough stack space is requested, if not, the results of running the task are undefined. `task_create_p()` calls `memory_allocate()` to allocate the stack from the memory partition specified.

OS21 mandates a minimum task stack on each platform, which is given by the value `OS21_DEF_MIN_STACK_SIZE`. Although this value is defined separately for each platform, the correct value is obtained from the `os21.h` header file:

```
#include <os21.h>
```

If you are sure of your task's stack requirements, you can override the enforcement of this check by specifying `task_flags_no_min_stack_size` in the `flags` parameter. With this flag set there is no minimum stack size enforced.

`param` is a pointer to the arguments to `function`. If `function` has several parameters, these should be combined into a structure and the address of the structure provided as the argument to `task_create_p()`. When the task is started it begins executing as if `function` were called with the single argument `param`.

The task's data structures are also allocated by `task_create_p()` calling `memory_allocate()`. The task state (`task_t`) is allocated from the nominated memory partition.

`priority` is the task's scheduling priority.

`name` is the name of the task, which is passed to the debugger (if present) so that the task can be correctly identified in the debugger's task list.

`flags` is used to give additional information about the task. Normally `flags` should be specified as 0, which results in the default behavior, however, the value `task_flags_suspended` can be used to create tasks which are initially suspended. This means that the task does not run until it is resumed using the `task_resume` call.

Possible values for `flags` are:

`0`	Create a runnable OS21 task (default).
`task_flags_suspended`	Create a suspended OS21 task.

Example:

```
struct sig_params{
   semaphore_t *Ready;
   int Count;
};

partition_t* my_partition;

void signal_task(void* p)
{
   struct sig_params* Params = (struct sig_params*)p;
   int j;
   for (j = 0; j < Params->Count; j++) {
      semaphore_signal (Params->Ready);
      task_delay(ONE_SECOND);
   }
}

foo(void) {
   task_t* Task;
   struct sig_params params;
   Task = task_create_p (my_partition, signal_task, &params,
        my_partition, USER_WS_SIZE, USER_PRIORITY, "Signal",
0);
   if (Task == NULL) {
      printf ("Error : create. Unable to create task\n");
      exit (EXIT_FAILURE);
   }
   ...
}
```

See also: task_delete

task_data

Retrieve a task's data pointer

Definition:

```
#include <os21.h>
void* task_data(
  task_t* task);
```

Arguments:

`task_t* task`	Pointer to the task structure

Returns: Returns the task data pointer of the task pointed to by `task`. If `task` is `NULL`, the return result is the data pointer of the calling task.

Errors: None

Context: Callable from task only.

Description: `task_data()` retrieves the task-data pointer of the task specified by `task`, or the currently active task if `task` is `NULL`. See *Section 4.13: Task data on page 51*.

See also: `task_data_set`

task_data_set

Set a task's data pointer

Definition:

```
#include <os21.h>
void* task_data_set(
  task_t* task,
  void* data);
```

Arguments:

`task_t* task`	Pointer to the task structure
`void* data`	New data pointer for the task

Returns: `task_data_set()` returns the task's previous data pointer. If `task` is `NULL`, the return result is the data pointer of the calling task.

Errors: None

Context: Callable from task only.

Description: `task_data_set()` sets the task-data pointer of the task specified by `task`, or of the currently active task if `task` is `NULL`. See *Section 4.13: Task data on page 51*.

See also: `task_data`

task_delay

Delay the calling task for a period of time

Definition:

```
#include <os21.h>
void task_delay(
  osclock_t delay);
```

Arguments:

`osclock_t delay`	The period of time to delay the calling task

Returns: None

Errors: None

Context: Callable from task only.

Description: Delay the calling task for the specified period of time. `delay` is specified in ticks, which is an implementation dependent quantity, see *Chapter 10: Real-time clocks on page 140*.

See also: `task_delay_until`

task_delay_until

Delay the calling task until a specified time

Definition:

```
#include <os21.h>
void task_delay_until(
  osclock_t this_time);
```

Arguments:

`osclock_t this_time`	The time period during which the calling task is delayed

Returns: None

Errors: None

Context: Callable from task only.

Description: Delay the calling task until the specified time. If `this_time` is before the current time, this function returns immediately. `this_time` is specified in ticks, which is an implementation dependent quantity, see *Chapter 10: Real-time clocks on page 140*.

See also: `task_delay`

task_delete

Delete an OS21 task

Definition:

```
#include <os21.h>
int task_delete(
  task_t* task);
```

Arguments:

`task_t *task`	Task to delete

Returns: Returns `OS21_SUCCESS` on success, `OS21_FAILURE` on failure.

Errors: If the task has not yet terminated, this fails.

Context: Callable from task only.

Description: This function allows a task to be deleted. The task must have terminated (by returning from its entry point function) before this can be called. Attempting to delete a task which has not yet terminated fails.

See also: `task_create`

task_exit

Exit the current task

Definition:

```
#include <os21.h>
void task_exit(
  int param);
```

Arguments:

`int param`	Parameter to pass to **onexit** handler

Returns: None

Errors: None

Context: Callable from task only.

Description: This causes the current task to terminate, after having called the **onexit** handler. It has the same effect as the task returning from its entry point function.

See also: `task_onexit_set`

task_id

Find current task's ID

Definition:
```
#include <os21.h>
task_t* task_id(void);
```

Arguments: None

Returns: Returns a pointer to the OS21 task structure of the calling task.

Errors: None

Context: Callable from task only.

Description: Returns a pointer to the task structure of the currently active task.

See also: `task_create`

task_immortal

Make the current task immortal

Definition:
```
#include <os21.h>
void task_immortal(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task only.

Description: `task_immortal()` makes the current task immortal. If an attempt is made to kill a task while it is immortal, it does not die immediately but continues running until it becomes mortal again, and dies.

This is callable from tasks only.

See also: `task_kill`, `task_mortal`

task_kill

Kill a task

Definition:

```
#include <os21.h>
int task_kill(
  task_t* task,
  int status,
  task_kill_flags_t flags);
```

Arguments:

`task_t* task`	The task to be killed
`int status`	The task's exit status
`task_kill_flags_t flags`	Additional flags

Returns: Returns `OS21_SUCCESS` if the task is successfully killed, `OS21_FAILURE` if it cannot be killed.

Errors: If the task has been deleted, this call fails.

Context: Callable from a task or system context. Only valid from a system context if `task` is not `NULL`.

Description: `task_kill()` kills the task specified by `task`, causing it to stop running, and call its exit handler. If `task` is `NULL`, the current task is killed. If the task was waiting on any objects when it is killed, it is removed from the list of tasks waiting for that object before the exit handler is called.

`status` is the exit status for the task. Therefore `task_kill()` can be viewed as a way of forcing the task to call `task_exit(status)`.

Normally `flags` should have the value `0`. However, by specifying the value `task_kill_flags_no_exit_handler`, it is possible to prevent the task calling its exit handler, and so it terminates immediately, never running again.

A task can temporarily make itself immune to being killed by calling `task_immortal()`, see *Section 4.10: Killing a task on page 49* for more details. When a task which has made itself immortal is killed, `task_kill()` returns immediately, but the killed task does not die until it makes itself mortal again.

Note: `task_kill()` *may return before the task has died. A* `task_kill()` *should normally be followed by a* `task_wait()` *to be sure that the task has made itself mortal again, and completed its exit handler. If the task is mortal, its exit handlers are called from the killing task's context; not the context of the task being killed.*

`task_kill()` cannot be called from an interrupt handler.

Example:

```
void tidy_up(task_t* task, int status)
{
   task_kill(task, status, 0);
   task_wait(&task, 1, TIMEOUT_INFINITY);
   task_delete(task);
}
```

See also: `task_delete`, `task_mortal`, `task_immortal`

task_list_next

Return the next task in the task list

Definition:

```
#include <os21.h>
task_t* task_list_next(
  task_t* task);
```

Arguments:

`task_t* task`	Previous task descriptor or `NULL`

Returns: The task descriptor for the next task on OS21's task list. `NULL` if list exhausted.

Errors: None

Context: Callable from task or system context.

Description: This function returns the task descriptor of the next task on OS21's internal list of tasks. Passing a `NULL` parameter returns the first task on the list. This enables the caller to enumerate all OS21 tasks on the system.

The caller should bracket calls to this function with `task_lock()` and `task_unlock()` to ensure that a consistent list of tasks are returned.

Example:

```
task_t * task = NULL;

task_lock();
while( (task = task_list_next(task)) != NULL )
{
   ...process this task descriptor...
}
task_unlock();
```

task_lock

Prevent task rescheduling for current task

Definition:
```
#include <os21.h>
void task_lock(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task only.

Description: This function prevents the kernel scheduler from pre-empting or timeslicing the current task, although the task can still be interrupted by interrupt handlers.

This function should always be called as a pair with `task_unlock()`, so that it can be used to create a critical region in which the task cannot be pre-empted by another task. If the task deschedules, the lock is terminated while the thread is not running. When the task is rescheduled the lock is re-instated. Calls to `task_lock()` can be nested, and the lock is not released until an equal number of calls to `task_unlock()` have been made.

Note: `task_lock()` *and* `task_unlock()` *can be called before the kernel is started with* `kernel_start()`*. This allows the C runtime library to use* `task_lock()`*/*`task_unlock()` *for its critical sections. These may occur (for example, calls to* `malloc()`*) before the kernel is started.*

See also: `task_unlock, task_lock_task`

task_lock_task

Prevent task rescheduling

Definition:

```
#include <os21.h>
void task_lock_task(task_t *taskp);
```

Arguments:

`task_t* taskp`	Task to lock

Returns: None

Errors: None

Context: Callable from task only.

Description: This function increments the lock count of the given task. If the given task is NULL, then the lock count of the active task (the caller) is incremented. If a task is running on the CPU and its lock count is greater than zero, then it cannot be pre-empted. Only when the count reaches zero again can the task be pre-empted.

The following calls are all equivalent:

```
task_lock_task(NULL);
```

```
task_lock_task(task_id());
```

```
task_lock();
```

Since they all act on the currently executing task, the effect is to lock the task so that it cannot be pre-empted or switched off the CPU.

Interrupt handling is not locked out by this call.

`task_lock_task(taskp)` calls (where `taskp != NULL` and `taskp != task_id()`) cause the lock count of the given task to be incremented, but since `taskp` is not on the CPU at the time of the call, pre-emption is not disabled. However, if `taskp` gains the CPU, it gains the CPU with pre-emption disabled.

When a task is on the CPU with its lock count greater than zero (pre-emption disabled), the only way of unlocking it (re-enabling pre-emption) is through it calling `task_unlock()`. The fact that pre-emption is disabled prevents another task from calling `task_unlock_task()`. However, another task may call `task_unlock_task()` to decrement the lock count of a task before that task gains the CPU.

Note: Great care must be taken when using this function. Misuse can easily result in program deadlock and undesired behavior. Use of `task_lock_task()` *is* ***not*** *recommended.*

See also: `task_lock`, `task_unlock_task`. `task_unlock_task()`

task_mortal

Make the current task mortal

Definition:
```
#include <os21.h>
void task_mortal(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task only.

Description: `task_mortal()` makes the current task mortal again. If an attempt had been made to kill the task while it was immortal, it dies as soon as `task_mortal()` is called. Calls to `task_immortal()` are cumulative. A task makes two calls to `task_immortal()`, then two calls to `task_mortal()` are required before it becomes mortal again.

`task_mortal()` is not callable from interrupt handlers

See also: `task_immortal`, `task_kill`

task_name

Return the name of the specified task

Definition:
```
#include <os21.h>
const char* task_name(
  task_t *task);
```

Arguments:

`task_t* task`	Task to return the name of

Returns: The name of the specified task.

Errors: None

Context: Callable from task or system context. Only valid from system context if `task` is not `NULL`.

Description: This function returns the name of the specified task, or if `task` is `NULL`, the current task. The task's name is set when the task is created.

See also: `task_create`, `task_create_p`

task_onexit_set

Set the global task onexit handler

Definition:

```
#include <os21.h>
task_onexit_fn_t task_onexit_set(
  task_onexit_fn_t fn);
```

Arguments:

`task_onexit_fn_t fn`	Task **onexit** handler to be called

Returns: Returns the previous global **onexit** handler, or `NULL` if none had previously been set.

Errors: None

Context: Callable from task or system context.

Description: Sets the global task **onexit** handler to be `fn`. This handler is called whenever a task exits. The handler is called by the task which exits, before the task is marked as terminated. `fn` is a pointer to a function which must have the following prototype:

```
void task_onexit_fn(task_t* task, int param)
```

where:

`task`	is the task pointer of the task which has just exited,
`param`	is the parameter which was passed to `task_exit()`.

The global task **onexit** handler is called after all task private **onexit** handlers.

See also: `task_exit`, `task_private_onexit_set`

task_priority

Retrieve a task's priority

Definition:

```
#include <os21.h>
int task_priority(
  task_t* task);
```

Arguments:

`task_t* task`	Pointer to the task structure

Returns: Returns the OS21 priority of the task pointed to by `task`. If `task` is `NULL` the return result is the priority of the calling task.

Errors: None

Context: Callable from task or system context. Only valid from system context if `task` is not `NULL`.

Description: `task_priority()` retrieves the OS21 priority of the task specified by `task` or the priority of the currently active task if `task` is `NULL`.

If the specified task is currently subject to a temporary priority boost by the priority inversion logic, the nominal priority is returned, not the boosted priority.

See also: `task_priority_set`

task_priority_set

Set a task's priority

Definition:

```
#include <os21.h>
int task_priority_set(
  task_t* task,
  int priority);
```

Arguments:

`task_t* task`	Pointer to the task structure
`int priority`	Desired OS21 priority value for the task

Returns: `task_priority_set()` returns the task's previous OS21 priority. If `task` is `NULL` the return result is the priority of the calling task.

Errors: None

Context: Callable from task or system context. Only valid from system context if `task` is not `NULL`.

Description: `task_priority_set()` sets the priority of the task specified by `task`, or of the currently active task if `task` is `NULL`. If this results in the current task's priority falling below that of another task which is ready to run, or a ready task now has a priority higher than the current task's, tasks are rescheduled.

If the specified task owns a priority mutex, priority inversion logic is also run. If the task owns a priority mutex for which a higher priority task is waiting, and the call attempted to lower the task's priority, the lowering of the priority is deferred until the mutex is released.

If the specified task is waiting for a priority mutex or semaphore, its position in the queue of waiting tasks is re-calculated. If the call attempted to raise the specified task's priority, and it was queuing for a priority mutex, priority inversion logic is also run and may result in the temporary priority boosting of the mutex's current owning task. See *Section 7.1.1: Priority inversion on page 111*.

See also: `task_priority`

task_private_data

Retrieve a task's private data pointer

Definition:

```
#include <os21.h>
void* task_private_data(
  task_t* task
  void* cookie);
```

Arguments:

`task_t* task`	Pointer to the task structure
`void* cookie`	Unique identifier

Returns: Returns the address of the private data registered for the task pointed to by `task`, under the unique identifier `cookie`, or `NULL` if no data has been registered.

Errors: None

Context: Callable from task only.

Description: `task_private_data()` retrieves the address of the private data for the task identified by `task`, under the unique identifier `cookie`. If `task` is `NULL` the calling task is used for the operation. This interface is intended to be used by libraries which have to store private data on a per task basis.

If this API is used before kernel initialization, the operation is performed on the root task.

See also: `task_private_data_set`

task_private_data_set

Set a task's private data pointer

Definition:

```
#include <os21.h>
int task_private_data_set(
  task_t* task,
  void* data,
  void* cookie,
  void (*destructor)( void* data ));
```

Arguments:

`task_t* task`	Pointer to the task structure
`void* data`	Pointer to task private data
`void* cookie`	Unique identifier
`void (*destructor)(void* data)`	Deallocation routine

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure

Errors: If OS21 runs out of memory, or private data for this task already exits under the specified cookie (and `data` is not `NULL`), `OS21_FAILURE` is returned.

Context: Callable from task only.

Description: `task_private_data_set()` is used to store private data for the task identified by `task`, under the unique identifier `cookie`. If `task` is `NULL`, the calling task is used for the operation. This interface is intended to be used by libraries which have to store private data on a per task basis.

The destructor routine is called when the task is deleted, so that the client can free the memory allocated.

If a piece of data registered with this call is no longer required, call this routine with a `NULL` data pointer. This will cause the destructor for the old data to be called and leaves the task with no data registered under the cookie given.

If this API is used before kernel initialization, the operation is performed on the root task.

See also: `task_private_data`

task_private_onexit_set

Set a per task onexit handler

Definition:

```
#include <os21.h>
int task_private_onexit_set(
  task_t* task,
  task_onexit_fn_t fn);
```

Arguments:

`task_t* task`	Task for which to register handler
`task_onexit_fn_t fn`	Task **onexit** handler to be called

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: If OS21 cannot allocate memory, `OS21_FAILURE` is returned.

Context: Callable from task only.

Description: Registers `fn` as a task private **onexit** handler. This handler is called when the task exits. The handler is called by the task which exits, before the task is marked as terminated. `fn` is a pointer to a function which must have the following prototype:

```
void task_onexit_fn(task_t* task, int param)
```

where:

`task`	is the task pointer of the task which has just exited,
`param`	is the parameter which was passed to `task_exit()`.

Per task private **onexit** handlers are called in the reverse of the order they were registered, and before the global task **onexit** handler.

See also: `task_exit`, `task_onexit_set`

task_reschedule

Reschedule the current task

Definition:
```
#include <os21.h>
void task_reschedule(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task only.

Description: This function reschedules the current task, moving it to the back of the current priority scheduling list, and selecting the new task from the front of the list. If the scheduling list was empty before this call, it has no effect, otherwise it performs a timeslice at the current priority.

If `task_reschedule()` is called while a `task_lock()` is in effect, it does not cause a reschedule.

See also: `task_yield`

task_resume

Resume a suspended task

Definition:

```
#include <os21.h>
int task_resume(
  task_t* task);
```

Arguments:

`task_t* task`	Pointer to the task structure

Returns: Returns `OS21_SUCCESS` if the task was successfully resumed, or `OS21_FAILURE` if it could not be resumed.

Errors: If the task is not suspended, the call fails.

Context: Callable from task or system context.

Description: This function resumes the specified task. The task must previously have been suspended, either by calling `task_suspend()`, or created by specifying a flag of `task_flags_suspended` to `task_create()` or `task_create_p()`.

If the task is suspended multiple times, by more than one call to `task_suspend()`, an equal number of calls to `task_resume()` are required before the task starts to execute again.

If the task was waiting for an object (for example, waiting on a semaphore) when it was suspended, that event must also occur before the task starts executing. When a task is resumed it starts executing the next time it is the highest priority task, and so may preempt the task calling `task_resume()`.

See also: `task_suspend`

task_stack_fill

Retrieve task stack fill settings

Definition:

```
#include <os21.h>
int task_stack_fill(
  task_stack_fill_t* fill);
```

Arguments:

`task_stack_fill_t* fill`	A pointer to the structure to be filled in

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: Returns `OS21_FAILURE` if `fill` is `NULL`.

Context: Callable from task or system context.

Description: `task_stack_fill()` retrieves the current settings for task stack filling and writes them to a structure provided by the pointer `fill`.

Table 12 on page 80 shows the layout of the structure `task_stack_fill_t`.

Example:

```
#include <os21/task.h>
int result;
task_stack_fill_t settings;
result = task_stack_fill(&settings);
```

See also: `task_create`, `task_create_p`, `task_stack_fill_set`

task_stack_fill_set

Set task stack fill settings

Definition:

```
#include <os21.h>
int task_stack_fill_set(
  task_stack_fill_t* fill);
```

Arguments:

`task_stack_fill_t* fill`	A pointer to new settings

Returns: Returns `OS21_SUCCESS` on success, `OS21_FAILURE` if an error occurs.

Errors: Returns `OS21_FAILURE` if the new settings are invalid.

Context: Callable from task or system context.

Description: `task_stack_fill_set()` allows task stack fill settings to be changed by reading the new settings from the structure provided by the pointer `fill`. Task stack filling can be enabled/disabled or the fill pattern redefined.

Any subsequent calls to the functions `task_create()` or `task_create_p()` use these settings when initializing the stack.

By default, task stack filling is enabled with a fill pattern of `0x12345678`. Any task that is created using `task_create()` or `task_create_p()` has it's stack initialized by overwriting the whole contents of the stack with the value `0x12345678`. *Table 12* shows the layout of the `task_stack_fill_t` structure.

Table 12. Layout of structure task_stack_fill_t

Field	Description
`task_stack_fill_state`	Enable/disable stack filling (see *Table 13*)
`task_stack_fill_pattern`	Pattern value used when a stack is initialized

Table 13 shows all the flag values which can be used in the field `task_stack_fill_state`. Any other value not in the table causes `task_stack_fill_set()` to return `OS21_FAILURE`.

Table 13. Flags used by task_stack_fill_state

Flag	Description
`task_stack_fill_state_off`	Disable task stack filling
`task_stack_fill_state_on`	Enable task stack filling

Example:

```
#include <os21.h>

task_stack_fill_t options = {
  task_stack_fill_state_on,
  0x76543210
};

int result = task_stack_fill_set(&options);
```

See also: `task_create`, `task_create_p`, `task_stack_fill`

task_stackinfo

Obtain task stack information

Definition:

```
#include <os21.h>
int task_stackinfo(
  task_t * taskp,
  char ** stack_basep,
  size_t * stack_sizep);
```

Arguments:

`taskp`	A pointer to the task for which stack information will be returned.
`stack_basep`	A pointer to the location where the address of the base of the stack for the given task will be stored.
`stack_sizep`	A pointer to the location where the size of the stack for the given task will be stored.

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: Returns `OS21_FAILURE` if `stack_basep` or `stack_sizep` is `NULL`, `taskp` is not a valid task, or if `taskp` is `NULL` and the function is called from system context.

Context: Callable from task or system context.

Description: `task_stackinfo()` retrieves stack information for the given task. If `taskp` is `NULL`, information is provided for the currently executing task.

See also: `task_stackinfo_set`

task_stackinfo_set

Set task stack information

Definition:

```
#include <os21.h>
int task_stackinfo_set(
  task_t * taskp,
  char * stack_base,
  size_t * stack_size);
```

Arguments:

`taskp`	A pointer to the task for which stack information will be set.
`stack_base`	The address of the base of the stack for the given task.
`stack_size`	The size of the stack for the given task.

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: Returns `OS21_FAILURE` if an invalid combination of `stack_base` and `stack_size` is specified, `taskp` is not a valid task, or if `taskp` is `NULL` and the function is called from system context.

Context: Callable from task or system context.

Description: `task_stackinfo_set()` sets stack information for the given task. If `taskp` is `NULL`, information is provided for the currently executing task.

See also: `task_stackinfo`

task_status

Return information about the specified task

Definition:

```
#include <os21.h>
int task_status(
  task_t* task,
  task_status_t* status,
  task_status_flags_t flags);
```

Arguments:

`task_t* task`	Pointer to the task structure
`task_status_t* status`	Where to return the status information
`task_status_flags_t flags`	What information to return

Returns: Returns `OS21_SUCCESS` if status successfully reported, `OS21_FAILURE` if failed.

Errors: If the task does not exist, the call fails.

Context: Callable from task or system context. Only valid from system context if `task` is not `NULL`.

Description: This function returns information about the specified task. If `task` is `NULL`, information is returned about the current task. Information is returned by filling in the fields of `status`, which must be allocated by the user, and is of type `task_status_t`. The fields of this structure are shown in *Table 14*.

Table 14. task_status_t fields

Field name	Description
`task_stack_base`	Base address of the task's stack
`task_stack_size`	Size of the task's stack in bytes
`task_stack_used`	Amount of stack used by the task in bytes
`task_stack_pointer`	The task's stack pointer at the time `task_status()` was called
`task_time`	CPU time used by the task
`task_state`	Running, terminated, or suspended
`task_exit_value`	Value set when `task_exit()` was called

The `flags` parameter is used to indicate which values should be returned. Values which can be determined immediately (`task_stack_base`, `task_stack_size`, `task_state`, `task_exit_value` and `task_time`) are always returned. If only these fields are required, `flags` should be set to `0`. However, calculating how much stack has been used may take a while, and so is only returned when `flags` is set to `task_status_flags_stack_used`. `task_exit_value` is only valid if `task_state` indicates that the task has terminated.

See also: `task_stack_fill_set`

task_suspend

Suspend a specified task

Definition:

```
#include <os21.h>
int task_suspend(
  task_t* task);
```

Arguments:

`task_t* task`	Pointer to the task structure

Returns: Returns `OS21_SUCCESS` if the task was successfully suspended, or `OS21_FAILURE` if it could not be suspended.

Errors: If the task has been deleted, the call fails.

Context: Callable from task or system context. Only valid from system context if `task` is not `NULL`.

Description: This function suspends the specified task. If `task` is `NULL`, this suspends the current task. `task_suspend()` stops the task from executing immediately, until it is resumed using `task_resume()`.

See also: `task_resume`

task_unlock

Allow task rescheduling for current task

Definition:
```
#include <os21.h>
void task_unlock(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task only.

Description: This function allows the scheduler to resume scheduling following a call to `task_lock()`. The highest priority task currently available (which may not be the task that calls this function) continues running.

This function should always be called as a pair with `task_lock()`, so that it can be used to create a critical region in which the task cannot be preempted by another task. As calls to `task_lock()` can be nested, the lock is not released until an equal number of calls to `task_unlock()` have been made.

Note: `task_lock()` *and* `task_unlock()` *can be called before the kernel is started with* `kernel_start()`*. This allows the C runtime library to use* `task_lock()`*/*`task_unlock()` *for its critical sections. These may occur (for example, calls to* `malloc()`*) before the kernel is started.*

See also: `task_lock`, `task_unlock_task`

task_unlock_task

Allow task rescheduling

Definition:

```
#include <os21.h>
void task_unlock_task(task_t * taskp);
```

Arguments:

`task_t* taskp`	Task to unlock

Returns: None

Errors: None

Context: Callable from task only.

Description: This function decrements the lock count of the given task. If the given task is NULL, then the lock count of the active task (the caller) is decremented. If a task is running on the CPU and its lock count is greater than zero, then it cannot be pre-empted. Only when the count reaches zero again can the task be pre-empted.

The following calls are all equivalent:

```
task_unlock_task(NULL);
```

```
task_unlock_task(task_id());
```

```
task_unlock();
```

Since they all act on the currently executing task, the effect is to decrement the lock count, and if as a result it reaches zero then the task is unlocked so that pre-emption can occur once more.

`task_unlock_task(taskp)` calls (where `taskp != NULL` and `taskp != task_id()`) cause the lock count of the given task to be decremented.

When a task is on the CPU with its lock count greater than zero (pre-emption disabled), the only way of unlocking it (re-enabling pre-emption) is through it calling `task_unlock()`. The very fact that pre-emption is disabled prevents another task from calling `task_unlock_task()`. Another task may call `task_unlock_task()` to decrement the lock count of a task before that task gains the CPU.

Note: Great care must be taking when using this function. Misuse can easily result in program deadlock and undesired behavior. Use of `task_unlock_task()` *is not recommended.*

See also: `task_lock`, `task_unlock`, `task_lock_task`

task_wait

Waits until one of a list of tasks completes

Definition:

```
#include <os21.h>
int task_wait(
  task_t** tasklist,
  int ntasks,
  osclock_t* timeout);
```

Arguments:

`task_t** tasklist`	Pointer to a list of `task_t` pointers
`int ntasks`	The number of tasks in `tasklist`
`const osclock_t* timeout`	Maximum time to wait for tasks to terminate Expressed in ticks or as `TIMEOUT_IMMEDIATE` or `TIMEOUT_INFINITY`

Returns: The index into the array of the task which has terminated, or `OS21_FAILURE` if the timeout occurs.

Errors: None

Context: Callable from task only.

Description: `task_wait()` waits until one of the indicated tasks has terminated (by returning from its entry point function or calling `task_exit()`), or the timeout period has passed. Only once a task has been waited for in this way is it safe to free or otherwise reuse its stack, and `task_t` data structure.

`tasklist` is a pointer to a list of `task_t` structure pointers, with `ntasks` elements. Task pointers may be `NULL`, in which case that element is ignored.

`timeout` is a pointer to the timeout value. If this time is reached, the function returns the value `OS21_FAILURE`.

The timeout value is specified in ticks, which is an implementation dependent quantity, see *Chapter 10: Real-time clocks on page 140*.

Two special values can be specified for `timeout`:

`TIMEOUT_IMMEDIATE`	indicates that the function should return immediately, even if no tasks have terminated,
`TIMEOUT_INFINITY`	indicates that the function should ignore the timeout period, and only return when a task terminates.

See also: `task_create, task_create_p`

task_yield

Reschedule the current task

Definition:
```
#include <os21.h>
void task_yield(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task only.

Description: This function reschedules the current task, moving it to the back of the current priority scheduling list, and selecting the new task from the front of the list. If the scheduling list was empty before this call, it has no effect, otherwise it performs a timeslice at the current priority.

`task_yield()` always causes a reschedule, even if it is called while a `task_lock()` is in effect.

See also: `task_reschedule`

5 Callbacks

The callback API is provided to enable user supplied hook routines to be called whenever a given OS21 event occurs. These events are scheduler or interrupt events like task creation, task deletion, task switching and begin/end of interrupt processing. It is the intent of these callback functions to provide a mechanism by which performance profiling code can be added to applications.

Note: 1 To improve performance, interrupt handlers may loop to service more than one interrupt from a device, so it is possible for user interrupt code to run more times than indicated by the interrupt enter and exit callbacks.

2 The callback API works only if callbacks are enabled. If callbacks are not enabled, OS21 does not call user-specified callbacks. See Section 17.2: BSP data on page 207 for details of how to enable and disable callbacks.

3 The fact that an interrupt or exception handler is called does not necessarily mean that the interrupt or the exception occurred. Multiple handlers can share the same exception or interrupt. See Chapter 11: Interrupts on page 145 and Chapter 14: Exceptions on page 180 for more information.

5.1 Callback API summary

All the definitions related to callbacks can be obtained by including the header file `os21.h`, which itself includes the header file `callback.h`. See *Table 15* and *Table 16* for a complete list.

Table 15. Functions defined in callback.h

Function	Description
`callback_exception_enter()`	Registers an exception enter callback routine
`callback_exception_exit()`	Registers an exception exit callback routine
`callback_exception_install()`	Registers an exception install callback routine
`callback_exception_uninstall()`	Registers an exception uninstall callback routine
`callback_interrupt_enter()`	Registers an interrupt enter callback routine
`callback_interrupt_exit()`	Registers an interrupt exit callback routine
`callback_interrupt_install()`	Registers an interrupt install callback routine
`callback_interrupt_uninstall()`	Registers an interrupt uninstall callback routine
`callback_task_create()`	Registers a task create callback routine
`callback_task_delete()`	Registers a task delete callback routine
`callback_task_exit()`	Registers a task exit callback routine
`callback_task_switch()`	Registers a task switch callback routine

Table 16. Types defined in callback.h

Type	Description
callback_excp_install_fn_t	Callback function type for exception install events
callback_excp_uninstall_fn_t	Callback function type for exception uninstall events
callback_excp_enter_fn_t	Callback function type for exception enter events
callback_excp_exit_fn_t	Callback function type for exception exit events
callback_intr_install_fn_t	Callback function type for interrupt install events
callback_intr_uninstall_fn_t	Callback function type for interrupt uninstall events
callback_intr_enter_fn_t	Callback function type for interrupt enter events
callback_intr_exit_fn_t	Callback function type for interrupt exit events
callback_task_create_fn_t	Callback function type for task create events
callback_task_delete_fn_t	Callback function type for task delete events
callback_task_exit_fn_t	Callback function type for task exit events
callback_task_switch_fn_t	Callback function type for task switch events

5.2 Callback function definitions

callback_exception_enter

Register a callback routine for exception entry events

Definition:

```
#include <os21.h>
callback_excp_enter_fn_t callback_exception_enter(
  callback_excp_enter_fn_t fn);
```

Arguments:

callback_excp_enter_fn_t fn	Function to be called on exception handler entry

Returns: Pointer to previously installed callback function. NULL if none.

Errors: None

Context: Callable from task or system context.

Description: callback_exception_enter() registers a function to be called whenever an exception handler is called.

callback_excp_enter_fn_t is defined in callback.h as follows:

```
typedef void (*callback_excp_enter_fn_t)(void* handler);
```

where handler is the address of the exception handler being called.

See also: exception_install()

Note: The fact that an exception handler is called does not necessarily mean that the exception occurred. Multiple handlers can share the same exception. See Chapter 14: Exceptions on page 180 for more information.

callback_exception_exit

Register a callback routine for exception exit events

Definition:

```
#include <os21.h>
callback_excp_exit_fn_t callback_exception_exit(
  callback_excp_exit_fn_t fn);
```

Arguments:

`callback_excp_exit_fn_t fn`	Function to be called on exception handler exit

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_exception_exit()` registers a function to be called whenever an exception handler exits.

`callback_excp_exit_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_excp_exit_fn_t)(void* handler);
```

where `handler` is the address of the handler being exited.

See also: `exception_install()`

Note: The fact that an exception handler is called does not necessarily mean that the exception occurred. Multiple handlers can share the same exception. See Chapter 14: Exceptions on page 180 for more information.

callback_exception_install

Register a callback routine for exception install events

Definition:

```
#include <os21.h>
callback_excp_install_fn_t callback_exception_install(
  callback_excp_install_fn_t fn);
```

Arguments:

`callback_excp_install_fn_t fn`	Function to be called on exception handler install

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_exception_install()` registers a function to be called whenever an exception handler install occurs.

`callback_excp_install_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_excp_install_fn_t)(void* handler);
```

where `handler` is the address of the handler being installed.

See also: `exception_install()`

callback_exception_uninstall

Register a callback routine for exception delete events

Definition:

```
#include <os21.h>
callback_excp_uninstall_fn_t callback_exception_uninstall(
  callback_excp_uninstall_fn_t fn);
```

Arguments:

`callback_excp_uninstall_fn_t fn`

Function to be called on exception handler uninstall.

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_exception_uninstall()` registers a function to be called whenever an exception handler uninstall occurs.

`callback_excp_uninstall_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_excp_uninstall_fn_t)(void* handler);
```

where `handler` is the address of the handler being uninstalled.

See also: `exception_uninstall()`

callback_interrupt_enter

Register a callback routine for interrupt entry events

Definition:

```
#include <os21.h>
callback_intr_enter_fn_t callback_interrupt_enter(
  callback_intr_enter_fn_t fn);
```

Arguments:

`callback_intr_enter_fn_t fn`	Function to be called on interrupt handler entry

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_interrupt_enter()` registers a function to be called whenever an interrupt handler is called.

`callback_intr_enter_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_intr_enter_fn_t)(void* handler);
```

where `handler` is the address of handler being called.

See also: `interrupt_install()`

Note: The fact that an interrupt handler is called does not necessarily mean that the interrupt occurred. Multiple handlers can share the same interrupt. See Chapter 11: Interrupts on page 145 for more information.

callback_interrupt_exit

Register a callback routine for interrupt exit events

Definition:

```
#include <os21.h>
callback_intr_exit_fn_t callback_interrupt_exit(
  callback_intr_exit_fn_t fn);
```

Arguments:

`callback_intr_exit_fn_t fn`	Function to be called on interrupt handler exit

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_interrupt_exit()` registers a function to be called whenever an interrupt handler exits.

`callback_intr_exit_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_intr_exit_fn_t)(void* handler);
```

where `handler` is the address of the handler being exited.

See also: `interrupt_install()`

Note: The fact that an interrupt handler is called does not necessarily mean that the interrupt occured. Multiple handlers can share the same interrupt. See Chapter 11: Interrupts on page 145 for more information.

callback_interrupt_install

Register a callback routine for interrupt install events

Definition:

```
#include <os21.h>
callback_intr_install_fn_t callback_interrupt_install(
  callback_intr_install_fn_t fn);
```

Arguments:

`callback_intr_install_fn_t fn`

Function to be called on interrupt handler install

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_interrupt_install()` registers a function to be called whenever an interrupt handler install occurs.

`callback_intr_install_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_intr_install_fn_t)(void* handler);
```

where `handler` is the address of the handler being installed.

See also: `interrupt_install()`

callback_interrupt_uninstall

Register a callback routine for interrupt delete events

Definition:

```
#include <os21.h>
callback_intr_uninstall_fn_t callback_interrupt_uninstall(
  callback_intr_uninstall_fn_t fn);
```

Arguments:

`callback_intr_uninstall_fn_t fn`

Function to be called on interrupt handler uninstall.

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_interrupt_uninstall()` registers a function to be called whenever an interrupt handler uninstall occurs.

`callback_intr_uninstall_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_intr_uninstall_fn_t)(void* handler);
```

where `handler` is the address of the handler being uninstalled.

See also: `interrupt_uninstall()`

callback_task_create

Register a callback routine for task create events

Definition:
```
#include <os21.h>
callback_task_create_fn_t callback_task_create(
  callback_task_create_fn_t fn);
```

Arguments:

`callback_task_create_fn_t fn` Function to be called on task create

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_task_create()` registers a function to be called whenever a task create occurs.

`callback_task_create_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_task_create_fn_t)(task_t* new_task);
```

where `new_task` is the task being created.

See also: `task_create()`

callback_task_delete

Register a callback routine for task delete events

Definition:
```
#include <os21.h>
callback_task_delete_fn_t callback_task_delete(
  callback_task_delete_fn_t fn);
```

Arguments:

`callback_task_delete_fn_t fn` Function to be called on task delete

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_task_delete()` registers a function to be called whenever a task delete occurs.

`callback_task_delete_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_task_delete_fn_t)(task_t* task);
```

where `task` is the task being deleted.

See also: `task_delete()`

callback_task_exit

Register a callback routine for task exit events

Definition:

```
#include <os21.h>
callback_task_exit_fn_t callback_task_exit(
  callback_task_exit_fn_t fn);
```

Arguments:

`callback_task_exit_fn_t fn` Function to be called on task exit

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_task_exit()` registers a function to be called whenever a task exit occurs.

`callback_task_exit_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_task_exit_fn_t)(task_t* task);
```

where `task` is the task being exited.

See also: `task_exit()`

callback_task_switch

Register a callback routine for task switch events

Definition:

```
#include <os21.h>
callback_task_switch_fn_t callback_task_switch(
  callback_task_switch_fn_t fn);
```

Arguments:

`callback_task_switch_fn_t fn` Function to be called on task switch

Returns: Pointer to previously installed callback function. `NULL` if none.

Errors: None

Context: Callable from task or system context.

Description: `callback_task_switch()` registers a function to be called whenever a task switch occurs.

`callback_task_switch_fn_t` is defined in `callback.h` as follows:

```
typedef void (*callback_task_switch_fn_t)(
  task_t* old_task,
  task_t* new_task);
```

where:

`old_task` the task to switch from.

`new_task` the task to switch to.

Either `old_task` or `new_task` can be `NULL` to indicate that the CPU is either leaving or entering an idle state.

6 Semaphores

Semaphores provide a simple and efficient way to synchronize multiple tasks. They can also be used to ensure mutual exclusion and control access to a shared resource.

6.1 Semaphore overview

A semaphore structure `semaphore_t` contains two pieces of data:

- a count of the number of times the semaphore can be taken
- a queue of tasks waiting to take the semaphore

Semaphores are created using one of the following functions:

```
semaphore_t* semaphore_create_fifo (
  int value);
semaphore_t* semaphore_create_fifo_p(
  partition_t* partition,
  int value);
semaphore_t* semaphore_create_priority (
  int value);
semaphore_t* semaphore_create_priority_p (
  partition_t* partition,
  int value);
```

Note: *OS20 provides an API which differentiates between timeout and non-timeout semaphores at creation time. This is because OS20's target processor, the ST20, supports hardware semaphores. This feature is not generally available on other processors, hence this API is not present in OS21. To aid porting from OS20, OS21 presents this interface with a set of veneer macros in* `os21/semaphore.h`*. They are functionally equivalent to the standard OS21 non-timeout calls.*

The semaphores which OS21 provides differ in the way in which tasks are queued. Normally tasks are queued in the order in which they call `semaphore_wait()`. This is termed a FIFO semaphore. Semaphores of this type are created using `semaphore_create_fifo()` or `semaphore_create_fifo_p()` or by using one of the `_timeout` versions of these functions.

However, sometimes it is useful to allow higher priority tasks to jump the queue, so that they are blocked for a minimum amount of time. In this case a second type of semaphore can be used, a priority based semaphore. For this type of semaphore, tasks are queued based on their priority first, and the order which they call `semaphore_wait()` second. Semaphores of this type are created using `semaphore_create_priority()` or `semaphore_create_priority_p()`.

Semaphores may be acquired by the functions:

```
int semaphore_wait(
  semaphore_t* sem);
```

and

```
int semaphore_wait_timeout(
  semaphore_t* sem
  const osclock_t *timeout);
```

When a task wants to acquire a semaphore, it calls `semaphore_wait()`. If the semaphore count is greater than 0, then the count is decremented, and the task continues. If however, the count is already 0, then the task adds itself to the queue of tasks waiting for the semaphore and deschedules itself. Eventually another task should release the semaphore, and the first waiting task can continue. In this way, when the task returns from the function it has acquired the semaphore.

If you want to make certain that the task does not wait indefinitely for a particular semaphore then use `semaphore_wait_timeout()`, which enables a timeout to be specified. If this time is reached before the semaphore is acquired then the function returns and the task continues without acquiring the semaphore. Two special values may be specified for the timeout period.

`TIMEOUT_IMMEDIATE`	Causes the semaphore to be polled and the function to return immediately. The semaphore may or may not be acquired and the task continues.
`TIMEOUT_INFINITY`	Causes the function to behave the same as `semaphore_wait()`, that is, the task waits indefinitely for the semaphore to become available.

When a task wants to release the semaphore, it calls `semaphore_signal()`:

```
void semaphore_signal (semaphore_t* sem);
```

This looks at the queue of waiting tasks, and if the queue is not empty, removes the first task from the queue, and starts it running. If there are no tasks waiting, then the semaphore count is incremented, indicating that the semaphore is available.

An important use of semaphores is for synchronization between interrupt handlers and tasks. This is possible because while an interrupt handler cannot call `semaphore_wait()`, it can call `semaphore_signal()`, and so cause a waiting task to start running.

The current value of a semaphore may be queried with the following call:

```
int semaphore_value (semaphore_t* sem);
```

This returns the instantaneous value of the given semaphore. Note that the value returned may be out of date if the calling task is preempted by another task or interrupt service routine which modifies the semaphore.

6.2 Use of semaphores

Semaphores can be defined to allow a given number of tasks simultaneous access to a shared resource. The maximum number of tasks allowed is determined when the semaphore is initialized. When that number of tasks have acquired the resource, the next task to request access to it waits until one of those holding the semaphore relinquishes it.

Semaphores can protect a resource only if all tasks that wish to use the resource also use the same semaphore. It cannot protect a resource from a task that does not use the semaphore and accesses the resource directly.

Typically, semaphores are set up to allow at most one task access to the resource at any given time. This is known as using the semaphore in **binary mode**, where the count either has the value zero or one. This is useful for mutual exclusion or synchronization of access to shared data. Areas of code protected using semaphores are sometimes called **critical regions**.

When used for mutual exclusion the semaphore is initialized to 1, indicating that no task is currently in the critical region, and that at most one can be. The critical region is surrounded with calls to `semaphore_wait` at the start and `semaphore_signal` at the end. Therefore the first task which tries to enter the critical region successfully takes the semaphore, and any others are forced to wait. When the task currently in the critical region leaves, it releases the semaphore, and allows the first of the waiting tasks into the critical region.

Semaphores are also used for synchronization. Usually this is between a task and an interrupt handler, with the task waiting for the interrupt handler. When used in this way the semaphore is initialized to zero. The task then performs a `semaphore_wait()` on the semaphore, and deschedules. Later the interrupt handler performs a `semaphore_signal()`, which reschedules the task. This process can then be repeated, with the semaphore count never changing from zero.

All the OS21 semaphores can also be used in a **counting** mode, where the count can be any positive number. The typical application for this is controlling access to a shared resource, where there are multiple resources available. Such a semaphore allows N tasks simultaneous access to a resource and is initialized with the value N. Each task performs a `semaphore_wait()` when it wants a device. If a device is available the call returns immediately having decremented the counter. If no devices are available then the task is added to the queue. When a task has finished using a device it calls `semaphore_signal()` to release it.

6.3 Semaphore API summary

All the definitions related to semaphores can be accessed by including the header file `os21.h`, which itself includes the header file `semaphore.h`. See *Table 17*, *Table 18* and *Table 19* for a complete list.

Table 17. Functions defined in semaphore.h

Function	Description
`semaphore_create_fifo()`	Creates a FIFO queued semaphore
`semaphore_create_fifo_p()`	Creates a FIFO queued semaphore
`semaphore_create_priority()`	Creates a priority queued semaphore
`semaphore_create_priority_p()`	Creates a priority queued semaphore
`semaphore_delete()`	Deletes a semaphore
`semaphore_signal()`	Signals a semaphore
`semaphore_value()`	Gets the current value of a semaphore's count
`semaphore_wait_timeout()`	Waits for a semaphore or a timeout

Table 18. Types define in semaphore.h

Type	Description
`semaphore_t`	A semaphore

Table 19. Macros defined in semaphore.h

Macro	Description
`semaphore_create_fifo_timeout()`	Creates a FIFO queued semaphore
`semaphore_create_priority_timeout()`	Creates a priority queued semaphore
`semaphore_wait()`	Waits for a semaphore

All semaphore functions are callable from an OS21 task, however only `semaphore_signal()` and `semaphore_wait_timeout()` can be called from an interrupt service routine.

Note: When using `semaphore_wait_timeout()` *with in an interrupt service routine, the timeout value* ***must*** *be* `TIMEOUT_IMMEDIATE`*.*

6.4 Semaphore function definitions

semaphore_create_fifo

Create a FIFO queued semaphore

Definition:
```
#include <os21.h>
semaphore_t* semaphore_create_fifo(
  int value);
```

Arguments:

`int value`	The initial value of the semaphore

Returns: The address of an initialized semaphore, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the semaphore.

Context: Callable from task only.

Description: `semaphore_create_fifo()` creates a counting semaphore, initialized to `value`. The memory for the semaphore structure is allocated from the system heap. Semaphores created with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

See also: `semaphore_create_fifo_p`, `semaphore_create_priority`, `semaphore_create_priority_p`

semaphore_create_fifo_p

Create a FIFO queued semaphore

Definition:
```
#include <os21.h>
semaphore_t* semaphore_create_fifo_p(
  partition_t* partition,
  int value);
```

Arguments:

`partition_t* partition`	The partition in which to create the semaphore
`int value`	The initial value of the semaphore

Returns: The address of an initialized semaphore, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the semaphore.

Context: Callable from task only.

Description: `semaphore_create_fifo_p()` creates a counting semaphore, allocated from the given `partition`, and initialized to `value`. Semaphores created with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

See also: `semaphore_create_fifo`, `semaphore_create_priority`, `semaphore_create_priority_p`

semaphore_create_priority

Create a priority queued semaphore

Definition:

```
#include <os21.h>
semaphore_t* semaphore_create_priority(
  int value);
```

Arguments:

`int value`	The initial value of the semaphore

Returns: The address of an initialized semaphore, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the semaphore.

Context: Callable from task only.

Description: `semaphore_create_priority()` creates a counting semaphore, initialized to `value`. The memory for the semaphore structure is allocated from the system heap. Semaphores created with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `semaphore_signal()`, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

See also: `semaphore_create_fifo`, `semaphore_create_fifo_p`, `semaphore_create_priority_p`

semaphore_create_priority_p

Create a priority queued semaphore

Definition:

```
#include <os21.h>
semaphore_t* semaphore_create_priority_p(
  partition_t* partition,
  int value);
```

Arguments:

`partition_t* partition`	Partition in which to allocate semaphore
`int value`	The initial value of the semaphore

Returns: The address of an initialized semaphore, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the semaphore.

Context: Callable from task only.

Description: `semaphore_create_priority_p()` creates a counting semaphore, initialized to `value`. The memory for the semaphore structure is allocated from the specified memory `partition`. Semaphores created with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `semaphore_signal()`, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

See also: `semaphore_create_fifo`, `semaphore_create_fifo_p`, `semaphore_create_priority`

semaphore_delete

Delete a semaphore

Definition:

```
#include <os21.h>
int semaphore_delete(
  semaphore_t *sem);
```

Arguments:

`semaphore_t *sem`	Semaphore to delete

Returns: `OS21_SUCCESS` or `OS21_FAILURE`.

Errors: Fails if `sem` is `NULL`.

Context: Callable from task only.

Description: `semaphore_delete()` deletes the semaphore, `sem`.

Note: The results are undefined if a task attempts to use a semaphore once it has been deleted.

See also: `semaphore_create_priority`, `semaphore_create_fifo`, `semaphore_create_priority_p`, `semaphore_create_fifo_p`

semaphore_signal

Signal a semaphore

Definition:

```
#include <os21.h>
void semaphore_signal(
  semaphore_t* sem);
```

Arguments:

`semaphore_t* sem`	A pointer to a semaphore

Returns: None

Errors: None

Context: Callable from task or system context.

Description: `semaphore_signal()` performs a signal operation on the specified semaphore. The exact behavior of this function depends on the semaphore type. The operation checks the queue of tasks waiting for the semaphore, if the list is not empty, then the first task on the list is restarted, possibly preempting the current task. Otherwise the semaphore count is incremented, and the task continues running.

See also: `semaphore_wait`, `semaphore_wait_timeout`

semaphore_value

Return the instantaneous value of semaphore

Definition:

```
#include <os21.h>
int semaphore_value(
  semaphore_t* sem);
```

Arguments:

`semaphore_t* sem`	A pointer to a semaphore

Returns: The current value of the semaphore's count.

Errors: None

Context: Callable from task or system context.

Description: `semaphore_value()` returns the current value of the given semaphore's count. The value returned may be out of date if the calling task is preempted by another task or ISR which modifies the semaphore.

semaphore_wait

Wait for a semaphore

Definition:

```
#include <os21.h>
int semaphore_wait(
  semaphore_t* sem);
```

Arguments:

`semaphore_t* sem`	A pointer to a semaphore

Returns: Always returns `OS21_SUCCESS`.

Errors: None

Context: Callable from task only.

Description: `semaphore_wait()` performs a wait operation on the specified semaphore. The exact behavior of this function depends on the semaphore type. The operation checks the semaphore counter, and if it is 0, adds the current task to the list of queued tasks, before descheduling. Otherwise the semaphore counter is decremented, and the task continues running.

This function is implemented as a macro and evaluates to:

```
semaphore_wait_timeout(sem, TIMEOUT_INFINITY)
```

See also: `semaphore_signal`, `semaphore_wait_timeout`

semaphore_wait_timeout

Wait for a semaphore or a timeout

Definition:

```
#include <os21.h>
int semaphore_wait_timeout(
  semaphore_t* sem
  const osclock_t *timeout);
```

Arguments:

`semaphore_t* sem`	A pointer to a semaphore
`const osclock_t* timeout`	Maximum time to wait for the semaphore Expressed in ticks or as `TIMEOUT_IMMEDIATE` or `TIMEOUT_INFINITY`

Returns: Returns `OS21_SUCCESS` on success, `OS21_FAILURE` if timeout occurs.

Errors: None

Context: Callable from task or system context. Only valid from system context if `timeout` is `TIMEOUT_IMMEDIATE`.

Description: `semaphore_wait_timeout()` performs a wait operation on the specified semaphore (`sem`). If the time specified by the `timeout` is reached before a signal operation is performed on the semaphore, then `semaphore_wait_timeout()` returns the value `OS21_FAILURE` indicating that a timeout occurred, and the semaphore count is unchanged. If the semaphore is signalled before the timeout is reached, then `semaphore_wait_timeout()` returns `OS21_SUCCESS`.

Note: Timeout is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the following example.

The `timeout` value may be specified in ticks, which is an implementation dependent quantity. Two special time values may also be specified for `timeout`. `TIMEOUT_IMMEDIATE` causes the semaphore to be polled, that is, the function always returns immediately. This must be the value used if `semaphore_wait_timeout()` is called from an interrupt service routine. If the semaphore count is greater than zero, then it is successfully decremented, and the function returns `OS21_SUCCESS`, otherwise the function returns a value of `OS21_FAILURE`. A `timeout` of `TIMEOUT_INFINITY` behaves exactly as `semaphore_wait()`.

Example:

```
osclock_t time;
time = time_plus(time_now(), time_ticks_per_sec());
semaphore_wait_timeout(semaphore, &time);
```

See also: `semaphore_signal`, `semaphore_wait`

7 Mutexes

Mutexes provide a simple and efficient way to ensure mutual exclusion and control access to a shared resource.

7.1 Mutexes overview

A mutex structure `mutex_t` contains several pieces of data including:

- the current owning task
- a queue of tasks waiting to take the mutex

Mutexes are created using one of the following functions:

```
mutex_t* mutex_create_fifo(void);
mutex_t* mutex_create_fifo_p(partition_t* partition);
mutex_t* mutex_create_priority(void);
mutex_t* mutex_create_priority_p(partition_t* partition);
mutex_t* mutex_create_priority_noinherit(void);
mutex_t* mutex_create_priority_noinherit_p(
          partition_t* partition);
```

A mutex can be owned by only one task at time. In this sense they are like OS21 semaphores initialized with a count of 1 (also known as **binary semaphores**). Unlike semaphores, once a task owns a mutex, it can re-take it as many times as necessary, provided that it also releases it an equal number of times. In this situation binary semaphores would deadlock.

The mutexes which OS21 provide differ in the way in which tasks are queued when waiting for it. For FIFO mutexes tasks are queued in the order in which they call `mutex_lock()`. Mutexes of this type are created using `mutex_create_fifo()` or `mutex_create_fifo_p()`.

However, sometimes it is useful to allow higher priority tasks to jump the queue, so that they are blocked for a minimum amount of time. In this case, a second type of mutex can be used, a priority based mutex. For this type of mutex, tasks are queued based on their priority first, and the order in which they call `mutex_lock()` second. Mutexes of this type are created using `mutex_create_priority()` or `mutex_create_priority_p()`.

Mutex may be acquired by the functions:

```
void mutex_lock(mutex_t* mutex);
```

and

```
int mutex_trylock(mutex_t* mutex);
```

When a task wants to acquire a mutex, it calls `mutex_lock()`. If the mutex is currently unowned, or already owned by the same task, then the task gets the mutex and continues. If however, the mutex is owned by another task, then the task adds itself to the queue of tasks waiting for the mutex and deschedules itself. Eventually another task should release the mutex, and the first waiting task gets the mutex and can continue. In this way, when the task returns from the function it has acquired the mutex.

Note: The same task can acquire a mutex any number of times without deadlock, but it must release it an equal number of times.

To make certain that the task does not wait indefinitely for a mutex, use `mutex_trylock()`. This attempts to gain ownership of the mutex, but fails immediately if it is not available.

A task is automatically made immortal while it has ownership of a mutex.

When a task wants to release the mutex, it calls `mutex_release()`:

```
int mutex_release(mutex_t* mutex);
```

This looks at the queue of waiting tasks. If the queue is not empty, it removes the first task from the queue and, if it is not of a lower priority, it assigns ownership of the mutex to that task and makes it runnable. If there are no tasks waiting, then the mutex becomes free.

Note: If a task exits while holding a mutex, the mutex remains locked, and a deadlock is inevitable.

7.1.1 Priority inversion

Priority mutexes also provide protection against priority inversion. This can occur when a low priority task acquires a mutex, and then a high priority task tries to claim it. The high priority task is then forced to wait for the low priority task to release the mutex before it can proceed. If an intermediate priority task now becomes ready to run, it preempts the low priority task. A lower priority task (that is not holding the mutex in question) is therefore blocking the execution of a higher priority task, this is termed priority inversion. Priority mutexes are able to detect when this occurs, and correct the situation. This is done by temporarily boosting the low priority task's priority to be the same as the priority of the highest priority waiting task, all the while the low priority task owns the mutex.

Priority inversion detection occurs every time a task has to queue to get a priority mutex, every time a task releases a priority mutex, and every time a task changes priority.

OS21 also provides priority mutexes that do not protect against priority inversion. Mutexes of this type are created using `mutex_create_priority_noinherit()` or `mutex_create_priority_noinherit_p()`.

7.2 Use of mutexes

Mutexes can only be used to protect a resource if all tasks that wish to use the resource also use the same mutex. It cannot protect a resource from a task that does not use the mutex and accesses the resource directly.

Mutexes allow at most one task access to the resource at any given time. Areas of code protected using mutexes are sometimes called **critical regions**.

The critical region is surrounded with calls to `mutex_lock()` at the start and `mutex_release()` at the end. Therefore the first task which tries to enter the critical region successfully takes the mutex, and any others are forced to wait. When the task currently in the critical region leaves, it releases the mutex, and allows the first of the waiting tasks into the critical region.

7.3 Mutex API summary

All the definitions related to mutexes are in the single header file `os21.h`, which itself includes the header file `mutex.h`. See *Table 20* and *Table 21* for a complete list.

Table 20. Functions defined in mutex.h

Function	Description
`mutex_create_fifo()`	Creates a FIFO queued mutex
`mutex_create_fifo_p()`	Creates a FIFO queued mutex
`mutex_create_priority()`	Creates a priority queued mutex (with priority inheritance)
`mutex_create_priority_p()`	Creates a priority queued mutex (with priority inheritance)
`mutex_create_priority_ noinherit()`	Creates a priority queued mutex (without priority inheritance)
`mutex_create_priority_ noinherit_p()`	Creates a priority queued mutex (without priority inheritance)
`mutex_delete()`	Deletes a mutex
`mutex_lock()`	Acquires a mutex, block if not available
`mutex_release()`	Releases a mutex
`mutex_trylock()`	Try to get a mutex, fail if not available

Table 21. Types define in mutex.h

Type	Description
`mutex_t`	A mutex

All mutex functions are callable from OS21 tasks, and not from interrupt handlers.

7.4 Mutex function definitions

mutex_create_fifo

Create a FIFO queued mutex

Definition:
```
#include <os21.h>
mutex_t* mutex_create_fifo(void);
```

Arguments: None

Returns: The address of an initialized mutex, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the mutex.

Context: Callable from task only.

Description: `mutex_create_fifo()` creates a mutex. The memory for the mutex structure is allocated from the system heap. Mutexes created with this function have the usual mutex semantics, except that when a task calls `mutex_lock()` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

See also: `mutex_create_fifo_p`, `mutex_create_priority`

mutex_create_fifo_p

Create a FIFO queued mutex

Definition:
```
#include <os21.h>
mutex_t* mutex_create_fifo_p(
  partition_t* partition);
```

Arguments:

`partition_t* partition`	The partition in which to create the mutex

Returns: The address of an initialized mutex, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the mutex.

Context: Callable from task only.

Description: `mutex_create_fifo_p()` creates a mutex, allocated from the given `partition`. Mutexes created with this function have the usual mutex semantics, except that when a task calls `mutex_lock()` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

See also: `mutex_create_fifo`, `mutex_create_priority_p`

mutex_create_priority

Create a priority queued mutex (with priority inheritance)

Definition:
```
#include <os21.h>
mutex_t* mutex_create_priority(void);
```

Arguments: None

Returns: The address of an initialized mutex, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the mutex.

Context: Callable from task only.

Description: `mutex_create_priority()` creates a mutex. The memory for the mutex structure is allocated from the system heap. Mutexes created with this function have the usual mutex semantics, except that when a task calls `mutex_lock()` it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `mutex_release()`, it is guaranteed to be the task with the highest priority of all those waiting for the mutex.

Mutexes created with this function also guarantee to detect and correct priority inversion.

See also: `mutex_create_fifo`, `mutex_create_priority_noinherit`, `mutex_create_priority_p`

mutex_create_priority_p

Create a priority queued mutex (with priority inheritance)

Definition:

```
#include <os21.h>
mutex_t* mutex_create_priority_p(
  partition_t* partition);
```

Arguments:

`partition_t* partition`	The partition in which to create the mutex

Returns: The address of an initialized mutex, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the mutex.

Context: Callable from task only.

Description: `mutex_create_priority_p()` creates a mutex. The memory for the mutex structure is allocated from the specified memory partition. Mutexes created with this function have the usual mutex semantics, except that when a task calls `mutex_lock()` it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `mutex_release()`, it is guaranteed to be the task with the highest priority of all those waiting for the mutex.

Mutexes created with this function also guarantee to detect and correct priority inversion.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C run-time heap is used.*

See also: `mutex_create_fifo_p`, `mutex_create_priority`, `mutex_create_priority_noinherit_p`

mutex_create_priority_noinherit

Create a priority queued mutex (without priority inheritance)

Definition:
```
#include <os21.h>
mutex_t* mutex_create_priority_noinherit(void);
```

Arguments: None

Returns: The address of an initialized mutex, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the mutex.

Context: Callable from task only.

Description: `mutex_create_priority_noinherit()` creates a mutex. The memory for the mutex structure is allocated from the system heap. Mutexes created with this function have the usual mutex semantics, except that when a task calls `mutex_lock()` it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `mutex_release()`, it is guaranteed to be the task with the highest priority of all those waiting for the mutex.

Mutexes created with this function do not detect and correct priority inversion.

See also: `mutex_create_fifo`, `mutex_create_priority`, `mutex_create_priority_noinherit_p`

mutex_create_priority_noinherit_p

Create a priority queued mutex (without priority inheritance)

Definition:
```
#include <os21.h>
mutex_t* mutex_create_priority_noinherit_p(
  partition_t* partition);
```

Arguments:

`partition_t* partition`	The partition in which to create the mutex

Returns: The address of an initialized mutex, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the mutex.

Context: Callable from task only.

Description: `mutex_create_priority_noinherit_p()` creates a mutex. The memory for the mutex structure is allocated from the specified memory partition. Mutexes created with this function have the usual mutex semantics, except that when a task calls `mutex_lock()` it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `mutex_release()`, it is guaranteed to be the task with the highest priority of all those waiting for the mutex.

Mutexes created with this function do not detect and correct priority inversion.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C run-time heap is used.*

See also: `mutex_create_fifo_p`, `mutex_create_priority_noinherit`, `mutex_create_priority_p`

mutex_delete

Delete a mutex

Definition:

```
#include <os21.h>
int mutex_delete(
  mutex_t *mutex);
```

Arguments:

`mutex_t* mutex`	Mutex to delete

Returns: `OS21_SUCCESS` or `OS21_FAILURE`.

Errors: Fails if `mutex` is `NULL`.

Context: Callable from task only.

Description: `mutex_delete()` deletes the mutex, `mutex`.

Note: The results are undefined if a task attempts to use a mutex after it has been deleted.

See also: `mutex_create_priority`, `mutex_create_fifo`, `mutex_create_priority_p`, `mutex_create_fifo_p`

mutex_lock

Acquire a mutex, block if not available

Definition:

```
#include <os21.h>
void mutex_lock(
  mutex_t* mutex);
```

Arguments:

`mutex_t* mutex`	A pointer to a mutex

Returns: None

Context: Callable from task only.

Description: `mutex_lock()` acquires the given mutex. The exact behavior of this function depends on the mutex type. If the mutex is currently not owned, or is already owned by the task, then the task acquires the mutex, and carries on running. If the mutex is owned by another task, then the calling task is added to the queue of tasks waiting for the mutex, and deschedules.

Once the task acquires the mutex it is made immortal, until it releases the mutex.

See also: `mutex_release`, `mutex_trylock`, `task_immortal`

mutex_release

Release a mutex

Definition:

```
#include <os21.h>
int mutex_release(
  mutex_t* mutex);
```

Arguments:

`mutex_t* mutex`	A pointer to a mutex to release

Returns: `OS21_SUCCESS` or `OS21_FAILURE`

Errors: Returns `OS21_FAILURE` if the task releasing the mutex does not own it.

Context: Callable from task only.

Description: `mutex_release()` releases the specified mutex. The exact behavior of this function depends on the mutex type. The operation checks the queue of tasks waiting for the mutex, if the list is not empty, then the first task on the list is restarted and granted ownership of the mutex, possibly preempting the current task. Otherwise the mutex is released, and the task continues running.

If the releasing task had its priority temporarily boosted by the priority inversion logic, then once the mutex is released the task's priority is returned to its correct value.

Once the task has released the mutex, it is made mortal again.

See also: `mutex_lock`, `mutex_trylock`, `task_mortal`

mutex_trylock

Acquire a mutex, return immediately if not available

Definition:

```
#include <os21.h>
int mutex_trylock(
  mutex_t* mutex);
```

Arguments:

`mutex_t* mutex`	A pointer to a mutex

Returns: `OS21_SUCCESS` or `OS21_FAILURE`

Errors: Call fails if the mutex is currently owned by another task.

Context: Callable from task only.

Description: `mutex_trylock()` checks to see if the mutex is free or already owned by the current task, and acquires it if it is. If the mutex is not free, then the call fails and returns `OS21_FAILURE`.

If the task acquires the mutex it is automatically made immortal, until it releases the mutex.

See also: `mutex_release`, `mutex_lock`, `task_immortal`

8 Event flags

Event flags provide a means for OS21 tasks to synchronize with multiple events. Tasks are able to block, waiting for one or more events to occur. The occurrence of events can be signalled from both tasks and interrupt handlers.

8.1 Event flags overview

Event flags are managed by an event group structure, which is a container object describing a task wait queue and a collection of event flags. Each event flag corresponds to a single event, and is represented by an individual bit. When an event flag is set, it is said to be posted and the associated event is considered to have occurred. Otherwise, the event flag is said to be unposted, and the associated event is considered to have not yet occurred.

A task can wait for a conjunctive (AND) or disjunctive (OR) subset of events within one event group. Several tasks may be waiting on the same or different events within an event group. Waiting tasks are made runnable when the subset of events for which they are waiting occurs, or a timeout happens.

The number of event flags within an event group is implementation dependent, but is defined to be the same as the number of bits in an `unsigned int` on that platform. This typically yields 32 or 64 bits (event flags) per event group.

An `event_group_t` is created with one of the following functions:

```
event_group_t* event_group_create (
  event_option_t options);

event_group_t* event_group_create_p (
  event_option_t options);
```

The event flags within the newly created group are all initialized to the unposted state. The `options` specify whether or not the flags in this group automatically clear.

A task can wait for one or more event flags within an event group to be posted with the following calls:

```
int event_wait_any (
  event_group_t* event_group,
  const unsigned int in_mask,
  unsigned int* out_mask,
  const osclock_t* timeout );

int event_wait_all (
  event_group_t* event_group,
  const unsigned int in_mask,
  unsigned int* out_mask,
  const osclock_t* timeout );
```

`event_wait_any()` is used to perform a wait on a disjunctive subset of events.
`event_wait_all()` is used to perform a wait on a conjunctive subset of events.

in_mask is used to identify which event flags within the group the task wishes to wait for. out_mask points to a location in memory which receives the state of the event flags at the point the task was made runnable.

The timeout parameter is used to specify what type of timeout is required. If you want to make certain that the task does not wait indefinitely for a particular subset of events to occur, then set timeout to be the time at which the task stops waiting. The time specified is an absolute one, not a relative one. If this time is reached before the specified events have occurred, the function returns and the task continues. In this case, the out_mask contains the events of the subset specified which were posted (if any). Two special values may be specified for the timeout period:

TIMEOUT_IMMEDIATE	Causes the event flags to be polled and the function to return immediately, whatever the state of the event flags.
TIMEOUT_INFINITY	Causes the function to wait indefinitely for the events to be posted.

Note: If in_mask *is zero and timeout is* TIMEOUT_IMMEDIATE*, then this call can effectively poll the state of the event flags within the event group.*

Events are posted with the following function, which is callable by tasks and interrupt service routines:

```
void event_post(
  event_group_t* event_group,
  const unsigned int mask );
```

The events specified by mask are posted. If they satisfy the waiting conditions of any of the tasks waiting in the event group, then those tasks are made runnable. Following this call, the event flags specified by mask remain in their posted state until explicitly cleared by a call to the following function:

```
void event_clear(
  event_group_t* event_group,
  const unsigned int mask );
```

Any task which attempts to wait for event flags that are already in the posted state does not block, since the wait terminating condition is immediately satisfied.

When the event group is no longer required it can be deleted with the function:

```
void event_group_delete(
  event_group_t* event_group);
```

8.1.1 Uses for event flags

Event flags provide a useful mechanism for tasks to wait for one or more events to occur before proceeding. They can be used ‘point to point’, where one task or interrupt service routine communicates the occurrence of events to just one task, or they can be used to ‘broadcast’ events, where multiple tasks can wait on the occurrence of certain events.

It should be noted that event flags do not nest or count. An event flag that has been posted once is indistinguishable from one that has been posted many times.

8.2 Event API summary

All the definitions related to events can be obtained by including the header file `os21.h`, which itself includes the header file `event.h`. See *Table 22* and *Table 23* for a complete list.

Table 22. Functions defined in event.h

Function	Description
`event_clear()`	Clears a set of event flags
`event_group_create()`	Creates an event group
`event_group_create_p()`	Creates an event group
`event_group_delete()`	Deletes an event group
`event_post()`	Posts a set of event flags
`event_wait_all()`	Waits for a set of events to occur
`event_wait_any()`	Waits for a set of events to occur

Table 23. Types define in event.h

Type	Description
`event_option_t`	Flags to the event group create call
`event_group_t`	A group of event flags

8.3 Event function definitions

event_clear

Clear a subset of event flags within an event group

Definition:
```
#include <os21.h>
void event_clear(
  event_group_t* event_group,
  const unsigned int mask);
```

Arguments:

`event_group_t* event_group`	The event group in which to clear event flags
`unsigned int mask`	The event flags to clear

Returns: None

Errors: None

Context: Callable from task or system context.

Description: `event_clear()` sets the state of the subset of event flags specified by `mask` in the event group specified by `event_group` back to the unposted state.

See also: `event_post`, `event_wait_all`, `event_wait_any`

event_group_create

Create an event group

Definition:
```
#include <os21.h>
event_group_t* event_group_create(
  event_option_t options);
```

Arguments:

`event_option_t options`	Creation flags

Returns: The address of an initialized event group, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the event group.

Context: Callable from task only.

Description: `event_group_create()` creates an event group. The memory for the event group structure is allocated from the system heap.

The created event group contains `sizeof(int) * 8` event flags, all initialized to the unposted state.

`options` can be one of the following values:

– `0` the event flags stay posted until explicitly cleared by `event_clear()`,

– `event_auto_clear` the event flags are automatically cleared once they have been delivered to all the waiting threads.

See also: `event_group_create_p`, `event_group_delete`

event_group_create_p

Create an event group

Definition:

```
#include <os21.h>
event_group_t* event_group_create_p(
  partition_t* partition,
  event_option_t options);
```

Arguments:

`partition_t* partition`	The partition in which to create the event group
`event_option_t options`	Creation flags

Returns: The address of an initialized event group, or `NULL` if an error occurs.

Errors: `NULL` if there is insufficient memory for the event group.

Context: Callable from task only.

Description: `event_group_create_p()` creates an event group. The memory for the event group structure is allocated from the specified memory partition.

Note: If a null pointer is specified for `partition`*, instead of a valid partition pointer, the C runtime heap is used.*

The created event group contains `sizeof(int) * 8` event flags, all initialized to the unposted state.

`options` can be one of the following values:

– `0` the event flags stay posted until explicitly cleared by `event_clear()`,

– `event_auto_clear` the event flags are automatically cleared once they have been delivered to all the waiting threads.

See also: `event_group_create`, `event_group_delete`

event_group_delete

Delete an event group

Definition:

```
#include <os21.h>
int event_group_delete(
  event_group_t* event_group);
```

Arguments:

`event_group_t* event_group`	The event group to delete

Returns: `OS21_SUCCESS` or `OS21_FAILURE`.

Errors: Fails if `event_group` is `NULL`.

Context: Callable from task only.

Description: `event_group_delete()` deletes an event group, freeing the memory used for it back to the partition from which it was allocated.

Note: The results are undefined if a task attempts to use an event group once it has been deleted.

See also: `event_group_create`, `event_group_create_p`

event_post

Set the state of event flags to the posted state

Definition:

```
#include <os21.h>
void event_post(
  event_group_t* event_group,
  const unsigned int mask);
```

Arguments:

`event_group_t* event_group`	The event group in which to post events
`unsigned int mask`	The subset of events to post

Returns: None

Errors: None

Context: Callable from task or system context.

Description: `event_post()` sets the state of the event flags specified by `mask` within the event group given by `event_group` to the posted state. The event flags remain in the posted state until explicitly cleared by the function `event_clear()`.

Any tasks which were waiting on the event group, and whose termination condition is satisfied by the events posted, are made runnable by this call.

See also: `event_wait_all`, `event_wait_any`, `event_clear`

event_wait_all

Wait for a subset of events to be posted

Definition:

```
#include <os21.h>
int event_wait_all(
  event_group_t* event_group,
  const unsigned int in_mask,
  unsigned int* out_mask,
  const osclock_t* timeout );
```

Arguments:

`event_group_t* event_group`	Event group to wait on
`unsigned int in_mask`	Mask defining a subset of event flags to wait on
`unsigned int* out_mask`	Receives mask of posted events on waking
`osclock_t timeout`	Time limit of wait

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for a timeout

Errors: `OS21_FAILURE` is returned if the timeout is reached before the wait condition is satisfied, or the `event_group` parameter is `NULL`.

Context: Callable from task or system context. Only valid from system context if `timeout` is `TIMEOUT_IMMEDIATE`.

Description: `event_wait_all()` allows the calling task to synchronize with the specified subset of events. If all the event flags specified by `in_mask` are already in the posted state then the task returns immediately with `OS21_SUCCESS`. If the event group was created with autoclear semantics, then the event flags are also cleared. If this is not the case, then the calling task is suspended until either all the event flags specified by `in_mask` are simultaneously in the posted state, or the timeout limit specified by `timeout` is reached.

Note: `timeout` *is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the following example.*

The `timeout` value is specified in ticks, which is an implementation dependent quantity. Two special time values may be specified for `timeout`. `TIMEOUT_IMMEDIATE` causes the event flags to be polled, that is, the function always returns immediately. This must be the value used if `event_wait_all()` is called from an interrupt service routine. If the current state of the event flags within the event group satisfy the conditions given, then the function returns `OS21_SUCCESS`, otherwise the function returns a value of `OS21_FAILURE`. A `timeout` of `TIMEOUT_INFINITY` causes the function to exit only when the event flags satisfy the conditions specified.

When the caller returns from this function, the location in memory pointed to by `out_mask` contains the state of the event flags, before any autoclearing. If the function succeeded, the event flags passed out to `out_mask` are guaranteed to satisfy the conditions specified. If the call failed, then `out_mask` can be examined to determine which event flags are set (if any), and which are not. `out_mask` can be `NULL`, if the state of the flags is not required.

If in_mask is 0, then the call always returns immediately, passing back the current event flags state using out_mask if it is not NULL. This allows the state of the event flags to be polled.

Example: A task waits for up to one second for two events to occur:

```
#define EVENT_A 0x00000001
#define EVENT_B 0x00000002

osclock_t time;
event_group_t* my_event_group;
...
time = time_plus(time_now(), time_ticks_per_sec());
event_wait_all (
  my_event_group,
  EVENT_A | EVENT_B,
  &out_mask,
  &time);
```

See also: event_post, event_clear, event_wait_any

event_wait_any

Wait for a subset of events to be posted

Definition:

```
#include <os21.h>
int event_wait_any(
  event_group_t* event_group,
  const unsigned int in_mask,
  unsigned int* out_mask,
  const osclock_t* timeout );
```

Arguments:

`event_group_t* event_group`	Event group on which to wait
`unsigned int in_mask`	Mask defining a subset of event flags on which to wait
`unsigned int* out_mask`	Receives mask of posted events on waking
`osclock_t timeout`	Time limit of wait

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for a timeout

Errors: `OS21_FAILURE` is returned if the timeout is reached before the wait condition is satisfied, or `event_group` is `NULL`.

Context: Callable from task or system context. Only valid from system context if `timeout` is `TIMEOUT_IMMEDIATE`.

Description: `event_wait_any()` allows the calling task to synchronize with the specified subset of events. If any of the event flags specified by `in_mask` are already in the posted state then the task returns immediately with `OS21_SUCCESS`. If the event group was created with autoclear semantics, then the event flags are also cleared. If this is not the case, then the calling task is suspended until either at least one of the event flags specified by `in_mask` is in the posted state, or the timeout limit specified by `timeout` is reached.

Note: `timeout` *is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the following example.*

The `timeout` value is specified in ticks, which is an implementation dependent quantity. Two special time values may be specified for `timeout`. `TIMEOUT_IMMEDIATE` causes the event flags to be polled, that is, the function always returns immediately. This must be the value used if `event_wait_any()` is called from an interrupt service routine. If the current state of the event flags within the event group satisfy the conditions given, then the function returns `OS21_SUCCESS`, otherwise the function returns a value of `OS21_FAILURE`. A `timeout` of `TIMEOUT_INFINITY` causes the function to exit only when the event flags satisfy the conditions specified.

When the caller returns from this function, the location in memory pointed to by `out_mask` contains the state of the event flags, before any autoclearing. If the function succeeded, the event flags passed out to `out_mask` are guaranteed to satisfy the conditions specified. If the call failed, then `out_mask` can be examined to determine which event flags are set (if any), and which are not. `out_mask` can be set to `NULL`, if the state of the flags is not required.

If in_mask is 0, then the call always returns immediately, passing back the current event flags state using out_mask if it is not NULL. This allows the state of the event flags to be polled.

Example: A task waits for up to one second for either of two events to occur:

```
#define EVENT_A 0x00000001
#define EVENT_B 0x00000002

osclock_t time;
event_group_t* my_event_group;
...
time = time_plus(time_now(), time_ticks_per_sec());
event_wait_any (
  my_event_group,
  EVENT_A | EVENT_B,
  &out_mask,
  &time);
```

See also: event_post, event_clear, event_wait_all

9 Message handling

A message queue provides a buffered communication method for tasks. Message queues also provide a way to communicate without copying the data, which can save time.

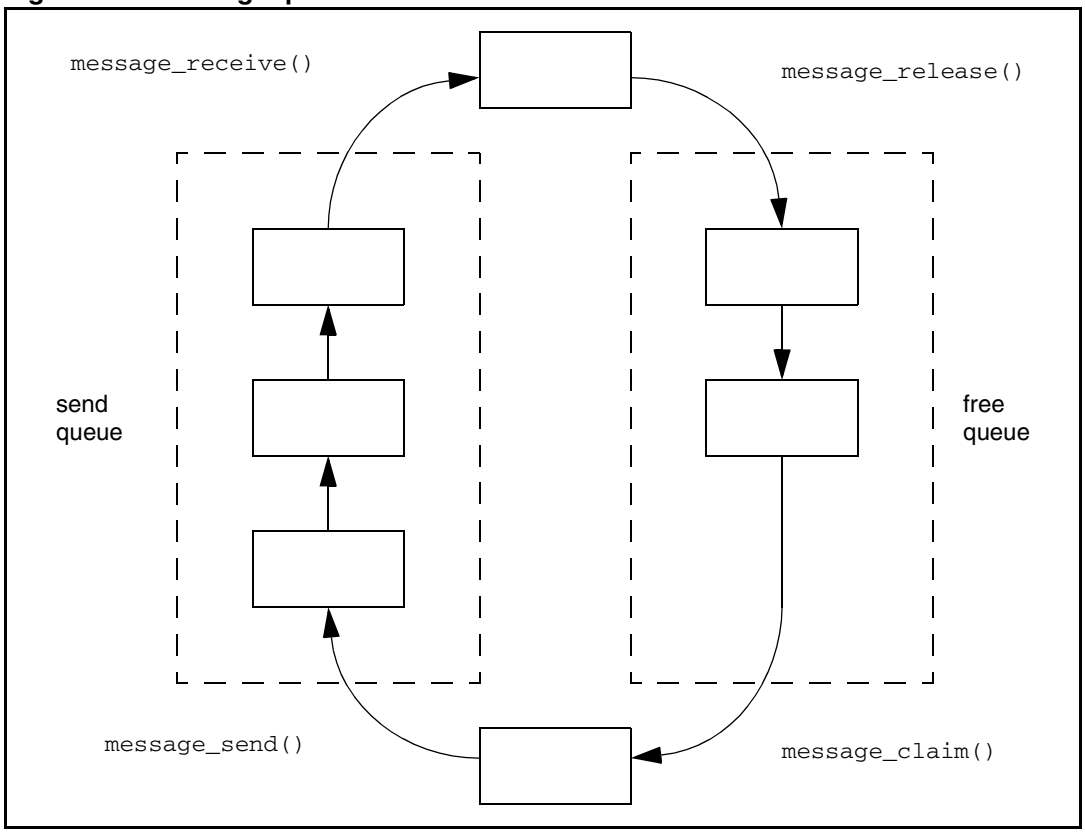
Note: Message queues are subject to a restriction when used from interrupt handlers. For interrupt handlers, use the timeout versions of the message handling functions with a timeout period of `TIMEOUT_IMMEDIATE` (see Section 9.3: Using message queues on page 131). This prevents the interrupt handler from blocking on a message claim.

9.1 Message queues

An OS21 message queue implements two queues of messages, one for message buffers which are currently not being used (known as the **free** queue), and the other holds messages which have been sent but not yet received (known as the **send** queue). Message buffers rotate between these queues, as a result of the user calling the various message functions.

Figure 1 shows the movement of messages between the two queues.

Figure 1. Message queues



9.2 Creating message queues

Message queues are created using one of the following functions:

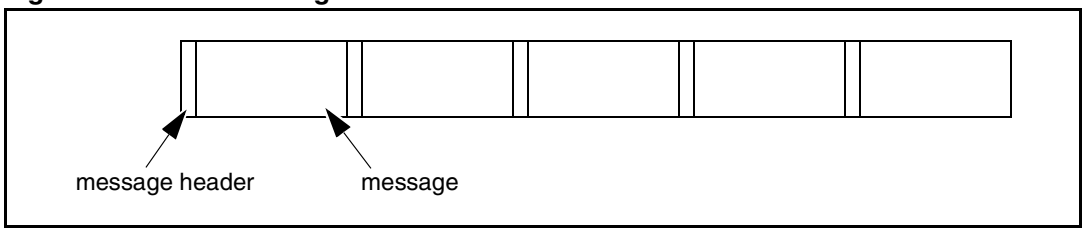
```
#include <os21.h>
message_queue_t* message_create_queue(
   size_t max_message_size,
   unsigned int max_messages );

message_queue_t* message_create_queue_p (
   partition_t* partition,
   partition_t* message_partition,
   size_t max_message_size,
   unsigned int max_messages );
```

Note: OS20 implements message queues created with the above calls using the ST20's hardware semaphores. Hardware semaphores provide no timeout facility, hence OS20 provides `_timeout()` *variants of message calls. Since OS21 does not support the notion of non-timeout or hardware semaphores, it does not provide the non* `_timeout()` *message API directly. These functions are provided as macros in the file* `os21/message.h` *to aid porting from OS20.*

These functions create a message queue for a fixed number of fixed sized messages, each message being preceded by a header, see *Figure 2*. The user must specify the maximum size for a message element and the total number of elements required.

Figure 2. OS21 message elements



`message_create_queue()` allocates the memory for the queue automatically from the system heap.

`message_create_queue_p()` allows the user to specify which partition to allocate the control structures and message buffers from.

9.3 Using message queues

Initially all the messages are on the free queue. The user allocates free message buffers by calling either of the following functions, which can then be filled in with the required data:

```
void* message_claim(
  message_queue_t* queue);

void* message_claim_timeout(
  message_queue_t* queue,
  const osclock_t* time);
```

Both functions claim the next available message in the message queue. `message_claim_timeout()` enables a timeout to be specified. If the timeout is reached before a message buffer is acquired then the function returns `NULL`. Two special values may be specified for the timeout period.

- `TIMEOUT_IMMEDIATE` causes the message queue to be polled and the function to return immediately. A message buffer may or may not be acquired and the task continues.
- `TIMEOUT_INFINITY` causes the function to behave the same as `message_claim()`, that is, the task waits indefinitely for a message buffer to become available.

When the message is ready it is sent by calling `message_send()`, at which point it is added to the send queue.

Messages are removed from the send queue by a task calling either of the functions:

```
void* message_receive(
  message_queue_t* queue );

void* message_receive_timeout(
  message_queue_t* queue,
  const osclock_t* time );
```

Both functions return the next available message. `message_receive_timeout()` provides a timeout facility which behaves in a similar manner to `message_claim_timeout()` in that it returns `NULL` if the message does not become available. If `TIMEOUT_IMMEDIATE` is specified the task continues whether or not a message is received and if `TIMEOUT_INFINITY` is specified the function behaves as `message_receive()` and waits indefinitely.

Finally when the receiving task has finished with the message buffer it should free it by calling `message_release()`. This function adds it to the free queue, where it is again available for allocation.

If the size of the message is variable, the user should specify that the message is `sizeof(void*)`, and then use pointers to the messages as the arguments to the message functions. The user is then responsible for allocating and freeing the real messages using whatever techniques are appropriate.

Message queues may be deleted by calling `message_delete_queue()`. If the message queue was created using `message_create_queue()` then this also frees the memory allocated for the message queue.

9.4 Message handling API summary

All the definitions related to messages can be accessed by including the header file `os21.h`, which itself includes the header file `message.h`. See *Table 24*, *Table 25* and *Table 26* for a complete list.

Table 24. Functions defined in message.h

Function	Description
`message_claim_timeout()`	Claims a message buffer with timeout
`message_create_queue()`	Creates a fixed size message queue
`message_create_queue_p()`	Creates a fixed size message queue
`message_delete_queue()`	Deletes a message queue
`message_receive_timeout()`	Receives the next available message from a queue or timeout
`message_release()`	Releases a message buffer
`message_send()`	Sends a message to a queue

Table 25. Types defined in message.h

Types	Description
`message_queue_t`	A message queue

Table 26. Macros defined in message.h

Macro	Description
`message_claim()`	Claims a message buffer
`message_create_queue_timeout()`	Creates a message queue
`message_create_queue_timeout_p()`	Creates a message queue
`message_receive()`	Receives the next available message from a queue with no timeout

9.5 Message function definitions

message_claim

Claim a message buffer

Definition:

```
#include <os21.h>
void* message_claim(
  message_queue_t* queue);
```

Arguments:

`message_queue_t* queue`	The message queue from which the message is claimed

Returns: The next available message buffer.

Errors: None

Context: Callable from task only.

Description: `message_claim()` claims the next available message buffer from the message queue, and returns its address. If no message buffers are currently available then the task blocks until one becomes available (by another task calling `message_release()`).

This function is not callable from an interrupt handler, and is equivalent to:

```
message_claim_timeout(queue, TIMEOUT_INFINITY)
```

See also: `message_receive`, `message_release`, `message_send`

message_claim_timeout

Claim a message buffer or timeout

Definition:

```
#include <os21.h>
void* message_claim_timeout(
  message_queue_t* queue
  const osclock_t* time);
```

Arguments:

`message_queue_t* queue`	The message queue from which the message is claimed
`const osclock_t* time`	The maximum time to wait for a message

Returns: The next available message buffer, or `NULL` if a timeout occurs.

Errors: None

Context: Callable from task or system context. Only valid from system context if `time` is `TIMEOUT_IMMEDIATE`.

Description: `message_claim_timeout()` claims the next available message buffer from the message queue, and returns its address. If no message buffers are currently available then the task blocks until one becomes available (by another task calling `message_release()`), or the time specified by `time` is reached.

Note: `time` *is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the following example.*

`time` is specified in ticks. A tick is an implementation dependent quantity.

Two special time values may also be specified for `time`. `TIMEOUT_IMMEDIATE` causes the message queue to be polled, that is, the function always returns immediately. If a message is available then it is returned, otherwise the function returns immediately with a result of `NULL`. A timeout of `TIMEOUT_INFINITY` behaves exactly as `message_claim()`.

`message_claim_timeout()` can be used from an interrupt handler, as long as `time` is `TIMEOUT_IMMEDIATE`.

Example:

```
osclock_t time;
time = time_plus(time_now(), time_ticks_per_sec());
message_claim_timeout(message_queue, &time);
```

See also: `message_receive_timeout`, `message_send`, `message_release`

message_create_queue

Create a fixed size message queue

Definition:

```
#include <os21.h>
message_queue_t* message_create_queue(
  size_t max_message_size,
  unsigned int max_messages);
```

Arguments:

`size_t max_message_size`	The maximum size of a message, in bytes
`unsigned int max_messages`	The maximum number of messages

Returns: The message queue identifier, or `NULL` on failure.

Errors: Returns `NULL` if there is insufficient memory for the message queue.

Context: Callable from task only.

Description: `message_create_queue()` creates a message queue with buffering for a fixed number of fixed size messages. Buffer space for the messages and the `message_queue_t` structure, is created automatically by the function from the system heap.

See also: `memory_allocate`, `message_claim`, `message_send`, `message_delete_queue`, `message_receive`, `message_release`

message_create_queue_p

Create a fixed size message queue

Definition:

```
#include <os21.h>
message_queue_t* message_create_queue_p(
  partition_t* partition,
  partition_t* message_partition,
  size_t max_message_size,
  unsigned int max_messages);
```

Arguments:

`partition_t* partition`	Where to allocate the control structures
`partition_t* message_partition`	Where to allocate the message buffers
`size_t max_message_size`	The maximum size of a message, in bytes
`unsigned int max_messages`	The maximum number of messages

Returns: The message queue identifier, or `NULL` on failure.

Errors: Returns `NULL` if there is insufficient memory for the message queue.

Context: Callable from task only.

Description: `message_create_queue_p()` creates a message queue with buffering for a fixed number of fixed size messages. Buffer space for the messages and the `message_queue_t` structure, is created automatically by the function calling `memory_allocate()` on the specified memory partitions.

Note: If a null pointer is specified for `partition` *or* `memory_partition`*, instead of a valid partition pointer, the C runtime heap is used.*

See also: `memory_allocate`, `message_claim`, `message_send`, `message_delete_queue`, `message_receive`, `message_release`, `message_create_queue`

message_delete_queue

Delete a message queue

Definition:

```
#include <os21.h>
int message_delete_queue(
  message_queue_t* message_queue);
```

Arguments:

`message_queue_t* message_queue`	The message queue to be deleted

Returns: `OS21_SUCCESS` or `OS21_FAILURE`.

Errors: Fails if `message_queue` is `NULL`.

Context: Callable from task only.

Description: `message_delete_queue()` deletes the message queue, `message_queue`, and frees the memory allocated for it.

Note: The results are undefined if a task attempts to use a message queue after it has been deleted.
Tasks using `message_claim_timeout()` *or* `message_receive_timeout()` *to wait on the message queue are protected from this possibility by a timeout period, which enables the task to continue.*

See also: `message_create_queue`, `message_create_queue_p`, `message_create_queue_timeout`

message_receive

Receive the next available message from a queue

Definition:

```
#include <os21.h>
void* message_receive(
  message_queue_t* queue);
```

Arguments:

`message_queue_t* queue`	The message queue that delivers the message

Returns: The next available message from the queue.

Errors: None

Context: Callable from task only.

Description: `message_receive()` receives the next available message from the message queue, and returns its address. If no messages are currently available then the task blocks until one becomes available (by another task calling `message_send()`).

See also: `message_claim`, `message_receive_timeout`, `message_release`, `message_send`

message_receive_timeout

Receive the next available message from a queue or timeout

Definition:

```
#include <os21.h>
void* message_receive_timeout(
  message_queue_t* queue
  const osclock_t* time);
```

Arguments:

`message_queue_t* queue`	The message queue that delivers the message
`const osclock_t* time`	The maximum time to wait for a message

Returns: The next available message from the queue, or `NULL` if a timeout occurs.

Errors: None

Context: Callable from task or system context. Only valid from system context if `time` is `TIMEOUT_IMMEDIATE`.

Description: `message_receive_timeout()` receives the next available message from the message queue, and returns its address. If no messages are currently available then the task blocks until one becomes available (by another task calling `message_send()`), or the time specified by `time` is reached.

Note: `time` *is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the example.*

`time` is specified in ticks. A tick is an implementation dependent quantity.

Two special time values may also be specified for `time`. `TIMEOUT_IMMEDIATE` causes the message queue to be polled, that is, the function always returns immediately. If a message is available then it is returned, otherwise the function returns immediately with a result of `NULL`. A timeout of `TIMEOUT_INFINITY` behaves exactly as `message_receive()`.

This function can only be used from an interrupt handler if `TIMEOUT_IMMEDIATE` is specified for `time`.

Example:

```
osclock_t time;
time = time_plus(time_now(), time_ticks_per_sec());
message_receive_timeout(message_queue, &time);
```

See also: `message_claim`, `message_receive`, `message_release`, `message_send`

message_release

Release a message buffer

Definition:

```
#include <os21.h>
void message_release(
  message_queue_t* queue,
  void* message);
```

Arguments:

`message_queue_t* queue`	The message queue to which the message is released
`void* message`	The message buffer

Returns: None

Errors: None

Context: Callable from task or system context.

Description: `message_release()` returns a message buffer to the message queue's free list. This function should be called when a message buffer (received by `message_receive()`) is no longer required. If a task is waiting for a free message buffer (by calling `message_claim()`) this causes the task to be restarted and the message buffer returned.

See also: `message_claim`, `message_receive`, `message_send`

message_send

Send a message to a queue

Definition:

```
#include <os21.h>
void message_send(
  message_queue_t* queue,
  void* message);
```

Arguments:

`message_queue_t* queue`	The message queue to which the message is sent
`void* message`	The message to send

Returns: None

Errors: None

Context: Callable from task or system context.

Description: `message_send()` sends the specified message to the message queue. This adds the message to the end of the queue of sent messages, and if any tasks are waiting for a message they are rescheduled and the message returned.

See also: `message_claim`, `message_receive`, `message_release`

10 Real-time clocks

Time is a very important issue for real-time systems. OS21 provides some basic functions for manipulating quantities of time.

Historically, OS20 regarded time as circular. That is, the counters which represent time could wrap round, with half the time period being in the future, and half of it in the past. This behavior meant that clock values had to be interpreted with care, and manipulated using time functions which took account of wrapping. These functions were used to:

- add and subtract quantities of time
- determine if one time is after another
- return the current time

OS21 maintains compatibility with the existing OS20 time API, but has effectively removed the notion of wrapping time by extending the range of the data types used to represent clock ticks. Time is represented in clock ticks, with the `osclock_t` type. This is defined to be a signed 64-bit integer. Even if the hardware driving the system clock has a one nanosecond cycle time, this representation of time does not wrap for 293 years; sufficiently far enough away to be discounted. Using this representation of time for 'absolute' time means that only positive values have any meaning. When subtracting two absolute times for purposes of comparison, a negative result means that one time is before the other and a positive result means that it is after the other.

10.1 Reading the current time

The value of system time is read using `time_now()`.

```
#include <os21.h>
osclock_t time_now(void);
```

The time at which counting starts is no later than the call to `kernel_start()`.

10.2 Time arithmetic

Arithmetic on timer values in OS21 can be performed directly by the application since wrapping can be discounted. OS21 still provides the old OS20 time manipulation functions for backwards compatibility. These routines perform no overflow checking and so allow for timer values 'wrapping round' to the most negative integer on the next tick after the most positive integer.

```
osclock_t time_plus(
   const osclock_t time1,
   const osclock_t time2);

osclock_t time_minus(
   const osclock_t time1,
   const osclock_t time2);

int time_after(
   const osclock_t time1,
   const osclock_t time2);
```

`time_plus()` adds two timer values together and returns the sum. For example, if a certain number of ticks is added to the current time using `time_plus()` then the result is the time after that many ticks.

`time_minus()` subtracts the second value from the first and returns the difference. For example, if one time is subtracted from another using `time_minus()` then the result is the number of ticks between the two times. If the result is positive then the first time is after the second. If the result is negative then the first time is before the second.

`time_after()` determines whether the first time is after the second time. The first time is considered to be after the second time if the result of subtracting the second time from the first time is positive. The function returns the integer value `1` if the first time is after the second, otherwise it returns `0`.

The precise time for a tick is implementation specific, but should be in the order of 1 to 10 microseconds. This would give a roll over time on a 63-bit counter of between 293,274 and 2,932,747 years.

For a discussion on the implementation of timers on specific targets, see the *OS21 implementation specific documentation*.

10.3 Time API summary

All the definitions related to time can be accessed by including the header file `os21.h`, which itself includes the header file `ostime.h`. See *Table 27* and *Table 28* for a complete list.

Table 27. Functions defined in ostime.h

Function	Description
`time_after()`	Returns whether one time is after another
`time_minus()`	Subtracts two clock values
`time_now()`	Returns the current time
`time_plus()`	Adds two clock values
`time_ticks_per_sec()`	Returns the number of ticks per second

Table 28. Types defined by ostime.h

Type	Description
`osclock_t`	Number of processor clock ticks

10.4 Timer function definitions

time_after

Return whether one time is after another

Definition:

```
#include <os21.h>
int time_after(
  const osclock_t time1,
  const osclock_t time2);
```

Arguments:

`const osclock_t time1`	A clock value, returned by `time_now`, for example
`const osclock_t time2`	A clock value, returned by `time_now`, for example

Returns: Returns `1` if `time1` is after `time2`, otherwise `0`.

Errors: None

Context: Callable from task or system context.

Description: `time_after()` returns the relationship between `time1` and `time2`. Returns `1` if `time1` is after `time2`, otherwise `0`.

See also: `time_minus`, `time_now`, `time_plus`

time_minus

Subtract two clock values

Definition:

```
#include <os21.h>
osclock_t time_minus(
  const osclock_t time1,
  const osclock_t time2);
```

Arguments:

`const osclock_t time1`	A clock value, returned by `time_now`, for example
`const osclock_t time2`	A clock value, returned by `time_now`, for example

Returns: Returns the result of subtracting `time2` from `time1`.

Errors: None

Context: Callable from task or system context.

Description: `time_minus()` subtracts one clock value from another. A negative time represents several clock ticks into the past.

See also: `time_plus`

time_now

Return the current time

Definition:
```
#include <os21.h>
osclock_t time_now(void);
```

Arguments: None

Returns: Returns the number of ticks since the system started.

Errors: None

Context: Callable from task or system context.

Description: `time_now()` returns the number of ticks since the system started running. The exact time at which counting starts is implementation specific, but it is no later than the call to `kernel_start()`.

The units of ticks is an implementation dependent quantity, but it is in the range of 1 to 10 microseconds.

See also: `task_delay`

time_plus

Add two clock values

Definition:
```
#include <os21.h>
osclock_t time_plus(
  const osclock_t time1,
  const osclock_t time2);
```

Arguments:

`const osclock_t time1`	A clock value, returned by `time_now`, for example
`const osclock_t time2`	A clock value, returned by `time_now`, for example

Returns: Returns the result of adding `time1` to `time2`.

Errors: None

Context: Callable from task or system context.

Description: `time_plus()` adds one clock value to another.

See also: `time_minus`

time_ticks_per_sec

Returns the number of clock ticks per second

Definition:
```
#include <os21.h>
osclock_t time_ticks_per_sec(void);
```

Arguments: None

Returns: The number of ticks.

Errors: None

Context: Callable from task or system context.

Description: `time_ticks_per_sec()` returns the number of clock ticks per second.

11 Interrupts

Interrupts provide a mechanism for external events to control the CPU. Normally, as soon as an interrupt is asserted, the CPU stops executing the current task, and starts executing the interrupt handler for that interrupt. In this way the program is made aware of external changes as soon as they occur. This switch is performed completely in hardware, and so is extremely rapid. Similarly when the interrupt handler has completed, the CPU resumes execution of the interrupted task, which is unaware that it has been interrupted.

The interrupt handler which the CPU executes in response to the interrupt is called the first level interrupt handler. This piece of code is supplied as part of OS21, and sets up the environment so that a normal C function can be called. The OS21 API enables a different user function to be associated with each interrupt, and this is called when the interrupt occurs. Each interrupt also has a parameter associated with it, which is passed into the function when it is called. This allows the same code to be shared between different interrupt handlers.

OS21 differentiates each interrupt by assigning it a unique name (`interrupt_name_t`).

11.1 Chip variants

Each version of the CPU can have its own set of peripherals, and these peripherals are allocated interrupts. There is no guarantee that the assignments will remain the same from variant to variant. To accommodate this, OS21 requires a table of definitions which describes the interrupt mappings for a given part. This table is provided to the OS21 kernel using the board support package (BSP) mechanism. The BSP is a library containing target specifics, which is linked with the application and the OS21 kernel at final link time. OS21 is shipped with BSP libraries for all supported variants.

Providing the source for the board support packages enables users to limit the number of declared interrupts to just those used by the application, and so save memory, if necessary.

OS21 uses the interrupt description table from the BSP to build its own table which is used to dispatch interrupts from the first level interrupt handler.

Along with the target specific OS21 interrupt code, the BSP describes the interrupt system to OS21 and together they implement the generic interrupt API.

11.2 Initializing the interrupt handling subsystem

Before interrupts can be used, the OS21 interrupt subsystem must be initialized. This is done using the BSP, see the *OS21 implementation specific documentation*.

11.3 Obtaining a handle for an interrupt

OS21 abstracts the concept of an interrupt behind a type called "interrupt_t". Before programming an interrupt you must therefore obtain the appropriate handle for the interrupt. The following example shows how to do this for an interrupt called MY_INTERRUPT.

```
extern interrupt_name_t MY_INTERRUPT;
       /* This is specified in the BSP tables */
interrupt_t * my_interrupt_handle_ptr;
       /* This is my handle to my interrupt */

my_interrupt_handle_ptr = interrupt_handle (MY_INTERRUPT);
if (!my_interrupt_handle_ptr)
{
  printf ("ERROR: Failed to obtain handle for my interrupt\n");
}
```

Once a valid interrupt handle is obtained you can for example, attach an interrupt handler, enable it and change its priority.

Note: It is not necessary to declare MY_INTERRUPT *as an* extern*. Instead you can include the appropriate header file.*

```
#include <os21/processor/variant.h>
```

For example:

```
#include <os21/st40/st40gx1.h>
```

However doing this makes code non-portable, since the name of the include file has to be changed when the same code is ported to a different platform.

11.4 Attaching interrupt handlers

OS21 supports both shareable and nonshareable interrupts.

OS21 defines an interrupt handler as follows:

```
typedef int (*interrupt_handler_t)(void * param);
```

An interrupt handler must return OS21_SUCCESS if it successfully identified and handled an interrupt or OS21_FAILURE if it did not.

```
int example_interrupt_handler (void * param)
{
  if (this_is_my_interrupt)
  {
    ... handle and clear the interrupt
    return (OS21_SUCCESS);
  }
  return (OS21_FAILURE);
}
```

11.4.1 Attaching an interrupt handler to a nonshared interrupt

An interrupt handler is attached to an interrupt, using the `interrupt_install()` function:

```
int result;

result = interrupt_install (my_interrupt_handle_ptr, my_handler,
my_handler_parameter);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to attach handler for my interrupt\n");
}
```

The handler can be uninstalled as follows:

```
int result;

result = interrupt_uninstall (my_interrupt_handle_ptr);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to detach handler for my interrupt\n");
}
```

11.4.2 Attaching an interrupt handler to a shared interrupt

OS21 provides a mechanism for more than one interrupt handler to share an interrupt. Handlers are chained by `interrupt_install_shared()` which adds an interrupt handler to the chain of handlers for a given interrupt. This increases interrupt latency, but may be required where different hardware interrupts are routed to the same interrupt vector.

When the interrupt occurs OS21 automatically calls all the handlers in the chain until one of them returns `OS21_SUCCESS` (indicating that it handled the interrupt). If no handlers in the chain return `OS21_SUCCESS` then OS21 panics, because the interrupt is unhandled.

The following shows how to install two handlers that share an interrupt pointed to by `ip`.

```
int result1, result2;
result1 = interrupt_install_shared (ip, handler1, param1);
if (result1 != OS21_SUCCESS)
{
  printf ("ERROR: Failed to install shared handler number 1\n");
}
result2 = interrupt_install_shared (ip, handler2, param2);
if (result2 != OS21_SUCCESS)
{
  printf ("Failed to install shared handler number 2\n");
}
```

The following shows how to uninstall two handlers that share an interrupt pointed to by `ip`.

```
int result1, result2;
result1 = interrupt_uninstall_shared (ip, handler1, param1);
if (result1 != OS21_SUCCESS)
{
  printf ("ERROR: Failed to uninstall shared handler number 1\n");
}
```

```
result2 = interrupt_uninstall_shared (ip, handler2, param2);
if (result2 != OS21_SUCCESS)
{
  printf ("ERROR: Failed to uninstall shared handler number 2\n");
}
```

11.5 Interrupt priority

It may be possible to program the priority of a given interrupt. If possible this can be done using the following functions:

```
result = interrupt_priority_set (my_interrupt_handle_ptr,
required_priority);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to set priority of my interrupt\n");
}

result = interrupt_priority (my_interrupt_handle_ptr,
&current_priority);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to get priority of my interrupt\n");
}
printf ("Current priority is %d\n", current_priority);
```

11.6 Enabling and disabling interrupts

Interrupts are enabled at the interrupt controllers using the `interrupt_enable()` function and can be disabled using the corresponding `interrupt_disable()` function:

```
result = interrupt_enable (my_interrupt_handle_ptr)
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to enable my interrupt\n");
}

result = interrupt_disable (my_interrupt_handle_ptr)
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to disable my interrupt\n");
}
```

11.7 Clearing interrupts

It is possible to clear an interrupt at the interrupt controllers. This can be achieved using the following API:

```
result = interrupt_clear (my_interrupt_handle_ptr);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to clear my interrupt\n");
}
```

11.8 Polling interrupts

A function is provided for polling interrupts if required:

```
result = interrupt_poll (my_interrupt_handle_ptr, &value);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to poll my interrupt\n");
}
printf ("My interrupt is %s\n", value ? "high" : "low");
```

11.9 Raising interrupts

It may be possible to raise interrupts from software for testing or signalling purposes (if hardware support is available). Similarly it may be possible to unassert a software interrupt. These can be achieved at the interrupt controller using the following API:

```
result = interrupt_raise (my_interrupt_handle_ptr);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to raise my interrupt\n");
}

result = interrupt_unraise (my_interrupt_handle_ptr);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to unraise my interrupt\n");
}
```

11.10 Masking interrupts

Interrupts can be selectively masked and unmasked from the CPU with the interrupt masking API.

```
int previous_level;

previous_level = interrupt_mask (new_level);
/* Interrupts are now masked to the new level. */
interrupt_unmask (previous_level);
/* Interrupts are now masked to the previous level. */
```

or

```
previous_level = interrupt_mask_all ();
/* Interrupts are now masked completely. */
interrupt_unmask (previous_level);
/* Interrupts are now masked to the previous level. */
```

These functions must always be called in pairs, `interrupt_mask()` and `interrupt_unmask()`, or `interrupt_mask_all()` and `interrupt_unmask()`. By raising the processor's interrupt priority level, `interrupt_mask()` is able to block all interrupts up to and including that level from reaching the CPU, where as `interrupt_mask_all()` is able to block all interrupts from reaching the CPU.

Interrupts of a higher priority than that specified are not masked so some higher level interrupts can still be serviced while `interrupt_mask()` is in effect. `interrupt_mask()`, `interrupt_mask_all()` and `interrupt_unmask()` can be used to create a critical region around code which, for instance, has to manipulate a data structure shared with an interrupt handler.

A task must not deschedule with interrupts masked, as this causes the scheduler to fail. When interrupts are masked, calling any function that may not be called by an interrupt service routine is illegal.

Once `interrupt_mask()` and `interrupt_mask_all()` are called from task context, no preemption can occur. Similarly pre-emption is re-enabled once `interrupt_unmask()` finally restores priority back to the base level.

11.11 Contexts and interrupt handler code

Code running under OS21 may run in one of two environments (or contexts). These are called task context and system context. OS21 interrupt handlers are run from system context.

The main difference between system context and task context is that code running in system context is not allowed to block. Undefined behavior occurs if code running in system context blocks. As a result of this contraint, code running from system context should **never** call an OS21 function that may block. Please refer to the individual function descriptions for details of which contexts the OS21 functions may be run from.

11.12 Interrupt API summary

All the definitions related to interrupts can be obtained by including the header file `os21.h`, which itself includes the header file `interrupt.h`. See *Table 29* and *Table 30* for a complete list.

Table 29. Functions defined in interrupt.h

Function	Description
`interrupt_clear()`	Clears an interrupt
`interrupt_disable()`	Disables an interrupt
`interrupt_enable()`	Enables an interrupt
`interrupt_handle()`	Obtains an `interrupt_t *` for an interrupt
`interrupt_install()`	Installs an interrupt handler
`interrupt_install_shared()`	Installs an interrupt handler and mark the interrupt source as shared
`interrupt_lock()`	Disables all interrupts (deprecated)
`interrupt_mask()`	Raises the processor's interrupt level
`interrupt_mask_all()`	Raises the processor's interrupt level to the maximum possible
`interrupt_poll()`	Polls an interrupt
`interrupt_priority()`	Gets an interrupt's priority
`interrupt_priority_set()`	Sets an interrupt's priority
`interrupt_raise()`	Raises an interrupt
`interrupt_uninstall()`	Uninstalls an interrupt handler
`interrupt_uninstall_shared()`	Uninstalls an interrupt handler where the interrupt source is shared
`interrupt_unlock()`	Enables all interrupts (deprecated)
`interrupt_unmask()`	Lowers the processor's interrupt level
`interrupt_unraise()`	Unraises an interrupt

Table 30. Types defined in interrupt.h

Type	Description
`interrupt_t`	An abstract interrupt type
`interrupt_init_flags_t`(1)	Interrupt initialization flags
`interrupt_handler_t`	An interrupt handler
`interrupt_name_t`	A name for an interrupt source
`ilc_mode_t`(1)	Modes for an Interrupt Level Controller

1. This type is only used by the BSP.

11.13 Interrupt function definitions

interrupt_clear

Clears an interrupt

Definition:
```
#include <os21.h>
int interrupt_clear(
  interrupt_t * ip);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt cannot be cleared or is invalid.

Context: Callable from task or system context.

Description: Attempts to clear the specified interrupt. Many interrupts are automatically cleared when the hardware stops asserting them. Some interrupt controllers however latch the interrupt, these have to be cleared otherwise the interrupt remains asserted, causing the processor to take an unwanted interrupt.

See also: `interrupt_raise`

interrupt_disable

Disable an interrupt

Definition:
```
#include <os21.h>
int interrupt_disable(
  interrupt_t * ip);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt cannot be disabled or is invalid.

Context: Callable from task or system context.

Description: Attempts to disable the specified interrupt. Some interrupts cannot be disabled and in this case the call fails.

See also: `interrupt_enable`

interrupt_enable

Enable an interrupt

Definition:

```
#include <os21.h>
int interrupt_enable(
  interrupt_t * ip);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt is invalid.

Context: Callable from task or system context.

Description: Attempts to enable the specified interrupt. Some interrupts are always enabled and in this case the call always succeeds.

See also: `interrupt_disable`

interrupt_handle

Obtains an interrupt handler for a named interrupt.

Definition:

```
#include <os21.h>
interrupt_t * interrupt_handle (
  interrupt_name_t name);
```

Arguments:

`interrupt_name_t name`	The name of the interrupt for which a handle is required.

Returns: Returns a valid interrupt handle for success, or `NULL` for failure.

Errors: `NULL` if the interrupt name could not be found.

Context: Callable from task only.

Description: Obtains a handle for a named interrupt. Once an `interrupt_t *` handle has been obtained, it can then be passed to the other functions to carry out actions on that interrupt.

interrupt_install

Install an interrupt handler and mark the interrupt source as nonshareable

Definition:

```
#include <os21.h>
int interrupt_install (
  interrupt_t * ip,
  interrupt_handler_t handler,
  void * param);
```

Arguments:

`interrupt_t * ip`	The handle of the interrupt for which a handler is to be installed.
`interrupt_handler_t handler`	The handler function which is called when the interrupt is taken.
`void * param`	A parameter which is passed to the handler when it is called.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the interrupt source `ip` is invalid, is in sharing mode (that is, a call to `interrupt_install_shared()` has already been made for `ip`), or a handler has already been installed for this interrupt source.

Context: Callable from task only.

Description: This installs the specified user interrupt handler for the interrupt source described by `ip`. The handler function is called with its the single parameter set to `param`. The user handler should return `OS21_SUCCESS` if it handled the interrupt, otherwise it should return `OS21_FAILURE`. This call allows a single interrupt handler to be registered for the given source.

Once a handler has been registered with this call, further calls to `interrupt_install()` or `interrupt_install_shared()` for this `ip` will fail.

See also: `interrupt_install_shared`, `interrupt_uninstall`, `interrupt_uninstall_shared`

interrupt_install_shared

Install an interrupt handler and mark the interrupt source as shared

Definition:

```
#include <os21.h>
int interrupt_install_shared (
  interrupt_t * ip,
  interrupt_handler_t handler,
  void * param);
```

Arguments:

`interrupt_t * ip`	The handle of the interrupt for which a handler is to be installed.
`interrupt_handler_t handler`	A handler function which is called when the interrupt is taken.
`void * param`	A parameter which is passed to the handler when it is called.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the interrupt source `ip` is invalid or is in non-sharing mode (that is, a call to `interrupt_install()` has already been made for `ip`).

Context: Callable from task only.

Description: This installs the specified user interrupt handler for the interrupt source described by `ip`. The handler function is called with its the single parameter set to `param`. The user handler returns `OS21_SUCCESS` if it handled the interrupt, otherwise it returns `OS21_FAILURE`. This call allows a multiple interrupt handler to be registered for the given source, therefore allowing interrupt vector sharing. When an interrupt from source `ip` is detected by OS21, it calls each handler that has been registered, until one returns `OS21_SUCCESS`. If no handler accepts the interrupt OS21 will panic.

Once any handlers have been registered with the call, any call to `interrupt_install()` for this `ip` will fail, since it is now set for shared use.

See also: `interrupt_install`, `interrupt_uninstall`, `interrupt_uninstall_shared`

interrupt_lock

Disable all interrupts

Definition:
```
#include <os21.h>
void interrupt_lock(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function disables all interrupts to the CPU.

Note: This call is deprecated, and will be removed from future releases of OS21. Use `interrupt_mask_all()` *instead.*

This function must always be called as a pair with `interrupt_unlock()`, so that it can be used to create a critical region in which the task cannot be preempted by any other task or interrupt. Calls to `interrupt_lock()` can be nested, and the lock is not released until an equal number of calls to `interrupt_unlock()` are made.

A task must not deschedule while an interrupt lock is in effect. When interrupts are locked, calling any function that may not be called by an interrupt service routine is illegal.

See also: `interrupt_unlock`, `interrupt_mask`, `interrupt_mask_all`, `task_lock`

interrupt_mask

Raise the processor's interrupt priority level

Definition:

```
#include <os21.h>
int interrupt_mask(
  int priority);
```

Arguments:

`int priority`	Interrupt priority level to set

Returns: Returns the old priority level of the processor.

Errors: None

Context: Callable from task or system context.

Description: This function allows the processor to protect itself against interrupts up to a specific priority level. This can be used when synchronizing with a device driver interrupt handler for instance.

This call must be used as a pair with `interrupt_unmask()` to create a critical region. While in such a critical region the executing task must not deschedule.

Once this function is called from task context, no preemption can occur.

Example:

```
#include <os21.h>

int old_priority;
old_priority = interrupt_mask( 4 );
... critical section code ...
interrupt_unmask( old_priority );
```

See also: `interrupt_mask_all`, `interrupt_unmask`

interrupt_mask_all

Raise the processor's interrupt priority level to its maximum

Definition:
```
#include <os21.h>
int interrupt_mask_all(void);
```

Arguments: None

Returns: Returns the old priority level of the processor.

Errors: None

Context: Callable from task or system context.

Description: This function allows the processor to protect itself against interrupts to the maximum priority level. This can be used when synchronizing with a device driver interrupt handler for instance.

This call must be used as a pair with `interrupt_unmask()` to create a critical region. While in such a critical region the executing task must not deschedule.

Once this function is called from task context, no preemption can occur.

Example:
```
#include <os21.h>

int old_priority;
old_priority = interrupt_mask_all();
... critical section code ...
interrupt_unmask( old_priority );
```

See also: `interrupt_mask`, `interrupt_unmask`

interrupt_poll

Polls an interrupt

Definition:
```
#include <os21.h>
int interrupt_poll (
  interrupt_t * ip,
  int * value);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.
`int * value`	A location to place the result of the poll.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt cannot be polled or is invalid.

Context: Callable from task or system context.

Description: Attempts to poll the specified interrupt. Some interrupts cannot be polled and in this case the poll fails. For successful calls the result of the poll is placed in the location pointed to by `value`.

interrupt_priority

Obtains an interrupt's priority

Definition:

```
#include <os21.h>
int interrupt_priority (
  interrupt_t * ip,
  int * priority);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.
`int * priority`	A location to place the priority of the interrupt.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt is invalid.

Context: Callable from task or system context.

Description: This function is used to query the priority of an interrupt. The current priority of the given interrupt is written to the location pointed to by priority.

See also: `interrupt_priority_set`

interrupt_priority_set

Sets an interrupt's priority

Definition:

```
#include <os21.h>
int interrupt_priority_set (
  interrupt_t * ip,
  int priority);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.
`int priority`	The required priority for the interrupt.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt is invalid or if the specified priority is invalid.

Context: Callable from task or system context.

Description: This function is used to set the priority of an interrupt. The priority of the given interrupt is set to the value given by `priority`.

See also: `interrupt_priority`

interrupt_raise

Assert a software interrupt

Definition:
```
#include <os21.h>
int interrupt_raise(
  interrupt_t * ip);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.

Returns: `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt cannot be raised or is invalid.

Context: Callable from task or system context.

Description: Attempts to raise the specified interrupt. Some interrupts cannot be raised and in this case the call fails.

See also: `interrupt_unraise`

interrupt_uninstall

Uninstall an interrupt handler

Definition:
```
#include <os21.h>
int interrupt_uninstall(
  interrupt_t * ip);
```

Arguments:

`interrupt_t * ip`	The handle of the interrupt for which a handler is to be uninstalled.

Returns: Returns `OS21_SUCCESS` on success, `OS21_FAILURE` on failure.

Errors: `OS21_FAILURE` if the interrupt source `ip` is invalid, if no handler is currently installed for this interrupt source, or if the interrupt is marked as shareable.

Context: Callable from task only.

Description: This uninstalls the single interrupt handler associated with interrupt source `ip`.

See also: `interrupt_install`, `interrupt_install_shared`, `interrupt_uninstall_shared`

interrupt_uninstall_shared

Uninstall an interrupt handler where the interrupt source is shared

Definition:

```
#include <os21.h>
int interrupt_uninstall_shared (
  interrupt_t * ip,
  interrupt_handler_t handler,
  void * param);
```

Arguments:

`interrupt_t * ip`	The handle of the interrupt for which a handler is to be uninstalled.
`interrupt_handler_t handler`	A handler function which is called when the interrupt is taken.
`void * param`	A parameter which is passed to the handler when it is called.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the interrupt source `ip` is invalid, if the combination of `handler` and `param` are not currently registered for the interrupt source, or if the interrupt has been marked as nonshareable.

Context: Callable from task only.

Description: Uninstalls an interrupt handler from shared interrupt source `ip`.

See also: `interrupt_install`, `interrupt_install_shared`, `interrupt_uninstall`

interrupt_unlock

Enable all interrupts

Definition:

```
#include <os21.h>
void interrupt_unlock(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function re-enables all interrupts to the CPU. Any interrupts which have been prevented from executing start immediately.

Note: This call is deprecated, and will be removed from future releases of OS21. Use `interrupt_unmask()` *instead.*

This function must always be called as a pair with `interrupt_lock()`, so that it can be used to create a critical region in which the task cannot be preempted by another task or interrupt. As calls to `interrupt_lock()` can be nested, the lock is not released until an equal number of calls to `interrupt_unlock()` are made.

See also: `interrupt_lock`, `interrupt_mask`, `interrupt_mask_all`, `task_lock`

interrupt_unmask

Lower the processor's interrupt priority level

Definition:

```
#include <os21.h>
void interrupt_unmask(
  int priority);
```

Arguments:

`int priority`	Interrupt priority level to set

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function lowers the processor's interrupt priority level to the level before an earlier call to `interrupt_mask()` or `interrupt_mask_all()`. For instance, this can be used when synchronizing with a device driver interrupt handler.

This call must always be used as a pair with `interrupt_mask()` or `interrupt_mask_all()` to create a critical region. While in such a critical region the executing task must not deschedule.

Pre-emption resumes once `interrupt_unmask()` finally restores the masking level to the base level.

Example:

```
#include <os21.h>

int old_priority;
old_priority = interrupt_mask( 4 );
... critical section code ...
interrupt_unmask( old_priority );
```

See also: `interrupt_mask`, `interrupt_mask_all`

interrupt_unraise

Unraises an interrupt.

Definition:

```
#include <os21.h>
int interrupt_unraise (
  interrupt_t * ip);
```

Arguments:

`interrupt_t * ip`	A handle for the interrupt obtained using `interrupt_handle()`.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if the specified interrupt cannot be unraised or is invalid.

Context: Callable from task or system context.

Description: Attempts to unraise the specified interrupt. Some interrupts cannot be raised and in this case the call fails.

See also: `interrupt_raise`

12 Caches and memory areas

12.1 Caches and memory overview

Caches provide a way to reduce the time taken for the CPU to access memory and so can greatly increase system performance. Most processors provide an instruction cache (I-cache) and a data (or operand) cache (D-cache). The I-cache is read only, while the D-cache is read/write. When OS21 is started, both caches are enabled.

There is a risk when using cache that the cache can become **incoherent** with main memory, meaning that the contents of the cache conflicts with the contents of main memory. For example, devices that perform **direct memory access** (DMA) modify the main memory without updating the cache, leaving its contents invalid. For this reason extra care must be taken when performing DMA.

If a level 2 cache is available, then OS21 can (optionally) drive it. In this case, there are no specific function calls for level 2 cache support. Instead, OS21 maintains the state of the level 2 cache from within the simple cache management API.

To enable level 2 cache support, OS21 must be handed the base address of the L2 cache controller. This is done using the Board Support Package. See *Chapter 17: Board support package on page 207*.

Note: For full details of the caches provided on a given core, see the appropriate core architecture manual.

12.2 Initializing the cache support system

When OS21 boots, the default configuration is for both the I-cache and D-cache to be enabled. Where possible, the cache API allows caches to be disabled, however, this may not always be possible.

12.3 Flushing, invalidating and purging D-cache lines

The OS21 cache API supports three classes of operation that can be performed on caches.

Flushing	This causes any affected cache lines which contain dirty (unwritten) data to be written back to memory. The cache lines remain in the cache as clean lines. Flushing only makes sense on D-caches.
Purging	This causes any affected cache lines which contain dirty (unwritten) data to be written back to memory. All affected cache lines are then invalidated. Purging only makes sense on D-caches.
Invalidating	This causes all affected cache lines to be invalidated. Any unwritten data is lost by this operation. I-cache and D-cache data may be invalidated.

Purging is required when writing to data structures in memory which are accessed through the D-cache, but are to be shared with another bus master, for instance another CPU, or DMA device. OS21 provides the user with the ability to manipulate shared data either by avoiding the cache altogether, or through the cache with software cache coherency support. This allows users maximum flexibility.

In a similar way, the read-only I-cache can be invalidated in order to safely handle dynamic code loading.

12.4 Cache API summary

All the definitions relating to the cache API can be obtained by including the header file `os21.h`, which itself includes the header file `cache.h`. See *Table 31*, *Table 32* and *Table 33* for a complete list.

Table 31. Functions defined in cache.h

Function	Description
`cache_allocate_data()`	Allocates a range of D-cache
`cache_disable_data()`	Disables the D-cache
`cache_disable_instruction()`	Disables the I-cache
`cache_enable_data()`	Enables the D-cache
`cache_enable_instruction()`	Enables the I-cache
`cache_flush_data()`	Flushes any dirty D-cache lines in range
`cache_flush_data_all()`	Flushes any dirty D-cache
`cache_invalidate_data()`	Invalidates D-cache lines in range
`cache_invalidate_data_all()`	Invalidates all D-cache lines
`cache_invalidate_instruction()`	Invalidates I-cache lines in range
`cache_invalidate_instruction_all()`	Invalidates all I-cache lines
`cache_purge_data()`	Purges D-cache lines in range
`cache_purge_data_all()`	Purges all D-cache lines
`cache_status()`	Returns current cache status information

Table 32. Types defined in cache.h

Type	Description
`cache_data_mode_t`	Additional flags for `cache_enable_data()`
`cache_instruction_mode_t`	Additional flags for `cache_enable_instruction()`
`cache_status_t`	Type for cache status information
`cache_status_flags_t`	Flags for `cache_status()`

Table 33. Macros defined in cache.h

Macro	Description
`ICACHE_LINE_SIZE`	Returns the size of an I-cache line in bytes
`DCACHE_LINE_SIZE`	Returns the size of a D-cache line in bytes

Table 33. Macros defined in cache.h (continued)

Macro	Description
`ICACHE_LINE_ALIGN(P)`	Returns an address which corresponds to the start of the I-cache line which encapsulates pointer `P`
`DCACHE_LINE_ALIGN(P)`	Returns an address which corresponds to the start of the D-cache line which encapsulates pointer `P`

12.5 Cache function definitions

cache_allocate_data

Allocate an address range in the D-cache

Definition:

```
#include <os21.h>
void cache_allocate_data(
  void* base_address,
  size_t length);
```

Arguments:

`void * base_address`	Start address of range to allocate
`size_t length`	Length of range in bytes

Returns: None

Errors: None

Context: Callable from task or system context.

Description: Where possible, this function allocates a range of addresses in the D-cache. Following this call the cache lines corresponding to the address range given are marked as valid in the cache, and tagged as belonging to the address range. This is useful when the caller knows in advance that the entire cache line will be written to, or only certain words, which the caller will write to, are to be used. This avoids the penalty of having to fetch the cache line in from memory, when it is known that its contents are to be overwritten. The effect is to reduce memory bandwidth and the latency of writing to the cache line. Some cores may not support cache allocation, in which case this function will have no effect.

Caution: The contents of the D-cache lines affected by this call are undefined. The caller must not rely on their contents. It is up to the caller to write the desired values to the cache lines.

cache_disable_data

Disable the data cache

Definition:
```
#include <os21.h>
int cache_disable_data(
  int flush);
```

Arguments:

`int flush`	Indicates whether a flush is required prior to disabling

Returns: `OS21_SUCCESS` for success, otherwise `OS21_FAILURE`.

Errors: `OS21_FAILURE` if D-cache cannot be disabled.

Context: Callable from task or system context.

Description: Where possible, this function disables the data cache on the processor. If `flush` is `1`, the cache is flushed prior to disabling to ensure that any dirty cache lines are flushed back to memory. If `flush` is `0`, the entire content of the data cache is lost. Some cores may have a D-cache that is permanently enabled.

See also: `cache_enable_data`

cache_disable_instruction

Disable the instruction cache

Definition:
```
#include <os21.h>
int cache_disable_instruction(void);
```

Arguments: None

Returns: `OS21_SUCCESS` for success, otherwise `OS21_FAILURE`.

Errors: `OS21_FAILURE` if I-cache cannot be disabled.

Context: Callable from task or system context.

Description: This function disables the instruction cache on the processor.

See also: `cache_enable_instruction`

cache_enable_data

Enable the data cache

Definition:
```
#include <os21.h>
int cache_enable_data(
  cache_data_mode_t mode);
```

Arguments:

`cache_data_mode_t mode`	Reserved for future use, set to 0.

Returns: `OS21_SUCCESS` for success, otherwise `OS21_FAILURE`.

Errors: `OS21_FAILURE` if the specified mode parameter is not supported, or if an invalid mode is specified.

Context: Callable from task or system context.

Description: This function enables the data cache. `mode` is currently a reserved parameter which is ignored and should be set to 0.

See also: `cache_disable_data`

cache_enable_instruction

Enable the instruction cache

Definition:
```
#include <os21.h>
int cache_enable_instruction(
  cache_instruction_mode_t mode);
```

Arguments:

`cache_instruction_mode_t mode`	The desired instruction cache mode

Returns: `OS21_SUCCESS` for success, otherwise `OS21_FAILURE`.

Errors: `OS21_FAILURE` if the specified `mode` parameter is not supported, or if an invalid mode is specified.

Context: Callable from task or system context.

Description: This function enables the instruction cache. As a side effect of enabling the instruction cache, its contents are invalidated. `mode` is a bit field parameter which must be given as zero.

See also: `cache_disable_instruction`

cache_flush_data

Flushes addresses within the specified range from the data cache

Definition:
```
#include <os21.h>
void cache_flush_data(
  void * base_address,
  size_t length );
```

Arguments:

`void * base_address`	Start address of range to flush
`size_t length`	Length of range in bytes

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function flushes any valid and dirty data cache lines, which fall within the address range specified, back to memory. Where possible, the cache is not invalidated by flushing, so the affected lines remain in the cache as valid clean lines.

See also: `cache_invalidate_data`, `cache_purge_data`

cache_flush_data_all

Flushes all data cache lines from the D-cache

Definition:
```
#include <os21.h>
void cache_flush_data_all(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function flushes any valid and dirty data cache lines back to memory from the data cache. Where possible, the cache is not invalidated by flushing, so the affected lines remain in the cache as valid clean lines.

See also: `cache_invalidate_data`, `cache_purge_data`

cache_invalidate_data

Invalidates addresses within the specified range from the data cache

Definition:

```
#include <os21.h>
void cache_invalidate_data(
  void * base_address,
  size_t length );
```

Arguments:

`void * base_address`	Start address of range to invalidate
`size_t length`	Length of range in bytes

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function invalidates any valid data cache lines which fall within the address range specified.

*Note: Any dirty cache lines are **not** guaranteed to be written back to memory by this call.*

See also: `cache_flush_data`, `cache_purge_data`

cache_invalidate_data_all

Invalidates all lines in the D-cache

Definition:

```
#include <os21.h>
void cache_invalidate_data_all(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function invalidates the entire D-cache.

Note: Calling this function while running from cached memory is extremely dangerous. Any live data in the data cache is lost, possibly causing a crash.

*Note: Any dirty cache lines are **not** guaranteed to be written back to memory by this call.*

See also: `cache_flush_data`, `cache_purge_data`

cache_invalidate_instruction

Invalidates addresses within the specified range from the instruction cache

Definition:

```
#include <os21.h>
void cache_invalidate_instruction(
  void * base_address,
  size_t length );
```

Arguments:

`void * base_address`	Start address of range to invalidate
`size_t length`	Length of range in bytes

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function invalidates any valid instruction cache lines which fall within the address range specified.

See also: `cache_invalidate_instruction_all`

cache_invalidate_instruction_all

Invalidates the entire instruction cache

Definition:

```
#include <os21.h>
void cache_invalidate_instruction_all(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function invalidates the entire instruction cache.

See also: `cache_invalidate_instruction`

cache_purge_data

Purges addresses within the specified range from the data cache

Definition:

```
#include <os21.h>
void cache_purge_data(
  void * base_address,
  size_t length );
```

Arguments:

`void * base_address`	Start address of range to purge
`size_t length`	Length of range in bytes

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function purges any valid data cache lines which fall within the address range specified. Any dirty cache lines are first written back to memory, then the cache line is invalidated.

See also: `cache_invalidate_data`, `cache_flush_data`

cache_purge_data_all

Purges the entire D-cache

Definition:

```
#include <os21.h>
void cache_purge_data_all(void);
```

Arguments: None

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function purges any valid data cache lines within the D-cache. Any dirty cache lines are first written back to memory, then the cache line is invalidated.

See also: `cache_invalidate_data`, `cache_flush_data`

cache_status

Get details of the current cache configuration

Definition:

```
#include <os21.h>
void cache_status(
  cache_status_t* status
  cache_status_flags_t flags);
```

Arguments:

`cache_status_t* status`	Gets cache status information
`cache_status_flags_t flags`	Reserved for future use, set to `0`

Returns: None

Errors: None

Context: Callable from task or system context.

Description: This function returns information about the current cache configuration. The fields in `status` are described in *Table 34*.

Table 34. Fields in cache_status_t structure

Field name	Description
`instruction_cache_size`	Size of I-cache, 0 if disabled
`data_cache_size`	Size of D-cache, 0 if disabled
`instruction_cache_line_size`	Number of bytes per instruction cache line
`instruction_cache_ways`	Number of ways in the instruction cache
`data_cache_line_size`	Number of bytes per data cache line
`data_cache_ways`	Number of ways in the data cache
`data_cache_enabled`	1 if the D-cache is enabled, otherwise 0
`instruction_cache_enabled`	1 if the I-cache is enabled, otherwise 0

13 Virtual memory

13.1 Virtual memory overview

When devices or memory are accessed over a bus, the processor uses an address to specify the location that is to be accessed. If this address relates to an actual physical memory location or device, it is referred to as a **physical address**.

Both memory and device registers can generally be accessed in several modes (for example cached or uncached) and it may also be possible to implement protection mechanisms on memory regions (such as read-only or read/write). For instance, device registers are usually accessed without going through the processor's cache, while accesses to memory are usually done through the cache to give performance benefits. Consequently, using physical addresses directly may not be a suitable method for accessing memory or devices.

An alternative way of accessing memory is to use **virtual addresses**, where the address used by the processor represents (or is "mapped to") a physical address. When using this method of addressing memory, an address translation mechanism translates virtual addresses into physical addresses before they go out on the bus. The translation mechanism may also associate the mode of access with the virtual address.

Often these address translations are created dynamically. When a translation has been created, the physical address is said to be **mapped**. If no translation exists to enable a given physical address to be accessed, that physical address is said to be **unmapped**. A mapping where the virtual address is the same as the physical address is called an identity mapping.

There are other benefits of virtual memory (such as paging, dynamic loading and so on), but these are not relevant to this document.

Note: OS21 does not implement paging, although it may overcommit the address translation mechanism and implement a fault handler.

Various means of address translation exist, one common implementation being a Memory Management Unit (MMU).

Address translations are generally implemented by using one or more pages of potentially various sizes. A page is therefore the unit of address translation. Each page of the translation requires an entry in the translation mechanism. When the processor references a virtual address, a lookup is performed in the translation mechanism. If a translation is present, the virtual address is converted to a physical address and accessed using the mode(s) given in the translation.

If the translation mechanism is overcommitted, the required translation may not be present in the address translation mechanism when needed. In this case, a page fault is said to have occurred, and to overcome this, the operating system has to swap translation entries in and out of the translation mechanism. This process is called fault handling.

When handling a fault, OS21 attempts to find the required translation in a software table. If a translation is found, it swaps it into the address mechanism before restarting the instruction that caused the exception. If no translation is found, a fatal exception occurs and program execution is stopped.

13.2 Virtual memory support functions

OS21 provides support for virtual memory in a way which hides the actual address translation mechanism. A very simple virtual memory API is provided.

13.2.1 Creating and deleting mappings

In most cases, an address translation only exists if it is created. This is done using the OS21 function `vmem_create()`. (An exception to this is when the ST40 toolset sets up PMB; this appears as static mappings and `vmem_create()` is not required.)

The function `vmem_create()` returns the base address of a new virtual address range. The base address can be used to access the indicated physical address range within the mode specified. Accesses to the virtual address are translated as they go onto the bus. When the mapping has been created, no further action is needed.

When an address translation is no longer needed, the mapping can be removed using `vmem_delete()`.

If a virtual address is accessed for which no translation exists, a fatal exception occurs.

One or more mappings are created automatically by the startup process. These mappings exist for the lifetime of the system.

13.2.2 Obtaining information about a mapping

Given a virtual address, it is often necessary to obtain certain information from it. For example, it is a common requirement to obtain the physical address to which the virtual address is translated. Various devices such as DMA engines require the use of physical addresses. The function `vmem_virt_to_phys()` may be used to convert a virtual address to the corresponding physical address.

In a similar way, `vmem_virt_mode()` can be used to discover the addressing modes used when the given virtual address is used.

13.2.3 Other information

To minimize faults, memory chunks can be aligned so they straddle the least page boundaries, since each page in the translation must be aligned on a boundary of its own size. OS21 supplies a function `vmem_min_page_size()` as an aid to this technique.

13.3 Virtual memory API summary

All the definitions relating to virtual memory can be obtained by including the header file `os21.h`, which itself includes the header file `vmem.h`. See *Table 35* for a complete list.

Table 35. Functions defined in vmem.h

Function	Description
`vmem_create()`	Create a mapped address range
`vmem_delete()`	Remove a mapped address range
`vmem_min_page_size()`	Return the minimum page size
`vmem_virt_mode()`	Return the actual mode of a mapped address range
`vmem_virt_to_phys()`	Return the physical address of a mapped address range

13.4 Virtual memory function definitions

vmem_create

Creates an address translation

Definition:

```
#include <os21.h>
void * vmem_create(
      void * pAddr,
      unsigned int length,
      void * vAddr,
      unsigned int mode);
```

Arguments:

`pAddr`	Start address of range to map.
`length`	Length of address range to map.
`vAddr`	Required virtual address or `NULL` to allow OS21 to return any available address.
`mode`	Required mode for the mapping.

Returns: The virtual address corresponding to `pAddr` is returned, or `NULL` if the mapping could not be created.

Errors: `NULL` if:

- the mode is invalid
- the requested virtual address is used already or not available
- the physical address is not available for mapping
- out of memory
- out of virtual address space
- the requested virtual address is not aligned to the physical address
- either the physical or virtual address range wraps

Description: `vmem_create()` attempts to create a mapping using the given parameters. If successful the virtual address corresponding to the physical address given is

returned. If the mapping cannot be created, NULL is returned (the virtual address NULL and a page around it is reserved by OS21).

If the mapping matches a fixed mapping which has been created or inherited by OS21 at start up, that mapping is used to obtain the required virtual address; in this case no new mapping is created. These fixed mappings exist for the lifetime of the system. If no fixed mapping exists to satisfy the request, vmem_create() attempts to create a new one. OS21 allows any physical address range to be mapped, but in practice slightly more than the given range may be mapped to implement the requested mapping. All mappings are aligned to the smallest page boundary below and above the requested address range.

If the required virtual address is specified (non-NULL), OS21 attempts to place the mapping accordingly. If it cannot do this, the request fails. The translation mechanism usually requires the virtual address to be mutually aligned to the physical address, so the bits that address into the minimum page size must match. For example, if the minimum page size is 4 Kbytes, the bottom 12 bits of the required virtual address must match the bottom 12 bits of the given physical address.

If no required virtual address is specified (vAddr is NULL), OS21 finds a suitable virtual address that fulfils the request, places the mapping and then returns that virtual address.

OS21 may prohibit various address ranges from being mapped; this causes the request to fail. An example of this might be the translation mechanism registers. A similar case to this is where the caller requests exclusive access to the memory: all requests to map any of the same addresses in future requests will fail.

The mode consists of mode flags which can be logically OR-ed. These are described in *Table 36*.

If invalid combinations of mode flags are given, the call fails. Some of the mode flags are taken as hints. The exact mode used for the mapping may be discovered by calling vmem_virt_mode() on the returned virtual address range from a successful vmem_create() call.

If no caching mode is specified, the mapping is created uncached by default. For uncached mappings, if no write buffer mode is specified, no write buffer is used. If no permissions are specified, the mapping is created to enable read, write, and execute.

Table 36. vmem_create() mode flags

Flag name	Description
VMEM_CREATE_CACHED	Accesses to the virtual address range are cached.
VMEM_CREATE_UNCACHED	Accesses to the virtual address range are uncached.
VMEM_CREATE_WRITE_BUFFER	Only valid when the virtual address range is uncached. Write accesses go to a write buffer where possible.
VMEM_CREATE_NO_WRITE_BUFFER	Only valid when the virtual address range is uncached. Write accesses do not go to a write buffer.
VMEM_CREATE_READ	Read accesses to the virtual memory range are allowed.
VMEM_CREATE_WRITE	Write accesses to the virtual memory range are allowed.

Table 36. vmem_create() mode flags (continued)

Flag name	Description
VMEM_CREATE_EXECUTE	Execute accesses to the virtual memory range are allowed.
VMEM_CREATE_LOCK	The translation is locked into the translation mechanism so that it never has to be swapped in.
VMEM_CREATE_EXCL	Further mappings to the physical address range will not be allowed.

See also: vmem_delete, vmem_min_page_size, vmem_virt_mode

vmem_delete

Removes an address translation

Definition:

```
#include <os21.h>
int vmem_delete(void * vAddr);
```

Arguments:

vAddr — Virtual address of translation to remove.

Returns: OS21_SUCCESS on success, otherwise OS21_FAILURE.

Errors: OS21_FAILURE if:

- the virtual address does not match the address returned by a previous call to vmem_create()
- the mapping no longer exists because it has already been deleted
- out of memory

Description: This function attempts to remove a previously created address mapping.

For fixed mappings (which are present for the lifetime of the system), any virtual address in the mapped range may be given to vmem_delete(). The call returns OS21_SUCCESS, but the fixed mapping still exists. This allows code to be made portable. The same vmem_create(), vmem_delete() sequence works regardless of whether the mapping happened to be fixed or not. The cache, if it exists, is *not* purged.

For non-fixed mappings, the virtual address passed to vmem_delete() must be the address returned by the call to vmem_create() that created the mapping. When it removes a non-fixed, cached mapping, vmem_delete() also purges the data cache for that mapping.

See also: vmem_create

vmem_min_page_size

Returns the minimum page size for a given implementation

Definition:
```
#include <os21.h>
unsigned int vmem_min_page_size(void);
```

Arguments: None.

Returns: The size of the smallest page size for the implementation in use.

Errors: None.

Description: This function returns to the caller, the smallest page size in use for a given implementation.

See also: `vmem_create`

vmem_virt_mode

Returns the translation mode for a given virtual address

Definition:
```
#include <os21.h>
int vmem_virt_mode(void * vAddr, unsigned int * modep);
```

Arguments:

`vAddr`	Virtual address for which mode information is to be provided.
`modep`	Pointer to a location which receives the mode information.

Returns: `OS21_SUCCESS` on success, otherwise `OS21_FAILURE`.

Errors: `OS21_FAILURE` if the virtual address is not in use.

Description: This function returns the translation mode in operation for a given virtual address. The mode is a combination of flags which are OR-ed together and are described in *Table 37*. If the virtual address is not in use, an error is returned.

Table 37. vmem_virt_mode() mode flags

Flag name	Description
`VMEM_CREATE_CACHED`	Accesses to the virtual address range are cached.
`VMEM_CREATE_UNCACHED`	Accesses to the virtual address range are uncached.
`VMEM_CREATE_WRITE_BUFFER`	Only valid when the virtual address range is uncached. Write accesses go to a write buffer where possible.
`VMEM_CREATE_NO_WRITE_BUFFER`	Only valid when the virtual address range is uncached. Write accesses do not go to a write buffer.
`VMEM_CREATE_READ`	Read accesses to the virtual memory range are allowed.
`VMEM_CREATE_WRITE`	Write accesses to the virtual memory range are allowed.
`VMEM_CREATE_EXECUTE`	Execute accesses to the virtual memory range are allowed.

Table 37. vmem_virt_mode() mode flags (continued)

Flag name	Description
VMEM_CREATE_LOCK	The translation is locked into the translation mechanism so that it never has to be swapped in.
VMEM_CREATE_EXCL	Further mappings to the physical address range will not be allowed.

See also: vmem_create

vmem_virt_to_phys

Returns the physical address to which a virtual address translates

Definition:

```
#include <os21.h>
int vmem_virt_to_phys(
      void * vAddr, void ** pAddr);
```

Arguments:

vAddr	Virtual address for which a physical address is required.
pAddr	Pointer to a location which receives the physical address to which the virtual address is translated.

Returns: OS21_SUCCESS on success, otherwise OS21_FAILURE.

Errors: OS21_FAILURE if the virtual address is not in use.

Description: This function returns the physical address to which the given virtual address is translated. If no such translation exists, an error is returned.

See also: vmem_create

14 Exceptions

An exception is an unexpected event which occurs during the execution of an instruction. When an exception occurs, the CPU jumps to a different address to handle the exception. The code that it finds at this address is called an exception handler. Many exceptions are fatal to program operation and in this case the exception handler can at best output a useful message before processing terminates. Other exceptions may require some remedial processing to be carried out before the instruction that caused the exception is re-executed.

When an exception occurs, the CPU stops executing the current task, and starts executing the exception handler for that exception. This switch is performed completely in hardware, and so is extremely rapid. Similarly, when the exception handler has completed (and assuming the exception was not fatal), the CPU resumes execution of the task that was running when the exception occurred. The task is unaware that it has been interrupted.

Note: It is not just tasks that can generate exceptions. Any code (task, interrupt handler or even exception handler) can generate an exception.

The exception handler that the CPU executes in response to an exception, is called the first level exception handler. This piece of code is supplied as part of OS21. The first level exception handler in OS21 dispatches exception handling to an appropriate function, if it is an exception reserved for use by OS21, or to the toolset. Reserved exceptions include debug event exceptions or software traps used by OS21 to implement a context switch.

For non-reserved exceptions, OS21 maintains a list of user supplied functions which are called sequentially with a description of the exception. The function gives the user an opportunity to process the exception. For example it may correct a misaligned address before resuming a task that generated a misaligned access instruction, or it may kill a task that has generated a fatal exception such as a bus error. The function then returns a code telling OS21 whether it processed the exception and what it requires OS21 to do next. In the case of a fixed-up misaligned access, OS21 can resume the task that generated the exception. In the case of a task terminated due to a bus error, OS21 needs to run another task.

If no user defined function deals with the exception, then OS21 terminates that application with a message about the unexpected exception. It is up to the user code to add exception handling functions to the list of functions that OS21 calls on taking an exception.

14.1 Attaching exception handlers

OS21 defines an exception handler as follows:

```
typedef int (*exception_handler_t)(exception_t * excepp);
```

The contents of `exception_t` are hardware specific, please see the appropriate header files.

An exception handler must return `OS21_SUCCESS` if it successfully identified and handled an exception or `OS21_FAILURE` if it did not.

```
int example_exception_handler (exception_t * excepp)
{
  if (i_can_handle_this_exception (excepp))
  {
    ... handle the exception
    return (OS21_SUCCESS);
  }
  return (OS21_FAILURE);
}
```

An exception handler is attached to the list of exception handlers using the `exception_install()` function:

```
int result;
exception_handler_t my_exception_handler;

result = exception_install (my_exception_handler);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to attach exception handler\n");
}
```

The handler can be uninstalled as follows:

```
int result;
exception_handler_t my_exception_handler;

result = exception_uninstall (my_exception_handler);
if (result != OS21_SUCCESS)
{
  printf ("ERROR: Failed to detach exception handler\n");
}
```

14.2 Contexts and exception handler code

Code running under OS21 may run in one of two environments (or contexts). These are called task context and system context. OS21 exception handlers are run from system context.

The main difference between system context and task context is that code running in system context is not allowed to block. Undefined behavior occurs if code running in system context blocks. As a result of this constraint, code running from system context should **never** call an OS21 function that may block. Please refer to the individual function descriptions for details of the context from which the OS21 functions may be run.

14.3 Exception API summary

All the definitions relating to exceptions can be obtained by including the header file `os21.h`, which itself includes the header file `exception.h`. See *Table 38* and *Table 39* for a complete list.

Table 38. Functions defined in exception.h

Function	Description
`exception_install()`	Installs an exception handler
`exception_uninstall()`	Uninstalls an exception handler

Table 39. Types defined in exception.h

Type	Description
`exception_t`	An abstract exception type
`exception_handler_t`	An exception handler

14.4 Exception function definitions

exception_install

Install an exception handler to the chain of exception handlers called when OS21 takes an exception

Definition:
```
#include <os21.h>
int exception_install (
  exception_handler_t handler);
```

Arguments:

`exception_handler_t handler`	The handler function which is called when an exception is taken.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` for failure.

Errors: `OS21_FAILURE` if this handler is already attached, or if there is insufficient memory to complete the operation.

Context: Callable from task only.

Description: This installs the specified user exception handler into the chain of exception handlers called by OS21 when an exception is taken. The user handler should return `OS21_SUCCESS` if it handled the exception, otherwise it should return `OS21_FAILURE`.

See also: `exception_uninstall`

exception_uninstall

Uninstall an exception handler

Definition:
```
#include <os21.h>
int exception_uninstall(
  exception_handler_t handler);
```

Arguments:

`exception_handler_t`	The handler to be removed from the chain of exception handlers called when OS21 takes an exception.

Returns: Returns `OS21_SUCCESS` on success, `OS21_FAILURE` on failure.

Errors: `OS21_FAILURE` if the handler is not attached to the chain of exception handlers called when OS21 takes an exception.

Context: Callable from task only.

Description: This removes the exception handler from the chain of exception handlers called when OS21 takes an exception.

See also: `exception_install`

15 Profiling

OS21 provides a simple, flexible flat profiler that can be used to analyze the performance characteristics of a target system. The OS21 profiler allows profiling of a single task, a single interrupt level or the system as a whole. The profiler can be configured, started and stopped under program control. This means that it can be setup and started when required, so that only the situation under investigation is analyzed.

Profiling is the term used to describe the gathering and subsequent analysis of a system's performance data. The OS21 profiler gathers information about a system by sampling the program counter (PC) at regular intervals. These samples are collected into 'buckets'. A bucket is a counter associated with an address range. Each time a PC sample is taken, the profiler determines the bucket to which it belongs, and increments the bucket value accordingly. These samples are accumulated over a period of time, and this profile data is then written to a file on the host system.

A host application analyses the profile data in the file and cross-references it with the target application's symbol table to produce a report. This report identifies where the processor spent its time during the period being profiled.

15.1 Initializing the profiler

The OS21 profiler is initialized with `profile_init()`. This call takes two parameters which control the resolution of the profiler: the number of instructions in each bucket, and the PC sampling frequency. Fewer instructions per bucket result in more accurate matching of symbols to PC locations when the profile report is generated and higher PC sampling frequencies yield more accurate results over a short profile run. However, higher sampling frequencies also result in the profiler becoming intrusive. Too high a sampling frequency may impact the real-time responsiveness of a system. Normally a sampling period of a few hundred hertz is sufficient to get good results, however the best sampling frequency depends on the particular application.

Note: Due to hardware limitations, the actual sampling frequency used by the profiler may not be exactly as requested. The profiler endeavors to achieve a sampling frequency as close as possible to the frequency requested. The actual frequency used is recorded in the profile data.

To release the resources used by the profiler, `profile_deinit()` is used, `profile_init()` may be called after a call to `profile_deinit()` to re-initialize the profiler. Each time the profiler is initialized, it is set up to use the given parameters, any information previously gathered is lost.

Note: 1 It is not possible to operate the profiler at frequencies less than the timeslice frequency, regardless of whether timeslicing is enabled or not.

*2 It is not possible to operate the profiler if the application has been built to use the GNU profiler (**gprof**).*

15.2 Starting the profiler

The OS21 profiler is started with one of the following calls:

- `profile_start_all()`,
- `profile_start_task()`,
- `profile_start_interrupt()`.

`profile_start_all()` starts the profiler gathering information for the whole system, that is, every task and interrupt level. `profile_start_task()` takes a single `task_t` pointer as its parameter and starts the profiler gathering information for just the specified task. `profile_start_interrupt()` takes an interrupt level as its parameter and starts the profiler gathering information for that interrupt level.

Note: 1 The valid range of interrupt levels depends on the target.

2 The profiler relies on the timeslice interrupt to trigger a single sample. Therefore it is not possible to profile code that is run with interrupts masked to a level equal to or higher than the level of the timeslice interrupt.

The timeslice interrupt is normally at the highest available interrupt level.

Examples of use:

```
profile_start_all(); /* Profile the whole system */
profile_start_task(my_task); /* Profile just my_task */
profile_start_interrupt(5); /* Profile just interrupt level 5 */
```

Any profile information already gathered is lost when the profiler is started.

15.3 Stopping the profiler

The profiler is stopped with `profile_stop()`. This call takes no parameters, and stops the profiler from gathering any further PC samples. If a task is deleted before the profiler has been stopped, the task is removed from the profiler data. If the gathered profile information is not written to the host before the profiler is restarted or re-initialized following a call to `profile_stop()`, the data is lost.

15.4 Writing profile data to the host

The gathered profile data is written to the host using `profile_write()`. This call takes the name of a host file as its single parameter. The formatted profile data is written to the specified file. Once the data is written to the host, a new profile session can be started, if required, by restarting the profiler.

15.5 Processing the profile data

The profile files written to the host contain binary profile data. This data must be analyzed in conjunction with the executable file which generated the data. The Perl tool **os21prof.pl** is provided with OS21 to perform this analysis. It is located in the `bin` directory of the toolset installation directory and is invoked as follows:

```
os21prof executable-file profile-file
```

The following example displays a profile report for the application `test.out`, from the profile data held in `profile.log`.

```
os21prof test.out profile.log
```

os21prof produces a formatted report of the amount of time spent in each task, interrupt level and named program location, as appropriate for the type of profile data that has been collected.

15.6 Profile data binary file format

This section describes the format of the binary files generated by the `profile_write()` function. The format is described using a modified Backus-Naur Form (BNF) notation (see *Software notation on page 8* for more details concerning BNF).

```
format ::= profile-all
   | profile-task
   | profile-interrupt

profile-all ::= profile-all-magic
   frequency
   number-interrupts
   number-timer-ticks
   number-bucket-arrays
   bucket-step-size
   number-tasks
   task-profile-list
   interrupt-profile-list
   bucket-array-list

profile-all-magic ::= INT32 (0x0521d23c)

profile-task ::= profile-task-magic
   frequency
   number-timer-ticks
   number-bucket-arrays
   bucket-step-size
   task-profile
   bucket-array-list

profile-task-magic ::= INT32 (0x0521d23e)
```

```
profile-interrupt ::= profile-interrupt-magic
   frequency
   interrupt-level
   number-timer-ticks
   number-bucket-arrays
   bucket-step-size
   bucket-array-list

profile-interrupt-magic ::= INT32 (0x0521d23d)

task-profile-list ::= task-profile
   | task-profile-list task-profile

task-profile ::= handle
   counter
   task-name

interrupt-profile-list ::= interrupt-profile
   | interrupt-profile-list interrupt-profile

interrupt-profile ::= counter

bucket-array-list ::= bucket-array
   | bucket-array-list bucket-array

bucket-array ::= number-buckets
   number-compressed-buckets
   address
   bucket-list

bucket-list ::= bucket
   | bucket-list bucket

bucket ::= counter

frequency ::= INT32

number-interrupts ::= INT32

number-timer-ticks ::= INT32

number-bucket-arrays ::= INT32

bucket-step-size ::= INT32

number-tasks ::= INT32

interrupt-level ::= INT32

number-buckets ::= INT32

number-compressed-buckets ::= INT32

handle ::= INT32

address ::= INT32

counter ::= INT32

task-name ::= BYTE[16]
```

where `INT32` is a 32-bit integer and `BYTE[16]` is an array made up of 16 `char` elements.

15.7 Profile API summary

All the definitions related to the OS21 profiler can be accessed by including the header file `os21.h`, which itself includes the header file `profile.h`. See *Table 40* for a complete list.

Table 40. Functions defined in profile.h

Function	Description
`profile_deinit()`	De-initializes the profiler
`profile_init()`	Initializes the profiler
`profile_start_all()`	Starts profiling the whole system
`profile_start_interrupt()`	Starts profiling a single interrupt level
`profile_start_task()`	Starts profiling a single task
`profile_stop()`	Stops the profiler
`profile_write()`	Writes profile data to the host

15.8 Profile function definitions

profile_deinit

De-initialize the profiler

Definition:
```
#include <os21.h>
int profile_deinit (void);
```

Arguments: None.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: Fails if not called from task context, if the profiler has not been initialized or if the profiler is running.

Context: Callable from task only.

Description: `profile_deinit()` de-initializes the profiler. It releases memory and all resources allocated during `profile_init()`.

See also: `profile_init`

profile_init

Initialize the profiler

Definition:

```
#include <os21.h>
int profile_init (
  const size_t instrs_per_bucket,
  const int hz);
```

Arguments:

`instrs_per_bucket`	The number of instructions included in each bucket.
`hz`	The desired sampling frequency.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: Fails if the sampling frequency is invalid (less than the timeslicing frequency), the number of instructions per bucket is zero, if the OS21 kernel has not been started, if not called from task context, if the profiler has already been initialized, if the GNU profiler is present or if out of memory.

Context: Callable from task only.

Description: `profile_init()` initializes the profiler. It allocates memory for the buckets that are required to cover the application's static text section, and gets ready to sample at the frequency specified. The exact sampling frequency requested may not be possible on every platform, in which case the profiler selects a frequency as close as possible to the frequency requested. The frequency used by the profiler is given in the profile data.

See also: `profile_deinit`

profile_start_all

Start the profiler collecting system wide profile information

Definition:

```
#include <os21.h>
int profile_start_all (void);
```

Arguments: None

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: The profiler has not been initialized, is already running, or another profiler call was in progress at the time this call was made.

Context: Callable from task or system context.

Description: `profile_start_all()` starts the profiler collecting profile information for the whole system.

See also: `profile_start_interrupt`, `profile_start_task`

profile_start_interrupt

Start the profiler collecting profile information for an interrupt level

Definition:

```
#include <os21.h>
int profile_start_interrupt (
  const int level);
```

Arguments:

`level`	The interrupt level to profile.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: The profiler has not been initialized, the profiler is already running, an illegal interrupt level was given, or another profiler call was in progress at the time this call was made.

Context: Callable from task or system context.

Description: `profile_start_interrupt()` starts the profiler collecting profile information for a single interrupt level. The legal range for `level` is dependant on the platform.

See also: `profile_start_all`, `profile_start_task`

profile_start_task

Start the profiler collecting profile information for a single task

Definition:

```
#include <os21.h>
int profile_start_task (
  task_t *taskp);
```

Arguments:

`taskp`	The task to profile.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: The profiler has not been initialized, is already running, `taskp` is `NULL` and this call was made from system context, or another profile call was in progress at the time this call was made.

Context: Callable from task or system context.

Description: `profile_start_task()` starts the profiler collecting profile information for the given task. If the task pointer is `NULL`, then the current task is profiled.

See also: `profile_start_all`, `profile_start_interrupt`

profile_stop

Stop the profiler collecting profile information

Definition:
```
#include <os21.h>
int profile_stop (void);
```

Arguments: None

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: The profiler was not running or another profile call was in progress at the time this call was made.

Context: Callable from task or system context.

Description: `profile_stop()` stops the profiler collecting profile information. Once the profiler has been stopped the collected data can be written to the host with `profile_write()`.

See also: `profile_start_all`, `profile_start_interrupt`, `profile_start_task`, `profile_write`

profile_write

Write the collected profile information to the host

Definition:
```
#include <os21.h>
int profile_write (
  const char *filename);
```

Arguments:

`filename`	The file on the host to receive the profile data.

Returns: Returns `OS21_SUCCESS` for success, `OS21_FAILURE` if an error occurs.

Errors: No profile data has been collected, the profiler is still running, the profiler has not been initialized, the call was made from system context, or a file I/O error occurred.

Context: Callable from task only.

Description: `profile_write()` writes the collected profile data to the given file on the host system. The collected profile information can then be analyzed with the `os21prof.pl` tool, see *Section 15.5: Processing the profile data on page 186*.

See also: `profile_stop`

16 Power management

OS21 provides a support framework for power management. A number of power levels are defined, along with a mechanism for transitioning between the levels.

Application software may add (and delete) callbacks to be called whenever a transition to a new power level occurs, with the new power level passed as a parameter to each callback. There is also a mechanism for performing operations such as RAM power management and wake up interrupt validation when in standby power mode.

16.1 Power levels

Three power levels are defined:

`OS21_POWER_LEVEL_ON`	This is the normal power level, when the system is running and fully operational. When running at this power level, call the function `power_level_set()` to transition OS21 to one of the two standby levels. When `power_level_set()` returns, the power level is once again `OS21_POWER_LEVEL_ON`.
`OS21_POWER_LEVEL_ACTIVE_STANDBY`	The definition of this level is determined by the application, but it is normally a standby state where the processor may sleep, but with clocks and RAM fully active.
`OS21_POWER_LEVEL_PASSIVE_STANDBY`	The definition of this level is determined by the application, but it is normally a standby state where the processor may sleep, with the RAM put into self refresh, and clocks slowed down to conserve power.

The OS21 timers run in the `OS21_POWER_LEVEL_ON` power level only. The OS21 timers stop whenever OS21 enters a standby power level and restart when OS21 re-enters the `OS21_POWER_LEVEL_ON` power level.

16.2 Power callbacks

OS21 maintains a list of callbacks that are called in a defined order when the system transitions to one of the standby levels, and in the reverse order when the system transitions back to the `OS21_POWER_LEVEL_ON` level. The order is controlled by an order number passed in when the callback is added. The order number must lie between `OS21_POWER_CALLBACK_ORDER_FIRST` and `OS21_POWER_CALLBACK_ORDER_LAST`, inclusive. When the callback is called, OS21 passes a parameter to identify the power level that the system is about to enter.

16.3 Power pCode

OS21 defines a virtual machine that runs pseudo-code (pCode) to perform operations such as RAM power management and wake up interrupt validation. The pCode to be run on the virtual machine is specified by calling `power_pcode_set()`. The pCode supplied in this way is run when the machine enters one of the standby power levels.

Where possible, the pCode is run entirely from the caches, so it is possible for it to put the RAM into self refresh to conserve power. However, care must be taken not to access RAM when this is done.

When pCode execution is complete, the machine transitions back to the `OS21_POWER_LEVEL_ON` power level, calling any callbacks in reverse order.

16.3.1 Virtual machine

The virtual machine has three registers, called A, B and C. These registers can be set to any value, or various mechanisms may be used to load and store values from memory. Basic arithmetic and logical operations can be performed on, or between the registers.

The virtual machine has some basic logical status flags, which allow comparison and optional branching to be performed. Two status flags are supported: EQUAL (`BEQ` and `BNE`) and PASSIVE (`BPA` and `BAC`).

16.3.2 pCode definition

The pCode to be run when the machine transitions to one of the standby modes is specified by calling the `power_pcode_set()` function. The pCode to be executed is passed to this function as a simple data array.

At the start of pCode execution, register A is initialized with the input parameter. The final value of register A is passed back as an output parameter.

When pCode execution is completed, the system is brought out of the standby power level back to the `OS21_POWER_LEVEL_ON` power level. The pCode can be used to put RAM into self refresh, switch off unnecessary clocks and sleep until an interrupt is received. Once the sleep is over, the pCode can be used to check that the interrupt was valid and then bring the RAM into active mode.

16.3.3 pCode macros

A number of C macros are provided to make construction of the pCode table easier. The macros are defined by including the OS21 header file:

```
#include <os21.h>
```

Table 41 provides a list of the macros.

Table 41. pCode macros

Macro Name	Description
SET*x* pCode Macros	
OS21_POWER_PCODE_SETA(VALUE)	Load register *x* with VALUE.
OS21_POWER_PCODE_SETB(VALUE)	
OS21_POWER_PCODE_SETC(VALUE)	
LD*x* pCode Macros	
OS21_POWER_PCODE_LDA(ADDRESS)	Load data from ADDRESS to register *x*. Loads may be 32, 16 or 8 bits wide. LD*x* is the same as LD*x*32.
OS21_POWER_PCODE_LDB(ADDRESS)	
OS21_POWER_PCODE_LDC(ADDRESS)	
OS21_POWER_PCODE_LDA32(ADDRESS)	
OS21_POWER_PCODE_LDB32(ADDRESS)	
OS21_POWER_PCODE_LDC32(ADDRESS)	
OS21_POWER_PCODE_LDA16(ADDRESS)	
OS21_POWER_PCODE_LDB16(ADDRESS)	
OS21_POWER_PCODE_LDC16(ADDRESS)	
OS21_POWER_PCODE_LDA8(ADDRESS)	
OS21_POWER_PCODE_LDB8(ADDRESS)	
OS21_POWER_PCODE_LDC8(ADDRESS)	
ST*x* pCode Macros	
OS21_POWER_PCODE_STA(ADDRESS)	Store data to ADDRESS from register *x*. Stores may be 32, 16 or 8 bits wide. ST*x* is the same as ST*x*32.
OS21_POWER_PCODE_STB(ADDRESS)	
OS21_POWER_PCODE_STC(ADDRESS)	
OS21_POWER_PCODE_STA32(ADDRESS)	
OS21_POWER_PCODE_STB32(ADDRESS)	
OS21_POWER_PCODE_STC32(ADDRESS)	
OS21_POWER_PCODE_STA16(ADDRESS)	
OS21_POWER_PCODE_STB16(ADDRESS)	
OS21_POWER_PCODE_STC16(ADDRESS)	
OS21_POWER_PCODE_STA8(ADDRESS)	
OS21_POWER_PCODE_STB8(ADDRESS)	
OS21_POWER_PCODE_STC8(ADDRESS)	

Table 41. pCode macros (continued)

Macro Name	Description
MOV*xy* pCode Macros	
OS21_POWER_PCODE_MOVAB	Copy the contents of register *x* to register *y*.
OS21_POWER_PCODE_MOVAC	
OS21_POWER_PCODE_MOVBA	
OS21_POWER_PCODE_MOVBC	
OS21_POWER_PCODE_MOVCA	
OS21_POWER_PCODE_MOVCB	
NOT*x* pCode Macros	
OS21_POWER_PCODE_NOTA	Perform a logical NOT operation on register *x*.
OS21_POWER_PCODE_NOTB	
OS21_POWER_PCODE_NOTC	
AND*x*(VALUE) pCode Macros	
OS21_POWER_PCODE_ANDA(VALUE)	Perform a logical AND between register *x* and VALUE. The macro stores the result in register *x*.
OS21_POWER_PCODE_ANDB(VALUE)	
OS21_POWER_PCODE_ANDC(VALUE)	
AND*xy* pCode Macros	
OS21_POWER_PCODE_ANDAB	Perform a logical AND between register *x* and register *y*. The macro stores the result in register *x*.
OS21_POWER_PCODE_ANDAC	
OS21_POWER_PCODE_ANDBA	
OS21_POWER_PCODE_ANDBC	
OS21_POWER_PCODE_ANDCA	
OS21_POWER_PCODE_ANDCB	
OR*x*(VALUE) pCode Macros	
OS21_POWER_PCODE_ORA(VALUE)	Perform a logical OR between register *x* and VALUE. The macro stores the result in register *x*.
OS21_POWER_PCODE_ORB(VALUE)	
OS21_POWER_PCODE_ORC(VALUE)	
OR*xy* pCode Macros	
OS21_POWER_PCODE_ORAB	Perform a logical OR between register *x* and register *y*. The macro stores the result in register *x*.
OS21_POWER_PCODE_ORAC	
OS21_POWER_PCODE_ORBA	
OS21_POWER_PCODE_ORBC	
OS21_POWER_PCODE_ORCA	
OS21_POWER_PCODE_ORCB	

Table 41. pCode macros (continued)

Macro Name	Description
ADD*x*(VALUE) pCode Macros	
OS21_POWER_PCODE_ADDA(VALUE)	Perform an unsigned ADD between register *x* and VALUE. The macro stores the result in register *x*.
OS21_POWER_PCODE_ADDB(VALUE)	
OS21_POWER_PCODE_ADDC(VALUE)	
ADD*xy* pCode Macros	
OS21_POWER_PCODE_ADDAB	Perform an unsigned ADD between register *x* and register *y*. The macro stores the result in register *x*.
OS21_POWER_PCODE_ADDAC	
OS21_POWER_PCODE_ADDBA	
OS21_POWER_PCODE_ADDBC	
OS21_POWER_PCODE_ADDCA	
OS21_POWER_PCODE_ADDCB	
SUB*x*(VALUE) pCode Macros	
OS21_POWER_PCODE_SUBA(VALUE)	VALUE is subtracted (unsigned) from register *x*. The macro stores the result in register *x*.
OS21_POWER_PCODE_SUBB(VALUE)	
OS21_POWER_PCODE_SUBC(VALUE)	
SUB*xy* pCode Macros	
OS21_POWER_PCODE_SUBAB	Register *y* is subtracted (unsigned) from register *x*. The macro stores the result in register *x*.
OS21_POWER_PCODE_SUBAC	
OS21_POWER_PCODE_SUBBA	
OS21_POWER_PCODE_SUBBC	
OS21_POWER_PCODE_SUBCA	
OS21_POWER_PCODE_SUBCB	
CMP*x*(VALUE) pCode Macros	
OS21_POWER_PCODE_CMPA(VALUE)	Register *x* is COMPARED (unsigned) with VALUE, and the status flags are updated. The status flags can then be used to perform conditional jump operations.
OS21_POWER_PCODE_CMPB(VALUE)	
OS21_POWER_PCODE_CMPC(VALUE)	
CMP*xy* pCode Macros	
OS21_POWER_PCODE_CMPAB	Register *x* is COMPARED (unsigned) with register *y*, and the status flags are updated. The status flags can then be used to perform conditional jump operations.
OS21_POWER_PCODE_CMPAC	
OS21_POWER_PCODE_CMPBA	
OS21_POWER_PCODE_CMPBC	
OS21_POWER_PCODE_CMPCA	
OS21_POWER_PCODE_CMPCB	

Table 41. pCode macros (continued)

Macro Name	Description
B*xx* pCode Macros	
OS21_POWER_PCODE_BRA(LABELNUM)	Branch always to LABELNUM.
OS21_POWER_PCODE_BEQ(LABELNUM)	Branch to LABELNUM if the EQUAL status flag is set.
OS21_POWER_PCODE_BNE(LABELNUM)	Branch to LABELNUM if the EQUAL status flag is not set.
OS21_POWER_PCODE_BAC(LABELNUM)	Branch to LABELNUM if the ACTIVE STANDBY status flag is set.
OS21_POWER_PCODE_BPA(LABELNUM)	Branch toLABELNUM if the PASSIVE STANDBY status flag is set.
LIA*x* pCode Macros	
OS21_POWER_PCODE_LIAB(OFFSET)	Load the data at the address (A + OFFSET) into register *x*. Loads may be 32, 16 or 8 bits wide. LIA*x* is the same as LIA*x*32.
OS21_POWER_PCODE_LIAC(OFFSET)	
OS21_POWER_PCODE_LIAB32(OFFSET)	
OS21_POWER_PCODE_LIAC32(OFFSET)	
OS21_POWER_PCODE_LIAB16(OFFSET)	
OS21_POWER_PCODE_LIAC16(OFFSET)	
OS21_POWER_PCODE_LIAB8(OFFSET)	
OS21_POWER_PCODE_LIAC8(OFFSET)	
SIA*x* pCode Macros	
OS21_POWER_PCODE_SIAB(OFFSET)	Store register *x* to the address (A + OFFSET). Stores may be 32, 16 or 8 bits wide. SIA*x* is the same as SIA*x*32.
OS21_POWER_PCODE_SIAC(OFFSET)	
OS21_POWER_PCODE_SIAB32(OFFSET)	
OS21_POWER_PCODE_SIAC32(OFFSET)	
OS21_POWER_PCODE_SIAB16(OFFSET)	
OS21_POWER_PCODE_SIAC16(OFFSET)	
OS21_POWER_PCODE_SIAB8(OFFSET)	
OS21_POWER_PCODE_SIAC8(OFFSET)	
LAM*x* pCode Macros	
OS21_POWER_PCODE_LAMA(MEMORYNUM)	Load register *x* with the address of MEMORYNUM.
OS21_POWER_PCODE_LAMB(MEMORYNUM)	
OS21_POWER_PCODE_LAMC(MEMORYNUM)	

Table 41. pCode macros (continued)

Macro Name	Description
LM*x* pCode Macros	
OS21_POWER_PCODE_LMA(MEMORYNUM)	Load register *x* from MEMORYNUM. Loads may be 32, 16 or 8 bits wide. LM*x* is the same as LM*x*32.
OS21_POWER_PCODE_LMB(MEMORYNUM)	
OS21_POWER_PCODE_LMC(MEMORYNUM)	
OS21_POWER_PCODE_LMA32(MEMORYNUM)	
OS21_POWER_PCODE_LMB32(MEMORYNUM)	
OS21_POWER_PCODE_LMC32(MEMORYNUM)	
OS21_POWER_PCODE_LMA16(MEMORYNUM)	
OS21_POWER_PCODE_LMB16(MEMORYNUM)	
OS21_POWER_PCODE_LMC16(MEMORYNUM)	
OS21_POWER_PCODE_LMA8(MEMORYNUM)	
OS21_POWER_PCODE_LMB8(MEMORYNUM)	
OS21_POWER_PCODE_LMC8(MEMORYNUM)	
SM*x* pCode Macros	
OS21_POWER_PCODE_SMA(MEMORYNUM)	Store data from register *x* to MEMORYNUM. Stores may be 32, 16 or 8 bits wide. SM*x* is the same as SM*x*32.
OS21_POWER_PCODE_SMB(MEMORYNUM)	
OS21_POWER_PCODE_SMC(MEMORYNUM)	
OS21_POWER_PCODE_SMA32(MEMORYNUM)	
OS21_POWER_PCODE_SMB32(MEMORYNUM)	
OS21_POWER_PCODE_SMC32(MEMORYNUM)	
OS21_POWER_PCODE_SMA16(MEMORYNUM)	
OS21_POWER_PCODE_SMB16(MEMORYNUM)	
OS21_POWER_PCODE_SMC16(MEMORYNUM)	
OS21_POWER_PCODE_SMA8(MEMORYNUM)	
OS21_POWER_PCODE_SMB8(MEMORYNUM)	
OS21_POWER_PCODE_SMC8(MEMORYNUM)	
Control pCode Macros	
OS21_POWER_PCODE_SLEEP	Cause the CPU to sleep until the next interrupt arrives.
OS21_POWER_PCODE_EXIT	Terminate execution of pCode. The virtual machine returns the value of the A register to the calling power_level_set() function.

Table 41. pCode macros (continued)

Macro Name	Description
Label pCode Macros	
`OS21_POWER_PCODE_LABEL(LABELNUM)`	Insert a label at the point of insertion, identified by `LABELNUM` (which must be unique).
`OS21_POWER_PCODE_MEMORY(MEMORYNUM, VALUE)`	Reserve a word of memory at the point of insertion, identified by `MEMORYNUM` (which must be unique). The word is initialized with `VALUE`.

16.3.4 pCode example

The following example is a pCode table that can be used on the ST40 of the STi7111 chip to put the RAM into self refresh and sleep until the next interrupt arrives. The example uses the OS21 timeslice timer to wake up the SLEEP. Normally this would be an interrupt from something like an infra-red remote control.

```
static pcode_data_t pcode1[] =
{
  /*
   * Skip if we are going into ACTIVE_STANDBY
   */
  OS21_POWER_PCODE_BAC(3),


  /*
   * Send the LMI into self refresh
   */
  OS21_POWER_PCODE_LDA(0xFE001198),/* SYS_CFG38 */
  OS21_POWER_PCODE_ORA(1 << 20),
  OS21_POWER_PCODE_STA(0xFE001198),

  /*
   * Wait for ACK
   */
  OS21_POWER_PCODE_LABEL(1),
  OS21_POWER_PCODE_LDA(0xFE001018),/* SYS_STATUS4 */
  OS21_POWER_PCODE_ANDA(1 << 0),
  OS21_POWER_PCODE_CMPA(1 << 0),
  OS21_POWER_PCODE_BNE(1),

  /*
   * Disable the analogue input buffer of the pads.
   */
  OS21_POWER_PCODE_LDA(0xFE001130),/* SYS_CFG12 */
  OS21_POWER_PCODE_ORA(1 << 10),
  OS21_POWER_PCODE_STA(0xFE001130),

  /*
   * Power down LMI PLL
   */
  OS21_POWER_PCODE_LDA(0xFE00112C),/* SYS_CFG11 */
```

```
OS21_POWER_PCODE_ORA(1 << 12),
OS21_POWER_PCODE_STA(0xFE00112C),

/*
 * Wait for ACK
 */
OS21_POWER_PCODE_LABEL(2),
OS21_POWER_PCODE_LDA(0xFE001014),/* SYS_STATUS3 */
OS21_POWER_PCODE_ANDA(1 << 0),
OS21_POWER_PCODE_CMPA(1 << 0),
OS21_POWER_PCODE_BNE(2),

/*
 * Global power down
 */
OS21_POWER_PCODE_LDA(0xFE00111C),/* SYS_CFG7 */
OS21_POWER_PCODE_ORA(1 << 23),
OS21_POWER_PCODE_STA(0xFE00111C),

OS21_POWER_PCODE_LABEL(3),

/*
 * Start the timeslice timer - to interrupt and wake us up....
 */
OS21_POWER_PCODE_LDA8(0xFFD80004),
OS21_POWER_PCODE_ORA(0x2),
OS21_POWER_PCODE_STA8(0xFFD80004),

/*
 * Sleep
 */
OS21_POWER_PCODE_SLEEP,

/*
 * Jump to the RAM wakeup code if we are coming out of
 * PASSIVE_STANDBY, otherwise we can exit now.
 */
OS21_POWER_PCODE_BPA(4),

/*
 * Set exit argument and exit.
 */
OS21_POWER_PCODE_SETA(1),
OS21_POWER_PCODE_EXIT,

OS21_POWER_PCODE_LABEL (4),
/*
 * Enable the analogue input buffers of the pads.
 */
OS21_POWER_PCODE_LDA(0xFE001130),/* SYS_CFG12 */
OS21_POWER_PCODE_ANDA(~(1 << 10)),
OS21_POWER_PCODE_STA(0xFE001130),
```

```
  /*
   * Power on LMI PLL
   */
  OS21_POWER_PCODE_LDA(0xFE00112C),/* SYS_CFG11 */
  OS21_POWER_PCODE_ANDA(~(1 << 12)),
  OS21_POWER_PCODE_STA(0xFE00112C),

  /*
   * Wait for ACK
   */
  OS21_POWER_PCODE_LABEL(5),
  OS21_POWER_PCODE_LDA(0xFE001014),/* SYS_STATUS3 */
  OS21_POWER_PCODE_ANDA(1 << 0),
  OS21_POWER_PCODE_CMPA(1 << 0),
  OS21_POWER_PCODE_BEQ(5),

  /*
   * Exit LMI from self refresh
   */
  OS21_POWER_PCODE_LDA(0xFE001198),/* SYS_CFG38 */
  OS21_POWER_PCODE_ANDA(~(1 << 20)),
  OS21_POWER_PCODE_STA(0xFE001198),

  /*
   * Wait for ACK
   */
  OS21_POWER_PCODE_LABEL(6),
  OS21_POWER_PCODE_LDA(0xFE001018),/* SYS_STATUS4 */
  OS21_POWER_PCODE_ANDA(1 << 0),
  OS21_POWER_PCODE_CMPA(1 << 0),
  OS21_POWER_PCODE_BEQ(6),

  OS21_POWER_PCODE_SETA(2),
  OS21_POWER_PCODE_EXIT
};
```

16.4 Power management API summary

All definitions relating to the power management API are declared by including the header file `os21.h` which itself includes the header file `power.h`. See *Table 42* for the functions and *Table 43* for the types defined by `power.h`.

Table 42. Functions defined in power.h

Function	Description
`power_callback_add()`	Function to add a power callback
`power_callback_delete()`	Function to remove a power callback
`power_level_set()`	Function to set the current power level
`power_pcode_set()`	Function to set pCode to be used

Table 43. Types defined in power.h

Type	Description
power_callback_fn_t	A power callback function.
pcode_data_t	A pseudo-code (pcode) instruction or data.

16.5 Power management function definitions

power_callback_add

Add a power management callback

Definition:

```
#include <os21.h>

int power_callback_add(
  power_callback_fn_t fn,
  unsigned int order);
```

Arguments:

power_callback_fn_t fn	Function to add.
unsigned int order	Number specifying order in call chain.

Returns: OS21_SUCCESS for success.

OS21_FAILURE on error.

Errors: fn is NULL.

fn already added.

Order out of range.

Not enough memory.

Order is OS21_POWER_CALLBACK_ORDER_FIRST and a callback has already been added with the same order.

Order is OS21_POWER_CALLBACK_ORDER_LAST and a callback has already been added with the same order.

Context: Callable from task context.

Description: Adds a function to be called to the list of functions to be called when OS21 transitions from one power mode to another. Functions are called in ascending order when going to a standby power mode, in reverse mode when coming from a standby power mode. The order must be in the range OS21_POWER_CALLBACK_ORDER_FIRST to OS21_POWER_CALLBACK_ORDER_LAST inclusive and only one function can be given the order OS21_POWER_CALLBACK_ORDER_FIRST, and only one function can be given the order OS21_POWER_CALLBACK_ORDER_LAST.

See Also: power_level_set()

power_callback_delete

Remove a power management callback

Definition:

```
#include <os21.h>

int power_callback_delete(power_callback_fn_t fn);
```

Arguments:

`power_callback_fn_t fn`	Function to remove.

Returns: `OS21_SUCCESS` for success.

`OS21_FAILURE` on error.

Errors: `fn` is `NULL`.

`fn` not in list.

Context: Callable from task context.

Description: Removes a function from the list of functions to be called when OS21 transitions from one power mode to another.

See Also: `power_level_set()`

power_level_set

Set the power level

Definition:

```
#include <os21.h>

int power_level_set(
  unsigned int level,
  unsigned int pCodeArgIn,
  unsigned int * pCodeArgOutp);
```

Arguments:

`unsigned int level`	Power level to set.
`unsigned int pCodeArgIn`	Argument to pass to pCode, placed in register A.
`unsigned int * pCodeArgOutp`	Pointer to location to place pCode result, taken from register A.

Returns: `OS21_SUCCESS` for success.

`OS21_FAILURE` on error.

Errors: The function is called from system context.

Level is not one of `OS21_POWER_LEVEL_ACTIVE_STANDBY` or `OS21_POWER_LEVEL_PASSIVE_STANDBY`.

Level is `OS21_POWER_LEVEL_PASSIVE_STANDBY`, but the caches are disabled, or there are no caches.

A power callback returns `OS21_FAILURE` when called.

Context: Callable from task context.

Description: `power_level_set()` causes the system to transition to the given level. All the power callbacks are called in order, and then the set pCode (if any) is run, with ACTIVE and PASSIVE status flags set to correspond with the selected level. The system then transitions back to `OS21_POWER_LEVEL_ON`, calling the power level callbacks in reverse order before returning.

`power_level_set()` is always called from the power level `OS21_POWER_LEVEL_ON`, and the system is back in `OS21_POWER_LEVEL_ON` when it returns. Passing `OS21_POWER_LEVEL_ON` as the level is therefore an error; the level must always be one of `OS21_POWER_LEVEL_ACTIVE_STANDBY` or `OS21_POWER_LEVEL_PASSIVE_STANDBY`.

It is an error to specify `OS21_POWER_LEVEL_PASSIVE_STANDBY` if there are no caches, or the caches are disabled. This is because it is assumed that the RAM will be put into self refresh, and therefore the pCode (if supplied) must be run entirely from cache. OS21 loads the pCode and its pCode interpreter into the cache before executing it, so no action is required by the user in this respect.

The value passed into `pCodeArgIn` is placed in the register A before pCode execution starts. When pCode execution is complete, the contents of the register A are copied to the location specified by `pCodeArgOutp`, providing that it is non-`NULL`.

The PASSIVE and ACTIVE flags (see the pCode instructions BPA, BAC) are set up in line with the specified power level. This allows the pCode to execute code conditionally based on the power level being entered.

For example, if the power level is `OS21_POWER_LEVEL_ACTIVE_STANDBY`, the pCode can use the flags to skip pCode that puts the RAM into self refresh.

The pCode to be executed (if any) is supplied by the user by calling the `power_pcode_set()` function.

If a callback fails during the transition to the specified power level, then the other callbacks called up to that point are called in reverse order with a parameter of `OS21_POWER_LEVEL_ON` before returning with a failure.

During a call to `power_level_set()`, the OS21 timers are stopped and restarted before the call completes. Therefore, on return from a call to `power_level_set()`, it may be necessary to resynchronize with the real time.

See Also: `power_callback_add()`, `power_callback_delete()`, `power_pcode_set()`

power_pcode_set

Set the pcode to be executed

Definition:

```
#include <os21.h>

int power_pcode_set(
  pcode_data_t * pcode,
  unsigned int sizePCode);
```

Arguments:

`pcode_data_t * pcode`	Pointer to the pCode, which is an array of `pcode_data_t` objects.
`unsigned int sizePCode`	The size of the `pcode` array in bytes.

Returns: `OS21_SUCCESS` for success.

`OS21_FAILURE` on error.

Errors: `pcode` is `NULL`.

`sizePCode` is 0.

`sizePCode` is not a multiple of `sizeof(pcode_data_t)`.

Not enough memory.

Validation checks on the pCode fail.

Context: Callable from task context.

Description: `power_pcode_set()` installs `pcode` as the pCode to be executed from `power_level_set()`. It performs a number of basic checks on the pCode, and fails if an error is found.

Errors include invalid instructions, missing labels, and a missing exit instruction. Note that not all pCode errors can be detected - it is possible to write pCode that can hang, crash, or produce undefined results.

See Also: `power_level_set()`

16.6 Interrupt management in pCode

When pCode is being run, interrupt handling is disabled. Normal interrupt handling is resumed only when the system is brought back into the `OS21_POWER_LEVEL_ON` state. At this point, any pending interrupts are dispatched to the appropriate handler in the usual way.

The pCode sleep instruction, `OS21_POWER_PCODE_SLEEP`, waits for an interrupt to occur, but no handling of the interrupt takes place.

It is possible to poll for interrupts from within pCode, and to deal with them, but take care not to alter the state or the hardware being maintained by OS21 or other drivers. Failure to do this may result in undefined behavior.

16.7 Exceptions in pCode

No exception or fault handling is provided when pCode is running. It is up to the user to ensure that pCode is correct and does not cause any exceptions.

17 Board support package

17.1 Board support package overview

OS21 Board Support Packages (BSPs) are supplied for all supported platforms, both as pre-built libraries and as accompanying sources. A BSP consists of various board, chip and CPU dependent declarations as well as some generic configuration options. The BSP declarations provide the following:

- allow customization of OS21
- describe the interrupt subsystem to OS21
- describe any required MMU mappings to OS21
- provide "hooks" to allow the user to insert code to be executed at certain key OS21 events

To achieve this, the BSP exports the following to OS21:

- variables that determine how OS21 operates
- functions, which OS21 can call on key events
- a description of the interrupt system, which is made up of tables and declarations of interrupt names for these tables
- an optional list of MMU mappings in the form of a mappings table

Many of the functions and variables in OS21 are defined in the supplied BSPs as “weak”, which means they can easily be overridden in user code.

The source code for each BSP is partitioned into four sections:

- a source file for generic configuration options and hook functions
- a source file for the CPU
- a source file for the chip
- a source file for the board

Combining these four source files provides a complete BSP for a given target.

17.2 BSP data

The BSP can export data to OS21, allowing a degree of customization. The following data is exported to OS21 on all targets:

- timeslice frequency
- board crystal frequency
- callback enable flag

In addition to these, there may also be other target-specific data items. Target-specific data items, where they exist, are described in the appropriate OS21 manual for the target.

OS21 timeslice frequency

```
unsigned int bsp_timeslice_frequency_hz;
```

This variable informs OS21 of the desired timeslice frequency in hertz. It is the number of times per second that a timeslice occurs when timeslicing is switched on.

OS21 panics if you try to set this to either an invalid or an unrealistic value, that is, less than 1 or greater than 500. The default value is 50.

This value is weakly defined and may be changed in any of the following ways.

- Change the value in the supplied source file (`src/os21/bsp/bsp.c`), recompile the BSP and relink.
- Change the value in user code before initializing and starting the OS21 kernel. For example:

  ```
  extern unsigned int bsp_timeslice_frequency_hz;
  bsp_timeslice_frequency_hz = 100;
  ```

- Override the weak definition by inserting your own declaration in user code. For example:

  ```
  unsigned int bsp_timeslice_frequency_hz = 25;
  ```

Timeslicing is switched on or off using the `kernel_timeslice()` function. See *Section 2.3 on page 18* for information about this function.

Board crystal frequency

```
unsigned int bsp_xtal_frequency_hz;
```

This variable informs OS21 of the frequency (in hertz) of the on-board crystal. This value is not used by OS21 directly, but will be used elsewhere in the BSP when determining the input clock frequency. Usually the BSP function `bsp_timer_input_clock_frequency_hz()` makes use of this value.

This value is weakly defined and may be changed in any of the following ways.

- Change the value in the supplied source file (usually `src/`*`platform`*`/bsp/board_`*`platform`*`.c`, where *`platform`* is the name of the reference platform), recompile the BSP and relink.
- Change the value in user code before initializing and starting the OS21 kernel. For example:

  ```
  extern unsigned int bsp_xtal_frequency_hz;
  bsp_xtal_frequency_hz = 27000000;/* 27 MHz clock */
  ```

- Override the weak definition by inserting your own declaration in user code. For example:

  ```
  unsigned int bsp_xtal_frequency_hz = 33000000;/* 33 MHz clock */
  ```

See *Section 17.3 on page 209* for details of the `bsp_timer_input_clock_frequency_hz()` function.

OS21 callbacks enabled

```
unsigned int bsp_callbacks_enabled;
```

This variable informs OS21 whether to enable the callback API. By default the callback API is enabled, but a small performance benefit can be obtained by switching it off if it is not required. A zero value switches callbacks off, a non-zero value switches callbacks on.

This value is weakly defined and may be changed in any of the following ways.

- Change the value in the supplied source file (`src/os21/bsp/bsp.c`), recompile the BSP and relink.
- Change the value in user code before initializing and starting the OS21 kernel. For example:

  ```
  extern unsigned int bsp_callbacks_enabled;
  bsp_callbacks_enabled = 0;
  ```

- Override the weak definition by inserting your own declaration in user code. For example:

  ```
  unsigned int bsp_callbacks_enabled = 0;
  ```

Full details of the callback API can be found in *Chapter 5: Callbacks on page 89* of this manual.

17.3 BSP functions summary

The BSP also exports some functions as part of the BSP. These are “hooks” from OS21 into user defined functions. This allows the user to insert code at key OS21 events. The functions listed in *Table 44* are exported to OS21 on all targets:

Table 44. Functions exported by the board support package

Function	Description
`bsp_timer_input_clock_frequency_hz`	Return the frequency of the input clock.
`bsp_initialize`	The OS21 initialize hook function.
`bsp_start`	The OS21 start hook function.
`bsp_exp_handler`	The OS21 exception hook function.
`bsp_panic`	The OS21 panic hook function.
`bsp_shutdown`	The OS21 shutdown hook function.
`bsp_terminate`	The OS21 terminate hook function.
`bsp_board_type`	Returns the board type.
`bsp_chip_type`	Returns the chip type.
`bsp_cpu_type`	Returns the CPU type.

There may also be target specific functions. If these exist, they are described in the appropriate OS21 manual for that target.

17.4 BSP function definitions

bsp_timer_input_clock_frequency_hz

OS21 input clock frequency

Definition:

```
#include <os21.h>
unsigned int bsp_timer_input_clock_frequency_hz (void)
```

Arguments: None.

Returns: The input clock frequency.

Description: OS21 calls this function to discover the input clock frequency to the timer units. The function may either return the frequency directly, or it may read a series of configuration registers to determine the value. For example, it may read CLOCKGEN registers and use these values with the board crystal frequency (`bsp_xtal_frequency_hz`) to calculate the actual timer input clock frequency.

This function is weakly defined and may be changed in any of the following ways.

– Change the implementation in the supplied source file (usually `src/`*`platform`*`/bsp/chip_`*`variant`*`.c`, where *`variant`* is the name of the SoC device), recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
unsigned int bsp_timer_input_clock_frequency_hz (void)
{
          return (32768);/* Directly return 32.768 kHz clock */
}
```

The implementation of this function may make use of the `bsp_xtal_frequency_hz` value which is normally defined in the BSP. More information about this variable is given in *Section 17.2 on page 207*.

bsp_initialize

OS21 initialize hook

Definition:

```
#include <os21.h>
void bsp_initialize (void);
```

Arguments: None.

Returns: None.

Description: OS21 calls this function prior to initialization in `kernel_intialize()`. It provides users with the facility to add code to be executed just prior to kernel initialization. It provides a hook where users can change aspects of kernel behavior, and perform board specific initialization.

This function is weakly defined and may be changed in any of the following ways.

– Change the implementation in the supplied source file (`src/os21/bsp/bsp.c`), recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
void bsp_initialize (void)
{
          printf ("OS21 initializing\n");
}
```

bsp_start

OS21 start hook

Definition:

```
#include <os21.h>
void bsp_start (void);
```

Arguments: None.

Returns: None.

Description: OS21 calls this function following kernel startup in `kernel_start()`. It provides users with the facility to add code to be executed just after kernel startup. It provides a hook where users can add final board initialization code.

This function is weakly defined and may be changed in any of the following ways.

– Change the implementation in the supplied source file (`src/os21/bsp/bsp.c`), recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
void bsp_start (void)
{
          printf ("OS21 starting\n");
}
```

bsp_exp_handler

OS21 exception hook

Definition:

```
#include <os21.h>
void bsp_exp_handler (unsigned int exp_code);
```

Arguments:

`exp_code`	A code that identifies the exception that has occurred.

Returns: None.

Description: OS21 calls this function whenever it takes an unexpected exception. The function receives a single parameter describing the exception that occurred. When this routine returns, the kernel announces the exception to the console and enters a tight spin with interrupts disabled.

This function is weakly defined and may be changed in any of the following ways.

– Change the implementation in the supplied source file (`src/os21/bsp/bsp.c`), recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
void bsp_exp_handler (unsigned int exp_code)
{
          printf("OS21 took unexpected exception: 0x%x\n",
                                                  exp_code);
}
```

bsp_panic

OS21 panic hook

Definition:

```
#include <os21.h>
void bsp_panic (const char * message);
```

Arguments:

`message`	Message to the user to indicate the nature of the panic.

Returns: None.

Description: OS21 calls this function whenever it detects an internal error and panics. The function receives a single parameter, a pointer to a character string describing the panic that occurred. When this routine returns, the kernel announces the panic to the console and enters a tight spin with interrupts disabled.

This function is weakly defined and may be changed in a number of ways.

– Change the implementation in the supplied source file (`src/os21/bsp/bsp.c`) and recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
void bsp_panic (const char * message)
{
          printf ("OS21 about to panic: %s\n", message);
}
```

bsp_shutdown

OS21 shutdown hook

Definition:

```
#include <os21.h>
void bsp_shutdown (void);
```

Arguments: None.

Returns: None.

Description: OS21 calls this function when it shuts down as a result of `exit()` being called. When this routine returns, the kernel proceeds with its shutdown sequence, and finally enters a tight spin with interrupts disabled.

This function is weakly defined and may be changed in any of the following ways.

– Change the implementation in the supplied source file (`src/os21/bsp/bsp.c`), recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
void bsp_shutdown (void)
{
          printf ("OS21 shutting down\n");
}
```

bsp_terminate

OS21 terminate hook

Definition:
```
#include <os21.h>
void bsp_terminate (void);
```

Arguments: None.

Returns: None.

Description: OS21 calls this function when it takes an illegal exception, or detects an internal error, and it is not connected to a debugger. If this function returns, the kernel enters a tight spin with interrupts disabled.

This function is weakly defined and may be changed in any of the following ways.

– Change the implementation in the supplied source file (`src/os21/bsp/bsp.c`) and recompile the BSP and relink.

– Override the weak function definition with your own implementation. For example:

```
void bsp_terminate (void)
{
          printf ("OS21 terminating\n");
}
```

bsp_board_type

Return board type

Definition:
```
#include <os21.h>
const char * bsp_board_type (void);
```

Arguments: None.

Returns: A string describing the board type.

Description: This function returns a string describing the board type. This is the value returned by a call to `kernel_board ()`.

The function is not weakly defined. The string may be changed by changing the source file (usually `src/`*`platform`*`/bsp/board_`*`platform`*`.c`, where *`platform`* is the name of the reference platform), recompiling the BSP and relinking.

bsp_chip_type

Return chip type

Definition:
```
#include <os21.h>
const char * bsp_chip_type (void);
```

Arguments: None.

Returns: A string describing the chip type.

Description: This function returns a string describing the chip type. This is the value returned by a call to `kernel_chip()`.

The function is not weakly defined. The string may be changed by changing the source file (usually `src/`*`platform`*`/bsp/chip_`*`variant`*`.c`, where *`variant`* is the name of the SoC device), recompiling the BSP and relinking.

bsp_cpu_type

Return CPU type

Definition:
```
#include <os21.h>
const char * bsp_cpu_type (void);
```

Arguments: None.

Returns: A string describing the CPU type.

Description: This function returns a string describing the CPU type. This will be the value returned by a call to `kernel_cpu()`.

The function is not weakly defined. The string may be changed by changing the source file (usually `src/`*`platform`*`/bsp/cpu_`*`variant`*`.c`, where *`variant`* is the name of the SoC device), recompiling the BSP and relinking.

17.5 BSP interrupt system description

The BSP is responsible for describing the interrupt system to OS21. The BSP interrupt system description and the platform specific interrupt code together implements OS21's generic interrupt API.

The BSP interrupt system description is platform specific. Full details are provided in the OS21 manual for the given target.

17.6 BSP MMU mappings description

When OS21 starts, it inherits the MMU state and mappings constructed by the toolset. After any pre-existing mappings have been added to the OS21 page tables, additional mappings can be created by placing a mapping table in the BSP. The use of this table is optional. Any mappings in this table are created as fixed mappings and exist for the lifetime of the system. If OS21 cannot create any of these mappings (this is normally because they interfere with previous mappings made by the toolset) then the OS21 startup fails.

17.6.1 Mapping table

The mapping table in the BSP is a list of `mapping_table_entry_t` types. A `mapping_table_entry_t` is defined as follows:

```
typedef struct mapping_table_entry_s
{
  unsigned int pAddr;
  unsigned int vAddr;
  unsigned int size;
  unsigned int mode;

} mapping_table_entry_t;
```

`pAddr` is the physical start address of the mapping. `vAddr` is the virtual start address of the mapping. `size` is the size of the mapping, and `mode` is the mode of the mapping, made up by logically ORing a series of `vmem_create()` flags. For details of these flags see the `vmem_create()` function description (see *vmem_create on page 175*).

`mapping_table_entry_t bsp_mapping_table []` describes the list of mappings to be created by OS21 upon startup. For example:

```
mapping_table_entry_t bsp_mapping_table [] =
{
  { 0x00000000, 0x00000000, 0x08000000, VMEM_CREATE_READ |
VMEM_CREATE_WRITE | VMEM_CREATE_EXECUTE | VMEM_CREATE_CACHED },
  { 0x10000000, 0x10000000, 0x00230000, VMEM_CREATE_READ |
VMEM_CREATE_WRITE | VMEM_CREATE_UNCACHED |
VMEM_CREATE_NO_WRITE_BUFFER },
  { 0x30000000, 0x30000000, 0x02000000, VMEM_CREATE_READ |
VMEM_CREATE_WRITE | VMEM_CREATE_EXECUTE | VMEM_CREATE_CACHED },
  { 0x90000000, 0x90000000, 0x10000000, VMEM_CREATE_READ |
VMEM_CREATE_WRITE | VMEM_CREATE_UNCACHED |
VMEM_CREATE_NO_WRITE_BUFFER }
};
```

17.7 Level 2 cache support

OS21 can drive a level 2 cache controller if one is present and if its base address is provided to OS21. This is done using the BSP variable `bsp_l2cache_base_address`.

For example:

```
void * bsp_l2cache_base_address = 0xFD130000;
```

If the variable is not defined in the BSP, or if it is set to NULL, OS21 does not attempt to operate the level 2 cache controller, even if one is present. If the variable is defined and non-NULL, then OS21 attempts to operate a level 2 cache controller at the given address. If there is no level 2 cache controller present at the given address, then undefined behavior will result.

18 Revision history

Table 45. Document revision history

Date	Revision	Changes
12-Aug-2010	V	Updated *vmem_create on page 175* and *vmem_delete on page 177*.
1-Dec-2009	U	Revised *Section 12.1: Caches and memory overview on page 163* to include level 2 cache. Added *Section 16.6: Interrupt management in pCode on page 206* and *Section 16.7: Exceptions in pCode on page 206*. Added *Section 17.7: Level 2 cache support on page 217*.
2-Jun-2009	T	Updated *Chapter 5: Callbacks on page 89* to add four `callback_exception_*()` functions. Added *task_lock_task* and *task_unlock_task* to *Chapter 4: Tasks on page 44*. Added *Chapter 16: Power management on page 192* and a reference to the power management facility in *Section 1.17: Power management on page 15*.
30-Oct-2008	S	Several minor revisions have been made throughout the manual. `profile.h` and `cache.h` added to *Table 1: OS21 include files on page 9*. Added *Section 15.6: Profile data binary file format on page 186*. BSP variable bsp_timelogging_enabled has been removed from *Section 17.2: BSP data*.
9-May-2008	R	Added *Section 17.6: BSP MMU mappings description on page 216*.
12-Nov-2007	Q	Updated details of *kernel_idle* and *kernel_time* in *Chapter 2: Kernel on page 17*. Updated details of *task_status* in *Chapter 4: Tasks on page 44*. Updated *Chapter 5: Callbacks on page 89*. Added new *Chapter 17: Board support package on page 207*.
17-May-2007	P	Added *12.5: Cache function definitions on page 165* as this is now generic to both platforms. Updated *Chapter 13: Virtual memory on page 173* in response to user comments.
18-Jan-2007	O	Added *Chapter 13: Virtual memory on page 173*. Throughout: Edited to include virtual memory.
27-Nov-2006	N	Moved to new template. No technical changes.

Table 45. Document revision history (continued)

Date	Revision	Changes
Jul 06	M	Throughout: Updated function context information. Introduction: Added Exceptions section. Kernel: Added kernel_printf(). Tasks: Updated task_context() definition. Added task_stackinfo() and task_stackinfo_set(). Mutexes: Added mutex_create_priority_noinherit() and mutex_create_priority_noinherit_p(). Interrupts: Added Contexts and interrupt handler code section Exceptions: Added as new chapter. Profiling: Added profile_deinit() and updated the errors given for the other functions.
Apr 05	L	Interrupts: Updated Context section for interrupt_handle(). Profiling: Added if a task is deleted, before the profiler has been stopped, the task is removed from the profiler data. Changed the order of the files provided to perl -w os21prof.pl.
Oct 04	K	Profiling: Added as new chapter.
Sep 03	J	Interrupts: Updated Context section for interrupt_unmask().
Jul 03	I	Introduction: Removed note that some systems do not support long long. Tasks: In Scheduling, added details of pre-emption and locked tasks. Real-time clocks: Removed note that some systems do not support long long. Interrupts: Rephrased the Initializing the interrupt handling subsystem section. Added to note in Obtaining a handle for an interrupt. Changed Raising interrupts to include that hardware support is required.

Table 45. Document revision history (continued)

Date	Revision	Changes
Apr 03	H	Introduction: Changed references to “high resolution 64-bit timers” to “high resolution timers”. Added note that some systems do not support long long. Tasks: Removed references to interrupt_lock() and interrupt_unlock(). Added examples of a platform header file. Real-time clocks: Added note that some systems do not support long long. Interrupts: Updated overview to remove incorrect statement. In Initializing the interrupt handling subsystem, change external interrupts to interrupts. Added note in Obtaining a handle for an interrupt. Combined the sections Attaching interrupt handlers and Chained interrupt handlers. Updated results and description of interrupt_handler(), interrupt_poll() and interrupt_unraise(). Added note that function is deprecated to interrupt_lock() and interrupt_unlock(). Updated description of interrupt_mask() and interrupt_mask_all().
Mar 03	G	Introduction: Updated chapter to allow for interrupts. Kernel: Added new kernel_chip function. Interrupts: Added as new chapter.
Aug 02	F	Throughout: Added Context section to each of the functions.
Aug 02	E	Throughout: Added note to all relevant functions about null pointers being passed instead of a valid partition pointer. Memory and partitions: Changed the names of the flags available for partition_status_type. Tasks: Added task_yield and updated task_reschedule and all references. Callbacks: Renamed callback_interrupt_delete to callback_interrupt_uninstall.
May 02	D	Changed the definition of the following functions: semaphore_delete, mutex_delete, event_group_delete, message_delete_queue and task_create. Added the following functions: kernel_board and kernel_cpu. Corrected minor typing and grammatical errors throughout.
Feb 02	C	Clarifications denoted by Change Bars for Beta Release of R2.0.
Dec 01	B	Clarifications denoted by Change Bars.
Nov 01	A	Initial release.

Index

F

H

I

K

M

N

O

P

R

S

T

U

V

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST REPRESENTATIVE, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER’S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com