Information furnished is believed to be accurate and reliable. However, STMicroelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of STMicroelectronics. Specification mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. STMicroelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of STMicroelectronics.

STMicroelectronics GROUP OF COMPANIES
Australia - Brazil - Canada - China - France - Germany - Italy - Japan - Korea - Malaysia - Malta - Mexico - Morocco
The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.

Document number: 72-TDS-525-03
# Contents

Preface .................................................................................. v

STLite/OS20 Real-Time Kernel ............................................. 1

1 Introduction ....................................................................... 3
   1.1 Overview ................................................................. 3
   1.2 Classes and Objects ............................................... 4
   1.3 Defining memory partitions .................................... 6
   1.4 Tasks .................................................................... 7
   1.5 Priority .................................................................. 7
   1.6 Semaphores ........................................................... 8
   1.7 Message queues .................................................... 8
   1.8 Clocks .................................................................. 8
   1.9 Interrupts ............................................................... 9
   1.10 Device ID .............................................................. 9
   1.11 Cache ................................................................. 9
   1.12 Processor specific functions ................................. 9

2 Getting Started ................................................................. 11
   2.1 Building for STLite/OS20 ...................................... 11
   2.2 Starting STLite/OS20 Manually ............................... 17

3 Kernel ............................................................................... 19
   3.1 Implementation .................................................... 19
   3.2 STLite/OS20 kernel ............................................... 19
   3.3 Kernel header file: kernel.h ................................. 20
   3.4 Kernel function definitions .................................. 21

4 Memory and partitions ...................................................... 25
   4.1 Partitions .............................................................. 25
   4.2 Allocation strategies ............................................ 25
   4.3 Pre-defined partitions ......................................... 27
   4.4 Obtaining information about partitions ................. 28
   4.5 Partition header file: partitio.h ............................ 30
   4.6 Memory and partition function definitions ............ 31

5 Tasks ............................................................................... 45
   5.1 STLite/OS20 tasks ............................................... 45
   5.2 Implementation of priority and timeslicing .......... 46
   5.3 STLite/OS20 priorities .......................................... 48
5.4 Scheduling ................................................. 48
5.5 Creating and running a task .............................. 49
5.6 Synchronizing tasks ........................................ 51
5.7 Communicating between tasks ........................... 51
5.8 Timed delays ............................................... 51
5.9 Rescheduling ............................................... 52
5.10 Suspending tasks ......................................... 52
5.11 Killing a task ............................................. 53
5.12 Getting the current task’s id .............................. 53
5.13 Stack usage ............................................... 54
5.14 Task data .................................................. 56
5.15 Task termination .......................................... 57
5.16 Waiting for termination .................................. 58
5.17 Deleting a task ............................................ 58
5.18 Task header file: task.h .................................. 59
5.19 Task function definitions ................................. 61
6 Semaphores .................................................. 101
6.1 Overview ................................................... 101
6.2 Use of Semaphores ........................................ 103
6.3 Semaphore header file: semaphor.h ....................... 104
6.4 Semaphore function definitions ........................... 106
7 Message handling ............................................ 119
7.1 Message queues ............................................ 119
7.2 Creating message queues .................................. 120
7.3 Using message queues ..................................... 122
7.4 Message header file: message.h ......................... 123
7.5 Message handling function definitions .................. 125
8 Real-time clocks .............................................. 137
8.1 ST20-C1 clock peripheral ................................ 137
8.2 The ST20 timers on the ST20-C2 ......................... 137
8.3 Reading the current time .................................. 138
8.4 Time arithmetic ............................................ 138
8.5 Time header file: ostime.h ................................. 140
8.6 STLite/OS20 time function definitions ................... 141
9 Interrupts ..................................................... 145
9.1 Interrupt models .......................................... 145
9.2 Selecting the correct interrupt handling system ........ 147
9.3 Initializing the interrupt handling support system ........ 149
9.4 Attaching an interrupt handler in STLite/OS20 ........................................ 150
9.5 Initializing the peripheral device ............................................................... 153
9.6 Enabling and disabling interrupts ............................................................ 153
9.7 Example: setting an interrupt for an ASC .............................................. 154
9.8 Locking out interrupts .............................................................................. 155
9.9 Raising interrupts ..................................................................................... 156
9.10 Retrieving details of pending interrupts .................................................. 156
9.11 Clearing pending interrupts ..................................................................... 156
9.12 Changing trigger modes .......................................................................... 157
9.13 Low power modes and interrupts ............................................................ 157
9.14 Obtaining information about interrupts ................................................. 157
9.15 Uninstalling interrupt handlers and deleting interrupts ......................... 158
9.16 Restrictions on interrupt handlers ......................................................... 159
9.17 Interrupt header file: interrup.h .............................................................. 159
9.18 Interrupt function definitions ................................................................. 162
10 Device information ...................................................................................... 195
10.1 Device ID header file: device.h ............................................................... 195
10.2 Device ID function definitions ............................................................... 197
11 Caches ......................................................................................................... 199
11.1 Introduction ............................................................................................... 199
11.2 Initializing and configuring the cache support system ............................ 199
11.3 Enabling the caches ............................................................................... 200
11.4 Locking the cache configuration ............................................................. 200
11.5 Example: setting up the caches .............................................................. 201
11.6 Flushing and invalidating caches ............................................................. 201
11.7 Cache header file: cache.h ..................................................................... 202
11.8 Cache function definitions ..................................................................... 203
12 ST20-C1 specific features ............................................................................ 213
12.1 Building the plug-in module .................................................................. 215
13 ST20-C2 specific features ............................................................................ 217
13.1 Channels .................................................................................................. 218
13.2 Two dimensional block move support ................................................... 238
Appendices ...................................................................................................... 243
A ST20-MC2 plug-in timer module ................................................................. 245
A.1 Overview ................................................................................................. 245
A.2 PWM peripheral ...................................................................................... 246
Contents

A.3 ST20-MC2 Plug-in timer functions .................................................. 247

B Using RCUs with STLite/OS20 ......................................................... 253
B.1 Recommendations ................................................................. 253
B.2 Tips and Tricks ................................................................. 254

C Compiling and configuring ......................................................... 257
C.1 Runtime configuration ........................................................ 257
C.2 Compiling STLite/OS20 ........................................................ 259
C.3 Compilation option file: conf.h ............................................. 260
C.4 Performance considerations ................................................ 262
C.5 Callback functions ............................................................ 263

Index ................................................................. 265
Preface

This manual forms a combined user guide and reference manual for the STLite/OS20 Real-Time Kernel.

About this document set

The document set provided with the toolset comprises:

- **ST20 Embedded Toolset Delivery manual** - provides installation instructions and a summary of the release.
- **ST20 Embedded Toolset User manual** - provides user and reference information to support the ST20 Embedded Toolset.
- **STLite/OS20 Real-Time Kernel Reference manual** - (this manual) provides function descriptions and usage details for each of the services provided, e.g., tasks, semaphores, message handling, memory management, timers, interrupts etc. A getting started chapter is provided and more specialist topics are covered.

Details of the CPU core used and any known bugs associated with a particular version of a silicon device can be found in the bug list for that version of the device.
Preface

Conventions used
The following typographical conventions may occur in this manual:

**Bold type**  Used to denote special terminology e.g. register or pin names.

**Teletype**  Used to distinguish command options, command line examples, code fragments, and program listings from normal text.

**Italic type**  In command syntax definitions, used to represent an argument or parameter. Used within text for emphasis and for book titles.

*Braces* { }  Used to denote a list of optional items in command syntax.

*Brackets* [ ]  Used to denote optional items in command syntax.

**Ellipsis** . . .  In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.

|  In command syntax, separates two mutually exclusive alternatives.

A change bar in the left margin, indicates a change to the previous version of the manual. This may indicate a change to the functionality of the toolset or merely an updated description. The delivery manual which accompanies this release, summarizes the main functional changes to the product.
STLite/OS20 Real-Time Kernel
1 Introduction

Multi-tasking is widely accepted as an optimal method of implementing real-time systems. Applications may be broken down into a number of 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 via interrupts.

The STLite/OS20 real-time kernel provides comprehensive multi-tasking services: Tasks synchronize their activities and communicate with each other via semaphores and message queues. Real world events are handled via interrupt routines and communicated to tasks using semaphores. Memory allocation for tasks is selectively managed by STLite/OS20 or the user and tasks may be given priorities and are scheduled accordingly. Timer functions are provided to implement time and delay functions.

The STLite/OS20 real-time kernel is common across all ST20 microprocessors, facilitating the portability of code. The kernel is re-implemented for each core, taking advantage of chip-specific features to produce a highly efficient multi-tasking environment for embedded systems developed for the ST20.

The API (Application Programming Interface) defined in this document corresponds to the 2.07 version of STLite/OS20.

1.1 Overview

The STLite/OS20 kernel features:

- A high degree of hardware integration.
- Multi-priority pre-emptive scheduling based on sixteen levels of priority.
- Semaphores.
- Message queues.
- Timers.
- Memory management.
- Interrupt handling.
- Very small memory requirement.
- Context switch time of 6 microseconds or less.
- Common across all ST20 microprocessors.

Each STLite/OS20 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 1.1.
- all the symbols defined by a service have the service name as the first component of the name, see below.
1.2 Classes and Objects

STLite/OS20 uses an object oriented style of programming. This will be familiar to many people from C++, however it is useful to understand how this has been applied to STLite/OS20, and how it has been implemented in the C language.

### 1.1.1 Naming

All the functions in STLite/OS20 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, for example, most `interrupt_` functions use interrupt levels, however those with a `_number` suffix use interrupt numbers.

### 1.1.2 How this manual is organized

The division of STLite/OS20 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 STLite/OS20 is founded. It is advisable to read the remainder of this chapter if you are a first time user.

A ‘Getting started’ which describes how to start using STLite/OS20 is provided in Chapter 2.

### 1.2 Classes and Objects

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan.h</td>
<td>ST20-C2 specific functions</td>
</tr>
<tr>
<td>device.h</td>
<td>Device information functions</td>
</tr>
<tr>
<td>interrup.h</td>
<td>Interrupt handling support functions</td>
</tr>
<tr>
<td>kernel.h</td>
<td>Kernel functions</td>
</tr>
<tr>
<td>message.h</td>
<td>Message handling functions</td>
</tr>
<tr>
<td>move2d.h</td>
<td>Two dimensional block move functions (ST20-C2 specific).</td>
</tr>
<tr>
<td>ostime.h</td>
<td>Timer functions</td>
</tr>
<tr>
<td>partitio.h</td>
<td>Memory functions</td>
</tr>
<tr>
<td>semaphor.h</td>
<td>Semaphore functions</td>
</tr>
<tr>
<td>tasks.h</td>
<td>Task functions</td>
</tr>
</tbody>
</table>

Table 1.1 STLite/OS20 header files
Each of the major services of STLite/OS20 is represented by a class, i.e.:

- Memory partitions.
- Tasks.
- Semaphores.
- Message queues.
- Channels.

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, and so consists of a data structure in memory which describes the current state of the object, together with information which describes how operations which are applied to that object will affect it, and the rest of the system.

For many classes within STLite/OS20, there are different flavors. For example, the semaphore class has FIFO and priority flavors. When a particular object is created, which flavor is required must be specified by using a qualifier on the object creation function, and that is then 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 will behave may depend on the flavor of that class. So the exact behavior of `semaphore_wait()` will depend 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.

**Warning:** the internal layout of any of the structure should not be referenced directly. This can and does change between implementations and releases, although the size of the structure will not change.

To provide this abstraction within STLite/OS20, 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.

### 1.2.1 Object Lifetime

All objects can be created using one of two functions:

```
class_create
class_init
```
1.3 Defining memory partitions

Normally the class\textit{create} version of the call can be used. This will allocate whatever memory is required to store the object, and will return a pointer to the object which can then be used in all subsequent operations on that object.

However, if it is necessary to build a system with no dynamic memory allocation features or to have more control over the memory which is allocated, then the class\textit{init} calls can be used. This leaves memory allocation up to the user, and allowing a completely static system to be created if required. For class\textit{init} calls the user must provide pointers to the data structures, and STLite/OS20 will use these data structures instead of allocating them itself.

When using class\textit{create} calls, the memory for the object structure is normally allocated from the system partition (the one exception to this is that tdesc\_t structures are allocated from the internal partition). Thus the partitions must be initialized before any class\textit{create} calls are made. Normally this is done automatically as described in Chapter 2. Chapter 4 describes the system and internal partitions in more detail.

The number of objects which can be created is only limited to the available memory, there are no fixed size lists within STLite/OS20’s implementation.

When an object is no longer required, it should be deleted by calling the appropriate class\textit{delete} function. If objects are not deleted and memory is reused, then STLite/OS20 and the debugger’s knowledge of valid objects will become corrupted. For example, if an object is defined on the stack and initialized using class\textit{init} then it must be deleted before the function returns and the object goes out of scope.

Using the appropriate class\textit{delete} function will have a number of effects:

- The object is removed from any lists within STLite/OS20, and so will no longer appear in the debugger’s list of known objects.
- The object is marked as deleted so any future attempts to use it will result in an error.
- If the object was created using class\textit{create} then the memory allocated for the object will be freed back to the appropriate partition.

\textbf{Note:} the objects created using both class\textit{create} and class\textit{init} are deleted using class\textit{delete}.

Once an object has been deleted, it cannot continue to be used. Any attempt to use a deleted object will cause a fatal error to be reported. In addition, if a task is blocked on an object (for example it has performed a \texttt{semaphore\_wait()}), and the object is then deleted, the task will be rescheduled, but will immediately raise a fatal error.

### 1.3 Defining memory partitions

Memory blocks are allocated and freed from memory partitions for dynamic memory management. STLite/OS20 supports three different types of memory partition, \textit{heap}, \textit{fixed} and \textit{simple}, as described in Chapter 4. The different styles of memory partition allow trade-offs between execution times and memory utilization.
An important use of memory partitions is for object allocation. When using the \texttt{class\_create\_} versions of the library functions to create objects, STLite/OS20 will allocate memory for the object. In this case STLite/OS20 uses two pre-defined memory partitions (system and internal) for its own memory management. These partitions need to be defined before any of the \texttt{create\_} functions are called. This is normally performed automatically, see Chapter 2.

1.4 Tasks

Tasks are the main elements of the STLite/OS20 multi-tasking facilities. 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.

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 STLite/OS20 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 5.

1.5 Priority

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

In some cases, when there are two or more tasks of the same priority waiting to run, they will each be run for a short period, dividing the use of the CPU between the tasks. This is called timeslicing.

A task's priority is set when the task is created, although it may be changed later. STLite/OS20 provides the user with sixteen levels of priority.

Some members of the ST20 family of micro-cores implement an additional level of priority via hardware processes.

STLite/OS20 supports the following system of priority for tasks running on a ST20-C2 processor:

- Tasks are normally run as low priority processes, and within this low priority rating may be given a further priority level specified by the user. Low priority tasks of equal priority are timesliced to share the processor time. Low priority tasks only run when there are no high priority processes waiting to run.
1.6 Semaphores

- Tasks may be created to run as high priority *processes*, in which case they are never timesliced and will run until they terminate or have to wait for a time or communication before they deschedule themselves. High priority tasks should be kept as short as possible to prevent them from monopolizing system resources. High priority tasks can interrupt low priority tasks that are running.

On an ST20-C1 there is no hardware priority support. STLite/OS20 allows the user to define individual task priorities, and tasks of equal priority will be timesliced. High priority *processes* are not supported on the ST20-C1.

To implement multi-priority scheduling, STLite/OS20 uses a scheduling kernel which needs to be installed and started, before any tasks are created. This is described in Chapter 3. Further details of how priority is implemented is given in section 5.2.

1.6 Semaphores

STLite/OS20 uses semaphores to synchronize multiple tasks. They can be used to ensure mutual exclusion and control access to a shared resource.

Semaphores may also be used for synchronization between interrupt handlers and tasks and to synchronize the activity of low priority tasks with high priority processes.

Semaphores are described in more detail in Chapter 6.

1.7 Message queues

Message queues provide a buffered communication method for tasks and are described in Chapter 7. On the ST20-C2 they should not be used from tasks running as high priority processes and there are some restrictions on their use from interrupt handlers, see Chapter 7.

1.8 Clocks

STLite/OS20 provides a number of 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 8 provides an overview of how time is handled in STLite/OS20. Time-out related functions are described in Chapter 5, Chapter 6 and Chapter 7.

On the ST20-C2 microprocessor, STLite/OS20 makes use of the device’s two clock registers, one high resolution, the other low resolution. The number of clock ticks is device dependent and is documented in the device datasheet.

The ST20-C1 microprocessor does not have its own clock and so a clock peripheral is required when using STLite/OS20. This may be provided on the ST20 device or on an external device. A number of functions are required, one to initialize the clock and the others to provide the interface between the clock and the STLite/OS20 functions, see Chapter 12. STLite/OS20 provides some example sources of such functions which the user can modify for their particular device, see Appendix A for details.
1.9  **Interrupts**
A comprehensive set of interrupt handling functions is provided by STLite/OS20 to enable external events to interrupt the current task and to gain control of the CPU. These functions are described in Chapter 9.

1.10 **Device ID**
Support is provided for obtaining the ID of the current device, see Chapter 10.

1.11 **Cache**
A number of functions are provided to use the cache support provided on ST20 devices, see Chapter 11.

1.12 **Processor specific functions**
The STLite/OS20 API has been designed to be consistent across the full range of ST20 processors. However, some processors have additional features which it may be useful to take advantage of. It should be remembered that using these functions may reduce the portability of any programs to other ST20 processors. See Chapter 12 and Chapter 13.
1.12 Processor specific functions
# Getting Started

This chapter describes how to start using STLite/OS20 and write a simple application. The concepts and terminology used in this chapter are introduced in Chapter 1.

## 2.1 Building for STLite/OS20

Normally using STLite/OS20 can be almost transparent. All that is necessary is to specify to the linker that the STLite/OS20 runtime system is to be used using the `-runtime` option. For example:

```
st20cc -p dxx -runtime os20 app.tco -o system.lku
```

This ensures that by the time the user’s `main` function starts executing:

- the STLite/OS20 scheduler has been initialized and started.
- the interrupt controller has been initialized.
- the system and internal partitions have been initialized.
- thread safe versions of `malloc` and `free` have been set up.
- protection has been installed to ensure that when multiple threads call `debug` functions or `stdio` functions concurrently, all operations are handled correctly.


### 2.1.1 How it works

To initialize STLite/OS20 requires some cooperation between the linker configuration files, and the run time start up code:

- By specifying the `-runtime os20` option to the linker, the configuration file `os20lku.cfg` or `os20rom.cfg` is used instead of the normal C runtime files. This replaces a number of the standard library files with STLite/OS20 specific versions.
- Some modules within the STLite/OS20 libraries contain functions which are executed at start time automatically (through the use of the `#pragma ST_onstartup`).
- A number of symbols are defined by the linker in the STLite/OS20 configuration files, and through the use of the `chip` command. This allows the library code to pick up chip specific definitions, for example, the base address of the interrupt level controller and the amount of available internal memory.
- The heap defined in the configuration files is used for the system partition and so memory for objects defined via `class_create` functions is allocated from this heap area. `malloc` and `free` are redefined to allocate memory from the system partition.
• The internal partition is defined to be whatever memory is left unused in the INTERNAL memory segment.

All the functions which are called at start up time are standard STLite/OS20 functions. So if the start up code is not doing what is required for a particular application, it is simple to replace it with a custom runtime system and pick and choose which libraries to replace from the C or STLite/OS20 runtimes. Appendix C provides details of how the STLite/OS20 kernel may be recompiled or reconfigured to meet specific application needs. Although this should be done with care and may not be suitable for a production system.

Note: that a ST20-C1 timer module is not installed automatically, because this requires knowledge of how any timer peripherals are being used by the application. See Chapter 12 and Appendix A for further details.

2.1.2 Initializing partitions
The two partitions used internally by STLite/OS20, the system and internal partitions, are set up automatically when the st20cc -runtime os20 linker command line option is used. However, this relies on information which the user must provide in the linker configuration file.

The system partition uses the memory which is reserved using the heap command. As malloc and free have been redefined to operate on the system partition, the two statements:

\[
\text{malloc(size)};
\]

and

\[
\text{memory\_allocate(system\_partition, size)};
\]

are now equivalent.

The internal partition is defined to be whatever memory is left unused in the INTERNAL segment. Thus an INTERNAL segment must be defined.

This involves the STLite/OS20 configuration files defining a number of global variables which are read by STLite/OS20 at start up. These are defined using the \text{offsetof}, \text{sizeof} and \text{sizeused} commands in the configuration file to give details of the unused portion of the INTERNAL segment.
2.1.3 Example

The following example shows how to write a simple STLite/OS20 program, in this case a simple terminal emulator, see Figure 2.1. The code is written to run on an STi5500 evaluation board, but can be easily ported to another target. The device datasheet should be referred to for device specific details.

![Figure 2.1 Example program schematic](image)

To keep the example concise, some code which does not demonstrate the use of STLite/OS20 is omitted here. The full source code is provided with the STLite/OS20 examples in the examples/os20/getstart directory.

The software is structured as two tasks, one handling characters passing from the keyboard and out of the serial port, the other handling characters received from the serial port and being displayed on the console. In addition there is an interrupt handler which services interrupts from the serial hardware.

First some constants and global variables need to be defined:

```c
#define CPU_FREQUENCY 40000000
#define BAUD_RATE 9600
#define SERIAL_TASK_STACK_SIZE 1024
#define SERIAL_TASK_PRIORITY 10
#define SERIAL_INT_STACK_SIZE 1024

ring_t serial_rx_ring, serial_tx_ring;
semaphore_t serial_rx_sem;
int serial_mask = ASC_STATUS_RxBUF_FULL;
task_t *serial_tasks[2];
char serial_int_stack[SERIAL_INT_STACK_SIZE];
```

This defines some constants which are needed to initialize the serial port hardware, in particular the CPU frequency, which is needed when programming the serial port hardware’s baud rate generator and may need to be changed when run on another CPU.
2.1 Building for STLite/OS20

It also defines some constants which are needed when setting up the tasks and interrupts, and global variables which are used for communication between the interrupt handler and tasks (the ring buffers and semaphore).

To initialize this system, an initialization function `serial_init` is provided:

```c
void serial_init(int loopback)
{
    #pragma ST_device(asc)
    volatile asc_t* asc = asc1;

    /* Initialise the PIO pins */
    pio1->pio_pc0_rw = PIO1_PC0_DEFAULT;
    pio1->pio_pc1_rw = PIO1_PC1_DEFAULT;
    pio1->pio_pc2_rw = PIO1_PC2_DEFAULT;

    /* Initial the Rx semaphore */
    semaphore_init_fifo(&serial_rx_sem, 0);

    /* Initialise the ring buffers */
    ring_init(&serial_rx_ring);
    ring_init(&serial_tx_ring);

    /* Install the interrupt handler */
    interrupt_install(ASC1_INT_NUMBER, ASC1_INT_LEVEL,
                     serial_int, (void*)asc);
    interrupt_enable(ASC1_INT_LEVEL);

    /* Initialise the serial port hardware */
    asc->asc_baud = CPU_FREQUENCY / (16 * BAUD_RATE);
    asc->asc_control = ASC_CONTROL_DEFAULT |
                      (loopback ? ASC_CONTROL_LOOPBACK : 0);
    asc->asc_intenable = serial_mask;

    /* Create the tasks */
    serial_tasks[0] = task_create(serial_to_tty, (void*)asc,
                                  SERIAL_TASK_STACK_SIZE,
                                  SERIAL_TASK_PRIORITY, "serial0", 0);
    serial_tasks[1] = task_create(tty_to_serial, (void*)asc,
                                  SERIAL_TASK_STACK_SIZE,
                                  SERIAL_TASK_PRIORITY, "serial1", 0);
    if ((serial_tasks[0] == NULL) || (serial_tasks[1] == NULL)) {
        printf("task_create failed\n");
        debugexit(1);
    }
}
```

First the PIO pins need to be set up so that the serial port is connected to the PIO pins (this involves configuring them as ‘alternate mode’ pins, see the device datasheet for details). Next the semaphore used to synchronize the interrupt handler with the receiving task is initialized. Initially this is set to zero to indicate that there are no buffered characters. Each time a character is received, the semaphore will be signalled, in effect keeping a count of the number of buffered characters. This means that the receiving task does not need to check whether the buffer is empty or not when it is run, as long as it waits on the semaphore once per character.
After initializing the ring buffers, the interrupts are initialized. This connects the interrupt handler (`serial_int`) to the interrupt number (`ASC1_INT_NUMBER`). Note that the interrupt level is not configured here. As this may be shared by several interrupt numbers it is good practice to initialize all the levels which are being used in one central location rather than in each module which uses them (see the definition of `main` at the end of this example).

Next the serial port hardware needs to be configured. This sets up the baud rate, enables the port (possibly enabling loopback mode), and enables the interrupts. Initially only receive interrupts are enabled, as there are no characters to transmit yet. However, the handler needs to be notified as soon as a character is received, so receive interrupts are permanently enabled.

Finally the two tasks which will manage the serial communication are created. This will allocate the task’s stacks from the system partition, and start them running immediately.

The next part of the software is the interrupt handler:

```c
void serial_int(void* param) {
    int status;
    #pragma ST_device(asc)
    volatile asc_t* asc = (volatile asc_t*)param;

    while ((status = (asc->asc_status & serial_mask)) != 0) {
        switch(status) {
        case ASC_STATUS_RxBUF_FULL:
            ring_write(&serial_rx_ring, asc->asc_rxbuf);
            semaphore_signal(&serial_rx_sem);
            break;
        case ASC_STATUS_TxBUF_EMPTY:
            asc->asc_txbuf = ring_read(&serial_tx_ring);
            if (ring_empty(&serial_tx_ring)) {
                serial_mask &= ~ASC_STATUS_TxBUF_EMPTY;
                asc->asc_intenable = serial_mask;
            }
            break;
        }
    }
}
```

This is constructed as a `while` loop, so that when the loop exits, there are certain to be no interrupts pending¹. The code needs to be written this way, as the interrupt level is set up to trigger on a rising edge, and so the interrupt must go inactive to guarantee that the next interrupt is seen as a low-to-high transition. An alternative way of constructing this as a high level triggered interrupt is possible, which would cause the interrupt handler to be entered as long as there are pending interrupts.

Inside the loop the code checks for the two cases we are interested in, the receive buffer being full (i.e. containing a character), and the transmit buffer being empty. Note that the status register is masked by the variable `serial_mask`. This ensures that the

¹. The possibility of error interrupts is ignored in this simple example!
code does not check for the transmit buffer being empty when there are no characters to transmit.

The first task takes characters received from the serial port and displays them on the console:

```c
void serial_to_tty(void* param)
{
    char c;
    while (running) {
        semaphore_wait(&serial_rx_sem);
        c = ring_read(&serial_rx_ring);
        debugwrite(1, &c, 1);
    }
}
```

This just waits for the semaphore to be signalled, at which point there must be a character in the ring buffer, so this is read and printed.

The second task is slightly more complex. This takes characters typed on the keyboard and sends them to the serial port:

```c
void tty_to_serial(void* param)
{
    long int c;
    long int flag;
    const clock_t initial_delay = ONE_SECOND / 100;
    clock_t delay = initial_delay;
    #pragma ST_device(asc)
    volatile asc_t* asc = (volatile asc_t*)param;

    while (running) {
        flag = debugpollkey(&c);
        if (flag == 1) {
            interrupt_lock();
            ring_write(&serial_tx_ring, (char)c);
            serial_mask |= ASC_STATUS_TxBUF_EMPTY;
            asc->asc_intenable = serial_mask;
            interrupt_unlock();
        } else {
            task_delay(delay);
            if (delay < (ONE_SECOND / 10)) delay *= 2;
        }
    }
}
```

This code has to poll the keyboard, otherwise while it was waiting for keyboard input it would prevent other tasks doing output. So the code polls the keyboard, and if no character is read, waits for a short while.

If a character is received, then it needs to be written into the transmit ring buffer, and the transmit serial interrupt enabled. This is the only piece of code which needs to be executed with interrupts disabled, as the updating of the ring buffer, `serial_mask` and the serial port’s interrupt enable register needs to be atomic.
Finally a small test harness needs to be provided:

```c
int main(int argc, char* argv[]) {
    int loopback = (argc > 1);
    device_id_t    devid = device_id();

    printf("--- Simple Terminal Emulator ---\n");
    printf("OS/20 version %s\n", kernel_version());
    printf("Device %x (%s)\n", devid.id, device_name(devid));

    /* Initialise the interrupt system for the chip */
    interrupt_init(ASC1_INT_LEVEL, serial_int_stack,
                   sizeof(serial_int_stack),
                   interrupt_trigger_mode_rising,
                   interrupt_flags_low_priority);
    interrupt_enable(INTERRUPT_GLOBAL_ENABLE);

    serial_init(loopback);

    while (1) {
        debugmessage(".");
        task_delay(ONE_SECOND);
    }
}
```

First this dumps some information to the screen about the STLite/OS20 version and which chip it is running on. Next the interrupt system is initialized, setting up the stack and trigger mode for the interrupt which is going to be used, before enabling global interrupts. The test application is then started, and finally the task goes into an infinite loop dumping a character periodically.

The application can be built as follows:

```
st20cc -p dxx example.c -o example.tco -g -c
st20cc -p dxx example.tco -o system.lku -runtime os20 -M system.map
```

The first `st20cc` command compiles the source file into a `.tco`. The second command links the application code with the run time libraries, specifying that an STLite/OS20 runtime is to be used.

It can now be run as normal:

```
st20run -t major2 system.lku -args loopback
```

This uses a target called `major2`, and specifies an argument so that the code can be run in loopback mode.

### 2.2 Starting STLite/OS20 Manually

If the `-runtime` option to `st20link` cannot be used, then it is still possible to use STLite/OS20.

The linker is called with the normal ANSI C runtime libraries and the OS20 libraries are included. This could be achieved, for example, by the following:

```
st20cc -p dxx -T myfile.cfg app.tco -o system.lku
```
Where `myfile.cfg` includes the following command:

```
file os20.lib
```

STLite/OS20 must then be started and initialized by making the relevant calls from the user code. The order in which initialization and object creation can occur is strictly defined:

1. `partition_init_type` can be called to initialize the system and internal partitions. Being able to call this before `kernel_initialize` is a special dispensation for backward compatibility, is not required, and is not encouraged for new programs.

2. `kernel_initialize` should normally be the first STLite/OS20 call made.

3. All class `_init` and `_create` functions can now be called, apart from tasks. This allows objects to be created while guaranteed to still be in single threaded mode.

4. `kernel_start` can now be called to start the multi-tasking kernel. STLite/OS20 is now fully up and running.

5. Tasks can now be created by calling `task_create` or `task_init`, together with any other STLite/OS20 call.

The one exception to this list is the interrupt system. This has been designed so that it can be used even when the remainder of STLite/OS20 is not being used. Thus calls to `interrupt_init_controller`, `interrupt_init`, `interrupt_install` and any other `interrupt_` function can be made at any point. Obviously any interrupt handlers which run before the kernel has started, should not make calls which can cause tasks to be scheduled, for example `semaphore_signal`.

There is one other piece of initialization which must be performed for the ST20-C1. Before any `time` functions are used, a timer module needs to be installed. For an example of how to do this see Chapter 12.
3 Kernel

To implement multi-priority scheduling, STLite/OS20 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. It is the kernel's responsibility to ensure that it is always the task which has the highest scheduling priority that is the one which is currently running.

3.1 Implementation

The kernel maintains two vitally important pieces of information:

1. Which is the currently executing task, and thus what priority is currently being executed.
2. A list of all the tasks which are currently ready to run. This is actually stored as a number of queues, one for each priority, with the tasks stored in the order in which they will be executed.

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

1. 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, then the state of the current task is saved, and the new one installed in its place, so that the new task starts to run. This is termed ‘preemption’, because the new task has preempted the old one.

2. When a task deschedules, for example it waits on a message queue which does not have any messages available, then the scheduler will be invoked to decide which task to run next. The kernel examines the list of processes which are ready to run, and picks the one with the highest priority.

3. Periodically the scheduler is called to timeslice the currently executing task. If there are other tasks which are of the same priority as the current task, then the state of the current task will be saved onto the back of the current priority queue, and the task at the front of the queue installed in its place. In this way all processes at the same priority get a chance to run.

In this way the kernel ensures that it is always the highest priority task which runs.

The scheduler code is installed as a scheduler trap handler, which causes the ST20 hardware to invoke the scheduling software whenever a scheduling operation is required.

3.2 STLite/OS20 kernel

The only operations which can be performed on the STLite/OS20 kernel is its installation and start. This is done by calling the functions kernel_initialize() and kernel_start(). Normally, if the st20cc -runtime os20 option is specified when linking, this is performed automatically. However, if STLite/OS20 is being started
manually the initialization of the STLite/OS20 kernel is usually performed as the first operation in `main()`:

```c
if (kernel_initialize () != 0) {
    printf ("Error : initialise. kernel_initialize failed\n");
    exit (EXIT_FAILURE);
}
... initialize memory and semaphores ...
if (kernel_start () != 0) {
    printf("Error: initialize. kernel_start failed\n");
    exit(EXIT_FAILURE);
}
```

### 3.3 Kernel header file: `kernel.h`

All the definitions related to the kernel are in the single header file, `kernel.h`, see Table 3.1:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/ HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kernel_initialize</code></td>
<td>Initialize for preemptive scheduling.</td>
<td></td>
</tr>
<tr>
<td><code>kernel_start</code></td>
<td>Starts preemptive scheduling regime.</td>
<td></td>
</tr>
<tr>
<td><code>kernel_version</code></td>
<td>Return the STLite/OS20 version number.</td>
<td>ISR and HPP.</td>
</tr>
</tbody>
</table>

**Table 3.1 Functions defined in `kernel.h`**

All functions are callable from an STLite/OS20 task. Functions in Table 3.1 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor.
3.4 Kernel function definitions

**kernel_initialize**

Initialize for preemptive scheduling.

**Synopsis:**

```c
#include <kernel.h>

int kernel_initialize(void);
```

**Arguments:**

None.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

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

**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 process is initialized as the root task in the system.

**Note:** that on the ST20-C2, if this function is called at high priority, it will return executing at low priority. This is a requirement for the correct functioning of the STLite/OS20 kernel.

**Note:** this function may be called automatically if `st20cc -runtime os20` is specified when linking, see Chapter 2. It is important that this function is not called twice.

**See also:**

`kernel_start` `kernel_initialize`
3.4 Kernel function definitions

**kernel_start**

Starts preemptive scheduling regime.

**Synopsis:**

```c
#include <kernel.h>
int kernel_start(void);
```

**Arguments:**

None.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Failure is caused by insufficient space to create the scheduler trap handler.

**Description:**

`kernel_start()` must be called before any tasks are created. It creates and installs the scheduler trap handler and enables the desired scheduler traps. On return from the function the preemptive scheduler is running, and the calling function is installed as the first STLite/OS20 task, and is now running at `MAX_USER_PRIORITY`.

**Note:** this function may be called automatically if `st20cc -runtime os20` is specified when linking, see Chapter 2. It is important that this function is not called twice.

**See also:**

`kernel_initialize` `task_create`
**kernel_version**  
Return the STLite/OS20 version number.

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

**Arguments:**
None.

**Results:**
Returns a pointer to the STLite/OS20 version string.

**Errors:**
None.

**Description:**
`kernel_version()` returns a pointer to a string which gives the STLite/OS20 version number. This string takes the form:

```
{major number}.{release number}.{minor number} [text]
```

that is, a major, release and minor number, separated by decimal points, and optionally followed by a space and a printable text string.

**See also:**
`kernelInitialize`
4 Memory and partitions

Memory management on many embedded systems is vitally important, because available memory is often quite small, and must be used efficiently. For this reason three different styles of memory management have been provided with STLite/OS20, see section 4.2. These give the user flexibility in controlling how memory is allocated, allowing a space/time trade-off to be performed.

4.1 Partitions

The basic 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 STLite/OS20 these concepts have been combined into a partition, which has three properties:

1. The block of memory for which the partition is responsible.
2. The current state of allocated and free memory.
3. 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 thus 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). This can be useful to avoid 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 STLite/OS20 system of partitions can also be exploited to build fault-tolerance into an application, by implementing different parts of the application, using different memory partitions. Then if a fault occurs in one part of the application it does not necessarily effect the whole application.

4.2 Allocation strategies

Three types of partition are currently supported in STLite/OS20:

1. 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 will be returned to the user. Blocks of
4.2 Allocation strategies

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 will be 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 upon 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.

2 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_init_fixed`. This means that allocating and freeing a block takes constant time (i.e. is deterministic), and there is a very small memory overhead. Thus this partition ignores the size argument when an allocation is performed by `memory_allocate` and uses instead the size argument passed by either `partition_create_fixed` or `partition_init_fixed`.

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

3 Finally 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. Thus 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 4.1.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Heap</th>
<th>Fixed</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocation method</td>
<td>As requested by <code>memory_allocate</code> or <code>memory_reallocate</code></td>
<td>Fixed at creation by <code>partition_create_fixed</code> or <code>partition_init_fixed</code>.</td>
<td>As requested by <code>memory_allocate</code> or <code>memory_reallocate</code></td>
</tr>
<tr>
<td>Deallocation possible</td>
<td>Yes.</td>
<td>Yes.</td>
<td>No.</td>
</tr>
<tr>
<td>Overhead size (bytes)</td>
<td>12</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Deterministic</td>
<td>No.</td>
<td>Yes.</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

Table 4.1 Partition properties
4.3 Pre-defined partitions

STLite/OS20 has been designed not to require any dynamic memory allocation itself. This allows the construction of deterministic systems, or for the user to take over all memory allocation.

However, for convenience, all of the object initialization functions (e.g. task_init, semaphore_init_fifo etc.) are also available with a creation style of interface (e.g. task_create, semaphore_create_fifo), where STLite/OS20 will perform the memory allocation for the object. In these cases STLite/OS20 will use two pre-defined partitions:

- The system_partition is used for all object allocation, including semaphores, message queues and the static portion of the task’s data structure, including the task’s stack. Normally this is managed as a heap partition.

- The internal_partition is used just for the allocation of the dynamic part of a task’s data structure by task_create. To minimize context switch time, this data should be placed in internal memory (see section 5.1 for more information about a task’s state). Thus the internal_partition should manage a block of memory from the ST20’s internal memory. Normally this is managed as a simple partition, to minimize wastage of internal memory.

These partitions must be defined before any of the object creation functions are called, and because they are independent of the kernel this can be done before kernel initialization if required.

Normally, if the st20cc -runtime os20 option is specified when linking, this initialization is performed automatically, see Chapter 2.

If STLite/OS20 is being started manually, the following can be done:

```c
partition_t *system_partition;
partition_t *internal_partition;
static int internal_block[200];
static int external_block[100000];
#pragma ST_section(internal_block, "internal_part")

void initialize_partitions(void)
{
    static partition_t the_system_partition;
    static partition_t the_internal_partition;

    if (partition_init_simple(&the_internal_partition,
               (unsigned char*)internal_block, sizeof(internal_block) !=0){
        printf("partition creation failed \n");
        return;
    }
    if (partition_init_heap(&the_system_partition,
               (unsigned char*)external_block, sizeof(external_block) !=0){
        printf("partition creation failed \n");
        return;
    }

    system_partition = &the_system_partition;
    internal_partition = &the_internal_partition;
}
```
4.4 Obtaining information about partitions

The section `internal_part` is then placed into internal memory by adding a line to the application configuration file:

```
place internal_part INTERNAL
```

### 4.3.1 Calculating partition sizes

In order to calculate the size of system and internal partitions, several pieces of information are needed for each object created using the `_create` functions (e.g. `task_create`, `message_create_queue`, `semaphore_create_fifo` etc.):

- The amount of memory the object requires. Each object is defined by a data structure or type e.g. `task_t`, (refer to individual chapters e.g. ‘Chapter 5 Tasks’). Table 4.2 lists the different types of object structure that may be created and their memory requirement.

<table>
<thead>
<tr>
<th>Object structure</th>
<th>Size (words)</th>
<th>Size (Bytes)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan_t</td>
<td>4</td>
<td>16</td>
<td>ST20-C2 specific.</td>
</tr>
<tr>
<td>semaphore_t</td>
<td>6</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>message_queue_t</td>
<td>19</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>partition_t</td>
<td>15</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>task_t</td>
<td>9</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>tdesc_t</td>
<td>6, 9</td>
<td>24, 36</td>
<td>ST20-C2 specific, ST20-C1 specific</td>
</tr>
</tbody>
</table>

*Table 4.2 Object size requirement*

- The amount of memory needed for tasks’ stacks, created using `task_create`.
- The amount of overhead the memory allocation function requires for the object. The number of words used depend on whether the object is allocated from a `heap`, `fixed` or `simple` partition. All objects are allocated from the system partition which is managed as a heap partition. The exception is the object structure `tdesc_t` which is allocated from the internal partition (normally managed as a simple partition). Table 4.1 shows the memory overhead associated for each partition type.
- Any additional allocations performed by the user’s application.

### 4.4 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 <partitio.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()` will return 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 will always be the same as the block size of a given *fixed* partition.
4.5 Partition header file: partitio.h

All the definitions related to memory partitions are in the single header file, partitio.h, see Table 4.3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory_allocate</td>
<td>Allocate a block of memory from a partition.</td>
<td>HPP</td>
</tr>
<tr>
<td>memory_allocate_clear</td>
<td>Allocate a block of memory from a partition and clear to zero.</td>
<td>HPP</td>
</tr>
<tr>
<td>memory_deallocate</td>
<td>Free a block of memory back to a partition.</td>
<td>HPP</td>
</tr>
<tr>
<td>memory_reallocate</td>
<td>Reallocation a block of memory from a partition.</td>
<td>HPP</td>
</tr>
<tr>
<td>partition_create_simple</td>
<td>Create a simple partition.</td>
<td></td>
</tr>
<tr>
<td>partition_create_heap</td>
<td>Create a heap partition.</td>
<td></td>
</tr>
<tr>
<td>partition_create_fixed</td>
<td>Create a fixed partition.</td>
<td></td>
</tr>
<tr>
<td>partition_delete</td>
<td>Delete a partition.</td>
<td></td>
</tr>
<tr>
<td>partition_init_simple</td>
<td>Initialize a simple partition.</td>
<td></td>
</tr>
<tr>
<td>partition_init_heap</td>
<td>Initialize a heap partition.</td>
<td></td>
</tr>
<tr>
<td>partition_init_fixed</td>
<td>Initialize a fixed partition.</td>
<td></td>
</tr>
<tr>
<td>partition_status</td>
<td>Get the status of a partition.</td>
<td>HPP</td>
</tr>
</tbody>
</table>

Table 4.3 Functions defined in partitio.h

All functions are callable from an STLite/OS20 task. Functions in Table 4.3 which are marked with 'ISR' can also be called from an interrupt service routine, those marked with 'HPP' can be called from a high priority process on an ST20-C2 processor.

Table 4.4 lists the types defined by partitio.h.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition_t</td>
<td>A memory partition.</td>
</tr>
<tr>
<td>partition_status_flags_t</td>
<td>Additional flags for partition_status.</td>
</tr>
</tbody>
</table>

Table 4.4 Types defined by partitio.h
4.6 Memory and partition function definitions

memory_allocate Allocate a block of memory from a partition.

Synopsis:

```c
#include <partitio.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.

Results:

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

Errors:

If there is insufficient memory for the allocation, it will fail and return `NULL`.

Description:

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

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

See also:

- `memory_deallocate`
- `memory_reallocate`
- `partition_create_fixed`
- `partition_create_heap`
- `partition_create_simple`
4.6 Memory and partition function definitions

**memory_allocate_clear**

Allocate and zero a block of memory from a partition.

**Synopsis:**

```
#include <partitio.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.

**Results:**

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

**Errors:**

If there is insufficient memory for the allocation, it will fail and return `NULL`.

**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 will return the base address of the array, which is suitably aligned to contain any type. The memory is initialized to zero.

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

**See also:**

- `memory_allocate`
- `memory_deallocate`
- `memory_reallocate`
- `partition_create_fixed`
- `partition_create_heap`
- `partition_create_simple`
**memory_deallocate**

Free a block of memory back to a partition.

**Synopsis:**

```
#include <partitio.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.

**Results:**

None.

**Errors:**

None.

**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.

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

**See also:**

- `memory_allocate`
- `memory_reallocate`
- `partition_create_fixed`
- `partition_create_heap`
- `partition_create_simple`
- `partition_delete`
memory_reallocate  Reallocate a block of memory from a partition.

Synopsis:

```c
#include <partitio.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.

Results:

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

Errors:

If there is insufficient memory for the allocation, it will fail and return `NULL`.

Description:

`memory_reallocate()` changes the size of a memory block allocated from a partition, preserving the current contents. This function may only be used for heap or simple partitions, it has no effect for fixed partitions. It will try and do the reallocation efficiently, changing the size of the existing block, and returning a pointer to the original block. However if this is not possible, then a new block will be allocated of the requested size (which is suitably aligned to contain any type), the data copied, the original block freed, and a pointer to the new block returned.

`block` must have been allocated from `part` originally.

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

See also:

- `memory_allocate`
- `memory_deallocate`
- `partition_create_fixed`
- `partition_create_heap`
- `partition_create_simple`
**partition_create_fixed**  
Create a fixed size partition.

**Synopsis:**

```c
#include <partitio.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.

**Results:**

The partition identifier or NULL if an error occurs.

**Errors:**

If the amount of memory is insufficient it will fail and return NULL.

**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 will be allocated, with no overhead for the partition manager. Allocating and freeing simply involves removing and adding blocks to a linked list, so is constant time.

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

**See also:**

- `memory_allocate`
- `memory_deallocate`
- `partition_create_heap`
- `partition_create_simple`
### partition_create_heap

Create a heap partition.

#### Synopsis:

```c
#include <partition.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.

#### Results:

- The partition identifier or NULL if an error occurs.

#### Errors:

- If the amount of memory is insufficient it will fail and return NULL.

#### 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 will be 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 will only result in a copy if the block cannot be expanded because subsequent memory locations have been allocated.

#### See also:

- `memory_allocate`
- `memory_deallocate`
- `partition_create_fixed`
- `partition_create_simple`
**partition_create_simple**  
Create a simple partition.

**Synopsis:**
```c
#include <partitio.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.

**Results:**
- The partition identifier or NULL if an error occurs.

**Errors:**
- If the amount of memory is insufficient it will fail and return NULL.

**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 will be allocated, with no overhead. Allocation simply 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 will always return NULL.

**See also:**
- `memory_allocate`  
- `memory_deallocate`  
- `partition_create_fixed`  
- `partition_create_heap`
**partition_delete**

Delete a partition.

**Synopsis:**

```c
#include <partition.h>
void partition_delete( partition_t *Partition )
```

**Arguments:**

- `partition_t *Partition`: Partition to delete.

**Results:**

None.

**Errors:**

None.

**Description:**

This function allows a partition to be deleted. If the partition was created using a `_create` function e.g. `partition_create_heap` then this function will free the data structure used to manage the partition (`partition_t`). If the partition was created using an `_init` function i.e. `partition_init_heap` then the user is responsible for freeing the partition data structure.

The deletion of 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 the example below.

**Example:**

```c
partition_t *part;
char *ptr;

ptr = memory_allocate(system_partition, size);
part = partition_create_fixed(ptr, size);

...memory_allocate(part)
   memory_deallocate(part)
...memory_allocate(part)

partition_delete(part);
memory_deallocate(system_partition, ptr);
```

**See also:**

- `partition_create_simple`
- `partition_init_simple`
- `partition_create_heap`
- `partition_init_heap`
- `partition_create_fixed`
- `partition_init_fixed`
partition_init_fixed

Initialize a fixed size partition.

Synopsis:

```c
#include <partitio.h>

int partition_init_fixed(
  partition_t* partition,
  void* memory,
  size_t memory_size,
  size_t block_size);
```

Arguments:

- `partition_t* partition`: Pointer to the partition to initialize.
- `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.

Results:

Returns 0 on success or -1 on error.

Errors:

If the amount of memory is insufficient it will fail and return -1.

Description:

`partition_init_fixed()` initializes 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 will be allocated, with no overhead for the partition manager. Allocating and freeing simply involves removing and adding blocks to a linked list, so is constant time.

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

`partition_t` should be declared before the call to `partition_init_fixed` is made.

See also:

- `memory_allocate`
- `memory_deallocate`
- `partition_init_heap`
- `partition_init_simple`
4.6 Memory and partition function definitions

partition_init_heap

Initialize a heap partition.

Synopsis:

```c
#include <partition.h>

int partition_init_heap(
    partition_t* partition,
    void* memory,
    size_t size);
```

Arguments:

- `partition_t* partition`: Pointer to the partition to initialize.
- `void* memory`: The start address for the memory partition.
- `size_t size`: The size of the memory block in bytes.

Results:

Returns 0 on success or -1 on error.

Errors:

If the amount of memory is insufficient it will fail and return -1.

Description:

`partition_init_heap()` initializes 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 will be 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 will only result in a copy if the block cannot be expanded because subsequent memory locations have been allocated.

`partition_t` should be declared before the call to `partition_init_heap` is made.

See also:

- `memory_allocate`
- `memory_deallocate`
- `partition_init_fixed`
- `partition_init_simple`
**partition_init_simple**

Initialize a simple partition.

**Synopsis:**

```c
#include <partitio.h>

int partition_init_simple(
    partition_t* partition,
    void* memory,
    size_t size);
```

**Arguments:**

- `partition_t* partition`  
  Pointer to the partition to initialize.
- `void* memory`  
  The start address for the memory partition.
- `size_t size`  
  The size of the memory block in bytes.

**Results:**

Returns 0 on success or -1 on error.

**Errors:**

If the amount of memory is insufficient it will fail and return -1.

**Description:**

`partition_init_simple()` initializes 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 will be allocated, with no overhead. Allocation simply 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 will always return `NULL`.

`partition_t` should be declared before the call to `partition_init_simple` is made.

**See also:**

- `memory_allocate`
- `memory_deallocate`
- `partition_init_fixed`
- `partition_init_heap`
### partition_status

Get status of a partition.

**Synopsis:**

```c
#include <partitio.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.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if `Partition` or `Status` is NULL, or if `partition` has not been initialized using one of the `_create` or `_init` functions. Partitions previously deleted with `partition_delete()` will also return -1.

**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()` will use to store the results.

Table 4.5 shows the layout of the structure `partition_status_t`.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition_status_state</td>
<td>Partition state (See Table 4.6).</td>
</tr>
<tr>
<td>partition_status_type</td>
<td>Type of partition (See Table 4.7).</td>
</tr>
<tr>
<td>partition_status_size</td>
<td>Total number of bytes within partition.</td>
</tr>
<tr>
<td>partition_status_free</td>
<td>Total number of bytes free within partition.</td>
</tr>
<tr>
<td>partition_status_free_largest</td>
<td>Total number of bytes within the largest free block in partition.</td>
</tr>
<tr>
<td>partition_status_used</td>
<td>Total number of bytes which are allocated/in use within the partition.</td>
</tr>
</tbody>
</table>

Table 4.5  Layout of structure `partition_status_t`
Table 4.6 show all the possible values which are available to the field partition_status_state.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Flag description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition_status_state_valid</td>
<td>Partition is valid.</td>
</tr>
<tr>
<td>partition_status_state_invalid</td>
<td>Partition is corrupt.</td>
</tr>
</tbody>
</table>

Table 4.6 Flag values for partition_status_state.

Table 4.7 shows all the possible values which are available to the field partition_status_type.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Flag description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partition_status_state_type_simple</td>
<td>Partition is a Simple partition.</td>
</tr>
<tr>
<td>partition_status_state_type_fixed</td>
<td>Partition is a Fixed partition.</td>
</tr>
<tr>
<td>partition_status_state_type_heap</td>
<td>Partition is a Heap partition.</td>
</tr>
</tbody>
</table>

Table 4.7 Flag values for partition_status_type.

If partition_status() returns successfully then the structure pointed to by Status will contain 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 will contain one of the flags as shown in Table 4.7.

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/_init functions, therefore partition_status_size will not change with subsequent calls to partition_status().

partition_status_used is the total number of bytes which have been 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 which have been 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/de-allocation, these overheads are taken into account in the results. (See Table 4.1).
4.6 Memory and partition function definitions

Example:

```c
#include <partitio.h>

unsigned char buffer[BUFFER_SIZE];
partition_status_t status;
partition_t partition;

partition_init_heap(&partition, &buffer, BUFFER_SIZE);

if (partition_status(&partition, &status) == 0) {
    ... process status of partition
} else {
    ... process partition error
}
```

See also:

- partition_create_simple
- partition_init_simple
- partition_create_fixed
- partition_init_fixed
- partition_create_heap
- partition_init_heap
5 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.

5.1 STLite/OS20 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 is processor dependent. In STLite/OS20 it is divided into two parts and includes the following elements:

- **dynamic state** defined in the data structure `tdesc_t`, which is used directly by the CPU to execute the process. The fields of this structure vary depending on the processor type. The most important elements of this structure are the machine registers, in particular the instruction (Iptr) and workspace (Wptr) pointers. A task priority is also used to make scheduling decisions. While the task is running the Iptr and Wptr are maintained by the CPU, when the task is not executing they are stored in tdesc_t. On the ST20-C1 the Tdesc register points to the current task's tdesc_t.

- **static state** defined in the data structure `task_t`, which is used by STLite/OS20 to describe the task, and which does not usually change while the task is running. It includes the state of the task e.g. being created, executing, terminated and the stack range which is used for stack checking.

The dynamic state should be stored in internal memory to minimize context switch time. The state is divided into two in this way so that only the minimum amount of internal memory needs to be used to store tdesc_t.

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 5.12.

The task's data structure may either be allocated by STLite/OS20 or by the user declaring the `tdesc_t` and `task_t` data structures. (These structures are defined in the header file `task.h`). The code for the task to execute is provided by the user function. To create a task, the `tdesc_t` and `task_t` data structures must be allocated and initialized and a stack and function must be associated with them. This is done using the `task_create` or `task_init` functions depending on whether the user wishes to control the allocation of the data structures or not. See section 5.5.
5.2 Implementation of priority and timeslicing

Readers familiar with the ST20 micro-core and STLite/OS20 priority handling may wish to skip to section 5.3 which introduces the facilities provided by STLite/OS20 for influencing priority.

STLite/OS20 implements 16 levels of priority. Tasks are run as the lowest priority hardware process for the target hardware with a STLite/OS20 priority specified by the user. STLite/OS20 tasks sit on top of the processes implemented by the hardware and use features of the hardware to ensure efficient implementation.

On the ST20-C1, there is no hardware support for multiple priorities.

However on the ST20-C2, the hardware supports two priorities of processes, high and low, see Figure 5.1.

High priority processes take precedence over low priority processes, e.g. STLite/OS20 tasks. Thus on the ST20-C2, for critical sections of code it is possible to create tasks which use the hardware’s high priority processes directly.

ST20-C2 high priority processes run outside of the STLite/OS20 scheduler, and so some restrictions have to be placed on them:

- They cannot use priority based semaphores
- They cannot use message queues

In addition they inherit two features of the hardware scheduler:

- Tasks are not timesliced, they execute until they voluntarily deschedule
- The units of time are different with high priority processes running considerably faster than low priority processes. The clock times are device dependent so check the datasheet for actual timings.

Figure 5.1 ST20-C2 priorities
5.2.1 Timeslicing on the ST20-C1

On the ST20-C1 microprocessor timeslicing is supported by a *timeslice* instruction. By default timeslicing is disabled by the compiler. However, if the application is compiled with the `st20cc` option `-finl-timeslice` then *timeslice* instructions will be inserted by the compiler. Note, that the runtime libraries are compiled without timeslicing, so it is not possible to timeslice in a library function.

If a *timeslice* instruction is executed when a timeslice is due and timeslicing is enabled then the current *process* will be timesliced, i.e. the current *process* is placed on the back of the scheduling queue and the *process* on the front of the scheduling queue is loaded into the CPU for execution.

Note timeslicing is implemented independent of the clocking peripheral discussed in Chapter 12.

Further details are given in the ‘*ST20-C1 Core Instruction Set Reference Manual 72-TRN-274*’.

5.2.2 Timeslicing on the ST20-C2

The ST20-C2 microprocessor contains two clock registers. The high priority clock register and the low priority clock, see Chapter 8.

After a set number of ticks of the high priority clock a timeslice period is said to have ended. When two timeslice period ends have occurred while the same task (i.e. low priority hardware process) has been continuously executing, the processor will attempt to deschedule the task. This will occur after the next *j* or *lend* instruction executed. When this happens the task is descheduled and the next waiting task is scheduled, see Figure 5.2.

High priority processes are never timesliced and will run until completion, or until they have to wait for a communication.

![Figure 5.2 Timeslicing on the ST20-C2](image-url)
5.3 STLite/OS20 priorities

A task will nominally run for between one and two timeslice periods. The compiler inserts instructions which allow timeslicing (for example \( j \)) at suitable points in the code, in order to minimize latency and prevent tasks monopolizing processor time. If an STLite/OS20 task is preempted by a higher priority STLite/OS20 task then when the lower priority tasks resumes it will start its timeslice period from the beginning of the timeslice period. However, if an STLite/OS20 task is interrupted by an interrupt or preempted by a high priority process then it will resume the timeslice period from the point where the interrupt or high priority process released the period. Therefore the STLite/OS20 task will loose some of its timeslice.


5.3 STLite/OS20 priorities

The number of STLite/OS20 task priorities and the highest and lowest task priorities are defined using the macros in the header file `task.h`, see section 5.18. Numerically higher priorities preempt lower priorities e.g. 3 is a higher priority than 2.

A task’s initial priority is defined when it is created, see section 5.5. The only task which does not have its priority defined in this way is the root task, that is, the task which starts STLite/OS20 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:

```c
int task_priority (task_t* Task)
```

task_priority() retrieves the STLite/OS20 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:

```c
int task_priority_set (task_t* Task, int NewPriority);
```

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, then tasks may be rescheduled. This function is only applicable to STLite/OS20 tasks not to high priority hardware processes.

5.4 Scheduling

An active task may either be running or waiting to run. STLite/OS20 ensures that:

- The currently executing task is always the one with the highest priority.

If a task with a higher priority becomes ready to run then the STLite/OS20 scheduler will save the current task’s state and will make the higher priority task the current task. The current task will run to completion unless it is
preempted by a higher priority task, and so on. Once a task has completed, the next highest priority task will start executing.

- Tasks of equal priority are timesliced, to ensure that they all get the chance to run. (When compiling for an ST20-C1 a command line option needs to be given, see section 5.2.1).

Each task of the same priority level will execute in turn for a period of time known as a timeslice. See section 5.2.

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

```c
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 will not be released until an equal number of calls to `task_unlock()` have been made. Once `task_unlock()` is called, the scheduler will start the highest priority task which is available, running. This may not be the task which calls `task_unlock()`.

If a task voluntarily deschedules, e.g. by calling `semaphore_wait` then the critical region will be unlocked and normal scheduling resumes. In this case the subsequent `task_unlock` has no effect. It should still be included in case the task did not deschedule e.g. the semaphore count was already greater than zero.

**Note** that when this lock is in place the task can still be interrupted by interrupt handlers and high priority processes (on the ST20-C2). Interrupts can be disabled and enabled using the `interrupt_lock()` and `interrupt_unlock()` functions, see Chapter 9.

### 5.5 Creating and running a task

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

```c
#include <task.h>

task_t* task_create (int task_init(
    void (*Function)(void*),
    void* Param,
    int StackSize,
    int Priority,
    const char* Name,
    task_flags_t flags);

#include <task.h>

int task_init(
    void (*Function)(void*),
    void* Param,
    void* Stack,
    int StackSize,
    task_t* Task,
    tdesc_t* Tdesc,
    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 structures `tdesc_t` and `task_t` and associating a function with them.

Using either `task_create` or `task_init`, 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 in order to pass in a single word sized parameter (e.g. an `int`). Otherwise a data structure should be set up.

The functions differ in how the task’s data structure is allocated. `task_create` will allocate memory for the task’s stack, control block `task_t` and task descriptor `tdesc_t`, whereas `task_init` enables the user to control memory allocation. The task’s control block and task descriptor should be declared before the call to `task_init`.

`task_create` and `task_init` both require the stack size to be specified. Stack is used for a function’s local variables and parameters, as a guide each function uses:

- Four words for the task to remove itself if it returns.
- Four extra words for the initial user stack.
- On the ST20-C2 six words are needed by the hardware scheduler (for state which is saved into ‘negative workspace’).
- In some cases the full CPU context needs to be saved on the task’s stack. On the ST20-C1 this is always needed when a task is preempted (7 words). On the ST20-C2 it is only needed if a task’s priority is changed by another task, or it is suspended (11 words).
- Then recursively:
  - Space for local variables declared in the function, (add up the number of words).
  - Space for calls to extra functions. For a library function allow 150 words for worst case.

For details of data representation, see the ‘ST20 Embedded Toolset User Manual - 72-TDS-505’.

Both functions require an STLite/OS20 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 `task.h` by the macros `OS20_PRIORITY_LEVEL`, `MAX_USER_PRIORITY` and `MIN_USER_PRIORITY`, see section 5.18.

For tasks running on an ST20-C2, both functions also enable the task to be elevated to a high priority process. In this case the STLite/OS20 task priority will not be used. High priority processes have restrictions associated with them as described in section 5.2.
5 Tasks

5.5.1 Creating a task for an RCU

Two alternative functions are provided for creating a task in a relocatable code unit (RCU): task_create_sl and task_init_sl. These functions are very similar to task_create and task_init but allow a static link to be specified.

A pointer to the static area is normally passed as an extra parameter to every function. This parameter is called the static link and contains the address of the static area for the program. See the ‘Implementation details’ chapter and the section on ‘Relocatable code’ in the ‘st20cc compile/link tool’ chapter of the ‘ST20 Embedded Toolset User Manual’ - 72-TDS-505.

Appendix B of this manual provides some essential guidelines for using STLite/OS20 with RCUs.

5.6 Synchronizing tasks

Tasks synchronize their actions with each other using semaphores, as described in Chapter 6.

5.7 Communicating between tasks

Tasks communicate with each other by using message queues, as described in Chapter 7.

5.8 Timed delays

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

```c
void task_delay(clock_t delay);
void task_delay_until(clock_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, then the task does not wait.

task_delay waits until the given time has elapsed, i.e. 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 will preempt any lower priority task that is running and perform 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:
5.9 Rescheduling

#include <ostime.h>

clock_t time;
time = time_now();
for (;;)
{
    time = time_plus (time, delay);
task_delay_until(time);
    initiate_regular_event ();
}

5.9 Rescheduling

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

void task_reschedule (void);

This provides 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 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.

A task may be inadvertently rescheduled when the task_priority_set () function is used, see section 5.3.

5.10 Suspending tasks

Normally a task will only deschedule when it is waiting for an event, such as for a semaphore to be signalled. This requires that the task itself call 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 will stop executing immediately. When the task should start executing again, another task must resume it. When it is resumed the task will be unaware it has been suspended, other than the time delay.

Task suspension is in addition to any other reason that a task is descheduled. Thus a task which is waiting on a semaphore, and which is then suspended, will 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 0 if the task was successfully suspended, −1 if it failed.
This call will fail if the task has terminated. A task may be suspended multiple times by executing several calls to `task_suspend`. It will not start executing again until an equal number of `task_resume` calls have been made.

A task is resumed using the call:

```c
int task_resume(task_t* Task)
```

where `Task` is the task to be resumed. The result is 0 if the task was successfully resumed, -1 if it failed. The call will fail 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_init`. 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 will start executing from its entry point.

### 5.11 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:

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

This will stop the task immediately, cause 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:

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

and once it is willing to be killed again calling:

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

While the task is immortal, it cannot be killed. However, if an attempt was made to kill the task whilst it was immortal, it will die immediately it makes itself mortal again by calling `task_mortal`.

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

### 5.12 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:

```c
task_t* task_id (void)
```
5.13 Stack usage

While task_id is very efficient when called from a task, it will take a long time to execute when called from a high priority process, and cannot be called from an interrupt handler. To avoid these problems an alternative function is available:

\[
\text{task_context}_t \text{ task_context}(\text{task}_t** \text{ task}, \text{int}* \text{ level})
\]

This will return whether it was called from a task, interrupt, or high priority process. In addition if task is not NULL, and task_context is called from a task or high priority process, it will assign the current task ID to the task_t pointed to by task. Similarly if level is not NULL, and task_context is called from an interrupt handler, then it will assign the current interrupt level to the int pointed to by level. The advantage in not requiring the current task_t or interrupt level is that this function may operate considerably faster when this information does not have to be found.

Both of these function may be used in conjunction with task_wait, see section 5.16.

The function:

\[
\text{const char*task_name(task_t *task)};
\]

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).

### 5.13 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. STLite/OS20 provides a couple of techniques which can be used to address this.

The first technique is to enable stack checking in the compiler (see the section on ‘Runtime checking options’ in the ‘st20cc compile/link tool’ chapter of the ‘ST20 Embedded Toolset User Manual’ - 72-TDS-505 for more details). 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.

As STLite/OS20 is multi-threaded, a special version of the stack check function needs to be used, which can determine the current task, and details about the task’s stack. When using -runtime os20 to link the application, the stack check function is linked in automatically. Otherwise it is necessary to link with the configuration file os20scc1.cfg (for a C1 target) or os20scc2.cfg (for a C2 target) to ensure the correct function is linked in.

Whilst stack checking has the advantage that a stack overflow is reported immediately it occurs, it has a number of 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 STLite/OS20 normally fills a task's stack with a
known value. As the task runs it will write 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 STLite/OS20 provides the function:

```c
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 stack, in particular the base and size specified when the task was created, and the amount of stack which has been used.

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

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

returns details about the current stack fill settings, and:

```c
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 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 STLite/OS20 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.
5.14 Task data

STLite/OS20 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:

```c
void* task_data_set (task_t* Task, void* NewData);
```

task_data_set() sets the task-data pointer of the task specified by Task.

```c
void* task_data(task_t* Task);
```

task_data() 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:

```c
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(system_partition, sizeof(ptd_t));
    ptd->buffer_next = ptd->buffer_end = ptd->buffer;
    task_data_set(task, ptd);
}
```
5.15 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 both cases an exit status can be specified. When the task returns from its entry point function, the exit status is the value that the function returns. If task_exit is called then the exit status is specified as the parameter. This value is then made available to the ‘onexit’ handler if one has been installed (see below).

Just before the task terminates (either by returning from its entry point function, or calling task_exit), it will call an ‘onexit’ handler. This function allows any application specific tidying up to be performed before the task terminates. The onexit handler is installed by calling:

    task_onexit_fn_t task_onexit_set(task_onexit_fn_t fn);

The onexit handler function must have a prototype of:

    void onexit_handler(task_t *task, int param)

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

The function task_onexit_set_sl is provided to set the task onexit handler and specify a static link.

The following code example shows how a task’s exit code can be stored in its task-data (see section 5.14), and retrieved later by another task which is notified of the termination through task_wait.

    void onexit_handler(task_t* task, int param)
    { 
      task_data_set(NULL, (void*)param);
    }

    int main()
    { 
      task_t *Tasks[NO_USER_TASKS];

      /* Set up the onexit handler */
      task_onexit_set(onexit_handler);

      ... create the tasks ...

      /* Wait for the tasks to finish */
      for (i=0; i<NO_USER_TASKS; i++) { 
        int t;
        t = task_wait(Tasks, NO_USER_TASKS, TIMEOUT_INFINITY);
        printf("Task %d : exit code %d\n", t, (int)task_data(Tasks[t]));
        Tasks[t] = NULL;
      }
    }
5.16 Waiting for termination

It is only safe to free or otherwise reuse a task’s stack, once it has terminated. The following function waits until one of a list of tasks terminates or the specified timeout period is reached:

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

Timeouts for tasks are implemented using hardware and so 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 i.e. its control block `task_t` and descriptor `tdesc_t` may also be freed. (`task_delete` can be used to free `task_t` and `tdesc_t`, see section 5.17).

The timeout period for `task_wait` may be expressed as a 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 `ostime.h` must be included when using this function.

5.17 Deleting a task

A task can be deleted by using the `task_delete` function:

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

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

If the task was created using `task_create` then `task_delete` calls `memory_deAllocate` in order to free the task’s state (both static `task_t` and dynamic `tdesc_t`) and the task’s stack.

A task must have terminated before it can be deleted, if it has not `task_delete` will fail.
### 5.18 Task header file: `task.h`

All the definitions related to interrupts are in the single header file, `task.h`, see Table 5.1, Table 5.2 and Table 5.3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_context</td>
<td>Return the current execution context.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>task_create</td>
<td>Create an STLite/OS20 task.</td>
<td></td>
</tr>
<tr>
<td>task_create_sl</td>
<td>Create an STLite/OS20 task specifying a static link.</td>
<td></td>
</tr>
<tr>
<td>task_data</td>
<td>Retrieve a task's data pointer.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_data_set</td>
<td>Sets a task's data pointer.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_delay</td>
<td>Delay the calling task for a period of time.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_delay_until</td>
<td>Delay the calling task until a specified time.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_delete</td>
<td>Delete a task.</td>
<td>*</td>
</tr>
<tr>
<td>task_exit</td>
<td>Exits the current task.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_id</td>
<td>Find current task's id.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_immortal</td>
<td>Make the current task immortal.</td>
<td></td>
</tr>
<tr>
<td>task_init</td>
<td>Initialize an STLite/OS20 task.</td>
<td></td>
</tr>
<tr>
<td>task_init_sl</td>
<td>Initialize an STLite/OS20 task specifying a static link</td>
<td></td>
</tr>
<tr>
<td>task_kill</td>
<td>Kill a task.</td>
<td></td>
</tr>
<tr>
<td>task_lock</td>
<td>Prevent task rescheduling.</td>
<td></td>
</tr>
<tr>
<td>task_mortal</td>
<td>Make the current task mortal.</td>
<td></td>
</tr>
<tr>
<td>task_name</td>
<td>Returns the task's name.</td>
<td>HPP*</td>
</tr>
<tr>
<td>task_onexit_set</td>
<td>Setup a function to be called when a task exits.</td>
<td></td>
</tr>
<tr>
<td>task_onexit_set_sl</td>
<td>Setup a function to be called when a task exits and specify a static link.</td>
<td></td>
</tr>
<tr>
<td>task_priority</td>
<td>Retrieve a task's priority.</td>
<td>HPP*</td>
</tr>
<tr>
<td>task_priority_set</td>
<td>Set a task's priority.</td>
<td></td>
</tr>
<tr>
<td>task_reschedule</td>
<td>Reschedule the current task.</td>
<td>HPP</td>
</tr>
<tr>
<td>task_resume</td>
<td>Resume a suspended task.</td>
<td></td>
</tr>
<tr>
<td>task_suspend</td>
<td>Suspend a task.</td>
<td></td>
</tr>
<tr>
<td>task_stack_fill</td>
<td>Return the task fill configuration.</td>
<td></td>
</tr>
<tr>
<td>task_stack_fill_set</td>
<td>Set the task stack fill configuration.</td>
<td></td>
</tr>
<tr>
<td>task_status</td>
<td>Return status information about the task.</td>
<td></td>
</tr>
<tr>
<td>task_unlock</td>
<td>Allow task rescheduling.</td>
<td></td>
</tr>
<tr>
<td>task_wait</td>
<td>Waits until one of a list of tasks completes.</td>
<td>*</td>
</tr>
</tbody>
</table>

Table 5.1 Functions defined in `task.h`
All functions are callable from an STLite/OS20 task. Functions in Table 5.1 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor. Note: only those functions marked with an ‘*’ can take a task_t argument that refers to a high priority process.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task_context_t</td>
<td>Result of task_context.</td>
</tr>
<tr>
<td>task_flags_t</td>
<td>Additional flags for task_create and task_init.</td>
</tr>
<tr>
<td>task_kill_flags_t</td>
<td>Additional flags for task_kill.</td>
</tr>
<tr>
<td>task_onexit_fn_t</td>
<td>Function to be called on task exit.</td>
</tr>
<tr>
<td>task_state_t</td>
<td>State of a task (active, deleted etc.).</td>
</tr>
<tr>
<td>task_stack_fill_state_t</td>
<td>Whether stack filling is enabled or disabled.</td>
</tr>
<tr>
<td>task_stack_fill_t</td>
<td>Stack filling state (specifies enables and value).</td>
</tr>
<tr>
<td>task_status_flags_t</td>
<td>Additional flags for task_status.</td>
</tr>
<tr>
<td>task_status_t</td>
<td>Result of task_status.</td>
</tr>
<tr>
<td>task_t</td>
<td>A task’s static state.</td>
</tr>
<tr>
<td>tdesc_t</td>
<td>A task’s dynamic state.</td>
</tr>
</tbody>
</table>

Table 5.2 Types defined in task.h

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS20_PRIORITY_LEVELS</td>
<td>Number of STLite/OS20 task priorities. Default is 16.</td>
</tr>
<tr>
<td>MAX_USER_PRIORITY</td>
<td>Highest user task priority. Default is 15.</td>
</tr>
<tr>
<td>MIN_USER_PRIORITY</td>
<td>Lowest user task priority. Default is 0.</td>
</tr>
</tbody>
</table>

Table 5.3 Macros defined in task.h
5.19 Task function definitions

**task_context**

Return the current execution context.

**Synopsis:**

```c
#include <task.h>

task_context_t task_context(task_t **task, int* level);
```

**Arguments:**

- `task_t **task` Where to return the task descriptor.
- `int* level` Where to return the interrupt level.

**Results:**

Returns whether the function was called from a task, interrupt or high priority process.

**Errors:**

None.

**Description:**

The `task_context` function returns a description of the context from which it is called, whether this is a task, interrupt or high priority process. This is indicated by one of three possible values:

- If the function was called from an STLite/OS20 task, then it will return `task_context_task`.
- If the function was called from an interrupt handler, then it will return `task_context_interrupt`.
- If the function was called from a high priority process on an ST20-C2, then it will return `task_context_hpp`.

In addition, information about which particular task, interrupt or high priority process the function was called from can be returned. If `task` is not NULL, and the function was called from an STLite/OS20 task or a high priority process, then the corresponding `task_t` will be written into the variable pointed to by `task`. Similarly if `level` is not NULL, and the function was called from an interrupt handler, then the interrupt level will be written into the variable pointed to by `level`.

Determining the `task_t` for a high priority process on an ST20-C2, or the interrupt level on an ST20-C1, can take a variable length of time. So `task_context` will execute faster if these values are not required.

**See also:**

- `task_id`
5.19 Task function definitions

**task_create**

Create an STLite/OS20 task.

**Synopsis:**

```c
#include <task.h>

task_t* task_create(
    void (*Function)(void*),
    void* Param,
    size_t StackSize,
    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 will be passed into Function.
- `size_t StackSize`  
  Required stack size for the task, in bytes.
- `int Priority`  
  Task’s scheduling priority in the range \texttt{MIN\_USER\_PRIORITY} to \texttt{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.

**Results:**

Returns a pointer to the task structure if successful or \texttt{NULL} otherwise. The returned structure pointer should be assigned to a local variable for future use.

**Errors:**

Returns a \texttt{NULL} pointer if an error occurs, either because the task's priority is invalid, or there is insufficient memory for the task’s data structures or stack.

**Description:**

\texttt{task_create()} sets up a function as an STLite/OS20 task and starts the task executing. \texttt{task_create()} returns a pointer to the task control block \texttt{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.

StackSize 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. \texttt{task_create} will automatically call \texttt{memory_allocate()} in order to allocate the stack on the system memory partition.

Param is a pointer to the arguments to Function. If Function has a number of 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 will also be allocated by `task_create` calling
`memory_allocate()`. The task descriptor (`tdesc_t`) will be allocated from the
internal memory partition, the task state (`task_t`) from the system memory parti-
tion.

**Priority** is the task’s scheduling priority.

**Name** is the name of the task, which will be 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 will result in the default behavior, however, other options
can be specified which will change the behavior of the task.

For the ST20-C2 this is used to create tasks which will execute using the ST20’s
hardware high priority processes. Tasks which execute as high priority processes
are not scheduled using the STLite/OS20 scheduler, but the hardware scheduler
built into the ST20-C2. The effect of this is that they can be scheduled very rapidly,
but there are some restrictions on their usage. In particular tasks executing at high
priority cannot:

- use priority semaphores;
- use message queues;
- use task locks (although interrupt locks work for high priority processes);
- change their priority (using `task_priority_set`).

The **Priority** parameter is ignored for tasks created as high priority processes.

Also note that the units of time are different for high priority processes, see
Chapter 8.

The other possible value for **flags** is `task_flags_suspended`. This can be used
to create tasks which are initially suspended. This means that the task will not run
until it is resumed using the `task_resume` call.

Note that high priority processes cannot be created suspended.

Thus, current possible values for **flags** are:

<table>
<thead>
<tr>
<th>Task flags</th>
<th>Task behavior</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create an STLite/OS20 task. Default.</td>
<td>Any</td>
</tr>
<tr>
<td><code>task_flags_high_priority_process</code></td>
<td>Create the task as a high priority process.</td>
<td>ST20-C2</td>
</tr>
<tr>
<td></td>
<td>(This is ignored on ST20-C1 devices).</td>
<td></td>
</tr>
<tr>
<td><code>task_flags_suspended</code></td>
<td>Create the task already suspended.</td>
<td>Any</td>
</tr>
</tbody>
</table>

Table 5.4 Flag values
Example:

```c
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);
    }
}

main() {
    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);
    }
    ...
}
```

See also:

- `task_delete`
**task_create_sl**  Create an STLite/OS20 task specifying a static link.

**Synopsis:**

```c
#include <task.h>
task_t* task_create(
    void (*Function)(void*),
    void* Param,
    void* StaticLink
    size_t StackSize,
    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 will be passed into `Function`.
- `void* StaticLink`: Static link to be used when calling `Function`.
- `size_t StackSize`: 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.

**Results:**

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, either because the task’s priority is invalid, or there is insufficient memory for the task’s data structures or stack.

**Description:**

`task_create_sl()` sets up a function as an STLite/OS20 task and starts the task executing. `task_create_sl()` 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.

`Param` is a pointer to the arguments to `Function`. If `Function` has a number of parameters, these should be combined into a structure and the address of the structure provided as the argument to `task_create_sl()`. When the task is
started it begins executing as if \texttt{Function} were called with the single argument \texttt{Param}.

\texttt{StaticLink} is the static link which should be used when calling \texttt{Function}. This will normally be obtained as a result of loading an RCU. See section 5.5.1.

\texttt{StackSize} 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. \texttt{task_create_sl} will automatically call \texttt{memory_allocate()} in order to allocate the stack on the system memory partition.

The task’s data structures will also be allocated by \texttt{task_create_sl} calling \texttt{memory_allocate()}. The task descriptor (\texttt{tdesc_t}) will be allocated from the internal memory partition, the task state (\texttt{task_t}) from the system memory partition.

\texttt{Priority} is the task’s scheduling priority.

\texttt{Name} is the name of the task, which will be passed to the debugger (if present) so that the task can be correctly identified in the debugger’s task list.

\texttt{flags} is used to give additional information about the task. Normally flags should be specified as 0, which will result in the default behavior, however, other options can be specified which will change the behavior of the task.

For the ST20-C2 this is used to create tasks which will execute using the ST20’s hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the STLite/OS20 scheduler, but the hardware scheduler built into the ST20-C2. The effect of this is that they can be scheduled very rapidly, but there are some restrictions on their usage. In particular tasks executing at high priority cannot:

- use priority semaphores;
- use message queues;
- use task locks (although interrupt locks work for high priority processes);
- change their priority (using \texttt{task_priority_set}).

The \texttt{Priority} parameter is ignored for tasks created as high priority processes.

Also note that the units of time are different for high priority processes, see Chapter 8.

The other possible value for \texttt{flags} is \texttt{task_flags_suspended}. This can be used to create tasks which are initially suspended. This means that the task will not run until it is resumed using the \texttt{task_resume} call.

Note that high priority processes cannot be created suspended.
Thus, current possible values for flags are:

<table>
<thead>
<tr>
<th>Task flags</th>
<th>Task behavior</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create an STLite/OS20 task. Default.</td>
<td>Any</td>
</tr>
<tr>
<td>task_flags_high_priority_process</td>
<td>Create the task as a high priority process.</td>
<td>ST20-C2</td>
</tr>
<tr>
<td></td>
<td>(This is ignored on ST20-C1 devices).</td>
<td></td>
</tr>
<tr>
<td>task_flags_suspended</td>
<td>Create the task already suspended.</td>
<td>Any</td>
</tr>
</tbody>
</table>

Table 5.5 Flag values

See also:

task_create task_delete task_init_sl
**task_data**

Retrieve a task’s data pointer.

**Synopsis:**

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

**Arguments:**

- `task_t* Task`  
  Pointer to the task structure.

**Results:**

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.

**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 5.14.

**See also:**

- `task_data_set`
task_data_set

Set a task's data pointer.

Synopsis:

```c
#include <task.h>
void* task_data_set(task_t* Task, void* NewData);
```

Arguments:

- `task_t* Task`  
  Pointer to the task structure.
- `void* NewData`  
  New data pointer for the task.

Results:

`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.

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 5.14

See also:

- `task_data`
5.19 Task function definitions

\section*{task_delay}

Delay the calling task for a period of time.

**Synopsis:**

```
#include <task.h>
void task_delay(clock_t delay);
```

**Arguments:**

- **clock_t delay**
  
The period of time to delay the calling task.

**Results:**

None

**Errors:**

None

**Description:**

Delay the calling task for the specified period of time. `delay` is specified in ticks, which is an implementation dependent quantity, see Chapter 8.

**See also:**

- `task_delay_until`
task_delay_until

Delay the calling task until a specified time.

Synopsis:

```
#include <task.h>
void task_delay_until(clock_t delay);
```

Arguments:

- `clock_t delay` The time period during which the calling task is delayed.

Results:

None

Errors:

None

Description:

Delay the calling task until the specified time. If `delay` is before the current time, then this function returns immediately. `delay` is specified in ticks, which is an implementation dependent quantity, see Chapter 8.

See also:

- `task_delay`
**task_delete**

Delete an STLite/OS20 task.

**Synopsis:**

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

**Arguments:**

- `task_t *task`  
  Task to delete.

**Results:**

Returns 0 on success, -1 on failure.

**Errors:**

If the task has not yet terminated, then this will fail.

**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 will fail.

**See also:**

- `task_create`  
- `task_kill`
task_exit

Synopsis:

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

Arguments:

    int param

Parameter to pass to onexit handler.

Results:

    None.

Errors:

    None.

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.

Synopsis:

```c
#include <task.h>
task_t* task_id(void);
```

Arguments:

None.

Results:

Returns a pointer to the STLite/OS20 task structure of the calling task.

Errors:

None.

Description:

task_id returns a pointer to the task structure of the currently active task.

See also:

task_create
**task_immortal**

Make the current task immortal.

**Synopsis:**

```c
#include <task.h>
void task_immortal(void)
```

**Arguments:**

None.

**Results:**

None.

**Errors:**

None.

**Description:**

task_immortal makes the current task immortal. If an attempt is made to kill a task whilst it is immortal, it will not die immediately, but will continue running until it becomes mortal again, and will then die.

**See also:**

`task_kill` `task_mortal`
**task_init**

Initialize an STLite/OS20 task.

**Synopsis:**

```c
#include <task.h>

int task_init(
    void (*Function)(void*),
    void* Param,
    void* Stack,
    size_t StackSize,
    task_t* Task,
    tdesc_t* Tdesc,
    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 will be passed into `Function`.
- `void* Stack`: Pointer to the stack for the task.
- `size_t StackSize`: Size in bytes of Stack.
- `task_t* Task`: Pointer to the task’s task control block.
- `tdesc_t* Tdesc`: Pointer to the task’s descriptor.
- `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 effect task behavior.

**Results:**

Returns 0 on success, -1 if an error occurs.

**Errors:**

Returns -1 if the priority is illegal, or the stack size too small for the initial stack frame.
Description:

`task_init()` sets up a function as an STLite/OS20 task and starts the task executing. If the call succeeds, then `Task` should be used in any subsequent calls to refer to the task.

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

`Stack` is a pointer to the base of the stack for the task, which is of `StackSize` bytes. It is important that enough stack space is allocated, if not, the results of running the task are undefined.

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

`Task` and `Tdesc` are pointers to data structures which will be used by STLite/OS20 to store details about the task. These structures should be declared before `task_init` is called and after the task is created, these structures should not be modified by the user.

`Name` is the name of the task, which will be 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 will result in the default behavior, however other options can be specified which will change the behavior of the task.

For the ST20-C2 this is used to create tasks which will execute using the ST20’s hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the STLite/OS20 scheduler, but the hardware scheduler built into the ST20. The effect of this is that they can be scheduled very rapidly, but there are some restrictions on their usage. In particular tasks executing at high priority cannot:

- use priority semaphores
- use message queues
- use task locks (although interrupt locks work for high priority processes)
- change their priority (using `task_priority_set`)

The priority parameter is ignored for tasks created as high priority processes, and the `TDesc` should be specified as `NULL`.

Also note that the units of time are different for high priority processes, see Chapter 8.

The other possible value for `flags` is `task_flags_suspended`. This can be used to create tasks which are initially suspended. This means that the task will not run until it is resumed using the `task_resume` call.

Note that high priority processes cannot be created suspended.
5.19 Task function definitions

Thus, current possible values for flags are:

<table>
<thead>
<tr>
<th>Task flags</th>
<th>Task behavior</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create an STLite/OS20 task. Default.</td>
<td>Any</td>
</tr>
<tr>
<td>task_flags_high_priority_process</td>
<td>Create the task as a high priority process. (This is ignored on ST20-C1 devices).</td>
<td>ST20-C2</td>
</tr>
<tr>
<td>task_flags_suspended</td>
<td>Create the task already suspended</td>
<td>Any</td>
</tr>
</tbody>
</table>

Table 5.6 Flag values

Example:

```
#include <task.h.>
#define STACK_SIZE 1024

task_t task;
tdesc_t tdesc;

char stack[STACK_SIZE];

task_init(fn_ptr, NULL, stack, STACK_SIZE,
            &task, &tdesc, 10, "test", 0);
```

See also:

task_create
task_init_sl

Initialize an STLite/Os20 task specifying a static link.

Synopsis:

```c
#include <task.h>

int task_init_sl(
    void (*Function)(void*),
    void* Param,
    void* StaticLink,
    void* Stack,
    size_t StackSize,
    task_t* Task,
    tdesc_t* Tdesc,
    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 will be passed into `Function`.
- `void* StaticLink`  
  Static link to be used when calling `Function`.
- `void* Stack`  
  Pointer to the stack for the task
- `size_t StackSize`  
  Size in bytes of `Stack`
- `task_t* Task`  
  Pointer to the task's task control block
- `tdesc_t* Tdesc`  
  Pointer to the task's descriptor
- `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 effect task behavior.

Results:

Returns 0 on success, -1 if an error occurs.

Errors:

Returns -1 if the priority is illegal, or the stack size too small for the initial stack frame.
5.19 Task function definitions

Description:

`task_init()` sets up a function as an STLite/OS20 task and starts the task executing. If the call succeeds, then `Task` should be used in any subsequent calls to refer to the task.

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

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

`StaticLink` is the static link which should be used when calling `Function`. This will normally be obtained as a result of loading an RCU. See section 5.5.1.

`Stack` is a pointer to the base of the stack for the task, which is of `StackSize` bytes. It is important that enough stack space is allocated, if not, the results of running the task are undefined.

`Task` and `Tdesc` are pointers to data structures which will be used by STLite/OS20 to store details about the task. These structures should be declared before `task_init` is called and after the task is created, these structures should not be modified by the user.

`Name` is the name of the task, which will be 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 will result in the default behavior, however other options can be specified which will change the behavior of the task.

For the ST20-C2 this is used to create tasks which will execute using the ST20's hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the STLite/OS20 scheduler, but the hardware scheduler built into the ST20. The effect of this is that they can be scheduled very rapidly, but there are some restrictions on their usage. In particular tasks executing at high priority cannot:

- use priority semaphores
- use message queues
- use task locks (although interrupt locks work for high priority processes)
- change their priority (using `task_priority_set`)

The priority parameter is ignored for tasks created as high priority processes, and the `TDesc` should be specified as `NULL`.

Also note that the units of time are different for high priority processes, see Chapter 8.

The other possible value for `flags` is `task_flags_suspended`. This can be used to create tasks which are initially suspended. This means that the task will not run until it is resumed using the `task_resume` call.
Note that high priority processes cannot be created suspended.
Thus, current possible values for flags are:

<table>
<thead>
<tr>
<th>Interrupt flags</th>
<th>Interrupt behavior</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Create an STLite/OS20 task. Default.</td>
<td>Any</td>
</tr>
<tr>
<td>task_flags_high_priority_process</td>
<td>Create the task as a high priority process. (This is ignored on ST20-C1 devices).</td>
<td>ST20-C2</td>
</tr>
<tr>
<td>task_flags_suspended</td>
<td>Create the task already suspended</td>
<td>Any</td>
</tr>
</tbody>
</table>

Table 5.7 Flag values

See also:

task_create task_create_sl task_init
**task_kill**

**Synopsis:**

```c
#include <task.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.

**Results:**

Returns 0 if the task is successfully killed, -1 if it cannot be killed.

**Errors:**

If the task has been deleted, is a high priority process, then this call will fail.

**Description:**

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

`status` is the exit status for the task. Thus `task_kill` can be viewed as a way of forcing the task to call:

```c
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 will terminate immediately, never running again.

A task can temporarily make itself immune to being killed by calling `task_immortal`, see section 5.11 for more details on this. When a task which has made itself immortal is killed, `task_kill` will return immediately, but the killed task will not die until it makes itself mortal again.

**Note:** that `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.
Example:

```c
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_immortal task_mortal
```


**task_lock**

Prevent task rescheduling.

**Synopsis:**

```c
#include <task.h>
void task_lock(void);
```

**Arguments:**

None.

**Results:**

None.

**Errors:**

None.

**Description:**

This function prevents the kernel scheduler from preempting or timeslicing the current task, although the task can still be interrupted by interrupt handlers (and high-priority processes on the ST20-C2).

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 preempted by another task. If the task deschedules the lock will be terminated. Calls to `task_lock()` can be nested, and the lock will not be released until an equal number of calls to `task_unlock()` have been made.

**See also:**

`interrupt_lock` `task_unlock`
task_mortal

Make the current task mortal

Synopsis:

```c
#include <task.h>

void task_mortal(void)
```

Arguments:
None.

Results:
None.

Errors:
None.

Description:

`task_mortal` makes the current task mortal again. If an attempt had been made to kill the task whilst it was immortal, it will die as soon as `task_mortal` is called.

Calls to `task_immortal` are cumulative. That is, if a task makes two calls to `task_immortal`, then two calls to `task_mortal` will be required before it becomes mortal again.

See also:

`task_immortal` `task_kill`
### task_name

Return the name of the specified task.

#### Synopsis:

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

#### Arguments:

- `task_t* task`
  
  Task to return the name of.

#### Results:

- The name of the specified task.

#### Errors:

None.

#### 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_init`
**task_onexit_set**  
Set the task *onexit* handler.

**Synopsis:**

```c
#include <task.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.

**Results:**

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

**Errors:**

- None.

**Description:**

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

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

where:

- `task` is the task pointer of the task which has just exited, and
- `param` is the parameter which was passed to `task_exit`, or the value the task’s entry point function returned.

**See also:**

- `task_exit`
5.19 Task function definitions

**task_onexit_set_sl**  
Set the task onexit handler specifying a static link.

**Synopsis:**

```c
#include <task.h>

task_onexit_fn_t task_onexit_set_sl(
    task_onexit_fn_t fn
    void* sl);
```

**Arguments:**

- `task_onexit_fn_t fn`  
  Task onexit handler to be called.
- `void* sl`  
  Static link to be used when calling `fn`.

**Results:**

Returns the previous onexit handler, or NULL if none had previously been set.

**Errors:**

None.

**Description:**

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

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

where:

- `task` is the task pointer of the task which has just exited, and
- `param` is the parameter which was passed to `task_exit`, or the value the task's entry point function returned.

`sl` is the static link which should be used when calling `fn`. This will normally be obtained as a result of loading an RCU. This does not have to be the same static link which was used when the task was created. See section 5.5.1.

**See also:**

- `task_exit`  
  task_onexit_set_sl
**task_priority**

Retrieve a task’s priority.

**Synopsis:**

```c
#include <task.h>

int task_priority(task_t* Task);
```

**Arguments:**

- `task_t* Task`: Pointer to the task structure.

**Results:**

Returns the STLite/OS20 priority of the task pointed to by `Task`. If `Task` is NULL the return result is the priority of the calling task. If `Task` was created as an ST20-C2 high priority process then `task_priority` will return the value -1.

**Errors:**

None.

**Description:**

`task_priority()` retrieves the STLite/OS20 priority of the task specified by `Task` or the priority of the currently active task if `Task` is NULL.

**See also:**

- `task_priority_set`
task_priority_set

Set a task's priority.

Synopsis:

#include <task.h>

int task_priority_set(task_t* Task, int NewPriority);

Arguments:

  task_t* Task       Pointer to the task structure.
  int NewPriority    Desired STLite/OS20 priority value for the task.

Results:

  task_priority_set() returns the task's previous STLite/OS20 priority. If Task is NULL the return result is the priority of the calling task.

Errors:

  None.

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, then tasks may be rescheduled.

See also:

  task_priority
task_reschedule

Reschedule the current task.

Synopsis:

    #include <task.h>
    void task_reschedule(void);

Arguments:

    None.

Results:

    None.

Errors:

    None.

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, then it will have had no effect, otherwise it will have performed a timeslice at the current priority.

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

On the ST20-C2, this can be called from tasks running as high priority processes, in which case the task will effectively timeslice, something which high priority processes will never do automatically.
task_resume

Resume a suspended task.

Synopsis:

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

Arguments:

- `task_t* Task` Pointer to the task structure.

Results:

Returns 0 if the task was successfully resumed, or -1 if it could not be resumed.

Errors:

If the task is not suspended, then the call will fail.

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_init`.

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

If the task was waiting for an event when it was suspended, then the event must also occur before the task will start executing. When a task is resumed it will start 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.

Synopsis:

```c
#include <task.h>

int task_stack_fill(task_stack_fill_t* fill);
```

Arguments:

- `task_stack_fill_t* fill`: A pointer to a structure to be filled in.

Results:

- Returns 0 for success, -1 if an error occurs.

Errors:

- Returns -1 if `fill` is NULL.

Description:

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

Table 5.8 shows the layout of the structure `task_stack_fill_t`.

Example:

```c
#include <task.h>

int result;

int task_stack_fill_t settings;
result = task_stack_fill(&settings);
```

See also:

- `task_stack_fill_set`
- `task_init`
- `task_create`
**task_stack_fill_set** Set task stack fill settings.

**Synopsis:**

```c
#include <task.h>

int task_stack_fill_set(task_stack_fill_t* fill);
```

**Arguments:**

- `task_stack_fill_t* fill` A pointer to new settings.

**Results:**

Returns 0 on success, −1 if an error occurs.

**Errors:**

Returns −1 if the new settings are invalid.

**Description:**

The function `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_init()` or `task_create()` will 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_init()` or `task_create()` will have its stack initialized by overwriting the whole contents of the stack with the value 0x12345678.

Table 5.8 shows the layout of the `task_stack_fill_t` structure.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task_stack_fill_state</code></td>
<td>Enable/Disable stack filling (See Table 5.9).</td>
</tr>
<tr>
<td><code>task_stack_fill_pattern</code></td>
<td>Pattern value used when a stack is initialized.</td>
</tr>
</tbody>
</table>

Table 5.8 Layout of structure `task_stack_fill_t`

Table 5.9 shows all the flag values which can be used in the field `task_stack_fill_state`. Any other value not in the table will cause `task_stack_fill_set()` to return −1.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task_stack_fill_state_off</code></td>
<td>Disable task stack filling.</td>
</tr>
<tr>
<td><code>task_stack_fill_state_on</code></td>
<td>Enable task stack filling.</td>
</tr>
</tbody>
</table>

Table 5.9 Flags used by `task_stack_fill_state`
Example:

```c
#include <task.h>

task_stack_fill_t options = {
    task_stack_fill_state_on,
    0x76543210
};

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

See also:

```
task_create task_init task_stack_fill
```
**task_status**  
Return information about the specified task.

**Synopsis:**

```c
#include <task.h>
int task_status(
    task_t* Task,
    task_status_t *Status,
    task_status_flags_t Flag);
```

**Arguments:**

- `task_t* Task`  
  Pointer to the task structure.
- `task_status_t *Status`  
  Where to return the status information.
- `task_status_flags_t Flag`  
  What information to return.

**Results:**

Returns 0 if the status was successfully reported, -1 if it failed.

**Errors:**

If the task does not exist then the call will fail.

**Description:**

This function returns information about the specified task. If `Task` is NULL then 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:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task_stack_base</code></td>
<td>Base address of the task's stack.</td>
</tr>
<tr>
<td><code>task_stack_size</code></td>
<td>Size of the task's stack in bytes.</td>
</tr>
<tr>
<td><code>task_stack_used</code></td>
<td>Amount of stack used by the task in bytes.</td>
</tr>
<tr>
<td><code>task_time</code></td>
<td>CPU time used by the task.</td>
</tr>
</tbody>
</table>

Table 5.10 `task_status_t` fields

**Note** that the `task_time` field is only valid when STLite/OS20 has been built with time logging enabled.

The `Flags` parameter is used to indicate which values should be returned. Values which can be determined immediately (`task_stack_base`, `task_stack_size` and `task_time`) will always be returned. If only these fields are required then `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`.

**See also:**

`task_stack_fill_set`
**task_suspend**

Suspend a specified task.

**Synopsis:**

```
#include <task.h>

int task_suspend(task_t* Task);
```

**Arguments:**

- `task_t* Task`: Pointer to the task structure.

**Results:**

Returns 0 if the task was successfully suspended, or -1 if it could not be suspended.

**Errors:**

If the task has been deleted then the call will fail.

**Description:**

This function suspends the specified task. If `Task` is NULL then this will suspend the current task.

`task_suspend` will stop the task from executing immediately, until it is resumed using `task_resume`.

**See also:**

- `task_resume`
**task_unlock**

Allow task rescheduling.

**Synopsis:**

```c
#include <task.h>
void task_unlock(void);
```

**Arguments:**

None.

**Results:**

None.

**Errors:**

None.

**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 which calls this function) will continue 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 will not be released until an equal number of calls to `task_unlock()` have been made.

**See also:**

`interrupt_lock` `task_lock`
**task_wait**

Waits until one of a list of tasks completes.

**Synopsis:**

```c
#include <task.h>
#include <ostime.h>

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

**Arguments:**

- **task_t **tasklist**
  
  Pointer to a list of task_t pointers.

- **int ntasks**
  
  The number of tasks in tasklist.

- **const clock_t *timeout**
  
  Maximum time to wait for tasks to terminate. Expressed in ticks or as TIMEOUT_IMMEDIATE or TIMEOUT_INFINITY.

**Results:**

The index into the array of the task which has terminated, or -1 if the timeout occurs.

**Errors:**

None.

**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, task_t and tdesc_t data structures.

`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 will be ignored.

`timeout` is a pointer to the timeout value. If this time is reached then the function will return the value -1.

The timeout value may be specified in ticks, which is an implementation dependent quantity, see Chapter 8.

Two special values can be specified for `timeout`: 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.

**See also:**

`task_create` `task_init`
5.19 Task function definitions
6 Semaphores

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

6.1 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:

```c
semaphore_t* semaphore_create_fifo (int value);
void semaphore_init_fifo(semaphore_t *sem, int value);
semaphore_t* semaphore_create_priority (int value);
void semaphore_init_priority (semaphore_t *sem, int value);
```

or if a timeout capability is required while waiting for a semaphore, use the timeout versions of the above functions:

```c
semaphore_t* semaphore_create_fifo_timeout (int value);
void semaphore_init_fifo_timeout(
    semaphore_t *sem, int value);
semaphore_t* semaphore_create_priority_timeout (int value);
void semaphore_init_priority_timeout (semaphore_t *sem, int value);
```

The `create_` versions of the functions will allocate memory for the semaphore automatically, while the `init_` versions enable the user to specify a pointer to the semaphore, using the data structure `semaphore_t`.

The semaphores which STLite/OS20 provides differ in the way in which tasks are queued. Normally tasks are queued in the order which they call `semaphore_wait`, in which case this is termed a FIFO semaphore. Semaphores of this type are created using `semaphore_create_fifo` or `semaphore_init_fifo` 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 will be 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 will be 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_init_priority` or one of the timeout versions of these functions.

Semaphores may be acquired by the function:

```c
void semaphore_wait (semaphore_t* Sem);
```
For semaphores created via one of the timeout functions, then the following function may also be used:

```c
int semaphore_wait_timeout(
    semaphore_t* Sem
    const clock_t *timeout);
```

When a task wants to acquire a semaphore, it calls `semaphore_wait`. At this point if the semaphore count is greater than 0, then the count will be decremented, and the task continues. If however, the count is already 0, then the task will add itself to the queue of tasks waiting for the semaphore and deschedule itself. Eventually another task should release the semaphore, and the first waiting task will be able to continue. In this way, when the task returns from the function it will have acquired the semaphore.

If you want to make certain that the task does not wait indefinitely for a particular semaphore then the timeout versions of the semaphore functions may be used. Note that these functions cannot use the hardware support for semaphores, and so are larger and slower than the non-timeout versions.

`semaphore_wait_timeout` enables a timeout to be specified. If this time is reached before the semaphore is acquired then the function will return 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`, i.e. the task will wait indefinitely for the semaphore to become available.

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

```c
void semaphore_signal (semaphore_t* Sem);
```

This will look at the queue of waiting tasks, and if the queue is not empty, remove the first task from the queue, and start it running. If there are no tasks waiting, then the semaphore count will be incremented, indicating that the semaphore is available.

If a semaphore is deleted using `semaphore_delete` then how the memory is released will depend on whether the semaphore was created by the `create` or `init` version of the function. See the functional description of `semaphore_delete` in section 6.4.

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.

FIFO semaphores can also be used to synchronize the activity of low priority tasks with high priority tasks.
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 will wait 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 one, 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. Thus the first task which tries to enter the critical region will successfully take the semaphore, and any others will be 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 will deschedule. Later the interrupt handler will perform a semaphore_signal, which will reschedule the task. This process can then be repeated, with the semaphore count never changing from zero.

All the STLite/OS20 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 will return immediately having decremented the counter. If no devices are available then the task will be added to the queue. When a task has finished using a device it calls semaphore_signal to release it.
6.3 Semaphore header file: semaphor.h

All the definitions related to interrupts are in the single header file, semaphor.h, see Table 6.1 and Table 6.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore_create_fifo</td>
<td>Create a FIFO queued semaphore</td>
</tr>
<tr>
<td>semaphore_create_fifo_timeout</td>
<td>Create a FIFO queued semaphore with timeout</td>
</tr>
<tr>
<td>semaphore_create_priority</td>
<td>Create a priority queued semaphore</td>
</tr>
<tr>
<td>semaphore_create_priority_timeout</td>
<td>Create a priority queued semaphore with timeout</td>
</tr>
<tr>
<td>semaphore_delete</td>
<td>Delete a semaphore</td>
</tr>
<tr>
<td>semaphore_init_fifo</td>
<td>Initialize a FIFO queued semaphore</td>
</tr>
<tr>
<td>semaphore_init_fifo_timeout</td>
<td>Initialize a FIFO queued semaphore with timeout</td>
</tr>
<tr>
<td>semaphore_init_priority</td>
<td>Initialize a priority queued semaphore</td>
</tr>
<tr>
<td>semaphore_init_priority_timeout</td>
<td>Initialize a priority queued semaphore with timeout</td>
</tr>
<tr>
<td>semaphore_signal</td>
<td>Signal a signal</td>
</tr>
<tr>
<td>semaphore_wait</td>
<td>Wait for a signal</td>
</tr>
<tr>
<td>semaphore_wait_timeout</td>
<td>Wait for a semaphore or a timeout</td>
</tr>
</tbody>
</table>

Table 6.1 Functions defined in semaphor.h

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore_t</td>
<td>A semaphore</td>
</tr>
</tbody>
</table>

Table 6.2 Types defined in semaphor.h

All functions are callable from an STLite/OS20 task. Functions in Table 6.3 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor. Both the type of object which is being waited on, as well as the priority of the task affect whether the semaphore function can be called from an ISR or HPP.

<table>
<thead>
<tr>
<th>Function</th>
<th>Object</th>
<th>Task</th>
<th>ISR</th>
<th>HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore_signal</td>
<td>Non-timeout FIFO</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>semaphore_wait</td>
<td>Non-timeout FIFO</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>semaphore_wait_timeout</td>
<td>Non-timeout FIFO</td>
<td>Yes (1)</td>
<td>No</td>
<td>Yes (1)</td>
</tr>
<tr>
<td>semaphore_signal</td>
<td>Timeout FIFO</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>semaphore_wait</td>
<td>Timeout FIFO</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>semaphore_wait_timeout</td>
<td>Timeout FIFO</td>
<td>Yes</td>
<td>Yes (2)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.3 Semaphore functions callable from an ISR or HPP
### 6 Semaphores

<table>
<thead>
<tr>
<th>Function</th>
<th>Object</th>
<th>Task</th>
<th>ISR</th>
<th>HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>semaphore_signal</td>
<td>Non-timeout priority</td>
<td>Yes</td>
<td>No (3)</td>
<td>No (3)</td>
</tr>
<tr>
<td>semaphore_wait</td>
<td>Non-timeout priority</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>semaphore_wait_timeout</td>
<td>Non-timeout priority</td>
<td>Yes (1)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>semaphore_signal</td>
<td>Timeout priority</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>semaphore_wait</td>
<td>Timeout priority</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>semaphore_wait_timeout</td>
<td>Timeout priority</td>
<td>Yes</td>
<td>Yes (2)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 6.3  Semaphore functions callable from an ISR or HPP

**Notes:**

1. Timeout value is ignored. Behaves as though `TIMEOUT_INFINITY` was specified.
2. Can only be used with `TIMEOUT_IMMEDIATE`.
3. This may be changed in a later revision.
6.4 Semaphore function definitions

**semaphore_create_fifo**  Create a FIFO queued semaphore.

**Synopsis:**

```c
#include <semaphor.h>
semaphore_t* semaphore_create_fifo(int value);
```

**Arguments:**

- `int value`  The initial value of the semaphore.

**Results:**

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

**Errors:**

NULL if there is insufficient memory for the semaphore.

**Description:**

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

**See also:**

- `semaphore_init_fifo`
- `semaphore_create_priority`
semaphore_create_fifo_timeout

Create a FIFO queued semaphore with timeout capability.

Synopsis:

```c
#include <semaphor.h>
semaphore_t* semaphore_create_fifo_timeout(int value);
```

Arguments:

- `int value`  
  The initial value of the semaphore.

Results:

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

Errors:

- NULL if there is insufficient memory for the semaphore.

Description:

This function creates a counting semaphore, initialized to value, which can be used in calls to `semaphore_wait_timeout()`. The memory for the semaphore structure is allocated from the system memory partition. Semaphores created with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` or `semaphore_wait_timeout()`, it will always be appended to the end of the queue of waiting tasks, irrespective of its priority.

See also:

- `semaphore_create_fifo`
- `semaphore_create_priority_timeout`
- `semaphore_init_fifo_timeout`
semaphore_create_priority

Create a priority queued semaphore.

Synopsis:
#include <semaphor.h>
semaphore_t* semaphore_create_priority(int value);

Arguments:
int value
The initial value of the semaphore.

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

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

Description:
This function creates a counting semaphore, initialized to value. The memory for
the semaphore structure is allocated from the system memory partition. Semaph-

ores created with this function have the usual semaphore semantics, except that
when a task calls semaphore_wait() it will be 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_init_priority
**semaphore_create_priority_timeout**

Create a priority queued semaphore with timeout capability.

**Synopsis:**

```c
#include <semaphor.h>
semaphore_t* semaphore_create_priority_timeout(int value);
```

**Arguments:**

- `int value` The initial value of the semaphore.

**Results:**

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

**Errors:**

NULL if there is insufficient memory for the semaphore.

**Description:**

This function creates a counting semaphore, initialized to `value`, which can be used in calls to `semaphore_wait_timeout()`. The memory for the semaphore structure is allocated from the system memory partition. Semaphores created with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` or `semaphore_wait_timeout()` it will be 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_timeout`
- `semaphore_init_priority_timeout`
- `semaphore_create_priority`
semaphore_delete

Delete a semaphore.

Synopsis:

#include <semaphor.h>
void semaphore_delete(semaphore_t *sem);

Arguments:

semaphore_t *sem  Semaphore to delete.

Results:

None.

Errors:

None.

Description:

This function allows a semaphore to be deleted. If the semaphore was created using semaphore_create then this will also free the memory used by the semaphore. If it was created using semaphore_init then the user is responsible for freeing the semaphore data structure.

Note: that if any tasks are waiting on the semaphore when it is deleted, this will cause the following fatal error to be reported:

delete handler- operation on deleted object attempted

Similarly any attempt to use the deleted semaphore will report the same error.

See also:

semaphore_create_priority semaphore_create_fifo
semaphore_init_fifo semaphore_init_priority
semaphore_init_fifo

Initialize a FIFO queued semaphore.

Synopsis:

```c
#include <semaphor.h>
void semaphore_init_fifo(semaphore_t *sem, int value);
```

Arguments:

- `semaphore_t* sem` - The semaphore to be initialized.
- `int value` - The initial value of the semaphore.

Results:

None.

Errors:

None.

Description:

This function initializes a counting semaphore to `value`. Semaphores initialized with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` it will always be appended to the end of the queue of waiting tasks, irrespective of its priority.

`sem` should be declared before the call to `semaphore_init_fifo` is made.

See also:

`semaphore_create_fifo` `semaphore_init_priority`
semaphore_init_fifo_timeout

Initialize a FIFO queued semaphore with timeout capability.

Synopsis:

```c
#include <semaphor.h>
void semaphore_init_fifo_timeout(
    semaphore_t *sem,
    int value);
```

Arguments:

- `semaphore_t* sem` The semaphore to be initialized.
- `int value` The initial value of the semaphore.

Results:

None.

Errors:

None.

Description:

This function initializes a counting semaphore to `value`, which can be used in calls to `semaphore_wait_timeout()`. Semaphores initialized with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` or `semaphore_wait_timeout()` it will always be appended to the end of the queue of waiting tasks, irrespective of its priority.

`sem` should be declared before the call to `semaphore_init_fifo_timeout` is made.

See also:

- `semaphore_create_fifo_timeout`
- `semaphore_init_priority_timeout`
- `semaphore_init_fifo`
semaphore_init_priority

**Synopsis:**

```
#include <semaphor.h>
void semaphore_init_priority(semaphore_t *sem, int value);
```

**Arguments:**

- `semaphore_t* sem` (The semaphore to be initialized.)
- `int value` (The initial value of the semaphore.)

**Results:**

None.

**Errors:**

None.

**Description:**

This function initializes a counting semaphore to `value`. Semaphores initialized with this function have the usual semaphore semantics, except that when a task calls `semaphore_wait()` it will be 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.

`sem` should be declared before the call to `semaphore_init_priority` is made.

**See also:**

`semaphore_create_priority`
semaphore_init_priority_timeout  Initialize a priority queued semaphore with timeout capability.

Synopsis:

```c
#include <semaphor.h>

void semaphore_init_priority_timeout(
    semaphore_t *sem,
    int value);
```

Arguments:
- semaphore_t* sem  The semaphore to be initialized.
- int value  The initial value of the semaphore.

Results:
None.

Errors:
None.

Description:
This function initializes a counting semaphore to value, which can be used in calls to semaphore_wait_timeout(). Semaphores initialized with this function have the usual semaphore semantics, except that when a task calls semaphore_wait() or semaphore_wait_timeout() it will be 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.

sem should be declared before the call to semaphore_init_priority_timeout is made.

See also:
- semaphore_create_priority_timeout
- semaphore_init_fifo_timeout
- semaphore_init_priority
semaphore_signal

Signal a semaphore.

Synopsis:

```c
#include <semaphor.h>
void semaphore_signal(semaphore_t* Sem);
```

Arguments:

```
semaphore_t* Sem
```
A pointer to a semaphore.

Results:

None.

Errors:

None.

Description:

Perform a signal operation on the specified semaphore. The exact behavior of this function depends on the semaphore type. The operation will check the queue of tasks waiting for the semaphore, if the list is not empty, then the first task on the list will be restarted, possibly preemption the current task. Otherwise the semaphore count will be incremented, and the task continues running.

This function can be called from high priority processes (on the ST20-C2) and interrupt handlers, as well as STLite/OS20 tasks, see Table 6.3 for more details.

See also:

```
semaphore_wait
```
semaphore_wait

Wait for a semaphore.

Synopsis:

```c
#include <semaphor.h>

int semaphore_wait(semaphore_t* Sem);
```

Arguments:

- `semaphore_t* Sem` A pointer to a semaphore.

Results:

Always returns 0.

Errors:

None.

Description:

Perform a wait operation on the specified semaphore. The exact behavior of this function depends on the semaphore type. The operation will check the semaphore counter, and if it is 0, then add the current task to the list of queued tasks, before descheduling. Otherwise the semaphore counter will be decremented, and the task continues running.

See also:

- `semaphore_signal`
- `semaphore_wait_timeout`
semaphore_wait_timeout

Wait for a semaphore or a timeout.

Synopsis:

```c
#include <semaphor.h>
#include <ostime.h>

int semaphore_wait_timeout(
    semaphore_t* Sem
    const clock_t *timeout);
```

Arguments:

- `semaphore_t* Sem` A pointer to a semaphore.
- `const clock_t* timeout` Maximum time to wait for the semaphore. Expressed in ticks or as `TIMEOUT_IMMEDIATE` or `TIMEOUT_INFINITY`.

Results:

Returns 0 on success, -1 if timeout occurs.

Errors:

None.

Description:

Perform a wait operation on the specified semaphore. If the time specified by the timeout is reached before a signal operation is performed on the semaphore, then `semaphore_wait_timeout` will return the value -1 indicating that a timeout occurred, and the semaphore count will be unchanged. If the semaphore is signalled before the timeout is reached, then `semaphore_wait_timeout` will return 0. **Note** that 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 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` will cause the semaphore to be polled, that is, the function will always return immediately. If the semaphore count is greater than zero, then it will have been successfully decremented, and the function returns 0, otherwise the function will return a value of -1. A timeout of `TIMEOUT_INFINITY` will behave exactly as `semaphore_wait`.

Example:

```c
clock_t time;
time = time_plus(time_now(), 15625);
semaphore_wait_timeout(semaphore, &time);
```

See also:

`semaphore_signal` `semaphore_wait`
7 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. Message queues are, however, subject to the following restriction:

- Message queues may only be used from interrupt handlers if the timeout versions of the message handling functions are used and a timeout period of `TIMEOUT_IMMEDIATE` is used, see section 7.3. This prevents the interrupt handler from blocking on a message claim.

7.1 Message queues

An STLite/OS20 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.

The movement of messages between the two queues is illustrated in Figure 7.1.

![Figure 7.1 Message queues](image-url)
7.2 Creating message queues

Message queues are created using one of the following functions:

```c
#include <message.h>
message_queue_t* message_create_queue(
    size_t MaxMessageSize,
    unsigned int MaxMessages);

#include <message.h>
void message_init_queue (
    message_queue_t* MessageQueue,
    void* memory,
    size_t MaxMessageSize,
    unsigned int MaxMessages);
```

or by using timeout versions of the above functions:

```c
#include <message.h>
message_queue_t* message_create_queue_timeout(
    size_t MaxMessageSize,
    unsigned int MaxMessages);

#include <message.h>
void message_init_queue_timeout(
    message_queue_t* MessageQueue,
    void* memory,
    size_t MaxMessageSize,
    unsigned int MaxMessages);
```

These functions create a message queue for a fixed number of fixed sized messages, each message being preceded by a header, see Figure 7.2. The user must specify the maximum size for a message element and the total number of elements required.

![Figure 7.2 STLite/OS20 message elements](image)

message_create_queue and message_create_queue_timeout allocates the memory for the queue automatically from the system partition.

message_init_queue and message_init_queue_timeout requires the user to allocate the memory for the message queue. This needs to be large enough for storing all the messages (rounded up to the nearest word size) plus a header, for each message.

The total amount of memory needed (in bytes) can be calculated using the macro:

```
MESSAGE_MEMSIZE_QUEUE(maxMessageSize, maxMessages)
```

where maxMessageSize is the size of the message, and maxMessages is the number of messages.
As long as both of the parameters can be determined at compile time, this macro can be completely evaluated at compile time, and so can be used as the dimension of an array, for example:

```c
typedef struct {
    int tag;
    char msg[10];
} msg_t;
#define NUM_MSG 10
char msg_buffer[MESSAGE_MEMSIZE_QUEUE(sizeof(msg_t), NUM_MSG)];
```

Alternatively this can be done by calling the function `memory_allocate`. This function will return a pointer to the allocated memory, which should be passed to `message_init_queue` or `message_init_queue_timeout` as the parameter `MessageQueue`.

**Note** that these functions cannot use the hardware support for semaphores, and so are larger and slower than the non-timeout versions.

**Example**

```c
#include <message.h>
#include <partitio.h>
#define MSG_SIZE 512
#define MAX_MSGS 10
#define QUEUE_SIZE MESSAGE_MEMSIZE_QUEUE(MSG_SIZE,MAX_MSGS)
#define EXIT_SUCCESS 0
#define EXIT_FAILURE -1

int myqueue_create(void)
{
    void *msg_queue;
    message_queue_t *msg_queue_struct;

    /* allocate memory for message queue itself */
    msg_queue = memory_allocate(system_partition,QUEUE_SIZE);
    if (msg_queue == 0)
    {
        return(EXIT_FAILURE);
    }

    /* allocate memory for message struct which holds details of queue */
    msg_queue_struct = memory_allocate(system_partition,sizeof(message_queue_t));
    if (msg_queue_struct == 0)
    {
        memory_deallocate(system_partition,msg_queue);
        return(EXIT_FAILURE);
    }

    message_init_queue(msg_queue_struct,msg_queue,MSG_SIZE,MAX_MSGS);
    return(EXIT_SUCCESS);
}
```
### 7.3 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:

```c
void* message_claim( message_queue_t* queue);
void* message_claim_timeout( message_queue_t* queue; const clock_t* time);
```

Both functions claim the next available message in the message queue. `message_claim_timeout` enables a timeout to be specified but can only be used if the message queue was created with a timeout capability. If the timeout is reached before a message buffer is acquired then the function will return 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 will continue.
- **TIMEOUT_INFINITY** causes the function to behave the same as `message_claim`, i.e. the task will wait 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:

```c
void* message_receive( message_queue_t* queue);
void* message_receive_timeout( message_queue_t* queue; const clock_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 message does not become available. If `TIMEOUT_IMMEDIATE` is specified the task will continue whether or not a message is received and if `TIMEOUT_INFINITY` is specified the function will behave as `message_receive` and wait indefinitely.

Finally when the receiving task has finished with the message buffer it should free it by calling `message_release()`, which will add 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` or `message_create_queue_timeout` then this will also free the memory allocated for the message queue. If it was created using `message_init_queue` or
message_init_queue_timeout then the user is responsible for freeing any memory which was allocated for the queue.

### 7.4 Message header file: message.h

All the definitions related to interrupts are in the single header file, `message.h`, see Table 7.1 and Table 7.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message_claim</td>
<td>Claim a message buffer</td>
</tr>
<tr>
<td>message_claim_timeout</td>
<td>Claim a message buffer with timeout</td>
</tr>
<tr>
<td>message_create_queue</td>
<td>Create a fixed size message queue</td>
</tr>
<tr>
<td>message_create_queue_timeout</td>
<td>Create a fixed size message queue with timeout</td>
</tr>
<tr>
<td>message_delete_queue</td>
<td>Delete a message queue</td>
</tr>
<tr>
<td>message_init_queue</td>
<td>Initialize a fixed size message queue</td>
</tr>
<tr>
<td>message_init_queue_timeout</td>
<td>Initialize a fixed size message queue with timeout</td>
</tr>
<tr>
<td>message_receive</td>
<td>Receive the next available message from a queue</td>
</tr>
<tr>
<td>message_receive_timeout</td>
<td>Receive the next available message from a queue or timeout</td>
</tr>
<tr>
<td>message_release</td>
<td>Release a message buffer</td>
</tr>
<tr>
<td>message_send</td>
<td>Send a message to a queue</td>
</tr>
</tbody>
</table>

Table 7.1 Functions defined in `message.h`

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message_hdr_t</td>
<td>A message buffer header</td>
</tr>
<tr>
<td>message_queue_t</td>
<td>A message queue</td>
</tr>
</tbody>
</table>

Table 7.2 Types defined in `message.h`

All functions are callable from an STLite/OS20 task. Functions in Table 7.3 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor. Both the type of object which is being waited on, as well as the priority of the task affect whether the message function can be called from an ISR or HPP.

<table>
<thead>
<tr>
<th>Function</th>
<th>Object</th>
<th>Task</th>
<th>ISR</th>
<th>HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>message_claim</td>
<td>Non-timeout queue</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>message_send</td>
<td>Non-timeout queue</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>message_receive</td>
<td>Non-timeout queue</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>message_release</td>
<td>Non-timeout queue</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>message_claim_timeout</td>
<td>Non-timeout queue</td>
<td>Yes (1)</td>
<td>No</td>
<td>Yes (1)</td>
</tr>
<tr>
<td>message_receive_timeout</td>
<td>Non-timeout queue</td>
<td>Yes (1)</td>
<td>No</td>
<td>Yes (1)</td>
</tr>
</tbody>
</table>

Table 7.3 Message functions callable from an ISR or HPP
7.4 Message header file: message.h

<table>
<thead>
<tr>
<th>Function</th>
<th>Object</th>
<th>Task</th>
<th>ISR</th>
<th>HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>message_claim</td>
<td>Timeout queue</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>message_send</td>
<td>Timeout queue</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>message_receive</td>
<td>Timeout queue</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>message_release</td>
<td>Timeout queue</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>message_claim_timeout</td>
<td>Timeout queue</td>
<td>Yes</td>
<td>Yes (2)</td>
<td>Yes (2)</td>
</tr>
<tr>
<td>message_receive_timeout</td>
<td>Timeout queue</td>
<td>Yes</td>
<td>Yes (2)</td>
<td>Yes (2)</td>
</tr>
</tbody>
</table>

Table 7.3 Message functions callable from an ISR or HPP

Notes:

1. Timeout value is ignored. Behaves as though TIMEOUT_INFINITY was specified.
2. Can only be used with TIMEOUT_IMMEDIATE.
7 Message handling

7.5 Message handling function definitions

**message_claim**

Claim a message buffer.

**Synopsis:**

```c
#include <message.h>

void* message_claim(message_queue_t* queue);
```

**Arguments:**

- `message_queue_t* queue`
  
  The message queue from which the message is claimed.

**Results:**

The next available message buffer.

**Errors:**

None.

**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 will block until one becomes available (by another task calling `message_release()`).

**See also:**

`message_receive message_send message_release`
message_claim_timeout  
Claim a message buffer or timeout.

Synopsis:

```
#include <message.h>
#include <ostime.h>

void* message_claim_timeout(
    message_queue_t* queue
    const clock_t* time);
```

Arguments:

- `message_queue_t* queue`  
The message queue from which the message is claimed.
- `const clock_t* time`  
The maximum time to wait for a message.

Results:

The next available message buffer, or NULL if a timeout occurs.

Errors:

None.

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 will block until one becomes available (by another task calling message_release()), or the time specified by time is reached.

Note that 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 may be specified in ticks, which is an implementation dependent quantity.

Two special time values may also be specified for time. TIMEOUT_IMMEDIATE will cause the message queue to be polled, that is, the function will always return immediately. If a message was available then it will be returned, otherwise the function will return immediately with a result of NULL. A timeout of TIMEOUT_INFINITY will behave exactly as message_claim.

message_claim_timeout may be used from an interrupt handler, as long as time is TIMEOUT_IMMEDIATE.

Example:

```
clock_t time;
time = time_plus(time_now(), 15625);
message_claim_timeout(message_queue, &time);
```

See also:

message_receive_timeout  message_send  message_release
Message handling

message_create_queue Create a fixed size message queue.

Synopsis:

```c
#include <message.h>

message_queue_t* message_create_queue(
    size_t MaxMessageSize,
    unsigned int MaxMessages);
```

Arguments:

- `size_t MaxMessageSize` The maximum size of a message, in bytes.
- `unsigned int MaxMessages` The maximum number of messages.

Results:

The message queue identifier, or NULL on failure.

Errors:

Returns NULL if there is insufficient memory for the message queue.

Description:

Create 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 system memory partition.

See also:

- `memory_allocate`
- `message_claim`
- `message_send`
- `message_delete_queue`
- `message_receive`
- `message_release`
message_create_queue_timeout Create a fixed size message queue with timeout capability.

Synopsis:

```c
#include <message.h>

message_queue_t* message_create_queue_timeout(
    size_t MaxMessageSize,
    unsigned int MaxMessages);
```

Arguments:

- `size_t MaxMessageSize` The maximum size of a message, in bytes.
- `unsigned int MaxMessages` The maximum number of message elements.

Results:

The message queue identifier, or NULL on failure.

Errors:

Returns NULL if there is insufficient memory for the message queue.

Description:

Create 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 system memory partition.

See also:

- `memory_allocate`
- `message_claim_timeout`
- `message_send`
- `message_delete_queue`
- `message_receive_timeout`
- `message_release`
message_delete_queue

Delete a message queue.

Synopsis:

```
#include <message.h>

void message_delete_queue(message_queue_t* MessageQueue);
```

Arguments:

- `message_queue_t* MessageQueue` The message queue to be deleted

Results:

None.

Errors:

None.

Description:

This function allows a message queue to be deleted. If the message queue was
created using `message_create_queue` or `message_create_queue_timeout`
then this will also free the memory allocated for the message queue. If it was
created using `message_init_queue` or `message_init_queue_timeout` then
the user is responsible for freeing any memory which was allocated for the queue.

**Note:** that if any tasks are waiting on the message queue when it is deleted, this will
cause the following fatal error to be reported:

```
delete handler- operation on deleted object attempted
```

Similarly any attempt to use the deleted message queue will report the same error.

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_timeout`
- `message_init_queue`
- `message_init_queue_timeout`
message_init_queue Initialize a fixed size message queue.

Synopsis:

```
#include <message.h>

void message_init_queue(
    message_queue_t* MessageQueue,
    void* memory,
    size_t MaxMessageSize,
    unsigned int MaxMessages);
```

Arguments:

- `message_queue_t* MessageQueue` The message queue to be initialized.
- `void* memory` The memory which will hold the messages.
- `size_t MaxMessageSize` The maximum size of a message, in bytes.
- `unsigned int MaxMessages` The maximum number of messages.

Results:
None.

Errors:
None.

Description:
Initialize a message queue with buffering for a fixed number of fixed size messages. Buffer space for the messages must be allocated by the user, and passed to the function as the `memory` parameter. This needs to be large enough for storing all the messages (rounded up to the nearest word size) plus a header, for each message, see Figure 7.2.

The total size of `memory` (in bytes) can be calculated using the macro:

```
MESSAGE_MEMSIZE_QUEUE(MaxMessageSize, MaxMessages)
```

where `MaxMessageSize` is the size of the message, and `MaxMessages` is the number of messages.

See also:

- message_create_queue
- message_claim
- message_delete_queue
- message_send
- message_receive
- message_release
message_init_queue_timeout

Initialize a fixed size message queue with timeout capability.

Synopsis:

#include <message.h>

void message_init_queue_timeout(
    message_queue_t* MessageQueue,
    void* memory,
    size_t MaxMessageSize,
    unsigned int MaxMessages);

Arguments:

message_queue_t* MessageQueue The message queue to be initialized.
void* memory The memory which will hold the messages
size_t MaxMessageSize The maximum size of a message, in bytes.
unsigned int MaxMessages The maximum number of messages.

Results:

None.

Errors:

None.

Description:

Initialize a message queue with buffering for a fixed number of fixed size messages. Buffer space for the messages must be allocated by the user, and passed to the function as the memory parameter. This needs to be large enough for storing all the messages (rounded up to the nearest word size) plus a header, for each message, see Figure 7.2.

The total size of memory (in bytes) can be calculated using the macro:

MESSAGE_MEMSIZE_QUEUE(MaxMessageSize, MaxMessages)

where MaxMessageSize is the size of the message, and MaxMessages is the number of messages.

See also:

message_create_queue message_claim_timeout
message_delete_queue message_send message_receive_timeout
message_release
message_receive    Receive the next available message from a queue.

Synopsis:

    #include <message.h>
    void* message_receive(message_queue_t* queue);

Arguments:

message_queue_t* queue            The message queue that delivers the message.

Results:

    The next available message from the queue.

Errors:

    None.

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 will block 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.

Synopsis:

```c
#include <message.h>
#include <ostime.h>

void* message_receive_timeout(message_queue_t* queue
    const clock_t* time);
```

Arguments:

- `message_queue_t* queue` — The message queue that delivers the message.
- `const clock_t* time` — The maximum time to wait for a message.

Results:

The next available message from the queue, or NULL if a timeout occurs.

Errors:

None.

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 will block until one becomes available (by another task calling `message_send()`), or the time specified by `time` is reached.

Note that `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, which is an implementation dependent quantity.

Two special time values may also be specified for `time`. `TIMEOUT_IMMEDIATE` will cause the message queue to be polled, that is, the function will always return immediately. If a message was available then it will be returned, otherwise the function will return immediately with a result of NULL. A timeout of `TIMEOUT_INFINITY` will behave exactly as `message_receive`.

Example:

```c
clock_t time;
time = time_plus(time_now(), 15625);
message_receive_timeout(message_queue, &time);
```

See also:

`message_claim` `message_receive` `message_release` `message_send`
7.5 Message handling function definitions

message_release

Release a message buffer.

Synopsis:

```c
#include <message.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.

Results:

None.

Errors:

None.

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 will cause 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.

Synopsis:

```c
#include <message.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.

Results:

None.

Errors:

None.

Description:

`message_send()` sends the specified message to the message queue. This will add the message to the end of the queue of sent messages, and if any tasks are waiting for a message they will be rescheduled and the message returned.

See also:

- `message_claim`
- `message_receive`
- `message_release`
7.5 Message handling function definitions
8 Real-time clocks

Time is very important for real-time systems. STLite/OS20 provides some basic functions for manipulating quantities of time:

The ST20 traditionally regards time as circular. That is, the counters which represent time can wrap round, with half the time period being in the future, and half of it in the past. This behavior means that clock values should only be manipulated using time functions. STLite/OS20 provides functions to:

- Add and subtract quantities of time.
- Determine if one time is after another.
- Return the current time.

8.1 ST20-C1 clock peripheral

The ST20-C1 microprocessor does not have its own clock so a clock peripheral is required when using STLite/OS20.

A number of functions are required to support the clock functions provided by STLite/OS20, see Chapter 12. STLite/OS20 provides the sources for a number of library functions to support a clock peripheral running on an ST20-MC2 device. These functions may be copied and modified to support other clock peripherals, see Appendix A.

Note the STLite/OS20 kernel and interrupt controller must be initialized (as described in Chapter 12) before the clock peripheral is initialized.

8.2 The ST20 timers on the ST20-C2

An ST20-C2 processor has two on-chip real-time 32-bit clocks, called timers, one with low resolution and one with high resolution. The following details are relevant for some ST20-C2 devices. You should check the figures given in the device datasheet as the timing values vary with different processor revisions.

The low resolution clock can be used for timing periods up to approximately 38 hours, with a resolution of 64 µsec. The low resolution clock is accessed by low priority tasks. The high resolution clock can be used for timing periods up to approximately half an hour with a resolution of 1 µsec. The high resolution clock is accessed by high priority tasks. Longer periods can be timed with either timer by explicitly incrementing a counter.

The clocks start at an undefined value and wrap round to 0 on the next tick after hexadecimal FFFFFFFF, or, if treated as signed, to the most negative integer on the next tick after the most positive integer. The tick rate of the clocks is derived from the processor input ClockIn, and the speed and accuracy depends on the speed and accuracy of the input clock.
8.3 Reading the current time

For ST20 variants with power-down capability, the clocks pause when the ST20 is in power-down mode.

<table>
<thead>
<tr>
<th></th>
<th>Low priority</th>
<th>High priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval between ticks</td>
<td>64 µsec</td>
<td>1 µsec</td>
</tr>
<tr>
<td>Ticks per second</td>
<td>15625</td>
<td>1000000</td>
</tr>
<tr>
<td>Approximate full timer cycle</td>
<td>76.35 hours</td>
<td>1.193 hours</td>
</tr>
</tbody>
</table>

Table 8.1 Summary of clock intervals for parts operating at 40Mhz

8.3 Reading the current time

The value of a timer (or clock) is read using `time_now` which returns the value of the timer for the current priority.

```c
#include <ostime.h>

clock_t time_now (void);
```

The time at which counting starts will be no later than the call to `kernel_start`.

8.4 Time arithmetic

Arithmetic on timer values should always be performed using special modulo operators. 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.

```c
clock_t time_plus(const clock_t time1, const clock_t time2);
clock_t time_minus(const clock_t time1, const clock_t time2);
int time_after(const clock_t time1, const clock_t time2);
```

time_plus adds two timer values together and returns the sum allowing for any wrap-around. For example, if a number of ticks is added to the current time using time_plus then the result is the time after that many ticks.

time_after subtracts the second value from the first and returns the difference allowing for any wrap-around. For example, if one time is subtracted from another using time_after then the result will be 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. One time is considered to be after another if the one is not more than half a full timer cycle later than the other. Half a full cycle is $2^{31}$ ticks. The function returns the integer value 1 if the first time is after the second, and otherwise it returns zero.

Some of these concepts are shown in Figure 8.1.
Figure 8.1 Time arithmetic

Time arithmetic is modulo $2^{32}$. In applications running for a long time, some care must be taken to ensure that times are close enough together for arithmetic to be meaningful. For example, subtracting two times which are more than $2^{31}$ ticks apart will produce a result that must be interpreted with care. Very long intervals can be tracked by counting a number of cycles of the clock.
8.5 Time header file: ostime.h

All the definitions related to time are in the single header file, ostime.h, see Table 8.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/ HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_after</td>
<td>Return whether one time is after another</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>time_minus</td>
<td>Subtract two clock values</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>time_now</td>
<td>Return the current time</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>time_plus</td>
<td>Add two clock values</td>
<td>ISR and HPP</td>
</tr>
</tbody>
</table>

Table 8.2 Functions defined in ostime.h

All functions are callable from an STLite/OS20 task. Functions in Table 8.2 which are marked with 'ISR' can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor.

Table 8.3 lists the types defined by ostime.h.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_t</td>
<td>Number of processor clock ticks</td>
</tr>
</tbody>
</table>

Table 8.3 Types defined by ostime.h
8.6 STLite/OS20 time function definitions

**time_after**  
Return whether one time is after another.

**Synopsis:**
```c
#include <ostime.h>
int time_after(const clock_t time1, const clock_t time2);
```

**Arguments:**
- `const clock_t time1`  
  A clock value returned by `time_now`.  
- `const clock_t time2`  
  A clock value returned by `time_now`.

**Results:**
Returns 1 if `time1` is after `time2`, otherwise 0.

**Errors:**
None.

**Description:**
Returns the relationship between `time1` and `time2`. Time values are cyclic, so `time1` may be numerically less than `time2`, but still represent a later time, if the difference is larger than half of the complete time period.

**See also:**
- `time_minus`  
- `time_now`
**time_minus**

Subtract two clock values.

**Synopsis:**

```c
#include <ostime.h>

clock_t time_minus(const clock_t time1, const clock_t time2);
```

**Arguments:**

- `const clock_t time1` A clock value returned by `time_now`.
- `const clock_t time2` A clock value returned by `time_now`.

**Results:**

Returns the result of subtracting `time2` from `time1`.

**Errors:**

None.

**Description:**

Subtracts one clock value from another using modulo arithmetic. No overflow checking takes place because the clock values are cyclic.

**See also:**

`time_plus`
time_now

Return the current time.

Synopsis:

```c
#include <ostime.h>
clock_t time_now(void);
```

Arguments:
None.

Results:
Returns the number of ticks since the system started.

Errors:
None.

Description:
```
time_now() returns the number of ticks since the system started running. The exact time at which counting starts is implementation specific, but will be no later than the call to kernel_start.

The units of ticks is an implementation dependent quantity, see section 8.1 and section 8.2.
```

See also:
```
task_delay
```
time_plus

Add two clock values.

Synopsis:

```c
#include <ostime.h>
clock_t time_plus(const clock_t timel, const clock_t time2);
```

Arguments:

- `const clock_t timel` A clock value returned by `time_now`.
- `const clock_t time2` A clock value returned by `time_now`.

Results:

Returns the result of adding `timel` to `time2`.

Errors:

None.

Description:

Adds one clock value to another using modulo arithmetic. No overflow checking takes place because the clock values are cyclic.

See also:

`time_minus`
9 Interrupts

Interrupts provide a way for external events to control the CPU. Normally, as soon as an interrupt is asserted, the CPU will stop executing the current task, and start executing the interrupt handler for that interrupt. In this way the program can be made aware of external changes as soon as they occur. This switch is performed completely in hardware, and so can be extremely rapid. Similarly when the interrupt handler has completed, the CPU will resume execution of the interrupted task, which will be 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 STLite/OS20, and simply sets up the environment so that a normal C function can be called. The STLite/OS20 API allows a different user function to be associated with each interrupt, and this will be called when the interrupt occurs. Each interrupt also has a parameter associated with it, which will be passed into the function when it is called. This could be used to allow the same code to be shared between different interrupt handlers.

9.1 Interrupt models

The interrupt hardware on different ST20 processors is similar, but there are a number of variations.

The basic hardware unit is called the interrupt controller. This receives the interrupt signals, and alerts the CPU when interrupts go active. Interrupts can be programmed to be active when high, or low, or on a rising, falling or both edges of the signal, this is called the trigger mode by STLite/OS20.

On some processors, interrupt sources are connected directly to the interrupt controller, similar to the example shown in Figure 9.1.

![Diagram of interrupt controller](image)

Figure 9.1 Example: peripherals directly attached to the interrupt controller
The relative priority of the interrupts is defined by the *interrupt level*, with numerically higher interrupts interrupting numerically lower priority interrupts. Thus, an interrupt level 3 will interrupt an interrupt level 2 which will interrupt an interrupt level 1. As the connection between the peripheral and the interrupt controller is fixed when the device is designed, so is the relative priority of the peripheral’s interrupts.

Some ST20 processors have a second piece of interrupt hardware, called the interrupt level controller, see the example in Figure 9.2. This allows the relative priority of different interrupt sources to be changed. Each peripheral generates an *interrupt number*, which is fixed for the peripheral. This is fed into the interrupt level controller, which selects for each interrupt number which *interrupt level* should be generated. The interrupt level controller can be programmed to select the interrupt level which each interrupt number generates, thus allowing the relative priorities to be changed in software. As there are generally more interrupt numbers than interrupt levels, it is possible to multiplex several interrupt numbers onto a single interrupt level.

An important distinction is that interrupt levels are prioritized, numerically higher interrupt levels preempt lower ones, however there is no order between interrupt numbers.

![Figure 9.2 Example: peripherals mapped via an interrupt level controller](image)

There are two types of interrupt controller for ST20 processors: IntC-1 and IntC-2. Both interrupt controller provide the same services but the IntC-2 has a register layout that makes it capable of supporting more interrupt levels in the future, see Table 9.1.

There are three types of interrupt level controller for ST20 processors: ILC-1, ILC-2 and ILC-3.

ILC-1 type interrupt level controllers support up to 32 interrupt numbers and the trigger mode logic is part of the interrupt controller. All interrupt numbers attached to the same level share the same trigger mode.
ILC-2 type interrupt level controllers support up to 32 interrupt numbers but there is support for programmable trigger modes and an enable and disable facility for all interrupt numbers.

ILC-3 type interrupt level controllers currently supports up to 128 interrupt numbers, each of which can have a programmable trigger mode and enable status.

STLite/OS20 functions provide support for all ST20 interrupt models.

### 9.2 Selecting the correct interrupt handling system

STLite/OS20 contains two libraries to support different interrupt controller combinations:

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>os20intc1.lib</td>
<td>IntC-1</td>
<td>ST20GP6, ST20MC2, ST20TP3, ST20TP4, STi5500, STi5505, STi5508, STi5510, STi5512, STi5518, ST20-C1 simulator, ST20-C2 simulator</td>
</tr>
<tr>
<td>os20intc2.lib</td>
<td>IntC-2</td>
<td>ST20DC1</td>
</tr>
</tbody>
</table>

Table 9.1 Interrupt controller libraries

Additionally STLite/OS20 contains four libraries to support different interrupt level controller combinations:

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>os20ilcnone.lib</td>
<td>ILC-None</td>
<td>ST20-C1 simulator, ST20-C2 simulator.</td>
</tr>
<tr>
<td>os20ilc1.lib</td>
<td>ILC-1</td>
<td>ST20DC1, ST20GP6, ST20MC2, ST20TP3, ST20TP4, STi5500, STi5505, STi5508, STi5510, STi5512</td>
</tr>
<tr>
<td>os20ilc2.lib</td>
<td>ILC-2</td>
<td>STi5518.</td>
</tr>
<tr>
<td>os20ilc3.lib</td>
<td>ILC-3</td>
<td>None at present.</td>
</tr>
</tbody>
</table>

Table 9.2 Interrupt level controller libraries

In order for STLite/OS20 to operate properly the correct libraries must be linked in. When using the `st20cc -runtime os20` option, the linker needs to select the appropriate IntC and ILC libraries. When using the `chip` command, the correct libraries will always be selected. If the `chip` command is not used then IntC-1 and ILC-1 libraries will be used to preserve backward compatibility.

In addition to providing support for the ILC-1, the ILC-1 library can support systems without an interrupt level controller and systems that have an ILC-2. In both these cases the support is not optimal. The ILC-None library uses much less RAM than its ILC-1 counterpart. The ILC-2 library supports the extra features the ILC-2 provides.

Note that the interrupt function definitions given in this chapter, list the interrupt level controllers they can be used with. If a function is used which is not applicable to the interrupt level controller on the device used then that function will not be provided and the application will fail at link time. This can cause link errors if the interrupt calls are used inappropriately. There are no warnings issued at compile time.
9.2 Selecting the correct interrupt handling system

9.2.1 Compiling legacy code

The IntC-1 and IntC-2 libraries provide an identical set of function calls. There are no problems compiling code for either interrupt controller.

On ILC-2 and ILC-3 interrupt level controllers new function calls have been introduced to provide support for the newer features of these controllers. This may cause problems when reusing existing code. In particular be aware that calls to `interrupt_enable` and `interrupt_disable` should be replaced with calls to `interrupt_enable_number` and `interrupt_disable_number`. Code that does not do this will compile and link cleanly but interrupts will never be serviced because they are not enabled. Table 9.3 describes the migration path between ILCs.

<table>
<thead>
<tr>
<th>ILC library</th>
<th>Legacy code</th>
<th>Recommended replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILC-1</td>
<td><code>interrupt_enable (INTERRUPT_GLOBAL_ENABLE)</code></td>
<td><code>interrupt_enable_global()</code> †</td>
</tr>
<tr>
<td></td>
<td><code>interrupt_disable (INTERRUPT_GLOBAL_DISABLE)</code></td>
<td><code>interrupt_disable_global()</code> †</td>
</tr>
<tr>
<td></td>
<td><code>interrupt_pending_number</code></td>
<td><code>interrupt_test_number</code> †</td>
</tr>
<tr>
<td>ILC-2</td>
<td>All changes recommended for ILC-1 plus:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>interrupt_enable</code></td>
<td><code>interrupt_enable_number</code> ‡ (May require multiple calls.)</td>
</tr>
<tr>
<td></td>
<td><code>interrupt_disable</code></td>
<td><code>interrupt_disable_number</code> ‡ (May require multiple calls.)</td>
</tr>
<tr>
<td>ILC-3</td>
<td>All changes recommended for ILC-2 plus:</td>
<td>Use of chip command and <code>st20cc -runtime os20</code> mandatory. ‡</td>
</tr>
<tr>
<td></td>
<td><code>interrupt_pending_number</code></td>
<td><code>interrupt_test_number</code> ‡</td>
</tr>
</tbody>
</table>

† This change is optional and will make code easier to port in the future.
‡ This change is mandatory on the ILC-3 and the default ILC-2 library. See also the Note below.

Table 9.3 Migration path for ILCs

Note: that at reset the ILC-2 hardware is configured to be backwards compatible with the ILC-1. The fastest way to bring old code up on these chips is to link in the ILC-1 library instead of the ILC-2 support. An STLite/OS20 configuration option is provided to support this when `st20cc -runtime os20` is used. See section C.1.4.

9.2.2 Linking legacy code

The only method of linking that is recommended is to use the command `st20cc -runtime os20` in conjunction with the application using the chip command. New programs should all follow this methodology.

Linking STLite/OS20 applications using `st20cc -Tos20.cfg` is not recommended. For back compatibility, if it is used, the IntC-1 and ILC-1 support libraries are linked in. Do not write new code using this option.

Linking `os20.lib` directly is also no longer recommended. `os20.lib` does not contain any of the interrupt functions. To link legacy code `os20.lib` should be replaced with `os20.lib` followed by the correct combination of interrupt controller
and interrupt level controller libraries (see Table 9.1 and Table 9.2 above). Do not write new code using this option.

If porting legacy code to an ST20 variant with an ILC-3 it is not possible to link os20ilc3.lib in the manner described above. ILC-3 type interrupt level controllers require the use of st20cc -runtime os20 and the chip command.

9.2.3 Required memory partitions (ILC-3 only)

The ILC-3 type interrupt library assumes that there is a memory partition called EXTERNAL from which it can reserve memory. If there is no such partition or the ILC data structures are not required to be placed in external memory then this behavior can be altered by defining OS20_config.use_memory_segment in a linker configuration file.

Example:

```
st20cc -p link_proc -runtime os20 ...

proc link_proc {
    chip ST20TT1
    memory MY_SEGMENT 0x40000000 (1*1024*1024) RAM
    OS20_config.use_memory_segment=“MY_SEGMENT”
    ...
}
```

9.3 Initializing the interrupt handling support system

Before any interrupt handling routines are written the interrupt hardware needs to be configured and initialized in order that STLite/OS20 knows which hardware model is being targeted.

Both the interrupt controller and interrupt level controller have a number of configuration registers which must be correctly programmed before the peripheral can assert an interrupt signal. This also varies for each device and typically will include setting the Mask register to enable/disable individual interrupts (see section 9.6) and the TriggerMode register, see below.

The interrupt_init_controller function enables you to specify how the interrupt controller and interrupt level controller (if present) are configured.

```
#include <interrup.h>
void interrupt_init_controller(
    void* interrupt_controller,
    int interrupt_levels,
    void* level_controller,
    int interrupt_numbers,
    int input_offset);
```

The base address and number of inputs supported by the interrupt controller and (if applicable) the interrupt level controller, on the target ST20 device must be specified. These details are device specific and can be obtained from the device datasheet.

Normally if st20cc -runtime os20 is used when linking, then this will be performed automatically before the user’s application starts to run.
9.4 Attaching an interrupt handler in STLite/OS20

Next each interrupt level must be initialized. The `interrupt_init()` function is used to initialize a single interrupt level in the interrupt controller:

```c
#include <interrupt.h>
int interrupt_init(
    int interrupt,
    void* stack_base,
    size_t stack_size,
    interrupt_trigger_mode_t trigger_mode,
    interrupt_flags_t flags);
```

This function enables an area of stack to be defined and also specifies the trigger mode associated with an interrupt level i.e. whether the interrupt is active when the signal is high, or low, or on a rising, falling or both edges of the signal. The stack is used to execute all interrupt handlers attached at that level so must be large enough to accommodate the largest interrupt handler.

9.3.1 Calculating stack size

The area of stack must be large enough for each interrupt handler to execute within. It must accommodate all the local variables declared within a handler and must take account of any further function calls that the handler may make. **Note:** the following:

As a general rule an interrupt handler uses the following workspace:

- Eight words of save state.
- Five words for internal pointers etc. for ILC-None or ILC-1 interrupt libraries, seven words for ILC-2 and eight words for ILC-3.
- Space for the user’s initial stack frame (four words on an ST20-C2, three words on an ST20-C1).
- Then recursively:
  - Space for local variables declared in the function, (add up the number of words).
  - Space for calls to extra functions. For a library function allow 150 words for worst case.

For details of data representation, see the ‘ST20 Embedded Toolset User Manual - 72-TDS-505’.

9.4 Attaching an interrupt handler in STLite/OS20

An interrupt handler is attached to an interrupt, using the `interrupt_install()` function:

```c
#include <interrupt.h>
int interrupt_install (  
    int Number,  
    int Level,  
    void(*Handler)(void* Param),  
    void* Param);
```
Once the interrupt handler is attached the interrupt should be enabled by calling `interrupt_enable` or `interrupt_enable_number` as described in section 9.6. The function `interrupt_install_sl` enables an interrupt to be installed and the static link specified. It is intended for use with relocatable code units.

9.4.1 Attaching interrupt handlers directly to peripherals

If there is no interrupt level controller on the ST20, then only one handler can be attached to each interrupt level (and the interrupt number specified to the `interrupt_install` function must be specified as -1). `interrupt_install` will then associate the specified interrupt handler with a particular interrupt level.

Example

```c
#include <interrup.h>
int interrupt_stack[500];
void interrupt_handler(void* param);

int intrpt_stack[500];
void intrpt_handler(void* param);

interrupt_init(4, interrupt_stack, sizeof(interrupt_stack),
               interrupt_trigger_mode_rising, 0);
interrupt_install(-1, 4, interrupt_handler, NULL);

interrupt_init(2, intrpt_stack, sizeof(intrpt_stack),
               interrupt_trigger_mode_low_level, 0);
interrupt_install(-1, 2, intrpt_handler, NULL);
interrupt_enable(2);
interrupt_enable(4);
```

9.4.2 Attaching interrupt handlers using an interrupt level controller

On devices which have an interrupt level controller, then multiple handlers can be attached to each level, one for each interrupt number. The act of attaching the interrupt handler at a level will result in the interrupt controller being programmed to generate the chosen interrupt level. When an interrupt occurs at an interrupt level which has multiple interrupt numbers attached, STLite/OS20 will arrange to call all the appropriate handlers for interrupts which are pending. To do this it will loop, checking for pending interrupts at the current level, until there are none outstanding. When multiple interrupt numbers are pending, the numerically highest will be called first.
9.4 Attaching an interrupt handler in STLite/OS20

Example:

```c
#include <interrup.h>
int interrupt_stack[500];
void interrupt_handler(void* param);
void intrpt_handler(void* param);

interrupt_init(4, interrupt_stack, sizeof(interrupt_stack),
               interrupt_trigger_mode_rising, 0);
interrupt_install(10, 4, interrupt_handler, NULL);
interrupt_install(3, 4, intrpt_handler, NULL);

/* for ILC-3 type interrupt level controllers
 * interrupt_enable(4) would be replaced by
 * interrupt_enable_number(10)
 * interrupt_enable_number(3)
 */
interrupt_enable(4);
interrupt_enable_global();
```

9.4.3 Routing interrupts to external pins

ILC-3 supports the function of routing interrupts to external pins (called interrupt outputs), where additional hardware can handle the interrupt. Typically this would be used for multi-CPU systems. To set up this mode of operation, `interrupt_install` is used, and the interrupt level is specified as: `(INTERRUPT_EXTERNAL_PIN plus the number of the interrupt output)`. For example:

```c
/* Direct interrupt number 4 to external interrupt output 2 */
interrupt_install(4, INTERRUPT_EXTERNAL_PIN+2, NULL, NULL);
```

9.4.4 Efficient interrupt layouts

STLite/OS20 does not install the interrupt handler supplied to `interrupt_install` as the first level handler. Instead it installs its own optimized interrupt handlers to determine which interrupt number caused that interrupt level to be raised and then sets up the workspace to make C calls. STLite/OS20 picks the best code it can to minimize interrupt latency. Carefully laying out interrupts can assist this.

The most efficient case is when a single interrupt number is attached to an interrupt level, there is little work to be done and every address can be precalculated by `interrupt_install`. Devices that require absolute minimum latency should be attached like this.

For the ILC-1 and ILC-2 there are no further optimizations that can be made.

ILC-3 has more than 32 interrupt numbers. The ST20 is a 32-bit processor and therefore ILC-3 registers cross the word boundary of the machine. When two interrupt numbers attached to the same level are spread across more than one word the work
required to determine the source of the interrupt increases. Thus bunching interrupt numbers between word boundaries will minimize interrupt latency.

Example:

/* good layout (for ILC-3) */
interrupt_install(1, 1, intrpt_handler1, NULL);
interrupt_install(31, 1, intrpt_handler2, NULL);

/* poor layout (crosses word boundary) */
interrupt_install(3, 2, intrpt_handler3, NULL);
interrupt_install(33, 2, intrpt_handler4, NULL);

9.5 Initializing the peripheral device

Each peripheral device has its own interrupt control register(s) which must be programmed in order for the peripheral to assert an interrupt signal. This is device dependent and so will vary between devices, but will usually involve specifying which events should cause an interrupt to be raised. The example in section 9.7 shows a setup for an Asynchronous Serial Controller (ASC). It is important that these device registers are set up after the interrupt controller and interrupt level controller. (Likewise when deleting interrupts it is important that the peripheral device interrupt control register(s) are reprogrammed first, see section 9.15).

9.6 Enabling and disabling interrupts

The following two functions can be used to set or clear the global enables bit INTERRUPT_GLOBAL_ENABLE in the interrupt controller's Set_Mask register:

```
#include <interrup.h>
void interrupt_enable_global();
void interrupt_disable_global();
```

When the global enables bit is set then any enabled interrupt can be asserted. When the global enables bit is not set then no interrupts can be asserted regardless of whether they are individually enabled. These two functions apply to all interrupt controllers.

9.6.1 Enabling and disabling interrupts without an ILC or with ILC-1

The following two functions take an interrupt level and set or clear the corresponding bit in the interrupt controller Set_Mask register:

```
#include <interrup.h>
int interrupt_enable (int Level);
int interrupt_disable (int Level);
```

This can be used to enable or disable the associated interrupt level.

Although the global enables bit can be set or cleared by these functions (as INTERRUPT_GLOBAL_ENABLE) this use is no longer recommended. These functions return -1 if an illegal interrupt level is passed in.
Although both functions work on all existing ST20 processors they are not guaranteed to work for future processors with ILC-2 or ILC-3 interrupt level controllers. Thus their use is only recommended for use on chips with no interrupt level controller or with ILC-1.

The following two functions are similar to those above but take a mask which contains bits to be set or cleared in the interrupt controller Set Mask register depending on the operation being performed.

```c
#include <interrup.h>
void interrupt_enable_mask (int Mask);
void interrupt_disable_mask (int Mask);
```

Like the previous functions the global enables bit can be set or cleared using the mask functions (as 1 << INTERRUPT_GLOBAL_ENABLE) and again it is no longer recommended. Similarly these functions are only recommended for use on chips with no interrupt level controller or with ILC-1.

### 9.6.2 Enabling and disabling interrupts with ILC-2 or ILC-3

The following two functions apply only to ILC-2 or ILC-3 interrupt level controllers and are used to enable and disable interrupt numbers.

```c
#include <interrup.h>
int interrupt_enable_number (int Number);
int interrupt_disable_number (int Number);
```

These functions allow specific interrupt numbers to be enabled and disabled independently by writing to the interrupt level controllers Enable registers.

### 9.7 Example: setting an interrupt for an ASC

This example shows how an interrupt could be set for an Asynchronous Serial Controller on an STi5500 device, this device has an ILC-1 type interrupt level controller. The example demonstrates the steps described in the previous sections to:

- Initialize the interrupt controller, see section 9.3.
- Attach an interrupt handler, see section 9.4.
- Program the peripheral device registers, see section 9.5.
- Enable an interrupt, see section 9.6.

```c
#define INTERRUPT_NUMBERS 18
#define INTERRUPT_INPUT_OFFSET 18
#define INTERRUPT_CONTROLLER 0x20000000
#define INTERRUPT_LEVEL_CONTROLLER 0x20011000
#define ASC0_INTERRUPT_NUMBER 9
#define ASC_INTERRUPT_LEVEL 5

typedef struct {
    int asc_BaudRate;
    int asc_TxBuffer;
    int asc_RxBuffer;
    int asc_Control;
    int asc_IntEnables;
```
int asc_Status;
} asc_t;

volatile asc_t* asc0 = (asc_t*)0x20003000;

#define ASC_MODE_8D 0x01
#define ASC_STOP_1_0 0x08
#define ASC_RUN 0x80
#define ASC_RXEN 0x100

#define ASC_BAUD_9600 (40000000 / (16*9600))
#define ASC_RX_BUF_FULL 1

interrupt_init_controller((void*)INTERRUPT_CONTROLLER, 8,
(void*)INTERRUPT_LEVEL_CONTROLLER,
INTERRUPT_NUMBERS, INTERRUPT_INPUT_OFFSET);

interrupt_init(ASC_INTERRUPT_LEVEL, ser_stack, sizeof(ser_stack),
interrupt_trigger_mode_high_level, 0);

interrupt_enable_global();

if (interrupt_install(ASC0_INTERRUPT_NUMBER, ASC_INTERRUPT_LEVEL,
ser_handler, NULL) == 0) {
    asc->asc_Control = ASC_MODE_8D | ASC_STOP_1_0 | ASC_RUN | ASC_RXEN;
    asc->asc_BaudRate = ASC_BAUD_9600;
    asc->asc_intEnables = ASC_RX_BUF_FUL;
    interrupt_enable(ASC_INTERRUPT_LEVEL);
}

if this example were transferred to a device with an ILC-2 or ILC-3 interrupt level
controller the call to interrupt_enable (last line) would become:-

    interrupt_enable_number(ASC0_INTERRUPT_NUMBER);

In section 9.15 an example is given of how to remove this interrupt.

9.8 Locking out interrupts

All interrupts to the CPU can be globally disabled or re-enabled using the following two
commands:

#include <interrupt.h>

void interrupt_lock (void);
void interrupt_unlock (void);

These functions should always be called as a pair and will prevent any interrupts from
the interrupt controller having any effect on the currently executing task while the lock
is in place. These functions 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 will not be released until an equal number of calls to
interrupt_unlock() have been made.

Note: that locking out interrupts is slightly different from disabling an interrupt. Inter-
rupts are locked by changing the ST20's Enables register, which causes the CPU to
ignore the interrupt controller (and any other external device), while disabling an
interrupt modifies the interrupt controller’s **Mask** register, and so can be used much more selectively. On the ST20-C2 locking interrupts also prevents high priority processes from interrupting and disable channels and timers.

A task must not deschedule with interrupts locked, as this can cause the scheduler to fail.

### 9.9 Raising interrupts

The following functions can be used to force an interrupt to occur:

```c
#include "interrupt.h"
int interrupt_raise (int Level);
int interrupt_raise_number (int Number);
```

The first function will raise the specified interrupt level and should be used when peripherals are attached directly to the interrupt controller. The second function will raise the specified interrupt number and is for use when an interrupt level controller is present.

Note that neither function should be used to raise level sensitive interrupts. They will be immediately cleared by the interrupt hardware.

### 9.10 Retrieving details of pending interrupts

The following functions will return details of pending interrupts:

```c
#include <interrupt.h>
int interrupt_pending (void);
int interrupt_pending_number (void);
int interrupt_test_number (int Number);
```

The first function returns which interrupt levels are pending i.e. those interrupts which have been set by a peripheral, but their interrupt handlers have not yet run. This function should be used when peripherals are attached directly to the interrupt controller. The second function returns which interrupt numbers are pending i.e. all the interrupts which are currently set by peripherals. The ST20 C compiler treats `int` as a 32-bit quantity, thus `interrupt_pending_number` can not be used on ILC-3 type interrupt level controllers because they have too large a quantity of interrupt numbers.

The final function can be used to test if any one specific interrupt number is pending. This function applies to any interrupt level controller because it does not return a mask.

### 9.11 Clearing pending interrupts

The following functions can be used to prevent a raised interrupt signal from causing an interrupt event to occur:

```c
#include "interrupt.h"
int interrupt_clear (int level);
int interrupt_clear_number (int Number);
```
The first function clears the specified pending interrupt level and should be used when peripherals are attached directly to the interrupt controller. The second function will clear the specified interrupt number and is for use when an interrupt level controller is present. If the specified number is the only pending interrupt number attached to the interrupt level then the pending interrupt level is also cleared.

On ILC-1 only interrupts asserted in software by interrupt_raise_number can be cleared in this way.

### 9.12 Changing trigger modes

This section applies only to ST20 variants with ILC-2 or ILC-3. On these devices the following function can be used to change a specific interrupt number’s trigger mode.

```c
#include <interrup.h>
int interrupt_trigger_mode_number(
    int Number,
    interrupt_trigger_mode_t trigger_mode);
```

When interrupt_init is called the user supplies a default trigger mode for all interrupt numbers attached to that interrupt level. When an interrupt is installed then the trigger mode will be set to this default. interrupt_trigger_mode_number can be used to change away from the default behavior set by interrupt_init.

### 9.13 Low power modes and interrupts

This section applies only to ST20 variants with ILC-2 or ILC-3. On these devices the following function can be used to configure which external interrupts can wake the ST20 from low power mode.

```c
#include <interrup.h>
int interrupt_wakeup_number(
    int Number,
    interrupt_trigger_mode_t trigger_mode);
```

Once the ST20 has been placed in low power mode the device can be woken either when its real-time wake-up alarm triggers or when an external interrupt request is asserted. The external request is active high or active low, it cannot be edge triggered.

Note that on some ST20 variants not all external interrupt pins can be used to wake the device from low power mode, exact details can be found from the appropriate device datasheet.

### 9.14 Obtaining information about interrupts

The following two functions can be used to obtain interrupt state information:

```c
#include <interrup.h>
int interrupt_status(int Level, interrupt_status_t* Status,
    interrupt_status_flags_t flags);

int interrupt_status_number(
    int Number,
    interrupt_status_number_t* status,
    interrupt_status_number_flags_t flags);
```
The first function provides information about the state of an interrupt level. This includes the number of interrupt handlers attached to this level and the current state of the interrupt stack, specifically the stack's base, size and peak usage.

The second function provides information about the state of an interrupt number. For standard STLite/OS20 kernels this includes only the interrupt level to which this interrupt number is attached.

There are compile time options that permit extra information to be supplied by interrupt_status and interrupt_status_number but these are disabled for standard STLite/OS20 kernels because they decrease interrupt performance. Refer to section C.3.4 for further details.

9.15 Uninstalling interrupt handlers and deleting interrupts

The following function can be used to uninstall an interrupt handler:

```c
#include <interrup.h>
int interrupt_uninstall(
    int Number,
    int Level);
```

Before interrupt_uninstall is used, the interrupt must be disabled on the actual peripheral device by programming the peripheral's interrupt control register(s) and then using one of the functions: interrupt_disable, interrupt_disable_mask or interrupt_disable_number (see section 9.6).

A replacement trap handler may then be swapped in using interrupt_install or the interrupt may be deleted, using interrupt_delete if it is no longer required. If a replacement trap handler is installed the interrupt must be re-enabled on the peripheral device by programming its interrupt control register(s).

The following function will delete an initialized interrupt, allowing the interrupt handler's stack to be freed:

```c
#include <interrup.h>
int interrupt_delete(int Level);
```

The interrupt must be disabled by programming the peripheral's interrupt control register(s) and uninstalled by calling interrupt_uninstall before interrupt_delete is called.

Example

This example demonstrates how to delete the interrupt set up by the example given in section 9.7.

```c
asc->asc_intEnables = 0;
interrupt_disable(ASC_INTERRUPT_LEVEL);
interrupt_uninstall(ASC0_INTERRUPT_NUMBER, ASC_INTERRUPT_LEVEL);
interrupt_delete(ASC_INTERRUPT_LEVEL);
```
9.16 Restrictions on interrupt handlers

Certain restrictions must be kept in mind when using interrupts on the ST20. These restrictions, and their ramifications for C are as follows:

- Descheduling and timeslicing are automatically disabled for interrupt handlers. Channel communications (on the ST20-C2) and any other descheduling operation are not permitted.
- Interrupt handlers must not use 2D block move instructions unless the existing block move state is explicitly saved and restored by the handler.
- Interrupt handlers must not cause traps.

Care should be taken here to make sure that instruction sequences are not generated which could cause errors and therefore a trap to be taken. It is suggested that in the simplest case an interrupt handler should disable all traps.

9.17 Interrupt header file: `interrupt.h`

All the definitions related to interrupts are in the single header file, `interrupt.h`, see Table 9.4.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>ILC library</th>
<th>Callable from ISR/HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_clear</td>
<td>Clear a pending interrupt.</td>
<td>ILC-None, ILC-1</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_clear_number</td>
<td>Clear a pending interrupt number.</td>
<td>ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_delete</td>
<td>Delete an interrupt handler.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_disable</td>
<td>Disable an interrupt level.</td>
<td>ILC-None, ILC-1</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_disable_global</td>
<td>Global disable interrupts.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_disable_mask</td>
<td>Disable one or more interrupts.</td>
<td>ILC-None, ILC-1</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>interrupt_disable_number</td>
<td>Disable an interrupt number.</td>
<td>ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_enable</td>
<td>Enable an interrupt level.</td>
<td>ILC-None, ILC-1</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_enable_global</td>
<td>Globally enable interrupts.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_enable_mask</td>
<td>Enable one or more interrupts.</td>
<td>ILC-None, ILC-1</td>
<td>ISR, HPP</td>
</tr>
</tbody>
</table>

Table 9.4 Functions defined in `interrupt.h`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>ILC library</th>
<th>Callable from ISR/HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_enable_number</td>
<td>Enable an interrupt number.</td>
<td>ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_init</td>
<td>Initialize an interrupt level.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_init_controller</td>
<td>Initialize the interrupt controller.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_install</td>
<td>Install an interrupt handler.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_install_sl</td>
<td>Install an interrupt handler and specify a static link.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_lock</td>
<td>Lock all interrupts.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_pending</td>
<td>Return pending interrupt levels.</td>
<td>ILC-None, ILC-1</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_pending_number</td>
<td>Return pending interrupt numbers.</td>
<td>ILC-None, ILC-1, ILC-2</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_raise</td>
<td>Raise an interrupt level.</td>
<td>ILC-None, ILC-1</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_raise_number</td>
<td>Raise an interrupt number.</td>
<td>ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_status</td>
<td>Report the status of an interrupt level.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_status_number</td>
<td>Report the status of an interrupt number.</td>
<td>ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_test_number</td>
<td>Test whether an interrupt number is pending.</td>
<td>ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_trigger_mode_number</td>
<td>Change the trigger mode of an interrupt number.</td>
<td>ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_uninstall</td>
<td>Uninstall an interrupt handler.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td></td>
</tr>
<tr>
<td>interrupt_unlock</td>
<td>Unlock all interrupts.</td>
<td>ILC-None, ILC-1, ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
<tr>
<td>interrupt_wakeup_number</td>
<td>Set wakeup status of an interrupt number.</td>
<td>ILC-2, ILC-3</td>
<td>ISR, HPP</td>
</tr>
</tbody>
</table>

Table 9.4 Functions defined in interrup.h

All functions are callable from an STLite/OS20 task. Functions in Table 9.4 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor. The full ILC library names are given in Table 9.2.
Types and macros defined to support interrupts are listed in Table 9.5 and Table 9.6.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_flags_t</td>
<td>Additional flags for interrupt_init.</td>
</tr>
<tr>
<td>interrupt_status_t</td>
<td>Structure describing the status of an interrupt level.</td>
</tr>
<tr>
<td>interrupt_status_flags_t</td>
<td>Additional flags for interrupt_status.</td>
</tr>
<tr>
<td>interrupt_status_number_t</td>
<td>Structure describing the status of an interrupt number.</td>
</tr>
<tr>
<td>interrupt_status_number_flags_t</td>
<td>Additional flags for interrupt_status_number.</td>
</tr>
<tr>
<td>interrupt_trigger_mode_t</td>
<td>Interrupt trigger modes (used in interrupt_init).</td>
</tr>
</tbody>
</table>

Table 9.5 Types defined in intrerrup.h

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERRUPT_GLOBAL_ENABLE</td>
<td>Global interrupt enables bit number</td>
</tr>
</tbody>
</table>

Table 9.6 Macros defined in intrerrup.h
9.18 Interrupt function definitions

**interrupt_clear**

Clear a pending interrupt.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_clear(int Level)
```

**Arguments:**

- `int Level
  - Interrupt level.`

**Results:**

- Returns 0 for success, -1 if an error occurs.

**Errors:**

- Returns -1 if the interrupt level is illegal.

**Description:**

This function clears the specified pending interrupt level. Interrupts must be disabled when writing to the interrupt controller’s *Pending* register, and so this function first reads whether the interrupt is enabled, and if so disables it, before writing to the *Clear_Pending* register to clear the interrupt, and finally re-enabling the interrupt if it was previously enabled.

**Applies to:**

- ISR, HPP, ILC-None, ILC-1.

**See also:**

- `interrupt_clear_number` `interrupt_raise` `interrupt_pending`
**interrupt_clear_number**

Clear a pending interrupt number.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_clear_number(int Number)
```

**Arguments:**

- `int Number` Interrupt number.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt number is illegal.

**Description:**

This function clears the specified pending interrupt number. If this is the only interrupt number which is pending and that is attached to the interrupt level, then the pending interrupt level will be cleared as well.

Note that on an ILC-1 type interrupt level controller, *interrupt_clear_number* only works with interrupt numbers which have been triggered using *interrupt_raise_number().* It will have no effect on interrupts which have been triggered by a peripheral.

**Applies to:**

ISR, HPP, ILC-1, ILC-2, ILC-3.

**See also:**

*interrupt_clear* *interrupt_raise_number* *interrupt_pending*
interrupt_delete  Delete an interrupt handler.

Synopsis:

```
#include <interrup.h>
int interrupt_delete(int Level);
```

Arguments:

- **int Level**: Interrupt level

Results:

- Returns 0 on success, -1 on failure.

Errors:

- Returns -1 if the interrupt level is illegal.

Description:

This function allows an initialized interrupt to be deleted. This will then allow the interrupt handler's stack to be freed, as no more interrupts will be generated at this level.

Before calling this function the interrupt handlers must first be disabled at the peripheral level (to avoid unexpected interrupts) and uninstalled (by calling `interrupt_uninstall`).

Applies to:

- ILC-None, ILC-1, ILC-2, ILC-3.

See also:

- `interrupt_init`  `interrupt_uninstall`
**interrupt_disable**  
Disable an interrupt level.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_disable(int Level);
```

**Arguments:**

- int Level  
  Interrupt level.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt level is illegal.

**Description:**

Disable interrupt level Level. This involves writing to the interrupt controller’s **Mask** register.

Note that although the global enables bit can still be specified as **INTERRUPT_GLOBAL_ENABLE** this usage is no longer recommended, use the function **interrupt_disable_global instead**.

**Note:** this function is provided as part of the IntC library (see Table 9.1), and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library (see Table 9.2) enables all interrupt levels, and expects them to remain enabled, so that interrupts can be controlled using the function **interrupt_disable_number**. Thus the use of **interrupt_disable** is discouraged.

**Applies to:**

- ISR, HPP, ILC-None, ILC-1.

**See also:**

- **interrupt_disable_mask**  
- **interrupt_enable**  
- **interrupt_disable_global**
interrupt_disable_global

Globally disable interrupts.

Synopsis:

```c
#include <interru.h>
void interrupt_disable_global();
```

Arguments:

None.

Results:

None.

Errors:

None.

Description:

This function clears the global enables bit thus disabling all interrupts. This will prevent the interrupt controller from attempting to raise any interrupts.

Note that this operation is not the same `interrupt_lock`. It does not disable preemption or timeslicing and tasks are permitted to deschedule with interrupts disabled. On an ST20-C2 high priority processes, channels and timers will still be available. On an ST20-C1 core the timer interrupt will be disabled so timer waits will not be handled until interrupts are re-enabled.

Applies to:

ISR, HPP, ILC-None, ILC-1, ILC-2, ILC-3.

See also:

`interrupt_enable_global` `interrupt_lock` `interrupt_unlock`
interrupt_disable_mask

Disable one or more interrupts.

Synopsis:

```
#include <interrup.h>
void interrupt_disable_mask(int Mask);
```

Arguments:

- int Mask
  Interrupt mask.

Results:

- None.

Errors:

- None.

Description:

This function simply writes Mask into the Interrupt controller's Clear_Mask register, thus disabling all the specified interrupts.

Note that although the global enables bit can still be specified in this mask as 1 << INTERRUT_GLOBAL_ENABLE this usage is no longer recommended, use the function interrupt_disable_global instead.

Note: this function is provided as part of the IntC library (see Table 9.1), and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library (see Table 9.2) enables all interrupt levels, and expects them to remain enabled, so that interrupts can be controlled using the function interrupt_disable_number. Thus the use of interrupt_disable_mask is discouraged.

Applies to:

- ISR, HPP, ILC-None, ILC-1.
### interrupt_disable_number

**Synopsis:**

```c
#include <interrup.h>
int interrupt_disable_number(int Number);
```

**Arguments:**

- `int Number`  
  Interrupt number.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt number is illegal.

**Description:**

Disable interrupt number `Number`. This involves writing to one of the interrupt level controller's `Clear_Enable` registers.

**Applies to:**

- ISR, HPP, ILC-2, ILC-3.

**See also:**

- `interrupt_enable_number`
**interrupt_enable**

Enable an interrupt level.

**Synopsis:**

```c
#include <intrrup.h>
int interrupt_enable(int Level);
```

**Arguments:**

- `int Level`  
  Interrupt level.

**Results:**

- Returns 0 for success, -1 if an error occurs.

**Errors:**

- Returns -1 if the interrupt level is illegal.

**Description:**

Enable interrupt level Level. This involves writing to the interrupt controller’s `Set_Mask` register.

**Note:** that although the global enables bit can still be specified as `INTERRUPT_GLOBAL_ENABLE` this usage is no longer recommended, use the function `interrupt_enable_global` instead.

**Note:** this function is provided as part of the IntC library (see Table 9.1), and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library (see Table 9.2) enables all interrupt levels, and expects them to remain enabled, so that interrupts can be controlled using the function `interrupt_enable_number`. Thus the use of `interrupt_enable` is discouraged.

**Applies to:**

- ISR, HPP, ILC-None, ILC-1.

**See also:**

- `interrupt_disable`  
  `interrupt_enable_mask`
**interrupt_enable_global**

Globally enable interrupts.

**Synopsis:**

```c
#include <interrup.h>
void interrupt_enable_global();
```

**Arguments:**

None.

**Results:**

None.

**Errors:**

None.

**Description:**

This function sets the global enables bit thus permitting specifically enabled interrupts to generate interrupts. At power on the global enables bit is cleared. The user must call `interrupt_enable_global` before any interrupts are generated.

**Applies to:**

ISR, HPP, ILC-None, ILC-1, ILC-2, ILC-3.

**See also:**

`interrupt_disable_global` `interrupt_lock` `interrupt_unlock`
**interrupt_enable_mask**  
Enable one or more interrupts.

**Synopsis:**

```c
#include <interrup.h>
void interrupt_enable_mask(int Mask);
```

**Arguments:**

- `int Mask`  
  Interrupt mask.

**Results:**

None.

**Errors:**

None.

**Description:**

This function simply writes `Mask` into the Interrupt controller’s `Set_Mask` register, thus enabling all the specified interrupts.

Note that although the global enables bit can still be specified in this mask as `1 << INTERRUPT_GLOBAL_ENABLE` this usage is no longer recommended, use the function `interrupt_enable_global` instead.

**Note:** this function is provided as part of the IntC library (see Table 9.1), and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library (see Table 9.2) enables all interrupt levels, and expects them to remain enabled, so that interrupts can be controlled using the function `interrupt_enable_number`. Thus the use of `interrupt_enable_mask` is discouraged.

**Example:**

```c
int main()
{
    int Mask;

    /* Enable global interrupts and interrupt 1 */
    Mask = (1 << INTERRUPT_GLOBAL_ENABLE) | (1<<1);
    interrupt_enable_mask(Mask);
    ...
}
```

**Applies to:**

- ISR, HPP, ILC-None, ILC-1.

**See also:**

- `interrupt_disable_mask`
- `interrupt_enable`
**interrupt_enable_number**

Enable an interrupt number

**Synopsis:**

```c
#include <interrup.h>
int interrupt_enable_number(int Number);
```

**Arguments:**

- `int Number`: Interrupt number.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt number is illegal.

**Description:**

Enable interrupt number Number. This involves writing to one of the interrupt level controller's Set_Enable registers.

**Applies to:**

ISR, HPP, ILC-2, ILC-3.

**See also:**

`interrupt_enable_number`
interrupt_init

Initialize an interrupt level.

Synopsis:

```c
#include <interrup.h>

int interrupt_init(
    int interrupt,
    void* stack_base,
    size_t stack_size,
    interrupt_trigger_mode_t trigger_mode,
    interrupt_flags_t flags);
```

Arguments:

- **int interrupt**
  - Interrupt level.
- **void* stack_base**
  - Address of the base of the interrupt handler’s stack.
- **size_t stack_size**
  - Size of the interrupt handler’s stack in bytes.
- **interrupt_trigger_mode_t trigger_mode**
  - Interrupt trigger mode, see Table 9.7.
- **interrupt_flags_t flags**
  - Various flags which affect interrupt behavior. See Table 9.8.

Results:

Returns 0 for success, -1 if an error occurs.

Errors:

Returns -1 if the interrupt number or level are illegal, or if interrupt_init_controller() has not yet been called.

Description:

This function initializes a single interrupt level in the interrupt controller, ready for interrupt handlers to be installed (using interrupt_install). stack_base and stack_size specify a single stack area, which must be large enough to accommodate the largest interrupt handler routine for that interrupt level. Only one interrupt handler will ever use the stack at a time. trigger_mode is one of the supported trigger modes, selected from the list shown in Table 9.7.
flags is used to give additional information about the interrupt. Normally flags should be specified as 0, which will result in the default behavior, however, other options can be specified which will change the behavior of the interrupt. Possible values for flags are shown in Table 9.8.

Currently the only supported options to flags are for the ST20-C2 where the scheduling priority of the handler must be specified. This specifies how the interrupt interacts with ST20 processes. Low priority interrupts will only interrupt low priority processes (and can themselves be interrupted by high priority processes), while high priority interrupts will interrupt both high and low priority processes.

<table>
<thead>
<tr>
<th>Interrupt trigger mode name</th>
<th>Interrupt behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_trigger_mode_high_level</td>
<td>Trigger while the input is high</td>
</tr>
<tr>
<td>interrupt_trigger_mode_low_level</td>
<td>Trigger while the input is low</td>
</tr>
<tr>
<td>interrupt_trigger_mode_rising</td>
<td>Trigger on the rising edge of the input</td>
</tr>
<tr>
<td>interrupt_trigger_mode_falling</td>
<td>Trigger on the falling edge of the input</td>
</tr>
<tr>
<td>interrupt_trigger_mode_any</td>
<td>Trigger on rising and falling edges</td>
</tr>
</tbody>
</table>

Table 9.7 Interrupt trigger modes

<table>
<thead>
<tr>
<th>Interrupt flags</th>
<th>Interrupt behavior</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Trigger at high scheduling priority (Default)</td>
<td>Any</td>
</tr>
<tr>
<td>interrupt_flags_low_priority</td>
<td>Trigger at low scheduling priority</td>
<td>ST20-C2</td>
</tr>
<tr>
<td>interrupt_flags_high_priority</td>
<td>Trigger at high scheduling priority</td>
<td>ST20-C2</td>
</tr>
</tbody>
</table>

Table 9.8 Interrupt flags

### Applies to:
- ILC-None, ILC-1, ILC-2, ILC-3.

### See also:
- interrupt_install
interrupt_init_controller

Initialize the interrupt controller.

Synopsis:

```c
#include <interrup.h>

void interrupt_init_controller(
    void* interrupt_controller,
    int interrupt_levels,
    void* level_controller,
    int interrupt_numbers,
    int input_offset);
```

Arguments:

- `void* interrupt_controller`: Interrupt controller base address.
- `int interrupt_levels`: Number of interrupt levels.
- `void* level_controller`: Interrupt level controller base address.
- `int interrupt_numbers`: Number of interrupt numbers.
- `int input_offset`: Offset of the InputInterrupts register in the interrupt level controller.

Results:

None.

Errors:

None.

Description:

`interrupt_init_controller()` is used to tell STLite/OS20 how the interrupt controller and interrupt level controller are configured for a particular variant of the ST20. This function is always required to be executed once, prior to any interrupt handling routines. `interrupt_controller` and `interrupt_levels` specify the base address and number of interrupt levels (i.e. the number of inputs) supported by the interrupt controller. Similarly, `level_controller` and `interrupt_numbers` specify the base address and number of interrupt numbers (i.e. inputs) supported by the interrupt level controller. `input_offset` gives the offset in words into the interrupt level controller of the InputInterrupts register.

Example:

```c
/* Set up a STi5500*/
interrupt_init_controller((void*)0x20000000, 8,
                        (void*) 0x20011000, 18, 18);
```

Applies to:

ILC-None, ILC-1, ILC-2, ILC-3.
9.18 Interrupt function definitions

See also:

interrupt_init
interrupt_install

Install an interrupt handler.

Synopsis:

```c
#include <interrup.h>
int interrupt_install(
    int Number,
    int Level,
    void(*Handler)(void* Param),
    void* Param);
```

Arguments:

- **int Number**: Interrupt number.
- **int Level**: Interrupt level.
- **void (*Handler)(void* Param)**: Pointer to the interrupt handler entry point.
- **void* Param**: A parameter which will be passed to Handler when it is called.

Results:

Returns 0 for success, -1 if an error occurs.

Errors:

Returns -1 if the interrupt number or level are illegal, or if interrupt_init() has not yet been called for the interrupt level.

Description:

Install the interrupt handler to be called when the specified interrupt number occurs. Normally this involves programming the interrupt level controller to associate an interrupt level with an interrupt number, and setting up the function pointer and parameter in STLite/OS20's internal data structures.

However if the interrupt number is specified as -1, then no attempt will be made to program the interrupt level controller, and the interrupt function will be associated with the interrupt level. This technique must be used on ST20 hardware which does not have an interrupt level controller, but may also be useful when an interrupt is only triggered from software, and never from a hardware device.

The ILC-3 can route interrupts from internal peripherals to external pins allowing an external processor to handle that peripheral. interrupt_install is used to program this behavior. The interrupt level should be specified as the number of the external pin logically added to INTERRUPT_EXTERNAL_PIN. In this case the ST20 will not handle the interrupt so the handler and parameter must be specified as NULL.
Example:

```c
/* normal use */
int interrupt_stack[500];
void interrupt_handler(void* param);

interrupt_init(4, interrupt_stack, sizeof(interrupt_stack),
interrupt_trigger_mode_rising, 0);
interrupt_install(10, 4, interrupt_handler, NULL);

/* routing to external pin 1 (ILC-3 only) */
interrupt_install(12, 1 + INTERRUPTEXTERNAL_PIN, NULL, NULL);
```

Applicability:

ILC-None, ILC-1, ILC-2, ILC-3.

See also:

interrupt_init
**interrupt_install_sl**

Install an interrupt handler specifying a static link.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_install(
    int Number,
    int Level,
    void (*Handler)(void* Param),
    void* Param
    void* StaticLink);
```

**Arguments:**

- **int Number**: Interrupt number.
- **int Level**: Interrupt level.
- **void (*Handler)(void* Param)**: Pointer to the interrupt handler entry point.
- **void* Param**: A parameter which will be passed to Handler when it is called.
- **void* StaticLink**: Static link to be used when calling Function.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt number or level are illegal, or if `interrupt_init()` has not yet been called for the interrupt level.

**Description:**

Install the interrupt handler to be called when the specified interrupt number occurs. Normally this involves programming the interrupt level controller to associate an interrupt level with an interrupt number, and setting up the function pointer and parameter in STLite/OS20's internal data structures.

However, if the interrupt number is specified as -1, then no attempt will be made to program the interrupt level controller, and the interrupt function will be associated with the interrupt level. This technique must be used on ST20 hardware which does not have an interrupt level controller, but may also be useful when an interrupt is only triggered from software, and never from a hardware device.

`StaticLink` is the static link which should be used when calling `Handler`. This will normally be obtained as a result of loading an RCU.

**Applicability:**

ILC-None, ILC-1, ILC-2, ILC-3.
9.18 Interrupt function definitions

See also:

interrupt_init interrupt_install
**interrupt_lock**

Locks all interrupts.

**Synopsis:**

```c
#include <interrupt.h>
void interrupt_lock(void);
```

**Arguments:**

None.

**Results:**

None.

**Errors:**

None.

**Description:**

This function disables all interrupts to the CPU. This will prevent any interrupts from the interrupt controller having any effect on the currently executing task. In addition, on the ST20-C2 this will also disable high priority processes, channels and timers.

This function should 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 will not be released until an equal number of calls to `interrupt_unlock()` have been made.

A task must not deschedule while an interrupt lock is in effect.

**Applies to:**

ISR, HPP, ILC-None, ILC-1, ILC-2, ILC-3.

**See also:**

`interrupt_unlock` `task_lock`
**interrupt_pending**

Return pending interrupt levels.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_pending(void);
```

**Arguments:**

None.

**Results:**

A mask specifying which interrupt levels are currently pending.

**Errors:**

None.

**Description:**

Return which interrupt levels are currently pending. That is, those interrupts which have been set by peripheral devices, but their handlers have not yet been run. This simply involves reading the interrupt controller’s **Pending** register.

**Applies to:**

ISR, HPP, ILC-None, ILC-1.

**See also:**

`interrupt_clear interrupt_raise interrupt_pending_number`
**interrupt_pending_number**

Return pending interrupt numbers.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_pending_number(void);
```

**Arguments:**

None.

**Results:**

A mask specifying which interrupt numbers are currently pending.

**Errors:**

None.

**Description:**

Return which interrupt numbers are currently pending. That is, all the interrupts which are currently set by the peripherals. This simply involves reading the interrupt level controller's `InputInterrupts` register and combining it with the software register maintained by `interrupt_raise_number`.

**Note:** this function cannot be fully implemented for ILC-3, therefore its use is not recommended with any ILCs, `interrupt_test_number` should be used instead.

**Applies to:**

ISR, HPP, ILC-None, ILC-1, ILC-2.

**See also:**

`interrupt_test_number` `interrupt_pending`
**interrupt_raise**

Raise an interrupt level.

**Synopsis:**
```
#include <interrup.h>
int interrupt_raise(int Level);
```

**Arguments:**
- int Level
  
**Results:**
Returns 0 for success, -1 if an error occurs.

**Errors:**
Returns -1 if the interrupt level is illegal.

**Description:**
Raise the specified interrupt level. This involves writing to the interrupt controller’s Set-Pending register.

Note that this function does not write to the interrupt level controller, so it should only be used with interrupts levels which are attached to a single interrupt number. If interrupt Level has multiple interrupt numbers attached to it, the results are undefined.

**Applies to:**
ISR, HPP, ILC-None, ILC-1.

**See also:**
interrupt_clear interrupt_enable interrupt_install interrupt_raise_number
**interrupt_raise_number**  
Raise an interrupt number.

**Synopsis:**
```c
#include <interrupt.h>
int interrupt_raise_number(int Number);
```

**Arguments:**
- `int Number`  
  Interrupt number.

**Results:**
- Returns 0 for success, -1 if an error occurs.

**Errors:**
- Returns -1 if the interrupt number is illegal.

**Description:**
Simulate the raising of an interrupt number. This function is equivalent to `interrupt_raise()`, except that it works with interrupt levels which have multiple interrupt numbers attached. It does this by maintaining a software equivalent of the interrupt controller’s `InputInterrupts` register, which is checked by the first level interrupt handler, as well as the hardware register.

**Applies to:**
- ISR, HPP, ILC-1, ILC-2, ILC-3.

**See also:**
- `interrupt_clear`
- `interrupt_enable`
- `interrupt_install`
**interrupt_status**

Report the status of an interrupt level.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_status(int Level, interrupt_status_t* Status, interrupt_status_flags_t flags);
```

**Arguments:**

- `int Level`: Interrupt level.
- `interrupt_status_t* Status`: Pointer to a structure that the current status can be written to.
- `interrupt_status_flags_t flags`: What information to return.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt level is illegal.

**Description:**

This function returns useful information about an interrupt level. This information can be of benefit when debugging an application.

<table>
<thead>
<tr>
<th>Structure member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>interrupt_numbers</td>
<td>Number of interrupt handlers attached to this level.</td>
</tr>
<tr>
<td>interrupt_stack_base</td>
<td>Pointer to the base of the stack space for this level.</td>
</tr>
<tr>
<td>interrupt_stack_size</td>
<td>Size of the stack for this level, in bytes.</td>
</tr>
<tr>
<td>interrupt_stack_used</td>
<td>Peak stack usage in bytes.</td>
</tr>
<tr>
<td>interrupt_time</td>
<td>Time spent servicing interrupts at this level. †</td>
</tr>
<tr>
<td>interrupt_count</td>
<td>Number of times an interrupt at this level has been serviced. †</td>
</tr>
</tbody>
</table>

† Note that `interrupt_time` and `interrupt_count`, should not be used on standard STLite/OS20 kernels. Standard kernels do not record this data as this decreases interrupt performance. Refer to section C.3.4 for further details.

**Table 9.9** The `interrupt_status_t` structure

The Flags parameter is used to indicate which values should be returned. Values which can be determined immediately (all except `interrupt_stack_used`) will always be returned. If only these fields are required then Flags should be set to 0. However, calculating peak stack usage may take a while, and so is only returned when Flags is set to `interrupt_status_flags_stack_used`.

**Applies to:**

ILC-None, ILC-1, ILC-2, ILC-3.
See also:

interrupt_status_number
**interrupt_status_number**  Report the status of an interrupt number.

**Synopsis:**

```c
#include <interrup.h>

int interrupt_status_number(
    int Number,
    interrupt_status_number_t* Status,
    interrupt_status_number_flags_t flags);
```

**Arguments:**

- `int Number`  Interrupt number.
- `interrupt_status_number_t* Status`  Pointer to a structure where the current status can be written to.
- `interrupt_status_number_flags_t flags`  Reserved for future use, `flags` should be set to zero.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt level is illegal.

**Description:**

This function returns useful information about an interrupt number. This information can be of benefit when debugging an application.

<table>
<thead>
<tr>
<th>Structure Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>intnum_status_level</td>
<td>The level that this interrupt number is attached to.</td>
</tr>
<tr>
<td>intnum_time</td>
<td>Time spent servicing this interrupt number. †</td>
</tr>
<tr>
<td>intnum_count</td>
<td>Number of times this interrupt number has been serviced. †</td>
</tr>
</tbody>
</table>

† Note that `intnum_time` and `intnum_count`, should not be used on standard STLite/OS20 kernels. Standard kernels do not record this data as this decreases interrupt performance. Refer to section C.3.4 for further details.

**Table 9.10 The interrupt_status_number_t structure**

**Applies to:**

ILC-1, ILC-2, ILC-3.

**See also:**

`interrupt_status_number`
**interrupt_test_number**  Test whether an interrupt number is pending.

**Synopsis:**
```
#include <interrup.h>
int interrupt_test_number(int Number);
```

**Arguments:**
- `int Number`  Interrupt number

**Results:**
- Returns 1 if interrupt number `Number` is pending, 0 if it is not, -1 if an error occurs.

**Errors:**
- Returns -1 if the interrupt number is illegal.

**Description:**
Tests the interrupt level controllers **Pending** register to see if interrupt number `Number` is pending.

Note that if `Number` is not valid and returns the error code the function will evaluate to true if used in a conditional context (see below).

**Example:**
```
/* do not do this */
if (interrupt_test_number(n)) {
   /* n is pending OR n is not valid */
   ...
}

/* this is safer */
if (interrupt_test_number(n) == 1) {
   /* n is pending */
   ...
}
```

**Applies to:**
- ISR, HPP, ILC-1, ILC-2, ILC-3

**See also:**
- `interrupt_pending_number`
**interrupt_trigger_mode_number**  
Change the trigger mode of an interrupt number.

**Synopsis:**
```
#include <interrup.h>

int interrupt_trigger_mode_number(
    int Number,
    interrupt_trigger_mode_t trigger_mode);
```

**Arguments:**
- `int Number`: Interrupt number.
- `interrupt_trigger_mode_t trigger_mode`: Interrupt trigger mode, see Table 9.4.

**Results:**
Returns 0 for success, -1 if an error occurs.

**Errors:**
Returns -1 if the interrupt number or interrupt trigger mode is illegal.

**Description:**
This function changes the trigger mode of an interrupt number on ILC-2 or ILC-3.

Be aware that `interrupt_install` will set the trigger mode based on that default supplied to `interrupt_init`. Therefore `interrupt_trigger_mode_number` must be called after the interrupt handler has been installed to prevent its effects from being overwritten.

**Example:**
```
#include <interrup.h>

void interrupt_handler(void* param);

interrupt_install(10, 4, interrupt_handler, NULL);
interrupt_trigger_mode_number(10, interrupt_trigger_mode_falling);
interrupt_enable_number(10);
```

**Applies to:**
- ISR, HPP, ILC-2, ILC-3

**See also:**
- `interrupt_init`
**interrupt_uninstall**

Uninstall an interrupt handler.

**Synopsis:**

```c
#include <interrupt.h>
int interrupt_uninstall(
    int Number,
    int Level);
```

**Arguments:**

- **int Number**
  - Interrupt number.
- **int Level**
  - Interrupt level.

**Results:**

Returns 0 on success, -1 on failure.

**Errors:**

If the interrupt number or level are illegal, or no interrupt has been installed, then this will fail.

**Description:**

This function allows an interrupt handler to be uninstalled. This will then allow a replacement handler function to be installed as a replacement. No attempt is made to disable the interrupt, so before calling this function the interrupt must have been disabled at the peripheral level.

On systems which do not have an interrupt level controller, specify the **Number** parameter as -1.

**Applies to:**

ILC-None, ILC-1, ILC-2, ILC-3

**See also:**

`interrupt_delete` `interrupt_install`
interrupt_unlock

Unlock all interrupts.

Synopsis:

```
#include <interrup.h>
void interrupt_unlock(void);
```

Arguments:

None.

Results:

None.

Errors:

None.

Description:

This function re-enables all interrupts to the CPU. Any interrupts which have been prevented from executing will start immediately.

This function should 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 will not be released until an equal number of calls to interrupt_unlock() have been made.

Applies to:

ISR, HPP, ILC-None, ILC-1, ILC-2, ILC-3

See also:

interrupt_lock task_lock
**interrupt_wakeup_number**  
Set wakeup status of an interrupt number.

**Synopsis:**

```c
#include <interrup.h>
int interrupt_wakeup_number(int Number, 
interrupt_trigger_mode_t trigger_mode);
```

**Arguments:**

- `int Number`  
  Interrupt number.
- `interrupt_trigger_mode_t trigger_mode`  
  Interrupt trigger mode.

**Results:**

Returns 0 for success, -1 if an error occurs.

**Errors:**

Returns -1 if the interrupt number of trigger mode are illegal.

**Description:**

This function routes an external interrupt to the low power controller, enabling an ST20 with ILC-2 or ILC-3 to exit power-down mode only for specific interrupts.

Note that the set of possible trigger modes differs slightly from those used by `interrupt_init` and `interrupt_trigger_mode_number`.

<table>
<thead>
<tr>
<th>Interrupt trigger mode name</th>
<th>Wakeup behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>interrupt_trigger_mode_no_trigger</code></td>
<td>Do not wakeup</td>
</tr>
<tr>
<td><code>interrupt_trigger_mode_high_level</code></td>
<td>Wakeup while the input is high</td>
</tr>
<tr>
<td><code>interrupt_trigger_mode_low_level</code></td>
<td>Wakeup while the input is low</td>
</tr>
</tbody>
</table>

Table 9.11  Wakeup trigger modes

**Applies to:**

ISR, HPP, ILC-2, ILC-3.
9.18 Interrupt function definitions
10 Device information

Two functions are provided to return information about the ST20 family of devices. `device_id` returns the ID of the current device. `device_name` takes a device ID as input and returns a brief description of the device.

Device Identifiers are defined by the IEEE1149.1 (JTAG) Boundary-Scan Standard. This is a 32 bit number composed of a number of fields. STLite/OS20 defines a type to describe this, `device_id_t`. This is a union with three fields:

- `id` which allows the code to be manipulated as a 32 bit quantity.
- `jtag` which views the value as defined by the JTAG standard.
- `st` which views the value as used by STMicroelectronics. This divides the device code into a family and device code.

`jtag` and `st` are structs of bit-fields, which allows the elements to be accessed symbolically.

The identification code is made up as in Table 10.1.

<table>
<thead>
<tr>
<th>bits</th>
<th>jtag</th>
<th>st</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-28</td>
<td>revision</td>
<td>revision</td>
<td>Mask revision</td>
</tr>
<tr>
<td>27-22</td>
<td>device_code</td>
<td>family</td>
<td>$20_{10}$ – STAR family</td>
</tr>
<tr>
<td>21-12</td>
<td></td>
<td>device_code</td>
<td>Device code</td>
</tr>
<tr>
<td>11-1</td>
<td>manufacturer</td>
<td>manufacturer</td>
<td>$32_{10}$ – STMicroelectronics</td>
</tr>
<tr>
<td>0</td>
<td>JTAG_bit</td>
<td>JTAG_bit</td>
<td>1 – fixed by JTAG</td>
</tr>
</tbody>
</table>

Table 10.1 Composition of identification code

10.1 Device ID header file: `device.h`

All the definitions related to device identification are in the single header file, `device.h`, see Table 10.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/ HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>device_id</code></td>
<td>Returns the ID of the current device.</td>
<td>ISR and HPP.</td>
</tr>
<tr>
<td><code>device_name</code></td>
<td>Returns the name of the current device.</td>
<td>ISR and HPP.</td>
</tr>
</tbody>
</table>

Table 10.2 Functions defined in `device.h`

All functions are callable from an STLite/OS20 task. Functions in Table 10.2 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor.
<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device_id_t</td>
<td>Device ID.</td>
</tr>
</tbody>
</table>

Table 10.3 Types defined in `device.h`
10.2 Device ID function definitions

**device_id**

Return the current device ID.

Synopsis:

```c
#include <device.h>
device_id_t device_id(void);
```

Arguments:

None.

Results:

Returns the device ID for the current device.

Errors:

None.

Description:

`device_id` returns the device identification (ID) for the current device. The result is a union which breaks down the different fields of the device ID.

See also:

`device_name`
device_name

Return the name of the specified device.

Synopsis:

```
#include <device.h>
const char* device_name(device_id_t id);
```

Arguments:

- device_id_t id: The device ID

Results:

Returns a pointer to static data which contains the device name and whose content is overwritten by each call.

Errors:

None.

Description:

`device_name` returns the address of a buffer containing a text string describing the specified device ID. A typical result would be the device name and its revision, for example `STi5500-A`.

Example:

```
printf("Device name %s\n", device_name(device_id()));
```

See also:

- device_id
11 Caches

Cache provides a way to reduce the time taken for the CPU to access memory and so can greatly increase system performance.

11.1 Introduction

All ST20 processors that support cache use similar hardware and the operation of the caches is the same, however, the blocks of memory that can be cached vary between ST20 devices, see the appropriate device datasheets for details.

The ST20 cache system provides a read-only instruction cache and a write-back data cache. The memory used by the instruction cache may only be used for caching whereas the memory used by the data cache may be configured as extra internal SRAM.

The use of the data cache effectively reduces the amount of internal SRAM available to the application. It is therefore important that the data cache memory is protected from being used by the application by reserving it during the link process. This is done by using the memory command within the configuration procedures passed to the compile/link driver st20cc. (See the ‘ST20 Embedded Toolset User Manual - 72-TDS-505’ for further details). For example:

```
K = 1024
proc dxxdcache {
    chip STi5500
    memory DCACHE (0x80000000 + 2*K) (2*K) RESERVED
    ....
}
```

In this example the region of memory reserved as data cache is specific to the STi5500.

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 enabling the data cache for blocks of memory accessed by the DMA engine is not recommended. Note that on an ST20-C2 core, device access instructions (generated with #pragma ST_device) bypass the cache and can be used to solve some cache coherency issues.

11.2 Initializing and configuring the cache support system

Before any call is made to the cache handling routines the cache control hardware needs to be configured and initialized in order that STLite/OS20 knows which hardware model is being targeted.
11.3 Enabling the caches

If the `st20cc -runtime os20` command is used when linking, then the cache controller will be configured automatically before the user’s application starts to run.

If `st20cc -runtime os20` is not used the `cache_init_controller` function enables you to specify how the cache control hardware is configured:

```c
#include <cache.h>
void cache_init_controller(
  void* cache_controller,
  cache_map_data_t cache_map);
```

Both the cache controller address and the cache map are device specific. The cache controller address can be obtained from the device datasheet. The correct cache map can be found in the `cache_init_controller` function definition, see Table 11.4.

Next the data cache must be configured. The `cache_config_data` function is used to configure blocks of memory as cacheable or non-cacheable.

```c
#include <cache.h>
cache_config_data(
  char* start_address,
  char* end_address,
  cache_config_flags_t flags);
```

This function configures the data cache. It will enable or disable all cacheable blocks that fall within the specified address range based upon the flags it is given.

### 11.3 Enabling the caches

The caches are enabled using the following two functions:

```c
#include <cache.h>
int cache_enable_data();
int cache_enable_instruction();
```

The first function invalidates the data cache (see section 11.6), before writing to the `DCacheNotSRAM` register thereby enabling the data cache. The second function is similar but operates on the `EnableICache` register.

### 11.4 Locking the cache configuration

The cache can be locked using the following function:

```c
int cache_lock();
```

It is recommended that all cache configuration is performed at boot time and then never modified. To prevent accidental modification ST20 devices can lock the cache configuration preventing it from being changed until the hardware is reset.
11 Caches

11.5 Example: setting up the caches

This example shows how the caches could be set up for STi5510 devices. This example demonstrates the steps described in the previous sections to:

- Initialize and configure the cache hardware, see section 11.2.
- Enable the data and instruction caches, see section 11.3.
- Lock the cache configuration, see section 11.4.

This example uses the header file `<chip/STi5510addr.h>` supplied in the ST20 Embedded Toolset’s standard configuration files directory: `$ST20ROOT/include`. The header file contains the base address of the cache controller, defined as `CacheControlAddr`.

```c
#include <chip/STi5510addr.h>
#include <cache.h>

cache_init控制器((void*) CacheControlAddr, cache_map_sti5510);

cache_enable_instruction();

/* cache all possible memory */
cache_config_data(0x80000000, 0x7fffffff, cache_config_enable);

/* except region required for DMA */
cache_config_data(0x40010000, 0x4001ffff, cache_config_disable);

cache_enable_data();

cache_lock();
```

11.6 Flushing and invalidating caches

When the cache is enabled any data written to main memory will be stored in the cache and marked as *dirty* so that at some point in the future it can be properly stored to main memory. A cache flush causes all dirty cache lines to be written immediately to main memory.

Invalidating a cache causes the cache to forget its entire contents thus forcing it to reload all data from main memory.

**Note:** that on ST20 devices, flushing the cache will also cause it to be invalidated. After a cache flush all data will be retrieved from main memory.

In some applications it is useful to force a cache flush or invalidate, this can be achieved using the following three functions:

```c
int cache_flush_data(void* reserved1, void* reserved2);
int cache_invalidate_data(void* reserved1, void* reserved2);
int cache_invalidate_instruction(void* reserved1, void* reserved2);
```

Each of these functions takes two arguments that are reserved for future use by STLite/OS20, users must supply NULL as each argument.
11.6.1 Relocatable code units
When caches are enabled extra care must be taken when handling relocatable code units. The advise given in section B.1.5 should be followed to ensure cache coherency is maintained.

11.7 Cache header file: cache.h
All the definitions related to the caches are in the single header file, cache.h, see Table 11.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/ HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_config_data</td>
<td>Configure the data cache.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_enable_data</td>
<td>Enable the data cache.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_enable_instruction</td>
<td>Enable the instruction cache.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_flush_data</td>
<td>Flush the data cache.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_init_controller</td>
<td>Initialize the cache controller.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_invalidate_data</td>
<td>Invalidate the data cache.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_invalidate_instruction</td>
<td>Invalidate the instruction cache.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_lock</td>
<td>Lock the cache configuration.</td>
<td>ISR and HPP</td>
</tr>
<tr>
<td>cache_status</td>
<td>Report the cache status</td>
<td>ISR and HPP</td>
</tr>
</tbody>
</table>

Table 11.1 Functions defined in cache.h

All functions are callable from an STLite/OS20 task. Functions in Table 11.1 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor.

The types defined to support the cache API are listed in Table 11.2.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_config_flags_t</td>
<td>Additional flags for cache_config_data.</td>
</tr>
<tr>
<td>cache_map_data_t</td>
<td>Description of cacheable memory available on a particular ST20 variant (used by cache_init_controller).</td>
</tr>
<tr>
<td>cache_status_t</td>
<td>Structure describing the status of the cache.</td>
</tr>
</tbody>
</table>

Table 11.2 Types defined in cache.h
11.8 Cache function definitions

**cache_config_data**  Configure the data cache.

Synopsis:
```
#include <cache.h>
int cache_config_data(
    void* start_address,
    void* end_address
    cache_config_flags_t flags);
```

Arguments:
- `char* start_address`  Start of address range.
- `char* end_address`  End of address range.
- `cache_config_flags_t flags`  Flags which affect behavior.

Results:
Returns 0 for success, -1 if an error occurs.

Errors:
Returns -1 for any of the following is true:
- the cache configuration is locked;
- an attempt is made to disable a cacheable region when the data cache is enabled;
- the start address or end address fall in the middle of a cacheable region;
- the start address and end address do not span a cacheable region;
- the flags are invalid;

Description:
This function writes to the `CacheControl` registers to enable or disable data caching for the specified range. It will enable or disable caching for all cacheable regions between `start_address` and `end_address`. Neither `start_address` nor `end_address` may fall in the middle of a cacheable region. Refer to the appropriate datasheet to find the cache regions for a specific ST20 device.

The ST20 memory map runs from `MININT` to `MAXINT`, therefore addresses supplied to this function will wrap around from 0xffffffff to 0x00000000. To cache all possible memory the following ST20 address range could be specified:
```
cache_config_data (0x80000000, 0x7fffffff ....);
```
Alternatively, the address range in the next example would produce the same result:
```
cache_config_data (0x00000000, 0xffffffff ....);
```
The flags can be used to choose whether the function enables or disables caching for the specified range. Possible values for flags are shown in Table 11.3.

<table>
<thead>
<tr>
<th>Data cache configuration flags</th>
<th>Data cache configuration behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache_config_enable</td>
<td>Enable caching for the specified range</td>
</tr>
<tr>
<td>cache_config_disable</td>
<td>Disable caching for the specified range</td>
</tr>
</tbody>
</table>

Table 11.3 Data cache configuration flags

**Note:** that on STi5500 devices, a single bit in the CacheControl register is used to control the cacheability of non-contiguous blocks of memory. For this device enabling or disabling one such block of memory will actually affect both blocks, refer the device datasheet for further details.

**Example:**

```c
    cache_config_data(
        (void*) 0x80000000, (void*)0x7fffffff, cache_config_enable);
    cache_config_data(
        (void*) 0x40000000, (void*)0x4000ffff, cache_config_disable);
    cache_enable_data();
    cache_lock();
```

**See also:**

`cache_enable_data`
cache_enable_data

Enable the data cache.

Synopsis:

```c
#include <cache.h>
int cache_enable_data();
```

Arguments:
None.

Results:
Returns 0 on success, −1 if an error occurs.

Errors:
Returns −1 if the cache registers have been locked or if the data cache is already enabled.

Description:
This function enables the data cache by writing to the DCacheNotSRAM register. This action is not reversible, disabling the data cache once enabled is not recommended.

Before enabling the data cache it should have been configured by making calls to cache_config_data.

See also:
cache_config_data, cache_enable_instruction
### cache_enable_instruction

Enable the instruction cache.

**Synopsis:**

```c
#include <cache.h>
int cache_enable_instruction();
```

**Arguments:**

None.

**Results:**

Returns 0 on success, -1 if an error occurs.

**Errors:**

Returns -1 if the cache registers have been locked or if the instruction cache is already enabled.

**Description:**

This function enables the instruction cache by writing to the `EnableICache` register. This action is not reversible, disabling the instruction cache once enabled is not recommended.

**See also:**

`cache_enable_data`
cache_flush_data

Synopsis:

```
#include <cache.h>
int cache_flush_data(void* reserved1, void* reserved2);
```

Arguments:

- **void* reserved1**: Reserved for future use (must be NULL).
- **void* reserved2**: Reserved for future use (must be NULL).

Results:

Returns 0 on success, -1 if an error occurs.

Errors:

Returns -1 if the arguments are not NULL or if the data cache is not enabled.

Description:

This function flushes the data cache by writing to the `FlushDCache` register bit. Flushing the data cache causes all dirty lines in the data cache to be written back to memory. A dirty line is a line of cache that has been written to since it was loaded or last written back. Flushing the data cache will also cause the entire cache to be marked invalid. All data will be reloaded from main memory.

**Note:** that any accesses to cacheable memory will be blocked until the flush is complete.

Example:

```
cache_flush_data(NULL, NULL);
```

See also:

- `cache_invalidate_data`
### cache_init_controller

**Initialize the cache controller.**

**Synopsis:**

```c
#include <cache.h>

int cache_init_controller(
    void* cache_controller,
    cache_map_data_t* cache_map);
```

**Arguments:**

- `void* cache_controller`: Cache controller base address, see the appropriate device datasheet for details.
- `cache_map_data_t* cache_map`: Pointer to a description of cacheable memory.

**Results:**

Returns 0 on success, -1 if an error occurs.

**Errors:**

Returns -1 if the cache registers have been locked.

**Description:**

This function is used to tell STLite/OS20 how the cache controller is configured for a particular variant of the ST20. This function must be called prior to any cache handling routines.

If `st20cc -runtime os20` is used when linking, this function will be called automatically before the user’s application starts to run. If `st20cc -runtime os20` is not used then the cache map should be selected from the list below.

<table>
<thead>
<tr>
<th>ST20 variant</th>
<th>Cache map</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST20TP3</td>
<td>cache_map_st20tp3</td>
</tr>
<tr>
<td>STi5500, STi5505</td>
<td>cache_map_sti5500</td>
</tr>
<tr>
<td>STi5508, STi5510</td>
<td>cache_map_sti5510</td>
</tr>
<tr>
<td>STi5512, STi5518 (cache region1 in 64kB blocks)</td>
<td>cache_map_sti5512a</td>
</tr>
<tr>
<td>STi5512, STi5518 (cache region 1 in 512kB blocks)</td>
<td>cache_map_sti5512b</td>
</tr>
</tbody>
</table>

Table 11.4 Cache maps

**cache_init_controller** can also be used to restore the power on state of the **CacheControl** registers, providing that the cache has not been locked. Any work performed by **cache_config_data** will be undone.

**See also:**

- `cache_config_data`, `cache_enable_data`, `cache_enable_instruction`, `cache_lock`
**cache_invalidate_data**

Invalidate the data cache.

**Synopsis:**

```c
#include <cache.h>

int cache_invalidate_data(void* reserved1, void* reserved2);
```

**Arguments:**

- `void *reserved1`  
  Reserved for future use (must be NULL).
- `void *reserved2`  
  Reserved for future use (must be NULL).

**Results:**

Returns 0 on success, -1 if an error occurs.

**Errors:**

Returns -1 if the arguments are not NULL or if the data cache is not enabled.

**Description:**

This function flushes and invalidates the data cache by writing to the `InvalidateD-Cache` register bit. The entire data cache is marked invalid. If not used correctly this will cause data loss. In particular the return address stored when this function is called will be destroyed if the workspace occupies cacheable memory.

**Note:** that any accesses to cacheable memory will be blocked until the flushing and invalidation have completed.

**Example:**

```c
cache_invalidate_data(NULL, NULL);
```

**See also:**

- `cache_flush_data`
- `cache_invalidate_instruction`
**cache_invalidate_instruction**  Invalidate the instruction cache.

### Synopsis:
```
#include <cache.h>

int cache_invalidate_instruction(
    void* reserved1,
    void* reserved2);
```

### Arguments:
- `void *reserved1`: Reserved for future use (must be NULL).
- `void *reserved2`: Reserved for future use (must be NULL).

### Results:
Returns 0 on success, -1 if an error occurs.

### Errors:
Returns -1 if the arguments are not NULL or if the data cache is not enabled.

### Description:
This function invalidates the instruction cache by writing to the `InvalidateICache` register bit. Invalidating the instruction cache marks every line as not containing valid data. This function is intended for use when instruction code has been changed by some means such as replacing one relocatable code unit with another.

**Note:** that any accesses to cacheable memory will be blocked until the invalidation is complete.

### Example:
```
cache_invalidate_instruction(NULL, NULL);
```

### See also:
- `cache_invalidate_data`
### cache_lock

Lock the cache configuration.

**Synopsis:**

```c
int cache_lock();
```

**Arguments:**

None.

**Results:**

Returns 0 on success, -1 if an error occurs.

**Errors:**

Returns -1 if the cache registers have already been locked.

**Description:**

This function locks the cache configuration by writing to the `CacheControlLock` register bit. The cache configuration can only be unlocked by a hardware reset. After the configuration has been locked only invalidating and flushing operations can be performed.

**See also:**

- `cache_flush_data`
- `cache_invalidate_data`
- `cache_invalidate_instruction`
11.8 Cache function definitions

**cache_status**

Report the cache status.

**Synopsis:**

```
#include <cache.h>
cache_status_t cache_status();
```

**Arguments:**

None.

**Results:**

A structure describing the status of the cache.

**Errors:**

None.

**Description:**

This function returns a structure describing the current status of the cache.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCacheNotSRAM</td>
<td>1 if the data cache is enabled, 0 if the data cache memory is configured as extra RAM.</td>
</tr>
<tr>
<td>EnableICache</td>
<td>1 if the instruction cache is enabled, 0 otherwise.</td>
</tr>
<tr>
<td>InvalidatingDCache</td>
<td>1 if the cache controller is invalidating the data cache, 0 otherwise.</td>
</tr>
<tr>
<td>InvalidatingICache</td>
<td>1 if the cache controller is invalidating the instruction cache, 0 otherwise.</td>
</tr>
<tr>
<td>FlushingDCache</td>
<td>1 if the cache controller is flushing the data cache, 0 otherwise.</td>
</tr>
<tr>
<td>DCacheReady</td>
<td>1 if the data cache is ready to perform an operation, 0 otherwise.</td>
</tr>
<tr>
<td>ICacheReady</td>
<td>1 if the instruction cache is ready to perform an operation, 0 otherwise.</td>
</tr>
<tr>
<td>CacheControlLock</td>
<td>1 if the cache configuration is locked, 0 otherwise.</td>
</tr>
</tbody>
</table>

*Table 11.5 Cache status structure*

**Note:** ST20 variants that use `cache_map_sti5500` do not have a `CacheStatus` register so STLite/OS20 implements it in software. In software it is not possible to implement all of the features of the `CacheStatus` register. Therefore only `DCacheNotSRAM`, `EnableICache` and `CacheControlLock` should be used on these processors.

**Example:**

```
cache_status_t status = cache_status();
if ( status.CacheControlLock ) {
    /* cache is locked */
    ...
}
```
12 ST20-C1 specific features

STLite/OS20 has many features, some of which depend on a timer peripheral being present, for example, functions such as `semaphore_wait_timeout` and`time_now`.

For the ST20-C1 version of STLite/OS20 to provide the full API, the developer will need to incorporate a timer plug-in module into any applications built for the ST20-C1 cores. The plug-in module is board specific, but the interface is generic enough to allow STLite/OS20 to take advantage of any timer peripherals that are present.

STLite/OS20 can be used with or without the plug-in module, however, when accessing timer related functions without a plug-in module present, a run-time error will occur so care should be taken when not using the plug-in module.

Internally STLite/OS20 uses a standardized low level timer API which accesses functions provided by the plug-in module via function pointers, see Figure 12.1. This is so that the application can be built with or without the plug-in module. Linkage between STLite/OS20 and the plug-in module is performed at run-time as opposed to compile-time so that the only change needed to the application is an additional call to the plug-in module’s initialization function.

The plug-in module must provide an initialization function which the application can call. Upon calling the initialization function, the module will initialize the programmable timer and pass a structure detailing all of the functions’ locations into STLite/OS20 via a function called `timer_initialize`. This is a STLite/OS20 ST20-C1 specific function call. At this stage the plug-in module will be linked into the STLite/OS kernel.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer_read</td>
<td>Read the timer.</td>
</tr>
<tr>
<td>timer_set</td>
<td>Set the timer.</td>
</tr>
<tr>
<td>timer_enable_int</td>
<td>Enable the timer interrupt</td>
</tr>
<tr>
<td>timer_disable_int</td>
<td>Disable the timer interrupt</td>
</tr>
<tr>
<td>timer_raise_int</td>
<td>Raise a timer interrupt</td>
</tr>
</tbody>
</table>

Table 12.1 Internal STLite/OS20 Timer API

The syntax of the timer API is consistent with the remainder of STLite/OS20. The naming convention has an object oriented approach i.e:

```
<class>_<type_of_operation>
```

The ST20-MC2 plug-in timer module supplied with the product is described in Appendix A.

Each function within the timer API has a one-to-one mapping with a function within the plug-in module (see Table 12.2). The naming convention is consistent with STLite/OS20 with the board name inserted at the beginning of the function name.
Before the plug-in module is initialized, the STLite/OS20 kernel must be installed and initialized by calling `kernel_initialize` and the interrupt controller must be initialized by calling `interrupt_init_controller`.

A side effect of having the timer module outside of STLite/OS20 is that the units of `clock_t` depend on the plug-in timer module. This has the advantage that the length of each clock tick can be adjusted for the specific application, a short tick when high accuracy is required, a longer tick when long durations need to be timed. However this means that care needs to be taken when writing any code which needs to be portable.

<table>
<thead>
<tr>
<th>STLite/OS20 timer function</th>
<th>Plug-in function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>timer_read</code></td>
<td><code>mc2_timer_read</code></td>
</tr>
<tr>
<td><code>timer_set</code></td>
<td><code>mc2_timer_set</code></td>
</tr>
<tr>
<td><code>timer_enable_int</code></td>
<td><code>mc2_timer_enable_int</code></td>
</tr>
<tr>
<td><code>timer_disable_int</code></td>
<td><code>mc2_timer_disable_int</code></td>
</tr>
<tr>
<td><code>timer_raise_int</code></td>
<td><code>mc2_timer_raise_int</code></td>
</tr>
</tbody>
</table>

Table 12.2 One to one mapping between timer functions.

![Plug-in timer model for the ST20-C1](image)

Figure 12.1 Plug-in timer model for the ST20-C1
12.1 Building the plug-in module.

A plug-in module is currently provided for the ST20-MC2 evaluation board in source code format, this allows the developer to customize the plug-in module. An object file should be produced from the source file which can then be linked into the final application.

The plug-in module for the ST20-MC2 evaluation board is available in the examples/os20/c1timer directory. This contains the following files:

- mc2timer.c - plug-in timer
- mc2timer.h - plug-in timer header file
- user.c - example user application
- system.cfg - configuration command file

There are two ways of building the plug-in module.

Method 1

The plug-in module can be built separately to STLite/OS20 and linked in at the command line, when the application is built:

```
st20cc -T system.cfg -p mb157_cfg user.c mc2timer.c -T os20.cfg -o user.lku
st20run -t c1hw user.lku
```

Method 2

The plug-in module can be built and linked into the STLite/OS20 library so that the plug-in module is picked up automatically.

```
st20cc -cl -c mc2timer.c
st20libr os20.lib mc2timer.tco -o newos20.lib
st20cc -T system.cfg -p mb157_cfg newos20.lib user.c
st20run -t c1hw user.lku
```

All header files required by the plug-in module are available in the STLite/OS20 header directory, no extra header files will be required. Any changes to the module can be made to the source code itself, see Appendix A for more details.

For the plug-in module to “plug” itself in to STLite/OS20 the developer will need to call the module initialization function within the application file. Provided below is an example showing how the module is initialized.

```
assert( kernel_initialize() == 0);
assert( kernel_start() == 0);

interrupt_controller_init(..);
assert( mc2_timer_initialize() == 0);

time_now();
```
Please note that the interrupt controller must be initialized before the call to `mc2_timer_initialize` otherwise the call will fail. A run-time error can occur as a result of `mc2_timer_initialize` failing or being omitted all together, this will result in a string of the form “timer device driver not installed” being printed.
13 ST20-C2 specific features

Additional features:

- **Channels**
  The ST20-C2 support a point-to-point unidirectional communications channel, which can be used for communication between tasks on the same processor, and with hardware peripherals on the ST20.

- **High priority processes**
  High priority processes run outside of the normal STLite/OS20 scheduling regime, using the ST20’s hardware scheduler. A high priority process is created using the `task_create` or `task_init` functions and specifying the `task_flags_high_priority_process` flag. High priority processes will always pre-empt normal STLite/OS20 tasks (irrespective of the task’s priority), and as this takes advantage of the ST20’s hardware scheduler, high priority processes can respond faster than a normal STLite/OS20 task.

  In general, high priority processes should be regarded as the equivalent of interrupt handlers for those peripherals which have a channel style interface.

  However, because high priority processes run outside of the STLite/OS20 scheduling regime, they only have very limited access to STLite/OS20 library functions. In general they can only call functions which are implemented directly in hardware, in particular this means they can only use channels and FIFO based semaphores, not priority based semaphores or message queues.

- **Two dimensional block move**
  A number of instructions are provided which allow two dimensional blocks or memory to be moved efficiently. This is especially useful in graphical applications.
13.1 Channels

STLite/OS20 supports the use of channels by all tasks (both normal i.e. low and high priority).

Channels are a way of transferring data from one task to another, and they also provide a way of synchronizing the actions of tasks. If one task needs to wait for another to reach a particular state, then a channel is a suitable way of ensuring that happens.

If one task is sending and one receiving on the same channel then whichever tries to communicate first will wait until the other communicates. Then the data will be copied from the memory of the sending task to the memory of the receiving task and both tasks will then continue. If only one task attempts to communicate then it will wait forever.

A channel communicates in one direction, so if two tasks need bidirectional communication, then two channels are needed, one in each direction. Any data can be passed down a channel, but the user must ensure that the tasks agree a protocol in order to interpret the data correctly.

It is the responsibility of the programmer to ensure that:

- data sent by one task is received by another;
- there is never more than one task sending on one channel;
- there is never more than one task receiving on one channel;
- the amount of data sent and received are the same;
- the type of data sent and received are the same.

If any of these rules are broken then the effect is not defined.

Channels between tasks are created by using the data structure chan_t and initializing it by calling a library function. Channel input and output functions are then used to pass data. Separate functions exist for input and output and the two must be paired for communication between two tasks to take place. The header file chan.h declares the chan_t data type and channel library functions.

If one task has exclusive access to a particular resource and acts as a server for the other tasks, then channels can also act as a queuing mechanism for the server to wait for the next of several possible inputs and handle them in turn.

A channel used to communicate between two tasks on the same processor is known as a ‘soft channel’. A channel used to communicate with a hardware peripheral is known as a ‘hard channel’.

When the STLite/OS20 scheduler is enabled (by calling kernel_start), channel communication will result in traps to the kernel, which will ensure that correct scheduling semantics are maintained.
13 ST20-C2 specific features

13.1.1 Creating a channel

STLite/OS20 refers to channels using a chan_t structure. This needs to be initialized before it can be used, by using one of the following functions:

```c
chan_t *chan_create(void)
chan_t *chan_create_address(void *address)
chan_init(chan_t *chan);
void chan_init_address(chan_t *chan, void *address);
```

The _create versions allocate memory for the data structure from the system partition and initialize the channel to their default state. chan_create creates a ‘soft’ channel, chan_create_address creates a ‘hard’ channel.

The _init versions also initialize a channel, but the allocation of memory for chan_t is left to the user. chan_init initializes a ‘soft’ channel and chan_init_address initializes a ‘hard’ channel:

For example:

```c
#include <chan.h>
/* Initialize a soft channel */
chan_t soft_chan;
chan_init(&soft_chan);

/* Initialize a hard channel to link 0 input channel */
chan_t chan0;
chan_init_address(&chan0, (void*)0x80000010);
```

13.1.2 Communications over channels

Once a channel has been initialized, there are several functions available for communications:

```c
void chan_in(chan_t *chan, void* cp, int count);
void chan_out(chan_t *chan, const void* cp, int count);
int chan_in_int(chan_t *chan);
void chan_out_int(chan_t *chan, int data);
char chan_in_char(chan_t *chan);
void chan_out_char(chan_t *chan, char data);
```

These functions will transfer a block of data (chan_in and chan_out), an integer (chan_in_int and chan_out_int) or a character (chan_in_char and chan_out_char).

Each call of one these functions represents a single communication. The task will not continue until the transfer is complete.

Care needs to be taken to ensure that data is only transferred in one direction across the channel, and that the sending and receiving data is the same length, as this is not checked for at run time.
For example, the following code will use channel my_chan to send a character followed by an integer followed by a string:

```c
#include <chan.h>
char ch1;
int n1;

chan_out_char (my_chan, ch1);
chan_out_int (my_chan, n1);
chan_out (my_chan, "Hello", 5);
```

To receive this data on channel my_chan, the following code could be used:

```c
#include <chan.h>
char ch, buffer[5];
int n;

ch = chan_in_char (my_chan);
n = chan_in_int (my_chan);
chan_in (my_chan, buffer, 5);
```

### 13.1.3 Reading from several channels

There are many cases where a receiving task needs to listen to several channels and wishes to detect which one has data ready first. The ST20-C2 micro-kernel provides a mechanism to handle this situation called an alternative input. This is implemented in STLite/OS20 by the following function:

```c
int chan_alt(chan_t ** chanlist, int nchans, const clock_t *timeout);
```

*chan_alt* takes as parameters an array of channel pointers, and a count of the number of elements in the array. It returns the index of the selected channel, starting at zero for the first channel. The selected channel may then be read, using the input functions described above in section 13.1.2. Any channels that become ready and are not read will continue to wait. In addition an optional timeout may be provided, which allows *chan_alt* to be used in a polling mode, or wait until a specified time before returning, whether a channel has become ready for reading or not. Timeouts for channels are implemented using hardware and so do not increase the application's code size.

Normally *chan_alt* will be used with the time-out value *TIMEOUT_INFINITY*, in which case only one of the channels becoming ready (i.e. one of the sending tasks is trying to send) will cause it to return. When one or more channels are ready then one will be selected. If no channel becomes ready then the function will wait for ever. **Note:** that the header file *ostime.h* must be included when using this function.

To input from an array of channels, the returned index can be used as an index into the channel array, for example:

```c
#include <chan.h>
#include <ostime.h>
```
#define NUM_CHANS 5
chan_t *data_chan[NUM_CHANS];
int selected, x;
...

selected = chan_alt(data_chan, NUM_CHANS, TIMEOUT_INFINITY);
x = chan_in_int(data[selected]);
deal_with_data (x, selected);

`chan_alt` is implemented so that it does not poll while it is waiting, but is woken by one of the input channels becoming ready. This means that the processor is free to perform other processing while the task is waiting.

When it is necessary to poll channels, this can be performed by specifying a timeout of `TIMEOUT_IMMEDIATE`. This will cause the function to perform a single poll of the channels to identify whether any channel is ready. If no channel is ready then it returns -1.

Polling channels is inefficient and should only be used when there is a significant interval between polls, since otherwise the processor can be occupied entirely with polling. Polling is usually only used when a task is performing some regular or ongoing task and occasionally needs to poll one or more input channels for control signals or feedback.

Finally, it is also possible to specify that `chan_alt` should only wait until a specified time before returning, even if none of the specified channel has become ready for input. If the list consists of only one channel then this becomes a time-out for a single channel input. If no channel becomes ready before the clock reaches the given time, then the function returns and the task continues execution.

When used in this way `chan_alt` returns on the occurrence of the earlier of either an input becoming ready on any of the channels or the time. The time given is an absolute time which is compared with the timer for the current priority.

The value -1 is returned if the time expires with no channel becoming ready. If a channel becomes ready before the time then the index of the channel in the list (starting from 0) is returned.

For example, the following code imposes a time out of `wait` ticks when reading from a single channel `chan`:

```c
#include <ostime.h>
#include <chan.h>
int time_out_time, selected, x;

time_out_time = time_plus (time_now (), wait);
selected = chan_alt (&chan, 1, &time_out_time);
```
switch (selected)
{
    case 0:          /* channel input successful */
        x = chan_in_int (chan);
        deal_with_data (x);
        break;
    case -1:          /* channel input timed out */
        deal_with_time_out ();
        break;
    default:
        error_handler ();
        break;
}

The use of timers is described in Chapter 8.

13.1.4 Deleting channels

Channels may be deleted using channel_delete, see the function description in section 13.1.6, for full details.
13.1.5 Channel header file: `chan.h`

All the definitions related to ST20-C2 channel specific functions are in the single header file, `chan.h`, see Table 13.1 and Table 13.2.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/ HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan_alt</td>
<td>Waits for input on one of a number of channels</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_create</td>
<td>Create a soft channel.</td>
<td></td>
</tr>
<tr>
<td>chan_create_address</td>
<td>Create a hard channel.</td>
<td></td>
</tr>
<tr>
<td>Chan_delete</td>
<td>Delete a channel.</td>
<td></td>
</tr>
<tr>
<td>chan_in</td>
<td>Input data from a channel.</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_in_char</td>
<td>Input character from a channel</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_in_int</td>
<td>Input integer from a channel</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_init</td>
<td>Initialize a channel</td>
<td></td>
</tr>
<tr>
<td>chan_init_address</td>
<td>Initialize a hardware channel</td>
<td></td>
</tr>
<tr>
<td>chan_out</td>
<td>Output data to a channel.</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_out_char</td>
<td>Output character to a channel</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_out_int</td>
<td>Output integer to a channel</td>
<td>HPP</td>
</tr>
<tr>
<td>chan_reset</td>
<td>Reset channel.</td>
<td></td>
</tr>
</tbody>
</table>

Table 13.1 Functions defined in `chan.h`

All functions are callable from an STLite/OS20 task. Functions in Table 13.1 which are marked with ‘ISR’ can also be called from an interrupt service routine, those marked with ‘HPP’ can be called from a high priority process on an ST20-C2 processor.

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan_t</td>
<td>A channel</td>
</tr>
</tbody>
</table>

Table 13.2 Types defined in `chan.h`
13.1 Channels

13.1.6 Channel function definitions

**chan_alt** Waits for input on one of a number of channels.

**Synopsis:**

```c
#include <chan.h>
#include <ostime.h>

int chan_alt(
    chan_t ** chanlist,
    int nchans,
    const clock_t *timeout);
```

**Arguments:**

- `chan_t **chanlist` Pointer to a list of channels.
- `int nchans` The number of channels in `chanlist`.
- `const clock_t *timeout` Maximum time to wait for input from a channel.

**Results:**

Returns an index into `chanlist` for the ready channel, or -1 if the timeout expires.

**Errors:**

None.

**Description:**

`chan_alt` blocks the calling task until one of the channel arguments is ready to receive input, or the time-out expires. The index returned for the ready channel is an integer which is the index into the `chanlist` array, or -1 if the time-out occurred. `chan_alt` only returns when a channel is ready to receive input, it does not perform the input operation, which must be done by the code following the call to `chan_alt`.

The channels are considered in the order they appear in the list. The first channel in the list, which is ready, will be returned.

`timeout` is a pointer to the time-out value. If this time is reached then the function will return the value -1.

The timeout value may be specified in ticks, which is an implementation dependent quantity. Two special values can be specified for `timeout`: `TIMEOUT_IMMEDIATE` indicates that the function should return immediately, even if no channels are ready, and `TIMEOUT_INFINITY` indicates that the function should ignore the timeout period, and only return when a channel becomes ready.
Example:
/* select from one of two channels with a ten second timeout */

#include <chan.h>
#include <ostime.h>

chan_t c1, c2;
chan_t *chanlist[2];
int i;
clock_t timeout = time_plus(time_now(), CLOCKS_PER_SEC * 10);

/* initialize all the channels */

chanlist[0] = c1;
chanlist[1] = c2;

i = chan_alt(chanlist, 2, &timeout);
switch(i)
{
    case 0: /* c1 selected */
        /* consume input from c1 */
        break;
    case 1: /* c2 selected */
        /* consume input from c2 */
        break;
    case -1: /* timeout occurred */
        /* handle timeout */
        break;
}

See also:

chan_in
**13.1 Channels**

**chan_create**

Create a soft channel.

**Synopsis:**

```c
#include <chan.h>
chan_t *chan_create(void)
```

**Arguments:**

None.

**Results:**

The address of an initialized channel or NULL if an error occurs.

**Errors:**

Returns NULL if there is insufficient memory for the channel.

**Description:**

This function creates a soft channel and initializes it to its default state. The memory for the channel structure is allocated from the system memory partition, and the address of the channel is returned. The result can then be used by any of the channel input/output functions i.e. `chan_alt`, `chan_in`, `chan_in_char`, `chan_in_int`, `chan_out`, `chan_out_char` or `chan_out_int`.

A soft channel is one used to communicate between two tasks running on the same processor.

**See also:**

`chan_create_address` `chan_delete` `chan_init` `chan_init_address`
chan_create_address

Create a hard channel.

Synopsis:

```
#include <chan.h>
chan_t *chan_create_address(void *address)
```

Arguments:

- `void *address` The address of the hardware channel.

Results:

The address of an initialized channel or NULL if an error occurs.

Errors:

NULL if there is insufficient memory for the channel.

Description:

This function creates a channel which uses the hardware channel specified by `address` to communicate with a peripheral device. The `chan_t` structure is allocated from the system partition.

See also:

- `chan_create`
- `chan_init`
- `chan_init_address`
- `chan_delete`
chan_delete

Delete a channel.

Synopsis:

```c
#include <chan.h>
void chan_delete(chan_t *chan)
```

Arguments:

- `chan_t *chan` Channel to delete.

Results:

None.

Errors:

None.

Description:

This function allows a channel to be deleted. If the channel was created using `chan_create` or `chan_create_address` this function will free the memory used by the channel. If the channel was created using the `chan_init` or `chan_init_address` functions then the user is responsible for freeing the channel data structure (`chan_t`).

**Note:** that if any tasks are waiting on the channel when it is deleted, this will cause the following fatal error to be reported:

```
delete handler- operation on deleted object attempted
```

Similarly any attempt to use the deleted channel will report the same error.

See also:

- `chan_init`
- `chan_init_address`
- `chan_create`
- `chan_create_address`
**chan_in**

Input data from a channel.

**Synopsis:**

```
#include <chan.h>

int chan_in(chan_t *chan, void* cp, int count);
```

**Arguments:**

- `chan_t *chan`: A pointer to the input channel.
- `void* cp`: A pointer to where the data will be stored.
- `int count`: The number of bytes of data.

**Results:**

Always return 0.

**Errors:**

None.

**Description:**

Inputs `count` bytes of data on the specified channel and stores them in the array pointed to by `cp`.

**See also:**

- `chan_init`
- `chan_out`
13.1 Channels

chan_in_char
Input character from a channel.

Synopsis:
```c
#include <chan.h>
char chan_in_char(chan_t *chan);
```

Arguments:
- ```chan_t *chan``` A pointer to the input channel.

Results:
- Returns the input character.

Errors:
- None.

Description:
Inputs a single character on the specified channel and returns it.

See also:
chan_init chan_in chan_out_char
**chan_in_int**  
Input integer from a channel.

**Synopsis:**

```c
#include <chan.h>
int chan_in_int(chan_t *chan);
```

**Arguments:**

- `chan_t *chan`  
  A pointer to the input channel.

**Results:**

- Returns the input integer.

**Errors:**

- None.

**Description:**

- Inputs a single integer on the specified channel and returns it.

**See also:**

- `chan_init`  
- `chan_in_char`  
- `chan_out_int`
**chan_init** Initialize a soft channel.

**Synopsis:**
```
#include <chan.h>
void chan_init(chan_t *chan);
```

**Arguments:**

- `chan_t *chan` A pointer to the channel.

**Results:**
Returns no results.

**Errors:**
None.

**Description:**
Initializes the channel pointed to by `chan` to its default state. This function must be used to initialize a soft channel before it can be used by any of the channel input/output functions i.e. `chan_alt`, `chan_in`, `chan_in_char`, `chan_in_int`, `chan_out`, `chan_out_char` or `chan_out_int`.

A soft channel is one used to communicate between two tasks running on the same processor.

**See also:**

- `chan_create`
- `chan_create_address`
- `chan_delete`
- `chan_init_address`
**chan_init_address**

Initialize a hard channel.

**Synopsis:**

```c
#include <chan.h>
void chan_init_address(chan_t *chan, void *address);
```

**Arguments:**

- `chan_t *chan`: A pointer to the channel.
- `void *address`: The address of the hard channel.

**Results:**

Returns no results.

**Errors:**

None.

**Description:**

Initializes the channel pointed to by `chan` to point to the specified hardware channel at address `address`. This function must be used to initialize a hard channel before it can be used by any of the channel input/output functions i.e. `chan_alt`, `chan_in`, `chan_in_char`, `chan_in_int`, `chan_out`, `chan_out_char` or `chan_out_int`.

A hard channel is one used to communicate with a peripheral device.

**See also:**

- `chan_create`
- `chan_create_address`
- `chan_delete`
- `chan_init`
13.1 Channels

chan_out

Output data to a channel.

Synopsis:

```c
#include <chan.h>

void chan_out(chan_t *chan, const void* cp, int count);
```

Arguments:

- `chan_t *chan`: A pointer to the output channel.
- `const void* cp`: A pointer to where the data will be read.
- `int count`: The number of bytes of data.

Results:

Returns no results.

Errors:

None.

Description:

Outputs `count` bytes of data on the specified channel from the array pointed to by `cp`.

See also:

- `chan_in`
- `chan_init`
chan_out_char

Output character to a channel.

Synopsis:

```c
#include <chan.h>
void chan_out_char(chan_t *chan, char data);
```

Arguments:

- `chan_t *chan`: A pointer to the input channel.
- `char data`: The character to be output.

Results:

None.

Errors:

None.

Description:

Outputs a single character on the specified channel.

See also:

- `chan_init`
- `chan_in_char`
- `chan_out_int`
**13.1 Channels**

**chan_out_int**  
Output integer to a channel.

**Synopsis:**
```
#include <chan.h>
void chan_out_int(chan_t *chan, int data);
```

**Arguments:**
- `chan_t *chan`  
  A pointer to the input channel.
- `int data`  
  The integer to be output.

**Results:**
None.

**Errors:**
None.

**Description:**
Outputs a single integer on the specified channel.

**See also:**
- `chan_init`
- `chan_in_int`
- `chan_out_char`
**chan_reset**

Reset a channel.

**Synopsis:**

```c
#include <chan.h>
void* chan_reset(chan_t *chan);
```

**Arguments:**

- `chan_t *chan` A pointer to the channel.

**Results:**

The workspace descriptor of the process which was waiting on the channel, or `NotProcess.p` (0x80000000) if the channel was idle.

**Errors:**

None.

**Description:**

Performs a *resetch* operation on the channel. This returns the channel to the idle state. If the channel describes a hardware channel, then the link hardware will be reset. `chan_reset` returns the contents of the channel word prior to the operation.

**See also:**

- `chan_create`
- `chan_create_address`
- `chan_init`
- `chan_init_address`
13.2 Two dimensional block move support

Graphical applications often require the movement of two dimensional blocks of data to perform windowing, overlaying etc. The ST20-C2 contains instructions to perform efficient copying, overlaying and clipping of graphics data based on byte sized pixels.

A two dimensional array can be implemented by storing rows adjacently in memory. Given any two two-dimensional arrays implemented in this way, the instructions provided can copy a section (a block) of one array to a specified address in the other.

![Diagram of two dimensional block move]

Figure 13.1 Two dimensional block move

To perform a two dimensional move, 6 parameters are required (see Figure 13.1), these are:

- The address of the first element of the source block to be copied
  – this is called the source address.
- The address of the first element of the destination block
  – this is called the destination address.
- The number of bytes in each row in the block to be copied
  – this is called the width of the block.
- The number of rows in the block to be copied
  – this is called the length of the block.
- The number of bytes in each row in the source array
  – this is called the source stride.
- The number of bytes in each row in the destination array
  – this is called the destination stride.

The two stride values are needed to allow a block to be copied from part of one array to another array where the arrays can be of differing size.
None of the two dimensional moves has any effect if either the width or length of the block to copy is equal to zero. Also a two dimensional block move only makes sense if the source stride and destination stride are both greater or equal to the width of the block being moved. The effect of the two dimensional moves is undefined if the source and destination blocks overlap.

Instructions are provided which allow a whole block to be moved, or only the zero or non-zero values.

STLite/OS20 provides three functions which give access to these instructions:

```c
void move2d_all(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);

void move2d_non_zero(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);

void move2d_zero(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);
```

where:

- `move2_all` copies the whole of the block of `nrows` rows each of `length` bytes from the source to the destination.
- `move2d_non_zero` copies the non zero bytes in the block leaving the bytes in the destination corresponding to the zero bytes in the source unchanged. This can be used to overlay a non rectangular picture onto another picture.
- `move2d_zero` copies the zero bytes in the block leaving the bytes in the destination corresponding to the non zero bytes in the source unchanged. This can be used to mask out a non rectangular shape from a picture.

### 13.2.1 Two dimensional block move header file: `move2d.h`

All the definitions related to ST20-C2 two dimensional block move specific functions are in the single header file, `move2d.h`, see Table 13.3.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Callable from ISR/HPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>move2d_all</td>
<td>Two dimensional block move.</td>
<td>HPP</td>
</tr>
<tr>
<td>move2d_non_zero</td>
<td>Two dimensional block move of non-zero bytes.</td>
<td>HPP</td>
</tr>
<tr>
<td>move2d_zero</td>
<td>Two dimensional block move of zero bytes.</td>
<td>HPP</td>
</tr>
</tbody>
</table>

Table 13.3 Functions defined in `move2d.h`

All functions are callable from an STLite/OS20 task or a high priority process (HPP), however, none of them can be called from an interrupt service routine.
13.2 Two dimensional block move support

13.2.2 Two dimensional block move function definitions

**move2d_all**

Two dimensional block move.

**Synopsis:**

```c
#include <move2d.h>
void move2d_all(const void *src, void *dst,  
                int width, int nrows,  
                int srcwidth, int dstwidth);
```

**Arguments:**

- `const void *src`: Source address for the block move.
- `void *dst`: Destination address for the block move.
- `int width`: Width in bytes of each row to be copied.
- `int nrows`: Number of rows to be copied.
- `int srcwidth`: Stride of the source array in bytes.
- `int dstwidth`: Stride of the destination array in bytes.

**Results:**

None.

**Errors:**

The effect of the block move is undefined if either `width` or `nrows` is negative.

The effect of the block move is undefined if the source and destination blocks overlap.

The block move only makes sense if `srcwidth` and `dstwidth` are greater or equal to `width`.

**Description:**

`move2d_all` copies the whole of the block of `nrows` each of `width` bytes from `src` to `dst`. Each row of `src` is of `width` `srcwidth` bytes; and each row of `dst` is of `width` `dstwidth` bytes.

If either `width` or `nrows` are zero, the two dimensional block move has no effect.

**See also:**

`move2d_non_zero` `move2d_zero`
move2d_non_zero  Two dimensional block move of non-zero bytes.

Synopsis:

#include <move2d.h>

void move2d_non_zero(const void *src, void *dst,
    int width, int nrows,
    int srcwidth, int dstwidth);

Arguments:

const void *src  Source address for the block move.
void *dst  Destination address for the block move.
int width  Width in bytes of each row to be copied.
int nrows  Number of rows to be copied.
int srcwidth  Stride of the source array in bytes.
int dstwidth  Stride of the destination array in bytes.

Results:

None.

Errors:

The effect of the block move is undefined if either width or nrows is negative.
The effect of the block move is undefined if the source and destination blocks overlap.
The block move only makes sense if srcwidth and dstwidth are greater or equal to width.

Description:

move2d_non_zero copies a two dimensional block of memory, copying all the non-zero bytes from the source block to the destination, leaving the bytes in the destination corresponding to the zero bytes in the source unchanged.

move2d_non_zero copies the block of nrows each of width bytes from src to dst. Each row of src is of width srcwidth bytes; and each row of dst is of width dstwidth bytes.

If either width or nrows are zero, the two dimensional block move has no effect.

See also:

move2d_all  move2d_zero
move2d_zero

Two dimensional block move of zero bytes.

Synopsis:

```c
#include <move2d.h>
void move2d_zero(const void *src, void *dst,
     int width, int nrows,
     int srcwidth, int dstwidth);
```

Arguments:

- `const void *src`: Source address for the block move.
- `void *dst`: Destination address for the block move.
- `int width`: Width in bytes of each row to be copied.
- `int nrows`: Number of rows to be copied.
- `int srcwidth`: Stride of the source array in bytes.
- `int dstwidth`: Stride of the destination array in bytes.

Results:

None.

Errors:

The effect of the block move is undefined if either `width` or `nrows` is negative.

The effect of the block move is undefined if the source and destination blocks overlap.

The block move only makes sense if `srcwidth` and `dstwidth` are greater or equal to `width`.

Description:

move2d_zero copies a two dimensional block of memory, copying all the zero bytes from the source block to the destination, leaving the bytes in the destination corresponding to the non-zero bytes in the source unchanged.

move2d_zero copies the block of `nrows` each of `width` bytes from `src` to `dst`. Each row of `src` is of width `srcwidth` bytes; and each row of `dst` is of width `dstwidth` bytes.

If either `width` or `nrows` are zero, the two dimensional block move has no effect.

See also:

move2d_all  move2d_non_zero
Appendices
A ST20-MC2 plug-in timer module

STLite/OS20 has a subset of its API which requires the use of a programmable timer of some form. The ST20-C2 microcore has an on-chip timer available, however, the ST20-C1 microcore does not.

Therefore, the concept of a plug-in module has been designed to allow STLite/OS20 to use any type of programmable timer just as long as there is a timer module available for that specific timer. The module is similar to a device driver in a simplified form.

The plug-in module is to be exclusively used by STLite/OS20 and should not be called directly called by the developer.

Provided with STLite/OS20 is a plug-in module developed for use with the ST20-MC2 evaluation board. This module allows the application designer to take advantage of all of the STLite/OS20 API as opposed to just a subset.

Table A.1 shows the complete low-level API which is available for use by STLite/OS20, the kernel will be made aware of these functions upon the initialization of the timer module (mc2_timer_initialize).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc2_timer_enable_int</td>
<td>Enable programmable timer interrupt.</td>
</tr>
<tr>
<td>mc2_timer_disable_int</td>
<td>Disable programmable timer interrupt.</td>
</tr>
<tr>
<td>mc2_timer_raise_int</td>
<td>Raise programmable timer interrupt.</td>
</tr>
<tr>
<td>mc2_timer_read</td>
<td>Read current time from programmable timer.</td>
</tr>
<tr>
<td>mc2_timer_set</td>
<td>Set programmable timer to cause an event to occur at a user specified time.</td>
</tr>
</tbody>
</table>

Table A.1 ST20-MC2 low-level API used by STLite/OS20

Table 1.2 shows the functions which are not part of the plug-in API but are used for initializing or event capturing.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc2_timer_interrupt</td>
<td>Timer interrupt handler.</td>
</tr>
<tr>
<td>mc2_timer_initialize</td>
<td>Initialize the plug in module.</td>
</tr>
</tbody>
</table>

Table A.2 Support functions to the plug-in API.

A.1 Overview

The plug-in module can be added to any application code by simply compiling and linking in the module into the developer's application. Finally the application should call the initialization function (mc2_timer_initialize) which will initialize the programmable timer and make the links between the plug-in module and STLite/OS20. Any STLite/OS20 time dependent functions will then make use of the plug-in module.
A.2 PWM peripheral

The ST20-MC2 plug-in module uses the PWM peripheral on the ST20-MC2 evaluation board as a clock. The PWM peripheral has a programmable timer which can be programmed to cause an interrupt at a user specified time.

The register used to store the user specified time is called the Compare register, the timer keeps track of time with a 32 bit counter called the CaptureCount register.

When the value in the CaptureCount register becomes equal to the value in the Compare register an event occurs in the form of an interrupt.

Table 1.3 provides a list of registers which are actively used by the plug-in module.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>Used to initialize PWM peripheral.</td>
</tr>
<tr>
<td>InterruptEnable</td>
<td>Enable and disable interrupts by this register.</td>
</tr>
<tr>
<td>CaptureCount</td>
<td>32-bit counter.</td>
</tr>
<tr>
<td>Compare</td>
<td>Time at which an event should occur.</td>
</tr>
</tbody>
</table>

Table A.3 PWM registers used by the plug-in module.

A.2.1 Programmable timer registers

Control

The Control register contains a Capture enable bit and a Capture prescale value. The Capture enable bit must be set before the programmable timer can be used. The prescale value by default is set to 0.

InterruptEnable

The InterruptEnable register contains a bit which when set will cause an interrupt to be asserted upon the event of the CaptureCount register becoming equal to the Compare register.

CaptureCount

The CaptureCount register is a 32-bit counter that is clocked by the system clock. The counter can be prescaled by the Capture prescale value stored in the Control register.

Compare

The Compare register contains the time which is used to compare against the CaptureCount register which when equal to the timer will request an interrupt depending on the state of the InterruptEnable register.
A.3 ST20-MC2 Plug-in timer functions

**mc2_timer_enable_int**

Enable programmable timer interrupt.

**Synopsis:**

```
#include <mc2timer.h>
void mc2_timer_enable_int(void);
```

**Arguments:**

None

**Results:**

None.

**Errors:**

Undetermined behavior occurs as a result of not initializing the plug-in module.

**Description:**

The programmable timer interrupt is enabled by setting the corresponding interrupt bit in the **InterruptEnable** register within the programmable timer peripheral.

**See also:**

mc2_timer_initialize mc2_timer_disable_int
mc2_timer_disable_int  Disable programmable timer interrupt.

Synopsis:

```c
#include <mc2timer.h>
void mc2_timer_disable_int(void);
```

Arguments:

None.

Results:

None.

Errors:

Undetermined behavior occurs as a result of not initializing the plug-in module.

Description:

The programmable timer interrupt is disabled by clearing the corresponding interrupt bit in the `InterruptEnable` register in the programmable timer peripheral.

See also:

```
mc2_timer_initialize mc2_timer_enable_int
```
mc2_timer_initialize

Initialize the plug-timer module.

Synopsis:

```c
#include <mc2timer.h>
int mc2_timer_initialize(void);
```

Arguments:

None.

Results:

Returns 0 if initialization was successful, otherwise -1 if initialization failed.

Errors:

An error can occur if the timer module has already been initialized or if the function was unable to install the timer interrupt. The interrupt controller must also have been initialized otherwise an error will occur.

Description:

`mc2_timer_initialize` will:

- initialize the programmer timer;
- install the event handler (`timer_interrupt`);
- call the STLite/OS20 `timer_initialize` function, passing a structure which contains addresses of all functions provided by the plug-in module, which can be used by the STLite/OS20 low-level timer API.

`mc2_timer_initialize` must be called by the application.

See also:

`kernel_initialize` `kernel_start` `interrupt_controller_init`
mc2_timer_raise_int  Force programmable timer interrupt.

Synopsis:

```
#include <mc2timer.h>
void mc2_timer_raise_int(void);
```

Arguments:
None.

Results:
None.

Errors:
Undetermined behavior occurs as a result of not initializing the plug-in module.

Description:
A programmable timer interrupt is raised by performing an `interrupt_raise` on the interrupt level used by the programmable timer. A flag is also set internally to the plug-in module which is used by the programmable timer interrupt handler so that the handler can determine the source of the interrupt request. i.e programmable timer event or software.

See also:
mc2_timer_enable_int mc2_timer_disable_int
mc2_timer_read

Synopsis:

#include <mc2timer.h>
int mc2_timer_read(void):

Arguments:

None.

Results:

The value of the 32 bit counter within the programmable timer is returned.

Errors:

If the plug-in module is not initialized then the result returned will be undefined.

Description:

This function return the value held with the CaptureCount register on the programmable timer peripheral. The return value represents the time elapsed since the programmable timer was initialized. The resolution at which time runs can be modified to suit the developers application.

See also:

mc2_timer_initialize
A.3 ST20-MC2 Plug-in timer functions

mc2_timer_set
Set timer to cause an event at a user specified time.

Synopsis:
```
#include <mc2timer.h>
void mc2_timer_set(int Time);
```

Arguments:
- `int Time` Value to set the **Compare** register to.

Results:
None.

Errors:
Undetermined behavior occurs as a result of not initializing the plug-in module.

Description:
This function writes the value `Time` to the **Compare** register within the programmable timer. As a consequence when the value in the **Capture** register becomes equal to the value in the **Compare** register a Capture Compare event occurs, this is achieved via the interrupt handler `mc2_timer_interrupt`.

See also:
- `mc2_timer_intialize`
- `mc2_timer_enable_int`
B Using RCUs with STLite/OS20

This appendix provides some guidelines on using STLite/OS20 with Relocatable Code Units (RCUs). The appendix makes several recommendations in order to avoid some potential problems.

This appendix does not discuss the loading and initialization of RCUs in general. For this please see the ‘ST20 Embedded Toolset User Manual - 72-TDS-505’. This manual also gives details of st20cc (the Embedded Toolset's compile/link driver) and the pragma ST_translate referred to in the sections which follow. The import and export commands are described in the ‘ST20 Embedded Toolset Command Language Reference Manual - 72-TDS-533’.

B.1 Recommendations

There are a number of recommendations which should be followed when using RCUs with STLite/OS20.

B.1.1 STLite/OS20 kernel

Ensure that only one copy of the kernel is linked in. Many of the standard C run time libraries can be linked into both the main application code, and the RCU, with no problems. For simple functions, for example strlen, this may even be desirable, to reduce the function call overhead at the expense of code size.

However, for STLite/OS20 it is vital that there is only one copy of the kernel running on the processor. The easiest way to do this is to ensure that the application code (which is run when the processor is booted and loads any RCUs) also starts STLite/OS20. Any RCUs should only ever import STLite/OS20 functions from the application.

Note: that the st20cc -runtime os20 option can still be used with the st20cc -rcu or -dl options. The linker scripts are written so that when used with an RCU, run time libraries are never linked in.

B.1.2 Passing function pointers

Problems can occur when passing function pointers between RCUs and the application. This is because while the pointer is valid in all cases, it needs to be called with the correct static link. In particular this is a problem with some STLite/OS20 functions which take a function pointer as an argument. To overcome this a number of additional functions have been provided, which take a static link as well as function pointer:

```
task_init_sl
task_create_sl
```

These are all described in the appropriate chapters of the manual (Chapter 5 and Chapter 9).
B.2 Tips and Tricks

B.1.3 STLte/OS20 functions
All STLte/OS20 functions are declared in header files with a `#pragma ST_translate`. Normally this is transparent to the user, however, this does become visible when using `import` and `export` commands in a linker configuration file, as the name seen by the linker will have a ‘%os’ suffix.

To make functions of this type visible to other code units, the `-translate` option should be used on the `import` and `export` commands. For example:

```
import task_priority -translate task_priority%os
```

B.1.4 Hidden functions
As well as all the ‘user visible’ STLte/OS20 functions, there is a hidden function which is used by the `TIMEOUT_IMMEDIATE` and `TIMEOUT_INFINITY` macros. This is `time_pointers()`, which is used to ensure that the same values are used for these pointers in all code units. This function must also be imported by any RCUs which use these macros:

```
import time_pointers -translate time_pointers%os
```

B.1.5 Using the instruction and data caches

When either of the ST20 instruction or data caches are enabled extra care must be taken when handling relocatable code units. If the area of memory that the RCU resides in is cached then it is important to maintain cache coherency.

This is especially important when replacing one RCU with another. In this situation the following technique should be used. After calling `rcu_deinit()` and there are no further accesses to the RCU’s memory, the instruction cache should be invalidated and the data cache should be flushed. This purges any data or instructions belonging to the old RCU from the cache. The new RCU can then be loaded safely. See section 11.6 for further details.

B.2 Tips and Tricks

B.2.1 Code incompatibility

One situation that can be encountered when using RCUs is that it is sometimes useful to write code that can be compiled to run in both the main application and the loaded RCU, but where a different function may need to be called depending on whether it is called from the main application or the RCU.

This can occur when using STLte/OS20 because the application can call functions which only take a function pointer. If the same functionality is needed from the RCU a

---

1. This is to ensure that even when the STLte/OS20 library is linked in, it is still possible to define functions with the same name in the user’s code, by not including the STLte/OS20 header files.
static link needs to be specified as well. Thus it is difficult to write code which can be
used in both cases.

One solution to this is to use the -translate option of the export command. In this
way the name of the function called by the RCU code does not have to have the same
name in the main application. For example, if the RCU uses the command:

    import application_fn

the RCU will be built with a stub function called application_fn(), which calls
through a function pointer, and which will be initialized by the dynamic link library
dl.lib to point to the function which is then exported under the name
"application_fn".

However, the application can export this function using the command:

    export application_fn -translate application_fn_rcu

which exports the function application_fn_rcu(), but with the name
"application_fn".

Thus, when the RCU is loaded, the name “application_fn” exported by the appli-
cation will be matched with the name “application_fn” imported by the RCU.
When the RCU calls the stub application_fn(), the function pointer will point to
the function application_fn_rcu() in the application, and it is this function which
will be called. This function can either be a complete replacement for the normal
application_fn(), or it is possible to simply modify the parameters before calling
the original function.
B.2 Tips and Tricks
C Compiling and configuring

The standard STLite/OS20 kernel supplied with the ST20 toolset is preconfigured for use in a wide variety of applications. There are, however, some situations in which building a kernel that is tailored to the needs of a specific application is worthwhile.

Similarly there is a standard link process that can be tailored for a specific application using the command language.

Recompiling or reconfiguring the STLite/OS20 kernel will result in a configuration which has not been tested by STMicroelectronics and should only be done, if necessary, with due care and consideration.

C.1 Runtime configuration

In the ‘Getting started’ chapter, section 2.1.1 describes what the default actions are when st20cc -runtime os20 is used. These generic defaults may not be suitable for all applications, therefore STLite/OS20 supports configuration options to control the link process more finely. These configuration options are usually specified in a command language file that is read at link time.

For example if an application does not perform any input/output operations then initializing thread-safe stdio is not required and will cause code and data to be linked in that is never used. To suppress this initialization, the user may specify a configuration file, options.cfg, as follows:

```#
## do not install thread-safe stdio
OS20_config.initialize_stdio_protection_first=0
```

This configuration can be supplied to the linker as follows:

```
st20cc -p dxx <application>.c -runtime os20 -T options.cfg
```

A large proportion of the configuration options take a boolean argument. Boolean options use C-like truth values where any non-zero value is true.

STLite/OS20 is provided with an example configuration file: $ST20ROOT/lib/os20conf.cfg. This file lists all the STLite/OS20 configuration options together with a brief explanation and their default values. This can be copied into the application source directory and edited to suit the needs of a particular application. Note that this file is not read by STLite/OS20 at link time. Modifying it directly will not change the default behavior.

C.1.1 Specifying initialization code

By default the STLite/OS20 runtime system ensures that some initialization code runs before the user's application starts to run. By the time the user's main function is run the kernel has been started and peripherals supported by STLite/OS20 such as the interrupt and cache systems have been initialized.
C.1 Runtime configuration

It is clearly inefficient to initialize something that is never used. In addition legacy code often includes calls to the initialization functions, in this case disabling the automatic initialization means that there is no need to change the C code.

All initialization options have the following form:

```
OS20_config.initialize_ ... _first = <boolean>.
```

Their exact details are described in the example configuration file.

C.1.2 Specifying placement of code and data

By default the STLite/OS20 runtime system places some code and data into internal SRAM to increase scheduler and interrupt performance. The scheduler can disable interrupts for a relatively long period during its execution so its speed of execution has a critical effect on interrupt latency. Even taking this into account in some applications the internal SRAM is better employed storing application specific code or data. Configuration options are provided to prevent STLite/OS20 installing itself into internal SRAM.

All placement options have the following form:

```
OS20_config.place_ ... = "<memory segment>".
```

Their exact details are described in the example configuration file.

Note that on devices with only two kilobytes of internal SRAM (e.g. an STi5500 device with the data cache enabled) it is not always possible to store all the scheduler code and data in internal SRAM. The standard STLite/OS20 kernel will not present any problems, however, expanded kernels (particularly those with time logging enabled) do not fit. In this case it is recommended that the scheduler code is not placed in internal SRAM.

C.1.3 Caching peripheral memory in larger blocks

Currently this option applies only to STi5512 and STi5518 devices. These devices have configurable caches to change the range of memory addresses which are cached. This is most frequently used to permit Direct Memory Access (DMA) without causing cache coherency problems. In the memory range, 0xC0000000 to 0xC00FFFFF, the cacheable blocks which can be independently enabled or disabled, defaults to 64 kilobytes in size. This is to ensure back compatibility with the STi5510 devices.

The STi5512 and STi5518 can support larger memories than older devices. To cope with the extra memory the newer devices can cache memory in the range of 0xC0000000 to 0xC07FFFFF, in 512 kilobyte blocks. The option:

```
OS20_config.sti551x_cache_512kbyte_blocks
```

will select the larger cache blocks at link time.
C.1.4 Making devices with ILC-2 strictly backward compatible

By default the interrupt level controller ILC-2 (see section 9.1 and section 9.2) provides the same programmer interface as the interrupt level controller ILC-3 since these interrupt level controllers provide similar features. Unfortunately this means that existing application code has to be modified to make it run on the ILC-2.

The ILC-2 is designed to be hardware backward compatible with the ILC-1. This useful property allows the ILC-1 support library to be used to make the ILC-2 software backward compatible. The option to achieve this is:

\[ \text{OS20_config.interrupt\_force\_ilc1.} \]

C.2 Compiling STLite/OS20

There are a number of extensions that are not supported by the pre-built STLite/OS20 kernel. These extensions generally provide useful services but at a cost, both in speed of execution and memory footprint, that may be unacceptably high for some applications. Because some of the extensions give greater visibility of what STLite/OS20 is doing internally, the extensions are most useful when debugging, and should be used with caution in production systems.

The standard kernel also contains workarounds to silicon defects that do not affect the whole ST20 family of processors. Therefore it may be beneficial to disable a workaround if it is does not affect the target device.

The source for STLite/OS20 is located in $ST20ROOT/src/os20. Copy this directory locally and set an environment variable MYSTLITE to point to the copied directory. STLite/OS20 is provided with makefiles for Sun make under Solaris and Microsoft nmake under Windows.

From the source directory STLite/OS20 can be built under Solaris as follows:

\[ \text{make -f makefile.top c1dxx} \]

Under Windows STLite/OS20 can be built from the source directory with the command:

\[ \text{nmake /f makefile.top c1dxx-pc} \]

Finally st20cc must be directed to pick up the new libraries rather than those supplied with the toolset. Under Solaris, add the following options to the st20cc command line each time it is used:

\[ \text{st20cc -I%Mystlite/dist-cx/lib -L%Mystlite/dist-cx/lib ...} \]

Similarly for Windows:

\[ \text{st20cc -I%Mystlite/dist-cx/lib -L%Mystlite/dist-cx/lib ...} \]

See the chapter ‘st20cc compile/link tool’ in the ‘ST20 Embedded Toolset User Manual 72-TDS-505’ for details on how to make the above options permanent.
C.3 Compilation option file: conf.h

The conf.h header file is included by every source file in STLite/OS20 and is used to set the compile time options. It is located in $ST20ROOT/src/os20/include/conf.h. Every compile time option is listed in this file together with a brief description. Most options (such as silicon workarounds) are fully described in conf.h and are not mentioned in this document. There are some options that require additional explanation, these are listed here.

C.3.1 Callback Support

Callback support is enabled using the option CONF_CALLBACK_SUPPORT. This will cause STLite/OS20 to call a user supplied callback function whenever certain scheduler or interrupt events take place.

The following events can have a callback function attached to them:

- a task switch;
- when a task is initialized with task_init;
- when a task exits;
- when a task is deleted with task_delete;
- whenever a high priority process is removed from the scheduler queue (deschedules waiting for some event);
- whenever a high priority process is added to the scheduler queue (rescheduled after an event occurred);
- when an interrupt handler is installed with interrupt_install;
- when an interrupt handler is removed with interrupt_delete;
- when an interrupt handler is entered;
- when an interrupt handler exits.

The callback setup functions are described in detail at the end of this chapter, see section C.5.

Note that to increase performance the STLite/OS20 interrupt handlers can loop to service more then one interrupt without leaving the interrupt state. For this reason it is possible for the user’s interrupt handlers to run more times than the interrupt enter and exit callbacks.

C.3.2 Changing the number of priority levels

The pre-built STLite/OS20 kernel supports 16 priority levels. There are two factors which affect the number of priority levels the scheduler can support.

The scheduler code supports up to 32 priority levels by default. On ST20-C2 cores this can be extended to up to 64 priority levels using the option: CONF_PRIORITY_64.
The scheduler data structures support exactly 16 priority levels by default. There is a fixed overhead per priority level so memory overhead must be traded off against the flexibility provided by more or fewer priority levels. To change the scheduler data structure allocation, the header file `task.h` must be modified; `OS20_PRIORITY_LEVELS` can be changed from 16 to any number in the range 1 to 64.

C.3.3 Reducing interrupt latency (ST20-C2 core only)

The largest block of code for which interrupts are disabled is the scheduler trap handler, invoked by the hardware every time a context switch may be required. In many cases the interrupt latency introduced by the scheduler trap handler is acceptable. However, in some cases it may be necessary to reduce it even further.

The configuration option `BETWEEN_HIGH_AND_LOW` causes the scheduler trap handler to run with high priority interrupts still enabled. This permits high priority interrupts and high priority processes to run with near to hardware latency.

However, an unfortunate side effect of doing this is that it is now the user’s responsibility to ensure that the scheduler trap handler is not re-entered. Low priority interrupts are still disabled, so re-entrcency can only occur if a high priority interrupt or high priority process performs an operation which will generate a low priority scheduler trap.

In particular this means that some operations cannot be used from high priority interrupts or high priority processes:

- signalling a semaphore which could have a low priority task waiting on it;
- performing any channel operations where the other end of the channel is connected to a low priority task.

Communication between the high priority process and the low priority task can still be performed, as long as it is through a mechanism which will defer the communication until it is safe to enter the trap handler. The easiest way to do this is to use a low priority interrupt, which is triggered from a high priority interrupt or high priority process but does not run until the high priority interrupt or high priority process has descheduled, and the trap handler has completed (if it was executing).

C.3.4 Time Logging (ST20-C2 core only)

STLite/OS20 can be configured to maintain a record of the amount of time each task spends running on the processor using: `CONF_TIME_LOGGING`. The data collected can be accessed using the `task_status` function, see section 5.19.

To avoid changing interrupt behavior the time spent handling interrupts will be logged against the interrupted task. Obviously on a system with a very high interrupt load this is not acceptable. Therefore this behavior can be prevented by additionally logging the time spent servicing interrupts using: `CONF_INTERRUPT_TIME_LOGGING`. The extra data recorded can be accessed using the functions `interrupt_status` and `interrupt_status_number`, see section 9.18.
The ST20-C2 core has two internal timers that run at different speeds. Either timer can be used for time logging and will be selected based upon the value of `CONF_TIME_LOGGING_PRIORITY`. By default the faster (high priority) timer is used. This timer will roll-over after approximately 76 minutes.

**Note:** that all time logging is intrusive. Logging is performed by the host and may affect real-time behavior.

### C.3.5 Software Interrupts

The ST20 processor provides the useful facility to raise an interrupt level from software, this is a hardware feature and if the feature is not used then there is no software cost.

The situation is made more complex by ST20 variants that also have an interrupt level controller. When more then one interrupt number is attached to an interrupt level the STLite/OS20 interrupt handler interrogates the interrupt level controller to determine the source of interrupt. If this interrupt has been generated by software the interrupt level controller will not be able to provide this information. To support software interrupts extra code is added to one of the STLite/OS20 interrupt handlers to determine the source of the interrupt. This is only required when more than one interrupt number is attached to a single level.

Software interrupts are enabled by default as their cost is relatively low. For a well designed system disabling software interrupts will rarely be necessary because no code is added to the high performance interrupt handler. Refer to section 9.4.4 for details on efficient interrupt layout.

### C.4 Performance considerations

This section gives some hints on how to place portions of STLite/OS20 in memory to optimize performance. Normally the defaults will generate reasonably good results. However, in some circumstances it may be necessary to select where in memory certain sections should be placed, and this section gives some recommendations.

STLite/OS20 has been structured so that most of the important code exists within the scheduler trap handler. This code is responsible for all context switches and management of the kernel data structures. For this reason this code is normally executed with all interrupts disabled, and so can affect interrupt latency. Thus there are usually two objectives when trying to tune STLite/OS20 performance:

- to reduce context switch times
- to reduce interrupt latency caused by the scheduler disabling interrupts

The trap handler has been written to reduce execution time as far as possible, and so timings are now dominated by memory access times. This is why the task structures have been broken down into two components, the `task_t` structure which contains largely static information, and the `tdesc_t` which contains dynamic information, accessed on context switches, and thus allows the `tdesc_t` to be moved into on-chip memory.
There are five sections which STLite/OS20 uses:

- trap handler code (os20_th_code section)
- trap handler workspace (including many STLite/OS20 variables) (os20_th_data section)
- task tdesc_t structures
- task queues (os20_task_queue section)
- interrupt handler stacks

Three of these can be placed using the ST20 Embedded Toolset’s configuration files, using the section names indicated in brackets. The remaining two are under the user’s control. By default tdesc_t’s will be allocated from the internal_partition if task_create() is used, which is normally placed in internal memory, however, if task_init() is used then their location is completely up to the user.

Putting the trap handler code and data on chip can bring large performance gains, with fairly small usage of internal memory, and should be done if at all possible. This has been shown to give performance gains of 30%, while moving tdesc’s, queues and interrupt stacks on chips only yields an additional 9% improvement.

For the remaining three categories, the choices are not so clear. Moving task queues and tdesc’s on-chip will bring performance improvements in virtually all circumstances, however, these can be large data structures when there are lots of tasks and priorities. One option is to only place the tdesc’s of critical tasks on chip while others are still off-chip. This will improve the context switch times to those tasks which have their tdesc’s on-chip, although this will not result in the full performance gain seen with all tdesc’s on-chip, because details of the task being switched away from may have to saved.

Moving the interrupt stacks on-chip may be desirable to improve the performance of critical routines, and STLite/OS20 will also benefit from this, but this is unlikely to be possible for all interrupt stacks, and should only be considered where the interrupt handler itself needs to execute quickly, and the task response time is also important.

### C.5 Callback functions
C.5 Callback functions

**callback_ ...**

Register a callback for an event.

**Synopsis:**

```c
#include <callback.h>

callback_fn_t callback_task_switch( callback_fn_t Function);
callback_fn_t callback_task_init( callback_fn_t Function);
callback_fn_t callback_task_exit( callback_fn_t Function);
callback_fn_t callback_task_delete( callback_fn_t Function);
callback_fn_t callback_task_restart( callback_fn_t Function);
callback_fn_t callback_task_stop( callback_fn_t Function);
callback_fn_t callback_interrupt_install( callback_fn_t Function);
callback_fn_t callback_interrupt_delete( callback_fn_t Function);
callback_fn_t callback_interrupt_enter( callback_fn_t Function);
callback_fn_t callback_interrupt_exit( callback_fn_t Function);
```

**Arguments:**

- `callback_fn_t Function` Pointer to `void (*)(void)` function.

**Results:**

Point to the previously installed function.

**Errors:**

None.

**Description:**

This group of functions is used to install callback handlers to any of the supported internal STLite/OS20 events.

`task_context` can be used to determine what task/interrupt level the callback has been called for.

**Example:**

```c
#include <callback.h>

void my_callback_handler(void)
{
    static int level;
    task_context( NULL, &level );
    ...
}

callback_interrupt_enter( my_callback_handler );
```

**See also:**

`task_context`
Index

Numerics
2D block move, 217, 238–242

B
Backwards compatibility, 18, 148, 259

C
Cache, 199–212
  example, 201
cache_config_data library function, 200, 203
cache_enable_data library function, 200, 205
cache_enable_instruction library function, 200, 206
cache_flush_data library function, 201, 207
cache_init_controller library function, 200, 208
cache_invalidate_data library function, 201, 209
cache_invalidate_instruction library function, 201, 210
cache_lock library function, 200, 211
cache_status library function, 212
callback_interrupt_ ... library functions, 264
callback_task_ ... library functions, 264
chan_alt library function, 220, 224
chan_create library function, 226
chan_create_address library function, 227
chan_delete library function, 228
chan_in library function, 219, 229
chan_in_char library function, 219, 230
chan_in_init library function, 219
chan_in_int library function, 231
chan_init library function, 219, 232
chan_init_address library function, 219, 233
chan_out library function, 219, 234
chan_out_char library function, 219, 235
chan_out_int library function, 219
chan_out_int library function, 236
chan_reset library function, 237
chan_t
  data structure, 28, 218
Channel I/O, 218–237
Class, 4–6
Clocks see time, timers
Compiling/configuring
  STLite/OS20, 257–264
caching peripheral memory, 258
  initialization options, 257
  placing STLite/OS20 in memory, 258, 262
Conventions used in this manual, vi

D
Debugging. See Compiling/configuring
  STLite/OS20
device_id library function, 197
device_name library function, 198

I
Initialization
  of memory partitions for STLite/OS20, 27
Internal partition
  initialization, 12
  link error, 28
Interrupt controller, 145–150
Interrupt level controller, 145–150, 151–154, 157
interrupt_clear library function, 156, 162
interrupt_clear_number library function, 156, 163
interrupt_delete library function, 158, 164
interrupt_disable library function, 153, 154, 165
interrupt_disable_global library function, 153, 166
interrupt_disable_mask library function, 153, 154, 167
interrupt_disable_number library function, 154, 168
interrupt_enable library function, 153, 154, 169
interrupt_enable_global library function, 153, 170
interrupt_enable_mask library function, 153, 154, 171
interrupt_enable_number library function, 154, 172
interrupt_init library function, 150–152, 173
interrupt_init_controller library function, 149, 175
interrupt_install library function, 150–152, 177
interrupt_install_sl library function, 151, 179
interrupt_lock library function, 155, 181
interrupt_pending library function, 182
interrupt_pending library function, 156
interrupt_pending_number library function, 156, 183
interrupt_raise library function, 156, 184
interrupt_raise_number library function, 156, 185
interrupt_status library function, 157, 186
interrupt_status_number library function, 157, 188
interrupt_test_number library function, 156, 189
interrupt_trigger_mode_number library function, 157, 190
interrupt_uninstall library function, 158, 191
interrupt_unlock library function, 155, 192
interrupt_wakeup_number library function, 157, 193
Interrupts, 145–193
callback support, 260
changing the number of priority levels, 260
reducing latency, 261, 262
restrictions, 159

K
kernel_initialize library function, 20, 21
kernel_start library function, 22
kernel_version library function, 23

M
mc2_timer_disable_int library function, 248
mc2_timer_enable_int library function, 247
mc2_timer_initialize library function, 249
mc2_timer_raise_int library function, 250
mc2_timer_read library function, 251
mc2_timer_set library function, 252
Memory
set-up for STLite/OS20, 25–44
memory_allocate library function, 31
memory_allocate_clear library function, 32
memory_deallocate library function, 33
memory_reallocate library function, 34
Message handling
with STLite/OS20, 119–135
message_claim library function, 122, 125
message_claim_timeout library function, 122, 126, 129
message_create_queue library function, 120, 127
message_create_queue_timeout library function, 120, 128
message_delete_queue library function, 122, 129
message_hdr_t
data structure, 123
message_init_queue library function, 120, 130
message_init_queue_timeout library function, 120, 131
MESSAGE_MEMSIZE_QUEUE macro, 120
message_queue_t
data structure, 28, 123
message_receive library function, 122, 132
message_receive_timeout library function, 122, 129, 133
message_release library function, 122, 134
message_send library function, 122, 135
move2d_all library function, 239, 240
move2d_non_zero library function, 239, 241
move2d_zero library function, 239, 242

O
Objects
creating, 5
deleting, 6
os20lkucfg configuration command file, 11
os20romcfg configuration command file, 11

P
Partition
calculating size, 28
internal or system
link error, 28
partition_create_fixed library function, 35
partition_create_heap library function, 36
partition_create_simple library function, 37
partition_delete library function, 38
partition_init_fixed library function, 39
partition_init_heap library function, 27, 40
partition_init_simple library function, 27, 41
partition_status library function, 29, 42
partition_t
data structure, 28, 30
Performance consideration, 262–263
Priority
STLite/OS20 implementation, 7, 46
Programmable timer peripheral, 214

R
Real-time clocks, 137–144
Recompile/reconfigure. See Compiling/configuring
STLite/OS20
Relocatable code
STLite/OS20, 253–255
Reset, 148
Root task
STLite/OS20, 21, 48, 56
runtime os20, 11, 21, 22, 149
semaphore_create_fifo library function, 101, 106
semaphore_create_fifo_timeout library function, 101, 107
semaphore_create_priority library function, 101, 108
semaphore_create_priority_timeout library function, 101, 109
semaphore_delete library function, 110
semaphore_init_fifo library function, 101, 111
semaphore_init_fifo_timeout library function, 101, 112
semaphore_init_priority library function, 101, 113
semaphore_init_priority_timeout library function, 101, 114
semaphore_signal library function, 102, 115
semaphore_t data structure, 28, 101
semaphore_wait library function, 101, 116
semaphore_wait_timeout library function, 102, 117

Semaphores, 101–117

ST20-C2
channel communications, 217–237

Stack usage, 54
Static
link, 51

STLite/OS20 Real-time kernel

S

Static, 26
System partition
initialization, 12
link error, 28

T

task_context library function, 54, 61
task_create library function, 49, 62, 217
task_create_sl library function, 65
task_data library function, 56, 68
task_data_set library function, 56, 69
task_delay library function, 51, 70
task_delay_until library function, 51, 71
task_delete library function, 58, 72
task_exit library function, 57, 73
task_id library function, 53, 74
task_immortal library function, 53, 75
task_init library function, 49, 76, 217
task_init_sl library function, 79
task_kill library function, 53, 82
task_lock library function, 49, 84
task_mortal library function, 53, 85
task_name library function, 54, 86
task_onexit_set library function, 57, 87
task_onexit_set_sl library function, 88
task_onexit_set_t library function, 48, 89
task_priority_set library function, 48, 90
task_reschedule library function, 52, 91
task_resume library function, 53, 92
task_stack_fill library function, 55, 93
task_stack_fill_set library function, 55, 94
task_status library function, 55, 96
task_suspend library function, 52, 97
task_t

data structure, 28, 45
task_unlock library function, 49, 98
task_wait library function, 58, 99

Tasks

data, 56
killing, 53
terminating, 57
tdesc_t

data structure, 28, 45

Time

real-time clocks, 137–144
slicing, 49

ST20-C1, 47
ST20-C2, 47
time_after library function, 138, 141
time_minus library function, 138, 142
time_now library function, 138, 143
time_plus library function, 138, 144

Timers

support for ST20-c1, 213–216, 245–252
Trigger mode, 145, 157
Two dimensional block move, 217, 238–242