OS20 User Manual

ADCS 7473749D

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZE REPRESENTATIVE OF ST, ST PRODUCTS ARE NOT DESIGNED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS, WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2001, 2002, 2003, 2004, 2006 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

http://www.st.com

Contents

Preface

ST20 documentation suite

The document set provided with the toolset comprises the following documents.

OS20 User Manual (this document)

This manual is a user guide for the OS20 real-time kernel. It provides an introduction and getting started section then continues with separate chapters for each of the main features supported, such as, kernel, memory and partitions management, tasks, semaphores, message handling queues, real-time clocks and interrupts.

ST20 Embedded Toolset Delivery Manual (ADCS 7257995)

The delivery manual provides installation instructions, a summary of the release and a list of changes since the previous revision.

ST20 Embedded Toolset User Manual (ADCS 7143840)

This manual provides an overview of the toolset and a getting started guide for using the graphical user interface. It also describes how the core features of the toolset are used to build and run application programs. It includes compiling and linking, connecting to a target, loading programs and application debugging.

ST20 Embedded Toolset Reference Manual (ADCS 7250966)

This manual describes the advanced facilities of the toolset such as the assembler, librarian, lister, and ST20 instruction set simulator. It also describes facilities such as code and data placement, the stack depth and memory map files, the use of relocatable code units and profiling and trace facilities. Reference information is provided for the C and C++ implementations, the libraries and the command language.

Conventions used in this manual

Typographical conventions

The following typographical conventions are used in this manual:

Bold	Used within text for emphasis
Blue Italic	Denotes hyperlink cross-references
Italic	Denotes nonhyperlink cross-references
UPPER CASE	Denotes special terminology, for example, register or signal/pin names
`Monotype bold`	Denotes command options, command line examples, code fragments, and program listings
`Monotype bold italic`	Denotes arguments or parameters in command syntax definitions
`Monotype italic`	Denotes code comments
Braces {}	Denotes a list of optional items in command syntax
Brackets []	Denotes 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
__	Denotes two underscores together
	A change bar in the left margin indicates a change from the previous version of the manual. This may indicate a change in the functionality of the toolset or merely an updated description.

Command line conventions

Example command lines and directory path names are documented using UNIX®/Linux style notation which makes use of the forward slash (**`/`**) delimiter. In most cases this should be recognized by Windows hosts; if not, substitute the forward slash with a backslash (\). For example, the directory:

`release/examples/simple`

is the same as:

`release\examples\simple`

Command line options are prefixed by a hyphen (**`-`**); this is compatible with Linux, Windows and UNIX.

Examples of the debugging tools use the following convention to distinguish command prompts:

"**`%`**" is used to indicate the operating system command prompt, for example:

`% st20run ...`

"**`>`**" is used to indicate the interactive command language prompt, for example:

`> break ...`

Acknowledgements

Linux® is a registered trademark of Linus Torvalds.

Red Hat® is a registered trademark of Red Hat Software, Inc.

Sun and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the US and other countries.

UNIX® is a registered trademark of the The Open Group in the US and other countries.

Microsoft®, Windows® and Windows NT® are registered trademarks of Microsoft Corporation in the United States and other countries.

The C compiler implementation was developed from the Perihelion Software “C” Compiler and the Codemist Norcroft “C” Compiler.

The C++ language front-end was developed under a Licence Agreement between Edison Design Group, Inc. (EDG) and STMicroelectronics Limited.

This product incorporates innovative techniques which were developed with support from the European Commission under the ESPRIT Projects:

- P2701 PUMA (Parallel Universal Message-passing Architectures),
- P5404 GPMIMD (General Purpose Multiple Instruction Multiple Data Machines),
- P7250 TMP (Transputer Macrocell Project),
- P7267 OMI/STANDARDS,
- P6290 HAMLET (High Performance Computing for Industrial Applications),
- P606 STARLIGHT (Starlight core for Hard Disk Drive Applications).

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 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 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 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 2.10 version of OS20.

1.1 Overview

OS20 kernel features:

- a high degree of hardware integration,
- multi-priority pre-emptive scheduling based on 16 levels of priority,
- semaphores,
- message queues,
- timers,
- memory management,
- interrupt handling,
- very small memory requirement,
- context switch time of 6 µs or less,
- common across all ST20 microprocessors.

Each 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* below.
- All the symbols defined by a service have the service name as the first component of the name; see *Table 1* below.

Header	Description
c1timer.h	ST20-C1 timer plug-in functions
cache.h	Cache functions
callback.h	Callback functions
chan.h	ST20-C2 specific functions
device.h	Device information functions
interrup.h	Interrupt handling support functions
kernel.h	Kernel functions
message.h	Message handling functions
move2d.h	Two dimensional block move functions (ST20-C2 specific)
mutex.h	Mutex functions
ostime.h	Timer functions
partitio.h	Memory functions
semaphor.h	Semaphore functions
tasks.h	Task functions

Table 1: OS20 header files

1.1.1 Naming

All the functions in 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 OS20 functions into services is used throughout 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.

The remaining sections of this introductory chapter describe the main concepts on which 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 OS20 is provided in *Chapter 2: Getting started on page 9*.

Chapter 3: Kernel on page 17 describes the OS20 scheduling kernel.

Chapter 4: Memory and partitions on page 21 describes OS20 memory and partitions.

Chapter 5: Tasks on page 27 describes OS20 tasks.

Chapter 6: Semaphores on page 43 describes OS20 semaphores.

Chapter 8: Message handling on page 51 describes OS20 message handling.

Chapter 9: Real-time clocks on page 57 describes support for real-time clocks.

Chapter 10: Interrupts on page 61 describes OS20 interrupt handling.

Chapter 11: Device information on page 79 describes OS20 functions for obtaining ST20 device information.

Chapter 12: Caches on page 81 describes OS20 support for caches.

Chapter 13: ST20-C1 specific features on page 87 describes a facility for providing timer support for OS20 when run on an ST20-C1 core.

Chapter 14: ST20-C2 specific features on page 93 describes support for some ST20-C2 specific features such as channel communication, high priority processes and two dimensional block moves.

1.1.3 Related OS20 material

The *Advanced facilities* part of the *ST20 Embedded Toolset Reference Manual* contains information which is pertinent to OS20 in the chapters *Using st20run with OS20* and *Building and running relocatable code*.

1.2 Classes and objects

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 OS20, and how it has been implemented in the C language.

Each of the major services of OS20 is represented by a class, that is:

- 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 affect it and the rest of the system.

For many classes within OS20, there are different flavors. For example, the semaphore class has FIFO and priority flavors. When a particular object is created, the required flavor 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 behave may depend on the flavor of that class. So the exact behavior of **`semaphore_wait()`** depends on whether it is applied to a FIFO or priority semaphore object.

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

Caution: The internal layout of any of the structure should not be referenced directly. This changes between implementations and releases, although the size of the structure does not change.

To provide this abstraction within 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`**

Normally the ***`class`*`_create`** version of the call can be used. This allocates whatever memory is required to store the object, and returns 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`*`_init`** calls can be

used. This leaves memory allocation up to the user, allowing a completely static system to be created if required. For ***class*_init** calls, the user must provide pointers to the data structures, and OS20 uses these data structures instead of allocating them itself.

When using ***class*_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*_create** calls are made. Normally this is done automatically as described in *Chapter 2: Getting started on page 9*. *Chapter 4: Memory and partitions on page 21* 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 OS20's implementation.

When an object is no longer required, it should be deleted by calling the appropriate ***class*_delete** function. If objects are not deleted and memory is reused, then 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*_init** then it must be deleted before the function returns and the object goes out of scope.

Using the appropriate ***class*_delete** function has a number of effects.

- The object is removed from any lists within OS20, and no longer appears 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*_create** then the memory allocated for the object is freed back to the appropriate partition.

Note: The objects created using both ***class*_create** *and* ***class*_init** *are deleted using* ***class*_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 **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. OS20 supports three different types of memory partition - heap, fixed and simple - as described in *Chapter 4: Memory and partitions on page 21*. 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 ***class*_create_** versions of the library functions to create objects, OS20 allocates memory for the object. In this case 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 **create_** functions are called. This is normally performed automatically; see *Chapter 2: Getting started on page 9*.

1.4 Tasks

Tasks are the main elements of the 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 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: Tasks on page 27*.

1.5 Priority

Task scheduling is determined by priority. Normally the highest priority task is the one set to run, with all lower priority tasks paused until the highest priority task deschedules.

In some cases, when there are two or more tasks of the same priority waiting to run, they are each 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. 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.

OS20 supports the following system of priority for tasks running on an 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.
- Tasks may be created to run as high priority processes, in which case they are never timesliced and 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. OS20 allows the user to define individual task priorities, and tasks of equal priority are timesliced. High priority processes are not supported on the ST20-C1.

To implement multi-priority scheduling, OS20 uses a scheduling kernel which needs to be installed and started before any tasks are created. This is described in *Chapter 3: Kernel on page 17*. Further details of how priority is implemented is given in *Section 5.2: Implementation of priority and timeslicing on page 28*.

1.6 Semaphores

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: Semaphores on page 43*.

1.7 Message queues

Message queues provide a buffered communication method for tasks and are described in *Chapter 8: Message handling on page 51*. 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.

1.8 Clocks

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 9: Real-time clocks on page 57* provides an overview of how time is handled in OS20. Time-out related functions are described in *Chapter 5: Tasks on page 27*, *Chapter 6: Semaphores on page 43* and *Chapter 8: Message handling on page 51*.

On the ST20-C2 microprocessor, 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 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 OS20 functions. OS20 provides some example sources of such functions which the user can modify for their particular device; see *Chapter 13: ST20-C1 specific features on page 87* for details.

1.9 Interrupts

A comprehensive set of interrupt handling functions is provided by OS20 to enable external events to interrupt the current task and to gain control of the CPU. These functions are described in *Chapter 10: Interrupts on page 61*.

1.10 Device ID

Support is provided for obtaining the ID of the current device; see *Chapter 11: Device information on page 79*.

1.11 Cache

There are a number of functions available to exploit the cache support provided on ST20 devices; see *Chapter 12: Caches on page 81*.

1.12 Processor specific functions

The 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 13: ST20-C1 specific features on page 87* and *Chapter 14: ST20-C2 specific features on page 93*.

2 Getting started

This chapter describes how to start using OS20 and write a simple application. The concepts and terminology used in this chapter are introduced in *Chapter 1: Introduction on page 1*.

2.1 Building for OS20

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

```
st20cc -p STi5516MB382 -runtime os20 app.tco -o system.lku
```

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

- the 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, device-independent I/O functions or **`stdio`** functions concurrently, all operations are handled correctly.

st20cc is described in the *ST20 Embedded Toolset User Manual*, chapter *st20cc compile/link tool*; the toolset command language is described in the *ST20 Embedded Toolset Reference Manual*'.

2.1.1 How it works

To initialize 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 run-time files. This replaces a number of the standard library files with OS20 specific versions.
- Some modules within the 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 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 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 run-time system and pick and choose which libraries to replace from the C or OS20 run-time libraries. *Chapter 15: Advanced configuration on page 101* provides details of how the 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: An 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 13: ST20-C1 specific features on page 87 for further details.

2.1.2 Initializing partitions

The two partitions used internally by OS20, the system and internal partitions, are set up automatically when the **st20cc** linker command line option **-runtime os20** 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:

```
malloc(size);
```

and

```
memory_allocate(system_partition, size);
```

are now equivalent.

Similarly **calloc** is equivalent to **memory_allocate_clear**, **free** is equivalent to **memory_deallocate** and **realloc** is equivalent to **memory_reallocate**.

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 OS20 configuration files defining a number of global variables which are read by OS20 at start up. These are defined using the **addressof**, **sizeof** and **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 OS20 program, in this case a simple terminal emulator; see *Figure 1*. The code is written to run on an STi5500 evaluation board, but can be easily ported to another target. For device-specific details, refer to the device datasheet.

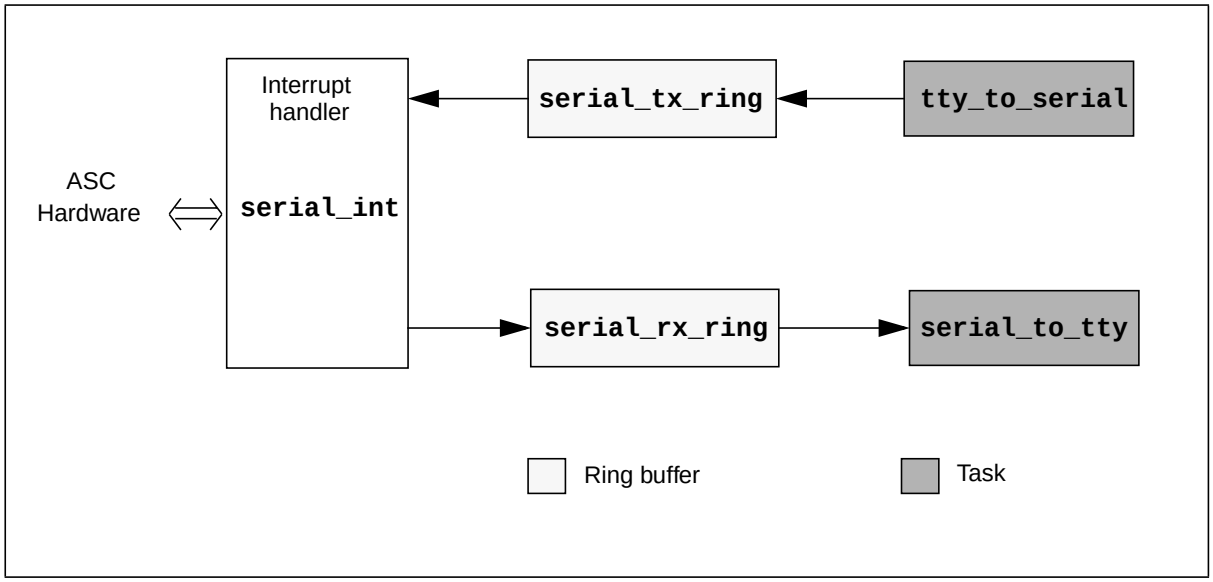

Figure 1: Example program schematic

To keep the example concise, some code which does not demonstrate the use of OS20 is omitted here. The full source code is provided with the 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:

```
#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.

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:

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

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

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

  /* Initialize 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);

  /* Initialize 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 is 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: The interrupt level is not configured here.

It is good practice to initialize all used interrupt levels in one central location rather than individual modules, as each one may be shared by several interrupt numbers (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 manage the serial communication are created. This allocates the tasks' stacks from the system partition, and immediately starts them running.

The next part of the software is the interrupt handler:

```
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 pending[1]. 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 (that is, containing a character), and the transmit buffer being empty.

Note: The STATUS *register is masked by the variable* **serial_mask***. This ensures that the code does not check for the transmit buffer being empty when there are no characters to transmit.*

1. The possibility of error interrupts is ignored in this simple example.

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

```
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:

```
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 writing data. 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:

```
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\n", devid.id, device_name(devid));

  /* Initialize 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 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 STi5500MB159 example.c -o example.tco -g -c
st20cc -p STi5500MB159 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 OS20 run-time library 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 OS20 manually

If the **-runtime** option to **st20cc** cannot be used, then it is still possible to use OS20.

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

```
st20cc -p STi5516MB382 -T myfile.cfg app.tco -o system.lku
```

where **myfile.cfg** includes the following commands:

```
file os20.lib
file os20intc1.lib
file os20ilc1.lib
```

Note: The two libraries **os20intc1.lib** *and* **os20ilc1.lib** *may need to be replaced by alternative libraries for some devices; see Section 10.2: Selecting the correct interrupt handling system on page 64 for further details.*

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 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. OS20 is now fully up and running.
5. Tasks can now be created by calling **task_create** or **task_init**, together with any other 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 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 13: ST20-C1 specific features on page 87*.

When OS20 is used, the heap functions (**malloc**, **calloc**, **free** and **realloc**), debug functions, device-independent I/O functions and stdio functions are thread-safe; see the *ST20 Embedded Toolset Reference Manual*, chapter *Libraries introduction*.

Note: Although thread-safe versions of the heap functions are used they are not mapped to the OS20 memory management functions as they are when the **-runtime** *option is used; see Section 2.1.2: Initializing partitions.*

3 Kernel

To implement multi-priority scheduling, 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.

The toolset is supplied with two prebuilt OS20 kernel libraries: the deployment kernel and debug kernel. The debug kernel is provided to support debugging. Currently the only difference between the two kernels is that the debug kernel has an additional time logging facility. Apart from specific references to the "debug kernel", the term "kernel" when used in this chapter applies to either kernel.

3.1 Implementation

The kernel maintains two vitally important pieces of information:

- the currently executing task, and thus what priority is currently being executed,
- a list of all the tasks which are currently ready to run.

  This is 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 happens on three possible occasions.

- When a task is about to be scheduled, the scheduler is called to determine if the new task is of higher priority than the currently executing task. If it is, 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.
- When a task deschedules, for example it is waiting on a message queue which does not have any messages available, then the scheduler is invoked to decide which task to run next. The kernel examines the list of processes which are ready to run, and picks the one with the highest priority.

- 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 is 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 of the same priority get a chance to run.

In this way the kernel ensures that it is always the highest priority task(s) which run.

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 Optional debug features

The OS20 kernel is supplied in two forms, the deployment kernel and the debug kernel. The debug kernel contains extra features designed to find bugs and assist with performance tuning. *Chapter 16: Alphabetical list of functions on page 111* contains function descriptions.

Note: All features in the debug kernel are slightly intrusive. This may subtly alter the real-time performance of the system being debugged, however, in most cases the difference in performance should be negligible.

3.2.1 Assertion checking

The debug kernel performs assertion checks on entry to every OS20 library call. These assertions catch a number of illegal calling conditions that can lead to a variety of difficult to debug scheduling failures. Specifically the assertions are triggered if the application illegally:

- passes a NULL pointer argument,
- calls a function from an interrupt handler,
- calls a function from a high priority process (ST20-C2 only),
- calls a function while interrupts are locked.

An assertion is also triggered if any of the following functions are called on a non-timeout object with a timeout that is not **`TIMEOUT_INFINITY`**:

- **`semaphore_wait_timeout`**,
- **`message_claim_timeout`**,
- **`message_receive_timeout`**.

When an assertion is triggered a message is displayed identifying the failure and the machine will stop in the debugger at the point of failure. If a debugger is not connected then the machine will enter a busy loop until a debugger is connected at which point it will display the message and stop the machine.

3.2.2 Time logging

For targets with an ST20-C2 core the debug kernel is also configured to maintain a record of the amount of time each task and interrupt spends executing. This facility is not available on ST20-C1 cores.

The following information is available in the debug kernel.

- **Task time logging** maintains a record of the amount of time each task spends running on the processor. The data collected can be accessed using the **`task_status`** function.
- **Interrupt time logging** maintains a record of the amount of time spent servicing each interrupt and the number of times the interrupt has been called. The functions

interrupt_status and **interrupt_status_number** are used to return this information.

- The functions **kernel_idle** and **kernel_time** return the time spent idle and the total up-time respectively. The total up-time being the time elapsed since the kernel started.

3.2.3 Using the debug kernel

If OS20 is started automatically using **-runtime os20** then the **st20cc** option **-debug-runtime** causes the application to link with the debug kernel. This is the recommended way to access the debug kernel.

If OS20 is started manually (see *Section 2.2: Starting OS20 manually*) then the three library files used must be prefixed with '**debug/**'.

For example:

```
file debug/os20.lib
file debug/os20intc1.lib
file debug/os20ilc1.lib
```

Note: No assertion checking is performed until kernel_start() has been called, for this reason assertion checking cannot be used to find problems during manual initialization.

3.3 OS20 kernel

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

```
if (kernel_initialize() != 0) {
   printf ("Error : initialize. 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);
}
```

3.4 Kernel header file: kernel.h

All the definitions related to the kernel are in the single header file, **kernel.h**; see *Table 2*:

Function	Description
kernel_idle	Return the kernel idle time
kernel_initialize	Initialize for preemptive scheduling
kernel_start	Starts preemptive scheduling regime
kernel_time	Return the kernel up-time
kernel_version	Return the OS20 version number

Table 2: Functions defined in kernel.h

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 OS20; see *Section 4.2: Allocation strategies*. These give the user flexibility in controlling how memory is allocated, allowing a space/speed trade-off.

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 OS20 these concepts have been combined into a partition, which has three properties:

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

The method of allocating/deallocating memory is the same whatever style of partition is used, only the algorithm used (and 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 *Chapter 16: Alphabetical list of functions*, function *partition_delete on page 222*). This arrangement can be useful to avoid having to explicitly free all the blocks allocated:

1. Allocate a block from a partition, and create a second partition to manage it.
2. Allocate memory from the partition as normal.
3. 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 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 affect the whole application.

4.2 Allocation strategies

OS20 supports three types of partition:

- **Heap**

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

  Although the heap style of allocator is very versatile, it does have some disadvantages. It is not deterministic, the time taken to allocate and free memory is variable because it depends 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.

- **Fixed**

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

- **Simple**

  The simple partition is a trivial allocator, which just increments a pointer to the next available block of memory. This means that it is impossible to free any memory back to the partition, but there is no wasted memory when performing memory allocations. 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 3*.

Properties	Heap	Fixed	Simple
Allocation method	As requested by **`memory_allocate`** or **`memory_reallocate`**	Fixed at creation by **`partition_create_fixed`** or **`partition_init_fixed`**.	As requested by **`memory_allocate`**
Deallocation possible	Yes	Yes	No
Overhead size (bytes)	8	0	0
Deterministic	No	Yes	Yes

Table 3: Partition properties

4.3 Predefined partitions

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 (for example, **task_init**, **semaphore_init_fifo**) are also available with a creation style of interface (for example, **task_create**, **semaphore_create_fifo**), where OS20 performs the memory allocation for the object. In these cases OS20 uses two predefined 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: OS20 tasks overview on page 27* 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** option **-runtime os20** is specified when linking, this initialization is performed automatically; see *Chapter 2: Getting started on page 9*.

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

```
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;
}
```

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

```
place internal_partition INTERNAL
```

4.3.1 Calculating partition sizes

In order to calculate the size of system and internal partitions, the memory allocation function requires several pieces of information for each object created using the **_create** functions (for example, **task_create**, **message_create_queue**, **semaphore_create_fifo**). The information consists of memory requirements for:

- the object (see *Object memory allocation* below),
- the object tasks' stacks (created using **task_create**),
- object overhead (see *Object overhead memory allocation* below),
- any additional allocations performed by the user's application.

Object memory allocation

Each object is defined by a data structure or type (refer to individual chapters). For example, **task_t** (refer to *Chapter 5: Tasks on page 27*). *Table 4* lists the different types of object structure that may be created and their memory requirement.

Object structure	Size (words)	Size (bytes)	Notes
chan_t	4	16	ST20-C2 specific
semaphore_t	6	24	
message_queue_t	19	76	
partition_t	15	60	
task_t	9	36	
tdesc_t	6 9	24 36	ST20-C2 specific ST20-C1 specific

Table 4: Object size memory requirement

Object overhead memory allocation

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 3* shows the memory overhead associated for each partition type.

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

For fixed partitions, the largest free block of memory is always 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 5*.

Function	Description
`memory_allocate`	Allocate a block of memory from a partition
`memory_allocate_clear`	Allocate a block of memory from a partition and clear to zero
`memory_deallocate`	Free a block of memory back to a partition
`memory_reallocate`	Reallocate a block of memory from a partition
`partition_create_simple`	Create a simple partition
`partition_create_heap`	Create a heap partition
`partition_create_fixed`	Create a fixed partition
`partition_delete`	Delete a partition
`partition_init_simple`	Initialize a simple partition
`partition_init_heap`	Initialize a heap partition
`partition_init_fixed`	Initialize a fixed partition
`partition_status`	Get the status of a partition

Table 5: Functions defined in partitio.h

Table 6 lists the types defined by **`partitio.h`**.

Types	Description
`partition_t`	A memory partition
`partition_stuatus_flags_t`	Additional flags for **`partition_status`**

Table 6: Types defined by partitio.h

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 OS20 tasks overview

A task consists of a data structure, stack and a section of code. A task's data structure is known as its state; its exact content and structure are processor-dependent. In OS20 it is divided into two parts, and includes the following elements:

- **Dynamic state**

  This is 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**

  This is defined in the data structure **`task_t`**, which is used by OS20 to describe the task, and which does not usually change while the task is running. It includes the task's state (that is; being created, executing, terminated) and the stack range (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: Getting the current task's ID on page 36*.

The task's data structure may either be allocated by 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: Creating and running a task*.

5.2 Implementation of priority and timeslicing

Readers familiar with the ST20 micro-core and OS20 priority handling may wish to skip to *Section 5.3: OS20 priorities* which introduces the facilities provided by OS20 for influencing priority.

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

The ST20-C1 has no hardware support for multiple priorities.

The ST20-C2 supports two priorities of processes, high and low; see *Figure 2*.

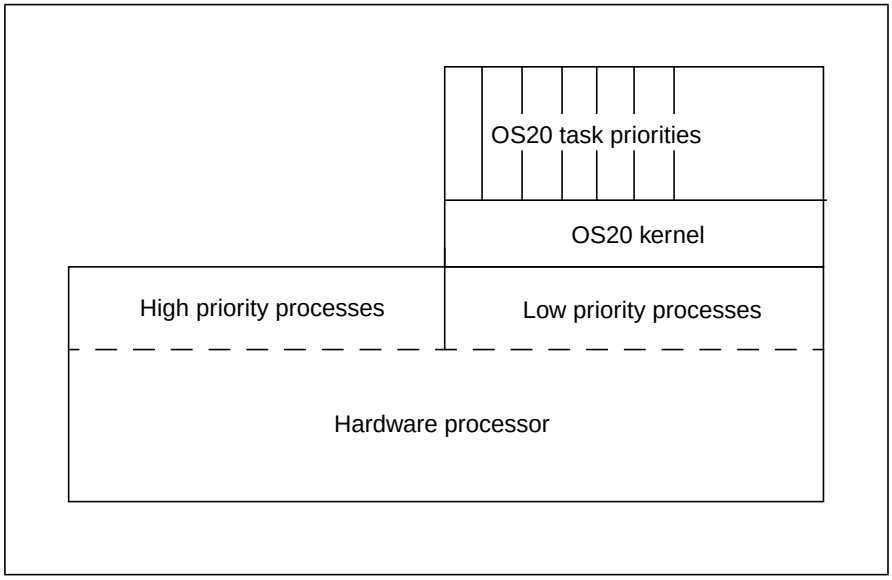

Figure 2: ST20-C2 priorities

High priority processes take precedence over low priority processes, for example, 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 OS20 scheduler, and so some restrictions have to be placed on them.

- They cannot use priority-based and timeout semaphores.
- They cannot use timeout message queues.

In addition, they inherit two features of the hardware scheduler.

- Tasks are not timesliced; they execute until they voluntarily deschedule.
- Units of time are different, with high priority processes running considerably faster than low priority processes. Clock times are device-dependent, so check datasheets for actual timings.

5.2.1 Timeslicing on the ST20-C1

On the ST20-C1 microprocessor, timeslicing is supported by the **`timeslice`** instruction. By default, the compiler disables timeslicing. However, if the application is compiled with the **st20cc** option **`-finl-timeslice`** then the compiler inserts **`timeslice`** instructions (except in hand-crafted assembler).

Note: The run time 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 is timesliced, that is, 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.

On some ST20-C1 devices, the timeslice clock is provided by peripheral modules and this timeslice clock must be enabled for timeslicing to work. See the device datasheet for details.

Note: Timeslicing is implemented independently of the clocking peripheral discussed in Chapter 13: ST20-C1 specific features on page 87.

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 register; see *Chapter 9: Real-time clocks on page 57*.

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 (low priority hardware process) has been continuously executing, the processor attempts to deschedule the task. This occurs after the next **j** or **lend** instruction is executed. When this happens the task is descheduled and the next waiting task is scheduled; see *Figure 3*.

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

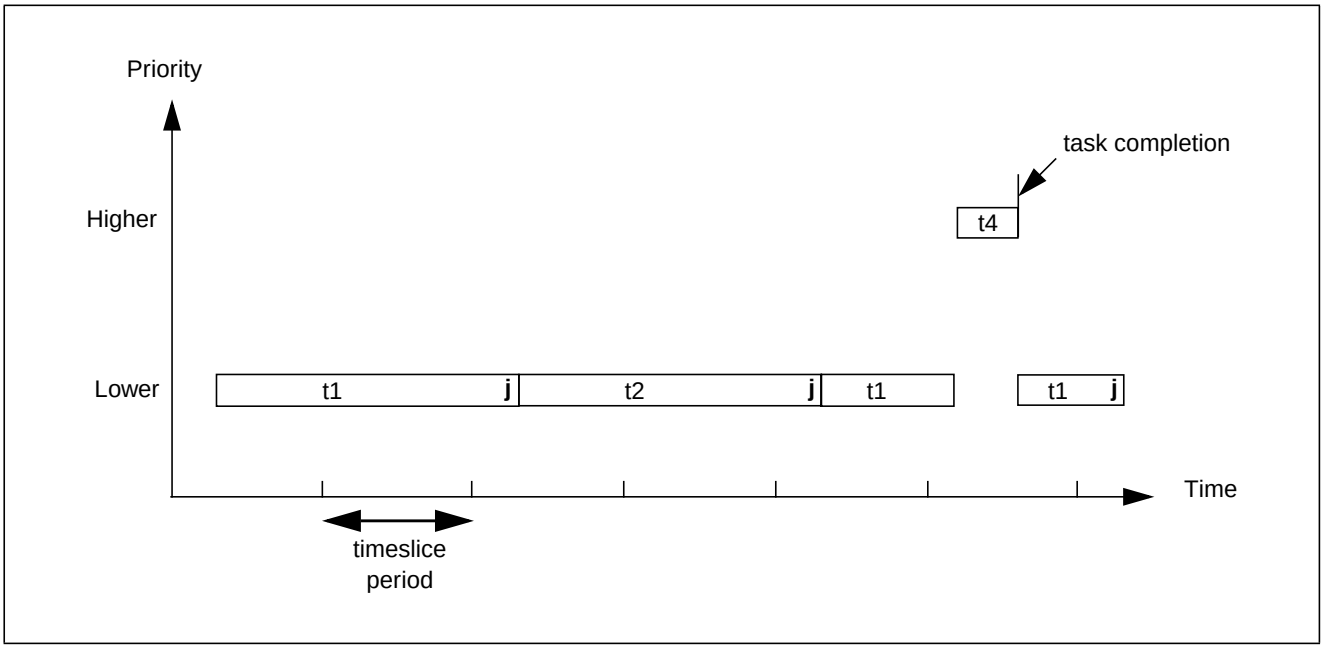

Figure 3: Timeslicing on the ST20-C2

A task nominally runs 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 OS20 task is preempted by a higher priority OS20 task then when the lower priority tasks resumes, it starts its timeslice period from the beginning of the timeslice period. However, if an OS20 task is interrupted by an interrupt or preempted by a high priority process then it resumes the timeslice period from the point where the interrupt or high priority process released the period. Therefore the OS20 task loses some of its timeslice.

Further details are given in the *ST20-C2 Core Instruction Set Reference Manual 72-TRN-273*.

5.3 OS20 priorities

The number of 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: Task header file: task.h on page 41*. Numerically higher priorities preempt lower priorities, for example, 3 is a higher priority than 2.

A task's initial priority is defined when it is created; see *Section 5.5: Creating and running a task*. The only task which does not have its priority defined in this way is the root task, that is, the task which starts 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:

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

`task_priority()` retrieves the 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:

```
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 OS20 tasks, not to high priority hardware processes.

5.4 Scheduling

An active task may either be running or waiting to run. OS20 ensures the following conditions are met.

- 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 OS20 scheduler saves the current task's state and makes the higher priority task the current task. The current task runs to completion unless it is preempted by a higher priority task, and so on. Once a task has completed, the next highest priority task starts executing.

- 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 executes 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:

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

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

If a task voluntarily deschedules, for example, by calling **semaphore_wait**, then the critical region is 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, for example, the semaphore count was already greater than zero.

Note: 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 10: Interrupts on page 61.*

5.5 Creating and running a task

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

```
#include <task.h>
task_t* task_create( 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 (for example an **`int`**) otherwise a data structure should be set up.

The functions differ in how the task's data structure is allocated. **`task_create`** allocates memory for the task's stack, control block **`task_t`** 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, functions use the following amounts of space.

- 4 words are used for the task to remove itself if it returns.
- 4 extra words are used for the initial user stack.
- On the ST20-C2, 6 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).
- Additional space is then used recursively:
  - ➢ for local variables declared in the function (add up the number of words),
  - ➢ for calls to extra functions (for a library function, allow a worst case of 150 words).

For details of data representation; see the *ST20 Embedded Toolset Reference Manual* chapter *Implementation Details*.

Both functions require an 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: Task header file: task.h on page 41*.

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 OS20 task priority should not be used. High priority processes have restrictions associated with them as described in *Section 5.2: Implementation of priority and timeslicing on page 28*.

5.5.1 Creating a task for an RCU

Two functions are provided for creating a task in a relocatable code unit: **task_create_sl** and **task_init_sl**. For details of using OS20 with relocatable code units, see the *ST20 Embedded Toolset Reference Manual*, chapter *Building and running relocatable code*.

5.6 Synchronizing tasks

Tasks synchronize their actions with each other using semaphores, as described in *Chapter 6: Semaphores on page 43*.

5.7 Communicating between tasks

Tasks communicate with each other by using message queues, as described in *Chapter 8: Message handling on page 51*.

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.

```
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, that is, it delays execution for the specified number of timer ticks. If the time given is negative, no delay takes place.

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

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

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

```
#include <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, that is, 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: OS20 priorities on page 30*.

5.10 Suspending tasks

Normally a task only deschedules when it is waiting for an event such as a semaphore signal. 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 stops executing immediately. When the task starts executing again, another task must resume it. When it is resumed, the task will be unaware that it was 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 has been suspended will not start executing again until both the task is resumed and the semaphore is signalled, although these can occur in either order.

Caution: Task suspension can easily cause deadlock; see *task_suspend* in *Chapter 16: Alphabetical list of functions on page 111*.

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:

```
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 starts 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:

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

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

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

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

and once it is willing to be killed again calling:

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

While the task is immortal, it cannot be killed. However, if an attempt was made to kill the task 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:

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

While **task_id** is very efficient when called from a task, it takes 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:

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

This returns 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 assigns 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 assigns 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: Waiting for termination*.

The function:

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

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

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. 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 *ST20 Embedded Toolset User Manual*, chapter *st20cc compile/link tool*. 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 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 stack overflows are reported immediately, it has a number of problems.

- There is a run-time cost incurred for every function call to perform the check.
- It can only report on functions which are 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 allow this, OS20 normally fills a task's stack with a known value. As the task runs, it writes its own data into the stack, altering this value, and later the stack can be inspected to determine the highest address which has not been altered.

To support this, OS20 provides the function:

```
int task_status( task_t* Task,
                 task_status_t *Status,
                 task_status_flags_t Flags );
```

This function can be used to determine information about the task's 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:

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

returns details about the current stack fill settings, and:

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

allows them to be altered. Stack filling can be enabled or disabled, or the fill value 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 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

5.14.1 Application data

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:

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

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

```
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:

```
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);
}
```

5.14.2 Library data

OS20 also provides a facility to manage multiple instances of task private data. This is to enable libraries to store their own per task private data. Two function calls provide access to this facility:

```
void* task_private_data(task_t* Task, void* Cookie);
int task_private_data_set(task_t* Task, void* Data, void* Cookie,
  void (*Destructor)( void* Data ));
```

This API allows a client to allocate and associate a block of data with a given **Task**, under a unique **Cookie** identifier. The **Cookie** is typically the address of some object within the client library, in order to guarantee uniqueness.

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

task_private_data_set() is used to request that a block of **Data** be associated with the given **Task** under the given **Cookie**. Only one data block can be registered under a given cookie for a given task. The **Destructor** parameter is the address of a routine which OS20 calls when the task is deleted. The **Destructor** is called with the address of the task private data allocated by the library, and it has the responsibility to deallocate this data.

If the task parameter is **NULL** the current task is used for the operation.

If **task_private_data()** or **task_private_data_set()** are called prior to kernel initialization, then the operations are performed on the root task.

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 calls 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: Task data*), 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:

```
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: 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 an absolute time 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:

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

This removes the task from the list of known tasks and allows 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 tasks are in the single header file, **task.h**; see *Table 7*, *Table 8* and *Table 9*.

Function	Description
task_context	Return the current execution context
task_create	Create an OS20 task
task_create_sl	Create an OS20 task specifying a static link
task_data	Retrieve a task's data pointer
task_data_set	Set a task's data pointer
task_delay	Delay the calling task for a period of time
task_delay_until	Delay the calling task until a specified time
task_delete	Delete a task
task_exit	Exits the current task
task_id	Find current task's ID
task_immortal	Make the current task immortal
task_init	Initialize an OS20 task
task_init_sl	Initialize an OS20 task specifying a static link
task_kill	Kill a task
task_lock	Prevent task rescheduling
task_mortal	Make the current task mortal
task_name	Return the task's name
task_onexit_set	Setup a function to be called when a task exits
task_onexit_set_sl	Setup a function to be called when a task exits and specify a static link
task_priority	Retrieve a task's priority

Table 7: Functions defined in task.h

Function	Description
task_priority_set	Set a task's priority
task_private_data	Retrieves some task private data
task_private_data_set	Registers some task private data
task_reschedule	Reschedule the current task
task_resume	Resume a suspended task
task_suspend	Suspend a task
task_stack_fill	Return the task fill configuration
task_stack_fill_set	Set the task stack fill configuration
task_status	Return status information about the task
task_unlock	Allow task rescheduling
task_wait	Wait until one of a list of tasks completes

Table 7: Functions defined in task.h

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

Table 8: Types defined in task.h

Macro	Description
OS20_PRIORITY_LEVELS	Number of OS20 task priorities. Default is 16.
MAX_USER_PRIORITY	Highest user task priority. Default is 15.
MIN_USER_PRIORITY	Lowest user task priority. Default is 0.

Table 9: Macros defined in task.h

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 Semaphores overview

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

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

Semaphores are created using one of the following functions:

```
semaphore_t* semaphore_create_fifo (int value);
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:

```
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 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 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 are blocked for a minimum amount of time. In this case a second type of semaphore can be used, a priority based semaphore. For this type of semaphore, tasks are queued based on their priority first, and the order which they call **semaphore_wait** second. Semaphores of this type are created using **semaphore_create_priority** or **semaphore_init_priority** or one of the timeout versions of these functions.

Semaphores may be acquired by the function:

```
void semaphore_wait(semaphore_t* Sem);
```

For semaphores created via one of the timeout functions, the following function may also be used:

```
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 zero, then the count is decremented, and the task continues. If however, the count is already zero, then the task adds 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 is 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: 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 returns and the task continues without acquiring the semaphore. Two special values may be specified for the timeout period.

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

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

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

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

If a semaphore is deleted using **semaphore_delete** then how the memory is released depends on whether the semaphore was created by the **create** or **init** version of the function. See the functional description in *Chapter 16: Alphabetical list of functions*, function *semaphore_delete on page 233*.

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 Using semaphores

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

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

Typically, semaphores are set up to allow at most one task access to a 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 successfully takes the semaphore, and any others are forced to wait. When the task currently in the critical region leaves, it releases the semaphore, and allows the first of the waiting tasks into the critical region.

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

All the 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 returns immediately, having decremented the counter. If no devices are available then the task is added to the queue. When a task has finished using a device it calls **`semaphore_signal`** to release it.

6.3 Semaphore header file: semaphor.h

All the definitions related to semaphores are in the single header file, **semaphor.h**; see *Table 10* and *Table 11*.

Function	Description
semaphore_create_fifo	Create a FIFO queued semaphore
semaphore_create_fifo_timeout	Create a FIFO queued semaphore with timeout
semaphore_create_priority	Create a priority queued semaphore
semaphore_create_priority_timeout	Create a priority queued semaphore with timeout
semaphore_delete	Delete a semaphore
semaphore_init_fifo	Initialize a FIFO queued semaphore
semaphore_init_fifo_timeout	Initialize a FIFO queued semaphore with timeout
semaphore_init_priority	Initialize a priority queued semaphore
semaphore_init_priority_timeout	Initialize a priority queued semaphore with timeout
semaphore_signal	Signal a signal
semaphore_wait	Wait for a signal
semaphore_wait_timeout	Wait for a semaphore or a timeout

Table 10: Functions defined in semaphor.h

Types	Description
semaphore_t	A semaphore

Table 11: Types defined in semaphor.h

7 Mutexes

Mutexes provide a simple and efficient way to ensure mutual exclusion and control access to a shared resource.

7.1 Mutexes overview

A mutex structure **`mutex_t`** contains several pieces of data including:

- the current owning task,
- a queue of tasks waiting to take the mutex.

A mutex can be owned by only one task at time. In this sense they are like OS20 semaphores initialized with a count of 1 (also known as **binary semaphores**). Unlike semaphores, once a task owns a mutex, it can re-take it as many times as necessary, provided that it also releases it an equal number of times. In this situation binary semaphores would deadlock.

Mutexes are created using one of the following functions:

```
mutex_t* mutex_create_fifo(void);
void mutex_init_fifo(mutex_t *mutex);
mutex_t* mutex_create_priority(void);
void mutex_init_priority(mutex_t *mutex)
```

The **`create_`** version of these functions allocates memory for the mutex automatically, while the **`init_`** versions enable the user to specify a pointer to the mutex, using the data structure **`mutex_t`**.

The mutexes which OS20 provide differ in the way in which tasks are queued when waiting for it. For FIFO mutexes tasks are queued in the order in which they call **`mutex_lock()`**. Mutexes of this type are created using **`mutex_create_fifo()`** or **`mutex_init_fifo()`**.

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

Mutex may be acquired by the functions:

```
void mutex_lock(mutex_t* mutex);
```

and

```
int mutex_trylock(mutex_t* mutex);
```

When a task wants to acquire a mutex, it calls **`mutex_lock()`**. If the mutex is currently unowned, or already owned by the same task, then the task gets the mutex and continues. If however, the mutex is owned by another task, then the task adds itself to the queue of tasks waiting for the mutex and deschedules itself. Eventually another task should release the mutex, and the first waiting task gets the mutex and is able to continue. In this way, when the task returns from the function it has acquired the mutex.

Note: The same task can acquire a mutex any number of times without deadlock, but it must release it an equal number of times.

If you want to make certain that the task does not wait indefinitely for a mutex then use **`mutex_trylock()`**, which attempts to gain ownership of the mutex, but fails immediately if it is not available.

A task is automatically made immortal while it has ownership of a mutex.

When a task wants to release the mutex, it calls **`mutex_release()`**:

```
int mutex_release(mutex_t* mutex);
```

This looks at the queue of waiting tasks, and if the queue is not empty, removes the first task from the queue and, if it is not of a lower priority, assigns ownership of the mutex to that task, and makes it executable. If there are no tasks waiting, then the mutex becomes free.

Note: If a task exits whilst holding a mutex, the mutex remains locked, and a deadlock is inevitable.

7.1.1 Priority inversion

Priority mutexes also provide protection against priority inversion. This can occur when a low priority task acquires a mutex, and then a high priority task tries to claim it. The high priority task is then forced to wait for the low priority task to release the mutex before it can proceed. If an intermediate priority task now becomes ready to run, it preempts the low priority task. A lower priority task (that is not holding the mutex in question) is therefore blocking the execution of a higher priority task, this is termed priority inversion. Priority mutexes are able to detect when this occurs, and correct the situation. This is done by temporarily boosting the low priority task's priority to be the same as the priority of the highest priority waiting task, all the while the low priority task owns the mutex.

Priority inversion detection occurs every time a task has to queue to get a priority mutex, every time a task releases a priority mutex, and every time a task changes priority.

7.2 Using mutexes

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

Mutexes allow at most one task access to the resource at any given time. Areas of code protected using mutexes are sometimes called **critical regions**.

The critical region is surrounded with calls to **`mutex_lock()`** at the start and **`mutex_release()`** at the end. Thus the first task which tries to enter the critical region successfully takes the mutex, and any others are forced to wait. When the task currently in the critical region leaves, it releases the mutex, and allows the first of the waiting tasks into the critical region.

7.3 Mutex header file: mutex.h

All the definitions related to mutexes are in the single header file, **`mutex.h`**, see *Table 12* and *Table 13*.

Function	Description
`mutex_create_fifo`	Create a FIFO queued mutex
`mutex_create_priority`	Create a priority queued mutex
`mutex_delete`	Delete a mutex
`mutex_init_fifo`	Initialize a FIFO queued mutex
`mutex_init_priority`	Initialize a priority queued mutex
`mutex_lock`	Acquire a mutex, block if not available
`mutex_release`	Release a mutex
`mutex_trylock`	Try to get a mutex, fail if not available

Table 12: Functions defined in mutex.h

Type	Description
`mutex_t`	A mutex

Table 13: Types define in mutex.h

8 Message handling

A message queue provides a buffered communication method for tasks. Message queues also provide a way to communicate without copying 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 8.3: Using message queues on page 54*. This prevents the interrupt handler from blocking on a message claim.

8.1 Message queues overview

An OS20 message queue implements two queues of messages: one for message buffers which are currently not being used (known as the “free” queue); 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 4*.

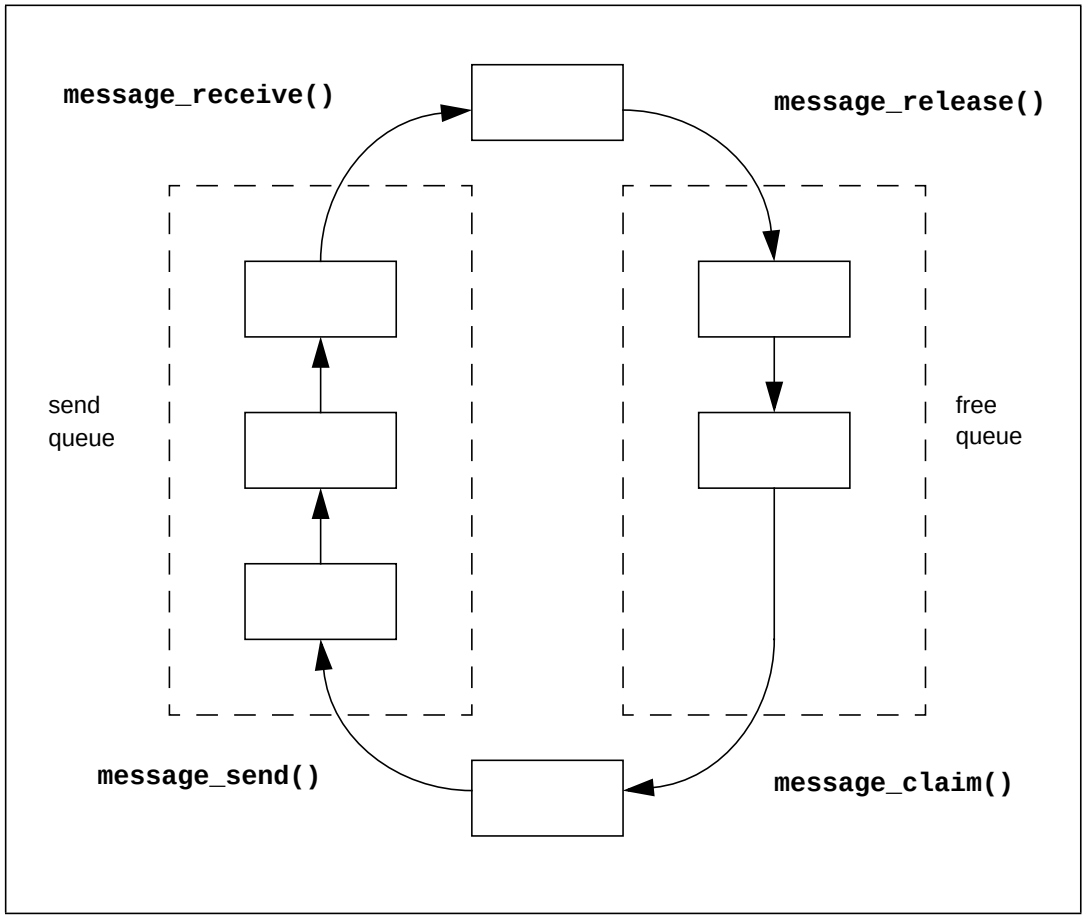

Figure 4: Message queues

8.2 Creating message queues

Message queues are created using one of the following functions:

```
#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:

```
#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 5*. The user must specify the maximum size for a message element and the total number of elements required.

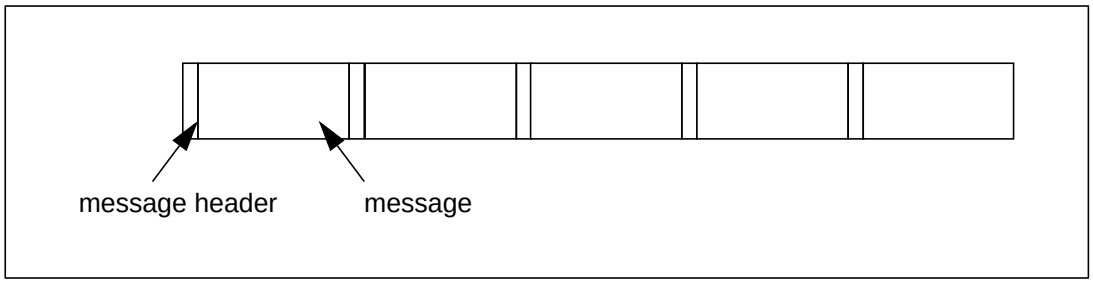

Figure 5: OS20 message elements

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

message_init_queue and **message_init_queue_timeout** require 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:

```
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 returns a pointer to the allocated memory, which should be passed to **message_init_queue** or **message_init_queue_timeout** as the parameter **MessageQueue**.

Note: These functions cannot use the hardware support for semaphores, and so are larger and slower than the nontimeout versions.

Example

```
#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);
}
```

8.3 Using message queues

Initially all the messages are on the free queue. The user allocates free message buffers by calling either of the following functions, which can then be filled in with the required data:

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

void* message_claim_timeout( message_queue_t* queue
                             const 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 returns **NULL**. Two special values may be specified for the timeout period.

- **TIMEOUT_IMMEDIATE** causes the message queue to be polled and the function to return immediately. A message buffer may or may not be acquired and the task continues.
- **TIMEOUT_INFINITY** causes the function to behave the same as **message_claim**, that is, the task waits indefinitely for a message buffer to become available.

When the message is ready it is sent by calling **message_send()**, at which point it is added to the send queue.

Messages are removed from the send queue by a task calling either of the functions:

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

void* message_receive_timeout( message_queue_t* queue
                               const 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 in place of ***time***, then the task continues whether or not a message is received and if **TIMEOUT_INFINITY** is specified the function behaves as **message_receive** and waits indefinitely.

Finally when the receiving task has finished with the message buffer, it should free it by calling **message_release()**, which adds it to the free queue, where it is again available for allocation.

If the size of the message is variable, the user should specify that the message is **sizeof(void*)**, and then use pointers to the messages as the arguments to the message functions. The user is then responsible for allocating and freeing the real messages using whatever techniques are appropriate.

Message queues may be deleted by calling **message_delete_queue()**. If the message queue was created using **message_create_queue** or **message_create_queue_timeout** then this also frees 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.

8.4 Message header file: message.h

All the definitions related to messages are in the single header file, **message.h**; see *Table 14* and *Table 15*.

Function	Description
message_claim	Claim a message buffer
message_claim_timeout	Claim a message buffer with timeout
message_create_queue	Create a fixed size message queue
message_create_queue_timeout	Create a fixed size message queue with timeout
message_delete_queue	Delete a message queue
message_init_queue	Initialize a fixed size message queue
message_init_queue_timeout	Initialize a fixed size message queue with timeout
message_receive	Receive the next available message from a queue
message_receive_timeout	Receive the next available message from a queue or timeout
message_release	Release a message buffer
message_send	Send a message to a queue

Table 14: Functions defined in message.h

Types	Description
message_hdr_t	A message buffer header
message_queue_t	A message queue

Table 15: Types defined in message.h

9 Real-time clocks

Time is very important for real-time systems. 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. OS20 provides functions to:

- add and subtract quantities of time,
- determine if one time is after another,
- return the current time.

9.1 ST20-C1 clock peripheral

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

OS20 for ST20-C1 contains a table of function pointers that it calls to handle time-related operations. By default, the function pointers are connected to fatal error handlers. Therefore, before using any time related API calls, these function pointers must be initialized by calling **`timer_initialize()`** or **`timer_init_pwm()`**. For more information, see *Chapter 13: ST20-C1 specific features on page 87*.

Note: The OS20 kernel and interrupt controller must be initialized before the clock peripheral is initialized.

9.2 The ST20 timers on the ST20-C2

The 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 0xFFFFFFFF, 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.

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

Parameter	Low priority	High priority
Interval between ticks	64 µs	1 µs
Ticks per second	15625	1000000
Approximate full timer cycle	76.35 hours	1.193 hours

Table 16: Summary of clock intervals for parts operating at 40 MHz

9.3 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.

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

The time at which counting starts is no later than the call to **`kernel_start`**.

9.4 Determining the tick rate

The function **`time_ticks_per_sec()`** can be used to determine the current tick rate of the timer.

On ST20-C1, the tick rate is the same across all tasks and interrupts.

On ST20-C2, the tick rate returned is that of the current CPU priority, that is, the value returned will be 64 times larger if the CPU is running a high priority process or high priority interrupt.

`time_ticks_per_sec_set()` is used to replace the estimated tick rate determined by the operating system with an actual tick rate observed on real hardware.

The estimated tick rate for ST20-C1 is configured automatically if **timer_init_pwm()** is used, however if **timer_initialize()** is used then there is no estimated tick rate and the user must call **time_ticks_per_sec_set()** before **time_ticks_per_sec()** is used.

The estimated tick rate for ST20-C2 is simply the values shown in *Table 16: Summary of clock intervals for parts operating at 40 MHz*.

9.5 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.

```
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_minus 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_minus** then the result is the number of ticks between the two times. If the result is positive then the first time is after the second. If the result is negative then the first time is before the second.

time_after determines whether the first time is after the second time. 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 one if the first time is after the second, otherwise it returns zero.

Some of these concepts are shown in *Figure 6*.

Time arithmetic is modulo 2^{32}. In applications running for a long time, take care 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 produces a result which may be ambiguous. Very long intervals can be tracked by counting a number of cycles of the clock.

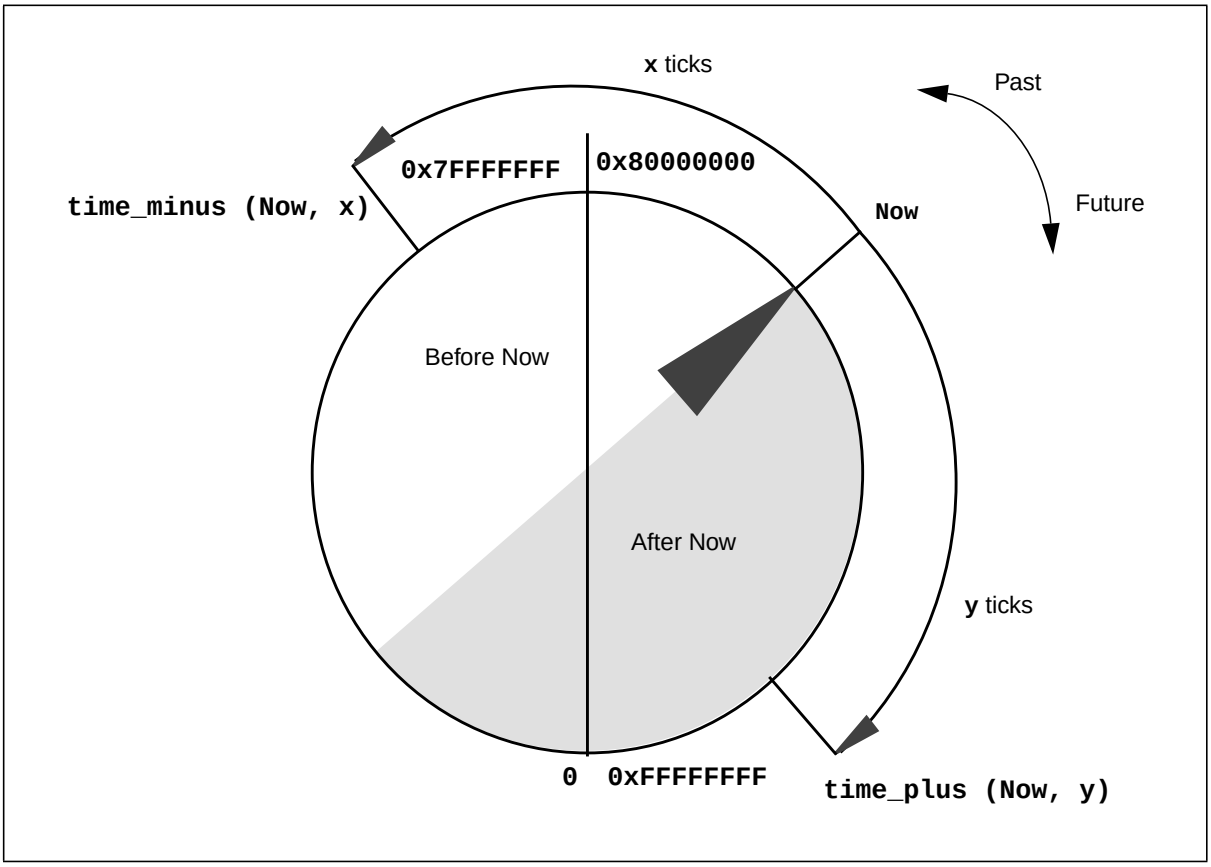

Figure 6: Time arithmetic

9.6 Time header file: ostime.h

All the definitions related to time are in the single header file, **`ostime.h`**; see *Table 17*.

Function	Description
`time_after`	Return whether one time is after another
`time_minus`	Subtract two clock values
`time_now`	Return the current time
`time_plus`	Add two clock values

Table 17: Functions defined in ostime.h

Table 18 lists the types defined by **`ostime.h`**.

Types	Description
`clock_t`	Number of processor clock ticks

Table 18: Types defined by ostime.h

10 Interrupts

Interrupts provide a way for external events to control the CPU. Normally, as soon as an interrupt is asserted, the CPU stops executing the current task, and starts executing the interrupt handler for that interrupt. In this way, the program 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 resumes execution of the interrupted task, which is unaware that it has been interrupted.

The interrupt handler which the CPU executes in response to the interrupt is called the first level interrupt handler. This piece of code is supplied as part of OS20, and simply sets up the environment so that a normal C function can be called. The OS20 API allows a different user function to be associated with each interrupt, and this is called when the interrupt occurs. Each interrupt also has a parameter associated with it, which is passed into the function when it is called. This could be used to allow the same code to be shared between different interrupt handlers.

10.1 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 OS20.

On some processors, interrupt sources are connected directly to the interrupt controller, similar to the example shown in *Figure 7*.

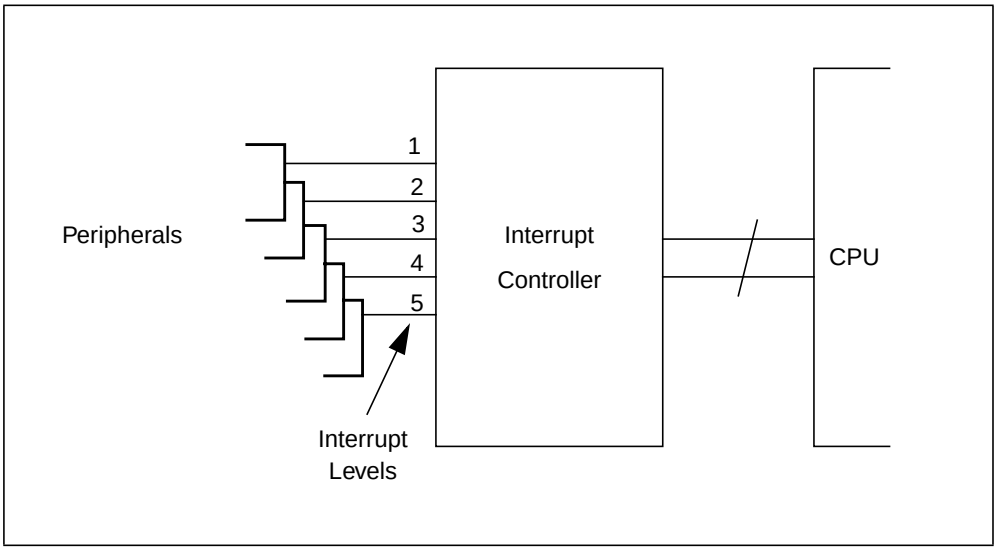

Figure 7: Peripherals directly attached to the interrupt controller - example

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 can interrupt an interrupt level 2, which can 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 8*. 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 maps interrupt numbers to interrupt levels. This mapping is programmable, allowing 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 between interrupt numbers and levels is that interrupt levels are prioritized (numerically higher interrupt levels preempt lower ones) however, interrupt numbers are not.

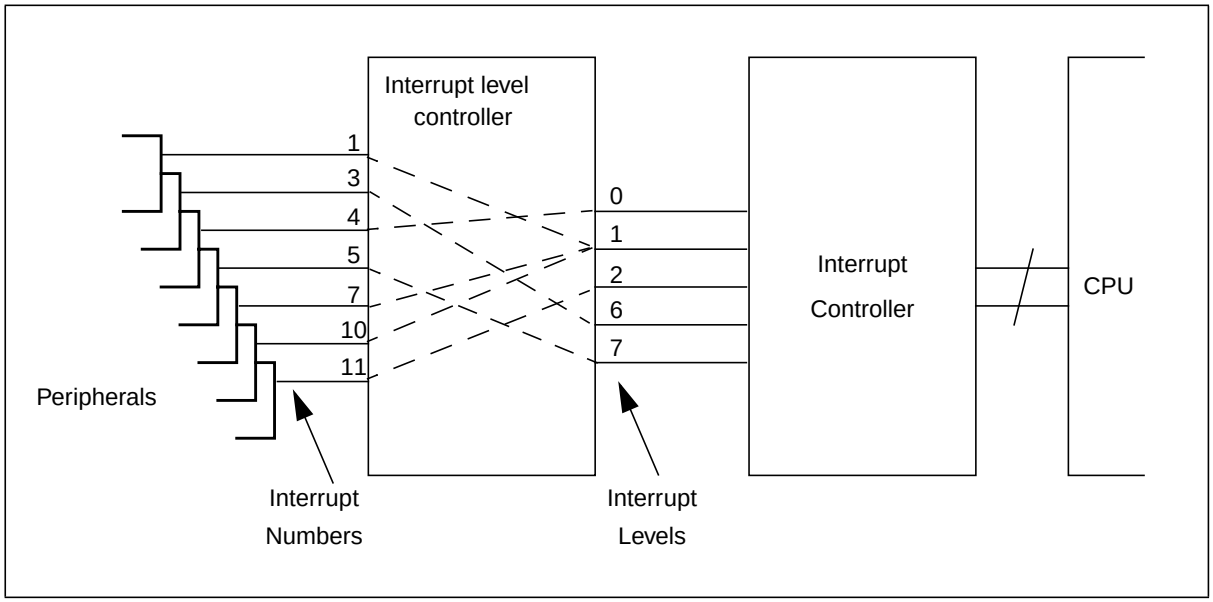

Figure 8: Peripherals mapped via an interrupt level controller - example

There are two types of **interrupt controller** for ST20 processors: IntC-1 and IntC-2. Both 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 19: Interrupt controller libraries*.

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

ILC-1 supports 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 supports 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 currently supports up to 128 interrupt numbers, each of which can have a programmable trigger mode and enable status.

OS20 functions provide support for all ST20 interrupt models.

10.2 Selecting the correct interrupt handling system

OS20 contains two libraries to support different interrupt controller combinations:

Library	Description
os20intc1.lib	IntC-1 (default)
os20intc2.lib	IntC-2

Table 19: Interrupt controller libraries

Additionally OS20 contains five libraries to support different interrupt level controller combinations:

Library	Description
os20ilcnone.lib	ILC-None
os20ilc1.lib	ILC-1 (default)
os20ilc2.lib	ILC-2
os20ilc2b.lib	ILC-2B
os20ilc3.lib	ILC-3

Table 20: Interrupt level controller libraries

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

Table 21 and *Table 22* cross-reference these libraries to specific ST chips.

Library	Description	Devices
os20intc1.lib	IntC-1 (default)	ST20GP6, ST20MC2, ST20TP3, ST20TP4, STi5100, STi5500, STi5505, STi5508, STi5510, STi5512, STi5514, STi5516, STi5517, STi5518, STi5519, STi5528, STi5580, STi5588, STi5589, STi5598, STV0396, STV3500, ST20-C1 simulator, ST20-C2 simulator.
os20intc2.lib	IntC-2	ST20DC1, ST20DC2, STi7710, STm5700, STV0684.

Table 21: Interrupt controller libraries

Library	Description	Devices
os20ilcnone.lib	ILC-None	ST20-C1 simulator, ST20-C2 simulator.
os20ilc1.lib	ILC-1 (default)	ST20DC1, ST20DC2, ST20GP6, ST20MC2, ST20TP3, ST20TP4, STi5500, STi5505, STi5508, STi5510, STi5512, STi5580.
os20ilc2.lib	ILC-2	STi5518, STi5519, STi5588, STi5589, STi5598.
os20ilc2b.lib	ILC-2B	STV0396
os20ilc3.lib	ILC-3	STi5100, STi5514, STi5516, STi5517, STi5528, STm5700, STi7710, STV0684, STV3500.

Table 22: Interrupt level controller libraries

All supported ST chips are listed in the *ST20 Embedded Toolset Reference Manual*, chapter *Alphabetical list of commands*, command *Chip*.

Note: All referenced technology and versions are listed in the ST20 Embedded Toolset Delivery Manual.

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: 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 is not 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.

10.2.1 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()`** or **`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.

If permanently migrating old code, it is advisable to change interrupt handling functions as shown in *Table 23*, which describes the migration path between ILCs.

ILC library	Legacy code	Recommended replacement
ILC-1	**interrupt_enable()** (INTERRUPT_GLOBAL_ENABLE)	**interrupt_enable_global()**[a]
	interrupt_disable() (INTERRUPT_GLOBAL_DISABLE)	**interrupt_disable_global()**[a]
	interrupt_pending_number	**interrupt_test_number()**[a]
ILC-2	All changes recommended for ILC-1 plus:	
	interrupt_enable()	**interrupt_enable_number()**[b] (May require multiple calls.)
	interrupt_disable()	**interrupt_disable_number()**[b] (May require multiple calls.)
ILC-3	All changes recommended for ILC-2 plus:	
	interrupt_pending_number()	**interrupt_test_number()**[b]

Table 23: Migration path for ILCs

a. This change is optional and makes code easier to port in the future.

b. This change is mandatory on the ILC-3 and the default ILC-2 library. See also the Note below.

Note: 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 OS20 configuration option is provided to support this when the ***st20cc*** *option* **-runtime os20** *is used. See* *Chapter 15: Advanced configuration on page 101**.*

10.2.2 Linking legacy code

The only recommended method of linking is to use the command **st20cc -runtime os20** in conjunction with the application using the **chip** command. All new programs should follow this methodology.

Linking OS20 applications using **st20cc -Tos20.cfg** is not recommended. Do not write new code using this option. If backwards compatibility is required, link in the IntC-1 and ILC-1 support libraries.

Linking **os20.lib** directly is also no longer recommended. **os20.lib** does not contain any interrupt functions. To link legacy code, **os20.lib** should be followed by the correct combination of interrupt controller and interrupt level controller libraries (see *Table 19* and *Table 20*). Do not write new code using this option.

10.3 Initializing the interrupt handling support system

Before writing any interrupt handling routines, configure and initialize the interrupt hardware so that 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 peripherals can assert interrupt signals. This varies for each device and typically includes setting the MASK register to enable/disable individual interrupts (see *Section 10.6: Enabling and disabling interrupts*) 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 is performed automatically before the user's application starts to run.

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

```
#include <interrup.h>
int interrupt_init( int interrupt_level,
                    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, that is, 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.

10.3.1 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.

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

- 8 words of save state,
- 5 words for internal pointers, for ILC-None or ILC-1 interrupt libraries, 7 words for ILC-2 and 8 words for ILC-3,

- space for the user's initial stack frame (4 words on an ST20-C2, 3 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 a worst case of 150 words.

Note: For details of data representation; see the ST20 Embedded Toolset Reference Manual, chapter Implementation Details.

10.4 Attaching an interrupt handler in OS20

An interrupt handler is attached to an interrupt using the **`interrupt_install()`** function:

```
#include <interrup.h>
int interrupt_install( int Number,
                       int Level,
                       void (*Handler)(void* Param),
                       void* Param );
```

Once the interrupt handler is attached, the interrupt is enabled by calling **`interrupt_enable`** or **`interrupt_enable_number`** as described in *Section 10.6: Enabling and disabling interrupts*.

The function **`interrupt_install_sl()`** enables an interrupt to be installed for use with relocatable code units. For details of using OS20 with relocatable code units, see the *ST20 Embedded Toolset Reference Manual*, chapter *Building and running relocatable code*.

10.4.1 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()`** then associates the specified interrupt handler with a particular interrupt level.

Example

```
#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);

interrupt_enable_global();
```

10.4.2 Attaching interrupt handlers using an interrupt level controller

On devices which have an interrupt level controller, multiple handlers can be attached to each level, one for each interrupt number. The act of attaching the interrupt handler at a level results 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, OS20 arranges to call all the appropriate handlers for interrupts which are pending. To do this it loops, checking for pending interrupts at the current level, until there are none outstanding. When multiple interrupt numbers are pending, the numerically highest is called first.

Example

```
#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-2 or 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();
```

10.4.3 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, use **`interrupt_install()`**, and specify the interrupt level is as: **`-1`** minus the number of the interrupt output. For example:

```
/* Direct interrupt number 4 to external interrupt output 2 */
interrupt_install(4, -3, NULL, NULL);
```

10.4.4 Efficient interrupt layouts

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. 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 minimizes 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);
```

10.5 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 varies between devices, but usually involves specifying which events should cause an interrupt to be raised. The example in *Section 10.7* 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 10.15: Uninstalling interrupt handlers and deleting interrupts on page 76*.

10.6 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.

10.6.1 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.

```
#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.

10.6.2 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.

```
#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.

10.7 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, which 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 10.3 on page 67*,
- attach an interrupt handler; see *Section 10.4 on page 68*,
- program the peripheral device registers; see *Section 10.5 on page 70*,
- enable an interrupt; see *Section 10.6* above.

```
#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_BASE_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);
```

Section 10.15: Uninstalling interrupt handlers and deleting interrupts gives an example of how to remove this interrupt.

10.8 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 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 not released until an equal number of calls to **interrupt_unlock()** have been made.

Note: Locking out interrupts is slightly different from disabling an interrupt. Interrupts 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 disables channels and timers.*

A task must not deschedule with interrupts locked, as this can cause the scheduler to fail. When interrupts are locked, calling any function that may not be called by an interrupt service routine is illegal.

10.9 Raising interrupts

The following functions can be used to force an interrupt to occur:

```
#include "interrup.h"
int interrupt_raise (int Level);
int interrupt_raise_number (int Number);
```

The first function raises the specified interrupt level and should be used when peripherals are attached directly to the interrupt controller. The second function raises the specified interrupt number and is for use when an interrupt level controller is present.

Note: Neither function should be used to raise level-sensitive interrupts. These are immediately cleared by the interrupt hardware.

10.10 Retrieving details of pending interrupts

The following functions return details of pending interrupts:

```
#include <interrupt.h>
int interrupt_pending(void);
int interrupt_pending_number(void);
int interrupt_test_number(int Number);
```

interrupt_pending() returns which interrupt levels are pending, that is, those interrupts which have been set by a peripheral, but whose interrupt handlers have not yet run. This function should be used when peripherals are attached directly to the interrupt controller.

interrupt_pending_number() returns which interrupt numbers are pending, that is, all the interrupts which are currently set by peripherals. The ST20 C compiler treats **int** as a 32-bit quantity, thus **interrupt_pending_number** cannot be used on ILC-3 type interrupt level controllers because they have too many interrupt numbers.

interrupt_test_number() 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.

Note: Shared interrupt handlers should not call **interrupt_pending_number** *or* **interrupt_test_number** *since they will find the called interrupt's* PENDING *bit already reset (this is done before running the applications handler). Instead, they should use the interrupt handlers argument to differentiate between different interrupt numbers.*

10.11 Clearing pending interrupts

The following functions can be used to prevent a raised interrupt signal from causing an interrupt event to occur:

```
#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 clears 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.

10.12 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.

```
#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 is set to this default. **interrupt_trigger_mode_number** can be used to change away from the default behavior set by **interrupt_init()**.

10.13 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.

```
#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: 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.

10.14 Obtaining information about interrupts

The following two functions can be used to obtain interrupt state information:

```
#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 OS20 kernels this includes only the interrupt level to which this interrupt number is attached.

On the debug kernel, **interrupt_status()** and **interrupt_status_number()** provide extra timing information. This extra data is not available on the deployment kernel because it would decrease interrupt performance. See *Chapter 15: Advanced configuration on page 101* for further details.

10.15 Uninstalling interrupt handlers and deleting interrupts

The following function can be used to uninstall an interrupt handler:

```
#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()
interrupt_disable_number()
```

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 deletes an initialized interrupt, allowing the interrupt level's stack to be freed:

```
#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 10.7: Example: setting an interrupt for an ASC*.

```
asc->asc_intEnables = 0;
interrupt_disable(ASC_INTERRUPT_LEVEL);
interrupt_uninstall(ASC0_INTERRUPT_NUMBER, ASC_INTERRUPT_LEVEL);
interrupt_delete(ASC_INTERRUPT_LEVEL);
```

10.16 Restrictions on interrupt handlers

Certain restrictions must be kept in mind when using interrupts on the ST20.

- Descheduling and timeslicing are automatically disabled for interrupt handlers.

  Channel communications (on the ST20-C2) and any other descheduling operation are not permitted.
- On the ST20-C2 interrupt handlers must not use 2D block move functions or instructions unless the existing block move state is explicitly saved and restored by the handler.
- Interrupt handlers cannot use C++ exception handling, that is, they must not be compiled with the **st20cc** option **-exceptions**.

10.17 Interrupt header file: interrup.h

All the definitions related to interrupts are in the single header file, **`interrup.h`**; see *Table 24*.

Function	Description	ILC library ILC-			
		None	1	2	3
`interrupt_clear`	Clear a pending interrupt	✓	✓		
`interrupt_clear_number`	Clear a pending interrupt number		✓	✓	✓
`interrupt_delete`	Delete an interrupt level	✓	✓	✓	✓
`interrupt_disable`	Disable an interrupt level	✓	✓		
`interrupt_diable_global`	Global disable interrupts	✓	✓	✓	✓
`interrupt_disable_mask`	Disable one or more interrupts	✓	✓		
`interrupt_disable_number`	Disable an interrupt number			✓	✓
`interrupt_enable`	Enable an interrupt level	✓	✓		
`interrupt_enable_global`	Globally enable interrupts	✓	✓	✓	✓
`interrupt_enable_mask`	Enable one or more interrupts	✓	✓		
`interrupt_enable_number`	Enable an interrupt number			✓	✓
`interrupt_init`	Initialize an interrupt level	✓	✓	✓	✓
`interrupt_init_controller`	Initialize the interrupt controller	✓	✓	✓	✓
`interrupt_install`	Install an interrupt handler	✓	✓	✓	✓
`interrupt_install_sl`	Install an interrupt handler and specify a static link	✓	✓	✓	✓
`interrupt_lock`	Lock all interrupts	✓	✓	✓	✓
`interrupt_pending`	Return pending interrupt levels	✓	✓		
`interrupt_pending_number`	Return pending interrupt numbers	✓	✓	✓	
`interrupt_raise`	Raise an interrupt level	✓	✓		
`interrupt_raise_number`	Raise an interrupt number		✓	✓	✓
`interrupt_status`	Report the status of an interrupt level	✓	✓	✓	✓
`interrupt_status_number`	Report the status of an interrupt number		✓	✓	✓
`interrupt_test_number`	Test whether an interrupt number is pending		✓	✓	✓
`interrupt_trigger_mode_number`	Change the trigger mode of an interrupt number			✓	✓
`interrupt_uninstall`	Uninstall an interrupt handler	✓	✓	✓	✓
`interrupt_unlock`	Unlock all interrupts	✓	✓	✓	✓
`interrupt_wakeup_number`	Set wakeup status of an interrupt number			✓	✓

Table 24: Functions defined in interrup.h

The full ILC library names are given in *Table 20 on page 64*.

Types and macros defined to support interrupts are listed in *Table 25* and *Table 26*.

Types	Description
interrupt_flags_t	Additional flags for **interrupt_init**
interrupt_status_t	Structure describing the status of an interrupt level
interrupt_status_flags_t	Additional flags for **interrupt_status**
interrupt_status_number_t	Structure describing the status of an interrupt number
interrupt_status_number_flags_t	Additional flags for **interrupt_status_number**
interrupt_trigger_mode_t	Interrupt trigger modes (used in **interrupt_init**)

Table 25: Types defined in interrup.h

Macro	Description
INTERRUPT_GLOBAL_ENABLE	Global interrupt enables bit number

Table 26: Macros defined in interrup.h

Device information 11

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. 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 breaks the device code down into a family and a 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 shown below in *Table 27*.

Bits	jtag	st	Meaning
[31:28]	**revision**	**revision**	Mask revision
[27:22]	**device_code**	**family**	20_{10} – CMG DVD family
[21:12]		**device_code**	Device code
[11:1]	**manufacturer**	**manufacturer**	32_{10} – STMicroelectronics
[0]	**JTAG_bit**	**JTAG_bit**	1 – fixed by JTAG

Table 27: Composition of identification code

11.1 Device ID header file: device.h

All the definitions related to device identification are in the single header file, **`device.h`**; see *Table 28*.

Function	Description
`device_id`	Returns the ID of the current device
`device_name`	Returns the name of the current device

Table 28: Functions defined in device.h

Types	Description
`device_id_t`	Device ID

Table 29: Types defined in device.h

12 Caches

Cache provides a way to reduce the time taken for the CPU to access memory and so can greatly increase system performance.

12.1 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.

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: 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.*

12.1.1 Data caches with internal SRAM

Some ST20 devices have a data cache which must be reserved by the linker in order to prevent it from being corrupted by the application. This is described in the *ST20 Embedded Toolset User Manual*, chapter *Defining a target system*.

12.2 Initializing 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 OS20 knows which hardware model is being targeted.

If the **st20cc -runtime os20** command is used when linking, the cache controller is 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:

```
#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 *Chapter 16: Alphabetical list of functions*, function *cache_init_controller on page 126*.

Note: Some ST20 devices have two base addresses, one for the instruction cache and one for the data cache, for example the STm5700 and the STi5516. It is the base address of the instruction cache that should be passed to the **cache_init_controller** *function.*

12.3 Configuring the caches

On any ST20 device with a data cache the **cache_config_data** function is used to configure the data cache to treat certain blocks of memory as cacheable or non-cacheable.

Note: By default all configurable blocks are set to non-cacheable, therefore for all devices with a data cache, the use of the **cache_config_data** *function is vital to achieve maximum performance.*

```
#include <cache.h>
int cache_config_data( void* start_address,
                       void* end_address,
                       cache_config_flags_t flags );
```

There are two types of ST20 instruction cache: configurable and fixed. A fixed instruction cache can only be enabled or disabled; it cannot be selectively applied to specific blocks of memory.

On devices which have a configurable instruction cache, the function **cache_config_instruction** is used to enable or disable specific blocks of memory. A configurable instruction cache, like the data cache, treats all configurable blocks as non-cacheable by default. For devices with a configurable instruction cache the use of **cache_config_instruction** is necessary to achieve maximum performance.

```
#include <cache.h>
int cache_config_instruction( void* start_address,
                              void* end_address,
                              cache_config_flags_t flags);
```

12.4 Enabling and disabling the caches

The caches are enabled using the following two functions:

```
#include <cache.h>
int cache_enable_data();
int cache_enable_instruction();
```

The first function invalidates the data cache (see *Section 12.7: Flushing and invalidating caches*) before writing to the ENABLEDCACHE register thereby enabling the data cache. The second function is similar but operates on the ENABLEICACHE register.

If the target application requires the caches to be disabled at some later point the following two functions can be used.

```
#include <cache.h>
int cache_disable_data();
int cache_disble_instruction();
```

Disabling the cache can potentially take a long time to complete; during this time the processor is unable to handle interrupts or perform any other time-critical task.

12.5 Locking the cache configuration

The cache can be locked using the following function:

```
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.

12.6 Example: setting up the caches

This example shows how the caches could be set up for an STi5516 device. The example demonstrates the steps described in the previous sections to:

- initialize and configure the cache hardware; see *Section 12.2*,
- enable the data and instruction caches; see *Section 12.4*,
- lock the cache configuration; see *Section 12.5*.

The example uses the header file **<chip/STi5516addr.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**.

```
#include <chip/STi5516addr.h>
#include <cache.h>

cache_init_controller((void*) CacheControlAddr, cache_map_c2_c200);

/* Configure instruction caches to cache all possible memory */
cache_config_instruction((void *)0x80000000, (void *)0x7fffffff,
cache_config_enable);

/* Configure data caches to cache all possible memory... */
cache_config_data((void *)0x80000000, (void *)0x7fffffff,
cache_config_enable);

/* ...except region required for DMA */
cache_config_data((void *)0x40010000, (void *)0x4001ffff,
cache_config_disable);

cache_enable_instruction();
cache_enable_data();
cache_lock();
```

12.7 Flushing and invalidating caches

When the cache is enabled, any data written to main memory is 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: On ST20 devices, flushing the cache also causes it to be invalidated. After a cache flush all data is reloaded 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:

```
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 OS20, users must supply **NULL** as each argument.

12.7.1 Relocatable code units

When caches are enabled, extra care must be taken when handling relocatable code units. To ensure cache coherency is maintained, follow the advice given in the *ST20 Embedded Toolset Reference Manual*, chapter *Building and running relocatable code*.

12.8 Cache header file: cache.h

All the definitions related to the caches are in the single header file, **`cache.h`**; see *Table 30*.

Function	Description
`cache_config_data`	Configure the data cache
`cache_config_instruction`	Configure the instruction cache
`cache_disable_data`	Disable the data cache
`cache_disable_instruction`	Disable the instruction cache
`cache_enable_data`	Enable the data cache
`cache_enable_instruction`	Enable the instruction cache
`cache_flush_data`	Flush the data cache
`cache_init_controller`	Initialize the cache controller
`cache_invalidate_data`	Invalidate the data cache
`cache_invalidate_instruction`	Invalidate the instruction cache
`cache_lock`	Lock the cache configuration
`cache_status`	Report the cache status

Table 30: Functions defined in cache.h

The types defined to support the cache API are listed in *Table 31*.

Types	Description
`cache_config_flags_t`	Additional flags for `cache_config_data`
`cache_map_data_t`	Description of cacheable memory available on a particular ST20 variant (used by `cache_init_controller`)
`cache_status_t`	Structure describing the status of the cache

Table 31: Types defined in cache.h

ST20-C1 specific features

13

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()`**.

The ST20-C1 core does not have a built-in timer peripheral. In order for the ST20-C1 version of OS20 to provide the full API, you need to incorporate a timer plug-in module into any applications built for the ST20-C1 cores. Plug-in modules are board-specific and must be written to manage whatever hardware is present on the development board; OS20 contains a generic timer plugin that uses the PWM peripheral if present, see *Section 13.1: In-built PWM support on page 89* for more details.

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 occurs, so take care when not using the plug-in module.

Internally, OS20 uses a standardized low level timer API, which accesses functions provided by the plug-in module via function pointers; see *Figure 9*. This is so that the application can be built with or without the plug-in module. Linkage between 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.

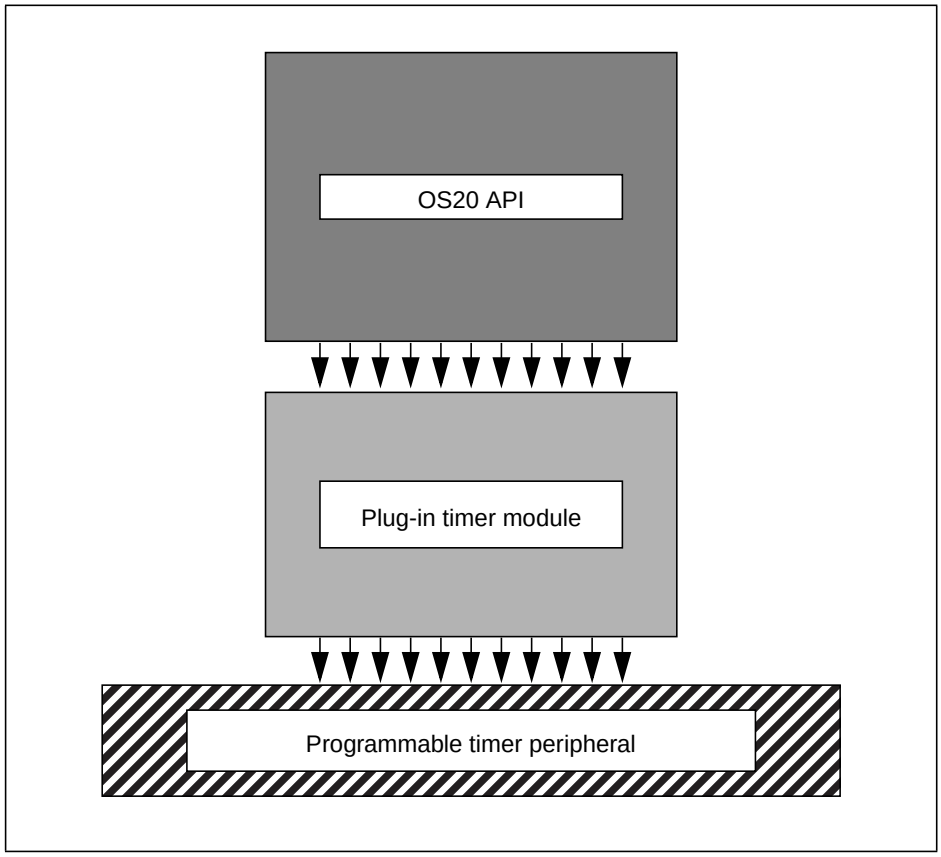

Figure 9: Plug-in timer model for the ST20-C1

The plug-in module must provide an initialization function which the application can call. Upon calling the initialization function, the module initializes the programmable timer and passes a structure detailing all of the functions' locations into OS20 via a function called **`timer_initialize`**. This is an OS20 ST20-C1 specific function call. At this stage the plug-in module is linked into the OS20 kernel.

The syntax of the timer API is consistent with the remainder of OS20. The naming convention has an object-oriented approach:

`<class>_<type_of_operation>`

Function	Description
`timer_read`	Read the timer
`timer_set`	Set the timer
`timer_enable_int`	Enable the timer interrupt
`timer_disable_int`	Disable the timer interrupt
`timer_raise_int`	Raise a timer interrupt

Table 32: Internal OS20 Timer API

Before the plug-in module is initialized, the OS20 kernel must be initialized and started by calling **`kernel_initialize()`** and **`kernel_start()`**; additionally the interrupt controller must be initialized by calling **`interrupt_init_controller()`**.

Note: The plug-in timer module's tick frequency is self-defined, because it is external to OS20. This arrangement has the advantage that the tick frequency can be tailored for specific applications: a short tick for high accuracy, a longer tick for timing long periods. Take care when porting code to different devices because the tick frequency is likely to change.

13.1 In-built PWM support

OS20 includes in-built support for PWM based timer peripherals. PWM timers are included in the majority of ST20-C1 based devices.

`timer_pwm_init()` is used to initialize the in-built timer support. This function is supplied with the PWM's input frequency and attempt to run both the system clock and the timeslice at rates similar to those found on ST20-C2 based parts. Specifically it attempts to provide a system clock running at 15625 ticks per second and a quiessant timeslice of 2 ms.

Note: Few PWM devices are capable of directly providing a 15625 tick rate so this should never appear as a manifest constant in ST20-C1 code; use the **`time_ticks_per_sec()`** *call instead.*

`timer_pwm_init()` requires an interrupt level to have been initialized before it is called. Similarly it requires the interrupt to be enabled after it has been called. The following is a typical initialization sequence.

```
/* Generally the operating system timer should run as the least
* priority interrupt.
*/
err = interrupt_init(0, stack, sizeof(stack),
         interrupt_trigger_mode_rising, 0);
/* error checking */

err = timer_init_pwm((void *) PWM_BASE_ADDRESS, PWM_INTERRUPT, 0,
         PWM_FREQ_IN_KHZ, 0);
/* error checking */

#ifdef ILC1
interrupt_enable(0);
#else
interrupt_enable_number(PWM_INTERRUPT);
#endif
interrupt_enable_global();
```

13.2 ST20-C1 example plug-in timer module

A plug-in module is provided as example code for ST evaluation boards. This can be found in the **`examples/os20/c1timer`** directory. The readme file supplied with the example explains how to build and run the example.

This example contains completely separate timer modules named after the products they work with. These can each be used standalone if required (that is, only one need be linked to your application). However, the supplied example has a single timer initialize function called **`c1_timer_initialize`** that uses **`device_id`** to determine which timer module to use.

13.2.1 PWM peripheral

Both timer plug-in modules use the on-chip PWM peripheral to provide the timer functionality. This peripheral is described here in sufficient detail to explain how the example works.

The PWM peripheral has a programmable timer which you program to cause an interrupt at a specified time.

The CAPTURECOUNT register is a 32-bit counter that is incremented regularly. The COMPARE register is set by your application. When the value in the CAPTURECOUNT register becomes equal to the value in the COMPARE register an interrupt is generated.

Table 33 provides a list of registers which are actively used by the plug-in module.

Register	Description
CONTROL	Used to initialize PWM peripheral
INTERRUPTENABLE	Enable and disable interrupts by this register
CAPTURECOUNT	32-bit counter
COMPARE	Time at which an event should occur

Table 33: PWM registers used by the plug-in module

Control

The CONTROL register controls the top level function of the PWM peripheral. In particular, it contains a Capture enable bit that causes the CAPTURECOUNT register to start counting and a Capture prescale value which controls the rate at which the CAPTURECOUNT register runs. By default, the prescale value is set to 0.

InterruptEnable

The INTERRUPTENABLE controls which events will cause an interrupt to be asserted. In particular, the register contains a bit which when set causes an interrupt to be asserted, the CAPTURECOUNT register then 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 compared against the CAPTURECOUNT register. When these are equal, the timer requests an interrupt, depending on the state of the INTERRUPTENABLE register.

13.3 Plug-in timer module header file: c1timer.h

All the definitions related to ST20-C1 plug-in timer modules are defined in a single header file, **`c1timer.h`**; see *Table 34*.

Function	Description
`timer_initialize`	Initialize the timer plug-in module
`timer_interrupt`	Notify OS20 that the timer has expired

Table 34: Functions defined in c1timer.h

Types	Description
`timer_api_t`	Set of function pointers to be used as a plug-in timer module

Table 35: Types defined in c1timer.h

ST20-C2 specific features 14

14.1 Overview

The ST20-C2 has the following additional features over the ST20-C1.

- **Channels**

  The ST20-C2 supports 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 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 always preempt normal 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 normal OS20 tasks.

  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 OS20 scheduling regime, they only have very limited access to 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 for graphical applications.

14.2 Channels

OS20 supports the use of channels by all tasks (both normal/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 waits until the other communicates. The data is copied from the memory of the sending task to the memory of the receiving task and both tasks 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 amounts of data sent and received are the same,
- the types 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 OS20 scheduler is enabled (by calling **`kernel_start`**), channel communication results in traps to the kernel, which ensure that correct scheduling semantics are maintained.

14.2.1 Creating a channel

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:

```
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:

```
#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);
```

14.2.2 Communications over channels

Once a channel has been initialized, there are several functions available for communications:

```
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 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 communications function call represents a single communication. The task does not continue until the transfer is complete.

Take care 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 uses channel **my_chan** to send a character followed by an integer followed by a string:

```
#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:

```
#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);
```

14.2.3 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 OS20 by the following function:

```
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 in *Section 14.2.2*. Any channels that become ready and are not read 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** is used with the time-out value **TIMEOUT_INFINITY**, in which case only one of the channels becoming ready (that is, one of the sending tasks that is trying to send) will cause it to return. When one or more channels are ready then one is selected. If no channel becomes ready then the function will wait for ever.

Note: The header file **ostime.h** *must be included when using this function.*

To read from an array of channels, the returned index can be used as an index into the channel array, for example:

```
#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 tasks while the task is waiting.

When it is necessary to poll channels, this can be performed by specifying a timeout of **TIMEOUT_IMMEDIATE**. This causes 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**:

```
#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 9: Real-time clocks on page 57*.

14.2.4 Deleting channels

Channels may be deleted using *chan_delete*; see *Chapter 16: Alphabetical list of functions on page 111* for full details.

14.2.5 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 36* and *Table 37*.

Function	Description
chan_alt	Wait for input on one of a number of channels
chan_create	Create a soft channel
chan_create_address	Create a hard channel
chan_delete	Delete a channel
chan_in	Read data from a channel
chan_in_char	Read character from a channel
chan_in_int	Read integer from a channel
chan_init	Initialize a channel
chan_init_address	Initialize a hardware channel
chan_out	Write data to a channel
chan_out_char	Write character to a channel
chan_out_int	Write integer to a channel
chan_reset	Reset channel

Table 36: Functions defined in chan.h

Types	Description
chan_t	A channel

Table 37: Types defined in chan.h

14.3 Two dimensional block move support

Graphical applications often require the movement of two dimensional blocks of data, for example to perform windowing, overlaying. 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 2-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.

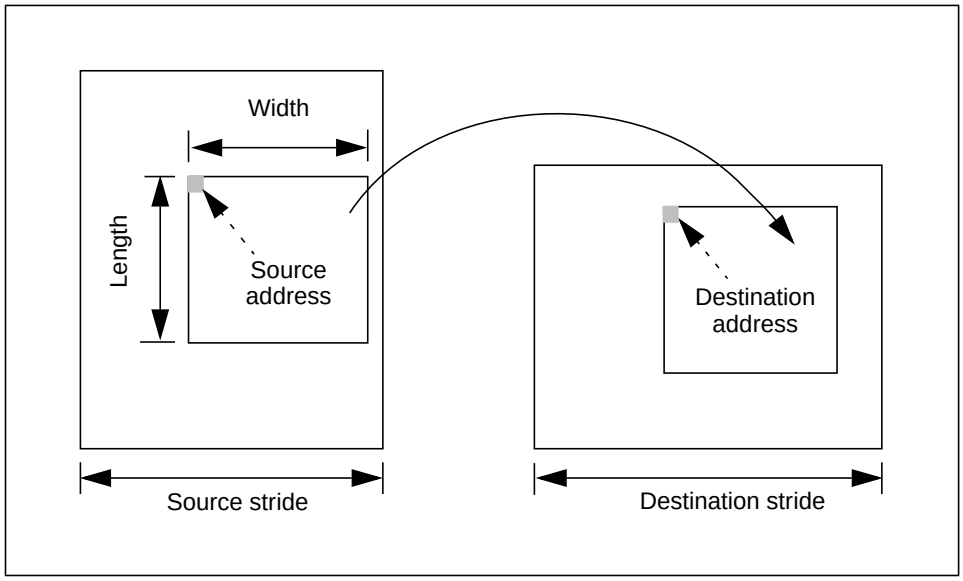

Figure 10: Two dimensional block move

To perform a two dimensional move, 6 parameters are required (see *Figure 10*).

Source address The address of the first element of the source block to be copied

Destination address The address of the first element of the destination block

Block width The number of bytes in each row in the block to be copied

Block length The number of rows in the block to be copied

Source stride The number of bytes in each row in the source array

Destination stride The number of bytes in each row in the destination array

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 block width or length is 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. If the source and destination blocks overlap, the effect of the two dimensional moves is undefined.

Instructions are provided which allow a whole block to be moved, or only the zero or nonzero values.

OS20 provides three functions which give access to these instructions:

```
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:

- **move2d_all** copies the whole of the block of ***nrows*** rows each of ***width*** 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.

14.3.1 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 38*.

Function	Description	Callable from ISR/HPP
move2d_all	Two dimensional block move	HPP
move2d_non_zero	Two dimensional block move of nonzero bytes	HPP
move2d_zero	Two dimensional block move of zero bytes	HPP

Table 38: Functions defined in move2d.h

All functions are callable from an OS20 task or a high priority process (HPP), however, none of them can be called from an interrupt service routine.

Advanced configuration

15

Two OS20 kernels are supplied with the ST20 toolset. A deployment kernel is preconfigured for use in a wide variety of applications and is the standard kernel. The second kernel is a debug kernel which provides time logging. The two kernels are described in *Chapter 3: Kernel on page 17*, and briefly in the *ST20 Embedded Toolset User Manual*, chapter *st20cc compile/ link tool*.

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 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.

The OS20 function descriptions can be found in *Chapter 16: Alphabetical list of functions on page 111*.

15.1 Run-time configuration

The OS20 run-time system is most easily invoked by specifying the **-runtime** option on the **st20cc** command line. The default actions that occur when the **st20cc** option **-runtime os20** is used are described in *Chapter 2: Getting started on page 9*. These generic defaults may not be suitable for all applications, therefore 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, since it would cause unused code and data to be linked in. To suppress this initialization, the user may specify a configuration file, **options.cfg**, as follows:

```
## do not install interrupt initialization
OS20_config.initialize_interrupt_first=0
```

This configuration can be supplied to the linker as follows:

```
st20cc -p STi5516MB382 <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.

OS20 is provided with an example configuration file: **$ST20ROOT/lib/os20conf.cfg**. This file lists all the 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: This file is not read by OS20 at link time. Modifying it directly does not change the default behavior.

15.1.1 Specifying initialization code

By default the OS20 run-time 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 OS20 such as the interrupt and cache systems have been initialized.

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.

15.1.2 Specifying placement of code and data

By default the OS20 run-time 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 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 **os20conf.cfg** located in the **lib** directory of the toolset installation.

Note: On devices with only two kilobytes of internal SRAM (for example, 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 OS20 kernel does 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.

15.1.3 Caching peripheral memory in larger blocks

Currently this option applies only to devices which use **cache_map_sti5512a** or **cache_map_sti5512b** (shown in *Table 44: Cache maps on page 126*).

By default, these devices have sixteen 64 Kbyte blocks, in the range 0xC0000000 to 0xC00FFFFF, for which the data cache can be selectively enabled and disabled in order to support non-coherent DMA peripherals.

Since these devices can support much larger memories than older chips, it is possible to configure them to have sixteen 512 Kbyte blocks in the range 0xC0000000 to 0xC07FFFFF instead. This permits much larger sections of DMA memory.

The option:

```
OS20_config.sti551x_cache_512kbyte_blocks
```

selects the larger cache blocks at link time.

15.1.4 Making devices with ILC-2 strictly backward compatible

By default, the interrupt level controller ILC-2 provides the same programmer interface as the interrupt level controller ILC-3 since they both provide similar features. Unfortunately this means that existing application code has to be modified to make it run on the ILC-2. (See *Chapter 10: Interrupts on page 61*).

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:

```
OS20_config.interrupt_force_ilc1
```

15.1.5 Altering the internal partition manager

The internal partition automatically created by **-runtime os20** uses the simple partition manager in order to conserve internal memory. The simple partition manager does not permit memory to be freed. Therefore if the application has any requirement to reuse internal memory it must alter the internal partition to permit memory to be freed.

This can be acheived by allocating all remaining memory from the original internal partition and creating a new internal partition. Since the simple partition manager has a zero memory overhead this does not waste any internal memory.

The code to migrate the internal partition is shown below:

```
partition_status_t status;
partition_status(internal_partition, &status, 0);
internal_partiton = partition_create_heap(
  memory_allocate(internal_parititon,
    status.partition_status_free_largest),
  status.partition_status_free_largest);
assert(internal_partition);
```

Note: No other threads can use the internal partition while it is being migrated. Even if this code were made thread-safe by retrying the allocation of the large block then other threads would fail since they would not be able to allocate memory.

15.2 Compiling OS20

There are a number of extensions that are not supported by the pre-built 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. The extensions are most useful when debugging, because some of them give greater visibility of what OS20 is doing internally, 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 OS20 is located in **$ST20ROOT/src/os20.** Copy this directory locally and set an environment variable **MYOS20** to point to the copied directory. OS20 is provided with makefiles for Sun **make** under Solaris, GNU **make** under Linux and Microsoft **nmake** under Windows.

From the source directory OS20 can be built under Solaris as follows:

```
make -f makefile.top c1dxx
```

Under Linux, OS20 can be built from the source directory with the following command:

```
make -f makefile.top c1dxx-linux
```

Under Windows, OS20 can be built from the source directory with the command:

```
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 and Linux, add the following options to the **st20cc** command line each time it is used:

```
st20cc -I$MYOS20/dist-cx/lib -L$MYOS20/dist-cx/lib ...
```

Similarly for Windows:

```
st20cc -I%MYOS20%\dist-cx\lib -L%MYOS20%\dist-cx\lib ...
```

For details on how to make the above options permanent, see the *ST20 Embedded Toolset User Manual*, chapter *st20cc compile/link tool*.

15.3 Compilation option file: conf.h

The **conf.h** header file is included by every source file in 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 in the following section.

15.3.1 Callback Support

Callback support is enabled using the option **CONF_CALLBACK_SUPPORT**. This causes 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 *Chapter 16: Alphabetical list of functions on page 111*.

Note: To increase performance the 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.

15.3.2 Changing the number of task priority levels

The pre-built OS20 kernel supports 16 priority levels. There are two factors which affect the number of priority levels the scheduler can support.

By default the scheduler code supports up to 32 priority levels on a ST20-C2 and16 priority levels on a ST20-C1. However this can be extended for both cores, 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**.

15.3.3 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-entry can only occur if a high priority interrupt or high priority process performs an operation which generates 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 take place, as long as it is through a mechanism which defers 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).

15.3.4 Time logging (ST20-C2 core only)

OS20 can be configured to maintain a record of the amount of time each task spends running on the processor. This feature is always enabled when the prebuilt debug kernel is selected using the **st20cc** option **-debug-runtime** (see the introduction at the start of this chapter).

Time logging is enabled by defining specific compile-time options in the **conf.h** file introduced in *Section 15.3: Compilation option file: conf.h*. These options may either be defined automatically using the **-debug-runtime st20cc** option or manually by editing **conf.h**.

- **CONF_TIME_LOGGING** enables task time logging and idle time logging. The data collected is accessed using the **task_status** function.
- **CONF_INTERRUPT_TIME_LOGGING** enables interrupt time logging. The recorded data is accessed using the functions **interrupt_status** and **interrupt_status_number**.

See *Chapter 16: Alphabetical list of functions on page 111* for function descriptions. Task time logging and interrupt time logging are described in *Section 3.2: Optional debug features on page 18*.

The ST20-C2 core has two internal 32-bit timers that run at different speeds. Either timer can be used for time logging; the one used is selected based upon the value of **CONF_TIME_LOGGING_PRIORITY**, defined in **conf.h**. If this is set to **0** then the high resolution clock is used, with 1 µs resolution, and a maximum time of about 71 minutes before the timer wraps around. If this is set to **1** then the low resolution clock is used, with 64 µs resolution, and a maximum time of about 76 hours before the timer wraps around.

Using **CONF_TIME_LOGGING** and **CONF_INTERRUPT_TIME_LOGGING** to control time logging has the advantage of being selective about which type of time logging is enabled. If the debug kernel is linked, all types of time logging are enabled.

Note: All time logging is slightly intrusive. Logging is performed by the target in the scheduler trap and when an interrupt is handled. This could subtly alter the real time performance of the system being logged, however, in most cases the difference in performance should be negligible.

15.3.5 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 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 cannot provide this information. To support software interrupts, extra code is added to one of the 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 is rarely necessary because no code is added to the high performance interrupt handler. Refer to *Chapter 10: Interrupts on page 61* for details on efficient interrupt layout.

15.3.6 Mutex initialization

The function **kernel_initialize()** initializes mutex code for the following parts of the C run-time system:

- **stdio** functions,
- device-independent I/O functions,
- debug functions,
- heap functions.

If any of these mutexes are not required there is an opportunity to reduce the memory requirements of OS20. This is done by editing **conf.h** to define the appropriate configuration options as listed in *Table 39 on page 108*.

Mutex	Configuration option to disable mutex
stdio functions	**CONF_NO_STDIO_MUTEX_INIT**
Device-independent I/O functions	**CONF_NO_DEVICEIO_MUTEX_INIT**
Debug functions	**CONF_NO_DEBUG_MUTEX_INIT**
Heap functions	**CONF_NO_HEAP_MUTEX_INIT**
All mutex code for the functions listed above.	**CONF_NO_MUTEX_INIT**

Table 39: Configuration options to disable mutex

The **stdio** layer uses the device-independent I/O layer and debug layer, so if **stdio** is not used, but debug functions are used, both the **stdio** and the device-independent I/O mutexes may be disabled.

When using C++, **kernel_initialize()** also initializes mutex code for C++ exception handling, I/O streams mutex protection and generic C++ mutex code (which is used by the I/O streams mutex code and for protecting function local static constructions).

CONF_NO_GENERIC_CPP_MUTEX_INIT is the configuration option to disable the generic C++ mutex code, **CONF_NO_EXCEPTION_MUTEX_INIT** disables the mutex code for thread-safe exception handling and **CONF_NO_DINKUM_THREAD_PROT_INIT** disables the IOstreams thread-safety initialization code.

15.4 Performance considerations

This section gives some hints on how to place portions of OS20 in memory to optimize performance. Normally the defaults 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.

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 the trap handler code is normally executed with all interrupts disabled, and so can affect interrupt latency. Thus there are usually two objectives when trying to tune 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; timings are 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. This breakdown allows the **tdesc_t** to be moved into on-chip memory.

There are five sections which OS20 uses:

- trap handler code (**os20_th_code** section),
- trap handler workspace (including many 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 are 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 decrease the time for a context switch by 30%, while moving **tdesc**s, queues and interrupt stacks on chips only yields a context switch latency improvement of 9%.

For the remaining three categories, the choices are not so clear. Moving task queues and **tdesc**s on-chip brings 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 improves the context switch times to those tasks which have their **tdesc**s on-chip, although this does 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 OS20 also benefits; however it 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.

Alphabetical list of functions 16

16.1 Header files

Table 40 lists the supplied OS20 header files. The functions defined in these header files are listed in *Table 41*. Full descriptions of the functions can be found in *Section 16.2*.

Header	Description
cache.h	Cache functions
callback.h	Callback functions
chan.h	Channel functions (ST20-C2 specific)
device.h	Device information functions
interrup.h	Interrupt handling support functions
kernel.h	Kernel functions
message.h	Message handling functions
move2d.h	Two dimensional block move functions (ST20-C2 specific)
mutex.h	Mutex functions
partitio.h	Memory functions
semaphor.h	Semaphore functions
tasks.h	Task functions
ostime.h	Timer functions
c1timer.h	ST20-C1 timer functions

Table 40: OS20 header files

Function	Description
Header file: cache.h	
cache_config_data	Configure the data cache
cache_config_instruction	Configure the instruction cache
cache_disable_data	Disable the data cache
cache_disable_instruction	Disable the instruction cache
cache_enable_data	Enable the data cache
cache_enable_instruction	Enable the instruction cache
cache_flush_data	Flush the data cache
cache_init_controller	Initialize the cache controller
cache_invalidate_data	Invalidate the data cache
cache_invalidate_instruction	Invalidate the instruction cache
cache_lock	Lock the cache configuration
cache_status	Report the cache status
Header file: callback.h	
callback_...	Register a callback for an event
Header file: chan.h	
chan_alt	Wait for input on one of a number of channels
chan_create	Create a soft channel
chan_create_address	Create a hard channel
chan_delete	Delete a channel
chan_in	Read data from a channel
chan_in_char	Read character from a channel
chan_in_int	Read integer from a channel
chan_init	Initialize a soft channel
chan_init_address	Initialize a hardware channel
chan_out	Write data to a channel
chan_out_char	Write character to a channel
chan_out_int	Write integer to a channel
chan_reset	Reset channel
Header file: device.h	
device_id	Return the ID of the current device
device_name	Return the name of the current device

Table 41: OS20 functions

Function	Description
Header file: interrup.h	
interrupt_clear	Clear a pending interrupt level
interrupt_clear_number	Clear a pending interrupt number
interrupt_delete	Delete an interrupt level
interrupt_disable	Disable an interrupt level
interrupt_disable_global	Disable interrupts globally
interrupt_disable_mask	Disable one or more interrupt levels
interrupt_disable_number	Disable an interrupt number
interrupt_enable	Enable an interrupt level
interrupt_enable_global	Enable interrupts globally
interrupt_enable_mask	Enable one or more interrupt levels
interrupt_enable_number	Enable an interrupt number
interrupt_init	Initialize an interrupt level
interrupt_init_controller	Initialize the interrupt controller
interrupt_install	Install an interrupt handler
interrupt_install_sl	Install an interrupt handler and specify a static link
interrupt_lock	Lock all interrupts
interrupt_pending	Return pending interrupt levels
interrupt_pending_number	Return pending interrupt numbers
interrupt_raise	Raise an interrupt level
interrupt_raise_number	Raise an interrupt number
interrupt_status	Report the status of an interrupt level
interrupt_status_number	Report the status of an interrupt number
interrupt_test_number	Test whether an interrupt number is pending
interrupt_trigger_mode_number	Change the trigger mode of an interrupt number
interrupt_uninstall	Uninstall an interrupt handler
interrupt_unlock	Unlock all interrupts
interrupt_wakeup_number	Set wakeup status of an interrupt number
Header file: kernel.h	
kernel_idle	Return the kernel idle time
kernel_initialize	Initialize for preemptive scheduling
kernel_start	Start preemptive scheduling regime
kernel_time	Return the kernel up-time
kernel_version	Return the OS20 version number

Table 41: OS20 functions

Function	Description
Header file: message.h	
message_claim	Claim a message buffer
message_claim_timeout	Claim a message buffer or timeout
message_create_queue	Create a fixed size message queue
message_create_queue_timeout	Create a fixed size message queue with timeout
message_delete_queue	Delete a message queue
message_init_queue	Initialize a fixed size message queue
message_init_queue_timeout	Initialize a fixed size message queue with timeout
message_receive	Receive the next available message from a queue
message_receive_timeout	Receive the next available message from a queue or timeout
message_release	Release a message buffer
message_send	Send a message to a queue
Header file: move2d.h	
move2d_all	Two dimensional block move
move2d_non_zero	Two dimensional block move of non-zero bytes
move2d_zero	Two dimensional block move of zero bytes
Header file: mutex.h	
mutex_create_fifo	Create a FIFO queued mutex
mutex_create_priority	Create a priority queued mutex
mutex_delete	Delete a mutex
mutex_init_fifo	Initialize a FIFO queued mutex
mutex_init_priority	Initialize a priority queued mutex
mutex_lock	Acquire a mutex, block if not available
mutex_release	Release a mutex
mutex_trylock	Try to get a mutex, fail if not available
Header file: partitio.h	
memory_allocate	Allocate a block of memory from a partition
memory_allocate_clear	Allocate a block of memory from a partition and clear to zero
memory_deallocate	Free a block of memory back to a partition
memory_reallocate	Reallocate a block of memory from a partition
partition_create_fixed	Create a fixed partition
partition_create_heap	Create a heap partition
partition_create_simple	Create a simple partition
partition_delete	Delete a partition

Table 41: OS20 functions

Function	Description
partition_init_fixed	Initialize a fixed partition
partition_init_heap	Initialize a heap partition
partition_init_simple	Initialize a simple partition
partition_status	Get the status of a partition
Header file: semaphor.h	
semaphore_create_fifo	Create a FIFO queued semaphore
semaphore_create_fifo_timeout	Create a FIFO queued semaphore with timeout
semaphore_create_priority	Create a priority queued semaphore
semaphore_create_priority_timeout	Create a priority queued semaphore with timeout
semaphore_delete	Delete a semaphore
semaphore_init_fifo	Initialize a FIFO queued semaphore
semaphore_init_fifo_timeout	Initialize a FIFO queued semaphore with timeout
semaphore_init_priority	Initialize a priority queued semaphore
semaphore_init_priority_timeout	Initialize a priority queued semaphore with timeout
semaphore_signal	Signal a semaphore
semaphore_wait	Wait for a semaphore
semaphore_wait_timeout	Wait for a semaphore or a timeout
Header file: task.h	
task_context	Return the current execution context
task_create	Create an OS20 task
task_create_sl	Create an OS20 task specifying a static link
task_data	Retrieve a task's data pointer
task_data_set	Set a task's data pointer
task_delay	Delay the calling task for a period of time
task_delay_until	Delay the calling task until a specified time
task_delete	Delete a task
task_exit	Exit the current task
task_id	Find the current task's id
task_immortal	Make the current task immortal
task_init	Initialize an OS20 task
task_init_sl	Initialize an OS20 task specifying a static link
task_kill	Kill a task
task_lock	Prevent task rescheduling
task_mortal	Make the current task mortal

Table 41: OS20 functions

Function	Description
task_name	Returns the task's name
task_onexit_set	Setup a function to be called when a task exits
task_onexit_set_sl	Setup a function to be called when a task exits and specify a static link
task_priority	Retrieve a task's priority
task_priority_set	Set a task's priority
task_private_data	Retrieve a task's private data pointer
task_private_data_set	Set a task's private data pointer
task_reschedule	Reschedule the current task
task_resume	Resume a suspended task
task_stack_fill	Return the task fill configuration
task_stack_fill_set	Set the task stack fill configuration
task_status	Return status information about the task
task_suspend	Suspend a task
task_unlock	Allow task rescheduling
task_wait	Wait until one of a list of tasks completes
Header file: ostime.h	
time_after	Return whether one time is after another
time_minus	Subtract two clock values
time_now	Return the current time
time_plus	Add two clock values
time_ticks_per_sec	Obtain the current system clock rate
time_ticks_per_sec_set	Specify the number of ticks per second observed on a hardware device
timer_init_pwm	Use OS20's in built timer management code for ST20-C1
Header file: c1timer.h	
timer_initialize	Initialize the timer plug-in library for ST20-C1 cores
timer_interrupt	Notify OS20 that the timer has expired

Table 41: OS20 functions

16.2 OS20 function descriptions

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

void* *start_address*	Start of address range
void* *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** if any of the following are 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 affects all cacheable regions between ***start_address*** and ***end_address***, neither of which 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 wrap around from 0xFFFFFFFF to 0x00000000. To cache all possible memory the following ST20 address range may be specified:

```
cache_config_data ((void *)0x80000000, (void *)0x7fffffff ...);
```

Alternatively, the address range in this example produces the same result:

```
cache_config_data ((void *)0x00000000, (void *)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 42*.

Data cache configuration flags	Data cache configuration behavior
`cache_config_enable`	Enable caching for the specified range
`cache_config_disable`	Disable caching for the specified range

Table 42: Data cache configuration flags

Note: 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 actually affects both blocks. Refer the device datasheet for further details.*

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
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_config_instruction

Configure the instruction cache

Synopsis

```
#include <cache.h>

int cache_config_data( void* start_address,
                       void* end_address,
                       cache_config_flags_t flags );
```

Arguments

void* *start_address*	Start of address range
void* *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 target device does not have a configurable instruction cache,
- the flags are invalid.

Description

This function writes to the CACHECONTROL registers to enable or disable instruction caching for the specified range. It affects all cacheable regions between ***start_address*** and ***end_address***, neither of which may fall in the middle of a cacheable region. Refer to the appropriate datasheet to find the instruction cache regions for a specific ST20 device.

The ST20 memory map runs from **MININT** to **MAXINT**, therefore addresses supplied to this function wrap around from 0xFFFFFFFF to 0x00000000. To cache all possible memory the following ST20 address range may be specified:

```
cache_config_instruction ((void *)0x80000000, (void *)0x7fffffff ...);
```

Alternatively, the address range in this example produces the same result:

```
cache_config_instruction ((void *)0x00000000, (void *)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 43*.

Instruction cache configuration flags	Instruction cache configuration behavior
`cache_config_enable`	Enable caching for the specified range
`cache_config_disable`	Disable caching for the specified range

Table 43: Instruction cache configuration flags

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
cache_config_instruction(
  (void*) 0x80000000, (void*)0x7fffffff, cache_config_enable);
cache_enable_instruction();
cache_lock();
```

See also

cache_config_data, *cache_enable_instruction*

cache_disable_data

Disable the data cache

Synopsis

```
#include <cache.h>
int cache_disable_data( void );
```

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 not enabled.

Description

This function disables the data cache by flushing it before writing to the ENABLEDCACHE register.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

cache_enable_data

cache_disable_instruction

Disable the instruction cache

Synopsis

```
#include <cache.h>
int cache_disable_instruction( void );
```

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 not enabled.

Description

This function disables the instruction cache by writing to the ENABLEICACHE register.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

cache_enable_data

cache_enable_data

Enable the data cache

Synopsis

```
#include <cache.h>
int cache_enable_data( void );
```

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 ENABLEDCACHE register.

The data cache should be configured before it is enabled, by making calls to **cache_config_data**.

Most ST20 caches must be invalidated prior to being enabled; on such processors, **cache_enable_data()** will automatically invalidate the cache before enabling it, to guard against data loss. As such, it is not necessary to call **cache_invalidate_data()** in order to enable the cache.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

cache_config_data, *cache_enable_data*, *cache_enable_instruction*, *cache_invalidate_data*

cache_enable_instruction

Enable the instruction cache

Synopsis

```
#include <cache.h>
int cache_enable_instruction( void );
```

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.

If the target device has a configurable instruction cache then this should be configured before enabling the instruction cache by making calls to **cache_config_instruction**.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

cache_config_instruction, *cache_enable_data*

cache_flush_data

Flush the data cache

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 also causes the entire cache to be marked invalid. All data is reloaded from main memory.

Note: Any accesses to cacheable memory are blocked until the flush is complete.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
cache_flush_data(NULL, NULL);
```

See also

cache_invalidate_data

cache_init_controller

Initialize the cache controller

Synopsis

```
#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 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 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 is 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 in *Table 44*. Cache map details can also be found in the file **%ST20ROOT%\stdcfg\chip.cfg**, by referring to the **_ST_AddDevice** lines.

ST20 variant	Cache map
ST20TP3	**cache_map_st20tp3**
ST20DC1, ST20DC2	**cache_map_st20dc1**
STi5100	**cache_map_c2_c200**
STi5105	**cache_map_c1_c100**
STi5500, STi5505.	**cache_map_sti5500**
STi5508, STi5510, STi5580.	**cache_map_sti5510**
STi5512, STi5518, STI5519, STi5588, STi5589, STi5598 (cache region 1 in 64kB blocks).	**cache_map_sti5512a**
STi5512, STi5518, STi5519, STi5588, STi5589, STi5598 (cache region 1 in 512kB blocks).	**cache_map_sti5512b**
STm5700, STV0684.	**cache_map_c1_c100**
STi5514, STi5516, STi5517, STi5528, STV396, STV3500.	**cache_map_c2_c200**

Table 44: 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** is undone.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

cache_config_data, *cache_enable_data*, *cache_enable_instruction*, *cache_lock*

cache_invalidate_data

Invalidate the data cache

Synopsis

```
#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 INVALIDATEDCACHE register bit. The entire data cache is marked invalid. If not used correctly this causes data loss. In particular the return address stored when this function is called is destroyed if the workspace occupies cacheable memory.

Note: Any accesses to cacheable memory are blocked until the flushing and invalidation have completed.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
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* ***breserved2***	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 instruction 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: Any accesses to cacheable memory are blocked until the invalidation is complete.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
cache_invalidate_instruction(NULL, NULL);
```

See also

cache_invalidate_data

cache_lock

Lock the cache configuration

Synopsis

```
int cache_lock( void );
```

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.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

cache_flush_data, *cache_invalidate_data*, *cache_invalidate_instruction*

cache_status

Report the cache status

Synopsis

```
#include <cache.h>
cache_status_t cache_status( void );
```

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.

Field name	Meaning
EnableDCache	**1** if the data cache is enabled, **0** otherwise
EnableICache	**1** if the instruction cache is enabled, **0** otherwise
InvalidatingDCache	**1** if the cache controller is invalidating the data cache, **0** otherwise
InvalidatingICache	**1** if the cache controller is invalidating the instruction cache, **0** otherwise
FlushingDCache	**1** if the cache controller is flushing the data cache, **0** otherwise
DCacheReady	**1** if the data cache is ready to perform an operation, **0** otherwise
ICacheReady	**1** if the instruction cache is ready to perform an operation, **0** otherwise
CacheControlLock	**1** if the cache configuration is locked, **0** otherwise

Table 45: Cache status structure

Note: ST20 variants that use **cache_map_sti5500** *do not have a* CACHESTATUS *register so OS20 implements it in software. In software it is not possible to implement all of the features of the* CACHESTATUS *register. Therefore only* ENABLEDCACHE, ENABLEICACHE *and* CACHECONTROLLOCK *should be used on these processors.*

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
cache_status_t status = cache_status();
if ( status.CacheControlLock ) {
  /* cache is locked */
  ...
}
```

callback_...

Register a callback for an event

Synopsis

```
#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

Pointer to the previously installed function.

Errors

None

Description

This group of functions is used to install callback handlers to any of the supported internal OS20 events. These functions can only be used if OS20 is rebuilt with **`CONF_CALLBACK_SUPPORT`** defined; see *Section 15.3.1: Callback Support on page 105*.

`task_context` can be used to determine what task/interrupt level the callback has been called for.

Callable from

Tasks only

Example

```
#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

chan_alt

Wait for input on one of a number of channels

Synopsis

```
#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

This function is ST20-C2 specific.

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 ready channel in the list is returned.

timeout is a pointer to the time-out value. If this time is reached then the function returns 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.

Callable from

A task or a high priority process (on an ST20-C2).

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

chan_create

Create a soft channel

Synopsis

```
#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 is ST20-C2 specific.

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: **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.

Callable from

Tasks only

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

voiiid *address 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 is ST20-C2 specific.

This function creates a channel which uses the hardware channel specified by address ***address*** to communicate with a peripheral device. The **chan_t** structure is allocated from the system partition.

Callable from

Tasks only

See also

chan_create, *chan_delete*, *chan_init*, *chan_init_address*

chan_delete

Delete a channel

Synopsis

```
#include <chan.h>
void chan_delete( chan_t *chan );
```

Arguments

chan_t *chan Channel to delete

Results

None

Errors

None

Description

This function is ST20-C2 specific.

This function allows a channel to be deleted. If the channel was created using **chan_create** or **chan_create_address** this function frees 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: *If any tasks are waiting on the channel when it is deleted, this causes 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.

Callable from

Tasks only

See also

chan_init, *chan_init_address*, *chan_create*, *chan_create_address*

chan_in

Receive data from a channel

Synopsis

```
#include <chan.h>

int chan_in( chan_t *chan,
             void* cp,
             int count );
```

Arguments

chan_t *chan	Pointer to the input channel
void* cp	Pointer to the data
int count	Number of bytes of data

Results

Always returns **0**.

Errors

None

Description

This function is ST20-C2 specific.

Receives ***count*** bytes of data on the specified channel and stores them in the array pointed to by ***cp***.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_init, *chan_out*

chan_in_char

Receive character from a channel

Synopsis

```
#include <chan.h>
char chan_in_char( chan_t *chan );
```

Arguments

chan_t *chan Pointer to the input channel

Results

Returns the input character.

Errors

None

Description

This function is ST20-C2 specific.

Receives a single character on the specified channel and returns it.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_in, *chan_init*, *chan_out_char*

chan_in_int

Receive integer from a channel

Synopsis

```
#include <chan.h>
int chan_in_int( chan_t *chan );
```

Arguments

chan_t *chan Pointer to the input channel

Results

Returns the input integer.

Errors

None

Description

This function is ST20-C2 specific.

Receives a single integer on the specified channel and returns it.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_in_char, *chan_init*, *chan_out_int*

chan_init

Initialize a soft channel

Synopsis

```
#include <chan.h>
void chan_init( chan_t *chan );
```

Arguments

chan_t *chan Pointer to the channel

Results

Returns no results.

Errors

None

Description

This function is ST20-C2 specific.

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: **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.

Callable from

Tasks only

See also

chan_create, *chan_create_address*, *chan_delete*, *chan_init_address*

chan_init_address

Initialize a hard channel

Synopsis

```
#include <chan.h>

void chan_init_address( chan_t *chan,
                        void *address );
```

Arguments

chan_t *chan	Pointer to the channel
void *address	Address of the hard channel

Results

Returns no results.

Errors

None

Description

This function is ST20-C2 specific.

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: **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.

Callable from

Tasks only

See also

chan_create, *chan_create_address*, *chan_delete*, *chan_init*

chan_out

Write data to a channel

Synopsis

```
#include <chan.h>
void chan_out( chan_t *chan,
               const void* cp, int count );
```

Arguments

chan_t *chan	Pointer to the output channel
const void* cp	Pointer to the data
int count	The number of bytes of data

Results

Returns no results.

Errors

None

Description

This function is ST20-C2 specific.

Writes ***count*** bytes of data on the specified channel from the array pointed to by ***cp***.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_in, *chan_init*

chan_out_char

Write character to a channel

Synopsis

```
#include <chan.h>

void chan_out_char( chan_t *chan,
                    char data );
```

Arguments

chan_t *chan	Pointer to the input channel
char data	The character to be output

Results

None

Errors

None

Description

This function is ST20-C2 specific.

Writes a single character on the specified channel.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_in_char, *chan_init*, *chan_out_int*

chan_out_int

Write integer to a channel

Synopsis

```
#include <chan.h>

void chan_out_int( chan_t *chan,
                   int data );
```

Arguments

chan_t *chan	Pointer to the input channel
int data	The integer to be output

Results

None

Errors

None

Description

This function is ST20-C2 specific.

Writes a single integer on the specified channel.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_in_int, *chan_init*, *chan_out_char*

chan_reset

Reset a channel

Synopsis

```
#include <chan.h>
void* chan_reset( chan_t *chan );
```

Arguments

chan_t *chan 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

This function is ST20-C2 specific.

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 is reset. **chan_reset** returns the contents of the channel word prior to the operation.

Callable from

A task or a high priority process (on an ST20-C2).

See also

chan_create, *chan_create_address*, *chan_init*, *chan_init_address*

device_id

Return the current device ID

Synopsis

```
#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.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

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 **STi5516-A**.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Example

```
printf("Device name %s\n", device_name(device_id()));
```

See also

device_id

interrupt_clear

Clear a pending interrupt level

Synopsis

```
#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

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_clear_number, *interrupt_raise*, *interrupt_pending*

interrupt_clear_number

Clear a pending interrupt number

Synopsis

```
#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 is cleared as well.

Note: 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 has no effect on interrupts which have been triggered by a peripheral.*

Applies to

ILC-1, ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_clear, *interrupt_raise_number*, *interrupt_pending*

interrupt_delete

Delete an interrupt level

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 then allows the interrupt level's stack to be freed, as no more interrupts will be generated at this level.

Before calling this function the interrupt 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.

Callable from

Tasks only

See also

interrupt_init, *interrupt_uninstall*

interrupt_disable

Disable an interrupt level

Synopsis

```
#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: 1 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.*

2 This function is provided as part of the IntC library, and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library 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.*

3 Any code running on an ILC-2 or ILC-3 which uses this function to dismiss a level-sensitive interrupt will become locked at interrupt. The function **interrupt_disable_number()** *should be used instead.*

Applies to

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_disable_global, *interrupt_disable_mask*, *interrupt_enable*

interrupt_disable_global

Globally disable interrupts

Synopsis

```
#include <interrup.h>
void interrupt_disable_global( void );
```

Arguments

None

Results

None

Errors

None

Description

This function clears the global enables bit thus disabling all interrupts. This prevents the interrupt controller from attempting to raise any interrupts.

Note: 1 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 are still available. On an ST20-C1 core the timer interrupt is disabled so timer waits are not handled until interrupts are re-enabled.*

2 Any code running on an ILC-2 or ILC-3 which uses this function to dismiss a level-sensitive interrupt will become locked at interrupt. The function **`interrupt_disable_number()`** *should be used instead.*

Applies to

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_disable_mask, *interrupt_disable_number*, *interrupt_enable_global*, *interrupt_lock*, *interrupt_unlock*

interrupt_disable_mask

Disable one or more interrupt levels

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 interrupt levels.

Note: 1 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_disable_global** *instead.*

2 This function is provided as part of the IntC library, and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library 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. See Chapter 10: Interrupts on page 61.*

3 Any code running on an ILC-2 or ILC-3 which uses this function to dismiss a level-sensitive interrupt will become locked at interrupt. The function **interrupt_disable_number()** *should be used instead.*

Applies to

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_disable, *interrupt_disable_global*, *interrupt_disable_number*, *interrupt_disable_global*, *interrupt_enable_mask*

interrupt_disable_number

Disable an interrupt number

Synopsis

```
#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

ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_enable_number

interrupt_enable

Enable an interrupt level

Synopsis

```
#include <interrup.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: 1 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.*

2 This function is provided as part of the IntC library, and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library 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. See* *Chapter 10: Interrupts on page 61**.*

Applies to

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_disable, *interrupt_enable_mask*, *interrupt_enable_number*

interrupt_enable_global

Globally enable interrupts

Synopsis

```
#include <interrup.h>
void interrupt_enable_global( void );
```

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

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_disable_global, *interrupt_lock*, *interrupt_unlock*

interrupt_enable_mask

Enable one or more interrupt levels

Synopsis

```
#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 interrupt levels.

Note: 1 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.*

2 This function is provided as part of the IntC library, and so is always available whichever ILC is used. However, when ILC-2 and ILC-3 are being used, the ILC library 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. See Chapter 10: Interrupts on page 61.*

Example

```
int main()
{
  int Mask;

  /* Enable global interrupts and interrupt 1 */
  Mask = (1 << INTERRUPT_GLOBAL_ENABLE) | (1<<1);
  interrupt_enable_mask(Mask);
  ...
}
```

Applies to

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_disable_mask, *interrupt_enable*

interrupt_enable_number

Enable an interrupt number

Synopsis

```
#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

ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_enable

interrupt_init

Initialize an interrupt level

Synopsis

```
#include <interrup.h>

int interrupt_init( int interrupt_level,
                    void* stack_base,
                    size_t stack_size,
                    interrupt_trigger_mode_t trigger_mode,
                    interrupt_flags_t flags );
```

Arguments

int *interrupt_level*	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 46*
interrupt_flags_t *flags*	Various flags which affect interrupt behavior; see *Table 47*

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 ever uses the stack at a time. ***trigger_mode*** is one of the supported trigger modes, selected from the list shown in *Table 46*.

Interrupt trigger mode name	Interrupt behavior
`interrupt_trigger_mode_high_level`	Trigger while the input is high
`interrupt_trigger_mode_low_level`	Trigger while the input is low
`interrupt_trigger_mode_rising`	Trigger on the rising edge of the input
`interrupt_trigger_mode_falling`	Trigger on the falling edge of the input
`interrupt_trigger_mode_any`	Trigger on rising and falling edges

Table 46: Interrupt trigger modes

flags is used to give additional information about the interrupt. Normally flags should be specified as 0, which results in the default behavior, however, other options can be specified which change the behavior of the interrupt. Possible values for ***flags*** are shown in *Table 47*.

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 only interrupt low priority processes (and can themselves be interrupted by high priority processes) while high priority interrupts interrupt both high and low priority processes.

Interrupt flags	Interrupt behavior	Target
`0`	Trigger at high scheduling priority (Default)	Any
`interrupt_flags_low_priority`	Trigger at low scheduling priority	ST20-C2
`interrupt_flags_high_priority`	Trigger at high scheduling priority	ST20-C2

Table 47: Interrupt flags

Applies to

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

Tasks only

See also

interrupt_install

interrupt_init_controller

Initialize the interrupt controller

Synopsis

```
#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 to the interrupt level controller
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 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 (that is, the number of inputs) supported by the interrupt controller. Similarly, ***level_controller*** and ***interrupt_numbers*** specify the base address and number of interrupt numbers (that is, inputs) supported by the interrupt level controller.

input_offset gives the offset in words into the interrupt level controller of the INPUTINTERRUPTS register.

Note: On ILC-2 and ILC-3 devices ***input_offset*** *is not required, so in this case use a value of zero for this argument.*

Applies to

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

Tasks only

Example

```
/* Set up an STi5516*/
interrupt_init_controller((void*)0x20000000, 16, (void*)0x20111000, 53, 0);
```

See also

interrupt_init

interrupt_install

Install an interrupt handler

Synopsis

```
#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 to 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 OS20's internal data structures.

However if the interrupt number is specified as **-1**, then no attempt is made to program the interrupt level controller, and the interrupt function is 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 -1 minus the number of the external pin. In this case the ST20 does not handle the interrupt, so the handler and parameter must be specified as **NULL**.

Applies to

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

Tasks only

Example

```
/* 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, -2, NULL, NULL);
```

See also

interrupt_init

interrupt_install_sl

Install an interrupt handler specifying a static link

Synopsis

```
#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 is 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 OS20's internal data structures.

However, if the interrupt number is specified as **-1**, then no attempt is made to program the interrupt level controller, and the interrupt function is 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 is normally obtained as a result of loading an RCU.

Applies to

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

Tasks only

See also

interrupt_init, *interrupt_install*

interrupt_lock

Lock all interrupts

Synopsis

```
#include <interrup.h>
void interrupt_lock( void );
```

Arguments

None

Results

None

Errors

None

Description

This function disables all interrupts to the CPU. This prevents any interrupts from the interrupt controller having any effect on the currently executing task. In addition, on the ST20-C2 this also disables 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. When interrupts are locked calling any function that may not be called by an interrupt service routine is illegal.

Applies to

ILC-None, ILC-1, ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_unlock, *task_lock*

interrupt_pending

Return pending interrupt levels

Synopsis

```
#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

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_clear, *interrupt_pending_number*, *interrupt_raise*

interrupt_pending_number

Return pending interrupt numbers

Synopsis

```
#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

ILC-None, ILC-1, ILC-2.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Note: Edge-triggered interrupt handlers should not call this function to query their own interrupt since they will find the PENDING *bit already reset before running the applications handler. Instead, they should use the interrupt handlers argument to differentiate between different interrupt numbers; see* *interrupt_install on page 167*.

See also

interrupt_pending, *interrupt_test_number*

interrupt_raise

Raise an interrupt level

Synopsis

```
#include <interrup.h>
int interrupt_raise( 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

Raise the specified interrupt level. This involves writing to the interrupt controller's SET-PENDING register.

*Note: 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

ILC-None, ILC-1.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_clear, *interrupt_enable*, *interrupt_install*, *interrupt_raise_number*

interrupt_raise_number

Raise an interrupt number

Synopsis

```
#include <interrup.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

ILC-1, ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_clear, *interrupt_enable*, *interrupt_install*

interrupt_status

Report the status of an interrupt level

Synopsis

```
#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.

Structure member	Meaning
interrupt_numbers	Number of interrupt handlers attached to this level
interrupt_stack_base	Pointer to the base of the stack space for this level
interrupt_stack_size	Size of the stack for this level, in bytes
interrupt_stack_used	Peak stack usage in bytes
interrupt_time	Ti level[a]
interrupt_count	Number of times an interrupt at this level has been serviced[a]

Table 48: The interrupt_status_t structure

a. **interrupt_time** and **interrupt_count** should not be used on the standard OS20 deployment kernel, which does not record this data as it decreases interrupt performance. Refer to *Section 15.3.4* for details of kernels which support these fields.

The ***Flags*** parameter is used to indicate which values should be returned. Values which can be determined immediately (all except **interrupt_stack_used**) are always 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.

Callable from

Tasks only

See also

interrupt_status_number

interrupt_status_number

Report the status of an interrupt number

Synopsis

```
#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.

Structure Member	Meaning
intnum_status_level	The level that this interrupt number is attached to
intnum_time	Time spent servicing this interrupt number[a]
intnum_count	Number of times this interrupt number has been serviced[a]

Table 49: The interrupt_status_number_t structure

a. **interrupt_time** and **interrupt_count** should not be used on the standard OS20 deployment kernel, which does not record this data as this decreases interrupt performance. Refer to *Section 15.3.4: Time logging (ST20-C2 core only) on page 106* for details of kernels which support these fields.

Applies to

ILC-1, ILC-2, ILC-3.

Callable from

Tasks only

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: If **Number** is not valid and returns the error code the function evaluates 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

ILC-1, ILC-2, ILC-3

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

Note: Edge-triggered interrupt handlers should not call this function to query their own interrupt number since they will find the PENDING *bit already reset before running the applications handler. Instead, they should use the interrupt handlers argument to differentiate between different interrupt numbers; see* *interrupt_install on page 167*.

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 the *ST20 Embedded Toolset Reference Manual*, chapter *Using the ST20 simulator tool*, table *Trace value identifiers for ST20-C1*

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** sets 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

ILC-2, ILC-3

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_init

interrupt_uninstall

Uninstall an interrupt handler

Synopsis

```
#include <interrup.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 fails.

Description

This function allows an interrupt handler to be uninstalled. This then allows 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

Callable from

Tasks only

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 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 is not released until an equal number of calls to **`interrupt_unlock()`** have been made.

Applies to

ILC-None, ILC-1, ILC-2, ILC-3

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

interrupt_lock, *task_lock*

interrupt_wakeup_number

Set wakeup status of an interrupt number

Synopsis

```
#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: The set of possible trigger modes differs slightly from those used by **interrupt_init** *and* **interrupt_trigger_mode_number**.

Interrupt trigger mode name	Wakeup behavior
interrupt_trigger_mode_no_trigger	Do not wakeup
interrupt_trigger_mode_high_level	Wakeup while the input is high
interrupt_trigger_mode_low_level	Wakeup while the input is low

Table 50: Wakeup trigger modes

Applies to

ILC-2, ILC-3.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

kernel_idle

Return the kernel idle time

Synopsis

```
#include <kernel.h>
clock_t kernel_idle( void );
```

Arguments

None

Results

Returns a clock value indicating the kernel idle time, or **0** if the kernel has not been built with time-logging enabled.

Errors

None

Description

kernel_idle() returns a clock value indicating the time the kernel has been idle, that is, the time not executing code. Idle time occurs when there is no valid task or interrupt and the task queues are empty.

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

Callable from

kernel_idle() can only be called from a task.

See also

kernel_time

kernel_initialize

Initialize for preemptive scheduling

Synopsis

```
#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.

kernel_initialize() installs a default mutual exclusion for the C run-time system. This may be disabled if it is not required; see *Section 15.3.6*.

Note: 1 On the ST20-C2, if this function is called at high priority, it returns executing at low priority. This is a requirement for the correct functioning of the OS20 kernel.

2 This function may be called automatically if **st20cc -runtime os20** *is specified when linking; see Chapter 2: Getting started on page 9. It is important that this function is not called twice.*

Callable from

Not applicable (must be called before scheduler starts).

See also

kernel_start

kernel_start

Start preemptive scheduling regime

Synopsis

```
#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 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: Getting started on page 9**. It is important that this function is not called twice.*

Callable from

Not applicable (must be called before scheduler starts).

See also

kernel_initialize, *task_create*

kernel_time

Return the kernel up-time

Synopsis

```
#include <kernel.h>
clock_t kernel_time( void );
```

Arguments

None

Results

Returns a clock value indicating how long has elapsed since the kernel started executing, or **0** if the kernel has not been built with time-logging enabled.

Errors

None

Description

kernel_time() returns the kernel up-time, a clock value indicating the elapsed time the kernel has been running; that is the total time spent executing code or in idle state.

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

Callable from

kernel_time() can only be called from a task.

See also

kernel_idle

kernel_version

Return the OS20 version number

Synopsis

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

Arguments

None

Results

Returns a pointer to the OS20 version string.

Errors

None

Description

kernel_version() returns a pointer to a string which gives the 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.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

kernel_initialize

memory_allocate

Allocate a block of memory from a partition

Synopsis

```
#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 fails and returns **NULL**.

Description

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

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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

memory_deallocate, *memory_reallocate*,
partition_create_fixed, *partition_create_heap*, *partition_create_simple*

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 fails and returns **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 returns 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.

Callable from

A task or a high priority process (on an ST20-C2).

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.

Callable from

A task or a high priority process (on an ST20-C2).

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

```
#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 fails and returns **NULL**.

Description

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

Note: This function may only be used for heap partitions.

This function tries to 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 is 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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

memory_allocate, *memory_deallocate*, *partition_create_fixed*, *partition_create_heap*, *partition_create_simple*

message_claim

Claim a message buffer

Synopsis

```
#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 blocks until one becomes available (by another task calling **message_release()**).

Callable from

For non-timeout queues, this function is callable from a task or a high priority process (on an ST20-C2).

For timeout queues, this function is callable from tasks only.

See also

message_receive, *message_release*, *message_send*

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

Note: ***time*** *is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the 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** causes the message queue to be polled, that is, the function always returns immediately. If a message is available then it is returned, otherwise the function returns immediately with a result of **NULL**. A timeout of **TIMEOUT_INFINITY** behaves exactly as **message_claim**.

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

Callable from

For non-timeout queues, this function is callable from a task or a high priority process (on an ST20-C2). Timeout value is ignored. Behaves as though **TIMEOUT_INFINITY** was specified. The debug kernel triggers an assertion if the timeout value is ignored, see *Section 3.2.1: Assertion checking on page 18*.

For timeout queues, this function is callable from a task, interrupt service routine or high priority process (on an ST20-C2). For interrupt service routines and high priority processes this function can only be used with a time value of **TIMEOUT_IMMEDIATE**.

Example

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

See also

message_receive_timeout, *message_release*, *message_send*

message_create_queue

Create a fixed size message queue

Synopsis

```
#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.

Callable from

Tasks only

See also

memory_allocate, *message_claim*, *message_delete_queue*, *message_receive*, *message_release*, *message_send*

message_create_queue_timeout

Create a fixed size message queue with timeout capability

Synopsis

```
#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.

Callable from

Tasks only

See also

memory_allocate, *message_claim_timeout*, *message_delete_queue*, *message_receive_timeout*, *message_release*, *message_send*

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 also frees 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: *If any tasks are waiting on the message queue when it is deleted, this causes 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.*

Callable from

Tasks only

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 *Chapter 8: Message handling on page 51*.

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.

Callable from

Tasks only

See also

message_claim, *message_create_queue*, *message_delete_queue*, *message_receive*, *message_release*, *message_send*

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 *Chapter 8: Message handling on page 51*.

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.

Callable from

Tasks only

See also

message_claim_timeout, *message_create_queue*, *message_delete_queue*, *message_receive_timeout*, *message_release*, *message_send*

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 blocks until one becomes available (by another task calling **message_send()**).

Callable from

For non-timeout queues, this function is callable from a task or a high priority process (on an ST20-C2).

For timeout queues, this function is callable from tasks only

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

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

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

time is specified in ticks, which is an implementation-dependent quantity.

Two special time values may also be specified as an alternative to ***time***. **TIMEOUT_IMMEDIATE** causes the message queue to be polled, that is, the function always returns immediately. If a message was available then it is returned, otherwise the function returns immediately with a result of **NULL**. A timeout of **TIMEOUT_INFINITY** behaves exactly as **message_receive**.

Note: These special values must be used directly in a **message_receive_timeout()** *command; setting* **time = TIMEOUT_INFINITY** *for example, then using* **time** *in the command will cause undefined results.*

Callable from

For non-timeout queues, this function is callable from a task or a high priority process (on an ST20-C2). Timeout value is ignored. Behaves as though **TIMEOUT_INFINITY** was specified. The debug kernel triggers an assertion if the timeout value is ignored, see *Section 3.2.1: Assertion checking on page 18*.

For timeout queues, this function is callable from an interrupt service routine or a high priority process (on an ST20-C2). Can only be used with **TIMEOUT_IMMEDIATE**.

Example

```
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*

message_release

Release a message buffer

Synopsis

```
#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 causes the task to be restarted and the message buffer returned.

Callable from

For non-timeout queues, this function is callable from a task or a high priority process (on an ST20-C2).

For timeout queues, this function is callable from a task, interrupt service routine or a high priority process (on an ST20-C2).

See also

message_claim, *message_receive*, *message_send*

message_send

Send a message to a queue

Synopsis

```
#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 adds the message to the end of the queue of sent messages; if any tasks are waiting for a message they are rescheduled and the message returned.

Callable from

For non-timeout queues, this function is callable from a task or a high priority process (on an ST20-C2).

For timeout queues, this function is callable from a task, interrupt service routine or a high priority process (on an ST20-C2).

See also

message_claim, *message_receive*, *message_release*

move2d_all

Two dimensional block move

Synopsis

```
#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

This function is ST20-C2 specific.

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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

move2d_non_zero, *move2d_zero*

move2d_non_zero

Two dimensional block move of all 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

This function is ST20-C2 specific.

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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

move2d_all, *move2d_zero*

move2d_zero

Two dimensional block move of all zero bytes

Synopsis

```
#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

This function is ST20-C2 specific.

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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

move2d_all, *move2d_non_zero*

mutex_create_fifo

Create a FIFO queued mutex

Synopsis

```
#include <mutex.h>
mutex_t* mutex_create_fifo(void);
```

Arguments

None

Results

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

Errors

NULL if there is insufficient memory for the mutex.

Description

mutex_create_fifo() creates a mutex. The memory for the mutex structure is allocated from the system heap. Mutexes created with this function have the usual mutex semantics, except that when a task calls **mutex_lock()** it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Callable from

Tasks only

See also

mutex_init_fifo, *mutex_create_priority*

mutex_create_priority

Create a priority queued mutex

Synopsis

```
#include <mutex.h>
mutex_t* mutex_create_priority(void);
```

Arguments

None

Results

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

Errors

NULL if there is insufficient memory for the mutex.

Description

mutex_create_priority() creates a mutex. The memory for the mutex structure is allocated from the system heap. Mutexes created with this function have the usual mutex semantics, except that when a task calls **mutex_lock()** it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by **mutex_release()**, it is guaranteed to be the task with the highest priority of all those waiting for the mutex.

Mutexes created with this function also guarantee to detect and correct priority inversion.

Callable from

Tasks only

See also

mutex_create_fifo, *mutex_init_priority*

mutex_delete

Delete a mutex

Synopsis

```
#include <mutex.h>

int mutex_delete(
  mutex_t *mutex);
```

Arguments

mutex_t* mutex Mutex to delete

Results

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

Errors

Fails if the mutex is locked, or has tasks blocked waiting for it.

Description

mutex_delete() deletes the mutex, **mutex**.

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

Callable from

Tasks only

See also

mutex_create_priority, *mutex_create_fifo*, *mutex_init_priority*, *mutex_init_fifo*

mutex_init_fifo

Initialize a FIFO queued mutex

Synopsis

```
#include <mutex.h>
void mutex_init_fifo(mutex_t *mutex);
```

Arguments

mutex_t *mutex	The mutex to be initialized.

Results

None

Errors

None

Description

mutex_init_fifo initializes a mutex. Mutexes initialized with this function have the usual mutex semantics, except that when a task calls **mutex_lock()** it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Callable from

Tasks only

See also

mutex_create_fifo, *mutex_init_priority*

mutex_init_priority

Initialize a priority queued mutex

Synopsis

```
#include <mutex.h>
void mutex_init_priority(mutex_t *mutex);
```

Arguments

mutex_t *mutex	The mutex to be initialized.

Results

None

Errors

None

Description

mutex_init_priority initializes a mutex. Mutexes initialized with this function have the usual mutex semantics, except that when a task calls **mutex_lock()** it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by **mutex_release()**, it is guaranteed to be the task with the highest priority of all those waiting for the mutex.

Callable from

Tasks only

See also

mutex_create_priority, *mutex_init_fifo*

mutex_lock

Acquire a mutex, block if not available

Synopsis

```
#include <mutex.h>
void mutex_lock(mutex_t* mutex);
```

Arguments

mutex_t* mutex A pointer to a mutex

Results

None

Errors

None

Description

mutex_lock acquires the given mutex. The exact behavior of this function depends on the mutex type. If the mutex is currently not owned, or is already owned by the task, then the task acquires the mutex, and carries on running. If the mutex is owned by another task, then the calling task is added to the queue of tasks waiting for the mutex, and deschedules. Once the task acquires the mutex it is made immortal, until it releases the mutex.

Note: Management of priority mutexes requires OS20 to allocate a control block when a thread first calls **mutex_lock***. If this allocation fails it causes a fatal error. This does not apply to FIFO mutexes because they do not require a control block.*

Callable from

Tasks only

See also

mutex_release, *mutex_trylock*, *task_immortal*

mutex_release

Release a mutex

Synopsis

```
#include <mutex.h>
int mutex_release(mutex_t* mutex);
```

Arguments

mutex_t* mutex A pointer to a mutex to release

Results

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

Errors

Returns -1 if the task releasing the mutex does not own it.

Description

mutex_release() releases the specified mutex. The exact behavior of this function depends on the mutex type. The operation checks the queue of tasks waiting for the mutex, if the list is not empty, then the first task on the list is restarted and granted ownership of the mutex, possibly preempting the current task. Otherwise the mutex is released, and the task continues running.

If the releasing task had its priority temporarily boosted by the priority inversion logic, then once the mutex is released the task's priority is returned to its correct value.

Once the task has released the mutex, it is made mortal again.

Callable from

Tasks only

See also

mutex_lock, *mutex_trylock*, *task_mortal*

mutex_trylock

Acquire a mutex, return immediately if not available

Synopsis

```
#include <mutex.h>

int mutex_trylock(
  mutex_t* mutex);
```

Arguments

mutex_t* mutex A pointer to a mutex

Results

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

Errors

Call fails if the mutex is currently owned by another task.

Description

mutex_trylock() checks to see if the mutex is free or already owned by the current task, and acquires it if it is. If the mutex is not free, then the call fails and returns **OS21_FAILURE**.

If the task acquires the mutex it is automatically made immortal, until it releases the mutex.

Callable from

Tasks only

See also

mutex_lock, *mutex_release*, *task_immortal*

partition_create_fixed

Create a fixed size partition

Synopsis

```
#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 fails 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 is 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.

Callable from

Tasks only

See also

memory_allocate, *memory_deallocate*, *partition_create_heap*, *partition_create_simple*

partition_create_heap

Create a heap partition

Synopsis

```
#include <partitio.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 fails and returns **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 is allocated, with a small overhead on each block for the partition manager. Allocating and freeing requires searching through lists, and so the length of time depends on the current state of the heap.

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

Callable from

Tasks only

See also

memory_allocate, *memory_deallocate*, *partition_create_fixed*, *partition_create_simple*

partition_create_simple

Create a simple partition

Synopsis

```
#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 fails and returns **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 is 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 always returns **NULL**.

Callable from

Tasks only

See also

memory_allocate, *memory_deallocate*, *partition_create_fixed*, *partition_create_heap*

partition_delete

Delete a partition

Synopsis

```
#include <partitio.h>

void partition_delete( partition_t *Partition )
```

Arguments

partition_t *PartitIion 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, for example **partition_create_heap** then this function frees the data structure used to manage the partition (**partition_t**). If the partition was created using an **_init** function, that is **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.

Callable from

Tasks only

Example

```
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_fixed, *partition_create_heap*, *partition_create_simple*, *partition_init_fixed*, *partition_init_heap*, *partition_init_simple*

partition_init_fixed

Initialize a fixed size partition

Synopsis

```
#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 fails and returns **-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 is 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.

Callable from

Tasks only

See also

memory_allocate, *memory_deallocate*, *partition_init_heap*, *partition_init_simple*

partition_init_heap

Initialize a heap partition

Synopsis

```
#include <partitio.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 fails and returns **-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 is allocated, with a small overhead on each block for the partition manager. Allocating and freeing requires searching through lists, and so the length of time depends on the current state of the heap.

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

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

Callable from

Tasks only

See also

memory_allocate, *memory_deallocate*, *partition_init_fixed*, *partition_init_simple*

partition_init_simple

Initialize a simple partition

Synopsis

```
#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 fails 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 is 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 always returns **NULL**.

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

Callable from

Tasks only

See also

memory_allocate, *memory_deallocate*, *partition_init_fixed*, *partition_init_heap*

partition_status

Get status of a partition

Synopsis

```
#include <partitio.h>

int partition_status( partition_t* Partition,
                      partition_status_t* Status,
                      partition_status_flags_t flags );
```

Arguments

partition_t* *Partition*	Pointer to a partition
partition_status_t* *Status*	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()** 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()** uses to store the results.

Table 51 shows the layout of the structure **partition_status_t**.

Name	Description
partition_status_state	Partition state (See *Table 52*)
partition_status_type	Type of partition (See *Table 53*)
partition_status_size	Total number of bytes within partition
partition_status_free	Total number of bytes free within partition
partition_status_free_largest	Total number of bytes within the largest free block in partition
partition_status_used	Total number of bytes which are allocated/in use within the partition

Table 51: Layout of structure partition_status_t

Table 52 show all the possible values which are available to the field **partition_status_state**.

Flag	Flag description
partition_status_state_valid	Partition is valid
partition_status_state_invalid	Partition is corrupt

Table 52: Flag values for partition_status_state

Table 53 shows all the possible values which are available to the field **partition_status_type**.

Flag	Flag description
partition_status_state_type_simple	Partition is a Simple partition
partition_status_state_type_fixed	Partition is a Fixed partition
partition_status_state_type_heap	Partition is a Heap partition

Table 53: Flag values for partition_status_type

If **partition_status()** returns successfully then the structure pointed to by ***Status*** contains statistics about the partition ***Partition***.

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

partition_status_type depending on the type of partition contains one of the flags as shown in *Table 53*.

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** or **_init** functions, therefore **partition_status_size** does 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/ deallocation, these overheads are taken into account in the results. (See *Chapter 4: Memory and partitions on page 21*).

Callable from

A task or a high priority process (on an ST20-C2).

Example

```
#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_fixed, *partition_create_heap*, *partition_create_simple*, *partition_init_fixed*, *partition_init_heap*, *partition_init_simple*

semaphore_create_fifo

Create a FIFO queued semaphore

Synopsis

```
#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 a task calls **semaphore_wait()** it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Callable from

Tasks only

See also

semaphore_create_priority, *semaphore_init_fifo*

semaphore_create_fifo_timeout

Create a FIFO queued semaphore with timeout capability

Synopsis

```
#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 a task calls **semaphore_wait()** or **semaphore_wait_timeout()**, it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Callable from

Tasks only

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. Semaphores created with this function have the usual semaphore semantics, except that when a task calls **semaphore_wait()** it is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by **semaphore_signal()**, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

Callable from

Tasks only

See also

semaphore_create_fifo, *semaphore_init_priority*

semaphore_create_priority_timeout

Create a priority queued semaphore with timeout capability

Synopsis

```
#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 is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by **semaphore_signal()**, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

Callable from

Tasks only

See also

semaphore_create_fifo_timeout, *semaphore_create_priority*, *semaphore_init_priority_timeout*,

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 also frees 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: *If any tasks are waiting on the semaphore when it is deleted, this causes 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.

Callable from

Tasks only

See also

semaphore_create_fifo, *semaphore_create_priority*, *semaphore_init_fifo*, *semaphore_init_priority*

semaphore_init_fifo

Initialize a FIFO queued semaphore

Synopsis

```
#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 is 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.

Callable from

Tasks only

See also

semaphore_create_fifo, *semaphore_init_priority*

semaphore_init_fifo_timeout

Initialize a FIFO queued semaphore with timeout capability

Synopsis

```
#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 is always 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.

Callable from

Tasks only

See also

semaphore_create_fifo_timeout, *semaphore_init_fifo*, *semaphore_init_priority_timeout*

semaphore_init_priority

Initialize a priority queued semaphore

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 is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by **semaphore_signal()**, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

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

Callable from

Tasks only

See also

semaphore_create_priority

semaphore_init_priority_timeout

Initialize a priority queued semaphore with timeout capability

Synopsis

```
#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 is inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by **semaphore_signal()**, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

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

Callable from

Tasks only

See also

semaphore_create_priority_timeout, *semaphore_init_fifo_timeout*, *semaphore_init_priority*

semaphore_signal

Signal a semaphore

Synopsis

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

Arguments

semaphre_t* *Sem* 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 checks the queue of tasks waiting for the semaphore, if the list is not empty, then the first task on the list is restarted, possibly preempting the current task. Otherwise the semaphore count is incremented, and the task continues running.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

semaphore_wait

semaphore_wait

Wait for a semaphore

Synopsis

```
#include <semaphor.h>
int semaphore_wait( semaphore_t* Sem );
```

Arguments

semaphore_t* ***Sem*** 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 checks the semaphore counter, and if it is **0**, adds the current task to the list of queued tasks before descheduling. Otherwise the semaphore counter is decremented, and the task continues running.

Callable from

For non-timeout FIFO, this function is callable from a task or a high priority process (on an ST20-C2).

For timeout FIFO, timeout priority and non-timeout priority, this function is callable from tasks only.

See also

semaphore_signal, *semaphore_wait_timeout*

semaphore_wait_timeout

Wait for a semaphore or a timeout

Synopsis

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

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

Arguments

semaphore_t* ***Sem***	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** returns 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** returns **0**.

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

The timeout value may be specified in ticks, which is an implementation dependent quantity. Two special time values may also be specified for **timeout**. **TIMEOUT_IMMEDIATE** causes the semaphore to be polled, that is, the function always returns immediately. If the semaphore count is greater than zero, then it has been successfully decremented, and the function returns **0**, otherwise the function returns **-1**. A timeout of **TIMEOUT_INFINITY** will behave exactly as **semaphore_wait**.

Callable from

For non-timeout FIFO, this function is callable from a task or a high priority process (on an ST20-C2). Timeout value is ignored. Behaves as though **TIMEOUT_INFINITY** was specified. The debug kernel triggers an assertion if the timeout value is ignored, see *Section 3.2.1: Assertion checking on page 18*.

For timeout FIFO, this function is callable from a task, interrupt service routine or high priority process (on an ST20-C2). For interrupt service routines and high priority processes this function can only be used with a time value of **TIMEOUT_IMMEDIATE**.

For non-timeout priority, this function is callable from tasks only. Timeout value is ignored. Behaves as though **TIMEOUT_INFINITY** was specified. The debug kernel triggers an assertion if the timeout value is ignored, see *Section 3.2.1: Assertion checking on page 18*.

For timeout priority, this function is callable from a task, interrupt service routine or high priority process (on an ST20-C2). For interrupt service routines and high priority processes this function can only be used with a time value of **TIMEOUT_IMMEDIATE**.

Example

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

See also

semaphore_signal, *semaphore_wait*

task_context

Return the current execution context

Synopsis

```
#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 OS20 task, then it returns **task_context_task**,
- an interrupt handler, then it returns **task_context_interrupt**,
- a high priority process on an ST20-C2, then it returns **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 OS20 task or a high priority process, then the corresponding **task_t** is 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 is 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** executes faster if these values are not required.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

task_id

task_create

Create an OS20 task

Synopsis

```
#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 passed into **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() sets up a function as an OS20 task and starts the task executing. **task_create()** 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.

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. **task_create** automatically calls **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 are also allocated by **task_create** calling **memory_allocate()**. The task descriptor (**tdesc_t**) is allocated from the internal memory partition, the task state (**task_t**) from the system memory partition.

Priority is the task's scheduling priority.

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

flags is used to give additional information about the task. Normally flags should be specified as **0**, which results in the default behavior, however, other options can be specified which change the behavior of the task.

For the ST20-C2 this is used to create tasks which execute using the ST20's hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the 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.

Note: The units of time are different for high priority processes; see Chapter 9: Real-time clocks on page 57.

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 does not run until it is resumed using the **task_resume** call.

Note: High priority processes cannot be created suspended.

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

Task flags	Task behavior	Target
0	Create an OS20 task. Default.	Any
task_flags_high_priority_process	Create the task as a high priority process (this is ignored on ST20-C1 devices).	ST20-C2
task_flags_suspended	Create the task already suspended.	Any

Table 54: Flag values for task_create

Note: This function allocates memory from the internal partition. It is common for the internal partition to use the simple partition manager in order to conserve internal memory, **-runtime os20** *does this. Since the simple partition manager does not support freeing memory this has the potential to cause memory leaks if tasks do not exist for the lifetime of the system. See Section 15.1.5: Altering the internal partition manager on page 104 for details of how to migrate a partition to the heap manager.*

Callable from

Tasks only

Example

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

void signal_task(void* p)
{
   struct sig_params* Params = (struct sig_params*)p;
   int j;

   for (j = 0; j < Params->Count; j++) {
    semaphore_signal (Params->Ready);
    task_delay(ONE_SECOND);
   }
}

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 OS20 task specifying a static link

Synopsis

```
#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 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 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 ***Function*** were called with the single argument ***Param***.

StaticLink is the static link which should be used when calling ***Function***. This is normally obtained as a result of loading an RCU. See the *ST20 Embedded Toolset Reference Manual*, chapter *Building and running relocatable code*.

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. **task_create_sl** automatically calls **memory_allocate()** in order to allocate the stack on the system memory partition.

The task's data structures are also allocated by **task_create_sl** calling **memory_allocate()**. The task descriptor (**tdesc_t**) is allocated from the internal memory partition, the task state (**task_t**) from the system memory partition.

Priority is the task's scheduling priority.

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

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

For the ST20-C2 this is used to create tasks which execute using the ST20's hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the 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.

Note: The units of time are different for high priority processes; see Chapter 9: Real-time clocks on page 57.

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 does not run until it is resumed using the **task_resume** call.

Note: High priority processes cannot be created suspended.

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

Task flags	Task behavior	Target
0	Create an OS20 task. Default.	Any
task_flags_high_priority_process	Create the task as a high priority process (this is ignored on ST20-C1 devices).	ST20-C2
task_flags_suspended	Create the task already suspended.	Any

Table 55: Flag values for task_create_sl

Note: This function allocates memory from the internal partition. It is common for the internal partition to use the simple partition manager in order to conserve internal memory, -runtime os20 does this. Since the simple partition manager does not support freeing memory this has the potential to cause memory leaks if tasks do not exist for the lifetime of the system. See Section 15.1.5: Altering the internal partition manager on page 104 for details of how to migrate a partition to the heap manager.

Callable from

Tasks only

See also

task_create, *task_delete*, *task_init_sl*

task_data

Retrieve a task's data pointer

Synopsis

```
#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: Task data on page 38*.

Callable from

A task or a high priority process (on an ST20-C2).

See also

task_data_set

task_data_set

Set a task's data pointer

Synopsis

```
#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: Task data on page 38*.

Callable from

A task or a high priority process (on an ST20-C2).

See also

task_data

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 9: Real-time clocks on page 57*.

Callable from

A task or a high priority process (on an ST20-C2).

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 *Section 9: Real-time clocks on page 57*.

Callable from

A task or a high priority process (on an ST20-C2).

See also

task_delay

task_delete

Delete an OS20 task

Synopsis

```
#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 instruction fails.

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.

Callable from

Tasks only

See also

task_create, *task_kill*

task_exit

Exit the current task

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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

task_onexit_set

task_id

Find current task's ID

Synopsis

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

Arguments

None

Results

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

Errors

None

Description

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

Callable from

A task or a high priority process (on an ST20-C2).

See also

task_create

task_immortal

Make the current task immortal

Synopsis

```
#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 continues running until it becomes mortal again, and will then die.

Callable from

Tasks only

See also

task_kill, *task_mortal*

task_init

Initialize an OS20 task

Synopsis

```
#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 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 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. On ST20-C2 cores **task_init** cannot be used on a high priority process.

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 are used by 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 is passed to the debugger (if present) so that the task can be correctly identified in the debugger's task list.

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

For the ST20-C2 this is used to create tasks which execute using the ST20's hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the 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**.

Note: The units of time are different for high priority processes; see Section 9: Real-time clocks on page 57.

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 does not run until it is resumed using the **task_resume** call.

Note: High priority processes cannot be created suspended.

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

Task flags	Task behavior	Target
0	Create an OS20 task. Default.	Any
task_flags_high_priority_process	Create the task as a high priority process (this is ignored on ST20-C1 devices).	ST20-C2
task_flags_suspended	Create the task already suspended	Any

Table 56: Flag values for task_init

Callable from

Tasks only

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 OS20 task specifying a static link

Synopsis

```
#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 is 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.

Description

task_init() sets up a function as an 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. On ST20-C2 cores **task_init_sl** cannot be used on a high priority process.

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 is normally obtained as a result of loading an RCU. See the *ST20 Embedded Toolset Reference Manual*, chapter *Building and running relocatable code*.

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 are used by 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 is passed to the debugger (if present) so that the task can be correctly identified in the debugger's task list.

flags is used to give additional information about the task. Normally flags should be specified as **0**, which results in the default behavior, however other options can be specified which change the behavior of the task.

For the ST20-C2 this is used to create tasks which execute using the ST20's hardware high priority processes. Tasks which execute as high priority processes are not scheduled using the 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**.

Note: The units of time are different for high priority processes; see Section 9: Real-time clocks on page 57.

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 does not run until it is resumed using the **task_resume** call.

Note: High priority processes cannot be created suspended.

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

Interrupt flags	Interrupt behavior	Target
0	Create an OS20 task. Default.	Any
task_flags_high_priority_process	Create the task as a high priority process (this is ignored on ST20-C1 devices).	ST20-C2
task_flags_suspended	Create the task already suspended.	Any

Table 57: Flag values for task_init_sl

Callable from

Tasks only

See also

task_create, *task_create_sl*, *task_init*

task_kill

Kill a task

Synopsis

```
#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 fails.

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 is removed from the list of tasks waiting for that object before the exit handler is called.

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

```
task_exit(status)
```

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

Note: When **task_kill** *is used to kill a task with the* **task_kill_flags_no_exit_handler** *flag set, any waiting tasks (caused by* **task_wait***) are not informed that the task has been killed. This is because the exit handler is used to inform waiting tasks, so if the exit handler is not called the waiting tasks are not informed about the event.*

A task can temporarily make itself immune to being killed by calling **task_immortal**; see *Section 5.11: Killing a task on page 36* for more details. When a task which has made itself immortal is killed, **task_kill** returns immediately, but the killed task does not die until it makes itself mortal again.

Note: **task_kill** *may return before the task has died. A task_kill should normally be followed by a* **task_wait** *to be sure that the task has made itself mortal again, and completed its exit handler.*

On ST20-C2 cores **task_kill** *cannot be used on a high priority process.*

Callable from

Tasks only

Example

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

See also

task_delete, *task_immortal*, *task_mortal*

task_lock

Prevent task rescheduling

Synopsis

```
#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 is terminated. Calls to **`task_lock()`** can be nested, and the lock not be released until an equal number of calls to **`task_unlock()`** have been made.

Callable from

Tasks only

See also

interrupt_lock, *task_unlock*

task_mortal

Make the current task mortal

Synopsis

```
#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 dies 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** are required before it becomes mortal again.

Callable from

Tasks only

See also

task_immortal, *task_kill*

task_name

Return the name of the specified task

Synopsis

```
#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.

Callable from

A task or a high priority process (on an ST20-C2).

See also

task_create, *task_init*

task_onexit_set

Set the task onexit handler

Synopsis

```
#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 is called whenever a task exits. The handler is 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:

```
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.

Callable from

Tasks only

See also

task_exit

task_onexit_set_sl

Set the task onexit handler specifying a static link

Synopsis

```
#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 is called whenever a task exits. The handler is 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:

```
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 is normally 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 the *ST20 Embedded Toolset Reference Manual*, chapter *Building and running relocatable code*.

Callable from

Tasks only

See also

task_exit, *task_onexit_set_sl*

task_priority

Retrieve a task's priority

Synopsis

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

Arguments

task_t* *Task* Pointer to the task structure

Results

Returns the 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** returns the value **-1**.

Errors

None

Description

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

Callable from

A task or a high priority process (on an ST20-C2)

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 OS20 priority value for the task

Results

task_priority_set() returns the task's previous 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.

On ST20-C2 cores **task_priority_set** cannot be used on a high priority process.

Callable from

Tasks only

See also

task_priority

task_private_data

Retrieve a task's private data pointer

Synopsis

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

Arguments

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

Results

task_private_data() returns the address of the private data registered for the task pointed to by ***task***, under the unique identifier ***cookie***, or **NULL** if no data has been registered.

Errors

None

Description

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

If this API is used prior to kernel initialization, then the operation is performed on the root task.

Callable from

Tasks only

See also

task_private_data_set

task_private_data_set

Set a task's private data pointer

Synopsis

```
#include <task.h>

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

Arguments

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

Results

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

Errors

If OS20 runs out of memory, or private data for this task already exits under the specified ***cookie*** (and ***data*** is not **NULL**) then **-1** is returned.

Description

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

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

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

If this API is used prior to kernel initialization, then the operation is performed on the root task.

Callable from

Tasks only.

See also

task_private_data

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 has no effect, otherwise it performs a timeslice at the current priority.

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

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

Callable from

A task or a high priority process (on an ST20-C2).

task_resume

Resume a suspended task

Synopsis

```
#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 fails.

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 starts to execute again.

If the task was waiting for an event when it was suspended, then the event must also occur before the task starts executing. When a task is resumed it starts executing the next time it is the highest priority task, and so may preempt the task calling **task_resume**.

On ST20-C2 cores, **task_resume** cannot be used on a high priority process.

Callable from

Tasks only

See also

task_suspend

task_stack_fill

Retrieve task stack fill settings

Synopsis

```
#include <task.h>

int task_stack_fill( task_stack_fill_t* fill );
```

Arguments

task_stack_fill_t* ***fill*** Pointer to 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 58 shows the layout of the structure **task_stack_fill_t**.

Callable from

Tasks only

Example

```
#include <task.h>

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

See also

task_create, *task_init*, *task_stack_fill_set*

task_stack_fill_set

Set task stack fill settings

Synopsis

```
#include <task.h>

int task_stack_fill_set( task_stack_fill_t* fill );
```

Arguments

task_stack_fill_t* ***fill*** Pointer to new settings

Results

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

Errors

Returns **-1** if the new settings are invalid.

Description

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

Any subsequent calls to the functions **task_init()** or **task_create()** 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()** has its stack initialized by overwriting the whole contents of the stack with the value 0x12345678. *Table 58* shows the layout of the **task_stack_fill_t** structure.

Field	Description
task_stack_fill_state	Enable or Disable stack filling (See *Table 59*).
task_stack_fill_pattern	Pattern value used when a stack is initialized.

Table 58: Layout of structure task_stack_fill_t

Table 59 shows all the flag values which can be used in the field t**ask_stack_fill_state**. Any other value not in the table causes **task_stack_fill_set()** to return **-1**.

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

Table 59: Flags used by task_stack_fill_state

Callable from

Tasks only

Example

```
#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

```
#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 fails.

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:

Field name	Description
task_stack_base	Base address of the task's stack.
task_stack_size	Size of the task's stack in bytes.
task_stack_used	Amount of stack used by the task in bytes.
task_time	CPU time used by the task.

Table 60: task_status_t fields

Note: The **task_time** *field is only valid when using the debug version of the kernel library or when OS20 has been built with time logging enabled. See Section 15.3.4: Time logging (ST20-C2 core only) for details.*

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***) are always 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**.

On ST20-C2 cores **task_status** cannot be used on a high priority process.

Callable from

Tasks only

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 fails.

Description

This function suspends the specified task. If ***Task*** is **NULL** then this suspends the current task.

task_suspend stops the task from executing immediately, until it is resumed using **task_resume**.

On ST20-C2 cores **task_suspend** cannot be used on a high priority process.

Caution: **task_suspend()** is an inherently dangerous API when applied to any task except the caller, because it takes no account of the state of the specified task.

If the suspended task holds any semaphores or other locks, these locks will not be released until the task is resumed.

Similarly, if the suspended task is waiting for a lock, the fact that the task is suspended does not prevent the lock being assigned to the task.

Both these situations make deadlock highly likely.

task_suspend() can be used for interactive debug/test tools but should not be used as a general syncronization mechanism.

Callable from

Tasks only

See also

task_resume

task_unlock

Allow task rescheduling

Synopsis

```
#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) continues running.

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

Callable from

Tasks only

See also

interrupt_lock, *task_lock*

task_wait

Wait until one of a list of tasks completes

Synopsis

```
#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 as an absolute time 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 is ignored.

timeout is a pointer to the timeout value. If this time is reached then the function returns **-1**.

The **timeout** value must be specified as an absolute time, which is an implementation dependent quantity; see *Chapter 9: Real-time clocks on page 57*.

Two special values can be specified for **timeout**: **TIMEOUT_IMMEDIATE** indicates that the function should return immediately, even if no tasks have terminated; **TIMEOUT_INFINITY** indicates that the function should ignore the timeout period, and only return when a task terminates.

Callable from

Tasks only

See also

task_create, *task_init*

time_after

Return whether one time is after another

Synopsis

```
#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.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

time_minus, *time_now*

time_minus

Subtract two clock values

Synopsis

```
#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.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

time_plus

time_now

Return the current time

Synopsis

```
#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 is no later than the call to **kernel_start**.

The units of ticks is an implementation dependent quantity; see *Chapter 9: Real-time clocks on page 57*.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

task_delay

time_plus

Add two clock values

Synopsis

```
#include <ostime.h>

clock_t time_plus( 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 adding ***time1*** 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.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

time_minus

time_ticks_per_sec

Obtain the current system clock rate

Synopsis

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

Arguments

None

Returns

The number of system clock ticks per second.

Errors

None

Description

Returns the number of system clock ticks per second.

On ST20-C2, this number will be different depending on the calling context. Specifically the system clock runs 64 times faster for high priority processes and interrupts than for low priority processes and interrupts. By default OS20 tasks and interrupts run at low priority.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

time_ticks_per_sec_set, *timer_init_pwm*

time_ticks_per_sec_set

Specify the number of ticks per second observed on a hardware device

Synopsis

```
#include <ostime.h>

void time_ticks_per_sec_set( clock_t tick_rate );
```

Arguments

clock_t *tick_rate*	Number of ticks per second. On ST20-C2 this is the number of HIGH PRIORITY ticks per second.

Errors

None

Description

This function is used to update the operating systems estimate of the system clock tick rate.

On ST20-C1 it is not always possible to estimate the tick rate, making the use of this function mandatory unless the timer is managed using **timer_init_pwm()**. If **timer_init_pwm()** is used the tick rate is calculated from the PWM's input frequency.

On ST20-C2 the tick rate is an architectural constant thus it is always possible to make an estimate. However, since this architectural constant is subject to a small percentage tolerance in some applications it is desirable to update the tick rate to a more precise value.

Callable from

A task, interrupt service routine or a high priority process (on an ST20-C2).

See also

time_ticks_per_sec, *timer_init_pwm*

timer_init_pwm

Use OS20's in built timer management code for ST20-C1

Synopsis

```
#include <c1timer.h>

int timer_init_pwm( void *base,
                    int number,
                    int level,
                    int freq_in_khz,
                    timer_init_pwm_flags_t flags);
```

Arguments

void *base	PWM base address
int *number*	Interrupt number
int *level*	Interrupt level
int *freq_in_khz*	PWM input frequency (in KHz)
timer_init_pwm_flags_t *flags*	Various flags which affect PWM behavior

Results

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

Errors

Returns **-1** if the PWM interrupt handler cannot be installed.

Description

This function installs an ST20-C1 timer peripheral implemented using the PWM hardware found on many ST20-C1 devices. As such this function is unique to the ST20-C1.

Before calling this function **interrupt_init()** must already have been called for the desired interrupt level. The trigger mode for that interrupt level must be **interrupt_trigger_mode_rising**. Unless there is a specific reason to avoid it, the PWM should use interrupt level 0 (least priority) since there are no real time constraints on interrupt latancy for the OS timer.

After calling this function the interrupt must be enabled using the combination of **interrupt_enable_global()**, **interrupt_enable()** and **interrupt_enable_number()** appropriate to the device.

The PWM input frequency is used to attempt to acheive target rates for both the system clock and the timeslice. It is also used to configure the value returned by **time_ticks_per_second()**. The target rates for the system clock is 15625 ticks per second while the target quiessant timeslice is 2 ms.

Note: Few PWM devices are capable of directly providing a 15625 tick rate, usually the achieved tick rate is much faster (sometimes by as much as two orders of magnitude).

Callable from

Tasks only.

See also

interrupt_init, *time_ticks_per_sec*

timer_initialize

Initialize the timer plug-in library for ST20-C1 cores

Synopsis

```
#include <c1timer.h>
void timer_initialize( timer_api_t* timer_functions );
```

Arguments

timer_api_t* *timer_functions*	Table of function pointers for use by the OS20 time functions

Results

None

Errors

None

Description

The **timer_initialize** function enables OS20 when it is run on an ST20-C1 core to fully implement the OS20 API. The ST20-C1 core does not have a built-in timer and therefore requires the timer code in order to support functions that require timed waits such as **task_delay** or **message_claim_timeout**. See *Chapter 13: ST20-C1 specific features on page 87*.

The plug-in functions are typically implemented using the on-chip PWM peripheral but there are other ways the timer can be implemented.

timer_functions should be populated with function pointers as described below.

```
int (*timer_read)(void);
```

timer_read is used by OS20 to determine the current time.

timer_read should return a 32-bit unsigned integer (although for legacy reasons it is prototyped to return a signed integer)). To ensure that the timer arithmetic works correctly it is important that the timer does not roll-over before it reaches 0xFFFFFFFF. For example a 16-bit counter should be multiplied by 65536 before being returned, in order to meet the roll-over requirements.

```
void (*timer_set)(int time);
```

timer_set is used by OS20 to set the point at which the timer expires.

timer_set should treat its argument as a 32-bit unsigned integer (although for legacy reasons it is prototyped to take a signed integer). When the current time reaches the supplied time, the timer plug-in should call **timer_interrupt**. Typically this function programs whatever hardware device is being used to generate an interrupt at the appropriate time.

```
void (*timer_enable_int)(void);
void (*timer_disable_int)(void);
```

The functions are used by OS20 to enable or disable the timer interrupt.

The **timer_interrupt** function should not be called unless the timer interrupt is enabled.

```
void (*timer_raise_int)(void);
```

timer_raise_int is used by OS20 to forcibly generate a timer interrupt.

This function should cause **timer_interrupt** to be called, typically by generating a software interrupt using **interrupt_raise_number**.

Callable from

Tasks only

Example

Refer to the examples supplied with the toolset in:

$ST20ROOT/examples/os20/c1timer

See also

timer_interrupt

timer_interrupt

Notify OS20 that the timer has expired

Synopsis

```
#include <c1timer.h>
void timer_interrupt( void );
```

Arguments

None

Results

None

Errors

None

Description

timer_interrupt should be used only on ST20-C1 cores and is used to notify OS20 that the timer has expired.

This function should be called by the ST20-C1 timer interrupt service routine. See *Chapter 13: ST20-C1 specific features on page 87*.

Callable from

Interrupt service routines only.

Example

An example is provided in the examples directory supplied with the toolset:

$ST20ROOT/examples/os20/c1timer

See also

timer_initialize

Revision history

Version	Date	Comments
D	Feb 06	Supports the R2.2.1 Product Release of the ST20 Toolset. *Preface on page xiii* updated. *Section 3.2: Optional debug features on page 18* updated. *Section 4.2: Allocation strategies on page 22*, *Table 3* updated. *Section 12.2: Initializing the cache support system on page 82* updated. *Section 15.3.2: Changing the number of task priority levels on page 105* updated. *Chapter 16: Alphabetical list of functions*: *cache_init_controller on page 126* updated. *cache_invalidate_instruction on page 129* updated. *message_claim_timeout on page 196* updated. *message_receive_timeout on page 204* updated. *mutex_lock on page 216* corrected. *semaphore_wait_timeout on page 240* updated. *task_kill on page 263* updated. *time_ticks_per_sec on page 288* corrected. *time_ticks_per_sec_set on page 289* corrected.
C	Nov 04	Supports the R2.1 Product Release of the ST20 Toolset. Tasks chapter: Added descriptions of task_private_data and task_private_data_set. Real-time clocks chapter: Added new section on Detemining the tick rate. ST20-C1 specific features chapter: Added new section on In-built PWM support. Alphabetical list of functions chapter: Added the functions task_private_data, task_private_data_set, time_ticks_per_sec, time_ticks_per_sec_set and timer_init_pwm.

Version	Date	Comments
B	Nov 03	Supports the R2.0 Product Release of the ST20 Toolset. Added Mutex functions and updated the ‘Kernel’ and ‘Interrupts’ chapters.
A	Aug 03	Split from *ST20 Embedded Toolset User Manual*, ADCS 7143840H.

Index

K

M

O

P

R

S

T