Automotive & Discrete Group
Automotive Digital Division

Infotainment Business Unit

OS20+ operating system specification

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 managed by the user and tasks may be given priorities and are scheduled accordingly. Timer functions are provided to implement time and delay functions.

The library does not depend on specific drivers, but can be integrated on any core and any drivers set by writing specific BSPs (**Board Support Package**) using given drivers set.

2 Contents

2.1 Index

1 INTRODUCTION 1

2 CONTENTS 2

2.1 INDEX 2
2.2 LIST OF TABLES 6
2.3 LIST OF FIGURES 6

3 DOCUMENT MANAGEMENT 7

3.1 REVISION HISTORY 7
3.2 ACRONYMS 8
3.3 REFERENCE DOCUMENTS 8
3.4 CONTACT INFO 8

4 PREFACE 9

4.1 CONVENTIONS USED 9
4.2 OVERVIEW 9
4.2.1 Naming 10
4.3 CLASSES AND OBJECTS 11
4.3.1 Object Lifetime 11
4.3.2 How this manual is organized 12
4.4 TASKS 12
4.5 PRIORITY 12
4.6 SEMAPHORES 12
4.7 MUTEXES 13
4.8 MESSAGE QUEUES 13
4.9 CLOCKS 13
4.10 INTERRUPTS 13
4.11 WAKELOCKS 13
4.12 OS20+ DATA TYPES 13
4.12.1 Basic types: gpOS_types.h 13
4.12.2 Enhanced data types 14
4.13 OS20+ GENERAL APIS SUMMARY 14
4.14 OS20+ FUNCTIONS DEFINITIONS 16
4.14.1 gpOS_version 16

5 KERNEL 17

5.1 IMPLEMENTATION 17
5.2 OS20+ KERNEL 17
5.3 SYSTEM CALLS 18
5.4 KERNEL APIS SUMMARY 19
5.5 KERNEL FUNCTIONS DEFINITIONS 20
5.5.1 gpOS_kernel_init 20
5.5.2 gpOS_kernel_start 21
5.5.3 gpOS_kernel_get_active_task 22
5.5.4 gpOS_kernel_set_timeslice 23

5.5.5 gpOS_kernel_get_timeslice ... 24
5.5.6 gpOS_kernel_user_system_call ... 25
5.5.7 gpOS_kernel_lock ... 26
5.5.8 gpOS_kernel_unlock ... 27
5.5.9 gpOS_kernel_get_context_switch_number ... 28
5.5.10 gpOS_kernel_get_interrupts_occurred_number ... 29
5.5.11 gpOS_kernel_get_interrupts_spent_time ... 30

6 MEMORY AND PARTITIONS ... 31

6.1 PARTITIONS ... 31
6.2 OBTAINING INFORMATION ABOUT PARTITIONS ... 32
6.3 CREATING A NEW PARTITION TYPE ... 32
6.4 ALLOCATING MEMORY ... 32
6.5 FREEING MEMORY ... 32
6.6 MEMORY AND PARTITION APIS SUMMARY ... 33
6.7 MEMORY AND PARTITION FUNCTIONS DEFINITIONS ... 35
6.7.1 gpOS_memory_init ... 35
6.7.2 gpOS_memory_allocate ... 36
6.7.3 gpOS_memory_allocate_p ... 37
6.7.4 gpOS_memory_deallocate ... 38
6.7.5 gpOS_memory_deallocate_p ... 39
6.7.6 gpOS_memory_create_partition ... 40
6.7.7 gpOS_memory_getheapsize ... 41
6.7.8 gpOS_memory_getheapsize_p ... 42
6.7.9 gpOS_memory_getheapfree ... 43
6.7.10 gpOS_memory_getheapfree_p ... 44
6.7.11 gpOS_memory_getheaprequested ... 45
6.7.12 gpOS_memory_getheaprequested_p ... 46
6.7.13 gpOS_memory_getnextpartition ... 47

7 TASKS ... 48

7.1 OS20+ TASKS ... 48
7.2 OS20+ PRIORITIES ... 48
7.3 SCHEDULING ... 49
7.4 CREATING A TASK ... 49
7.5 SYNCHRONIZING TASKS ... 50
7.6 COMMUNICATING BETWEEN TASKS ... 50
7.7 TIMED DELAYS ... 50
7.8 SUSPENDING TASKS ... 51
7.9 GETTING THE CURRENT TASK'S ID ... 52
7.10 TASK DATA ... 52
7.11 TASK TERMINATION ... 53
7.12 WAITING FOR TERMINATION ... 55
7.13 TASKS APIS SUMMARY ... 56
7.14 TASKS FUNCTIONS DEFINITIONS ... 59
7.14.1 gpOS_task_create ... 59
7.14.2 gpOS_task_create_p ... 61
7.14.3 gpOS_task_delete ... 62
7.14.4 gpOS_task_delay ... 63
7.14.5 gpOS_task_delay_until ... 64
7.14.6 gpOS_task_suspend ... 65
7.14.7 gpOS_task_resume ... 66

7.14.8 gpOS_task_get_id *67*
7.14.9 gpOS_task_set_priority *68*
7.14.10 gpOS_task_get_priority *69*
7.14.11 gpOS_task_get_stack_base *70*
7.14.12 gpOS_task_get_stack_ptr *71*
7.14.13 gpOS_task_get_stack_size *72*
7.14.14 gpOS_task_get_stack_used *73*
7.14.15 gpOS_task_get_cpuusage *74*
7.14.16 gpOS_task_get_runtime *75*
7.14.17 gpOS_task_get_head *76*
7.14.18 gpOS_task_get_next *77*
7.14.19 gpOS_task_get_name *78*
7.14.20 gpOS_task_set_data *79*
7.14.21 gpOS_task_get_data *80*
7.14.22 gpOS_task_get_timeout *81*
7.14.23 gpOS_task_exit *82*
7.14.24 gpOS_task_set_onexit_function *83*
7.14.25 gpOS_task_wait *84*
7.14.26 gpOS_task_lock *85*
7.14.27 gpOS_task_unlock *86*

8 SEMAPHORES 87

8.1 SEMAPHORES OVERVIEW 87
8.2 USE OF SEMAPHORES 88
8.3 SEMAPHORES APIS SUMMARY 89
8.4 SEMAPHORES FUNCTIONS DEFINITIONS 91
8.4.1 gpOS_semaphore_create *91*
8.4.2 gpOS_semaphore_create_p *92*
8.4.3 gpOS_semaphore_delete *93*
8.4.4 gpOS_semaphore_signal *94*
8.4.5 gpOS_semaphore_wait *95*
8.4.6 gpOS_semaphore_wait_timeout *96*

9 MUTEXES 98

9.1 MUTEXES OVERVIEW 98
9.2 USE OF MUTEXES 99
9.3 MUTEXES APIS SUMMARY 99
9.4 MUTEXES FUNCTIONS DEFINITIONS 101
9.4.1 gpOS_mutex_create *101*
9.4.2 gpOS_mutex_create_p *102*
9.4.3 gpOS_mutex_delete *103*
9.4.4 gpOS_mutex_lock *104*
9.4.5 gpOS_mutex_release *105*
9.4.6 gpOS_mutex_trylock *106*
9.4.7 gpOS_mutex_locked *107*

10 MESSAGE HANDLING 108

10.1 MESSAGE QUEUES 108
10.2 CREATING MESSAGE QUEUES 109
10.3 USING MESSAGE QUEUES 109
10.4 MESSAGE HANDLING APIS SUMMARY 110
10.5 MESSAGES FUNCTIONS DEFINITION 112

10.5.1 gpOS_message_create_queue 112
10.5.2 gpOS_message_create_queue_p 113
10.5.3 gpOS_message_delete_queue 114
10.5.4 gpOS_message_claim 115
10.5.5 gpOS_message_claim_timeout 116
10.5.6 gpOS_message_receive 118
10.5.7 gpOS_message_receive_timeout 119
10.5.8 gpOS_message_release 121
10.5.9 gpOS_message_send 122

11 REAL-TIME CLOCKS 123

11.1 READING THE CURRENT TIME 123
11.2 TIME ARITHMETIC 123
11.3 CHIP VARIANTS 125
11.4 TIME APIS SUMMARY 126
11.5 TIMER FUNCTIONS DEFINITIONS 127
11.5.1 gpOS_time_after 127
11.5.2 gpOS_time_minus 128
11.5.3 gpOS_time_plus 129
11.5.4 gpOS_time_now 130
11.5.5 gpOS_timer_ticks_per_second 131
11.5.6 gpOS_timer_ticks_per_msec 132
11.5.7 gpOS_timer_ticks_per_usec 133
11.5.8 gpOS_timer_set_clock 134

12 INTERRUPTS 135

12.1 CHIP VARIANTS 135
12.2 INITIALIZING THE INTERRUPT HANDLING SUBSYSTEM 135
12.3 ATTACHING AN INTERRUPT HANDLER 136
12.4 ENABLING AND DISABLING INTERRUPTS 136
12.5 DETACHING AN INTERRUPT HANDLER 136
12.6 LOCKING OUT INTERRUPTS 136
12.7 CONTEXTS AND INTERRUPT HANDLER CODE 137
12.8 INTERRUPT HOOKS 137
12.9 INTERRUPT APIS SUMMARY 137
12.10 INTERRUPT FUNCTION DEFINITIONS 139
12.10.1 gpOS_interrupt_init 139
12.10.2 gpOS_interrupt_set_hooks 140
12.10.3 gpOS_interrupt_install 141
12.10.4 gpOS_interrupt_uninstall 142
12.10.5 gpOS_interrupt_enable 143
12.10.6 gpOS_interrupt_disable 144
12.10.7 gpOS_interrupt_lock 145
12.10.8 gpOS_interrupt_unlock 146

13 EXCEPTIONS 147

13.1 EXCEPTIONS APIS SUMMARY 147
13.2 EXCEPTION FUNCTIONS DEFINITIONS 149
13.2.1 gpOS_exception_init 149
13.2.2 gpOS_exception_clear 150

14 WAKELOCKS 151

14.1 WAKELOCKS OVERVIEW 151
14.2 WAKELOCKS USAGE EXEMPLE 152
14.3 WAKELOCKS HANDLING APIS SUMMARY 153
14.4 WAKELOCKS FUNCTIONS DEFINITION 154
14.4.1 gpOS_wakelock_init 154
14.4.2 gpOS_wakelock_register 155
14.4.3 gpOS_wakelock_acquire 156
14.4.4 gpOS_wakelock_release 157
14.4.5 gpOS_wakelock_status 158
14.4.6 gpOS_wakelock_task_next_activity 159
14.4.7 gpOS_wakelock_get_value_unp 160

15 DISCLAIMER 161

2.2 List of Tables

Table 1: Revision history 7
Table 2: Acronyms 8
Table 3: OS20+ header files table 10
Table 4: OS20+ common data types 14
Table 5: OS20+ general function list 15
Table 6: Kernel module data types list 19
Table 7: Kernel module function list 19
Table 8:Types defined in gpOS_memory.h 33
Table 9: Function defined in gpOS_memory.h 34
Table 10:Types defined in gpOS_task.h 56
Table 11: Function defined in gpOS_task.h 58
Table 12: Types defined in gpOS_semaphore.h 89
Table 13: Functions defined in gpOS_semaphore.h 90
Table 14: Types defined in gpOS_mutex.h 100
Table 15: Functions defined in gpOS_mutex.h 100
Table 16: Types defined in gpOS_message.h 110
Table 17: Functions defined in gpOS_message.h 111
Table 18: Types defined in gpOS_time.h 126
Table 19: Functions defined in gpOS_time.h 126
Table 20: Types defined in gpOS_interrupt.h 138
Table 21: Functions defined in gpOS_interrupt.h 138
Table 22: Types defined in except.h 148
Table 23: Functions defined in except.h 148
Table 24: Types defined in gpOS_wakelock.h 153
Table 25: Functions defined in gpOS_wakelock.h 153

2.3 List of Figures

Figure 1: Message Queues 108
Figure 2: OS20+ Message Elements 109
Figure 3: Time arithmetic 125

3 Document Management

3.1 Revision History

Rev	Date	Author	Notes
1.0	2011-01-01	F. Boggia	Initial draft
1.1	2011-02-25	A. Occhipinti	Added missing topics
1.2	2011-03-31	F. Boggia	Reviewed and formatted
2.0	2011-04-08	F. Boggia	Updated to OS20+ release 3.1.8 Added missing topics Initial draft release,
2,1	2011-04-26	F. Boggia	Updated to OS20+ release 3.2.9 Added some missing APIs.
2.2	2011-09-08	F. Boggia	Updated to OS20+ release 3.2.10 Added new APIs
2.3	2011-11-02	F. Boggia	Updated to OS20+ release 3.2.11 Added new APIs
2.4	2012-02-07	F. Boggia	Updated to OS20+ release 3.3.12 Updated some APIs Added new APIs Corrected errors.
2.5	2013-04-23	F. Boggia	Updated to OS20+ release 3.4.22 Added new APIs. Corrected errors.
3.0	2015-02-23	A. Di Sena	Refactoring function signatures
3.1	2015-04-24	A. Di Sena F. Boggia	Refactoring function signatures Fixed typo errors Fixed some sections All APIs references are now links to proper section
3.2	2016-04-30	F. Boggia	Updated headers
3.3	2016-06-30	J. Durand	Added Wakelocks section

Table 1: Revision history

3.2 Acronyms

Keyword	Definition
BSP	Board Support Package
OS	Operating system

Table 2: Acronyms

3.3 Reference Documents

None

3.4 Contact info

Keyword	Definition
F. Boggia	fulvio.boggia@st.com
A. Occhipinti	aldo.occhipinti@st.com

4 Preface

This manual forms a combined user guide and reference manual for the ARM port of the OS20+ Real-Time Kernel.

4.1 Conventions used

The following typographical conventions may occur in this manual:

Bold type	Used to denote special terminology e.g. register or pin names.
`Teletype`	Used to distinguish command options, command line examples, code fragments, and program listings from normal text.
Italic type	In command syntax definitions, used to represent an argument or parameter. Used within text for emphasis and for book titles.
Braces {}	Used to denote a list of optional items in command syntax.
Brackets []	Used to denote optional items in command syntax.
Ellipsis …	In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
|	In command syntax, separates two mutually exclusive alternatives.

4.2 Overview

The OS20+ kernel features:

- Multi-priority pre-emptive scheduling based on several levels of priority.
- Semaphores.
- Message queues.
- Mutexes.
- Time lists.
- High resolution 32 bit timers.
- Interrupt handling.
- Wakelocks.

- Very small memory requirement.
- Pseudo-dynamic and full-dynamic memory allocation.

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.

All the symbols defined by a service have the service name as the first component of the name, see below.

Header	Description
`except.h`	Exceptions handling support functions
`gpOS_interrupt.h`	Interrupts handling support functions
`gpOS_kernel.h`	Kernel functions
`gpOS_message.h`	Message handling functions
`gpOS_memory.h`	Pseudo-dynamic memory allocation functions
`gpOS_mutex.h`	Mutual excluding semaphores
`gpOS_time.h`	Timer functions
`gpOS_semaphore.h`	Semaphore functions
`gpOS_tasks.h`	Task functions
`gpOS_timelist.h`	Timed lists functions
`gpOS_wakelock.h`	Wakelock functions

Table 3: OS20+ header files table

4.2.1 Naming

OS20+ is a custom implementation of the Generic Positioning Operating System (gpOS) abstraction layer, used by GNSS products as main operating system interface. All OS names will have `gpOS_` as prefix.

All the functions in OS20+ follow a common naming scheme. This is:

```
service_action
```

where `service` is the service name, which groups all the functions, and `action` is the operation to be performed.

4.3 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, i.e.:

- Tasks.
- Semaphores.
- Mutexes.

A class is a purely abstract concept, which describes a collection of data items and a list of operations which can be performed on it.

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

For many classes within OS20+, there are different flavours. For example, the semaphore class has FIFO and priority flavours. When a particular object is created, which flavour is required must be specified by using a qualifier on the object creation function, and that is then fixed for the lifetime of that object. All the operations specified by a particular class can be applied to all objects of that class; however, how they will behave may depend on the flavour of that class. So the exact behaviour of `gpOS_semaphore_wait()` will depend on whether it is applied to a FIFO or priority semaphore object.

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.

4.3.1 Object Lifetime

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

When using `class_create` calls, the memory for the object structure is allocated from the system partition. Therefore this partition must be initialized (by calling `gpOS_memory_init`) before any `class_create` calls are made. When using the `class_create_p` calls, OS20+ allocates space from a user nominated partition.

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

4.3.2 How this manual is organized

The division of OS20+ functions into services is also used in this manual. Each of the major service types is described separately, using a common layout:

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

The remaining sections of this chapter describe the main concepts on which OS20+ is founded. It is advisable to read the remainder of this chapter if you are a first time user.

4.4 Tasks

Tasks are the main elements of the OS20+ multi-tasking facilities. A task describes the behaviour of a discrete, separable component of an application, behaving like a separate program, except that it can communicate with other tasks.

Each task has its own data area in memory, including its own stack and the current state of the task. These data areas must be specified by the user. The code and global static data 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 section 7.

4.5 Priority

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

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

A task’s priority is set when the task is initialized, although it may be changed later. OS20+ provides the user with sixteen levels of priority.

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 section 5.

4.6 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.

Semaphores are described in more detail in section 8.

4.7 Mutexes

OS20+ provides to developer mutexes using semaphores. Mutexes are used to access protected areas inside same task. Using a simple semaphore, a couple of lock accesses will lock the task. Using mutex, the task will go on, but it will need to unlock it as many times as it has locked it. Mutexes are described in section 9.

4.8 Message queues

Message queues provide a buffered communication method for tasks and are described in section 10.

4.9 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. Section 11 provides an overview of how time is handled in OS20+. Time-out related functions are described in sections 7.7 for tasks, 8.2 for semaphores, 10.3 for messages.

OS20+ provides a 32 bit high resolution timer by efficiently using the hardware timer provided on the device.

4.10 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 section 12.

4.11 Wakelocks

The system clocking is under wakelocks control according to Tasks MIPS requirements. See chapter 14 for further details.

4.12 OS20+ Data types

4.12.1 Basic types: gpOS_types.h

Note: *this replaces os20_types.h in previous version*

OS20+ defines a set of general data types used in almost all modules for interfacing. Below you can find a list of what is defined in header file `gpOS_types.h`.

Type	Description

`gpOS_stack_t`	Stack type
`gpOS_error_t`	Error type
`gpOS_bool_t`	boolean type
`gpOS_cpsr_t`	Current processor status registers type

Table 4: OS20+ common data types

About the errors that a procedure can return, they are defined in type `gpOS_error_t`:

```
#define gpOS_SUCCESS  0
#define gpOS_FAILURE  -1

typedef int gpOS_error_t;
```

OS20+ also implements a specific boolean type (machine independent) used internally and for its APIs, `gpOS_bool_t`:

```
#define gpOS_TRUE    (1 == 1)
#define gpOS_FALSE   (0 == 1)

typedef int gpOS_bool_t;
```

4.12.2 Enhanced data types

OS20+ defines some other useful types. For exceptions installation, OS20+ specifies the type of the procedure that can be installed:

```
typedef void (*os20_exception_vector_t)(void);
```

In some occasion a portion of code must be executed in supervisor mode for several reasons (accessing some ARM features, coprocessor …). OS20+ provides an API to do that and uses the following types to specify procedure and parameter passed:

```
typedef void (*gpOS_syscall_func_t)(void *);
typedef void *gpOS_syscall_param_t;
```

See section **Error! Reference source not found.** to see how to use them.

4.13 OS20+ general APIs summary

All the definitions related to general OS20+ APIs are in the single header file, `gpOS.h`.

Note: *this replaces os20.h in previous version*

Function	Description
`gpOS_version`	Returns a string pointer to OS20+ version.

Table 5: OS20+ general function list

4.14 OS20+ functions definitions

4.14.1 gpOS_version

Note: this replaces os20_version in previous version

Return a string pointer containing OS20+ version.

Synopsis

```
#include <gpOS.h>

const tChar * gpOS_version( void);
```

Arguments

None.

Results

`const tChar *`:	pointer to a string containing the version of OS20+.

Errors

None.

Description

`gpOS_version` can be called to get the pointer to a string containing the OS20+ version.

5 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.

5.1 Implementation

The kernel maintains two vitally important pieces of information:

- Which is 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 actually stored as a number of queues, one for each priority, with the tasks stored in the order in which they will be executed.

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

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 'pre-emption', because the new task has pre-empted the old one.

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

Periodically the scheduler is called to time-slice the currently executing task. If there are other tasks which are of the same priority as the current task, then the state of the current task will be saved onto the back of the current priority queue, and the task at the front of the queue installed in its place. In this way all processes at the same priority get a chance to run. Programmer has the flexibility to enable or not the time slicing feature. By default, it is turned off.

When an interrupt has been serviced and there are no other lower-priority interrupts being serviced the kernel is called to see if a reschedule is required. For example an interrupt handler could have signalled a semaphore such that a higher priority task becomes ready to run when the interrupt handler completes.

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

5.2 OS20+ kernel

The operations which must be performed to run OS20+ kernel are its installation and start. This is done by calling the functions `gpOS_kernel_init` and `gpOS_kernel_start` which is usually performed as the first operation in `main`:

```
kernel_config_t kernel_config =
{
   OS20_SVC_MODE_STACK_SIZE,
   OS20_UND_MODE_STACK_SIZE,
   0,
   OS20_IRQ_MODE_STACK_SIZE,
   0
};

gpOS_kernel_init( kernel_config);

...

gpOS_kernel_start();
```

The `gpOS_kernel_init` function passes the `kernel_config` array containing information about stacks used for IRQ and SVC states. The `gpOS_kernel_start` function is used to start the kernel and detach the running code in a task named as root task.

These initialize and start functions can only be called once from the main body of the application.

Other operations that can be done on OS20+ kernel are `gpOS_kernel_lock` and `gpOS_kernel_unlock`. Locking the kernel will prevent the scheduler to deschedule it also if other tasks with higher priority are appointed to be scheduled after, for example, an interrupt. Once the critical section is executed, the task that locked the kernel must unlock it.

5.3 System calls

Sometimes user could want to execute a portion of code in a privileged status. For example, there are assembly instructions that can be executed only in supervisor mode.

OS20+ provides an API to execute a piece of code in privileged status:

```
extern void gpOS_kernel_user_system_call
(
   gpOS_syscall_func_t syscall_func,
   gpOS_syscall_param_t syscall_param
);
```

The function that is called in privileged status must be of the following type:

```
typedef void *gpOS_syscall_param_t;
typedef void (*gpOS_syscall_func_t)( gpOS_syscall_param_t);
```

5.4 Kernel APIs summary

All the definitions related to the kernel are in the single header file, `gpOS_kernel.h`.

Note: this replaces kernel.h in previous version

Type	Description
`gpOS_kernel_config_t`	Kernel initialization data
`gpOS_syscall_param_t`	Type of a system call parameter
`gpOS_syscall_func_t`	Type of a system call procedure

Table 6: Kernel module data types list

Function	Description
`gpOS_kernel_init`	Initialize for pre-emptive scheduling.
`gpOS_kernel_start`	Start pre-emptive scheduling regime.
`gpOS_kernel_get_active_task`	Get ID of current running task
`gpOS_kernel_set_timeslice`	Change time slicing status
`gpOS_kernel_get_timeslice`	Get time-slicing status
`gpOS_kernel_user_system_call`	Execute a function in super user mode
`gpOS_kernel_lock`	Lock scheduler
`gpOS_kernel_unlock`	Unlock scheduler
`gpOS_kernel_get_context_switch_number`	Returns number of context switches in latest second
`gpOS_kernel_get_interrupts_occurred_number`	Returns number of IRQs occurred in latest second
`gpOS_kernel_get_interrupts_spent_time`	Return number of ticks spent in IRQ mode during latest second

Table 7: Kernel module function list

5.5 Kernel functions definitions

5.5.1 gpOS_kernel_init

Note: this replaces kernel_init in previous version

Initialize for pre-emptive scheduling.

Synopsis

```
#include <gpOS.h>

gpOS_error_t gpOS_kernel_init
(
   gpOS_partition_t *part,
   gpOS_kernel_config_t arm_stacks
);
```

Arguments

`gpOS_partition_t *`:	The partition in which to create ARM processor modes stacks.
`gpOS_kernel_config_t`:	pointer to the array containing the sizes of stacks that will be used for ARM processor modes.

Results

`gpOS_error_t`:	`gpOS_SUCCESS` if kernel is correctly initialized, `gpOS_FAILURE` otherwise.

Errors

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

Description

`gpOS_kernel_init` must be called before any task is created. It will:

- Allocate memory for selected ARM processor modes stacks as defined in `arm_stacks` table using partition `part`. If `part` is `NULL`, OS20+ root partition is used;
- Initialize OS20+ scheduler;
- The root task is created and the calling procedure is initialized as the root task.

5.5.2 gpOS_kernel_start

Note: this replaces kernel_start in previous version

Start preemptive scheduling regime.

Synopsis

```
#include <gpOS.h>

gpOS_error_t gpOS_kernel_start( void);
```

Arguments

None.

Results

`gpOS_error_t`:	`gpOS_SUCCESS` if kernel is correctly initialized, `gpOS_FAILURE` otherwise.

Errors

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

Description

`gpOS_kernel_start` must be called before any tasks are created. 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 `gpOS_TASK_MAX_USR_PRIORITY`.

Note: prior to calling this function `gpOS_kernel_init` must have been called. This function should only be called once.

5.5.3 gpOS_kernel_get_active_task

Note: this replaces kernel_get_active_task in previous version

Return the pointer to the task class related to the process currently running.

Synopsis

```
#include <gpOS.h>

gpOS_task_t* gpOS_kernel_get_active_task( void);
```

Arguments

None.

Results

`gpOS_task_t*`:	pointer to the task class related to the process currently running.

Errors

None.

Description

`gpOS_kernel_get_active_task` is called to retrieve the pointer to the task class that is currently running and executing the calling code.

5.5.4 gpOS_kernel_set_timeslice

Note: this replaces kernel_set_timeslice in previous version

Configure OS20+ timeslicing (see section 5.1).

Synopsis

```
#include <gpOS.h>

void gpOS_kernel_set_timeslice( gpOS_bool_t);
```

Arguments

`gpOS_bool_t`:	new status for OS20+ timeslicing.

Results

None.

Errors

None.

Description

`gpOS_kernel_set_timeslice` is called to enable or disable the time slicing feature of OS20+. See section 5.1 for a description of how time slicing works in OS20+.

5.5.5 gpOS_kernel_get_timeslice

Note: this replaces kernel_get_timeslice in previous version

Return status of OS20+ timeslicing (see section 5.1).

Synopsis

```
#include <gpOS.h>

gpOS_bool_t gpOS_kernel_get_timeslice( void);
```

Arguments

None.

Results

`gpOS_bool_t`:	current status of OS20+ timeslicing.

Errors

None.

Description

`gpOS_kernel_get_timeslice` is called to retrieve status of slicing feature of OS20+. See section 5.1 for a description of how time slicing works in OS20+.

5.5.6 gpOS_kernel_user_system_call

Note: this replaces kernel_user_system_call in previous version

Execute a procedure in ARM supervisor mode.

Synopsis

```
#include <gpOS.h>

void gpOS_kernel_user_system_call
(
   gpOS_syscall_func_t,
   gpOS_syscall_param_t
);
```

Arguments

`gpOS_syscall_func_t`:	pointer to a function to be executed in supervisor mode;
`gpOS_syscall_param_t`:	parameter to be passed to that function.

Results

None.

Errors

None.

Description

`gpOS_kernel_user_system_call` is called to execute a procedure in supervisor mode. Following is an example to execute ARM coprocessor accesses in supervisor mode:

```
void enter_wfi_mode( void *)
{
  /* enters wait for interrupt mode for ARM946, */
  /* must be done in SVC mode !                 */
  __asm
  {
    MCR   p15, 0, r0, c7, c0, 4
  }
}

...

gpOS_kernel_user_system_call( enter_wfi_mode, NULL);
```

5.5.7 gpOS_kernel_lock

Note: this replaces kernel_lock in previous version

Lock kernel scheduler.

Synopsis

```
#include <gpOS.h>

void gpOS_kernel_lock( void);
```

Arguments

None.

Results

None.

Errors

None.

Description

This API locks kernel scheduler avoiding any event to deschedule task currently ongoing for one at higher or same priority. It can be used when critical portions of code must run. For example, it can be used when accessing data shared with other tasks.

Note: This API does not stops interrupts handling, so if the critical code can be affected by the execution of any interrupt service routine, `gpOS_kernel_lock` should be substituted (or used in conjunction) with `gpOS_interrupt_lock`.

This API can be called more than once, but at least an exact number of calls to `gpOS_kernel_unlock` must be done to ensure that scheduler starts operating again.

5.5.8 gpOS_kernel_unlock

Note: this replaces kernel_unlock in previous version

Unlock kernel scheduler.

Synopsis

```
#include <gpOS.h>

void gpOS_kernel_unlock( void);
```

Arguments

None.

Results

None.

Errors

None.

Description

This API tries to unlock kernel scheduler previously locked with one or more calls to `gpOS_kernel_lock`. To unlock completely the scheduler, it must be called at least an exact number of calls to `gpOS_kernel_lock`.

5.5.9 gpOS_kernel_get_context_switch_number

Note: this replaces kernel_get_context_switch_number in previous version

Returns number of context switches in latest second.

Synopsis

```
#include <gpOS.h>

tUInt gpOS_kernel_get_context_switch_number( void);
```

Arguments

None.

Results

`tUInt`:	number of context switches in latest second.

Errors

None.

Description

OS20+ stores information about how many context switches happened in a second. This information can be retrieved using this API. The information refers to a second and is refreshed every second.

5.5.10 gpOS_kernel_get_interrupts_occurred_number

Note: this replaces kernel_get_interrupts_occurred_number in previous version

Returns number of interrupts occurred in latest second.

Synopsis

```
#include <gpOS.h>

tUInt gpOS_kernel_get_interrupts_occurred_number( void);
```

Arguments

None.

Results

`tUInt`:	number of interrupts occurred in latest second.

Errors

None.

Description

OS20+ stores information about how many interrupts occurred in a second. This information can be retrieved using this API. The information refers to a second and is refreshed every second.

5.5.11 gpOS_kernel_get_interrupts_spent_time

Note: this replaces kernel_get_interrupts_spent_number in previous version

Get number of ticks spent in interrupt (IRQ) mode during last second.

Synopsis

```
#include <gpOS.h>

tUInt gpOS_kernel_get_interrupts_spent_time( void);
```

Arguments

None.

Results

`tUInt`:	number of ticks spent in interrupt (IRQ) mode during last second.

Errors

None.

Description

`gpOS_kernel_get_interrupts_spent_number()` retrieves the number of ticks spent by operating system while in IRQ mode during last second. The information refers to a second and is refreshed every second.

6 Memory and partitions

OS20+ introduces a simple method to allocate memory. It does not support standard ANSI C heap with `malloc` and `free` procedures, but implements a trivial allocator, which just increments a pointer to the next available block of memory. This means that it is quite impossible to freely free any memory back to the partition, but there is no wasted memory when performing memory allocations. This method 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 `gpOS_memory_allocate` and the time taken to allocate memory is constant.

Note: the user can safely free a memory chunk only immediately after it is allocated. This could be useful at initialization time to free memory if a module fails startup.

Currently OS20+ can handle common dynamic memory allocation (as `malloc`/`free` for standard C or `new`/`delete` for standard C++). The implementation relies on C runtime library implementation, but can be properly tailored by customer.

Note: OS20+ is not responsible to handle memory fragmentation. Dynamic memory must be carefully used by customer to avoid that dynamically allocated space will not ends up in unwanted out of memory conditions.

6.1 Partitions

OS20+ uses at startup a default portion of memory, defined as root partition, to allocate memory. But the user has the opportunity to handle a separate area of memory as a partition, so that it can allocate memory from that separate area as it is done on OS20+ root partition.

Partitions can be of different types. They are defined in following type:

```
typedef enum gpOS_memory_type_e
{
  gpOS_MEMORY_TYPE_SYSTEM_HEAP,
  gpOS_MEMORY_TYPE_SIMPLE,
  gpOS_MEMORY_TYPE_HEAP
} gpOS_memory_type_t;
```

Type `gpOS_MEMORY_TYPE_SIMPLE` defines a very basic memory type, for which the main feature is to reduce at minimum extra memory needed by allocation process. When a buffer is allocated in a partition of this type, there will be no extra information saved. So freeing memory in a partition of this type is possible only if buffers as freed in reverse order as they were allocated. If buffers are freed randomly, the result is unpredictable.

Type `gpOS_MEMORY_TYPE_HEAP` defines HEAP type as handled by specific toolchain C runtime library. Currently there is no possibility to create any partition of this type, as the implementation is toolchain specific. A partition of this type can only be setup at build time using proper symbols. This is platform and toolchain dependent and must be specified for each platform developing environment.

6.2 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 one of the following function:

```
extern tSize gpOS_memory_getheapsize_p( gpOS_partition_t *);
extern tSize gpOS_memory_getheapfree_p( gpOS_partition_t *);
```

The first one returns the size of the partition, the second one the free space in the partition.

6.3 Creating a new partition type

OS20+ allows a user to create partitions. To do so, the user must call the `gpOS_memory_create_partition` function providing pointer to user area and its size. The user must guarantee that that area will not be used by any other portion of software.

6.4 Allocating memory

OS20+ provides two APIs to allocate memory:

```
extern void *gpOS_memory_allocate( const tSize size);
extern void *gpOS_memory_allocate_p
(
   gpOS_partition_t *, const tSize size
);
```

The first uses OS20+ system partition to reserve requested memory, while the second one will use a custom user partition to reserve requested memory. If the partition does not have the requested space, a `NULL` pointer is returned, otherwise, they return the pointer to the allocated region.

6.5 Freeing memory

Blocks of memory allocated with `gpOS_memory_allocate` or `gpOS_memory_allocate_p` can be freed if needed.

As explained in chapter 6.1, if type is `gpOS_MEMORY_TYPE_SIMPLE`, this operation can be dangerous, as it is important that any deallocation corresponds to an allocation in the same sequence, otherwise the partition where memory is deallocated could become corrupted.

If the type of partition is `gpOS_MEMORY_TYPE_HEAP`, then memory allocation is handled by `malloc`/`free` APIs from either C runtime library specific for toolchain or tailored versions of them. So freeing can be accomplished also in an undefined order.

The APIs available to deallocate memory are:

```
extern void *gpOS_memory_deallocate( const tSize size);
extern void *gpOS_memory_deallocate_p
(
   gpOS_partition_t *, const tSize size
);
```

The first uses OS20+ system partition to free requested memory, while the second one will use a custom user partition to free requested memory. As stated, they should be used with care, and their usage should be limited to free memory if some user initialization goes wrong. Here is an example on how to use them in case of partition of type `gpOS_MEMORY_TYPE_SIMPLE`:

```
tChar *buf1, *buf2;

buf1 = (tChar *)gpOS_memory_allocate( 100);
buf2 = (tChar *)gpOS_memory_allocate( 200);

if( (buf1 == NULL) || (buf2 == NULL))
{
   // Not enough memory, free everything
   gpOS_memory_deallocate( 200);
   gpOS_memory_deallocate( 100);
}
```

6.6 Memory and partition APIs summary

All the definitions related to memory partitions can be obtained by including the header file, `gpOS.h`, which itself includes the header file `gpOS_memory.h`

Note: this replaces memory.h in previous version

Type	Description
`gpOS_partition_t`	A memory partition
`gpOS_memory_type_t`	Type of memory

Table 8:Types defined in gpOS_memory.h

Function	Description
`gpOS_memory_init`	Initialize OS20+ root partition
`gpOS_memory_allocate`	Allocate a block of memory from root partition
`gpOS_memory_allocate_p`	Allocate a block of memory from a user partition
`gpOS_memory_deallocate`	Remove a block of memory from root partition
`gpOS_memory_deallocate_p`	Remove a block of memory from a user partition
`gpOS_memory_create_partition`	Create a new partition; start pointer and size of the partition shall be passed as argument of the function
`gpOS_memory_getheapsize`	Get total size of root partition
`gpOS_memory_getheapsize_p`	Get total size of user partition
`gpOS_memory_getheapfree`	Get free size of root partition
`gpOS_memory_getheapfree_p`	Get free size of user partition
`gpOS_memory_getheaprequested`	Get size of heap requested for root partition
`gpOS_memory_getheaprequested_p`	Get size of heap requested for user partition
`gpOS_memory_getnextpartition`	Get next available partition

Table 9: Function defined in gpOS_memory.h

6.7 Memory and partition functions definitions

6.7.1 gpOS_memory_init

Note: this replaces memory_init in previous version

Initialize OS20+ root partition

Synopsis

```
#include <gpOS.h>

void gpOS_memory_init( void);
```

Arguments

None.

Results

None.

Errors

None.

Description

Initialize memory module of OS20+. It must be called before trying to use any memory API.

6.7.2 gpOS_memory_allocate

Note: this replaces memory_allocate in previous version

Allocate a block of memory from root partition

Synopsis

```
#include <gpOS.h>

void* gpOS_memory_allocate(const tSize size);
```

Arguments

`const tSize`:	size in bytes of the memory to allocate.

Results

`void *`:	pointer to the allocated memory, or `NULL` if there is not sufficient memory available.

Errors

Failure is caused by insufficient space to allocate the requested area.

Description

`gpOS_memory_allocate` allocates a block of memory of `size` bytes from root partition. It returns the address of a block of memory of the required size, which is suitably aligned to contain any type.

6.7.3 gpOS_memory_allocate_p

Note: *this replaces memory_allocate_p in previous version*

Allocate a block of memory from user partition

Synopsis

```
#include <gpOS.h>

void* gpOS_memory_allocate_p
(
   gpOS_partition_t * partition_ptr,
   const tSize size
);
```

Arguments

`gpOS_partition_t *`:	pointer to a user partition;
`const tSize`:	size in bytes of the memory to allocate.

Results

`void *`:	pointer to the allocated memory, or `NULL` if there is not sufficient memory available.

Errors

Failure is caused by insufficient space to allocate the requested area.

Description

`gpOS_memory_allocate_p` allocates a block of memory of `size` bytes from a specified user partition. It returns the address of a block of memory of the required size, which is suitably aligned to contain any type.

Note: *If a `NULL` pointer is specified for part, instead of a valid partition pointer, the OS20+ root partition is used.*

6.7.4 gpOS_memory_deallocate

Note: this replaces memory_deallocate in previous version

Free a block of memory from root partition

Synopsis

```
#include <gpOS.h>

void gpOS_memory_deallocate( void *mem_ptr);
```

Arguments

`void *`:	pointer to previous allocated area.

Results

None.

Errors

None.

Description

`gpOS_memory_deallocate` frees a block of memory previously allocated. Refer to section 6.5 to see how it must be used.

6.7.5 gpOS_memory_deallocate_p

Note: this replaces memory_deallocate_p in previous version

Free a block of memory from user partition

Synopsis

```
#include <gpOS.h>

void gpOS_memory_deallocate_p
(
   gpOS_partition_t * partition_ptr,
   void *mem_ptr
);
```

Arguments

`gpOS_partition_t *`:	pointer to a user partition;
`void *`:	pointer to previous allocated area.

Results

None.

Errors

Failure is caused by insufficient space to allocate the requested area.

Description

`gpOS_memory_deallocate_p` frees a block of memory of `size` bytes from root partition. Refer to 6.5 to see how it must be used.

Note *If a `NULL` pointer is specified for part, instead of a valid partition pointer, the OS20+ root partition is used.*

6.7.6 gpOS_memory_create_partition

Note: this replaces memory_create_partition in previous version

Create a partition from a specified user reserved area

Synopsis

```
#include <gpOS.h>

gpOS_partition_t* gpOS_memory_create_partition
(
   gpOS_memory_type_t type,
   void *user_area_ptr,
   const tSize size
);
```

Arguments

`gpOS_memory_type_t`:	type of memory of the new partition;
`void *`:	pointer to a user reserved area;
`unsigned int`:	size in bytes of the user reserved area.

Results

`gpOS_partition_t *`:	pointer to the partition handler for user reserved area, or `NULL` if there is not sufficient memory available in root partition to create partition handler.

Errors

Failure is caused by insufficient space to allocate the partition handler.

Description

`gpOS_memory_create_partition` allocates a partition handler over the root partition and initialize it with area pointer and size passed as parameters.

6.7.7 gpOS_memory_getheapsize

Note: this replaces memory_getheapsize in previous version

Return total size of OS20+ root partition

Synopsis

```
#include <gpOS.h>

tSize gpOS_memory_getheapsize( void);
```

Arguments

None.

Results

`tSize`:	total size in bytes of OS20+ root partition.

Errors

None.

Description

`gpOS_memory_getheapsize` returns the total size of OS20+ root partition.

6.7.8 gpOS_memory_getheapsize_p

Note: this replaces memory_getheapsize_p in previous version

Return total size of OS20+ user partition

Synopsis

```
#include <gpOS.h>

tSize gpOS_memory_getheapsize_p( gpOS_partition_t *part);
```

Arguments

`gpOS_partition_t *`:	pointer to a user partion;

Results

`tSize`:	total size in bytes of user partition.

Errors

None.

Description

`gpOS_memory_getheapsize_p` returns the total size of user partition.

6.7.9 gpOS_memory_getheapfree

Note: this replaces memory_getheapfree in previous version

Return free size of OS20+ root partition

Synopsis

```
#include <gpOS.h>

tSize gpOS_memory_getheapfree( void);
```

Arguments

None.

Results

`tSize`:	free size in bytes of OS20+ root partition.

Errors

None.

Description

`gpOS_memory_getheapfree` returns the free size of OS20+ root partition.

6.7.10 gpOS_memory_getheapfree_p

Note: this replaces memory_getheapfree_p in previous version

Return free size of a user partition

Synopsis

```
#include <gpOS.h>

tSize gpOS_memory_getheapfree_p( gpOS_partition_t *part);
```

Arguments

`gpOS_partition_t *`:	pointer to a user partion;

Results

`tSize`:	free size in bytes of a user partition.

Errors

None.

Description

`gpOS_memory_getheapfree_p` returns the free size of a user partition.

6.7.11 gpOS_memory_getheaprequested

Note: this replaces memory_getheaprequested in previous version

Return requested size of OS20+ root partition

Synopsis

```
#include <gpOS.h>

tSize gpOS_memory_getheaprequested( void);
```

Arguments

None.

Results

`tSize`:	requested size in bytes of OS20+ root partition.

Errors

None.

Description

`gpOS_memory_getheaprequested` returns the size of OS20+ root partition requested for allocation.

6.7.12 gpOS_memory_getheaprequested_p

Note: this replaces memory_getheaprequested_p in previous version

Return requested size of a user partition

Synopsis

```
#include <gpOS.h>

tSize gpOS_memory_getheaprequested_p( gpOS_partition_t *part);
```

Arguments

`gpOS_partition_t *`:	pointer to a user partion;

Results

`tSize`:	requested size in bytes of a user partition.

Errors

None.

Description

`gpOS_memory_getheapfree_p` returns the size of a user partition requested for allocation.

6.7.13 gpOS_memory_getnextpartition

Note: this replaces memory_getnextpartition in previous version

Get pointer to next partition in the system

Synopsis

```
#include <gpOS.h>

gpOS_partition_t *gpOS_memory_getnextpartition
(
   gpOS_partition_t *part
);
```

Arguments

`gpOS_partition_t *`:	pointer to a partion;

Results

`gpOS_partition_t *`:	pointer to next patition.

Errors

None.

Description

`gpOS_memory_getnextpartition` returns the pointer to the partition next to the provided one. If `part` is `NULL`, the partition next to the OS20+ standard heap is returned.

7 Tasks

Tasks are separate threads of control, which run independently. A task describes the behaviour 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.

7.1 OS20+ tasks

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

A task is identified by its `gpOS_task_t` structure and this should always be used when referring to the task. A pointer to the `gpOS_task_t` structure is called the task's ID.

The task's data structure must be allocated by the user declaring the `gpOS_task_t` data structure. (This structure is 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 `gpOS_task_t` data structure must be allocated and initialized and a stack and function must be associated with them. This is done using the `gpOS_task_create` or `gpOS_task_create_p` functions. See section 7.4.

7.2 OS20+ priorities

The number of OS20+ task priorities and the highest and lowest task priorities are defined using the macros in the header file `gpOS_task.h`. Numerically higher priorities pre-empt lower priorities e.g. 3 is a higher priority than 2.

A task's initial priority is defined when it is initialized or created, see section 7.4. 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 `gpOS_kernel_start`. This task starts running with the highest priority available, `gpOS_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:

```
tInt gpOS_task_get_priority( task_t* task)
```

that 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 following function:

```
tInt gpOS_task_set_priority( gpOS_task_t* task, tInt priority);
```

that 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.

7.3 Scheduling

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

The currently executing task is always the one with the highest priority. If a task with a higher priority becomes ready to run then the OS20+ scheduler will save the current task's state and will make the higher priority task the current task. The current task will run to completion unless it is pre-empted by a higher priority task, and so on. Once a task has completed, the next highest priority task will start executing.

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

```
void gpOS_task_lock( void);
void gpOS_task_unlock( void);
```

These functions (that has the same effect of `gpOS_kernel_lock` and `gpOS_kernel_unlock`) should always be called as a pair and can be used to create a critical region where one task is prevented from pre-empting another. Calls to `gpOS_task_lock` can be nested, and the lock will not be released until an equal number of calls to `gpOS_task_unlock` have been made. Once `gpOS_task_unlock` is called, the scheduler will start the highest priority task which is available, running. This may not be the task which calls `gpOS_task_unlock`.

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

Note that when this lock is in place the task can still be interrupted by interrupt handlers. Interrupts can be disabled and enabled using the `gpOS_interrupt_lock` and `gpOS_interrupt_unlock` functions.

7.4 Creating a task

The following function is provided for starting a task running:

```
gpOS_task_t* gpOS_task_create(
   gpOS_task_function_t  func_ptr,
   gpOS_task_param_t func_param,
   const tSize stack_size,
   const tInt priority,
   const tChar *name,
   gpOS_task_flags_t flags
);
```

`gpOS_task_create` will allocate memory for the task's stack, control block `gpOS_task_t` and task descriptor `tdesc_t`. This functions set up the task and starts it running the function specified at `func_ptr`, passing specified `func_param`. This is done by initializing the data structure `gpOS_task_t` and associating a function with it.

The function is passed in as a pointer to the task's entry point. It takes a single pointer to be used as the argument to the user function. A cast to `void*` should be performed in order to pass in a single word sized parameter (e.g. an `tInt`). Otherwise a data structure should be set up.

`gpOS_task_create` requires the stack size to be specified. Stack is used for a function's local variables and parameters and to save the register context when the task is pre-empted.

The function also requires 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 `gpOS_task.h` by the macros `gpOS_TASK_PRIORITY_LEVELS`, `gpOS_TASK_MAX_USR_PRIORITY` and `gpOS_TASK_MIN_USR_PRIORITY`.

7.5 Synchronizing tasks

Tasks synchronize their actions with each other using semaphores, as described in section 8.

7.6 Communicating between tasks

Tasks communicate with each other by using message queues, as described in section 10.

7.7 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 gpOS_task_delay(gpOS_clock_t delay);
void gpOS_task_delay_until(gpOS_clock_t delay);
```

Both functions wait for a period of time and then return. `gpOS_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.

`gpOS_task_delay` waits until the given time has elapsed, i.e. it delays execution for the specified number of timer ticks. If the time given is negative, no delay takes place.

`gpOS_task_delay` or `gpOS_task_delay_until` may be used for data logging or causing an event at a specific time. A high priority task can wait until a certain time; when it wakes it will pre-empt any lower priority task that is running and perform the time-critical function.

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

```
gpOS_clock_t time;

time = gpOS_time_now();

for (;;)
{
   time = gpOS_time_plus(time, delay);
   gpOS_task_delay_until(time);
   initiate_regular_event();
}
```

7.8 Suspending tasks

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

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

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

A task is suspended using the call:

```
gpOS_error_t gpOS_task_suspend( gpOS_task_t* task)
```

where `task` is the task to be suspended. A task may suspend itself by specifying Task as `NULL`. The result is `gpOS_SUCCESS` if the task was successfully suspended, `gpOS_FAILURE` if it failed. This call will fail if the task has terminated. A task may be suspended multiple times by executing several calls to `gpOS_task_suspend`. It will not start executing again until an equal number of `gpOS_task_resume` calls have been made.

A task is resumed using the call:

```
gpOS_error_t gpOS_task_resume( gpOS_task_t* task)
```

where `task` is the task to be resumed. The result is `gpOS_SUCCESS` if the task was successfully resumed, `gpOS_FAILURE` 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 `gpOS_TASK_FLAGS_SUSPENDED` when calling `gpOS_task_create`. 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 `gpOS_task_resume`, and it will start executing from its entry point.

Note: *If the task that is suspended is waiting on a semaphore or mutex or message queue, that resource will stay locked until task is resumed back.*

7.9 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:

```
gpOS_task_t* gpOS_task_get_id( void)
```

For each task, a set of information can be obtained using proper function:

```
const tChar* gpOS_task_get_name( gpOS_task_t* task);
void* gpOS_task_get_stack_base( gpOS_task_t* task);
void* gpOS_task_get_stack_ptr( gpOS_task_t* task);
tSize gpOS_task_get_stack_size( gpOS_task_t* task);
tSize gpOS_task_get_stack_used( gpOS_task_t* task);
tUInt gpOS_task_get_cpuusage( gpOS_task_t* task);
tUInt gpOS_task_get_runtime( gpOS_task_t* task);
```

See section specific for each function for details.

7.10 Task 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* gpOS_task_set_data( gpOS_task_t* task, void* data);
```

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

```
void* gpOS_task_get_data( gpOS_task_t* task);
```

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

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

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

```
typedef struct
{
   char buffer[BUFFER_SIZE];
   char* buffer_next;
   char* buffer_end;
} ptd_t;

char buffer_read(void)
{
   ptd_t *ptd;
   ptd = gpOS_task_get_data( NULL);
   if (ptd->buffer_next == ptd->buffer_end)
   {
   ... fill buffer ...
   }
   return *(ptd->buffer_next++);
}

int main()
{
   ptd_t *ptd;
   gpOS_task_t*task;
   ... create a task ...
   ptd = gpOS_memory_allocate( sizeof(ptd_t));
   ptd->buffer_next = ptd->buffer_end = ptd->buffer;
   gpOS_task_set_data( gpOS_task, ptd);
}
```

7.11 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 gpOS_task_exit( gpOS_task_exit_status_t 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 `gpOS_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 `gpOS_task_exit`), it will call an onexit handler. This function allows any application specific tidying up to be performed before the task terminates. The onexit handler is installed by calling:

```
gpOS_task_onexit_function_t gpOS_task_set_onexit_function
(
   gpOS_task_onexit_function_t function
);
```

The onexit handler function must have a prototype of:

```
typedef void (*gpOS_task_onexit_function_t)
(
   gpOS_task_t *task,
   gpOS_task_exit_status_t param
);
```

When the handler function is called, `task` specifies the task which has exited, and `param` is the task's exit status. The following code example shows how a task's exit code can be stored in its task data, and retrieved later by another task which is notified of the termination through `gpOS_task_wait`.

```
void onexit_handler(gpOS_task_t* task, int param)
{
   gpOS_task_set_data(NULL, (void*)param);
}

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

   ...  init OS20+ ...

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

   ...  create the tasks  ...

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

7.12 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 gpOS_task_wait
(
   gpOS_task_t** task_list,
   int ntasks,
   const gpOS_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 `gpOS_task_wait` can be used inside a loop.

The timeout period for `gpOS_task_wait` may be expressed as a number of ticks or it may take one of two values: `gpOS_TIMEOUT_IMMEDIATE` indicates that the function should return immediately, even if no tasks have terminated, and `gpOS_TIMEOUT_INFINITY` indicates that the function should ignore the timeout period, and only return when a task terminates.

7.13 Tasks APIs summary

All the definitions related to tasks handling can be obtained by including the header file, `gpOS.h`, which itself includes the header file `gpOS_task.h`.

Note: this replaces task.h in previous version

Type	Description
`gpOS_task_t`	Task handler
`gpOS_task_flags_t`	Task status flag
`gpOS_task_state_t`	Task state
`gpOS_task_param_t`	Parameter of procedure executed by task
`gpOS_task_function_t`	Procedure executed by task
`gpOS_task_exit_status_t`	Exit status returned by task when it ends
`gpOS_task_onexit_function_t`	Task one exit function

Table 10:Types defined in gpOS_task.h

Function	Description
gpOS_task_create	Create a new task in system partition
gpOS_task_create_p	Create a new task in user partition
gpOS_task_delete	Delete a task
gpOS_task_delay	Suspend task execution for a delay
gpOS_task_delay_until	Suspend task execution up to a given timeout
gpOS_task_suspend	Suspend task execution
gpOS_task_resume	Resume task execution
gpOS_task_get_id	Return current task handler
gpOS_task_set_priority	Change priority of a task
gpOS_task_get_priority	Get priority of a task
gpOS_task_get_stack_base	Get base pointer of task stack
gpOS_task_get_stack_ptr	Get current pointer of task stack
gpOS_task_get_stack_size	Get total size of task stack
gpOS_task_get_stack_used	Get used size of task stack
gpOS_task_get_cpuusage	Get task CPU usage in last second
gpOS_task_get_runtime	Get number of ticks a task was in active state
gpOS_task_get_head	Get pointer to the first created task
gpOS_task_get_next	Get pointer to next task of a task
gpOS_task_get_name	Get pointer to the name string of a task
gpOS_task_set_data	Set pointer to task custom private data
gpOS_task_get_data	Get pointer of task custom private data
gpOS_task_get_timeout	Get timeout of a task if it is waiting
gpOS_task_exit	Stop execution of current task
gpOS_task_set_onexit_function	Set function to be executed when a task stops
gpOS_task_wait	Wait for completion of a list of tasks

Function	Description
`gpOS_task_lock`	Lock current task execution
`gpOS_task_unlock`	Unlock current task execution

Table 11: Function defined in gpOS_task.h

7.14 Tasks functions definitions

7.14.1 gpOS_task_create

Note: this replaces task_create in previous version

Create a task on system partition.

Synopsis

```
#include <gpOS.h>

extern gpOS_task_t *gpOS_task_create
(
   gpOS_task_function_t  proc_func,
   gpOS_task_param_t proc_param,
   const tSize stack_size,
   const tInt priority,
   const tChar *name,
   gpOS_task_flags_t flags
);
```

Arguments

`gpOS_task_function_t` :	pointer to the task's entry point.
`gpOS_task_param_t`:	the parameter which will be passed into `proc_func`.
`const tSize`:	required stack size for the task, in bytes.
`const tInt`:	task's scheduling priority.
`const tChar *`:	the name of the task.
`gpOS_task_flags_t`:	start status of the task.

Results

`gpOS_task_t *`: pointer to the task structure if successful or `NULL` otherwise.

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

`gpOS_task_create` sets up a function as an OS20+ task and starts the task executing. `gpOS_task_create` returns a pointer to the task control block `gpOS_task_t`, which is subsequently used to refer to the task.

proc_func is a pointer to the function which is to be the entry point of the task. stack_size is the size of the stack space required in bytes. It is important that enough stack space is requested, if not, the results of running the task are unpredictable. gpOS_task_create will automatically call gpOS_memory_allocate in order to allocate the stack on the system memory partition.

proc_param is a pointer to the arguments to proc_func. If proc_func has a number of parameters, these should be combined into a structure and the address of the structure provided as the argument to gpOS_task_create. When the task is started it begins executing as if proc_func were called with the single argument proc_param.

The task's data structures will also be allocated by gpOS_task_create calling gpOS_memory_allocate. The task state (gpOS_task_t) will be allocated from the system memory partition.

priority is the task's scheduling priority.

name is the name of the task, so that the task can be correctly identified in the debugger's task list.

flags is used to give starting additional information about the task. Current available flags are:

```
typedef enum task_flags_e
{
  gpOS_TASK_FLAGS_ACTIVE     = 0,
  gpOS_TASK_FLAGS_SUSPENDED  = 1
} gpOS_task_flags_t;
```

If the gpOS_TASK_FLAGS_SUSPENDED is passed, the task will not be executed at all until its state is changed calling the gpOS_task_resume.

7.14.2 gpOS_task_create_p

Note: this replaces task_create_p in previous version

Create a task on custom user partition.

Synopsis

```
#include <gpOS.h>

extern gpOS_task_t *gpOS_task_create_p
(
   gpOS_partition_t* custom_part,
   gpOS_task_function_t  proc_func,
   gpOS_task_param_t proc_param,
   const tSize stack_size,
   const tInt priority,
   const tChar *name,
   gpOS_task_flags_t flags
);
```

Arguments

`gpOS_partition_t *`:	pointer to a custom partition.
`gpOS_task_function_t` :	pointer to the task's entry point.
`gpOS_task_param_t`:	the parameter which will be passed into `proc_func`.
`const tSize`:	required stack size for the task, in bytes.
`const tInt`:	task's scheduling priority.
`const tChar *`:	the name of the task.
`gpOS_task_flags_t`:	start status of the task.

Results

`gpOS_task_t *`:	pointer to the task structure if successful or `NULL` otherwise.

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

`gpOS_task_create_p` acts the same way as `gpOS_task_create`, but uses the custom user partition `custom_part` instead of system partition to allocate needed memory. Refer to 7.4 for all parameters explanation.

7.14.3 gpOS_task_delete

Note: this replaces task_delete in previous version

Delete a task.

Synopsis

```
#include <gpOS.h>

gpOS_error_t gpOS_task_delete( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`gpOS_error_t`:	`gpOS_SUCCESS` if task is correctly deleted, otherwise `gpOS_FAILURE`.

Errors

None.

Description

`gpOS_task_delete` tries to delete a task. It will try to remove task from scheduler and to free memory allocated for task handler and task stack.

Note: This API is effective only if used with care as explained in Section 6.5, otherwise it could break completely the execution of code.

7.14.4 gpOS_task_delay

Note: this replaces task_delay in previous version

Suspend task execution for a delay.

Synopsis:

```
#include <gpOS.h>

void gpOS_task_delay( gpOS_clock_t delay);
```

Arguments:

`gpOS_clock_t`:	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 section 11.

7.14.5 gpOS_task_delay_until

Note: this replaces task_delay_until in previous version

Suspend task execution up to a given timeout.

Synopsis:

```
#include <gpOS.h>

void gpOS_task_delay_until( gpOS_clock_t delay);
```

Arguments:

`gpOS_clock_t`:	The time up to 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 11.

7.14.6 gpOS_task_suspend

Note: this replaces task_suspend in previous version

Suspend task execution.

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_task_suspend( gpOS_task_t* task);
```

Arguments:

`gpOS_task_t *`:	Pointer to the task structure.

Results:

`gpOS_error_t`:	`gpOS_SUCCESS` if the task was successfully suspended, or `gpOS_FAILURE` if it could not be suspended.

Errors:

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

Description:

This function suspends the specified task. If `task` is `NULL` then this will suspend the current task. `gpOS_task_suspend` will stop the task from executing immediately, until it is resumed using `gpOS_task_resume`.

7.14.7 gpOS_task_resume

Note: this replaces task_resume in previous version

Resume task execution.

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_task_resume(gpOS_task_t* task);
```

Arguments:

`gpOS_task_t *`:	Pointer to the task structure.

Results:

`gpOS_error_t`:	`gpOS_SUCCESS` if the task was successfully resumed, or `gpOS_FAILURE` if it could not be resumed.

Errors:

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

Description:

This function resumes the specified task. The task must previously have been suspended, either by calling `gpOS_task_suspend`, or created by specifying a flag of `gpOS_TASK_FLAGS_SUSPENDED` to `gpOS_task_create`.

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

If the task was waiting for an event when it was suspended, then the event must also occur before the task will start executing. When a task is resumed it will start executing the next time it is the highest priority task, and so may pre-empt the task calling `gpOS_task_resume`.

7.14.8 gpOS_task_get_id

Note: this replaces task_get_id in previous version

Get pointer to the currently executing task.

Synopsis

```
#include <gpOS.h>

gpOS_task_t *gpOS_task_get_id( void);
```

Arguments

None.

Results

`gpOS_task_t *`:	Pointer to first currently executing task.

Errors

None.

Description

`gpOS_task_get_id` returns a pointer to the task structure of the currently executed task. It could be useful to inspect task data.

7.14.9 gpOS_task_set_priority

Note: this replaces task_set_priority in previous version

Change priority of a task.

Synopsis

```
#include <gpOS.h>

tInt gpOS_task_set_priority
(
   gpOS_task_t* task,
   const tInt new_priority
);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.
`const tInt`:	desired priority level for the task.

Results

`tInt`:	old priority of the task.

Errors

None.

Description

`gpOS_task_set_priority` 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.

7.14.10 gpOS_task_get_priority

Note: this replaces task_get_priority in previous version

Get priority of a task.

Synopsis

```
#include <gpOS.h>

tInt gpOS_task_get_priority( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`tInt`:	priority of the task.

Errors

None.

Description

`gpOS_task_priority` retrieves the priority of the task specified by `task` or the priority of the currently active task if `task` is `NULL`.

7.14.11 gpOS_task_get_stack_base

Note: this replaces task_get_stack_base in previous version

Get base pointer of task stack.

Synopsis

```
#include <gpOS.h>

void* gpOS_task_get_stack_base( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`void *`:	pointer to the task stack base.

Errors

None.

Description

`gpOS_task_get_stack_base` retrieves the pointer to the stack base of the task specified by `task` or the pointer to the stack base of the currently active task if `task` is `NULL`.

7.14.12 gpOS_task_get_stack_ptr

Note: this replaces task_get_stack_ptr in previous version

Get current pointer of task stack.

Synopsis

```
#include <gpOS.h>

void* gpOS_task_get_stack_ptr( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`void *`:	current stack pointer of the task.

Errors

None.

Description

`gpOS_task_get_stack_ptr` retrieves the current pointer of the stack of the task specified by `task` or the current pointer of the stack of the currently active task if `task` is `NULL`.

7.14.13 gpOS_task_get_stack_size

Note: this replaces task_get_stack_size in previous version

Get total size of task stack.

Synopsis

```
#include <gpOS.h>

tSize gpOS_task_get_stack_size( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`tSize`:	Total size of the stack of the task.

Errors

None.

Description

`gpOS_task_get_stack_size` retrieves the total size of the stack of the task specified by `task` or the total size of the stack of the currently active task if `task` is `NULL`.

7.14.14 gpOS_task_get_stack_used

Note: this replaces task_get_stack_used in previous version

Get used size of task stack.

Synopsis

```
#include <gpOS.h>

tSize gpOS_task_get_stack_used( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`tSize`:	Used size of the stack of the task.

Errors

None.

Description

`gpOS_task_get_stack_used` retrieves the used size of the stack of the task specified by `task` or the used size of the stack of the currently active task if `task` is `NULL`.

7.14.15 gpOS_task_get_cpuusage

Note: this replaces task_get_cpuusage in previous version

Get task CPU usage in latest second.

Synopsis

```
#include <gpOS.h>

tUInt gpOS_task_get_cpuusage( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`tUInt`:	CPU usage of given task.

Errors

None.

Description

`gpOS_task_get_cpuusage` retrieves the CPU usage of given task in latest second. The value returned is measured in OS20+ ticks.

7.14.16 gpOS_task_get_runtime

Note: this replaces task_get_runtime in previous version

Get number of ticks a task was in active state.

Synopsis

```
#include <gpOS.h>

tUInt gpOS_task_get_runtime( gpOS_task_t* task);
```

Arguments

`gpOS_task_t *`:	pointer to the task structure.

Results

`tUInt`:	Number of ticks the task was in active state.

Errors

None.

Description

`gpOS_task_get_runtime` retrieves the number of ticks a task was in active state. It can be used as a timestamp to evaluate how much time a portion of a task code needed to be executed.

7.14.17 gpOS_task_get_head

Note: this replaces task_get_head in previous version

Get pointer to the first created task.

Synopsis

```
#include <gpOS.h>

gpOS_task_t *gpOS_task_get_head( void);
```

Arguments

None.

Results

`gpOS_task_t *`:	Pointer to first created task.

Errors

None.

Description

`gpOS_task_get_head` returns a pointer to the task structure of the first created task. It could be useful to inspect all tasks data.

7.14.18 gpOS_task_get_next

Note: this replaces task_get_next in previous version

Get pointer to next task of a task.

Synopsis

```
#include <gpOS.h>

gpOS_task_t *gpOS_task_get_next( gpOS_task_t *task);
```

Arguments

`gpOS_task_t *`:	Pointer to a task.

Results

`gpOS_task_t *`:	Pointer to the next task of task `task`.

Errors

None.

Description

`gpOS_task_get_next` returns the pointer to the task structure of the next task of task `task`. If `NULL` is passed as argument, the pointer to the next task to current task is returned.

7.14.19 gpOS_task_get_name

Note: this replaces task_get_name in previous version

Get pointer to the name string of a task.

Synopsis

```
#include <gpOS.h>

const tChar * gpOS_task_get_name( gpOS_task_t *task);
```

Arguments

`gpOS_task_t *`:	Pointer to a task.

Results

`const tChar *`:	Pointer to the string name of the task `task`.

Errors

None.

Description

`gpOS_task_get_name` returns the pointer to string name of task `task`. If `NULL` is passed as argument, the pointer to the string name of calling task is returned.

7.14.20 gpOS_task_set_data

Note: this replaces task_set_data in previous version

Set pointer to task custom private data.

Synopsis

```
#include <gpOS.h>

void* gpOS_task_set_data( gpOS_task_t *task, void *data);
```

Arguments

`gpOS_task_t *`:	Pointer to a task.
`void *`:	Pointer to private data.

Results

`void *`:	Pointer to previous private data.

Errors

None.

Description

`gpOS_task_set_data` sets the task data pointer of task `task` to `data`. If `NULL` is passed as argument, it sets the task data pointer of calling task. The function returns the previous task data pointer value.

7.14.21 gpOS_task_get_data

Note: this replaces task_get_data in previous version

Get pointer to task custom private data.

Synopsis

```
#include <gpOS.h>

void* gpOS_task_get_data( gpOS_task_t *task);
```

Arguments

`gpOS_task_t *`:	Pointer to a task.

Results

`void *`:	Pointer to private data.

Errors

None.

Description

`gpOS_task_get_data` retrieves the task data pointer of task `task`. If `NULL` is passed as argument, it retrieves the task data pointer of calling task.

7.14.22 gpOS_task_get_timeout

Note: this replaces task_get_timeout in previous version

Get timeout of a task if it is waiting.

Synopsis

```
#include <gpOS.h>

gpOS_clock_t gpOS_task_get_timeout( gpOS_task_t *task);
```

Arguments

`gpOS_task_t *`:	Pointer to a task.

Results

`gpOS_clock_t`:	timeout of the task.

Errors

None.

Description

`gpOS_task_get_timeout` retrieves the timeout of task `task`. If `NULL` is passed as argument, it retrieves the timeout of calling task. This timeout represents the OS20+ time at which the task will be rescheduled again (if it is not waiting for any semaphore or message). If the task is not waiting, a zero is returned.

7.14.23 gpOS_task_exit

Note: this replaces task_exit in previous version

Stop execution of current task.

Synopsis:

```
#include <gpOS.h>

void gpOS_task_exit( gpOS_task_exit_status_t param);
```

Arguments:

`gpOS_task_exit_status_t`:	Exit status to pass to onexit handler.

Results:

None.

Errors:

None.

Description:

This forces the current task to terminate, after having called the onexit handler, if defined. It has the same effect as the task returning from its entry point function.

7.14.24 gpOS_task_set_onexit_function

Note: this replaces task_set_onexit_function in previous version

Sets the function to be called when `gpOS_task_exit` is called.

Synopsis:

```
#include <gpOS.h>

gpOS_task_onexit_function_t gpOS_task_set_onexit_function
(
   gpOS_task_onexit_function_t function,
);
```

Arguments:

`gpOS_task_onexit_function_t`: pointer to the procedure called at task exit.

Results:

`gpOS_task_onexit_function_t`: pointer to previous procedure called at task exit.

Errors:

None.

Description:

`gpOS_task_set_onexit_function` configures the procedure that will be called when a task must exit. If a procedure was already set, it will be returned as return value.

7.14.25 gpOS_task_wait

Note: *this replaces task_wait in previous version*

Wait for completion of a list of tasks.

Synopsis:

```
#include <gpOS.h>

tInt gpOS_task_wait
(
   gpOS_task_t **tasklist,
   tInt ntasks,
   const gpOS_clock_t *timeout
);
```

Arguments:

`gpOS_task_t **`:	pointer to a list of `gpOS_task_t` pointers.
`tInt`:	The number of tasks in tasklist.
`const gpOS_clock_t *`:	Maximum time to wait for tasks to terminate. Expressed in ticks or as `gpOS_TIMEOUT_IMMEDIATE` or `gpOS_TIMEOUT_INFINITY`.

Results:

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

Errors:

None.

Description:

`gpOS_task_wait` waits until one of the indicated tasks has terminated (by returning from its entry point function or calling `gpOS_task_exit`, or the timeout period has passed.

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

`timeout` is a pointer to the timeout value. If this time is reached then the function will return the value -1. The timeout value may be specified in ticks, which is an implementation dependent quantity, see section 11. Two special values can be specified for timeout: `gpOS_TIMEOUT_IMMEDIATE` indicates that the function should return immediately, even if no tasks have terminated, and `gpOS_TIMEOUT_INFINITY` indicates that the function should ignore the timeout period, and only return when a task terminates.

7.14.26 gpOS_task_lock

Note: this replaces task_lock in previous version

Lock current task execution.

Synopsis

```
#include <gpOS.h>

void gpOS_task_lock( void);
```

Arguments

None.

Results

None.

Errors

None.

Description

This API locks kernel scheduler avoiding any event to deschedule task currently ongoing for one at higher or same priority. It can be used when critical portions of code must run. For example, it can be used when accessing data shared with other tasks.

Note: This API does not stops interrupts handling, so if the critical code can be affected by the execution of any interrupt service routine, `gpOS_task_lock` should be substituted (or used in conjunction) with `gpOS_interrupt_lock`.

This API can be called more than once, but at least an exact number of calls to `gpOS_task_unlock` must be done to ensure that scheduler starts operating again.

7.14.27 gpOS_task_unlock

Note: this replaces task_unlock in previous version

Unlock current task execution.

Synopsis

```
#include <gpOS.h>

void gpOS_task_unlock( void);
```

Arguments

None.

Results

None.

Errors

None.

Description

This API tries to unlock kernel scheduler previously locked with one or more calls to `gpOS_task_lock`. To unlock completely the scheduler, it must be called at least an exact number of calls to `gpOS_task_lock`.

8 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.

8.1 Semaphores Overview

A semaphore structure `gpOS_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:

```
gpOS_semaphore_t* gpOS_semaphore_create
(
   gpOS_sem_type_t type,
   tInt value
);
gpOS_semaphore_t* gpOS_semaphore_create_p
(
   gpOS_sem_type_t type,
   gpOS_partition_t* partition, tInt value
);
```

The semaphores which OS20+ provides differ in the way in which tasks are queued. Normally tasks are queued in the order in which they call `gpOS_semaphore_wait`. This is termed a FIFO semaphore.

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 `gpOS_semaphore_wait` second.

Semaphores of both types are created using `gpOS_semaphore_create` or `gpOS_semaphore_create_p` and specifying the type of wanted semaphore in first argument, `gpOS_sem_type_t`, defined as:

```
typedef enum gpOS_sem_type_e
{
   SEM_FIFO,
   SEM_PRIO
} gpOS_sem_type_t;
```

Semaphores may be acquired by the following functions:

```
void gpOS_semaphore_wait( gpOS_semaphore_t* sem);
gpOS_error_t gpOS_semaphore_wait_timeout
(
   gpOS_semaphore_t* sem, const gpOS_clock_t *timeout
);
```

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

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

- `gpOS_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.

- `gpOS_TIMEOUT_INFINITY` causes the function to behave the same as `gpOS_semaphore_wait`, that is, the task waits indefinitely for the semaphore to become available.

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

```
void gpOS_semaphore_signal(gpOS_semaphore_t* sem);
```

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

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

8.2 Use of Semaphores

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

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

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

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

Semaphores are also used for synchronization. Usually this is between a task and an interrupt handler, with the task waiting for the interrupt handler. When used in this way the semaphore is initialized to zero. The task then performs a `gpOS_semaphore_wait` on the semaphore, and deschedules. Later the interrupt handler performs a `gpOS_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 `gpOS_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 `gpOS_semaphore_signal` to release it.

8.3 Semaphores APIs Summary

All the definitions related to semaphores can be accessed by including the header file `<gpOS.h>`, which itself includes the header file `gpOS_semaphore.h`.

Note: this replaces semaphore.h in previous version

Type	Description
`gpOS_semaphore_t`	Semaphore handler
`gpOS_sem_type_t`	Semaphore type

Table 12: Types defined in gpOS_semaphore.h

Function	Description
`gpOS_semaphore_create`	Create a queued semaphore
`gpOS_semaphore_create_p`	Create a queued semaphore
`gpOS_semaphore_delete`	Delete a semaphore
`gpOS_semaphore_signal`	Signal a semaphore
`gpOS_semaphore_wait`	Wait for a semaphore
`gpOS_semaphore_wait_timeout`	Wait for a semaphore or a timeout

Table 13: Functions defined in gpOS_semaphore.h

All semaphore functions are callable from an OS20+ task, however only `gpOS_semaphore_signal` and `gpOS_semaphore_wait_timeout` can be called from an interrupt service routine.

Note: *When using `gpOS_semaphore_wait_timeout` with in an interrupt service routine, the timeout value must be `gpOS_TIMEOUT_IMMEDIATE`.*

8.4 Semaphores functions Definitions

8.4.1 gpOS_semaphore_create

Note: this replaces semaphore_create_fifo and semaphore_create_priority in previous version

Create a semaphore of specific type.

Synopsis:

```
#include <gpOS.h>

gpOS_semaphore_t* gpOS_semaphore_create
(
   const gpOS_sem_type_t type,
   const tInt value
);
```

Arguments:

`const gpOS_sem_type_t`:	Type of the semaphore.
`const tInt`:	The initial value of the semaphore.

Returns:

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

Errors:

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

Description:

`gpOS_semaphore_create` creates a counting semaphore, initialized to value. The memory for the semaphore structure is allocated from the system memory partition.

If `type` is `SEM_FIFO`, semaphores created with this function have the usual semaphore semantics, except that when a task calls `gpOS_semaphore_wait` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

If `type` is `SEM_PRIO`, semaphores created with this function have the usual semaphore semantics, except that when a task calls `gpOS_semaphore_wait` it will be inserted into the queue of waiting tasks so that the list remains sorted by the task's priority, highest priority first. In this way when a task is removed from the front of the queue by `gpOS_semaphore_signal`, it is guaranteed to be the task with the highest priority of all those waiting for the semaphore.

8.4.2 gpOS_semaphore_create_p

Note: this replaces semaphore_create_fifo_p and semaphore_create_priority_p in previous version

Create a semaphore of specific type in user partition.

Synopsis:

```
#include <gpOS.h>

gpOS_semaphore_t* gpOS_semaphore_create_p
(
   const gpOS_sem_type_t type,
   gpOS_partition_t* partition,
   const tInt value
);
```

Arguments:

`const gpOS_sem_type_t`:	Type of the semaphore.
`gpOS_partition_t *`:	The partition in which to create the semaphore
`tInt`:	The initial value of the semaphore

Returns:

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

Errors

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

Description:

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

Note: If a `NULL` pointer is specified for partition, instead of a valid partition pointer, the system partition is used.

8.4.3 gpOS_semaphore_delete

Note: this replaces semaphore_delete in previous version

Delete a semaphore.

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_semaphore_delete(gpOS_semaphore_t* sem);
```

Arguments:

`gpOS_semaphore_t *`:	pointer to semaphore to delete.

Results:

`gpOS_error_t`:	Returns `gpOS_SUCCESS` if semaphore is successfully deleted, `gpOS_FAILURE` otherwise.

Errors:

None.

Description:

This function remove the semaphore from semaphores list and removes memory if possible.

Note: This API is effective only if used with care as explained in Section 6.5, otherwise it could break completely the execution of code.

8.4.4 gpOS_semaphore_signal

Note: this replaces semaphore_signal in previous version

Signal a semaphore.

Synopsis:

```
#include <gpOS.h>

void gpOS_semaphore_signal(gpOS_semaphore_t* sem);
```

Arguments:

`gpOS_semaphore_t *`:	a pointer to a semaphore.

Results:

None.

Errors:

None.

Description:

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

8.4.5 gpOS_semaphore_wait

Note: this replaces semaphore_wait in previous version

Wait on a semaphore.

Synopsis:

```
#include <gpOS.h>

void gpOS_semaphore_wait(gpOS_semaphore_t* sem);
```

Arguments:

`gpOS_semaphore_t *`:	A pointer to a semaphore.

Results:

None.

Errors:

None.

Description:

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

8.4.6 gpOS_semaphore_wait_timeout

Note: this replaces semaphore_wait_timeout in previous version

Wait on a semaphore up to a given timeout.

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_semaphore_wait_timeout
(
   gpOS_semaphore_t* sem
   const gpOS_clock_t *timeout
);
```

Arguments:

`gpOS_semaphore_t *`:	A pointer to a semaphore.
`const gpOS_clock_t *`:	Maximum time to wait for the semaphore. Expressed in ticks or as `gpOS_TIMEOUT_IMMEDIATE` or `gpOS_TIMEOUT_INFINITY`.

Results:

`gpOS_error_t`:	Returns `gpOS_SUCCESS` on success, `gpOS_FAILURE` 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 `gpOS_semaphore_wait_timeout` will return the value -1 indicating that a timeout occurred, and the semaphore count will be unchanged. If the semaphore is signalled before the timeout is reached, then `gpOS_semaphore_wait_timeout` will return 0. Note that timeout is an absolute not a relative value, so if a relative timeout is required this needs to be made explicit, as shown in the example.

The `timeout` value may be specified in ticks, which is an implementation dependent quantity. Two special time values may also be specified for timeout. `gpOS_TIMEOUT_IMMEDIATE` will cause the semaphore to be polled, that is, the function will always return immediately. If the semaphore count is greater than zero, then it will have been successfully decremented, and the function returns 0, otherwise the function will return a value of -1. A timeout of `gpOS_TIMEOUT_INFINITY` will behave exactly as `gpOS_semaphore_wait`.

Example:

```
gpOS_clock_t time;
time = gpOS_time_plus(gpOS_time_now(), 15625);
gpOS_semaphore_wait_timeout(semaphore, &time);
```

9 Mutexes

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

9.1 Mutexes overview

A mutex structure `gpOS_mutex_t` contains several pieces of data including:

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

Mutexes are created using one of the following functions:

```
gpOS_mutex_t* gpOS_mutex_create(gpOS_mutex_type_t type);
gpOS_mutex_t* gpOS_mutex_create_p
(
   gpOS_mutex_type_t type , gpOS_partition_t* partition
);
```

Currently, there is only a type of mutex allowed, as defined by `gpOS_mutex_type_t` type:

```
typedef enum gpOS_mutex_type_e
{
  MUTEX_FIFO
} gpOS_mutex_type_t;
```

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.

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 `gpOS_mutex_lock`. Mutexes of this type are created using `gpOS_mutex_create` or `gpOS_mutex_create_p`.

Mutex may be acquired by the functions:

```
void gpOS_mutex_lock(gpOS_mutex_t* mutex);
gpOS_error_t gpOS_mutex_trylock(gpOS_mutex_t* mutex);
```

When a task wants to acquire a mutex, it calls `gpOS_mutex_lock`. If the mutex is not currently owned, or already owned by the same task, then the task gets the mutex and continues. If however, the mutex is owned by another task, then the task adds itself to the queue of tasks waiting for the mutex and deschedules itself. Eventually another task should release the mutex, and the first waiting task gets the mutex and can continue. In this way, when the task returns from the function it has acquired the mutex.

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

To make certain that the task does not wait indefinitely for a mutex, use `gpOS_mutex_trylock`. This attempts to gain ownership of the mutex, but fails immediately if it is not available.

A task is automatically made immortal while it has ownership of a mutex. When a task wants to release the mutex, it calls `gpOS_mutex_release`:

```
gpOS_error_t gpOS_mutex_release(gpOS_mutex_t* mutex);
```

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

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

9.2 Use of mutexes

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

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

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

All mutex functions are callable from OS20+ tasks, and not from interrupt handlers.

9.3 Mutexes APIs summary

All the definitions related to mutexes are in the single header file gpOS.h, which itself includes the header file `gpOS_mutex.h`.

Note: *this replaces mutex.h in previous version*

Type	Description
`gpOS_mutex_t`	Mutex handler
`gpOS_mutex_type_t`	Mutex type

Table 14: Types defined in gpOS_mutex.h

Function	Description
`gpOS_mutex_create`	Create a FIFO queued mutex
`gpOS_mutex_create_p`	Create a FIFO queued mutex
`gpOS_mutex_delete`	Delete a FIFO queued mutex
`gpOS_mutex_lock`	Acquire a mutex, block if not available
`gpOS_mutex_release`	Release a mutex
`gpOS_mutex_trylock`	Try to get a mutex, fail if not available
`gpOS_mutex_locked`	Query for locking status of a mutex

Table 15: Functions defined in gpOS_mutex.h

9.4 Mutexes functions definitions

9.4.1 gpOS_mutex_create

Note: this replaces mutex_create_fifo in previous version

Create a FIFO queued mutex

Synopsis:

```
#include <gpOS.h>

gpOS_mutex_t* gpOS_mutex_create_fifo(gpOS_mutex_type_t type);
```

Arguments:

`gpOS_mutex_type_t`:	Mutex type.

Returns:

`gpOS_mutex_t *`:	the pointer to the handler of an initialized mutex, or `NULL` if an error occurs.

Errors:

`NULL` if there is insufficient memory for the mutex.

Description:

`gpOS_mutex_create` creates a mutex. The memory for the mutex structure is allocated from the system partition. Mutexes created with this function have the usual mutex semantics, except that when a task calls `gpOS_mutex_lock` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

9.4.2 gpOS_mutex_create_p

Note: this replaces mutex_create_fifo_p in previous version

Create a FIFO queued mutex

Synopsis:

```
#include <gpOS.h>

gpOS_mutex_t* gpOS_mutex_create_p
(
   gpOS_mutex_type_t type,
   gpOS_partition_t* partition
);
```

Arguments:

`gpOS_mutex_type_t`:	Mutex type.
`gpOS_partition_t *`:	The partition in which to create the mutex

Returns:

`gpOS_mutex_t *`:	the pointer to the handler of an initialized mutex, or `NULL` if an error occurs.

Errors:

`NULL` if there is insufficient memory for the mutex.

Description:

`gpOS_mutex_create_p` creates a mutex, allocated from the given partition. Mutexes created with this function have the usual mutex semantics, except that when a task calls `gpOS_mutex_lock` it is always appended to the end of the queue of waiting tasks, irrespective of its priority.

Note: If a `NULL` pointer is specified for partition, instead of a valid partition pointer, the system partition is used.

9.4.3 gpOS_mutex_delete

Note: *this replaces mutex_delete in previous version*

Delete a mutex.

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_mutex_delete( gpOS_mutex_t* mutex);
```

Arguments:

`gpOS_mutex_t *`:	mutex to delete

Results:

`gpOS_error_t`:	Returns `gpOS_SUCCESS` if mutex is successfully deleted, `gpOS_FAILURE` otherwise.

Errors:

None.

Description:

This function remove the mutex from mutexes list and removes memory if possible.

Note: *This API is effective only if used with care as explained in Section 6.5, otherwise it could break completely the execution of code.*

9.4.4 gpOS_mutex_lock

Note: this replaces mutex_lock in previous version

Acquire a mutex, block if not available

Synopsis:

```
#include <gpOS.h>

void gpOS_mutex_lock( gpOS_mutex_t* mutex);
```

Arguments:

`gpOS_mutex_t *`:	mutex to lock

Returns:

None

Errors:

None

Description:

`gpOS_mutex_lock` acquires the given mutex. The exact behaviour 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.

9.4.5 gpOS_mutex_release

Note: this replaces mutex_release in previous version

Release a mutex

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_mutex_release( gpOS_mutex_t* mutex);
```

Arguments:

`gpOS_mutex_t *`:	mutex to release

Returns:

`gpOS_error_t`:	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the task releasing the mutex does not own it.

Description:

`gpOS_mutex_release` releases the specified mutex. The exact behaviour 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 pre-empting 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.

9.4.6 gpOS_mutex_trylock

Note: *this replaces mutex_trylock in previous version*

Acquire a mutex, return immediately if not available

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_mutex_trylock( gpOS_mutex_t* mutex);
```

Arguments:

`gpOS_mutex_t *`:	mutex to lock

Returns:

`gpOS_error_t`:	`gpOS_SUCCESS` if the mutex has been locked, otherwise `gpOS_FAILURE`

Errors:

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

Description:

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

9.4.7 gpOS_mutex_locked

Note: this replaces mutex_locked in previous version

Check if current task is locking the mutex

Synopsis:

```
#include <gpOS.h>

unsigned tUInt gpOS_mutex_trylock( gpOS_mutex_t* mutex);
```

Arguments:

`gpOS_mutex_t *`:	mutex to check

Returns:

`tUInt`:	`1` if current task is locking the mutex, `0` elsewhere.

Errors:

None.

Description:

`gpOS_mutex_locked` checks if the mutex is locked by current task. If so, it returns `1`, else it returns `0`.

10 Message Handling

A message queue provides a buffered communication method for tasks. Message queues also provide a way to communicate without copying the data, which can save time.

Note: *Message queues are subject to a restriction when used from interrupt handlers. For interrupt handlers, use the timeout versions of the message handling functions with a timeout period of* `gpOS_TIMEOUT_IMMEDIATE` *(see 11). This prevents the interrupt handler from blocking on a message claim*

10.1 Message Queues

An OS20+ message queue implements two queues of messages, one for message buffers which are currently not being used (known as the free queue), and the other holds messages which have been sent but not yet received (known as the send queue). Message buffers rotate between these queues, as a result of the user calling the various message functions.

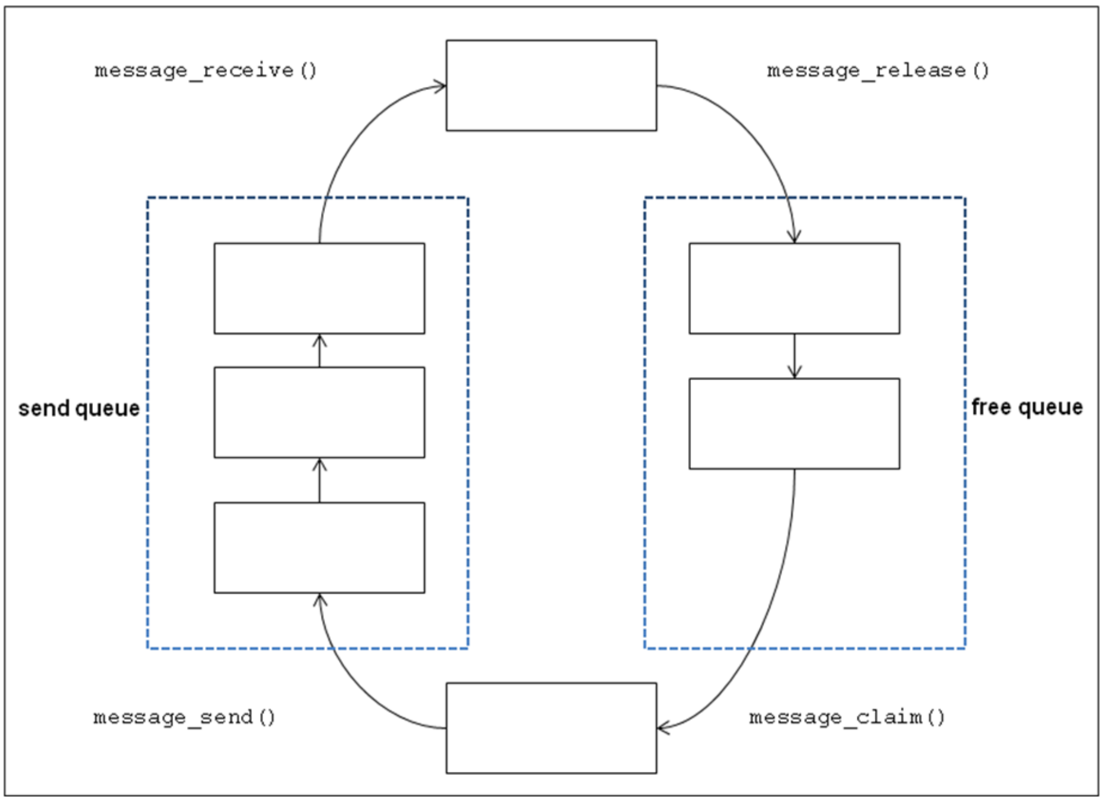

Figure 1: Message Queues

10.2 Creating Message Queues

Message queues are created using one of the following functions:

```
gpOS_message_queue_t* gpOS_message_create_queue
(
   tSize max_message_size,
   tUInt max_messages
);

gpOS_message_queue_t* gpOS_message_create_queue_p
(
   gpOS_partition_t* partition,
   tSize max_message_size,
   tUInt max_messages
);
```

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

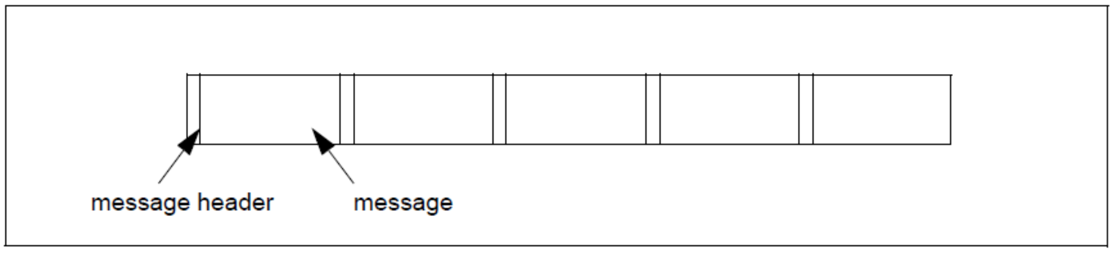

Figure 2: OS20+ Message Elements

`gpOS_message_create_queue` allocates the memory for the queue automatically from the system partition. `gpOS_message_create_queue_p` allows the user to specify which partition to allocate the control structures and message buffers from.

10.3 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* gpOS_message_claim( gpOS_message_queue_t* queue);
void* gpOS_message_claim_timeout
(
   gpOS_message_queue_t* queue, const gpOS_clock_t* time
);
```

Both functions claim the next available message in the message queue. `gpOS_message_claim_timeout` enables a timeout to be specified. If the timeout is reached before a message buffer is acquired then the function returns NULL. Two special values may be specified for the timeout period:

- gpOS_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.

- gpOS_TIMEOUT_INFINITY causes the function to behave the same as gpOS_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 gpOS_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* gpOS_message_receive( gpOS_message_queue_t* queue);
void* gpOS_message_receive_timeout
(
   gpOS_message_queue_t* queue, const gpOS_clock_t* time
);
```

Both functions return the next available message. gpOS_message_receive_timeout provides a timeout facility which behaves in a similar manner to gpOS_message_claim_timeout in that it returns NULL if the message does not become available. If gpOS_TIMEOUT_IMMEDIATE is specified the task continues whether or not a message is received and if gpOS_TIMEOUT_INFINITY is specified the function behaves as gpOS_message_receive and waits indefinitely.

Finally when the receiving task has finished with the message buffer it should free it by calling gpOS_message_release. This function adds it to the free queue, where it is again available for allocation.

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

10.4 Message handling APIs summary

All the definitions related to messages can be accessed by including the header file gpOS.h, which itself includes the header file gpOS_message.h.

Note: *this replaces message.h in previous version*

Types	Description
gpOS_message_queue_t	Message queue handler

Table 16: Types defined in gpOS_message.h

Function	Description
`gpOS_message_create_queue`	Create a fixed size message queue
`gpOS_message_create_queue_p`	Create a fixed size message queue
`gpOS_message_delete_queue`	Delete a message queue
`gpOS_message_claim`	Claim a message buffer
`gpOS_message_claim_timeout`	Claim a message buffer with timeout
`gpOS_message_receive`	Receive the next available message from a queue
`gpOS_message_receive_timeout`	Receive the next available message from a queue or timeout
`gpOS_message_release`	Release a message buffer
`gpOS_message_send`	Send a message to a queue

Table 17: Functions defined in gpOS_message.h

10.5 Messages functions definition

10.5.1 gpOS_message_create_queue

Note: this replaces message_create_queue in previous version

Create a fixed size message queue.

Synopsis:

```
#include <gpOS.h>

gpOS_message_queue_t* gpOS_message_create_queue
(
   tSize MaxMessageSize,
   tUInt MaxMessages
);
```

Arguments:

`tSize`:	The maximum size of a message, in bytes.
`tUInt`:	The maximum number of messages.

Returns:

`gpOS_message_queue_t *`:	message queue handler.

Results:

The message queue identifier, or `NULL` on failure.

Errors:

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

Description:

`gpOS_message_create_queue` creates a message queue with buffering for a fixed number of fixed size messages. Buffer space for the messages and the `gpOS_message_queue_t` structure, is created automatically by the function from the system partition.

10.5.2 gpOS_message_create_queue_p

Note: this replaces message_create_queue_p in previous version

Create a fixed size message queue.

Synopsis:

```
#include <gpOS.h>

gpOS_message_queue_t* gpOS_message_create_queue
(
   gpOS_partition_t *custom_part,
   tSize MaxMessageSize,
   tUInt MaxMessages
);
```

Arguments:

`gpOS_partition_t *`:	The partition in which to create the message
`tSize`:	The maximum size of a message, in bytes.
`tUInt`:	The maximum number of messages.

Returns:

`gpOS_message_queue_t *`:	message queue handler.

Results:

The message queue identifier, or `NULL` on failure.

Errors:

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

Description:

`gpOS_message_create_queue_p` creates a message queue with buffering for a fixed number of fixed size messages. Buffer space for the messages and the `gpOS_message_queue_t` structure, is created automatically by the function calling `gpOS_memory_allocate_p` on the specified memory partition.

Note: If a `NULL` pointer is specified for partition, instead of a valid partition pointer, the system partition is used.

10.5.3 gpOS_message_delete_queue

Note: this replaces message_delete_queue in previous version

Delete a message queue.

Synopsis:

```
#include <gpOS.h>

gpOS_error_t gpOS_message_delete_queue(gpOS_message_queue_t* queue);
```

Arguments:

`gpOS_message_queue_t *`:	message to delete.

Returns:

`gpOS_error_t`:	Returns `gpOS_SUCCESS` if queue is successfully deleted, `gpOS_FAILURE` otherwise.

Errors:

None.

Description:

This function remove the message from messages list and removes memory if possible.

Note: This API is effective only if used with care as explained in Section 6.5, otherwise it could break completely the execution of code.

10.5.4 gpOS_message_claim

Note: this replaces message_claim in previous version

Claim a message buffer

Synopsis:

```
#include <gpOS.h>

void* gpOS_message_claim( gpOS_message_queue_t* queue);
```

Arguments:

`gpOS_message_queue_t *`:	queue

Returns:

`void *`:	the next available message buffer.

Errors:

None

Description:

`gpOS_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 is blocked until one becomes available by a call to `gpOS_message_release`.

Note: This function is not callable from an interrupt handler.

10.5.5 gpOS_message_claim_timeout

Note: this replaces message_claim_timeout in previous version

Claim a message buffer or timeout

Synopsis:

```
#include <gpOS.h>

void* gpOS_message_claim_timeout
(
   gpOS_message_queue_t* queue
   const gpOS_clock_t* time
);
```

Arguments:

`gpOS_message_queue_t *`:	message queue handler.

Returns:

`void *`:	The next available message buffer, or `NULL` if a timeout occurs.
`const gpOS_clock_t *`:	The maximum time to wait for a message.

Errors:

None.

Description:

`gpOS_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 a call to `gpOS_message_release`, or the time specified by time is reached.

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

`time` is specified in ticks. A tick is an implementation dependent quantity. Two special time values may also be specified for time. `gpOS_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 result of `NULL`. A timeout of `gpOS_TIMEOUT_INFINITY` behaves exactly as `gpOS_message_claim`.

`gpOS_message_claim_timeout` can be used from an interrupt handler, as long as time is `gpOS_TIMEOUT_IMMEDIATE`.

Example:

```
gpOS_clock_t time;
time = gpOS_time_plus(gpOS_time_now(), time_ticks_per_sec());
gpOS_message_claim_timeout (message_queue, &time);
```

10.5.6 gpOS_message_receive

Note: this replaces message_receive in previous version

Receive the next available message from a queue.

Synopsis:

```
#include <gpOS.h>

void* gpOS_message_receive(gpOS_message_queue_t* queue);
```

Arguments:

`gpOS_message_queue_t *`:	The message queue that delivers the message.

Results:

`void *`:	The next available message from the queue.

Errors:

None.

Description:

`gpOS_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 a call to `gpOS_message_send`.

10.5.7 gpOS_message_receive_timeout

Note: this replaces message_receive_timeout in previous version

Receive the next available message from a queue or timeout.

Synopsis:

```
#include <gpOS.h>

void* gpOS_message_receive_timeout
(
   gpOS_message_queue_t* queue
   const gpOS_clock_t* time
);
```

Arguments:

`gpOS_message_queue_t *`:	The message queue that delivers the message.
`const gpOS_clock_t *`:	The maximum time to wait for a message.

Results:

`void *`:	The next available message from the queue, or NULL if a timeout occurs.

Errors:

None.

Description:

`gpOS_message_receive_timeout` receives the next available message from the message queue, and returns its address. If no messages are currently available then the task will block until one becomes available (by another task calling `gpOS_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 for time. `gpOS_TIMEOUT_IMMEDIATE` will cause the message queue to be polled, that is, the function will always return immediately. If a message was available then it will be returned, otherwise the function will return immediately with a result of `NULL`. A timeout of `gpOS_TIMEOUT_INFINITY` will behave exactly as `gpOS_message_receive`.

Example:

```
gpOS_clock_t time;
time = gpOS_time_plus(gpOS_time_now(), 15625);
gpOS_message_receive_timeout(message_queue, &time);
```

10.5.8 gpOS_message_release

Note: this replaces message_release in previous version

Release a message buffer.

Synopsis:

```
#include <gpOS.h>

void gpOS_message_release
(
   gpOS_message_queue_t* queue,
   void* message
);
```

Arguments:

`gpOS_message_queue_t *`:	The message queue to which the message is released.
`void *`:	The message buffer.

Results:

None.

Errors:

None.

Description:

`gpOS_message_release` returns a message buffer to the message queue's free list. This function should be called when a message buffer (received by `gpOS_message_receive`) is no longer required. If a task is waiting for a free message buffer (by calling `gpOS_message_claim`) this will cause the task to be restarted and the message buffer returned.

10.5.9 gpOS_message_send

Note: this replaces message_send in previous version

Send a message to a queue.

Synopsis:

```
#include <gpOS.h>

void gpOS_message_send(gpOS_message_queue_t* queue, void* message);
```

Arguments:

`gpOS_message_queue_t *`:	The message queue to which the message is sent.
`void *`:	The message to send.

Results:

None.

Errors:

None.

Description:

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

11 Real-time clocks

Time is a very important issue for real-time systems. OS20+ provides some basic functions for manipulating quantities of time.

OS20+ regarded time as circular. That is, the counters which represent time could wrap round, with half the time period being in the future, and half of it in the past. This behavior meant that clock values has to be interpreted with care, and manipulated using time functions which took account of wrapping. These functions were used to:

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

Time is represented in clock ticks, with the `gpOS_clock_t` type. This is defined to be a signed 32-bit integer.

11.1 Reading the current time

The value of system time is read using `gpOS_time_now`.

```
gpOS_clock_t gpOS_time_now(void);
```

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

11.2 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.

```
gpOS_clock_t gpOS_time_plus
(
   const gpOS_clock_t time1,
   const gpOS_clock_t time2
);
gpOS_clock_t gpOS_time_minus
(
   const gpOS_clock_t time1,
   const gpOS_clock_t time2
);
int gpOS_time_after
(
   const gpOS_clock_t time1,
   const gpOS_clock_t time2
);
```

`gpOS_time_plus` adds two timer values together and returns the sum. For example, if a certain number of ticks is added to the current time using `gpOS_time_plus` then the result is the time after that many ticks.

`gpOS_time_minus` subtracts the second value from the first and returns the difference. For example, if one time is subtracted from another using `gpOS_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.

`gpOS_time_after` determines whether the first time is after the second time. The first time is considered to be after the second time if the result of subtracting the second time from the first time is positive. The function returns the integer value 1 if the first time is after the second, otherwise it returns 0.

Some of these concepts are shown in Figure 3.

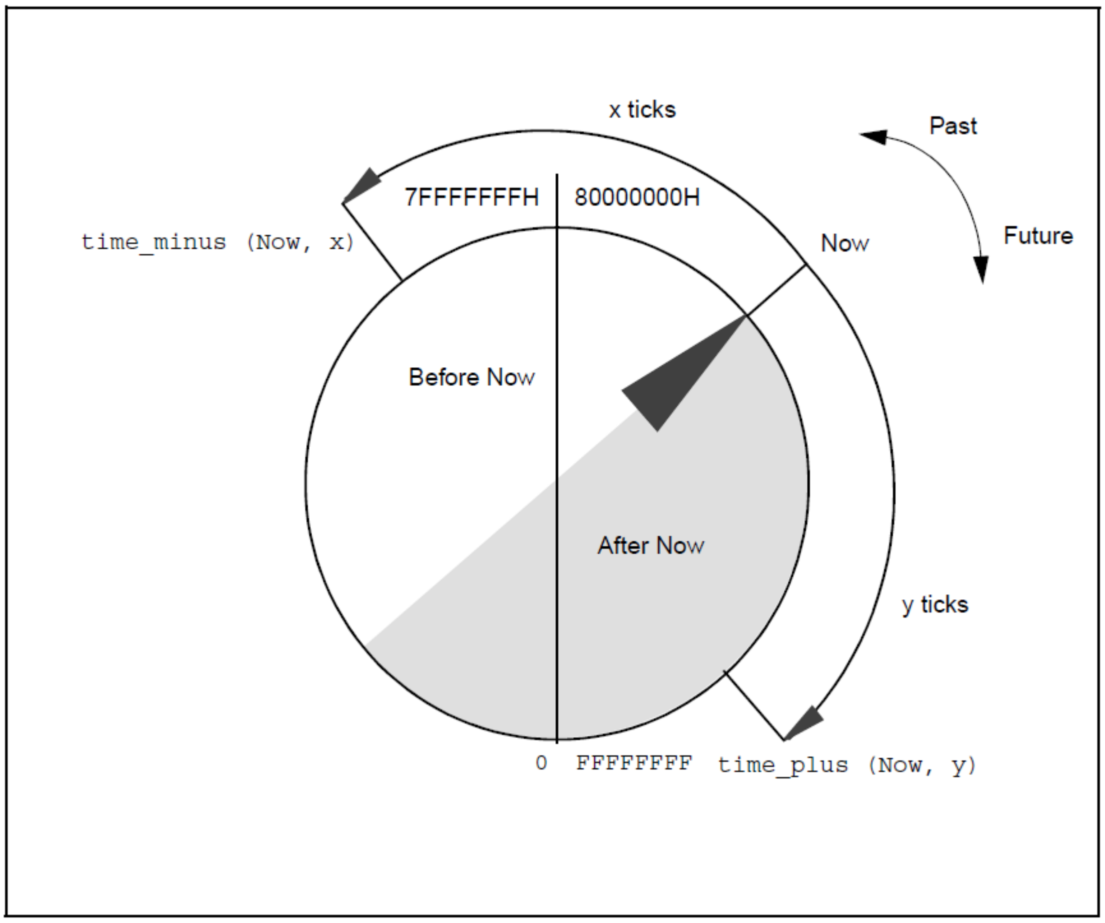

Figure 3: Time arithmetic

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

11.3 Chip variants

The time features of OS20+ depends on hardware implementation of a counter fed by a clock. OS20+ currently provides an abstraction of this. This abstraction is provided in OS20+ using the board support package (BSP) mechanism. The BSP is a library containing target specifics, which is linked with the application and the OS20+ kernel at final link time. OS20+ is shipped with BSP libraries for all supported variants.

At startup, OS20+ must be configured to know which is the period of the clock feeding the timers implemented by the BSP. This can be done through:

```
void gpOS_timer_set_clock( const gpOS_clock_t period);
```

Then, the following APIs could be used to know the reference unit for microsecond, millisecond and second:

```
unsigned gpOS_timer_ticks_per_usec( void);
unsigned gpOS_timer_ticks_per_msec( void);
unsigned gpOS_timer_ticks_per_second( void);
```

11.4 Time APIs summary

All the definitions related to time can be accessed by including the header file `gpOS.h`, which itself includes the header file `gpOS_time.h`.

Note: this replaces ostime.h in previous version

Type	Description
`gpOS_clock_t`	Processor clock ticks type

Table 18: Types defined in gpOS_time.h

Function	Description
`gpOS_time_after()`	Return whether one time is after another
`gpOS_time_minus()`	Subtract two clock values
`gpOS_time_plus()`	Add two clock values
`gpOS_time_now()`	Return the current time
`gpOS_timer_ticks_per_usec()`	Return the number of ticks per microsecond
`gpOS_timer_ticks_per_msec()`	Return the number of ticks per millisecond
`gpOS_timer_ticks_per_sec()`	Return the number of ticks per second
`gpOS_timer_set_clock()`	Set number of ticks in a second (hw clock period)

Table 19: Functions defined in gpOS_time.h

11.5 Timer functions definitions

11.5.1 gpOS_time_after

Note: this replace the time_after in previous version

Return whether one time is after another.

Synopsis:

```
#include <gpOS.h>

tInt gpOS_time_after
(
   const gpOS_clock_t time1,
   const gpOS_clock_t time2
);
```

Arguments:

`const gpOS_clock_t`:	A ticks value.
`const gpOS_clock_t`:	A ticks value.

Results:

`tInt`:	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.

11.5.2 gpOS_time_minus

Note: this replaces time_minus in previous version

Subtract two clock values.

Synopsis:

```
#include <gpOS.h>

gpOS_clock_t gpOS_time_minus
(
   const gpOS_clock_t time1,
   const gpOS_clock_t time2
);
```

Arguments:

`const gpOS_clock_t`:	A ticks value.
`const gpOS_clock_t`:	A ticks value.

Results:

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

Errors:

None.

Description:

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

11.5.3 gpOS_time_plus

Note: this replaces time_plus in previous version

Add two clock values.

Synopsis:

```
#include <gpOS.h>

gpOS_clock_t gpOS_time_plus
(
   const gpOS_clock_t time1,
   const gpOS_clock_t time2
);
```

Arguments:

`const gpOS_clock_t`:	A ticks value.
`const gpOS_clock_t`:	A ticks value.

Results:

`gpOS_clock_t`:	Returns the result of adding `time2` to `time1`.

Errors:

None.

Description:

It adds one clock value to another using modulo arithmetic.

11.5.4 gpOS_time_now

Note: this replaces time_now in previous version

Return the current time.

Synopsis:

```
#include <gpOS.h>

gpOS_clock_t gpOS_time_now( void);
```

Arguments:

None.

Results:

`gpOS_clock_t`:	Returns the number of ticks since the system started.

Errors:

None.

Description:

`gpOS_time_now` returns the number of ticks since the system started running. The exact time at which counting starts is implementation specific, but will be no later than the call to `gpOS_kernel_start`. The units of ticks is an implementation dependent quantity, use the functions `gpOS_timer_ticks_per_second`, `gpOS_timer_ticks_per_msec` and `gpOS_timer_ticks_per_usec` to turn ticks into time quantities.

11.5.5 gpOS_timer_ticks_per_second

Note: this replaces timer_ticks_per_second in previous version

Return number of ticks in a second.

Synopsis:

```
#include <gpOS.h>

gpOS_clock_t gpOS_ticks_per_second();
```

Arguments:

None

Results:

`gpOS_clock_t`:	The number of ticks.

Errors:

None.

Description:

Returns the number of ticks in a second.

11.5.6 gpOS_timer_ticks_per_msec

Note: this replaces timer_ticks_per_msec in previous version

Return number of ticks in a millisecond.

Synopsis:

```
#include <gpOS.h>

gpOS_clock_t gpOS_ticks_per_msec();
```

Arguments:

None

Results:

`gpOS_clock_t`:	The number of ticks.

Errors:

None.

Description:

Returns the number of ticks in a millisecond.

11.5.7 gpOS_timer_ticks_per_usec

Note: this replaces timer_ticks_per_usec in previous version

Return number of ticks in a microsecond.

Synopsis:

```
#include <gpOS.h>

gpOS_clock_t gpOS_ticks_per_usec();
```

Arguments:

None

Results:

`gpOS_clock_t`:	The number of ticks.

Errors:

None.

Description:

Returns the number of ticks in a microsecond.

11.5.8 gpOS_timer_set_clock

Note: this replaces timer_set_clock in previous version

Sets the period of the hardware clock.

Synopsis:

```
#include <gpOS.h>

void gpOS_timer_set_clock( const gpOS_clock_t clock);
```

Arguments:

`const gpOS_clock_t`:	Period of hardware clock.

Results:

None.

Errors:

None.

Description:

`gpOS_timer_set_clock` sets the period of the hardware clock. This must be called after OS20+ initialization to let the functions `gpOS_timer_ticks_per_second`, `gpOS_timer_ticks_per_msec` and `gpOS_timer_ticks_per_usec` to provide the proper values when called.

12 Interrupts

Interrupts provide a mechanism for external events to control the CPU. Normally, as soon as an interrupt is asserted, the CPU stops executing the current task, and starts executing the interrupt handler for that interrupt. In this way the program is made aware of external changes as soon as they occur. This switch is performed completely in hardware, and so is extremely rapid. Similarly when the interrupt handler has completed, the CPU resumes execution of the interrupted task, which is unaware that it has been interrupted.

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

OS20+ supports **interrupt nesting**. This means that if priority are available on a chip variant, and an interrupt `x` with higher priority occurs while an interrupt `y` is being serviced, then interrupt `y` service routine is stopped, interrupt `x` service routine is run, and at its end the interrupt `y` service routine continues its execution from where it was stopped.

12.1 Chip variants

Each version of the CPU can have its own set of peripherals, and these peripherals are allocated interrupts. There is no guarantee that the assignments will remain the same from variant to variant. To accommodate this, OS20+ requires a table of definitions which describes the interrupt mappings for a given part. This table is provided to the OS20+ kernel using the **Board Support Package** (BSP) mechanism. The BSP is a library containing target specifics, which is linked with the application and the OS20+ kernel at final link time. OS20+ is shipped with BSP libraries for all supported variants.

Providing the source for the board support packages enables users to limit the number of declared interrupts to just those used by the application, and so save memory, if necessary. OS20+ uses the interrupt description table from the BSP to build its own table which is used to dispatch interrupts from the first level interrupt handler.

Along with the target specific OS20+ interrupt code, the BSP describes the interrupt system to OS20+ and together they implement the generic interrupt API.

12.2 Initializing the interrupt handling subsystem

Before interrupts can be used, the OS20+ interrupt subsystem must be initialized. This is done through the BSP by calling the API:

```
void gpOS_interrupt_init
(
   gpOS_partition_t *part, const tSize stack_size
);
```

This API will initialize the interrupt mechanism of OS20+ on top of the hardware driven by the BSP, and configures the stack size that will be used to run the user services routines in system mode. `stack_size` must be properly sized, otherwise

12.3 Attaching an interrupt handler

An interrupt handler is attached to an interrupt, using the `gpOS_interrupt_install` function:

```
void gpOS_interrupt_install
(
   gpOS_interrupt_line_t line,
   gpOS_interrupt_priority_t priority,
   gpOS_interrupt_callback_t callback_func,
   void *callback_param
);
```

The `line` corresponds to the hardware interrupt controller vector. The available lines are specified in the specific BSP for each chip variant. The `priority` of the interrupt handler can be specified if the interrupt controller hardware supports it. If no specific priority is requested, the value `gpOS_INTERRUPT_NOPRIORITY` can be used. The `callback_func` is the function that is called and the `callback_param` is the value passed to the handler when it is invoked by the first level interrupt handler. Once the interrupt handler is attached the interrupt should be enabled by calling `gpOS_interrupt_enable`.

Priority levels are interrupt controller dependent. Available priority levels are available for all chip variants in related BSP.

12.4 Enabling and disabling interrupts

The following two functions are used to enable and disable specific interrupts.

```
int gpOS_interrupt_enable( gpOS_interrupt_line_t line);
int gpOS_interrupt_disable( gpOS_interrupt_line_t line);
```

12.5 Detaching an interrupt handler

When an interrupt must not be served anymore by a specific handler, this can be removed using the API:

```
void gpOS_interrupt_uninstall( gpOS_interrupt_line_t line);
```

This will disable the line and will remove the handler from OS20.

12.6 Locking out interrupts

All interrupts to the CPU can be globally disabled or re-enabled using the following two commands:

```
void gpOS_interrupt_lock(void);
void gpOS_interrupt_unlock(void);
```

These functions should always be called as a pair and will prevent any interrupts from the interrupt controller having any effect on the currently executing task while the lock is in place. These functions can be used to create a critical region in which the task cannot be pre-empted by any other task or interrupt. Calls to `gpOS_interrupt_lock` can be nested, and the lock will not be released until an equal number of calls to `gpOS_interrupt_unlock` have been made.

Note: *Locking out interrupts is slightly different from disabling an interrupt. Interrupts are locked by causing the CPU to ignore the interrupt controller, while disabling an interrupt modifies the interrupt controller's Mask register, and so can be used much more selectively.*

Note: *A task must not deschedule with interrupts locked, as this can cause the scheduler to fail.*

12.7 Contexts and interrupt handler code

Code running under OS20+ may run in one of two environments (or contexts). These are called task context and system context. OS20+ interrupt handlers are run from system context.

The main difference between system context and task context is that code running in system context is not allowed to block. Undefined behaviour occurs if code running in system context blocks. As a result of this constraint, code running from system context should never call an OS20+ function that may block. Please refer to the individual function descriptions for details of which contexts the OS20+ functions may be run from.

12.8 Interrupt hooks

When an interrupt is generated, OS20+ executes a sequence of operation before serving any installed interrupt service routine. There could be the need to execute a piece of code every time an interrupt occurs, at the beginning and/or at the end. This can be set in OS20+ through dedicated API, `gpOS_interrupt_set_hooks`. It accepts two pointers to functions of type `gpOS_interrupt_hook_t`: at each interrupt, the first function will be used before any installed ISR is executed, while the second function will be used after any installed ISR is executed.

Note: *It is not necessary to specify a function for both.*

12.9 Interrupt APIs summary

All the definitions related to interrupts can be obtained by including the header file `gpOS.h`, which itself includes the header file `gpOS_interrupt.h`.

Note: *this replaces interrupt.h in previous version*

Types	Description
`gpOS_interrupt_line_t`	Interrupt line
`gpOS_interrupt_priority_t`	Interrupt Priority
`gpOS_interrupt_callback_t`	Handler function prototype
`gpOS_interrupt_hook_t`	Hook function type

Table 20: Types defined in gpOS_interrupt.h

Function	Description
`gpOS_interrupt_init`	Initialize interrupt handling mechanism in OS20
`gpOS_interrupt_set_hooks`	Set hooks called when entering and exiting interrupts
`gpOS_interrupt_install`	Install an interrupt handler
`gpOS_interrupt_uninstall`	Uninstall an interrupt handler
`gpOS_interrupt_lock`	Disable all interrupts
`gpOS_interrupt_unlock`	Enable all interrupts
`gpOS_interrupt_enable`	Enable an interrupt
`gpOS_interrupt_disable`	Disable an interrupt

Table 21: Functions defined in gpOS_interrupt.h

12.10 Interrupt function definitions

12.10.1 gpOS_interrupt_init

Note: this replaces interrupt_init in previous version

Initialize interrupt handling mechanism in OS20+

Synopsis:

```
#include <gpOS.h>

void gpOS_interrupt_init
(
   gpOS_partition_t *part,
   const tSize stack_size
);
```

Arguments:

`gpOS_partition_t *`:	Partition to use to allocate stacks.
`const tSize`:	Size of stack used for interrupt service routines execution

Returns:

None.

Errors:

None.

Description:

Initialize the interrupt mechanism in OS20+ allocating the space `stack_size` on specified partition to be used as the stack that will be used by all user service routines.

12.10.2 gpOS_interrupt_set_hooks

Note: this replaces interrupt_set_hooks in previous version

Configure hooks called when interrupt handler enters and exits

Synopsis:

```
#include <gpOS.h>

void gpOS_interrupt_set_hooks
(
   gpOS_interrupt_hook_t enter_hook,
   gpOS_interrupt_hook_t exit_hook
);
```

Arguments:

`gpOS_interrupt_hook_t`:	Procedure to execute when interrupt handler starts execution.
`gpOS_interrupt_hook_t`:	Procedure to execute when interrupt handler exits execution.

Returns:

None.

Errors:

None.

Description:

This setups hooks that will be called during interrupt handler execution when it is entered and exited. Writing a `NULL` in a hook resets it.

12.10.3 gpOS_interrupt_install

Note: this replaces interrupt_install in previous version

Install an interrupt handler

Synopsis:

```
#include <gpOS.h>

void gpOS_interrupt_install
(
   gpOS_interrupt_line_t line,
   gpOS_interrupt_priority_t priority,
   gpOS_interrupt_callback_t callback_func,
   void *callback_param
);
```

Arguments:

`gpOS_interrupt_line_t:`	line to interrupt controller
`gpOS_interrupt_priority_t:`	priority of interrupt
`gpOS_interrupt_callback_t:`	procedure to be executed when the interrupt occurs
`void *:`	pointer to pass to callback

Returns:

None.

Errors:

None.

Description:

This installs the specified user interrupt handler for the interrupt line described by `line`. The handler function `callback_func` is called with its the single parameter set to `callback_param`. If the chip variant supports it, user can specify a `priority` for the interrupt line. If no prioritization is needed, `gpOS_INTERRUPT_NOPRIORITY` must be specified.

12.10.4 gpOS_interrupt_uninstall

Note: this replaces interrupt_install in previous version

Uninstall an interrupt handler

Synopsis:

```
#include <gpOS.h>

void gpOS_interrupt_uninstall( gpOS_interrupt_line_t line);
```

Arguments:

`gpOS_interrupt_line_t`:	line to interrupt controller

Returns:

None.

Errors:

None

Description:

This uninstalls the single interrupt handler associated with interrupt source `line`.

12.10.5 gpOS_interrupt_enable

Note: this replaces interrupt_enable in previous version

Enables an interrupt line

Synopsis:

```
#include <gpOS.h>

void gpOS_interrupt_enable( gpOS_interrupt_line_t line);
```

Arguments:

`gpOS_interrupt_line_t`:	line to interrupt controller

Returns:

None.

Errors:

None.

Description:

This enables the specified interrupt line described by `line`. The interrupt is served only if an handler function was installed using `gpOS_interrupt_install`.

Note: if no handler was installed and the line is enabled, the ARM could never leave the interrupt exception status.

12.10.6 gpOS_interrupt_disable

Note: this replaces interrupt_disable in previous version

Uninstall an interrupt handler

Synopsis:

```
#include <gpOS.h>

void gpOS_interrupt_disable( gpOS_interrupt_line_t line);
```

Arguments:

`gpOS_interrupt_line_t`:	line to interrupt controller

Returns:

None.

Errors:

None

Description:

This disables the interrupt source `line`. It does not remove the interrupt handler.

12.10.7 gpOS_interrupt_lock

Note: this replace interrupt_lock in previous version

Disable all interrupts

Definition:

```
#include <gpOS.h>

void gpOS_interrupt_lock( void);
```

Arguments:

None

Returns:

None

Errors:

None

Description:

This function disables all interrupts to the CPU.

This function must always be called as a pair with `gpOS_interrupt_unlock`, so that it can be used to create a critical region in which the task cannot be pre-empted by any other task or interrupt. Calls to `gpOS_interrupt_lock` can be nested, and the lock is not released until an equal number of calls to `gpOS_interrupt_unlock` are made.

Note: 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.

12.10.8 gpOS_interrupt_unlock

Note: this replace interrupt_unlock in previous version

Enables all interrupts

Definition:

```
#include <gpOS.h>

void gpOS_interrupt_unlock( void);
```

Arguments:

None

Returns:

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 must always be called as a pair with `gpOS_interrupt_lock`, so that it can be used to create a critical region in which the task cannot be pre-empted by another task or interrupt. As calls to `gpOS_interrupt_lock` can be nested, the lock is not released until at least an equal number of calls to `gpOS_interrupt_unlock` are made.

13 Exceptions

All ARM chip variants supports handling of exceptions. These are generated either externally (for example interrupts) or internally (for example undefined instruction). The processor state before the exception raising is automatically saved by CPU.

When an exception is raised, the CPU changes processor mode and branches to a fixed location. These fixed addresses are called exception vectors.

OS20+ provides APIs to execute a specific portion of code whenever an exception occurs:

```
void gpOS_exception_init
(
   os20_exceptions_t arm_exception,
   os20_exception_vector_t vector
);
```

In this way the user can install a specific routine for each exception handled by the processor. The exceptions are specified through the type `os20_exceptions_t`, defined as:

```
typedef enum os20_exceptions_e
{
  OS20_EXCEPTION_UND,
  OS20_EXCEPTION_SVC,
  OS20_EXCEPTION_PFABT,
  OS20_EXCEPTION_DABT,
  OS20_EXCEPTION_IRQ,
  OS20_EXCEPTION_FIQ
} os20_exceptions_t;
```

The vector executed by OS20+ whenever the configured exception is raised must have the following type:

```
typedef void (*os20_exception_vector_t)(void);
```

Note: *the exception associated to IRQ is automatically configured when `gpOS_kernel_start` is invoked. user should never reconfigure it.*

To remove the routine that handles a specific exception, OS20+ provides the following API:

```
void gpOS_exception_clear( os20_exceptions_t arm_exception);
```

13.1 Exceptions APIs summary

All the definitions related to time can be accessed by including the header file `gpOS.h`, which itself includes the header file `except.h`.

Type	Description
`os20_exceptions_t`	Exception that must be configured
`os20_exception_vector_t`	Type of exception routine

Table 22: Types defined in except.h

Function	Description
`gpOS_exception_init`	Configure an exception vector
`gpOS_exception_clear`	Reset an exception vector

Table 23: Functions defined in except.h

13.2 Exception functions definitions

13.2.1 gpOS_exception_init

Configure an exception vector

Synopsis:

```
#include <gpOS.h>

void gpOS_exception_init
(
   os20_exceptions_t arm_exception,
   os20_exception_vector_t vector
);
```

Arguments:

`os20_exceptions_t`:	Exception to be configured
`os20_exception_vector_t`:	Pointer to routine to be used for exception handling

Returns:

None.

Errors:

None.

Description:

This function configures the exception `arm_exception` to execute the routine `vector` when the exception is raised.

13.2.2 gpOS_exception_clear

Reset an exception vector

Synopsis:

```
#include <gpOS.h>

void gpOS_exception_clear( os20_exceptions_t arm_exception);
```

Arguments:

`os20_exceptions_t`:	Exception to be cleared

Returns:

None.

Errors:

None.

Description:

This function reset the exception `arm_exception` and configures it to do nothing when the exception is raised.

14 Wakelocks

14.1 Wakelocks overview

The wakelocks mechanism collects the MIPS requirements of all tasks and organise the switch from/to high/low frequency accordingly. A task must first register to use this functionality. Once registered, the tasks acquire or release their wakelock according to the processing they need to perform.

Typically, a task registers a wakelock in its “init” function, acquire a wakelock after the OS schedule the task, and release the wakelock before being descheduled by the OS. A task can decide to release its wakelock and continue any processing at low frequency.

All OS functionalities are valid, but timers may be affected by the frequency change. It is advised to consider calling `gpOS_timer_ticks_per_sec()` after the wakelock release to align task timers to the tick adjustment.

14.2 Wakelocks usage exemple

```
#include "gpOS.h"

/* Add a parameter in any of your handling data */
typedef struct
{
  …
  gpOS_wakelockid_t           wakelock_id;
  …
} my_handler _t;
my_handler_t my_handler;

/* Once during the init, register the wakelock */
my_task_init()
{
  …
  gpOS_wakelock_register( &my_handler.wakelock_id );
  …
}

my_task_process()
{
  my_task_init();
  while(1)
  {
    …
    /* Release and Acquire your wakelock according to your task
activity */
    gpOS_wakelock_release(my_handler.wakelock_id, my_timeout);
    message = gpOS_message_receive_timeout(&queue, &time);
    gpOS_wakelock_acquire(my_handler.wakelock_id);
    …
  }
}
```

14.3 Wakelocks handling APIs summary

All the definitions related to wakelocks can be accessed by including the header file `gpOS.h`, which itself includes the header file `gpOS_wakelock.h`.

Types	Description
`gpOS_wakelockid_t`	Wakelock id

Table 24: Types defined in gpOS_wakelock.h

Function	Description
`gpOS_wakelock_init()`	Initialize wakelocks handling
`gpOS_wakelock_register()`	Register a task wakelock
`gpOS_wakelock_acquire()`	Acquire a wakelock
`gpOS_wakelock_release()`	Release a wakelock
`gpOS_wakelock_status()`	Get one wakelock status
`gpOS_wakelock_task_next_activity()`	Get next activity timer
`gpOS_wakelock_get_value_unp()`	Get all wakelocks status (privilege mode)

Table 25: Functions defined in gpOS_wakelock.h

14.4 Wakelocks functions definition

14.4.1 gpOS_wakelock_init

Initialize wakelock handling mechanism in OS20+.

Synopsis:

```
#include "gpOS.h"

gpOS_error_t gpOS_wakelock_init(gpOS_partition_t *partition)
```

Arguments:

`gpOS_partition_t *`:	pointer to a custom partition.

Results:

`gpOS_error_t`:	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the wakelocks memory allocation went wrong.

Description:

This API initializes the wakelock handling mechanism. It must be called during the OS initialization and before any use of the functions below.

14.4.2 gpOS_wakelock_register

Register a wakelock for the caller task.

Synopsis:

```
#include "gpOS.h"

gpOS_error_t gpOS_wakelock_register(gpOS_wakelockid_t *id)
```

Arguments:

None

Results:

`gpOS_wakelockid_t *`:	The wakelock Id is returned, to be kept for further wakelock API calls.
`gpOS_error_t`:	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the wakelocks are not initialized.

Description:

This API must be called during the initialization of a software task willing to benefit from wakelock handling. The id must be kept whithin the task context, and reuse in the acquire and release functions.

14.4.3 gpOS_wakelock_acquire

Acquire a wakelock.

Synopsis:

```
#include "gpOS.h"

gpOS_error_t gpOS_wakelock_acquire (gpOS_wakelockid_t id)
```

Arguments:

`gpOS_wakelockid_t:`	Task wakelock Id.

Results:

`gpOS_error_t:`	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the wakelocks are not initialized or if the id is not correct.

Description:

This API must be called when a task requires the system to remain in high frequency clocking. It is typically called after the reception of a message or a timer expiration. If the wakelocks were all released before, acquiring a wakelock will trigger a transition from low to high frequency clocking.

14.4.4 gpOS_wakelock_release

Release a wakelock.

Synopsis:

```
#include “gpOS.h”

gpOS_error_t gpOS_wakelock_release (
   gpOS_wakelockid_t,
   const gpOS_clock_t *
)
```

Arguments:

`gpOS_wakelockid_t:`	Task wakelock Id.
`const gpOS_clock_t *:`	Time interval in ms or `gpOS_TIMEOUT_INFINITY`.

Results:

`gpOS_error_t:`	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the wakelocks are not initialized or if the id is not correct.

Description:

This API can be called when a task allows the system to switch to low frequency clocking. It is typically called before waiting for an OS event, when the task processing is over. If the wakelocks are all released, the system will perform the transition from high to low frequency.

The time parameter provides the wakelock handling the time of the next task activity in millisecond. It is only considered when the STANDBY mode is allowed and must be filled only by tasks willing to have their timer considered for the next system wake-up.

14.4.5 gpOS_wakelock_status

Get a wakelock status.

Synopsis:

```
#include "gpOS.h"

gpOS_error_t gpOS_wakelock_release (gpOS_wakelockid_t id , boolean_t
*status)
```

Arguments:

`gpOS_wakelockid_t:`	Task wakelock Id.

Results:

`boolean_t *:`	Pointer to a task local variable.
`gpOS_error_t:`	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the wakelocks are not initialized or if the id is not correct.

Description:

This API can be called to know whether the wakelock id has been acquired or released.

14.4.6 gpOS_wakelock_task_next_activity

Get the lowest task timer.

Synopsis:

```
#include "gpOS.h"

void gpOS_wakelock_task_next_activity(gpOS_clock_t** nearest_timer)
```

Arguments:

None

Results:

`gpOS_clock_t**:`	Closest absolute time in s or `gpOS_TIMEOUT_INFINITY`.

Errors:

None

Description:

While the release API fills a list of timers, this API returns the lowest timer of all tasks from the list. It is basically used before entering standby mode to program the RTC alarm.

14.4.7 gpOS_wakelock_get_value_unp

Get all wakelocks value.

Beware: this function is not semaphore protected. It can only be called in privilege mode (SWI).

Synopsis:

```
#include "gpOS.h"

gpOS_error_t gpOS_wakelock_get_value_unp(tU32 *value)
```

Arguments:

None

Results:

`tU32 *`	Wakelocks values (1 bit per Id).
`gpOS_error_t`:	`gpOS_SUCCESS` or `gpOS_FAILURE`

Errors:

Returns `gpOS_FAILURE` if the wakelocks are not initialized.

Description:

All wakelocks set to 0 means no task request the system to be active. While the wakelocks handling handle the low/high frequency requirements, the standby mode is not handled.

15 Disclaimer